CONTENTS

Paragraph Number	Title	Page Number

CONTENTS

Paragraph Number	Title	Page Number

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

Paragraph Number	Title	Page Number

CONTENTS

CONTENTS

Paragraph Number	Title	Page Number

CONTENTS

CONTENTS

Paragraph Number	Title	Page Number

CONTENTS

ILLUSTRATIONS

ILLUSTRATIONS

ILLUSTRATIONS

Figure Number	Title	Page Number

ILLUSTRATIONS

Figure Number	Title	Page Number

TABLES

CONTENTS

TABLES

CONTENTS

About This Book

The primary objective of this manual is to help hardware and software designers who are working with the PowerPC 604™ microprocessor. This book is intended as a companion to the *PowerPC™ Microprocessor Family: The Programming Environments*, 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 604 RISC Microprocessor User's Manual* summarizes features of the 604 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 604-specific registers and progressing to more specialized topics such as 604-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 604. 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, “PowerPC 604 Processor Programming Model,” is useful for software engineers who need to understand the 604-specific registers, operand conventions, and details regarding how PowerPC instructions are implemented on the 604.
- Chapter 3, “Cache and Bus Interface Unit Operation,” provides a discussion of the cache and memory model as implemented on the 604.
- Chapter 4, “Exceptions,” describes the exception model as implemented on the 604.
- Chapter 5, “Memory Management,” provides descriptions of the PowerPC address translation and memory protection mechanism as implemented on the 604.
- Chapter 6, “Instruction Timing,” describes instruction timing in the 604.
- Chapter 7, “Signal Descriptions,” describes individual signals defined for the 604.
- Chapter 8, “System Interface Operation,” describes interface operations on the 604.
- Chapter 9, “Performance Monitor,” describes the operation of the performance monitor diagnostic tool incorporated in the 604.
- Appendix A, “PowerPC Instruction Set Listings,” lists all the PowerPC instructions. Instructions are grouped according to mnemonic, opcode, function, and form.
- Appendix B, “Invalid Instruction Forms,” describes how invalid instructions are treated by the 604.
- This manual also includes a glossary and an index.

In this document, the terms “PowerPC 604 Microprocessor” and “604” are used to denote a microprocessor from the PowerPC architecture family. The PowerPC 604 microprocessors are available from IBM as PPC604 and from Motorola as MPC604.

Suggested Reading

This section lists additional reading that provides background for the information in this manual.

- *PowerPC Microprocessor Family: The Programming Environments*, MPCFPE/AD (Motorola Order Number) and MPRPPCFPE-01 (IBM Order Number)
- *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Second Edition, Morgan Kaufmann Publishers, Inc., San Francisco, CA
- John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, CA
- *PowerPC 601 RISC Microprocessor User’s Manual*, Rev 1
  MPC601UM/AD (Motorola Order Number) and 52G7484/(MPR601UMU-02) (IBM Order Number)
- *PowerPC 601 RISC Microprocessor Technical Summary*, Rev 1
  MPC601/D (Motorola order number) and MPR601TSU-02 (IBM order number)
- *PowerPC 603 RISC Microprocessor User’s Manual*, MPC603UM/AD (Motorola order number) and MPR603UMU-01 (IBM order number)

- *PowerPC 603 RISC Microprocessor Technical Summary*, Rev 3
  MPC603/D (Motorola order number) and MPR603TSU-03 (IBM order number)
- *PowerPC 604 RISC Microprocessor Technical Summary*, Rev 1
  MPC604/D (Motorola order number) and MPR604TSU-02 (IBM order number)
- *PowerPC 620 RISC Microprocessor Technical Summary*, MPC620/D (Motorola order number) and MPR620TSU-01 (IBM order number)

Additional literature on PowerPC implementations is being released as new processors become available.

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 AP0–AP3 (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
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

Table i. Acronyms and Abbreviated Terms (Continued)

Term	Meaning
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*
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

Table i. Acronyms and Abbreviated Terms (Continued)

Term	Meaning
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
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)
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 604™ microprocessor. It includes the following:

- A summary of 604 features
- Details about the 604 hardware implementation. This includes descriptions of the 604's execution units, cache implementation, memory management units (MMUs), and system interface.
- A description of the 604 execution model. This section includes information about the programming model, instruction set, exception model, and instruction timing.

1.1 Overview

This section describes the features of the 604, provides a block diagram showing the major functional units, and describes briefly how those units interact.

The 604 is an implementation of the PowerPC™ family of reduced instruction set computer (RISC) microprocessors. The 604 implements the PowerPC Architecture™ as it is specified for 32-bit addressing, which provides 32-bit effective (logical) addresses, integer data types of 8, 16, and 32 bits, and floating-point data types of 32 and 64 bits (single-precision and double-precision).

The 604 is a superscalar processor capable of issuing four instructions simultaneously. As many as six instructions can finish execution in parallel. The 604 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 that instructions be of uniform length, allows for rapid execution times, yields high efficiency and throughput. The 604'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 604 has separate memory management units (MMUs) and separate 16-Kbyte on-chip caches for instructions and data. The 604 implements two 128-entry, two-way set (64 x 2) associative translation lookaside buffers (TLBs), one for instructions and one for data. The 604 also provides support for demand-paged virtual memory address translation and variable-sized block translation. The TLBs and the cache use least-recently used (LRU) replacement algorithms.

The 604 has a 64-bit external data bus and a 32-bit address bus. The 604 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 604 supports single-beat and burst data transfers for memory accesses and memory-mapped I/O accesses.

The 604 uses an advanced, 3.3-V CMOS process technology and is fully compatible with TTL devices.

1.1.1 PowerPC 604 Microprocessor Features

This section summarizes features of the 604's implementation of the PowerPC architecture.

Figure 1-1 provides a block diagram showing features of the 604. 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.

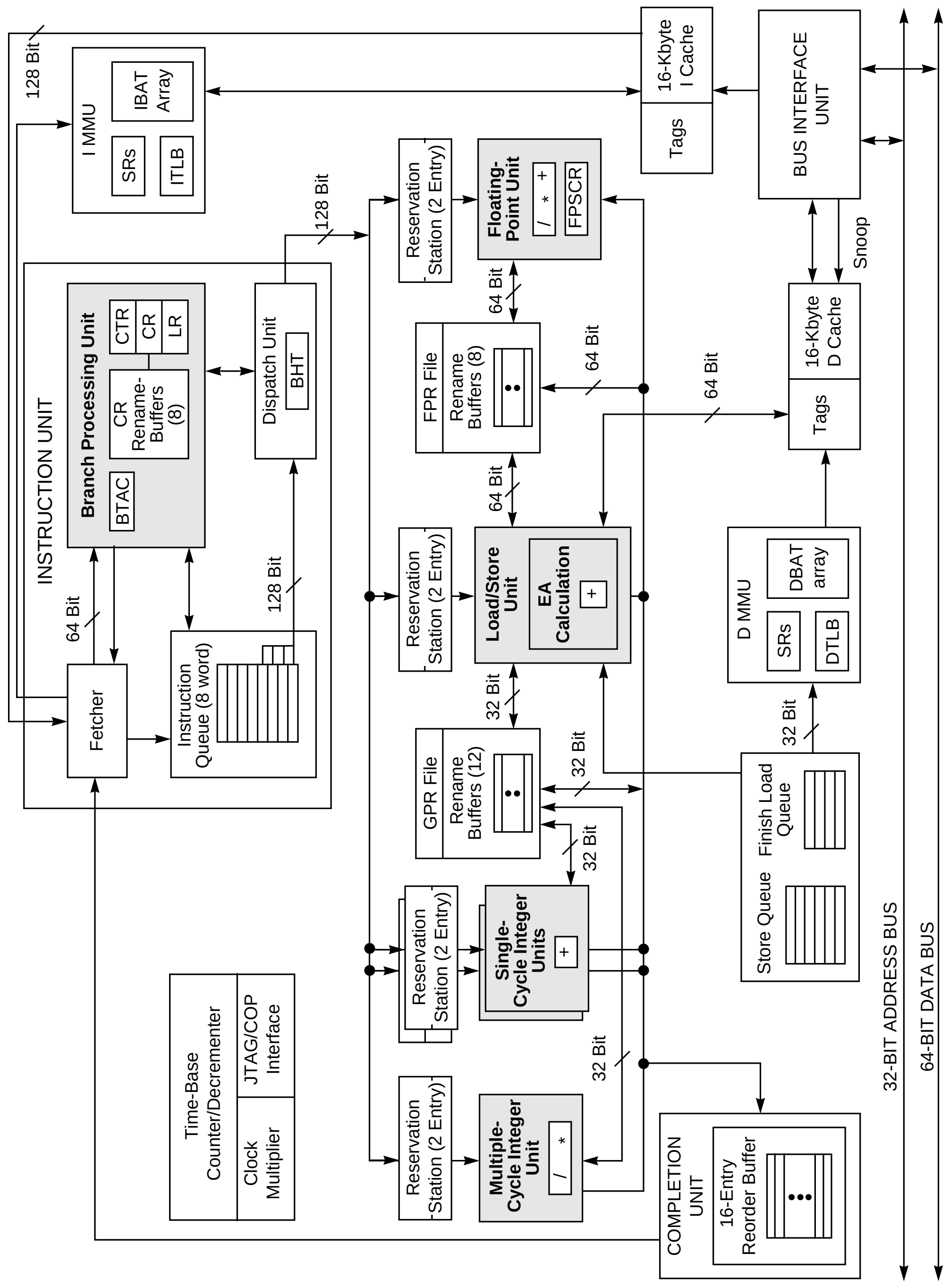


Figure 1-1. Block Diagram

Major features of the 604 are as follows:

- High-performance, superscalar microprocessor
  - As many as four instructions can be issued per clock cycle.
  - 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 two branches
    - 64-entry fully-associative branch target address cache (BTAC)
    - 512-entry, direct-mapped branch history table (BHT) with two bits per entry for 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
    - Twelve 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
  - Load/store unit (LSU)
    - Two-entry reservation station to minimize stalls
    - Single-cycle, pipelined cache access
    - Dedicated adder performs EA calculations
    - Performs alignment and precision conversion for floating-point data
    - Performs alignment and sign extension for integer data

    - Four-entry finish load queue (FLQ) provides load miss buffering
    - Six-entry store queue
    - Supports both big- and little-endian modes
- Rename buffers
  - Twelve GPR rename buffers
  - Eight FPR rename buffers
  - Eight condition register (CR) rename buffers

  The 604 rename buffers are described in Section 1.2.1.5, “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)
  - 16-Kbyte, four-way set-associative instruction and data caches
  - LRU replacement algorithm
  - 32-byte (eight word) cache block size
  - Physically indexed; physical tags. Note that the PowerPC architecture refers to physical address space as real address space.
  - Cache write-back or write-through operation programmable on a per page or per block basis
  - Instruction cache can provide four instructions per clock cycle; data cache can provide two words per clock cycle.
  - Caches can be disabled in software
  - Caches can be locked
  - Parity checking performed on both caches
  - Data cache coherency (MESI) maintained in hardware
  - Secondary data cache support provided
  - Instruction cache coherency maintained in software
  - Provides a no-DRTRY/data streaming mode, which allows consecutive burst read data transfers to occur without intervening dead cycles. This mode also disables data retry operations.
- 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
- Separate instruction and data translation lookaside buffers (TLBs)
- Both TLBs are 128-entry and two-way set associative
- Separate IBATs and DBATs (four each) also defined as SPRs
- LRU replacement algorithm
- Hardware table search (caused by TLB misses) through hashed page tables
- 52-bit virtual address; 32-bit physical address

- Bus interface features include the following:
    - Selectable processor-to-bus clock frequency ratios (1:1, 1.5:1, 2:1, and 3:1)
    - A 64-bit split-transaction external data bus with burst transfers
    - Support for address pipelining and limited out-of-order bus transactions
    - Additional signals and signal redefinition for direct-store operations
- Multiprocessing support features include the following:
    - 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.
    - Separate port into data cache tags for bus snooping
    - Load/store with reservation instruction pair for atomic memory references, semaphores, and other multiprocessor operations
- Power management
    - Operating voltage is 3.3 ± 0.3 V
    - Software-initiated NAP mode suspends instruction dispatch and waits for all activity in progress, including active and pending bus transactions, to complete. It then shuts down the internal chip clocks, and enters nap mode.
- 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 through JTAG boundary-scan capability

1.2 PowerPC 604 Microprocessor Hardware Implementation

This section provides an overview of the 604'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.

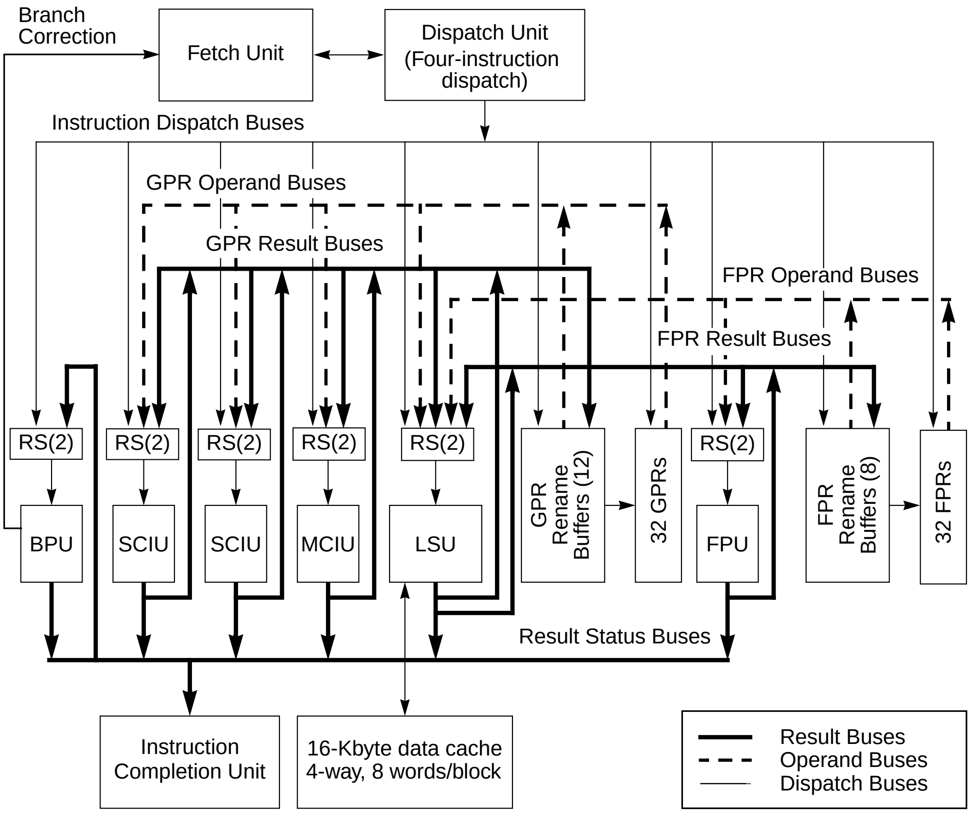


Figure 1-2. Block Diagram—Internal Data Paths

1.2.1 Instruction Flow

Several units on the 604 ensure the proper flow of instructions and operands and guarantee the correct update of the architectural machine state. These units include the following:

- 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 buffer.
- Decode/dispatch unit—The decode/dispatch unit decodes instructions and dispatches them 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 Fetch Unit

The fetch unit provides instructions to the eight-entry 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. The BPU recovers from an incorrect prediction when a branch instruction is in the decode stage. Subsequent instructions are flushed and fetching begins at the correct target address.
5. A fetch address is found in the BTAC. 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.
6. If none of the previous conditions exist, the instruction is fetched from the next sequential address.

1.2.1.2 Decode/Dispatch Unit

The decode/dispatch unit provides the logic for decoding instructions and issuing them to the appropriate execution unit. The eight-entry instruction queue consists of two four-entry queues—a decode queue (DEQ) and a dispatch queue (DISQ).

The decode logic decodes the four instructions in the decode queue. 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 decodes the instructions in the DISQ for possible dispatch. The dispatch logic resolves unconditional branch instructions and predicts conditional branch instructions using the branch decode logic, the BHT, and values in the CTR.

The 512-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. 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 next branch is not taken, the BHT then predicts taken.

The dispatch logic also allocates each instruction to the appropriate execution unit. A reorder buffer (ROB) entry is allocated for each instruction, and dependency checking is done 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.

1.2.1.3 Branch Processing Unit (BPU)

The BPU is used for branch instructions and condition register logical operations. All branches, including unconditional branches, are placed in a reservation station 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.

1.2.1.4 Completion Unit

The completion unit retires executed instructions from the reorder buffer (ROB) 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 decode/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 604 also updates any other resource affected by this instruction. Several instructions can complete simultaneously. Most exception conditions are recognized at completion time.

1.2.1.5 Rename Buffers

To avoid contention for a given register location, the 604 provides rename registers for storing instruction results before the completion unit commits them to the architected register. Twelve rename registers are provided for the GPRs, eight for the FPRs, and eight for the condition register. GPRs are described in Section 1.3.2.1, “General-Purpose Registers (GPRs),” FPRs are described in Section 1.3.2.2, “Floating-Point Registers (FPRs),” and the condition register is described in Section 1.3.2.3, “Condition Register (CR).”

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 604’s arithmetic execution units—the two single-cycle IUs, the multiple cycle IU, and the 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. Each IU has a dedicated result bus that connects to rename buffers and to all reservation stations. Each IU has a two-entry reservation station to reduce stalls. The reservation station can receive instructions from the decode/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 32-bit integer multiplier/divider and supports early exit on 16- x 32-bit multiplication operations. 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.

Note that the load and store instructions that update their address base register (specified by the **r**A operand) pass the update results on the MCIU's result bus. Otherwise, the MCIU's result bus is dedicated to MCIU operations.

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 decode/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. Instructions are executed from each reservation station in dispatch order.

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. As noted above, load and store instructions that update the base address register pass their results on the MCIU's result bus. This is the only exception to the dedicated use of result buses.

The LSU includes a 32-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 operations to precede pending store operations and resolves any dependencies incurred when a pending store is to the same address as the load. If such a dependency exists, the LSU delays the load operation until the correct data can be forwarded. If only the low-order 12 bits of the EAs match, both addresses may be aliases for the same physical address, in which case, the load operation is delayed until the store has been written back to the cache, ensuring that the load operation retrieves the correct data.

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 604’s MMUs support up to 4 Petabytes (2^{52}) of virtual memory and 4 Gigabytes (2^{32}) 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 604 implements two 128-entry, two-way set associative translation lookaside buffers (TLBs), one for instructions and one for data. These TLBs can be accessed simultaneously.

1.2.4 Cache Implementation

The 604 implements separate 16-Kbyte, four-way set-associative data and instruction caches (Harvard architecture). The PowerPC architecture defines the unit of coherency as a cache block, which for the 604 is a 32-byte (eight-word) line.

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

- Write-back/write-through mode
- Cache-inhibited mode
- Memory coherency
- Guarded memory (prevents access for speculative execution)

The caches implement an LRU replacement algorithm.

1.2.4.1 Instruction Cache

The 604's 16-Kbyte, four-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/enabled and invalidated by setting the HID0[16] and HID0[20] bits, respectively. The instruction cache can be locked by setting HID0[18].

1.2.4.2 Data Cache

The 604's data cache is a 16-Kbyte, four-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, the LSU is blocked internally for one cycle to allow the eight-word block of data to be copied to the write-back buffer.

To ensure cache coherency, the 604 data cache supports the four-state MESI (modified/exclusive/shared/invalid) protocol.

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/enabled and invalidated by setting the HID0[17] and HID0[21] bits, respectively. The data cache can be locked by setting HID0[19].

Each cache line contains eight contiguous words from memory that are loaded from an eight-word boundary (that is, bits A27–A31 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.

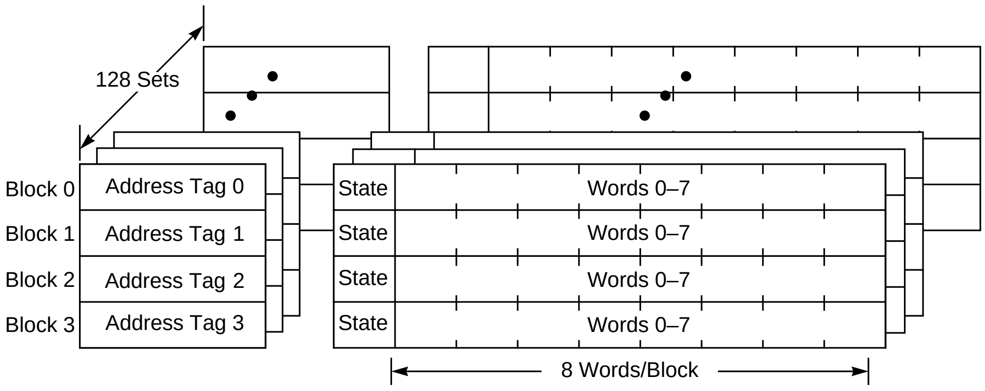


Figure 1-3. Cache Unit Organization

1.2.5 System Interface/Bus Interface Unit (BIU)

The 604 provides a versatile bus interface that allows a wide variety of system design options. The interface includes a 72-bit data bus (64 bits of data and 8 bits of parity), a 36-bit address bus (32 bits of address and 4 bits of parity), and sufficient control signals to allow for a variety of system-level optimizations. The 604 uses one-beat and four-beat data transactions, although it is possible for other bus participants to perform longer data transfers. The 604 clocking structure supports processor-to-bus clock ratios of 1:1, 1.5:1, 2:1, and 3:1, as described in Section 1.2.6, “Clocking.”

The system interface is specific for each PowerPC processor implementation. The 604 system interface is shown in Figure 1-4.

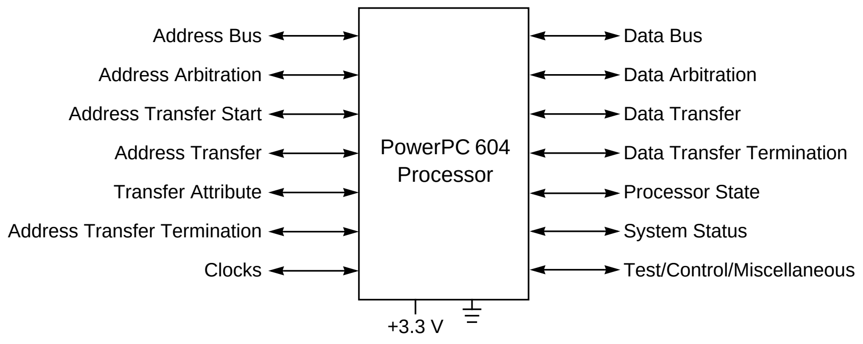


Figure 1-4. System Interface

Four-beat burst-read memory operations that load an eight-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, variants of the burst and single-beat operations (global memory operations that are snooped and atomic memory operations, for example), and address retry activity (for example, when a snooped read access hits a modified line in the data cache).

The BIU implements the critical double-word first access where the double-word requested by the fetcher or the load/store unit is fetched first and the remaining words in the line are fetched later. The critical double-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.

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. The 604 supports bus pipelining and out-of-order split-bus transactions. In general, the bus-pipelining mechanism allows as many as three address tenures to be outstanding before a data tenure is initiated. Address tenures for address-only transactions can exceed this limit.

Typically, memory accesses are weakly-ordered. Sequences of operations, including load/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 604 allows load operations to precede store operations (except when a dependency exists, of course). In addition, the 604 provides a separate queue for snoop push operations so these operations can access the bus ahead of previously queued operations. The 604 dynamically optimizes run-time ordering of load/store traffic to improve overall performance.

In addition, the 604 implements a data bus write only signal ($\overline{\text{DBWO}}$) that can be used for reordering write operations. Asserting $\overline{\text{DBWO}}$ causes the first write operation to occur before any read operations on a given processor. Although this may be used with any write operations, it can also be used to reorder a snoop push operation.

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 604 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 604 bus support for memory and direct-store operations. Note that some signals perform different functions depending upon the addressing protocol used.

1.2.5.1 Memory Accesses

Memory accesses allow transfer sizes of 8, 16, 24, 32, 40, 48, 56, or 64 bits in one bus clock cycle. Data transfers occur in either single-beat transactions or four-beat burst transactions. A single-beat transaction transfers as much as 64 bits. Single-beat transactions are caused by noncached accesses that access memory directly (that is, reads and writes when caching is disabled, cache-inhibited accesses, and stores in write-through mode). Burst transactions, which always transfer an entire cache block (32 bytes), are initiated when a block in the cache is read from or written to memory. Additionally, the 604 supports address-only transactions used to invalidate entries in other processors' TLBs and caches.

Typically I/O accesses are performed using the same protocol as memory accesses. Refer to Chapter 8, "System Interface Operation," for more information.

1.2.5.2 Signals

The 604's signals are grouped as follows:

- Address arbitration signals—The 604 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 cache-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 604 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- and hard-reset signals. These signals are used to interrupt and, under various conditions, to reset the processor.
- Processor state signals—These two signals are used to set the reservation coherency bit and set the size of the 604's output buffers.
- Miscellaneous signals—These signals are used in conjunction with such resources as secondary caches and the time base facility.
- Test/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).
- 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, $\overline{\text{ARTRY}}$ (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 AP0–AP3 (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.

1.2.5.3 Signal Configuration

Figure 1-5 illustrates the logical pin configuration of the 604, showing how the signals are grouped.

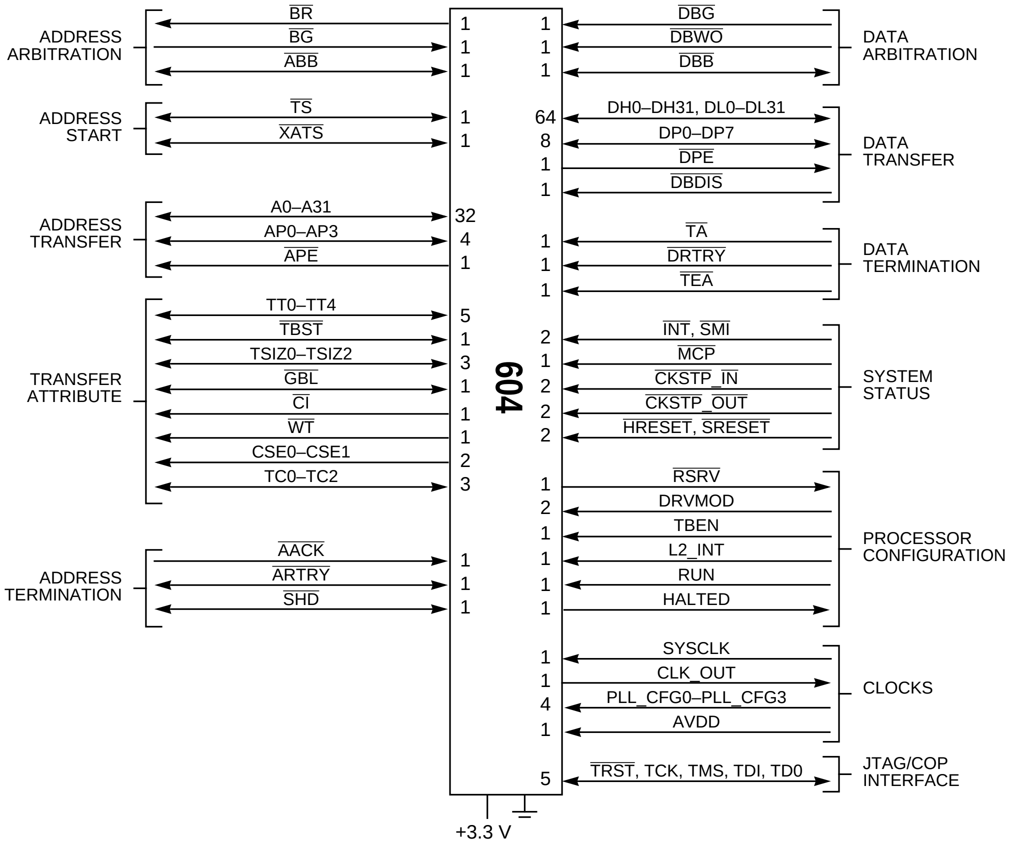


Figure 1-5. PowerPC 604 Microprocessor Signal Groups

1.2.6 Clocking

The 604 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 604 supports the following processor-to-bus clock frequency ratios—1:1, 1.5:1, 2:1, and 3:1, although not all ratios are available for all frequencies. For more information, refer to the 604 hardware specifications.

1.3 PowerPC 604 Microprocessor Execution Model

This section describes the following characteristics of the 604's execution model:

- The PowerPC architecture
- The 604 register set and programming model
- The 604 instruction set
- The 604 exception model
- Instruction timing on the 604

1.3.1 Levels of the PowerPC Architecture

The PowerPC architecture is derived from the IBM POWER Architecture™ (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, 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 604 complies with all three levels of the PowerPC architecture. Note that the PowerPC architecture defines additional instructions for 64-bit data types. These instructions cause an illegal instruction exception on the 604. PowerPC processors are allowed to have implementation-specific features that fall outside, but do not conflict with, the PowerPC architecture specification. Examples of features that are specific to the 604 include the performance monitor and nap mode.

The 604 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 604 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.

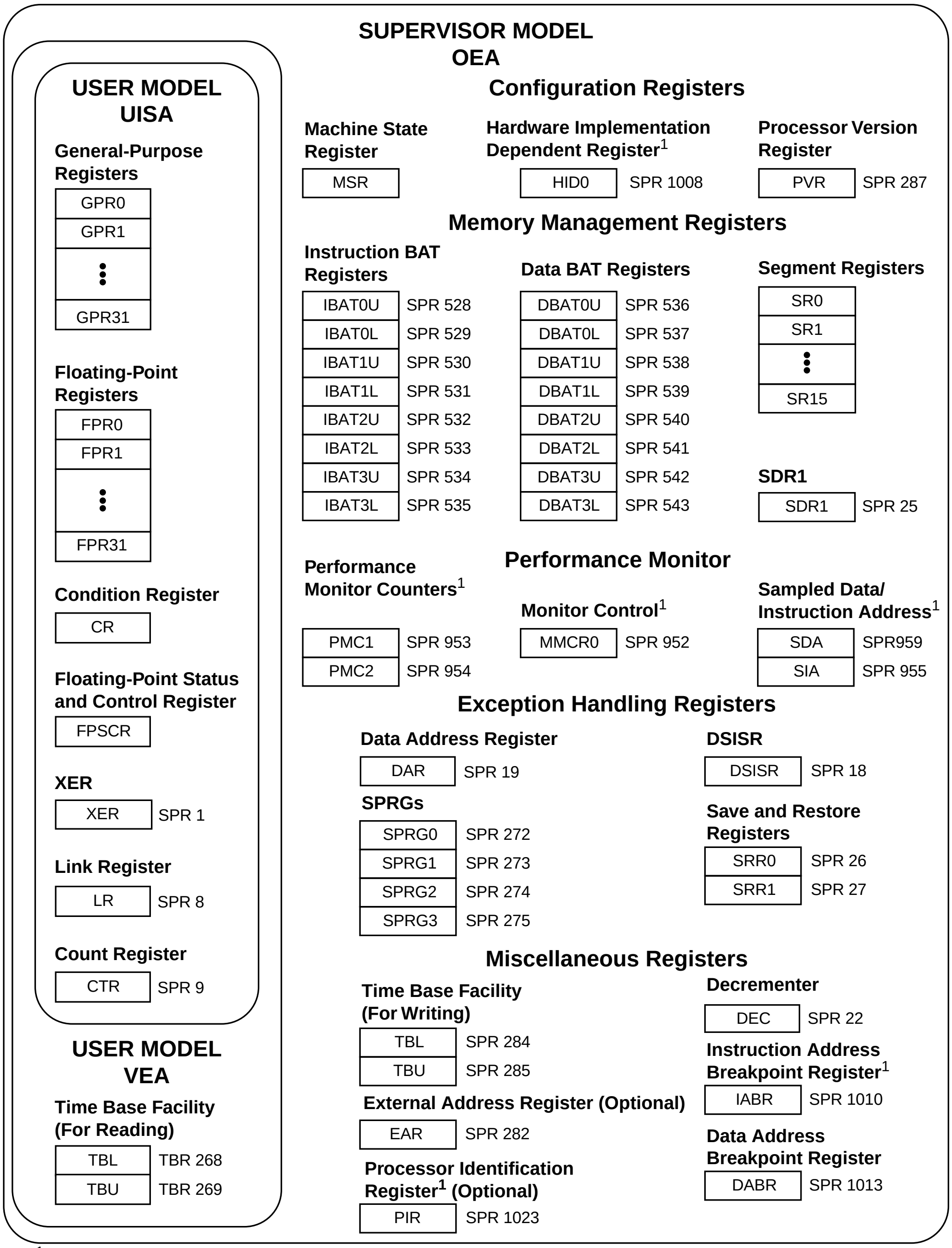
1.3.2 Registers and Programming Model

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 604, indicating those that are defined by the PowerPC architecture and those that are 604-specific. Note that all of these registers except the FPRs are 32 bits wide.



[1] 604-specific—not defined by the PowerPC architecture

Figure 1-6. Programming Model—PowerPC 604 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 following sections summarize the PowerPC registers that are implemented in the 604.

1.3.2.1 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. The 604 also has 12 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.

1.3.2.2 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 604 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. The FPRs and their associated rename buffers can contain data objects of either single- or double-precision floating-point formats.

1.3.2.3 Condition Register (CR)

The CR is a 32-bit user-level register that consists of eight four-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 604 also has eight CR rename buffers, which provide a way to buffer data intended for the CR. The rename buffers are not defined by the PowerPC architecture.

1.3.2.4 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.

1.3.2.5 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 604 implements the MSR as a 32-bit register; 64-bit PowerPC processors use a 64-bit MSR that provides a superset of the 32-bit functionality.

1.3.2.6 Segment Registers (SRs)

For memory management, 32-bit PowerPC implementations use sixteen 32-bit segment registers (SRs).

1.3.2.7 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**.

In the 604, all SPRs are 32 bits wide.

1.3.2.7.1 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 32 bits wide.
- Count register (CTR)—The CTR is decremented and tested automatically as a result of branch and count instructions. The CTR is 32 bits wide.
- XER—The 32-bit XER contains the integer carry and overflow bits.
- The time base registers (TBL and TBU) can be read by user-level software, but can be written to only by supervisor-level software.

1.3.2.7.2 Supervisor-Level SPRs

The 604 also contains SPRs that can be accessed only by supervisor-level software. These registers consist of the following:

- The 32-bit data DSISR defines the cause of DSI and alignment exceptions.
- The data address register (DAR) is a 32-bit register that holds the address of an access after an alignment or DSI exception.

- The decrementer register (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay. In the 604, the decrementer frequency is 1/4th of the bus clock frequency (as is the time base frequency).
- The 32-bit SDR1 register specifies the location and page table format used in logical-to-physical address translation for pages.
- The machine status save/restore register 0 (SRR0) is a 32-bit register that is used by the 604 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.
- The machine status save/restore register 1 (SRR1) is a 32-bit register used to save machine status on exceptions and to restore machine status when an **rfi** instruction is executed.
- SPRG0–SPRG3 registers are 32-bit registers provided for operating system use.
- The external access register (EAR) is 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.
- The processor version register (PVR) is a 32-bit, read-only register that identifies the version (model) and revision level of the PowerPC processor.
- The time base registers (TBL and TBU) 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 604, the time base frequency is 1/4th of the bus clock frequency (as is the decrementer frequency). Counting is enabled by the Time Base Enable signal (TBEN).
- 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 604 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 604, 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 604. 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 (MMCR0)—This is 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 HID registers, or other implementation-specific registers, be consistent among PowerPC processors.

1.3.3 Instruction Set and Addressing Modes

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

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 604 implements the entire PowerPC instruction set (for 32-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 604 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. For almost all floating-point instructions, 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 PowerPC architecture defines instructions 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 Instruction Synchronize instruction (**isync**)—This instruction causes the 604 to purge its instruction buffers and fetch the double word containing the next sequential instruction.
  - 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.
- Processor control instructions—These instructions are used for synchronizing memory accesses and managing caches, TLBs, and segment registers. These instructions include move to/from special-purpose register instructions (**mtspr** and **mfspr**).

- Memory/cache control instructions—These instructions provide control of caches, TLBs, and segment registers.
  - User- and supervisor-level cache instructions
  - Segment register manipulation instructions
  - Translation lookaside buffer management instructions
- Optional instructions—the 604 implements the following optional instructions:
  - The **eciwx**/**ecowx** instruction pair
  - The TLB Synchronize instruction (**tlbsync**)
  - Optional graphics instructions:
    - Store Floating-Point as Integer Word Indexed (**stfiwx**)
    - Floating Reciprocal Estimate Single (**fres**)
    - Floating Reciprocal Square Root Estimate (**frsqrte**)
    - 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, and 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, and 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 32-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 32-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 604 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 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 the following 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 604 implements only the imprecise nonrecoverable mode. The imprecise, recoverable mode is treated as the precise mode in the 604.
- 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 604, and in many PowerPC processors, the hardware interrupt is generated by the assertion of the Interrupt ($\overline{\text{INT}}$) signal, which is not defined by the architecture. In addition, the 604 implements 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, and the performance monitor interrupt.

    When these exceptions occur, their handling is postponed until all instructions, and any exceptions associated with those instructions, complete execution. These exceptions are maskable by setting MSR[EE].
  - — 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 machine state register (MSR)—FE0 and FE1—that determine how floating-point exceptions are handled. There are four combinations of bit settings, of which the 604 implements three. These are as follows:

- Ignore exceptions mode (FE0 = FE1 = 0). 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 (FE0 = 1; FE1 = x). This mode includes both the precise mode and imprecise recoverable mode 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 604 takes floating-point exceptions as defined by the PowerPC architecture.

- Imprecise nonrecoverable mode (FE0 = 0; FE1 = 1). In this mode, when a floating-point instruction causes a floating point exception, the save restore register 0 (SRR0) may point to an instruction following the instruction that caused the exception.

The 604 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 (imprecise nonrecoverable mode)

The 604'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 or hard reset signal.
Machine check	00200	A machine check exception is signaled by the assertion of a qualified $\overline{\text{TEA}}$ indication on the 604 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 or in the instruction or data caches. The assertion of the $\overline{\text{TEA}}$ signal is determined by load and store operations initiated by the processor; however, it is expected that the $\overline{\text{TEA}}$ 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.

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

Exception Type	Vector Offset (hex)	Causing Conditions
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 604 stops dispatching instructions and waits for all dispatched instructions to complete. Any exceptions associated with dispatched instructions are taken before the interrupt is taken.
Alignment	00600	An alignment exception is caused when the processor cannot perform a memory access for the following reasons: 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 cache-inhibited or write-through. A **dcbz** instruction has executed when the 604 data cache is locked or disabled. An access is not naturally aligned in little-endian mode. An **lmw**, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx** instruction is issued in little-endian mode.

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

Exception Type	Vector Offset (hex)	Causing Conditions
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 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 not required in the 604.
Reserved	00E10–00EFF	—
Performance monitoring interrupt	00F00	The performance monitoring interrupt is a 604-specific exception and is used with the 604 performance monitor, described in Section 1.5, "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	—

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

Exception Type	Vector Offset (hex)	Causing Conditions
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. This exception is provided for use with the nap mode, which is described in Section 1.4, “Power Management—Nap Mode.”
Reserved	01500–02FFF	Reserved, implementation-specific exceptions. These are not implemented in the 604.

1.3.5 Instruction Timing

As shown in Figure 1-7, the common pipeline of the 604 has six 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.

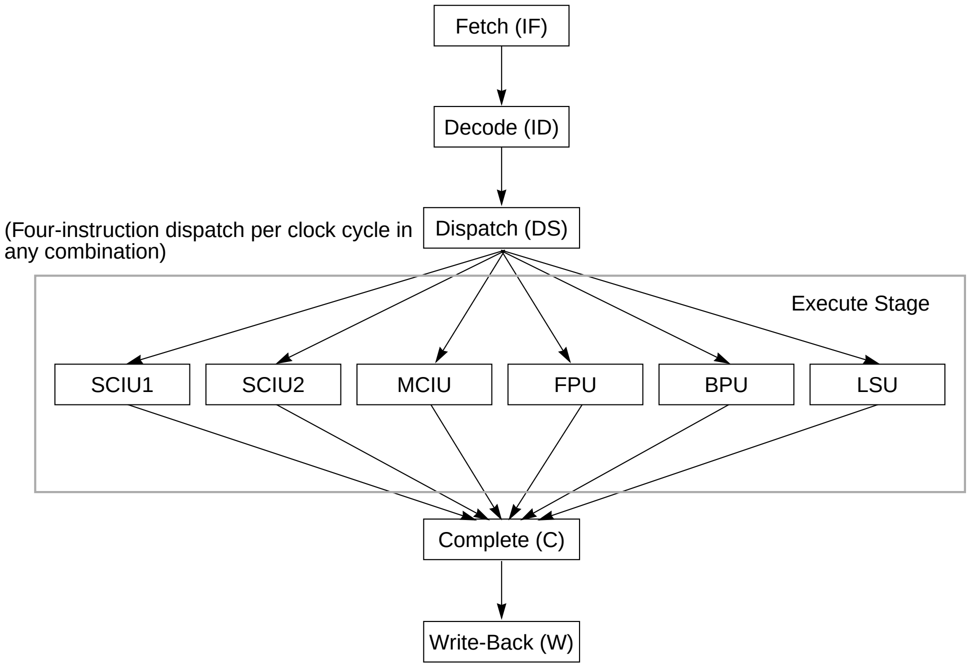


Figure 1-7. Pipeline Diagram

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 decode (ID)—During the ID stage, all time-critical decoding is performed on instructions in the dispatch queue (DISQ). The remaining decode operations are performed during the instruction dispatch stage.
- Instruction dispatch (DS)—During the dispatch stage, the decoding that is not time-critical is performed on the instructions provided by the previous ID 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 604 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 (ROB). 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.

  The CR, CTR, and LR are also updated during the complete stage.
- Writeback (W)—The writeback stage is used to write back any information from the rename buffers that was not written back during the complete stage.

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

The floating-point pipeline has three stages. Floating-point divide operations iterate in the first stage.

1.4 Power Management—Nap Mode

The 604 provides a power-saving mode, called nap mode, in which all internal processing and bus operations are suspended. Software initiates nap mode by setting the MSR[POW] bit. After this bit is set, the 604 suspends instruction dispatch and waits for all activity in progress, including active and pending bus transactions, to complete. It then powers down the internal clocks, and indicates nap mode by asserting the HALTED output signal.

When the 604 is in nap mode, all internal activity stops except for decrementer, time base, and interrupt logic, and the 604 does not snoop bus activity unless the system asserts the RUN input signal. Asserting the RUN signal causes the HALTED signal to be negated.

Nap mode is exited (clocks resume and MSR[POW] cleared) when any asynchronous interrupt is detected.

1.5 Performance Monitor

The 604 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 register 0 (MMCR0) 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 counter registers (PMC1 or PMC2) 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
PowerPC 604 Processor Programming Model

This chapter describes the PowerPC programming model with respect to the 604. It consists of three major sections, which describe the following:

- Registers implemented in the 604
- Operand conventions
- The 604 instruction set

2.1 The PowerPC 604 Processor Register Set

This section describes the registers in the 604 and includes an overview of the registers defined by the PowerPC architecture and a more detailed description of 604-specific registers and differences in how the registers defined by the PowerPC architecture are implemented in the 604. 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*.

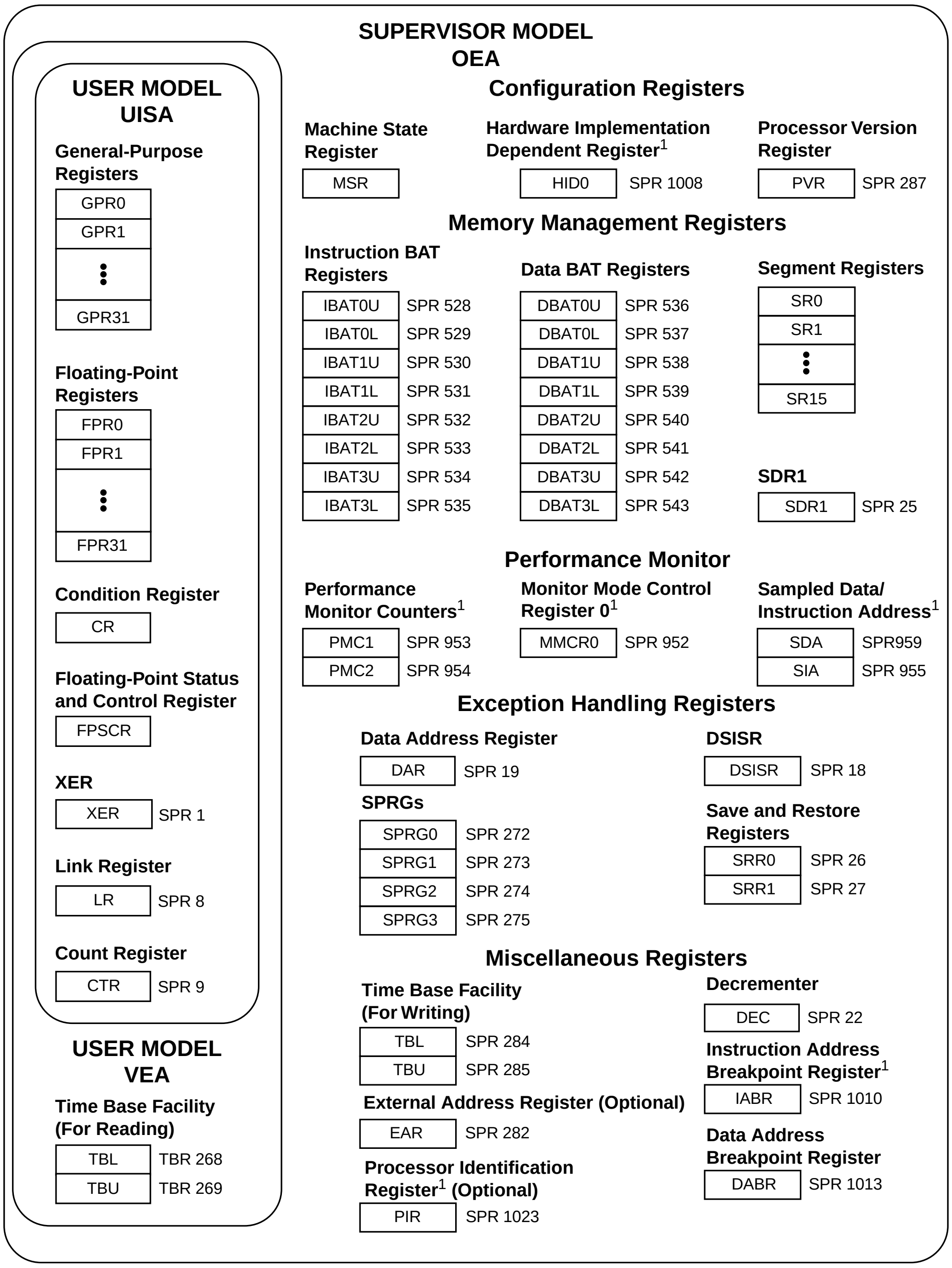
Note that registers are defined at all three levels of the PowerPC architecture—user instruction set architecture (UISA), virtual environment architecture (VEA), and operating environment architecture (OEA). The PowerPC architecture defines register-to-register operations for all computational instructions. 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 integer exception register (XER) is SPR 1). These registers can be accessed using the **mtspr** and **mfspr** instructions.

Implementation Note—The 604 fully decodes the SPR field of the instruction. If the SPR specified is undefined, the illegal instruction program exception occurs.



[1] 604-specific—not defined by the PowerPC architecture

Figure 2-1. Programming Model—PowerPC 604 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 604 is the same in both cases. In the 604, 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 6.6, "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 604 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.

  - Integer exception register (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 consists of 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. 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 32-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**), System Call (**sc**), and Return from Exception (**rfi**) 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 604 defines MSR[29] as the performance monitor marked mode bit (PM). This additional bit is described in Table 2-1.

Table 2-1. MSR[PM] Bit

Bit	Name	Description
29	PM	Performance monitor marked mode 0 Process is not a marked process. 1 Process is a marked process. This bit is specific to the 604, and is defined as reserved by the PowerPC architecture. For more information about the performance monitor, see Chapter 9, "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 4 for the 604. 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."
  – 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** 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** 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 consists of 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 604, the decrementer register is decremented at a speed that is one-fourth the speed of the bus clock.
  - 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 604 are described as follows. Note that in the 604, 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 604, 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 604. 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.
  - 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 (MMCR0)—This is 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.

2.1.2 604-Specific Registers

This section describes registers that are defined for the 604 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 these are all supervisor-level registers.

2.1.2.1 Instruction Address Breakpoint Register (IABR)

The 604 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.

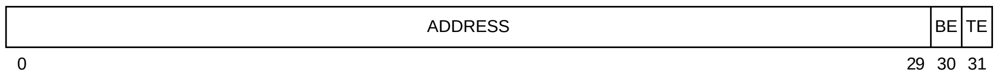


Figure 2-2. Instruction Address Breakpoint Register

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
0–29	Word address to be compared
30	Breakpoint enabled. Setting this bit indicates that breakpoint checking is to be done.
31	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 4.5.14, "Instruction Address Breakpoint Exception (0x01300)."

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

2.1.2.2 Processor Identification Register (PIR)

The processor identification register (PIR) 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.

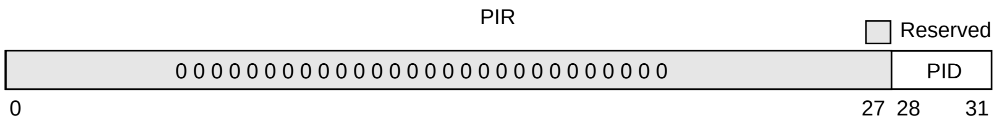


Figure 2-3. Processor Identification Register

The PIR can be accessed with the **mtspr** and **mfspr** instructions using the SPR number, 1013. 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

The hardware implementation dependent register 0 (HID0) is an SPR that controls the state of several functions within the 604.

Table 2-3. Hardware Implementation-Dependent Register 0 Bit Settings

Bit	Description
0	Enable machine check input pin 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 a 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.
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.
7	Disable snoop response high state restore HID bit 7, if active, alters bus protocol slightly by preventing the processor from driving the $\overline{\text{SHD}}$ and $\overline{\text{ARTRY}}$ signals to the high (negated) state. If this is done, then the system must restore the signals to the high state.
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, cache ops) 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.

Table 2-3. Hardware Implementation-Dependent Register 0 Bit Settings (Continued)

Bit	Description
18	Instruction cache lock 0 Normal operation 1 All misses are treated as cache-inhibited. Hits occur as normal. Snoop and cache operations continue to work as normal. This is the only method for “deallocating” an entry.
19	Data cache lock 0 Normal operation 1 All misses are treated as cache-inhibited. Hits occur as normal. Snoop and cache operations continue to work as normal. This is the only method for “deallocating” an entry. The **dcbz** instruction takes an alignment exception if the data cache is locked when it is executed, provided the target address had been translated correctly.
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 604 executes one instruction at a time. The 604 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.
29	Branch history table enable 0 The 604 uses static branch prediction as defined by the PowerPC architecture (UISA) for those branch instructions that the BHT would have otherwise been used to predict (that is, those that use the CR as the only mechanism to determine direction. For more information on static branch prediction, see section “Conditional Branch Control,” in Chapter 4 of *The Programming Environments Manual*. 1 Allows the use of the 512-entry branch history table (BHT). The BHT is initialized and disabled at power-on reset. The BHT is updated while it is disabled, so it can be initialized before it is enabled.

2.1.2.4 Performance Monitor Registers

The remaining five registers defined for use with the 604 are used by the performance monitor. For more information about the performance monitor, see Chapter 9, “Performance Monitor.”

2.1.2.4.1 Monitor Mode Control Register 0 (MMCR0)

The monitor mode control register 0 (MMCR0) is a 32-bit SPR (SPR 952) 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 2-4.

Table 2-4. MMCR0 Bit Settings

Bit	Name	Description
0	DIS	Disable counting unconditionally 0 The values of the PMCn counters can be changed by hardware. 1 The values of the PMCn counters cannot be changed by hardware.
1	DP	Disable counting while in supervisor mode 0 The PMCn 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	DU	Disable counting while in user mode 0 The PMCn 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	DMS	Disable counting while MSR[PM] is set 0 The PMCn counters can be changed by hardware. 1 If MSR[PM] is set, the PMCn counters are not changed by hardware.
4	DMR	Disable counting while MSR(PM) is zero. 0 The PMCn counters can be changed by hardware. 1 If MSR[PM] is cleared, the PMCn counters are not changed by hardware.
5	ENINT	Enable performance monitoring interrupt signaling. 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	DISCOUNT	Disable counting of PMC1 and PMC2 when a performance monitor interrupt is signaled (that is, ((PMCnINTCONTROL = 1) & (PMCn[0] = 1) & (ENINT = 1)) or the occurrence of an enabled time base transition with ((INTONBITTRANS =1) & (ENINT = 1)). 0 The signalling of a performance monitoring interrupt has no effect on the counting status of PMC1 and PMC2. 1 The signalling of a performance monitoring interrupt prevents the changing of the PMC1 counter. The PMC2 counter will not change if PMC2COUNTCTL = 0. Because a time base signal could have occurred along with an enabled counter negative condition, software should always reset INTONBITTRANS to zero, if the value in INTONBITTRANS was a one.
7–8	RTCSELECT	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

Table 2-4. MMCR0 Bit Settings (Continued)

Bit	Name	Description
9	INTONBITTRANS	Cause interrupt signalling on bit transition (identified in RTCSELECT) 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 INTONBITTRANS.
10–15	THRESHOLD	Threshold value. All 6 bits are supported by the 604; allowing threshold values from 0 to 63. The intent of the THRESHOLD support is to be able to characterize L1 data cache misses.
16	PMC1INTCONTROL	Enable interrupt signaling due to PMC1 counter negative. 0 Disable PMC1 interrupt signaling due to PMC1 counter negative 1 Enable PMC1 Interrupt signaling due to PMC1 counter negative
17	PMC2INTCONTROL	Enable interrupt signalling due to PMC2 counter negative. This signal overrides the setting of DISCOUNT. 0 Disable PMC2 interrupt signaling due to PMC2 counter negative 1 Enable PMC2 Interrupt signaling due to PMC2 counter negative
18	PMC2COUNTCTL	May be used to trigger counting of PMC2 after PMC1 has become negative or after a performance monitoring interrupt is signaled. 0 Enable PMC2 counting 1 Disable PMC2 counting until PMC1 bit 0 is set or until a performance monitor interrupt is signaled This signal can be used to trigger counting of PMC2 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	PMC1SELECT	PMC1 input selector, 128 events selectable; 25 defined. See Table 2-5.
26–31	PMC2SELECT	PMC2 input selector, 64 events selectable; 21 defined. See Table 2-6.

2.1.2.4.2 Performance Monitor Counter Registers (PMC1 and PMC2)

PMC1 and PMC2 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*[INTCONTROL] and MMCR0[ENINT] 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[DISCOUNT] forces the counters stop counting when a counter interrupt occurs.

PMC1 and PMC2 are SPRs 953 and 954, 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*[INTCONTROL] and MMCR0[ENINT] 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 2-5 lists the selectable events with their appropriate MMCR0 encodings.

Table 2-5. Selectable Events—PMC1

MMCR0[19–25] Encoding	Description
000 0000	Nothing
000 0001	Processor cycles
000 0010	Number of instructions completed
000 0011	RTCSELECT bit transition
000 0100	Number of instructions dispatched
000 0101	Icache misses
000 0110	**dtlb** misses
000 0111	Branch predicted incorrectly
000 1000	Number of reservations requested (LARX is ready for execution)
000 1001	Number of load dcache misses that exceeded the threshold value with lateral L2 intervention
000 1010	Number of store dcache misses that exceeded the threshold value with lateral L2 intervention
000 1011	Number of **mtspr** instructions dispatched
000 1100	Number of **sync** instructions
000 1101	Number of **eieio** instructions
000 1110	Number of integer instructions being completed every cycle (no loads or stores)
000 1111	Number of floating-point instructions being completed every cycle (no loads or stores)
001 0000	LSU produced result
001 0001	SCIU1 produced result
001 0010	FPU produced result
001 0011	Instructions dispatched to the LSU
001 0100	Instructions dispatched to the SCIU1
001 0101	Instructions dispatched to the FP unit
001 0110	Snoop requests received
001 0111	Number of load dcache misses that exceeded the threshold value without lateral L2 intervention
001 1000	Number of store dcache misses that exceeded the threshold value without lateral L2 intervention

Bits MMCR0[26–31] are used for selecting events associated with PMC2. These settings are shown in Table 2-6.

Table 2-6. Selectable Events—PMC2

MMCR0[26–31] Select Encoding	Description
00 0000	Nothing
00 0001	Processor cycles
00 0010	Number of instructions completed
00 0011	RTCSELECT bit transition
00 0100	Number of instructions dispatched
00 0101	Number of cycles a load miss takes
00 0110	Data cache misses
00 0111	Instruction tlb misses
00 1000	Branches completed
00 1001	Number of reservations successfully obtained (STCX succeeded)
00 1010	Number of **mfspr** instructions dispatched
00 1011	Number of **icbi** instructions
00 1100	Number of **isync** instructions
00 1101	Branch unit produced result
00 1110	SCIU0 produced result
00 1111	MCIU produced result
01 0000	Instructions dispatched to the branch unit
01 0001	Instructions dispatched to the SCIU0
01 0010	Number of loads completed
01 0011	Instructions dispatched to the MCIU
01 0100	Number of snoop hit occurred

2.1.2.4.3 Sampled Instruction Address Register (SIA)

The two address registers contain the addresses of the data or the instruction that caused a threshold-related performance monitor interrupt. For more information on threshold-related interrupts, see Section 9.1.2.2, “Threshold Events.”

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.

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 SDA contains an effective address that is not guaranteed to match the instruction in the SIA. The SIA and SDA are supervisor-level SPRs.

The SIA can be read by using the **mfspr** instruction and written to by using the **mtspr** instruction (SPR 955).

2.1.2.4.4 Sampled Data Address Register (SDA)

The SDA contains the effective address of an operand of an instruction executing at or around the time that the processor signals the performance monitor interrupt condition. In this case the SDA is not meant to have any connection with the value in the SIA. 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 SIA and SDA are supervisor-level SPRs.

The SDA can be read by using the **mfspr** instruction and written to by using the **mtspr** instruction (SPR 959).

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—UISA

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 2-7. (Although not permitted as memory operands, quad words are shown because quad-word alignment is desirable for certain memory operands).

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 604 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 604 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 2-7 summarizes the conditions and mode behavior for operands.

Table 2-7. 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]
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 2-8 summarizes the mode behavior for results.

Table 2-8. 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 604	Not supported by 604

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 chapter describes instructions and addressing modes defined for the 604. 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—UISA," Section 2.3.5.1, "Processor Control Instructions—VEA," and Section 2.3.6.2, "Processor Control Instructions—OEA."
- Memory synchronization instructions—These instructions are used for memory synchronizing. See Section 2.3.4.7, "Memory Synchronization Instructions—UISA," Section 2.3.5.2, "Memory Synchronization Instructions—VEA," 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—VEA," and Section 2.3.6.3, "Memory Control Instructions—OEA."
- 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 604'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, and word 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 604 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 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 604 provides hardware support for all instructions defined for 32-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 604 provides hardware support for all instructions defined for 32-bit implementations. The 604 does not support the optional **fsqrt**, **fsqrts**, and **tlbia** instructions.

A defined instruction can have invalid forms. The 604 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 604 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.
- Instructions defined in the PowerPC architecture but not implemented in a specific PowerPC implementation. For example, instructions that can be executed on 64-bit PowerPC processors are considered illegal by 32-bit processors such as the 604.

  The following primary opcodes are defined for 64-bit implementations only and are illegal on the 604:

  2, 30, 58, 62
- 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 604 invokes the system illegal instruction error handler (a program exception) when it detects any instruction from this class or any instructions defined only for 64-bit implementations.

See Section 4.5.7, “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 32-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 32-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 32-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**) 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** or **rfi** instruction 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**) instruction is 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 the **mtmsr** sets the MSR[PR] bit, unless an **isync** immediately follows the **mtmsr** 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 604—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 604 provides the following supervisor-level instructions: **dcbi**, **mfmsr**, **mfspr**, **mfsr**, **mfsrin**, **mtmsr**, **mtspr**, **mtsr**, **mtsrin**, **rfi**, **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 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 604 and highlights any special information with respect to how the 604 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 integer exception register (XER), and into condition register (CR) fields.

2.3.4.1.1 Integer Arithmetic Instructions

Table 2-9 lists the integer arithmetic instructions for the PowerPC processors.

Table 2-9. Integer Arithmetic Instructions

Name	Mnemonic	Operand Syntax
Add Immediate	**addi**	**r**D,**r**A,SIMM
Add Immediate Shifted	**addis**	**r**D,**r**A,SIMM
Add	**add** (**add. addo addo.**)	**r**D,**r**A,**r**B
Subtract From	**subf** (**subf. subfo subfo.**)	**r**D,**r**A,**r**B
Add Immediate Carrying	**addic**	**r**D,**r**A,SIMM
Add Immediate Carrying and Record	**addic.**	**r**D,**r**A,SIMM

Table 2-9. Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax
Subtract from Immediate Carrying	**subfic**	**r**D,**r**A,SIMM
Add Carrying	**addc** (**addc.** **addco** **addco.**)	**r**D,**r**A,**r**B
Subtract from Carrying	**subfc** (**subfc.** **subfco** **subfco.**)	**r**D,**r**A,**r**B
Add Extended	**adde** (**adde.** **addeo** **addeo.**)	**r**D,**r**A,**r**B
Subtract from Extended	**subfe** (**subfe.** **subfeo** **subfeo.**)	**r**D,**r**A,**r**B
Add to Minus One Extended	**addme** (**addme.** **addmeo** **addmeo.**)	**r**D,**r**A
Subtract from Minus One Extended	**subfme** (**subfme.** **subfmeo** **subfmeo.**)	**r**D,**r**A
Add to Zero Extended	**addze** (**addze.** **addzeo** **addzeo.**)	**r**D,**r**A
Subtract from Zero Extended	**subfze** (**subfze.** **subfzeo** **subfzeo.**)	**r**D,**r**A
Negate	**neg** (**neg.** **nego** **nego.**)	**r**D,**r**A
Multiply Low Immediate	**mulli**	**r**D,**r**A,SIMM
Multiply Low	**mullw** (**mullw.** **mullwo** **mullwo.**)	**r**D,**r**A,**r**B
Multiply High Word	**mulhw** (**mulhw.**)	**r**D,**r**A,**r**B
Multiply High Word Unsigned	**mulhwu** (**mulhwu.**)	**r**D,**r**A,**r**B
Divide Word	**divw** (**divw.** **divwo** **divwo.**)	**r**D,**r**A,**r**B
Divide Word Unsigned	**divwu** **divwu.** **divwuo** **divwuo.**	**r**D,**r**A,**r**B

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 (**r**B). 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 604 arithmetic instructions may suffer this penalty. The summary overflow bit (SO) and overflow bit (OV) in the integer exception register are set to reflect an overflow condition of a 32-bit 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 2-10 summarizes the integer compare instructions.

Table 2-10. 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 2-11 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 2-11. Integer Logical Instructions

Name	Mnemonic	Operand Syntax
AND Immediate	**andi.**	**r**A,**r**S,UIMM
AND Immediate Shifted	**andis.**	**r**A,**r**S,UIMM
OR Immediate	**ori**	**r**A,**r**S,UIMM
OR Immediate Shifted	**oris**	**r**A,**r**S,UIMM
XOR Immediate	**xori**	**r**A,**r**S,UIMM
XOR Immediate Shifted	**xoris**	**r**A,**r**S,UIMM
AND	**and** (**and.**)	**r**A,**r**S,**r**B
OR	**or** (**or.**)	**r**A,**r**S,**r**B

Table 2-11. Integer Logical Instructions (Continued)

Name	Mnemonic	Operand Syntax
XOR	**xor** (**xor.**)	**r**A,**r**S,**r**B
NAND	**nand** (**nand.**)	**r**A,**r**S,**r**B
NOR	**nor** (**nor.**)	**r**A,**r**S,**r**B
Equivalent	**eqv** (**eqv.**)	**r**A,**r**S,**r**B
AND with Complement	**andc** (**andc.**)	**r**A,**r**S,**r**B
OR with Complement	**orc** (**orc.**)	**r**A,**r**S,**r**B
Extend Sign Byte	**extsb** (**extsb.**)	**r**A,**r**S
Extend Sign Half Word	**extsh** (**extsh.**)	**r**A,**r**S
Count Leading Zeros Word	**cntlzw** (**cntlzw.**)	**r**A,**r**S

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 2-12.

Table 2-12. Integer Rotate Instructions

Name	Mnemonic	Operand Syntax
Rotate Left Word Immediate then AND with Mask	**rlwinm** (**rlwinm.**)	**r**A,**r**S,SH,MB,ME
Rotate Left Word then AND with Mask	**rlwnm** (**rlwnm.**)	**r**A,**r**S,**r**B,MB,ME
Rotate Left Word Immediate then Mask Insert	**rlwimi** (**rlwimi.**)	**r**A,**r**S,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 2-13.

Table 2-13. Integer Shift Instructions

Name	Mnemonic	Operand Syntax
Shift Left Word	**slw** (**slw.**)	**r**A,**r**S,**r**B
Shift Right Word	**srw** (**srw.**)	**r**A,**r**S,**r**B
Shift Right Algebraic Word Immediate	**srawi** (**srawi.**)	**r**A,**r**S,SH
Shift Right Algebraic Word	**sraw** (**sraw.**)	**r**A,**r**S,**r**B

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 2-14.

Table 2-14. Floating-Point Arithmetic Instructions

Name	Mnemonic	Operand Syntax
Floating Add (Double-Precision)	**fadd** (**fadd.**)	**fr**D,**fr**A,**fr**B
Floating Add Single	**fadds** (**fadds.**)	**fr**D,**fr**A,**fr**B
Floating Subtract (Double-Precision)	**fsub** (**fsub.**)	**fr**D,**fr**A,**fr**B
Floating Subtract Single	**fsubs** (**fsubs.**)	**fr**D,**fr**A,**fr**B
Floating Multiply (Double-Precision)	**fmul** (**fmul.**)	**fr**D,**fr**A,**fr**C
Floating Multiply Single	**fmuls** (**fmuls.**)	**fr**D,**fr**A,**fr**C
Floating Divide (Double-Precision)	**fdiv** (**fdiv.**)	**fr**D,**fr**A,**fr**B

Table 2-14. Floating-Point Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax
Floating Divide Single	**fdivs** (**fdivs.**)	**fr**D,**fr**A,**fr**B
Floating Square Root (Double-Precision)	**fsqrt** (**fsqrt.**)	**fr**D,**fr**B
Floating Square Root Single	**fsqrts** (**fsqrts.**)	**fr**D,**fr**B
Floating Reciprocal Estimate Single	**fres** (**fres.**)	**fr**D,**fr**B
Floating Reciprocal Square Root Estimate	**frsqrte** (**frsqrte.**)	**fr**D,**fr**B
Floating Select	**fsel**	**fr**D,**fr**A,**fr**C,**fr**B

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 2-15.

Table 2-15. Floating-Point Multiply-Add Instructions

Name	Mnemonic	Operand Syntax
Floating Multiply-Add (Double-Precision)	**fmadd** (**fmadd.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Multiply-Add Single	**fmadds** (**fmadds.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Multiply-Subtract (Double-Precision)	**fmsub** (**fmsub.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Multiply-Subtract Single	**fmsubs** (**fmsubs.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Negative Multiply-Add (Double-Precision)	**fnmadd** (**fnmadd.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Negative Multiply-Add Single	**fnmadds** (**fnmadds.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Negative Multiply-Subtract (Double-Precision)	**fnmsub** (**fnmsub.**)	**fr**D,**fr**A,**fr**C,**fr**B
Floating Negative Multiply-Subtract Single	**fnmsubs** (**fnmsubs.**)	**fr**D,**fr**A,**fr**C,**fr**B

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 2-16. Floating-Point Rounding and Conversion Instructions

Name	Mnemonic	Operand Syntax
Floating Round to Single	**frsp** (**frsp.**)	**fr**D,**fr**B
Floating Convert to Integer Word	**fctiw** (**fctiw.**)	**fr**D,**fr**B
Floating Convert to Integer Word with Round toward Zero	**fctiwz** (**fctiwz.**)	**fr**D,**fr**B

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 2-17.

Table 2-17. 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 604, **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 2-18.

Table 2-18. 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 2-19 summarizes the floating-point move instructions.

Table 2-19. Floating-Point Move Instructions

Name	Mnemonic	Operand Syntax
Floating Move Register	**fmr** (**fmr.**)	**fr**D,**fr**B
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
- Floating-point load instructions
- Floating-point store instructions
- Memory synchronization instructions

Implementation Notes—The following describes how the 604 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 604 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 604 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 he 604 requires a direct-store protocol “Reply” from the device. If two translations cross from T = 0 into T = 1 space, a DSI exception is 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**, **lhzx**, **lhzux**, **lhax**, **lhaux**, **lwzx**, **lwzux**), the integer store indexed instructions (**stbx**, **stbux**, **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 604, 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 604, 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 604 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 4.5.6, “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 604 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**, **lhzx**, **lhzux**, **lhax**, **lhaux**, **lwzx**, and **lwzux**). In the 604, executing one of these invalid instruction forms causes CR0 to be set to an undefined value.
- For load with update instructions *(***lbzu**, **lbzux**, **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 604 sets GPR0 to an undefined value. If **r**A = **r**D, the 604 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 604.

Table 2-20 summarizes the integer load instructions.

Table 2-20. Integer Load Instructions

Name	Mnemonic	Operand Syntax
Load Byte and Zero	**lbz**	**r**D,d(**r**A)
Load Byte and Zero Indexed	**lbzx**	**r**D,**r**A,**r**B
Load Byte and Zero with Update	**lbzu**	**r**D,d(**r**A)
Load Byte and Zero with Update Indexed	**lbzux**	**r**D,**r**A,**r**B
Load Half Word and Zero	**lhz**	**r**D,d(**r**A)
Load Half Word and Zero Indexed	**lhzx**	**r**D,**r**A,**r**B

Table 2-20. Integer Load Instructions (Continued)

Name	Mnemonic	Operand Syntax
Load Half Word and Zero with Update	**lhzu**	**r**D,d(**r**A)
Load Half Word and Zero with Update Indexed	**lhzux**	**r**D,**r**A,**r**B
Load Half Word Algebraic	**lha**	**r**D,d(**r**A)
Load Half Word Algebraic Indexed	**lhax**	**r**D,**r**A,**r**B
Load Half Word Algebraic with Update	**lhau**	**r**D,d(**r**A)
Load Half Word Algebraic with Update Indexed	**lhaux**	**r**D,**r**A,**r**B
Load Word and Zero	**lwz**	**r**D,d(**r**A)
Load Word and Zero Indexed	**lwzx**	**r**D,**r**A,**r**B
Load Word and Zero with Update	**lwzu**	**r**D,d(**r**A)
Load Word and Zero with Update Indexed	**lwzux**	**r**D,**r**A,**r**B

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 2-21 summarizes the integer store instructions.

Table 2-21. Integer Store Instructions

Name	Mnemonic	Operand Syntax
Store Byte	**stb**	**r**S,**d(r**A**)**
Store Byte Indexed	**stbx**	**r**S,**r**A,**r**B
Store Byte with Update	**stbu**	**r**S,**d(r**A**)**
Store Byte with Update Indexed	**stbux**	**r**S,**r**A,**r**B
Store Half Word	**sth**	**r**S,**d(r**A**)**
Store Half Word Indexed	**sthx**	**r**S,**r**A,**r**B
Store Half Word with Update	**sthu**	**r**S,**d(r**A**)**
Store Half Word with Update Indexed	**sthux**	**r**S,**r**A,**r**B
Store Word	**stw**	**r**S,**d(r**A**)**
Store Word Indexed	**stwx**	**r**S,**r**A,**r**B
Store Word with Update	**stwu**	**r**S,**d(r**A**)**
Store Word with Update Indexed	**stwux**	**r**S,**r**A,**r**B

Implementation Notes—The following notes describe the 604 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**, **sthx**, **sthux**, **stwx**, **stwux**). In the 604, executing one of these invalid instruction forms causes CR0 to be set to an undefined value.
- For the store with update instructions *(***stbu**, **stbux**, **sthu**, **sthux**, **stwu**, **stwux**, **stfsu**, **stfsux**, **stfdu**, **stfdux***)*, when **r**A = 0, the instruction form is considered invalid. In this case, the 604 sets GPR0 to an undefined value.

2.3.4.3.5 Integer Load and Store with Byte Reverse Instructions

Table 2-22 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-reversal instructions (**lhbrx**, **lwbrx**, **sthbrx**, **stwbrx**).

In the 604, executing one of these invalid instruction forms causes CR0 to be set to an undefined value.

Table 2-22. Integer Load and Store with Byte Reverse Instructions

Name	Mnemonic	Operand Syntax
Load Half Word Byte-Reverse Indexed	**lhbrx**	**r**D,**r**A,**r**B
Load Word Byte-Reverse Indexed	**lwbrx**	**r**D,**r**A,**r**B
Store Half Word Byte-Reverse Indexed	**sthbrx**	**r**S,**r**A,**r**B
Store Word Byte-Reverse Indexed	**stwbrx**	**r**S,**r**A,**r**B

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 604 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 604 provides hardware support for **lmw**, **stmw**, **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).
- Executing an **lmw** instruction in which **r**A is in the range of registers to be loaded or in which RA = RT = 0 is invalid in the architecture. In the 604, 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 604's implementation of the **lmw** instruction allows one 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 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 word per external bus tenure. Bus tenures are pipelined, allowing a maximum tenure rate of one address tenure every three bus-clock cycles.
- 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 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 2-23. Integer Load and Store Multiple Instructions

Name	Mnemonic	Operand Syntax
Load Multiple Word	**lmw**	**r**D,**d(r**A**)**
Store Multiple Word	**stmw**	**r**S,**d(r**A**)**

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 2-24 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 2-24. Integer Load and Store String Instructions

Name	Mnemonic	Operand Syntax
Load String Word Immediate	**lswi**	**r**D,**r**A,NB
Load String Word Indexed	**lswx**	**r**D,**r**A,**r**B
Store String Word Immediate	**stswi**	**r**S,**r**A,NB
Store String Word Indexed	**stswx**	**r**S,**r**A,**r**B

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

As described in Section 4.5.6, “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 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 604 implementation of the load/store string instruction:

- The 604 provides hardware support for **lmw**, **stmw**, **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 604, 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 604 executes load string operations to cacheable memory at two cycles per word if they are word-aligned. Two additional cycles per instruction are required if they are not word-aligned. Cache-inhibited load string instructions require one bus tenure per word if they are aligned. An additional tenure per instruction is required if a cache-inhibited load string operation is not word aligned.
- The 604 executes store string operations to cacheable memory at a rate of one cycle per word if they are word-aligned. Cacheable store string operations that are not word-aligned require five cycles per word. Cache-inhibited store string instructions require one bus tenure per word if they are word-aligned. Two bus tenures per word are required if a store string operation is not word aligned.
- 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.

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.

Implementation Notes—The following notes characterize how the 604 treats exceptions:

- On the 604, if a floating-point number is not aligned on a word boundary, an alignment exception occurs.
- The floating-point load and store indexed instructions (**lfsx**, **lfsux**, **lfdx**, **lfdux**, **stfsx**, **stfsux**, **stfdx**, **stfdux**) are invalid when the Rc bit is one. In the 604, executing one of these invalid instruction forms causes CR0 to be set to an undefined value.

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

Table 2-25. 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 2-26 summarizes the floating-point store instructions.

Table 2-26. Floating-Point Store Instructions

Name	Mnemonic	Operand Syntax
Store Floating-Point Single	**stfs**	**fr**S,**d(r**A**)**
Store Floating-Point Single Indexed	**stfsx**	**fr**S,**r** B
Store Floating-Point Single with Update	**stfsu**	**fr**S,**d(**rA**)**
Store Floating-Point Single with Update Indexed	**stfsux**	**fr**S,**r** B
Store Floating-Point Double	**stfd**	**fr**S,**d(r**A**)**
Store Floating-Point Double Indexed	**stfdx**	**fr**S,**r**B
Store Floating-Point Double with Update	**stfdu**	**fr**S,**d(**rA**)**

Table 2-26. Floating-Point Store Instructions (Continued)

Name	Mnemonic	Operand Syntax
Store Floating-Point Double with Update Indexed	**stfdux**	**fr**S,**r** B
Store Floating-Point as Integer Word Indexed	**stfiwx**	**fr**S,**r**B

Some floating-point store instructions require conversions in the LSU. Table 2-27 shows the conversions made by the LSU when performing a Store Floating-Point Single instruction.

Table 2-27. 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 2-28 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 2-28. 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 604. 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. The 604 supports this denormalization by shifting the mantissa one bit at a time. Anywhere from 1 to 23 clock cycles are required to complete the denormalization, depending upon the value to be stored.

Because of how floating-point numbers are implemented in the 604, 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. In these cases, shifting the mantissa takes from 1 to 23 clock cycles, depending upon the value to be stored. These cycles are incurred during the store.

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 604, 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 604 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 2-29 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 2-29. 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 2-30, and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions.

Table 2-30. 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 2-31 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 2-31. Trap Instructions

Name	Mnemonic	Operand Syntax
Trap Word Immediate	**twi**	TO,rA,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—UISA

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—OEA,” for additional information.

Table 2-32. System Linkage Instruction—UISA

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

2.3.4.6 Processor Control Instructions—UISA

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—VEA,” for the **mftb** instruction and Section 2.3.6.2, “Processor Control Instructions—OEA,” 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 2-33 summarizes the instructions for reading from or writing to the condition register.

Table 2-33. Move to/from Condition Register Instructions

Name	Mnemonic	Operand Syntax
Move to Condition Register Fields	**mtcrf**	CRM,**r**S
Move to Condition Register from XER	**mcrxr**	**crf**D
Move from Condition Register	**mfcr**	**r**D

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 (UISA)

Table 2-34 lists the **mtspr** and **mfspr** instructions.

Table 2-34. Move to/from Special-Purpose Register Instructions (UISA)

Name	Mnemonic	Operand Syntax
Move to Special Purpose Register	**mtspr**	SPR,**r**S
Move from Special Purpose Register	**mfspr**	**r**D,SPR

2.3.4.7 Memory Synchronization Instructions—UISA

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, "Cache and Bus Interface Unit Operation," for additional information about these instructions and about related aspects of memory synchronization.

Table 2-35. Memory Synchronization Instructions—UISA

Name	Mnemonic	Operand Syntax
Load Word and Reserve Indexed	**lwarx**	**r**D,**r**A,**r**B
Store Word Conditional Indexed	**stwcx.**	**r**S,**r**A,**r**B
Synchronize	**sync**	—

The proper paired use of the **lwarx** with **stwcx.** instructions allows programmers to emulate common semaphore operations such as "test and set," "compare and swap," "exchange memory," and "fetch and add." The **lwarx** instruction must be paired with an **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—VEA," for details about additional memory synchronization (**eieio** and **isync**) instructions.

Implementation Notes—The following notes describe the 604 implementation of memory synchronization instructions:

- The PowerPC architecture requires that memory operands for Load and Reserve (**lwarx**) and Store Conditional (**stwcx.**) instructions must be word-aligned. If the operands to these instructions are not word-aligned on the 604, an alignment exception occurs.
- The PowerPC architecture indicates that the granularity with which reservations for **lwarx** and **stwcx.** instructions are managed is implementation-dependent. In the 604 reservations, this granularity is a 32-byte cache block.
- The **sync** instruction causes the 604 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** and **lwarx** instructions. In the 604, executing one of these invalid instruction forms causes CR0 to be set to an undefined value. The **stwcx.** instruction is the only load/store instruction that has a valid form if Rc is set. If the Rc bit is zero, the result of executing this instruction in the 604 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—VEA

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, “Cache and Bus Interface Unit Operation,” for more information. Table 3-34 shows the **mftb** instruction.

Table 2-36. 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 604:

- The 604 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 32-bit PowerPC implementation, the 604 supports separate access to the TBU and TBL, whereas 64-bit implementations can access the entire TB register at once.
- The time base counter is clocked at a frequency that is one-fourth that of the bus clock. Counting is enabled by assertion of the timebase enable ($\overline{\text{TBE}}$) input signal.

2.3.5.2 Memory Synchronization Instructions—VEA

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, "Cache and Bus Interface Unit Operation," for additional information about these instructions and about related aspects of memory synchronization.

Table 2-37 describes the memory synchronization instruction s defined by the VEA.

Table 2-37. 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 604 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 604 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—VEA

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—OEA," for information about supervisor-level cache, segment register manipulation, and translation lookaside buffer management instructions.

2.3.5.3.1 User-Level Cache Instructions—VEA

The instructions summarized in this section provide user-level programs the ability to manage on-chip caches if they are implemented. See Chapter 3, "Cache and Bus Interface Unit 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 604'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 2-38 summarizes the cache instructions defined by the VEA. Note that these instructions are accessible to user-level programs.

Table 2-38. User-Level Cache Instructions

Name	Mnemonic	Operand Syntax	Implementation Notes
Data Cache Block Touch	**dcbt**	**r**A,**r**B	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 604 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**	**r**A,**r**B	This instructions behaves like the **dcbt** instruction.
Data Cache Block Set to Zero	**dcbz**	**r**A,**r**B	The effective address is computed, translated, and checked for protection violations as defined in the VEA. If the 604 does not have exclusive access to the block, it presents an operation onto the 604 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 604 writes all zeros into the cache block. If the 604 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, the an alignment exception occurs. If the operation is successful, the cache block is marked modified.
Data Cache Block Store	**dcbst**	**r**A,**r**B	The effective address is computed, translated, and checked for protection violations as defined in the VEA. If the 604 does not have exclusive access to the block, it broadcasts the essence of the instruction onto the 604 bus (using the clean operation, described in Table 3-4). If the 604 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 604 bus. In this situation, the cache block is marked exclusive. Otherwise this instruction is treated as a no-op.
Data Cache Block Flush	**dcbf**	**r**A,**r**B	The effective address is computed, translated, and checked for protection violations as defined by the VEA. If the 604 does not have exclusive access to the block, it broadcasts the essence of the instruction onto the 604 bus (using the flush operation described in Table 3-4). In addition, if the addressed block is present in the cache, the 604 marks this data as invalid. On the other hand, if the 604 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 604 bus. In this situation, the cache block is marked invalid.

Table 2-38. User-Level Cache Instructions (Continued)

Name	Mnemonic	Operand Syntax	Implementation Notes
Instruction Cache Block Invalidate	**icbi**	**r**A,**r**B	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 604 marks it invalid. This instruction changes neither the content nor status of the data cache. In addition, the ICBI operation is broadcast on the 604 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 2-39.

Table 2-39. External Control Instructions

Name	Mnemonic	Operand Syntax
External Control In Word Indexed	**eciwx**	**r**D,**r**A,**r**B
External Control Out Word Indexed	**ecowx**	**r**S,**r**A,**r**B

The **eciwx** and **ecowx** instructions cause an alignment exception if they are not word-aligned.

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—OEA

This section describes the system linkage instructions (see Table 2-40). 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** instruction is a supervisor-level instruction that is useful for returning from an exception handler.

Table 2-40. System Linkage Instructions—OEA

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

2.3.6.2 Processor Control Instructions—OEA

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

Table 2-41 summarizes the instructions used for reading from and writing to the MSR.

Table 2-41. Move to/from Machine State Register Instructions

Name	Mnemonic	Operand Syntax
Move to Machine State Register	**mtmsr**	**r**S
Move from Machine State Register	**mfmsr**	**r**D

The OEA defines encodings of the **mtspr** and **mfspr** instructions to provide access to supervisor-level registers. The instructions are listed in Table 2-42.

Table 2-42. Move to/from Special-Purpose Register Instructions (OEA)

Name	Mnemonic	Operand Syntax
Move to Special Purpose Register	**mtspr**	SPR,**r**S
Move from Special Purpose Register	**mfspr**	**r**D,SPR

Encodings for the 604-specific SPRs are listed in Table 2-43.

Table 2-43 SPR Encodings for 604-Defined Registers (mfspr)

SPR[1]			Register Name
Decimal	spr[5–9]	spr[0–4]	
952	11101	11000	MMCR0
953	11101	11001	PMC1
954	11101	11010	PMC2
955	11101	11011	SIA
959	11101	11111	SDA
1010	11111	10010	IABR
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—OEA

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—(OEA)

Table 2-44 lists the only supervisor-level cache management instruction.

Table 2-44. Cache Management Supervisor-Level Instruction

Name	Mnemonic	Operand Syntax	Implementation Notes
Data Cache Block Invalidate	**dcbi**	**r**A,**r**B	The EA is computed, translated, and checked for protection violations as defined in the OEA. The 604 broadcasts the essence of the instruction onto the 604 bus (using the kill operation). In addition, if the addressed block is present in the cache, the 604 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 (OEA)

The instructions listed in Table 2-45 provide access to the segment registers for 32-bit implementations. 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 2-45. Segment Register Manipulation Instructions

Name	Mnemonic	Operand Syntax
Move to Segment Register	**mtsr**	SR**,r**S
Move to Segment Register Indirect	**mtsrin**	**r**S**,r**B
Move from Segment Register	**mfsr**	**r**D**,**SR
Move from Segment Register Indirect	**mfsrin**	**r**D**,r**B

2.3.6.3.3 Translation Lookaside Buffer Management Instructions—(OEA)

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 2-46 summarizes the operation of the TLB instructions in the 604.

Table 2-46. Translation Lookaside Buffer Management Instruction

Name	Mnemonic	Operand Syntax	Implementation Notes
TLB Invalidate Entry	**tlbie**	**r**B	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 604 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 604 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 ($\overline{\text{ARTRY}}$ is not asserted). 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 Section 3.9.6, "Cache Reaction to Specific Bus Operations."

Implementation Note—The **tlbia** instruction is optional for an implementation if its effects can be achieved through some other mechanism. As described above, the **tlbie** instruction can be used to invalidate a particular index of the TLB based on EA[14–19]. With that concept in mind, a sequence of 64 **tlbie** instructions followed by a single **tlbsync** instruction would cause all the 604 TLB structures to be invalidated (for EA[14–19] = 0, 1, 2,..., 63). Therefore the **tlbia** instruction is not implemented on the 604. Execution of a **tlbia** instruction causes an illegal instruction program exception.

Because the presence and exact semantics of the TLB management instructions is implementation-dependent, system software should incorporate uses of these instructions into subroutines to minimize compatibility problems.

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
Cache and Bus Interface Unit Operation

This chapter describes the organization of the 604'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 604 contains separate 16-Kbyte, four-way set-associative instruction and data caches and also provides support for secondary (L2) caching. The cache block size is 32 bytes. The cache is designed to adhere to a write-back policy, but the 604 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 604 cache implementation has the following characteristics:

- Separate 16-Kbyte instruction and data caches (Harvard architecture)
- Instruction and data caches are four-way set associative.
- Caches implement an LRU replacement algorithm within each set.
- The cache directories are physically addressed. The physical (real) address tag is stored in the cache directory.
- Both the instruction and data caches have 32-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)
- Each cache can be invalidated or locked by setting the appropriate bits in the hardware implementation dependent register 0 (HID0), a special-purpose register (SPR) specific to the 604.

The 604 uses eight-word burst transactions to transfer cache blocks to and from memory. When requesting burst reads, the 604 presents a double-word–aligned address. Memory controllers are expected to transfer this double word of data first, followed by double words from increasing addresses, wrapping back to the beginning of the eight-word block as required.

Burst misses can be buffered into two 8-word line-fill buffers before being loaded into the cache. Writes of cache blocks by the 604 (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 604 instruction and data caches is shown in Figure 3-1.

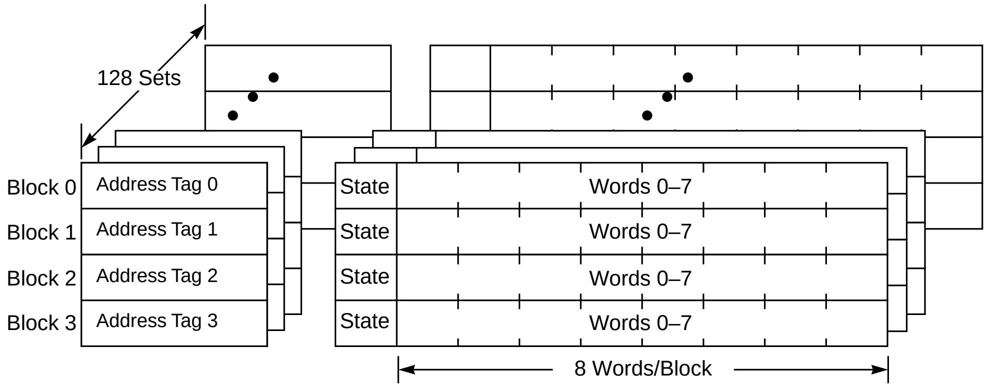


Figure 3-1. Cache Organization

As shown in Figure 3-2, the instruction cache is connected to the bus interface unit (BIU) with a 64-bit bus; likewise, the data cache is connected both to the BIU and the load/store unit (LSU) with a 64-bit bus. The 64-bit bus allows two instructions to be loaded into the instruction cache or a double word (for example, a double-precision floating-point operand) to be loaded into the data cache in a single clock. The instruction cache provides a 128-bit interface to the instruction fetcher, so four instructions can be made available to the instruction unit in a single clock cycle.

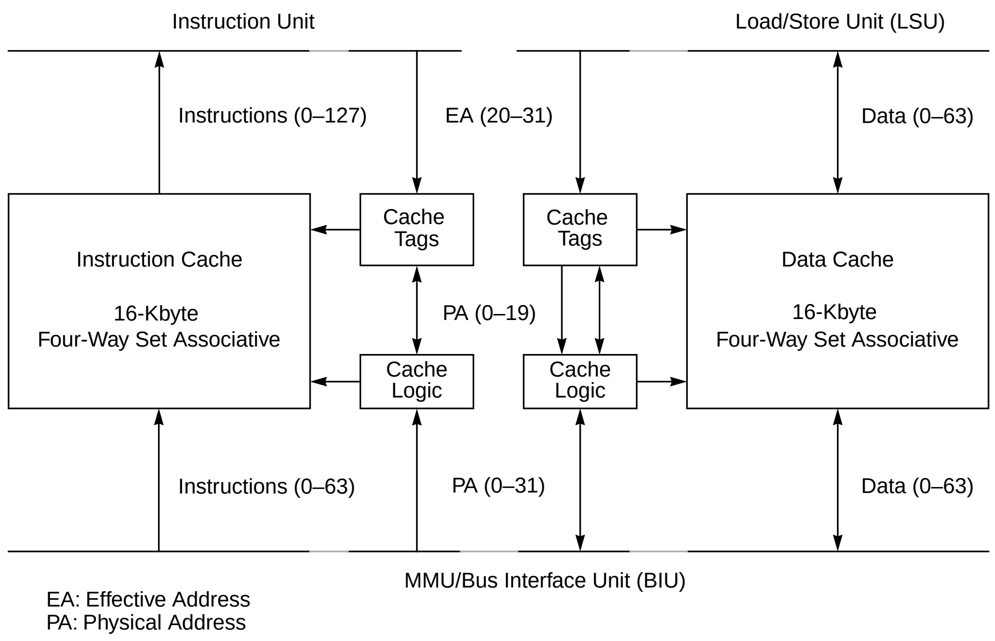


Figure 3-2. Cache Integration

3.1 Data Cache Organization

As shown in Figure 3-2, the physically-addressed 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 16-Kbyte, four-way set data cache is nonblocking write-back cache with hardware reload. The data cache can continue to process loads and stores while as many as four block fill requests are in progress.

The set associativity of the data cache is shown in Figure 3-1.

Each cache block contains eight contiguous words from memory that are loaded from an eight-word boundary (that is, bits A27–A31 of the EA are zero); as a result, cache blocks are aligned with page boundaries. Within a single cycle, the data cache provides a double-word access to the LSU.

The data cache supports a coherent memory system using the four-state MESI coherency (modified/exclusive/shared/invalid) protocol. Dual-ported data cache tags are implemented to prevent snooping accesses from affecting other bus traffic, except when snooping hits modified data. The LSU is blocked for one cycle to copy the cache block of data into a write-back buffer. The data cache can be invalidated on a block or invalidate-all granularity. Also, data cache enable, lock, and parity checking enable bits can be set in hardware implementation register 0 (HID0).

3.2 Instruction Cache Organization

The 16-Kbyte, four-way set-associative instruction cache is physically-indexed. The organization of the instruction cache, shown in Figure 3-1, is identical to that of the data cache. Each cache block contains eight contiguous words from memory that are loaded from an eight-word boundary (that is, bits A27–A31 of the effective addresses are zero); as a result, cache blocks are aligned with page boundaries.

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) is compatible with those of the PowerPC 601™ and PowerPC 603™ microprocessors. It implements both tenured and split-transaction modes and can handle as many as three outstanding transactions in pipelined mode. If permitted, the BIU can complete one or more write transactions between the address and data tenures of a read transaction. The BIU has 32-bit address and 64-bit data buses protected by byte parity.

The BIU implements the critical-double-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 double 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.

The bus can be run at 1x, 2/3x, 1/2x or 1/3x the speed of the processor. The programmable on-chip phase-locked loop (PLL) generates the necessary processor clocks from the bus clock.

When a memory access fails to hit in the cache, the 604 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 3-3.

The 604 implements separate MMUs, one for instruction accesses and one for data accesses. Virtual address translation uses two 128-entry, two-way set-associative (64 x 2) translation lookaside buffers (TLBs), one for instruction accesses and one for data accesses. The 604 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.

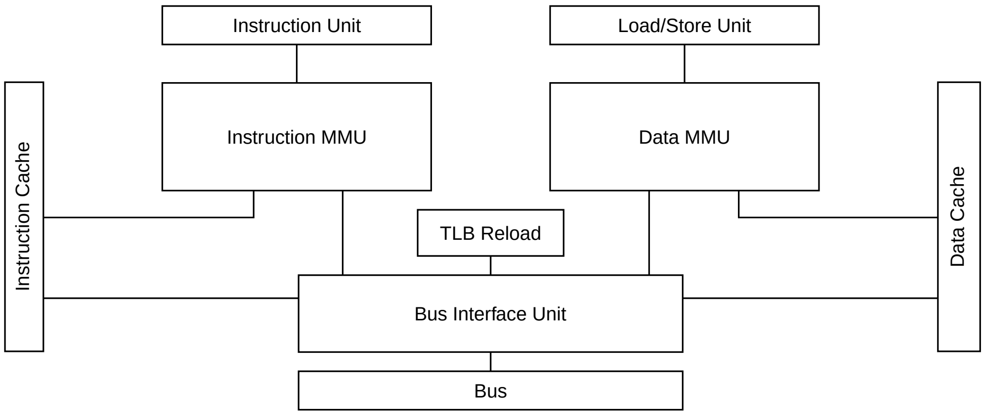


Figure 3-3. Bus Interface Unit and MMU

As shown in Figure 3-4, the 604 implements four types of memory queues to support the four types of operations—line-fill, write, copy-back, and invalidation operations. For a line-fill operation, the line-fill address from either the instruction or data cache is kept in the memory address queue until the address can be sent out in an address tenure. After the address tenure, the address is transferred to the line-fill address queue, which releases the address bus for other transactions in split-transaction mode. As each double word for the line-fill operation is returned, it is transferred to the line-fill buffer, where it is forwarded to the LSU.

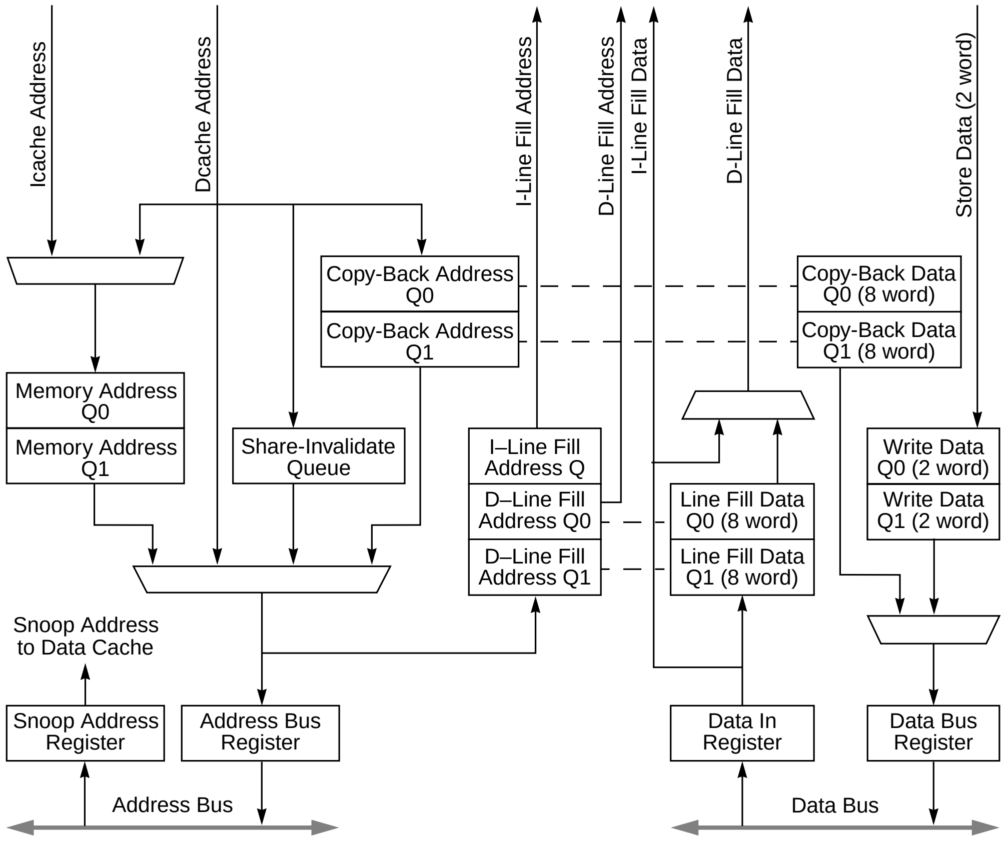


Figure 3-4. Memory Queue Organization

For write operations, the address is kept in the memory address queue and the data is kept in the write buffer until both can be sent out in a write transaction. Similarly, for copy-back operations the address is kept in the copy-back address queue and the data is kept in the copy-back buffer until both can be sent out in a burst write transaction. For a cache control instruction or a store to a shared cache block, the address is kept in the cache control address queue until an address-only transaction is sent out to broadcast the cache control command. Because all address queues in the 604 are treated as part of the coherent memory system, they are checked against the data cache and snoop addresses to ensure data consistency and to maintain MESI coherency protocol.

To support the increased bandwidth of the nonblocking caches, the BIU can handle as many as three pipelined transactions before data has to be provided by the memory system. The three outstanding transactions can be any combination of the following—two noncacheable or write-through write operations, two data cache reloads, one instruction cache reload, and two cache block copybacks. In addition, address-only transactions are not counted in the three outstanding transactions.

For details concerning the signals, see Chapter 7, "Signal Descriptions," and for information regarding bus protocol, see Chapter 8, "System Interface Operation."

3.4 Memory Coherency Actions

The following sections describe memory coherency actions in response to various operations and instructions.

3.4.1 604-Initiated Load and Store Operations

The following tables provide an overview of the behavior of the 604 with respect to load and store operations. Table 3-1 does not include noncacheable cases. The first three cases (load when the cache block is marked I) also involve selecting a replacement class and copying back any modified data that may have resided in that replacement class.

Table 3-1. Memory Coherency Actions on Load Operations

Cache State	Bus Operation	Snoop Response	Action
I	Read	–$\overline{\text{ARTRY}}$ –$\overline{\text{SHD}}$	Load data and mark E
I	Read	–$\overline{\text{ARTRY}}$ $\overline{\text{SHD}}$	Load data and mark S
I	Read	$\overline{\text{ARTRY}}$	Retry read operation
S	None	Don't care	Read from cache
E	None	Don't care	Read from cache
M	None	Don't care	Read from cache

Table 3-2 does not address the noncacheable or write-through cases and does not completely describe the exact mechanisms for the operations described. The first two cases also involve selecting a replacement class and copying back any modified data that may have resided in that replacement class. The state of the $\overline{\text{SHD}}$ signal is unimportant in this table.

Table 3-2. Memory Coherency Actions on Store Operations

Cache State	Bus Operation	Snoop Response	Action
I	RWITM	–$\overline{\text{ARTRY}}$	Load data, modify it, mark M
I	RWITM	$\overline{\text{ARTRY}}$	Retry the RWITM
S	Kill	–$\overline{\text{ARTRY}}$	Modify cache, mark M
S	Kill	$\overline{\text{ARTRY}}$	Retry the kill
E	None	Don’t care	Modify cache, mark M
M	None	Don’t care	Modify cache

3.5 Sequential Consistency

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

3.5.1 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 specified by the program with respect to exceptions and data dependencies. Note that all potential precise exceptions are resolved before memory accesses that miss in the cache are forwarded onto the memory queue for arbitration onto the bus. 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.

3.5.2 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. The ability to order memory accesses weakly provides opportunities for more efficient use of the system bus. Unless a dependency exists, the 604 allows read operations to precede store 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.

3.5.3 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 604, 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 604 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 race conditions associated with the order in which the processors are granted bus access.

If the 604 uses an L2 cache, the system designer must ensure the memory system responds to the SYNC and EIEIO bus operations in such a way that the required ordering of memory operations is preserved.

3.6 Memory and Cache Coherency

The 604 can support a fully coherent 4-Gbyte (2^{32}) 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 32-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.

3.6.1 Data Cache Coherency Protocol

Each 32-byte cache block in the 604 data cache is in one of four states. Addresses presented to the cache are indexed into the cache directory and are compared against the cache directory tags. If no tags match, the result is a cache miss. If a tag match occurs, a cache hit has occurred and the directory indicates the state of the block through three state bits kept with the tag.

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). The four MESI states are defined in Table 3-3 and illustrated in Figure 3-5.

Table 3-3. MESI State Definitions

MESI State	Definition
Modified (M)	The addressed block is valid in the cache and in only this cache. The block is modified with respect to system memory—that is, the modified data in the block has not been written back to memory.
Exclusive (E)	The addressed block is in this cache only. The data in this block is consistent with system memory.
Shared (S)	The addressed block is valid in the cache and in at least one other cache. This block is always consistent with system memory. That is, the shared state is shared-unmodified; there is no shared-modified state.
Invalid (I)	This state indicates that the addressed block is not resident in the cache and/or any data contained is considered not useful.

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 chip, or another processor had a queuing problem that prevented appropriate snooping from occurring).

To maximize performance, the 604 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.

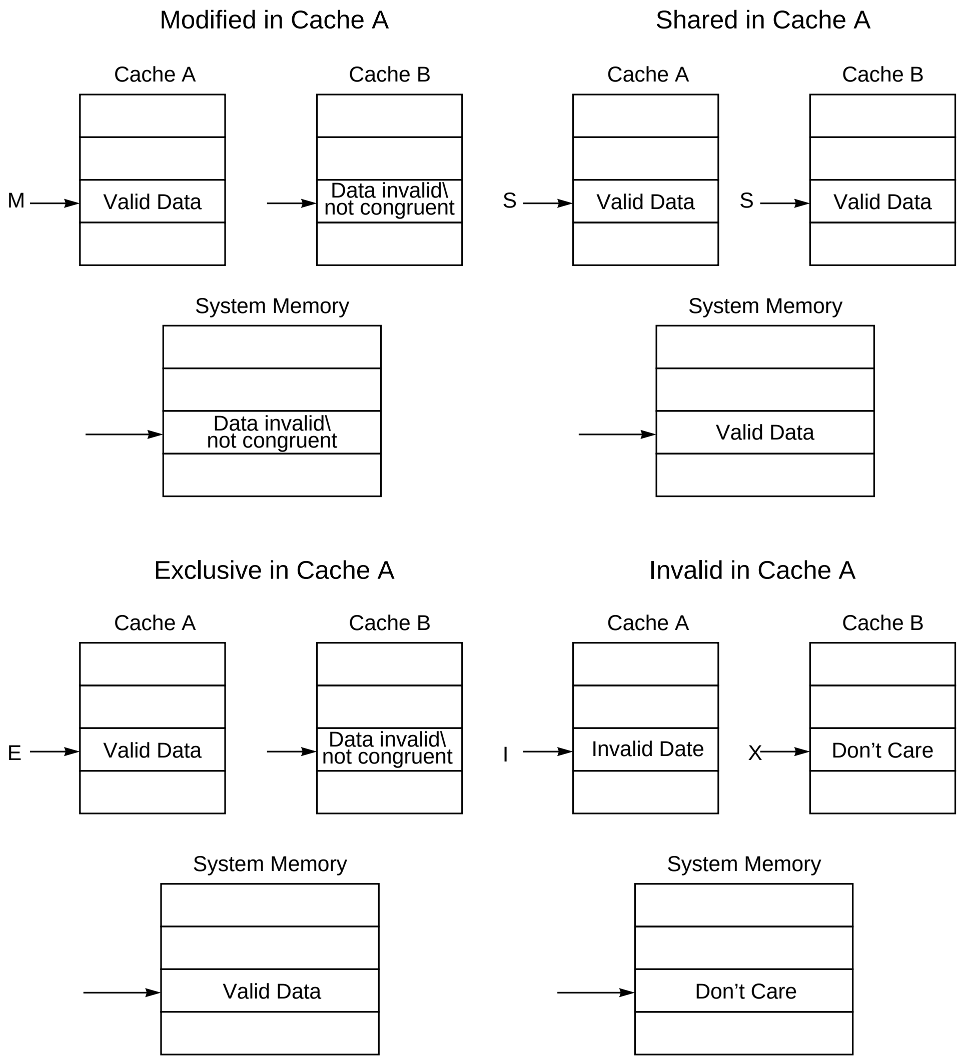


Figure 3-5. MESI States

3.6.2 Coherency and Secondary Caches

The 604 supports the use of a larger secondary cache that can be implemented in different configurations. The use of an L2 cache can serve to further improve performance by further 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.

L2 caches must forward all relevant system bus traffic onto the 604 so the 604 can take the appropriate actions to maintain memory coherency as defined by the PowerPC architecture.

3.6.3 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 storage 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 real 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.

3.6.4 MESI State Diagram

The 604 provides dedicated hardware to provide data cache coherency by snooping bus transactions. The address retry capability of the 604 enforces the MESI protocol, as shown in Figure 3-6. Figure 3-6 assumes that the WIM bits are set to 001; that is, write-back, caching-not-inhibited, and memory coherency enforced.

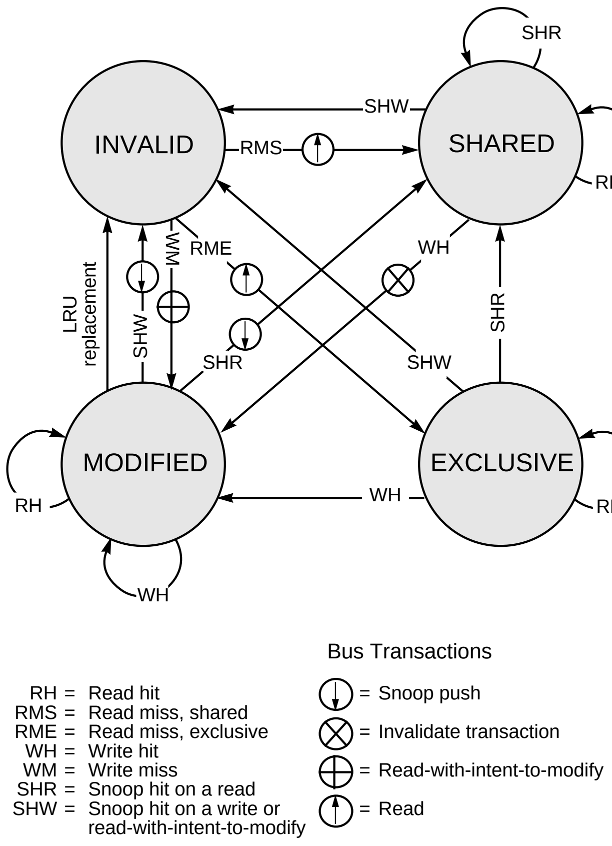


Figure 3-6. MESI Cache Coherency Protocol—State Diagram (WIM = 001)

Table 3-6 gives a detailed list of MESI transitions for various operations and WIM bit settings.

3.6.5 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 604 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 604 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.

3.6.6 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.

Additional information on what bus operations are generated for the various instructions and state conditions can be found in Chapter 8, “System Interface Operation.”

3.7 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 7—Disable snoop response high state restore. If this bit is set, the processor cannot drive the $\overline{\text{SHD}}$ and $\overline{\text{ARTRY}}$ signals to the high (negated) state, and the system must restore the signals to the high state. See Chapter 7, “Signal Descriptions,” for more information.
- 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 19—Data cache lock. Setting this bit locks the data cache, in which case all cache misses are treated as cache-inhibited. Cache hits occur as normal, and cache snoops and other operations continue to work as normal. This is the only way to deallocate an entry. If the data cache is locked when the **dcbz** instruction is executed, it takes an alignment exception, provided the target address had been translated correctly.
- 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. 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.

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

3.8 Cache Control 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 604 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 604 bus interface so that all processors in a multiprocessor system can take the appropriate actions. The 604 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 7, “Signal Descriptions.”

3.8.1 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 604 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 604 bus unconditionally to support this function throughout a system’s memory hierarchy.

3.8.2 Instruction Synchronize (isync)

The **isync** instruction causes the 604 to purge its instruction buffers and fetch the next sequential instruction.

3.8.3 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 604 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 604 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 604 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.

3.8.4 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 604 does not already have exclusive access to this cache block, it presents a kill operation onto the 604 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 604 writes all zeros into the cache block. In the event that the 604 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.

3.8.5 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 604 does not have modified data in this block, the 604 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 604 bus. In this situation, the cache block is marked as exclusive. Otherwise this instruction is treated as a no-op.

3.8.6 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 604 does not have modified data in this cache block, it broadcasts a flush operation onto the 604 bus. If the addressed cache block is in the cache, the 604 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 604 bus and the cache block is marked as invalid.

3.8.7 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 604 broadcasts a kill operation onto the 604 bus. If the addressed cache block is in the cache, the 604 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.

3.9 Basic Cache Operations

This section describes operations that can occur to the cache, and how these operations are implemented in the 604.

3.9.1 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.

3.9.2 Cache Cast-Out Operation

The 604 uses an LRU replacement algorithm to determine which of the four 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.

3.9.3 Cache Block Push Operation

When a cache block in the 604 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. The 604 supports two kinds of push operations—normal push operations and enveloped high-priority push operations, which are described in Section 3.9.7, “Enveloped High-Priority Cache Block Push Operation.”

3.9.4 Atomic Memory References

The **lwarx**/**stwcx.** instruction combination can be used to emulate atomic memory references. These instructions are described in Chapter 2, “PowerPC 604 Processor Programming Model.”

3.9.5 Snoop Response to Bus Operations

When the 604 is not the bus master, it monitors bus traffic and performs cache and memory-queue snooping as appropriate. The snooping operation is triggered by the receipt of a qualified snoop request. A qualified snoop request is generated by the simultaneous assertion of the $\overline{\text{TS}}$ and $\overline{\text{GBL}}$ bus signals.

Instruction processing is interrupted for one clock cycle only when a snoop hit occurs and the snoop state machine determines a push-out operation is required.

The 604 maintains a write queue of bus operations in progress and/or pending arbitration. This write queue is also snooped in response to qualified snoop requests. Note that block-length (four beat) write operations are always snooped in the write queue; however, single-beat writes are not snooped. Coherency for single-beat writes is maintained through the use of cache operations that are broadcast with the write on the system interface or the **lwarx**/**stwcx.** instructions.

The 604 drives two snoop status signals ($\overline{\text{ARTRY}}$ and $\overline{\text{SHD}}$) in response to a qualified snoop request that hits. These signals provide information about the state of the addressed block for the current bus operation. For more information about these signals, see Chapter 7, “Signal Descriptions.”

3.9.6 Cache Reaction to Specific Bus Operations

There are several bus transaction types defined for the 604 bus. The 604 must snoop these transactions and perform the appropriate action to maintain memory coherency; see Table 3-4. For example, because single-beat write operations are not snooped when they are queued in the memory unit, additional operations such as flush or kill operations, must be broadcast when the write is passed to the system interface to ensure coherency.

A processor may assert $\overline{\text{ARTRY}}$ for any bus transaction due to internal conflicts that prevent the appropriate snooping. In general, if $\overline{\text{ARTRY}}$ is not asserted, each snooping processor must take full ownership for the effects of the bus transaction with respect to the state of the processor.

The transactions in Table 3-4 correspond to the transfer type signals TT0–TT4, which are described in Section 7.2.4.1, “Transfer Type (TT0–TT4).”

Table 3-4. Response to Bus Transactions

Transaction	Response
Clean block	The clean operation is an address-only bus transaction, initiated by executing a **dcbst** instruction. This operation affects only blocks marked as modified (M). Assuming the $\overline{\text{GBL}}$ signal is asserted, modified blocks are pushed out to memory, changing the state to E.
Flush block	The flush operation is an address-only bus transaction initiated by executing a **dcbf** instruction. Assuming the $\overline{\text{GBL}}$ signal is asserted, the flush block operation results in the following: • If the addressed block is in the S or E state, the state of the addressed block is changed to I. • If the addressed block is in the M state, the snooping device asserts $\overline{\text{ARTRY}}$ and $\overline{\text{SHD}}$, the modified block is pushed out of the cache, and its state is changed to I.
Write-with-flush Write-with-flush-atomic	Write-with-flush and write-with-flush-atomic operations are issued by a processor after executing stores or **stwcx.**, respectively to memory in a variety of different states, particularly noncacheable and write-through. 60x processors do not use this transaction code for burst transfers, but system use for bursts is not precluded. If they appear on the bus and the GBL bit is asserted, the 60x processors have the same snoop response as for flush block, except that a hit on the reservation address causes loss of the reservation.
Kill block	Kill block is an address-only transaction issued by a processor after executing a **dcbi** instruction, a **dcbz** instruction to a location marked I or S, or a write operation to a block marked S. If a kill-block transaction appears on the bus, and the GBL bit is asserted, the addressed block is forced to the I state if it is in the cache.
Write-with-kill	In a write-with-kill operation, the processor snoops the cache for a copy of the addressed block. If one is found, an additional snoop action is initiated internally and the block is forced to the I state, killing modified data that may have been in the block. In addition to snooping the cache, the three-entry write queue is also snooped. A kill operation that hits an entry in the write queue purges that entry from the queue.
Read Read-atomic	Read is used by most single-beat or burst reads on the bus. A read on the bus with the GBL bit asserted causes the following snoop responses: • If the addressed block is in the cache in the I state, the processor takes no action. • If the addressed block is in the cache in the S state, the processor asserts the $\overline{\text{SHD}}$ snoop status signal. • If the addressed block is in the cache in the E state, the processor asserts the $\overline{\text{SHD}}$ snoop status signal and changes the state of that cache block to S. • If the addressed block is in the cache in the M state, the processor asserts both the $\overline{\text{ARTRY}}$ and $\overline{\text{SHD}}$ snoop status signals and changes the state of that block in the cache from E to S. Read-atomic operations appear on the bus in response to **lwarx** instruction and receive the same snooping treatment as a read operation.

Table 3-4. Response to Bus Transactions (Continued)

Transaction	Response
Read-with-intent-to-modify (RWITM) RWITM atomic	The RWITM transaction is issued to acquire exclusive use of a memory location for the purpose of modifying it. One example is a processor that writes to a block that is not currently in its cache. When GBL is asserted, RWITM transactions on the bus cause the processors to take the following snoop actions: • If the addressed block is not in the cache, it takes no action. • If the addressed block is in the cache in the S or E state, the processor changes the state of that block in the cache to I. • If the addressed block is present in the cache in the XM state, then the 60x asserts both the ARTRY and the SHARED snoop status signals, pushes the dirty block out of the cache and changes the state of that block in the cache from XM to INV. RWITM atomic appears on the bus in response to the **stwcx.** instruction and receives the same snooping treatment as RWITM.
TLBSYNC	This TLB synchronize operation is an address-only transaction placed onto the bus by a 604 when it executes a **tlbsync** instruction. When the TLBSYNC bus operation is detected by a snooping 604, the 604 asserts the $\overline{\text{ARTRY}}$ snoop status if any operations based on an invalidated TLB are pending.
TLB invalidate	A TLB invalidate transaction is an address-only transaction issued by a processor when it executes a **tlbie** instruction. The address transmitted as part of this transaction contains bits 12–19 of the EA in their correct respective bit positions. In response to a TLB invalidate operation, snooping processors invalidate the entire congruence class in any TLBs associated with the specified EA. In addition, a snooping 604 also asserts the $\overline{\text{ARTRY}}$ snoop status when it has a pending TLB invalidate operation, and a second TLB invalidate operation is detected. For more information on the **tlbie** instruction, see Section 2.3.6.3.3, “Translation Lookaside Buffer Management Instructions—(OEA).”
I/O reply	The I/O reply operation is part of the direct-store operation. It serves as the final bus operation in the series of bus operations that service a direct-store operation.
EIEIO	An EIEIO operation is put onto the bus as a result of executing an **eieio** instruction. The **eieio** instruction enforces ordered execution of accesses to noncacheable memory. The 604s internally enforce ordering of such accesses with respect to the **eieio** instruction in that noncacheable accesses due to instructions that occur before the **eieio** instruction in the program order are placed on the bus before any noncacheable accesses that result from instructions that occur after the **eieio** instruction with the EIEIO bus operation separating the two sets of bus operations. If the system implements a mechanism that allows reordering of noncacheable requests, the appearance of an EIEIO operation should cause it to force ordering between accesses that occurred before and those that occur after.
SYNC	The **sync** instruction generates an address-only transaction, which the 604 places onto the bus. When a 604 detects a SYNC operation on the bus, it asserts the $\overline{\text{ARTRY}}$ snoop status if any other snooped cache operations are pending in the device.
Read-with-no-intent-to-cache (RWNITC)	A RWNITC operation is issued by a bus-attached device as TT(4,0–3) = 0b10101—like a read, but with TT4 = 1). The 604 snoops this operation and if it gets a cache hit on a block marked M, it writes the block back to memory and marks it E. This operation is useful for a graphics adapter that reads display data from memory. This data may be in the processor’s cache and may be updated frequently. Because the adapter does not cache the data, the processor need not leave the block in the S state, requiring a bus operation to regain exclusive access.

Table 3-4. Response to Bus Transactions (Continued)

Transaction	Response
XFERDATA	XFERDATA read and write operations are bus transactions that result from execution of the **eciwx** or **ecowx** instructions, respectively. These instructions assist certain adapter types (especially displays) to make high-speed data transfers. They do this by calculating an effective address, translating it, and presenting the resulting physical address to the adapter. The XFERDATA read and write operations transfer a word of data to or from the processor, respectively. They also present the 4-bit resource ID (RID) field, using the concatenation of the bits TBST || TSIZ[0–2]. These transactions are unique in the sense that the address that is transferred does not select the slave device; it is simply being passed to the slave device for use in a subsequent transaction. Rather, the RID bits are used to select among the slave devices. Although the intent of these instructions is that the slave device that is selected by the RID bits will use the address that is transferred in a subsequent data transfer, the exact nature of this data transfer is not defined by 604 bus specifications. It is a private transfer that can be defined by the system like any other direct memory access.

3.9.7 Enveloped High-Priority Cache Block Push Operation

If the 604 has a read operation outstanding on the bus and another pipelined bus operation hits against a modified block, the 604 provides a high-priority push operation. This transaction can be enveloped within the address and data tenures of a read operation. This feature prevents deadlocks in system organizations that support multiple memory-mapped buses. More specifically, the 604 internally detects the scenario where one or more load requests are outstanding and the processor has pipelined a write operation on top of the load. Normally, when the data bus is granted to the 604, the resulting data bus tenure is used for the load operation.

The enveloped high-priority cache block push feature defines a bus signal, the data bus write only qualifier ($\overline{\text{DBWO}}$), which, when asserted with a qualified data bus grant, indicates that the resulting data tenure should be used for the first store operation instead. If no store operation is pending, the first read operation is performed. If no write operation is pending, the 604 can perform a read operation. This signal is described in detail in Section 8.11, “Using Data Bus Write Only.” Note that the enveloped copy-back operation is an internally pipelined bus operation.

3.9.8 Bus Operations Caused by Cache Control Instructions

Table 3-5 provides an overview of the bus operations initiated by cache control instructions. Note that Table 3-5 assumes that the WIM bits are set to 001; that is, since the cache is operating in write-back mode, caching is permitted and coherency is enforced.

3.9.9 Cache Control Instructions

Table 3-5 lists bus operations performed by the 604 when they execute cache control instructions.

Table 3-5. 604 Bus Operations Initiated by Cache Control Instructions

Instruction	Cache State	Next Cache State	Bus Operation	Comment
sync	Don’t care	No change	SYNC	First clears memory queue
eieio	Don’t care	No change	EIEIO	No clear meaning
icbi	Don’t care	I	ICBI	—
dcbi (invalidate)	Don’t care	I	Kill	—
dcbf (flush)	E, S, I	I	Flush	—
	M	I	Write-with-kill	Marked as write-through
dcbst (store)	E, S, I	No change	Clean	—
	M	E	Write-with-kill	Marked as write-through
dcbz (zero)	I	M	Kill	May also replace
	S	M	Kill	—
	M, E	M	None	Write over modified data
dcbt, **dcbtst**	I	E, S	Read	State change on reload
	M, E, S	No Change	None	—
tlbsync	Don’t care	No change	TLBSYNC	—

Table 3-5 does not include noncacheable or write-through cases, nor does it completely describe the mechanisms for the operations described. For more information, see Section 3.10, “Cache Actions.”

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 604 interface so that all processors in a multiprocessor system can take appropriate actions. The 604 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 604, see Chapter 8, “System Interface Operation.”

3.10 Cache Actions

Table 3-6 lists the actions that occur for various operations depending on different WIM bit settings.

Table 3-6. Cache Actions

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
000	I	Load	Read	000	01010	(n/a)	(None)	Load the block of data into cache forward data from load mark cache block E
000	I	Load	Read	000	01010	(n/a)	$\overline{\text{SHD}}$	Load the block of data into cache load from cache mark cache block S
000	I	Load	Read	000	01010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}\&\overline{\text{SHD}}$	Release the bus retry the operation
000	M E S	Load	(None)	(n/a)	(n/a)	(n/a)	(n/a)	Load from cache
001	I	Load	Read	001	01010	(n/a)	(None)	Load the block of data into cache mark cache block E load from cache
001	I	Load	Read	001	01010	(n/a)	$\overline{\text{SHD}}$	Load the block of data into cache load from cache mark cache block S
001	I	Load	Read	001	01010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}\&\overline{\text{SHD}}$	Release the bus retry the operation
001	M E S	Load	(None)	(n/a)	(n/a)	(n/a)	(n/a)	Load from cache
011 010 110 111	E S I	Load	Single- beat read	01M 11M	01010	(n/a)	(None) or $\overline{\text{SHD}}$	Load from main memory
011 010 110 111	E S I	Load	Single- beat read	01M 11M	01010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}\&\overline{\text{SHD}}$	Release the bus retry the operation
011 010 110 111	M	Load	Single- beat read	01M 11M	01010	(n/a)	(None) or $\overline{\text{SHD}}$	Paradox—cache should be I load from main memory
011 010 110 111	M	Load	Single- beat read	01M 11M	01010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}\&\overline{\text{SHD}}$	Paradox—cache should be I release the bus retry the operation
100	I	Load	Read	100	01010	(n/a)	(None)	Load the block of data into cache load from cache mark the cache block E
100	I	Load	Read	100	01010	(n/a)	$\overline{\text{SHD}}$	Load the block of data into cache load from cache mark cache block S

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
100	I	Load	Read	100	01010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
100	M E S	Load	(None)	(n/a)	(n/a)	(n/a)	(n/a)	Load from cache
101	I	Load	Read	101	01010	(n/a)	(None)	Load the block of data into cache load from cache mark cache E
101	I	Load	Read	101	01010	(n/a)	$\overline{\text{SHD}}$	Load the block of data into cache load from cache mark cache block S
101	I	Load	Read	101	01010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
101	M E S	Load	(None)	(n/a)	(n/a)	(n/a)	(n/a)	Load from cache
000	I	**lwarx**	Read atomic	000	11010	Set by this op	(None)	Load the block of data into cache set reservation load from cache mark cache block E
000	I	**lwarx**	Read atomic	000	11010	Set by this op	$\overline{\text{SHD}}$	Load the block of data into cache set reservation load from cache mark cache block S
000	I	**lwarx**	Read atomic	000	11010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
000	M E S	**lwarx**	**lwarx** reservation set	000	00001	Set by this op	(None) or $\overline{\text{SHD}}$	Set reservation load from cache
000	M E S	**lwarx**	**lwarx** reservation set	000	00001	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	I	**lwarx**	Read atomic	001	11010	Set by this op	(None)	Load the block of data into cache mark cache block E set reservation load from cache
001	I	**lwarx**	Read atomic	001	11010	Set by this op	$\overline{\text{SHD}}$	Load the block of data into cache set reservation load from cache mark cache block S
001	I	**lwarx**	Read atomic	001	11010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	M E S	**lwarx**	**lwarx** reservation set	001	00001	Set by this op	(None) or $\overline{\text{SHD}}$	Set reservation load from cache

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv’n	Snoop Response	Action
001	M E S	**lwarx**	**lwarx** reservation set	001	00001	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
011 010	I	**lwarx**	Single-beat read atomic	01M	11010	Set by this op	(None) or $\overline{\text{SHD}}$	Set reservation load from main memory
011 010	I	**lwarx**	Single-beat read atomic	01M	11010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
011 010	E S	**lwarx**	Single-beat read atomic	01M	11010	Set by this op	(None) or $\overline{\text{SHD}}$	Set the reservation load from main memory
011 010	E S	**lwarx**	Single-beat read atomic	01M	11010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
011 010	M	**lwarx**	Single-beat read atomic	01M	11010	Set by this op	(None) or $\overline{\text{SHD}}$	Paradox—cache should be I set the reservation load from main memory
011 010	M	**lwarx**	Single-beat read atomic	01M	11010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Paradox—cache should be I release the bus retry the operation
100 101	I	**lwarx**	(n/a)	(n/a)	(n/a)	(n/a)	(n/a)	A **lwarx** to a page marked write-through causes a data access exception; therefore no bus transaction results.
101	(n/a)	**lwarx**	(n/a)	(n/a)	(n/a)	(n/a)	(n/a)	A **lwarx** to a page marked write-through causes a data access exception; therefore no bus transaction results.
000	I	Store	RWITM	000	01110	(n/a)	(None) or $\overline{\text{SHD}}$	Load the block of data into cache store to cache mark cache M
000	I	Store	RWITM	000	01110	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
000	S	Store	Kill	000	01100	(n/a)	(None) or $\overline{\text{SHD}}$	Wait for the kill to be successfully presented store to cache mark cache block M
000	S	Store	Kill	000	01100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
000	E	Store	(None)	(n/a)	(n/a)	(n/a)	(n/a)	Store to cache mark cache block M
000	M	Store	(None)	(n/a)	(n/a)	(n/a)	(n/a)	Store to cache

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
001	I	Store	RWITM	001	01110	(n/a)	(None) or $\overline{\text{SHD}}$	Load the block of data into cache mark cache block E store to cache mark cache block M
001	I	Store	RWITM	001	01110	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	S	Store	Kill	001	01100	(n/a)	(None) or $\overline{\text{SHD}}$	Wait for kill to be successfully presented mark cache block E store to cache mark cache block M
001	S	Store	Kill	001	01100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	E	Store	(None)	(n/a)	(n/a)	(n/a)	(n/a)	Store to cache mark cache block M
001	M	Store	(None)	(n/a)	(n/a)	(n/a)	(n/a)	Store to cache
011 010 110 111	I	Store	Write with flush	01M 11M	00010	(n/a)	(None) or $\overline{\text{SHD}}$	Store to main memory
011 010 110 111	I	Store	Write with flush	01M 11M	00010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
011 010 110 111	E S	Store	Write with flush	01M 11M	00010	(n/a)	(None) or $\overline{\text{SHD}}$	Paradox—cache should be I store to main memory
011 010 110 111	E S	Store	Write with flush	01M 11M	00010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Paradox—cache should be I release the bus retry the operation
011 010 110 111	M	Store	Write with flush	01M 11M	00010	(n/a)	(None) or $\overline{\text{SHD}}$	Paradox—cache should be I store to main memory
011 010 110 111	M	Store	Write with flush	01M 11M	00010	(n/a)	ARTRY or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Paradox—cache should be I release the bus retry the operation
100	I	Store	Write with flush	100	00010	(n/a)	(None) or $\overline{\text{SHD}}$	Store to main memory
100	M E S I	Store	Write with flush	100	00010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
100	M E S	Store	Write with flush	100	00010	(n/a)	(None) or $\overline{\text{SHD}}$	Store to cache store to main memory
101	I	Store	Write with flush	101	00010	(n/a)	(None) or $\overline{\text{SHD}}$	Write to main memory (note: no reload on a store miss)
101	M E S I	Store	Write with flush	101	00010	(n/a)	$\overline{\text{ARTRY}}$ or ARTRY&$\overline{\text{SHD}}$	Release the bus retry the operation
101	M E S	Store	Write with flush	101	00010	(n/a)	(None) or $\overline{\text{SHD}}$	Store to cache store to main memory
000	S I	**stwcx.**	(None)	(n/a)	(n/a)	None	(n/a)	Update condition register
000	I	**stwcx.**	RWITM atomic	000	11110	Yes (and reset)	(None) or $\overline{\text{SHD}}$	Load the block of data into cache release the reservation update the condition register store to cache mark cache M
000	I	**stwcx.**	RWITM atomic	000	11110	Yes	$\overline{\text{ARTRY}}$ or ARTRY&$\overline{\text{SHD}}$	Release the bus retry the operation
000	S	**stwcx.**	Kill	000	01100	Yes (and reset)	(None) or $\overline{\text{SHD}}$	Wait for the kill to be successfully presented release reservation update condition register store to cache mark cache block M
000	S	**stwcx.**	Kill	000	01100	Yes	$\overline{\text{ARTRY}}$ or ARTRY&$\overline{\text{SHD}}$	Release the bus retry the operation
000	M E	**stwcx.**	(None)	(n/a)	(n/a)	None	(n/a)	Update condition register
000	E	**stwcx.**	(None)	(n/a)	(n/a)	Yes (and reset)	(n/a)	Release reservation update condition register store to cache mark cache block M
000	M E	**stwcx.**	(None)	(n/a)	(n/a)	Yes (and reset)	(n/a)	(n/a)
000	M	**stwcx.**	(None)	(n/a)	(n/a)	Yes (and reset)	(n/a)	Release reservation update condition register store to cache
001	S I	**stwcx.**	(None)	(n/a)	(n/a)	None	(n/a)	Update condition register
001	I	**stwcx.**	RWITM atomic	001	11110	Yes (and reset)	(None) or $\overline{\text{SHD}}$	Load the block of data into cache release the reservation update the condition register store to cache mark cache M
001	I	**stwcx**	RWITM atomic	001	11110	Yes	$\overline{\text{ARTRY}}$ or ARTRY&$\overline{\text{SHD}}$	Release the bus retry the operation

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
001	S	**stwcx.**	Kill	001	01100	Yes (and reset)	(None) or $\overline{\text{SHD}}$	Release reservation update condition register mark cache block E store to cache mark cache block M
001	S	**stwcx.**	Kill	001	01100	Yes	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	E	**stwcx.**	(None)	(n/a)	(n/a)	None	(n/a)	Update condition register
001	M E	**stwcx.**	(None)	(n/a)	(n/a)	Yes (and reset)	(n/a)	Release reservation update condition register store to cache mark cache block M
001	M E	**stwcx.**	(None)	(n/a)	(n/a)	Yes	(n/a)	(n/a)
001	M	**stwcx.**	(None)	(n/a)	(n/a)	Yes (and reset)	(n/a)	Release reservation update condition register store to cache
011 010	I	**stwcx.**	(None)	(n/a)	(n/a)	None	(n/a)	Update condition register
011 010	I	**stwcx.**	Write with flush atomic	01M	10010	Yes (and reset)	(None) or $\overline{\text{SHD}}$	Release reservation update condition register store to main memory
011 010	I	**stwcx.**	Write with flush atomic	01M	10010	Yes	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
011 010	M E S	**stwcx.**	(None)	(n/a)	(n/a)	None	(n/a)	Paradox**—**cache should be I update condition register
011 010	M E S	**stwcx.**	Write with flush atomic	01M	10010	Yes (and reset)	(None) or $\overline{\text{SHD}}$	Paradox—cache should be I check/release reservation update condition register store to main memory
011 010	M E S	**stwcx.**	Write with flush atomic	01M	10010	Yes	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Paradox**—**cache should be I release the bus retry the operation
011 010	M	**stwcx.**	(n/a)	(n/a)	(n/a)	None	(n/a)	(n/a)
100 101 11X	(n/a)	**stwcx.**	(n/a)	(n/a)	(n/a)	(n/a)	(n/a)	A **stwcx.** to a page marked write-though causes a data access exception; therefore, no bus transaction results.

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
100 101 11X	(n/a)	**stwcx.**	(n/a)	(n/a)	(n/a)	Yes	(n/a)	An **stwcx.** to a page marked write-though causes a data access exception; therefore, no bus transaction results.
000	I	**dcbt**	Read	000	01010	(n/a)	(None)	Load the block of data into cache mark the cache E
000	I	**dcbt**	Read	000	01010	(n/a)	$\overline{\text{SHD}}$	Load the block of data into cache mark the cache S
000	I	**dcbt**	Read	000	01010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
000	M E S	**dcbt**	(None)	(n/a)	(n/a)	(n/a)	(n/a)	No-op
001	I	**dcbt**	Read	001	01010	(n/a)	(None)	Load the block of data into cache mark the cache E
001	I	**dcbt**	Read	001	01010	(n/a)	$\overline{\text{SHD}}$	Load the block of data into cache mark the cache S
001	I	**dcbt**	Read	001	01010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	M E S	**dcbt**	(None)	(n/a)	(n/a)	(n/a)	(n/a)	No-op
011 010 110 111	I	**dcbt**	(None)	01M 11M	(n/a)	(n/a)	(n/a)	No-op
011 010 110 111	E S	**dcbt**	(None)	(n/a)	(n/a)	(n/a)	(n/a)	No-op
011 010 110 111	M	**dcbt**	(None)	(n/a)	(n/a)	None	(n/a)	No-op
011 010 110 111	M	**dcbt**	(n/a)	(n/a)	(n/a)	None	(n/a)	(n/a)
100	I	**dcbt**	Read	100	01010	(n/a)	(None)	Load the block of data into cache mark the cache E
100	I	**dcbt**	Read	100	01010	(n/a)	$\overline{\text{SHD}}$	Load the block of data into cache mark the cache S
100	I	**dcbt**	Read	100	01010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
100	M E S	**dcbt**	(None)	(n/a)	(n/a)	(n/a)	(n/a)	No-op
101	I	**dcbt**	Read	101	01010	(n/a)	(None)	Load the block of data into cache mark the cache E
101	I	**dcbt**	Read	101	01010	(n/a)	$\overline{\text{SHD}}$	Load the block of data into cache mark the cache S
101	I	**dcbt**	Read	101	01010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
101	M E S	**dcbt**	(None)	(n/a)	(n/a)	(n/a)	(n/a)	No-op
000	I	**dcbtst**	Read	000	01010	(n/a)	(None)	Load the block of data into cache mark the cache E
000	I	**dcbtst**	Read	000	01010	(n/a)	$\overline{\text{SHD}}$	Load the block of data into cache mark the cache S
000	I	**dcbtst**	Read	000	01010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
000	S	**dcbtst**	(None)	(n/a)	(n/a)	(n/a)	(n/a)	No-op
000	M E	**dcbtst**	(None)	000	(n/a)	(n/a)	(n/a)	No-op
001	I	**dcbtst**	Read	001	01010	(n/a)	(None)	Load the block of data into cache mark the cache E
001	I	**dcbtst**	Read	001	01010	(n/a)	$\overline{\text{SHD}}$	Load the block of data into cache mark the cache S
001	I	**dcbtst**	Read	001	01010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	M E S	**dcbtst**	(None)	(n/a)	(n/a)	(n/a)	(n/a)	No-op
011 010 110 111	I	**dcbtst**	(None)	01M 11M	(n/a)	(n/a)	(n/a)	No-op
011 010 110 111	E S	**dcbtst**	(None)	(n/a)	(n/a)	(n/a)	(n/a)	No-op
011 010 110 111	M	**dcbtst**	(None)	(n/a)	(n/a)	None	(n/a)	No-op
011 010 110 111	M	**dcbtst**	(n/a)	(n/a)	(n/a)	None	(n/a)	(n/a)

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
100	I	**dcbtst**	Read	100	01010	(n/a)	(None)	Load the block of data into cache mark cache E
100	I	**dcbtst**	Read	100	01010	(n/a)	$\overline{\text{SHD}}$	Load the block of data into cache mark cache as block S
100	I	**dcbtst**	Read	100	01010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
100	M E S	**dcbtst**	(None)	(n/a)	(n/a)	(n/a)	(n/a)	No-op
101	I	**dcbtst**	Read	101	01010	(n/a)	(None)	Load the block of data into cache mark cache block E
101	I	**dcbtst**	Read	101	01010	(n/a)	$\overline{\text{SHD}}$	Load the block of data into cache mark cache block S
101	I	**dcbtst**	Read	101	01010	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
101	S E	**dcbtst**	(None)	(n/a)	(n/a)	(n/a)	(n/a)	No-op
101	M	**dcbtst**	(None)	(n/a)	(n/a)	(n/a)	(n/a)	No-op
000	I	**dcbz**	Kill	000	01100	(n/a)	(None) or $\overline{\text{SHD}}$	Establish the block in data cache without fetching the block from main memory clear all bytes mark cache block M
000	S I	**dcbz**	Kill	000	01100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
000	S	**dcbz**	Kill	000	01100	(n/a)	(None) or $\overline{\text{SHD}}$	Clear all bytes in the block mark cache block M
000	E	**dcbz**	(None)	000	(n/a)	(n/a)	(n/a)	Clear all bytes in the block mark cache block M
000	M	**dcbz**	(None)	(n/a)	(n/a)	(n/a)	(n/a)	Write zeros to all bytes in the cache block
001	I	**dcbz**	Kill	001	01100	(n/a)	(None) or $\overline{\text{SHD}}$	Establish the block in data cache without fetching the block from main memory clear all bytes mark cache block M
001	I	**dcbz**	Kill	001	01100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	S	**dcbz**	Kill	001	01100	(n/a)	(None) or $\overline{\text{SHD}}$	Mark cache block E set all bytes of the block to zero mark the cache block M
001	S	**dcbz**	Kill	001	01100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus Retry the operation

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
001	E	**dcbz**	(None)	(n/a)	(n/a)	(n/a)	(n/a)	Write zeros to all bytes in the Cache block mark cache block M
001	M	**dcbz**	(None)	(n/a)	(n/a)	(n/a)	(n/a)	Write zeros to all bytes in the cache block
010 011 110 111 100 101	M E S I	**dcbz**	(n/a)	(n/a)	(n/a)	(n/a)	(n/a)	A **dcbz** to a page marked cache inhibited or write-through causes an alignment exception; therefore this transaction does not occur on the bus
000	E S I	**dcbst**	Clean	000	00000	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
000	E S I	**dcbst**	Clean	000	00000	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
000	M	**dcbst**	Write with kill	100	00110	(n/a)	(None) or $\overline{\text{SHD}}$	Write the block to main memory mark cache block E
000	M	**dcbst**	Write with kill	100	00110	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	E S I	**dcbst**	Clean	001	00000	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
001	E S I	**dcbst**	Clean	001	00000	(n/a)	ARTRY or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	M	**dcbst**	Write with kill	100	00110	(n/a)	(None) or $\overline{\text{SHD}}$	Write all bytes in the cache block to main memory mark cache block E
001	M	**dcbst**	Write with kill	100	00110	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
011 010 110 111	E S I	**dcbst**	Clean	W1M	00000	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
011 010 110 111	I	**dcbst**	Clean	W1M	00000	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
011 010 110 111	M	**dcbst**	Write with kill	100	00110	(n/a)	(None) or $\overline{\text{SHD}}$	Write all bytes in the cache block to main memory Mark cache block E
011 010 110 111	M	**dcbst**	Write with kill	100	00110	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
100	E S I	**dcbst**	Clean	100	00000	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
100	E S I	**dcbst**	Clean	100	00000	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
100	M	**dcbst**	Write with kill	100	00110	(n/a)	(None) or $\overline{\text{SHD}}$	Write the block back to memory mark cache block E
100	M	**dcbst**	Write with kill	100	00110	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
101	E S I	**dcbst**	Clean	101	00000	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
101	E S I	**dcbst**	Clean	101	00000	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
101	M	**dcbst**	Write with kill	100	00110	(n/a)	(None) or $\overline{\text{SHD}}$	Write the block back to memory mark cache block E
101	M	**dcbst**	Write with kill	100	00110	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
000	I	**dcbf**	Flush	000	00100	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
000	I	**dcbf**	Flush	000	00100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
000	E S	**dcbf**	Flush	000	00100	(n/a)	(None) or $\overline{\text{SHD}}$	Mark cache block I
000	E S	**dcbf**	Flush	000	00100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
000	M	**dcbf**	Write with kill	100	00110	(n/a)	(None) or $\overline{\text{SHD}}$	Write the block of data back to main memory mark the cache block I
000	M	**dcbf**	Write with kill	100	00110	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	I	**dcbf**	Flush	001	00100	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
001	E S	**dcbf**	Flush	001	00100	(n/a)	(None) or $\overline{\text{SHD}}$	Mark cache block I
001	E S I	**dcbf**	Flush	001	00100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	M	**dcbf**	Write with kill	100	00110	(n/a)	(None) or $\overline{\text{SHD}}$	Write all bytes in the cache block to main memory mark cache block I
001	M	**dcbf**	Write with kill	100	00110	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
011 010 110 111	I	**dcbf**	Flush	W1M	00100	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
011 010 110 111	I	**dcbf**	Flush	W1M	00100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
011 010 110 111	E S	**dcbf**	Flush	W1M	00100	(n/a)	(None) or $\overline{\text{SHD}}$	Mark cache block I
011 010 110 111	E S	**dcbf**	Flush	W1M	00100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Retry the operation
011 010 110 111	M	**dcbf**	Write with kill	100	00110	(n/a)	(None) or $\overline{\text{SHD}}$	Flush the block mark cache block I
011 010 110 111	M	**dcbf**	Write with kill	100	00110	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
100	I	**dcbf**	Flush	100	00100	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
100	E S	**dcbf**	Flush	100	00100	(n/a)	(None) or $\overline{\text{SHD}}$	Mark cache block I
100	E S I	**dcbf**	Flush	100	00100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
100	M	**dcbf**	Write with kill	100	00110	(n/a)	(None) or $\overline{\text{SHD}}$	Write the block back to memory mark cache block I
100	M	**dcbf**	Write with kill	100	00110	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
101	I	**dcbf**	Flush	101	00100	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
101	E S	**dcbf**	Flush	101	00100	(n/a)	(None) or $\overline{\text{SHD}}$	Mark cache block I
101	E S I	**dcbf**	Flush	101	00100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
101	M	**dcbf**	Write with kill	100	00110	(n/a)	(None) or $\overline{\text{SHD}}$	Flush the block mark cache block I
101	M	**dcbf**	Write with kill	100	00110	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
000	I	**dcbi**	Kill	000	01100	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
000	M E S	**dcbi**	Kill	000	01100	(n/a)	(None) or $\overline{\text{SHD}}$	Mark the cache block I
000	M E S I	**dcbi**	Kill	000	01100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	I	**dcbi**	Kill	001	01100	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
001	I	**dcbi**	Kill	001	01100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	S	**dcbi**	Kill	001	01100	(n/a)	(None) or $\overline{\text{SHD}}$	Mark cache block I
001	S	**dcbi**	Kill	001	01100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	E M	**dcbi**	Kill	001	01100	(n/a)	(None) or $\overline{\text{SHD}}$	Mark cache block I
001	E M	**dcbi**	Kill	001	01100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
011 010 110 111	I	**dcbi**	Kill	W1M	01100	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
011 010 110 111	M E S	**dcbi**	Kill	W1M	01100	(n/a)	(None) or $\overline{\text{SHD}}$	Mark cache block I
011 010 110 111	M E S I	**dcbi**	Kill	W1M	01100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
100	I	**dcbi**	Kill	100	01100	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
100	M E S I	**dcbi**	Kill	100	01100	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
100	M E S	**dcbi**	Kill	100	01100	(n/a)	(None) or $\overline{\text{SHD}}$	Mark cache block I
101	I	**dcbi**	Kill	101	01100	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
101	M E S I	**dcbi**	Kill	101	01100	(n/a)	$\overline{\text{ARTRY}}$ or ARTR&$\overline{\text{SHD}}$	Release the bus retry the operation

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
101	M E S	**dcbi**	Kill	101	01100	(n/a)	(None) or $\overline{\text{SHD}}$	Mark cache block I
000	INV	**icbi**	ICBI	000	01101	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
000	INV	**icbi**	ICBI	000	01101	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
000	VAL	**icbi**	ICBI	000	01101	(n/a)	(None) or $\overline{\text{SHD}}$	Mark icache block INV
000	VAL	**icbi**	ICBI	000	01101	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	INV	**icbi**	ICBI	001	01101	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
001	INV VAL	**icbi**	ICBI	001	01101	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
001	VAL	**icbi**	ICBI	001	01101	(n/a)	(None) or $\overline{\text{SHD}}$	Mark icache block INV
011 010 110 111	INV	**icbi**	ICBI	01M 11M	01101	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
011 010 110 111	INV VAL	**icbi**	ICBI	01M 11M	01101	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
011 010 110 111	VAL	**icbi**	ICBI	01M 11M	01101	(n/a)	(None) or $\overline{\text{SHD}}$	Mark icache block INV
100	INV	**icbi**	ICBI	100	01101	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
100	INV VAL	**icbi**	ICBI	100	01101	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
100	VAL	**icbi**	ICBI	100	01101	(n/a)	(None) or $\overline{\text{SHD}}$	Mark icache block INV
101	INV	**icbi**	ICBI	101	01101	(n/a)	(None) or $\overline{\text{SHD}}$	No-op
101	INV VAL	**icbi**	ICBI	101	01101	(n/a)	$\overline{\text{ARTRY}}$ or $\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Release the bus retry the operation
101	VAL	**icbi**	ICBI	101	01101	(n/a)	(None) or $\overline{\text{SHD}}$	Mark icache block INV

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
(n/a)	(n/a)	**sync**	SYNC	xx1	01000	(n/a)	(None) or $\overline{\mathrm{SHD}}$	The **sync** instruction completed. (Note: This table does not give an accurate representation of what the **sync** instruction does.)
(n/a)	(n/a)	**sync**	SYNC	xx1	01000	(n/a)	$\overline{\mathrm{ARTRY}}$ or $\overline{\mathrm{ARTRY}}$&$\overline{\mathrm{SHD}}$	Release the bus. Retry the operation.
(n/a)	(n/a)	**eieio**	EIEIO	xx1	10000	(n/a)	(None) or $\overline{\mathrm{SHD}}$	The **eieio** instruction has completed. (Note: This table does not give an accurate representation of what the **eieio** instruction does.)
(n/a)	(n/a)	**eieio**	EIEIO	xx1	10000	(n/a)	$\overline{\mathrm{ARTRY}}$ or $\overline{\mathrm{ARTRY}}$&$\overline{\mathrm{SHD}}$	Release the bus. Retry the operation.
(n/a)	(n/a)	**tlbie**	TLB invalidate	xx1	11000	(n/a)	(None) or $\overline{\mathrm{SHD}}$	Hold off any new storage instructions. Wait for the completion of any outstanding storage instructions Invalidate the requested TLB entry (Note: This table does not thoroughly characterize the **tlbie** instruction.)
(n/a)	(n/a)	**tlbie**	TLB invalidate	xx1	11000	(n/a)	$\overline{\mathrm{ARTRY}}$ or $\overline{\mathrm{ARTRY}}$&$\overline{\mathrm{SHD}}$	Release the bus. Retry the operation
		tlbsync	TLB sync	xx1	01001	(n/a)	(None) or $\overline{\mathrm{SHD}}$	The TLB sync instruction has completed. (Note: This table does not thoroughly characterize the **tlbsync** instruction.)
		tlbsync	TLB sync	xx1	01001	(n/a)	$\overline{\mathrm{ARTRY}}$ or $\overline{\mathrm{ARTRY}}$&$\overline{\mathrm{SHD}}$	Release the bus. Retry the operation.
	I		Snoop-kill	xx1	01100	None	(None)	No-op
	I		Snoop-kill	xx1	01100	Yes (and reset)	(None)	Release reservation.
	M E S		Snoop-kill	xx1	01100	None	(None)	Mark cache block I.
	M E S		Snoop-kill	xx1	01100	Yes (and reset)	(None)	Mark cache block I. Release reservation.
	I		Snoop-read	xx1	01010	None	(None)	No-op
	I		Snoop-read	xx1	01010	Yes	$\overline{\mathrm{SHD}}$	No-op

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
	S		Snoop-read	xx1	01010	(n/a)	$\overline{\text{SHD}}$	No-op
	E		Snoop-read	xx1	01010	(n/a)	$\overline{\text{SHD}}$	Mark cache block S.
	M		Snoop-read	x01	01010	(n/a)	$\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Attempt to write cache block back to main memory; if successful, mark cache block S
	M		Snoop-read	x11	01010	(n/a)	$\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Attempt to write cache block back to main memory; If successful, mark cache block S
	I		Snoop-read atomic	xx1	11010	None	(None)	No-op
	I		Snoop-read atomic	xx1	11010	Yes	$\overline{\text{SHD}}$	No-op
	S		Snoop-read atomic	xx1	11010	(n/a)	$\overline{\text{SHD}}$	No-op
	E		Snoop-read atomic	xx1	11010	(n/a)	$\overline{\text{SHD}}$	Mark cache block S
	M		Snoop-read atomic	xx1	11010	(n/a)	$\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Attempt to write cache block back to main memory; if successful, mark cache block S.
	I		Snoop-RWITM	xx1	01110	None	(None)	No-op
	I		Snoop-RWITM	xx1	01110	Yes (and reset)	(None)	Release reservation.
	E S		Snoop-RWITM	xx1	01110	None	(None)	Mark cache block I.
	E S		Snoop-RWITM	xx1	01110	Yes (and reset)	(None)	Mark cache block I. Release reservation.
	M		Snoop-RWITM	xx1	01110	None	$\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Attempt to write cache block back to main memory; if successful, mark cache block I.
	M		Snoop-RWITM	xx1	01110	Yes (and reset)	$\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Attempt to write cache block back to main memory; if successful, mark cache block I, release reservation

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
	I		Snoop-RWITM atomic	xx1	11110	None	(None)	No-op
	I		Snoop-RWITM atomic	xx1	11110	Yes (and reset)	(None)	Release reservation.
	S E		Snoop-RWITM atomic	xx1	11110	None	(None)	Mark cache block I.
	S E		Snoop-RWITM atomic	xx1	11110	Yes (and reset)	(None)	Mark cache block I. Release reservation.
	M		Snoop-RWITM atomic	xx1	11110	None	$\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Attempt to write cache block back to main memory; if successful, mark cache block I.
	M		Snoop-RWITM atomic	xx1	11110	Yes (and reset)	$\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Attempt to write cache block back to main memory; if successful, mark cache block I, release reservation.
	I		Snoop-flush	xx1	00100	None	(None)	No-op
	I		Snoop-flush	xx1	00100	Yes	(None)	No-op
	S E		Snoop-flush	xx1	00100	(n/a)	(None)	Mark cache block I.
	M		Snoop-flush	xx1	00100	(n/a)	$\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Attempt to write cache block back to main memory; if successful: mark cache block I.
	E S I		Snoop-clean	xx1	00000	(n/a)	(None)	No-op
	M		Snoop-clean	xx1	00000	(n/a)	$\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Attempt to write cache block back to main memory; if successful, mark cache block E.
	I		Snoop-write with flush	xx1	00010	None	(None)	No-op
	I		Snoop-write with flush	xx1	00010	Yes (and reset)	(None)	Release reservation.
	S		Snoop-write with flush	xx1	00010	None	(None)	Mark cache block I.

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
	S		Snoop-write with flush	xx1	00010	Yes (and reset)	(None)	Mark cache block I. Release reservation.
	E		Snoop-write with flush	xx1	00010	None	(None)	Paradox—no one else should be writing if this cache is E. Mark cache block I
	E		Snoop-write with flush	xx1	00010	Yes (and reset)	(None)	Paradox—no one else should be writing if this cache is E. Mark cache block I. Release reservation.
	M		Snoop-write with flush	xx1	00010	None	$\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Paradox—no one else should be writing if this cache is M. Attempt to write cache block back to main memory; if successful, mark cache block I
	M		Snoop-write with flush	xx1	00010	Yes (and reset)	$\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Paradox—no one else should be writing if this cache is M. Attempt to write cache block back to main memory; if successful, mark cache block I, release reservation
	I		Snoop-write with kill	xx1	00110	None	(None)	No-op
	I		Snoop-write with kill	xx1	00110	Yes (and reset)	(None)	Release reservation.
	S		Snoop-write with kill	xx1	00110	None	(None)	Mark cache block I.
	S		Snoop-write with kill	xx1	00110	Yes (and reset)	(None)	Mark cache block I. Release reservation.
	E		Snoop-write with kill	xx1	00110	None	(None)	Paradox—no one else should be writing if this cache is E. Mark cache block I.
	E		Snoop-write with kill	xx1	00110	Yes (and reset)	(None)	Paradox—no one else should be writing if this cache is E. Mark cache block I. Release reservation.
	M		Snoop-write with kill	xx1	00110	None	(None)	Paradox—no one else should be writing if this cache is M. Mark cache block I.

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
	M		Snoop-write with kill	xx1	00110	Yes (and reset)	(None)	Paradox—no one else should be writing if this cache is M. Mark cache block I. Release reservation.
	I		Snoop-write with flush atomic	xx1	10010	None	(None)	No-op
	I		Snoop-write with flush atomic	xx1	10010	Yes (and reset)	(None)	Release reservation.
	S		Snoop-write with flush atomic	xx1	10010	None	(None)	Mark cache block I.
	S		Snoop-write with flush atomic	xx1	10010	Yes (and reset)	(None)	Mark cache block I. Release reservation.
	E		Snoop-write with flush atomic	xx1	10010	None	(None)	Paradox—no one else should be writing if this cache is E. Mark cache block I.
	E		Snoop-write with flush atomic	xx1	10010	Yes (and reset)	(None)	Paradox—no one else should be writing if this cache is E. Mark cache block I, release reservation.
	M		Snoop-write with flush atomic	xx1	10010	None	$\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Paradox—no one else should be writing if this cache is M. Attempt to write block back to main memory; if successful, mark cache block I
	M		Snoop-write with flush atomic	xx1	10010	Yes (and reset)	$\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Paradox—no one else should be writing if this cache is M. Attempt to write block back to main memory; if successful: mark cache block I, release reservation.
	(n/a)		Snoop-TLB invalidate	xx1	11000	(n/a)	(None)	Respond with (none) when the TLB has been invalidated.

Table 3-6. Cache Actions (Continued)

Cache WIM	MESI State	Action	Bus Operation	Bus WIM	TT0-4	Rsv'n	Snoop Response	Action
	(n/a)		Snoop-TLB invalidate	xx1	11000	(n/a)	(None) but $\overline{\text{ARTRY}}$ is activated on the bus from another processor	Do not perform the TLB invalidate—this is to prevent a deadlock condition from occurring.
	(n/a)		Snoop-TLB invalidate	xx1	11000	(n/a)	$\overline{\text{ARTRY}}$	Respond with retry until the TLB has been invalidated.
	(n/a)		Snoop-SYNC	xx1	01000	(n/a)	(None)	If no TLB invalidates are pending, no-op.
	(n/a)		Snoop-SYNC	xx1	01000	(n/a)	$\overline{\text{ARTRY}}$	If a TLB invalidate is pending, respond with retry.
	(n/a)		Snoop-TLBSYNC	xx1	01001	(n/a)	(None)	If no TLB invalidates are pending, no-op.
	(n/a)		Snoop-TLBSYNC	xx1	01001	(n/a)	$\overline{\text{ARTRY}}$	If a TLB invalidate is pending, respond with retry.
	(n/a)		Snoop-EIEIO	xx1	10000	(n/a)	(None)	No-op
	(n/a)		Snoop-EIEIO	xx1	10000	(n/a)	$\overline{\text{ARTRY}}$	No-op
	I		Snoop-ICBI	xx1	01101	(n/a)	(None)	No-op
	VAL		Snoop-ICBI	xx1	01101	(n/a)	(None)	Invalidate entry in icache
	I		Snoop-RWNITC	xx1	01011	None	(None)	No-op
	I		Snoop-RWNITC	xx1	01011	Yes	$\overline{\text{SHD}}$	No-op
	E S		Snoop-RWNITC	xx1	01011	(n/a)	$\overline{\text{SHD}}$	No-op
	M		Snoop-RWNITC	xx1	01011	(n/a)	$\overline{\text{ARTRY}}$&$\overline{\text{SHD}}$	Attempt to write cache block back to main memory; if successful, mark cache block E.

3.11 Access to Direct-Store Segments

The 604 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 604 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 604 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 604 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.
Handling	Exception handling is performed by the software linked to the appropriate vector offset. Exception handling is begun in supervisor-level (referred to as privileged state in the architecture specification).

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 4.5.7, “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 604 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 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 (604-specific) Performance monitoring exception (604-specific)
Synchronous/precise	Instruction-caused exceptions
Synchronous/imprecise	Instruction-caused imprecise exceptions (Floating-point imprecise exceptions)

Exceptions implemented in the 604, 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 604 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 604 a machine check exception is signaled by the assertion of a qualified $\overline{\text{TEA}}$ indication on the 604 bus, or 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. The assertion of the $\overline{\text{TEA}}$ signal is determined by read, write, and instruction fetch operations initiated by the processor; however, it is expected that the $\overline{\text{TEA}}$ 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. 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. (Note that, physical address is referred to as the real address in the architecture specification.) 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.

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

Exception Type	Vector Offset (hex)	Causing Conditions
DSI	00300	A DSI exception occurs when a data memory access cannot be performed for any of the reasons described in Section 4.5.3, “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 4.5.4, “ISI Exception (0x00400).”
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 604 stops dispatching instructions and waits for all dispatched instructions to complete. Any exceptions associated with dispatched instructions are taken before the interrupt is taken.
Alignment	00600	An alignment exception may occur when the processor cannot perform a memory access for reasons described in Section 4.5.6, “Alignment Exception (0x00600).” Note that the PowerPC architecture defines a wider range of conditions that may cause an alignment exception than required in the 604. In these cases, the 604 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 3 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 4.5.7, “Program Exception (0x00700).”
Floating-point unavailable	00800	Defined by the PowerPC architecture, but not implemented in the 604.
Decrementer	00900	The decrementer interrupt exception is taken if the interrupt is enabled and the exception is pending. The exception is created when the most significant bit changes from 0 to 1. If it is not enabled, the exception remains pending until it is taken **.**

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

Exception Type	Vector Offset (hex)	Causing Conditions
Reserved	00A00	Reserved for implementation-specific exceptions. For example, the 601 uses this vector offset for direct-store exceptions.
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 604, is defined by the PowerPC architecture but is optional. A trace exception occurs if either MSR[SE] = 1 and any instruction (except **rfi**) successfully completed or MSR[BE] = 1 and a branch instruction is completed.
Performance monitoring interrupt	00F00	The performance monitoring interrupt is a 604-specific exception and is used with the 604 performance monitor, described in Section 4.5.13, "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 by 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 EE bit and cannot be taken until the EE bit is set.
Reserved	01000–012FF	Reserved for implementation-specific exceptions not implemented on the 604.
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 (bit 30) is set to 1.
System management interrupt	01400	A system management interrupt 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 604.

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 6, “Exceptions,” in *The Programming Environments Manual*.

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 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.

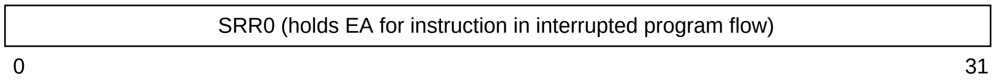


Figure 4-1. Machine Status Save/Restore Register 0

SRR0 is 32 bits wide in 32-bit implementations.

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 **rfi** is executed. SRR1 is shown in Figure 4-2.

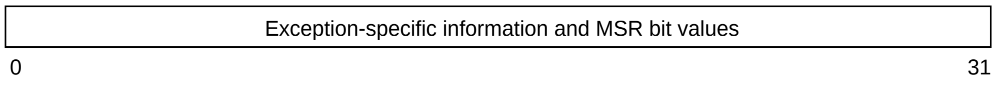


Figure 4-2. Machine Status Save/Restore Register 1

Typically, when an exception occurs, bits 2–4 and 10–12 of SRR1 are loaded with exception-specific information and bits 5–9, and 16–31 of MSR are placed into the corresponding bit positions of 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 604 and in other 32-bit PowerPC implementations, the MSR is 32 bits wide as shown in Figure 4-3.

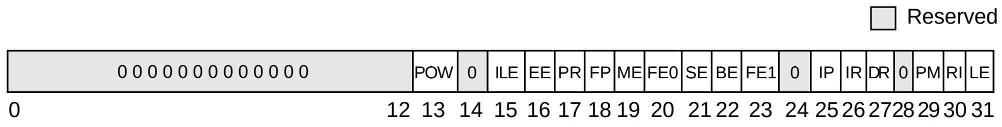


Figure 4-3. Machine State Register (MSR)

The MSR bits are defined in Table 4-3. Full function reserved bits are saved in SRR1 when an exception occurs; partial function reserved bits are not saved.

Table 4-3. MSR Bit Settings

Bit(s)	Name	Description
0	—	Reserved. Full Function.
1–4	—	Reserved. Partial function.
5–9	—	Reserved. Full function.
10–12	—	Reserved. Partial function.
13	POW	Power management enable 0 Power management disabled (normal operation mode). 1 Power management enabled (reduced power mode). Note that power management functions are implementation-dependent.
14	—	Reserved—Implementation-specific
15	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.
16	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.
17	PR	Privilege level 0 The processor can execute both user- and supervisor-level instructions. 1 The processor can only execute user-level instructions.

Table 4-3. MSR Bit Settings (Continued)

Bit(s)	Name	Description
18	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, and can take floating-point enabled exception type program exceptions.
19	ME	Machine check enable 0 Machine check exceptions are disabled. 1 Machine check exceptions are enabled.
20	FE0	IEEE floating-point exception mode 0 (See Table 4-4).
21	SE	Single-step trace enable 0 The processor executes instructions normally. 1 The processor generates a single-step trace exception upon the successful execution of the next instruction (unless that instruction is an **rfi** instruction). Successful execution means that the instruction caused no other exception.
22	BE	Branch trace enable 0 The processor executes branch instructions normally. 1 The processor generates a branch type trace exception upon the successful execution of a branch instruction.
23	FE1	IEEE floating-point exception mode 1 (See Table 4-4).
24	—	Reserved. This bit corresponds to the AL bit of the POWER architecture.
25	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. 0 Exceptions are vectored to the physical address 0x000*n_nnnn*. 1 Exceptions are vectored to the physical address 0xFFF*n_nnnn*.
26	IR	Instruction address translation 0 Instruction address translation is disabled. 1 Instruction address translation is enabled. For more information see Chapter 5, “Memory Management.”
27	DR	Data address translation 0 Data address translation is disabled. 1 Data address translation is enabled. For more information see Chapter 5, “Memory Management.”
28	—	Reserved, full function.
29	PM	Performance monitor marked mode 0 Process is not a marked process. 1 Process is a marked process. This bit is specific to the 604, and is defined as reserved by the PowerPC architecture. For more information about the performance monitor, see Section 4.5.13, “Performance Monitoring Interrupt (0x00F00).”

Table 4-3. MSR Bit Settings (Continued)

Bit(s)	Name	Description
30	RI	Indicates whether system reset or machine check exception is recoverable. 0 Exception is not recoverable. 1 Exception is recoverable. The RI bit indicates whether from the perspective of the processor, it is safe to continue (that is, processor state data such as that saved to SRR0 is valid), but it does not guarantee that the interrupted process is recoverable.
31	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 604 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 imprecise nonrecoverable
1	0	Floating-point imprecise recoverable. In the 604, this bit setting causes the 604 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.

4.3.1 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 and decrementer 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.

4.3.2 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 1–4 and 10–15 of SRR1 are loaded with information specific to the exception type.
3. Bits 5–9 and 16–31 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 0x000*n_nnnn*. If IP is set, exceptions are vectored to the physical address 0xFFF*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 4.5.2, “Machine Check Exception (0x00200).”

4.3.3 Setting MSR[RI]

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

- In the machine check and system reset exceptions—If SRR1[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 **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.

4.3.4 Returning from an Exception Handler

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

4.4 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 4.3.3, “Setting MSR[RI].”

4.5 Exception Definitions

Table 4-5 shows all the types of exceptions that can occur with the 604 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														
	POW	ILE	EE	PR	FP	ME	FE0	SE	BE	FE1	IP	IR	DR	RI	LE
System reset	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
Machine check	0	—	0	0	0	0	0	0	0	0	—	0	0	0	ILE
DSI	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
ISI	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
External	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
Alignment	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
Program	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
Floating-point unavailable	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
Decrementer	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
System call	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
Trace exception	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
System management	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
Performance monitor	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 0x000*n_nnnn* (where *nnnnn* is the vector offset); if IP is set, exceptions are vectored to the physical address 0xFFF*n_nnnn*. Table 4-2 shows the exception vector offset of the first instruction of the exception handler routine for each exception type.

4.5.1 System Reset Exception (0x00100)

The 604 implements the system reset exception as defined in the PowerPC architecture (OEA). The system reset exception is a nonmaskable, asynchronous exception signaled to the processor through the assertion of system-defined signals. In the 604, the exception is signaled by the assertion of either the $\overline{\text{SRESET}}$ or $\overline{\text{HRESET}}$ inputs, described more fully in Chapter 7, “Signal Descriptions.”.

Table 4-6. System Reset Exception—Register Settings

Register	Setting Description
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 Loaded with equivalent bits from the MSR 1–4 Cleared 5–9 Loaded with equivalent bits from the MSR 10–15 Cleared 16–31 Loaded with equivalent bits of the MSR Note that if the processor state is corrupted to the extent that execution cannot resume reliably, the MSR[RI] bit (SRR1[30]) is cleared.
MSR	POW 0 ILE --- EE 0 PR 0 FP 0 ME --- FE0 0 SE 0 BE 0 FE1 0 IP — IR 0 DR 0 RI 0 LE Set to value of ILE

The SRESET input provides a “warm” reset capability. This input is used to avoid causing the 604 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].

4.5.2 Machine Check Exception (0x00200)

The 604 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, after receiving a qualified transfer error acknowledge ($\overline{\text{TEA}}$) indication on the 604 bus, 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.

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 pin
1	Enable cache parity checking

Table 4-7. Machine Check Enable Bits

HID0 Bit	Description
2	Enable machine check on address bus parity error.
3	Enable machine check on data bus parity error.

A $\overline{\text{TEA}}$ indication on the bus can result from any load or store operation initiated by the processor. In general, the $\overline{\text{TEA}}$ signal is expected to be used by a memory controller to indicate that a memory parity error or an uncorrectable memory ECC error has occurred. Note that the resulting machine check exception is imprecise and unordered with respect to the instruction that originated the bus operation.

If the MSR[ME] bit and the appropriate bits in HID0 are set, the exception is recognized and handled; otherwise, the processor generates an internal checkstop condition. 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 4.5.2.1, “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 4.5.2.2, “Checkstop State (MSR[ME] = 0).”

4.5.2.1 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	0–9 Cleared 10 Set when a data cache parity error is detected, otherwise zero 11 Set when a instruction cache parity error is detected, otherwise zero 12 Set when Machine Check Pin ($\overline{\text{MCP}}$) is asserted, otherwise zero 13 Set when $\overline{\text{TEA}}$ pin is asserted, otherwise zero 14 Set when a data bus parity error is detected, otherwise zero 15 Set when an address bus parity error is detected, otherwise zero 16–29 MSR(16–29) 30 Zero 31 MSR(31)
MSR	POW 0 ILE --- EE 0 PR 0 FP 0 ME* 0 FE0 0 SE 0 BE 0 FE1 0 IP — IR 0 DR 0 RI 0 LE Set to value of ILE

* 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].

4.5.2.2 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.

A machine check exception may result from referencing a nonexistent physical address, either directly (with MSR[DR] = 0), or through an invalid translation. On such a system, for example, execution of a Data Cache Block Set to Zero (**dcbz**) instruction that introduces a block into the cache associated with a nonexistent physical address may delay the machine check exception until an attempt is made to store that block to main memory.

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

4.5.3 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). Note that there are some conditions for which the PowerPC architectures allow implementations to optionally take a DSI exception. Table 4-9 lists conditions defined by the architecture that optionally may cause a DSI exception.

Table 4-9. Other MMU Exception Conditions

Condition	Description	DSISR
lwarx or **stwcx.** with W = 1	Reservation instruction to write-through segment or block	DSISR[5] = 1
lwarx, **stwcx.**, **eciwx**, or **ecowx** instruction to direct-store segment	Reservation instruction or external control instruction when SR[T] = 1 or STE[T] = 1	DSISR[5] = 1
Load or store that results in a direct-store error	Direct-store interface protocol signalled with an error condition	DSISR[0] = 1
eciwx or **ecowx** attempted when external control facility disabled	**eciwx** or **ecowx** attempted with EAR[E] = 0	DSISR[11] = 1

4.5.4 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). In addition, an instruction fetch from a no-execute segment results in an ISI exception.

When an ISI exception is taken, instruction execution resumes at offset 0x00400 from the physical base address indicated by MSR[IP].

4.5.5 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 604 takes the external interrupt exception. If the external interrupt signal is negated early, recognition of the interrupt request is not guaranteed. After the 604 begins execution of the external interrupt handler, the system can safely negate the $\overline{\text{INT}}$. When the signal is detected, the 604 stops dispatching instructions and waits for all pending instructions to complete. This allows any instructions in progress that need to take an exception to do so before the external interrupt is taken. After all instructions have cleared, the 604 takes the external interrupt exception as defined in the PowerPC architecture (OEA).

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].

4.5.6 Alignment Exception (0x00600)

The 604 implements the alignment exception as defined by the PowerPC architecture (OEA). An alignment exception is initiated when any of the following conditions are met:

- A floating-point load or store, **lmw**, **stmw**, **lwarx**, or **stwcx.** instruction is not word-aligned.
- If a floating-point number is not word-aligned. The 604 provides hardware support for misaligned storage accesses for other memory access instructions. If a misaligned memory access crosses a 4-Kbyte page boundary within a memory segment, an exception may occur when the boundary is crossed (that is, there is a protection violation on an attempt to access the new page). In these cases, a DSI exception occurs and the instruction may complete partially.
- Some types of misaligned memory accesses are slower than aligned accesses. Accesses that cross a word boundary (and double-precision values aligned on a double-word boundary) are broken into multiple accesses by the LSU. More dramatically, any noncacheable memory access that crosses a double-word boundary requires multiple external bus tenures.
- Operations that cross a word boundary (and operations involving double-precision values aligned on a double-word boundary) require two accesses, which are translated separately. If either translation creates a DSI exception condition, that exception is signaled.
- If the T-bit settings are not the same for both portions of a misaligned memory access, (which is considered to be a programming error), the 604 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 604 requires a direct-store access reply from the device. If two translations cross memory locations that are T = 0 into T = 1, a DSI exception is signaled.
- A **dcbz** instruction references a page that is marked either cache-inhibited or write-through or has executed when the 604 data cache is locked or disabled. Note that this condition may not cause an alignment exception in other PowerPC processors.
- An access is not naturally aligned in little-endian mode.
- An **ecowx** or **eciwx** is not word-aligned.
- A **lmw**, **stmw**, **lswi**, **lswx**, **stswi,** or **stswx** instruction is issued in little-endian mode.

4.5.7 Program Exception (0x00700)

The 604 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 604 invokes the system illegal instruction program exception when it detects any instruction from the illegal instruction class.

The 604 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 604, 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 604, 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].

Note that the 604 supports one of the two floating-point imprecise modes supported by the PowerPC architecture. The three modes supported by the 604 are described as follows:

- Ignore exceptions mode (MSR[FE0] = MSR[FE1] = 0)—In ignore exceptions 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. IEEE floating-point exception conditions (as defined in the PowerPC architecture) do not cause any exceptions.
- Precise exceptions mode (MSR[FE0] = 1; MSR[FE1] = x)—In this mode, a floating point instruction that causes a floating-point exception brings the machine to a precise state. In doing so, the 604 sequencer unit can detect floating-point exception conditions and take floating-point exceptions as defined by the PowerPC architecture. Note that the imprecise recoverable mode supported by the PowerPC architecture (MSR[FE0] = 1; MSR[FE1] = 0) is implemented identically to precise exceptions mode in the 604.
- Imprecise nonrecoverable mode (MSR[FE0] = 0; MSR[FE1] = 1)—In this mode, floating-point exception conditions cause a floating-point exception to be taken, SRR0 may point to some instruction following the instruction that caused the exception.

Register settings for this exception are described in Chapter 6, “Exceptions,” in *The Programming Environments Manual.*

4.5.8 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].

4.5.9 Decrementer Exception (0x00900)

The decrementer exception is implemented in the 604 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 604, the decrementer register is decremented at one fourth 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].

4.5.10 System Call Exception (0x00C00)

A system call exception occurs when a System Call (**sc**) instruction is executed. In the 604, 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].

4.5.11 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 604 are shown in Table 4-10.

Table 4-10. Trace Exception—SRR1 Settings

Register	Setting
SRR1	0–2 010 3 Set for a load instruction, otherwise cleared 4 Set for a store instruction, otherwise cleared 5–9 Cleared 10 Set for **lswx** or **stswx**, otherwise cleared 11 Set for **mtspr** to SDR1, EAR, HID0, PIR, IBATs, DBATs, SRs 12 Set for taken branch, otherwise cleared 13–15 Cleared 16–31 MSR(16–31).

When a trace exception is taken, instruction execution resumes as offset 0x00D00 from the base address indicated by MSR[IP].

4.5.12 Floating-Point Assist Exception (0x00E00)

The optional floating-point assist exception defined by the PowerPC architecture is not implemented in the 604.

4.5.13 Performance Monitoring Interrupt (0x00F00)

The PowerPC 604 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.

The performance monitor uses the following SPRs:

- Performance monitor counters 1 and 2 (PMC1 and PMC2)—two 32-bit counters used to store the number of times a certain event has occurred.
- The monitor mode control register 0 (MMCR0), which establishes the function of the counters.
- Sampled instruction address and sampled data address registers (SIA and SDA). The two address registers contain the addresses of the data and of the instruction that caused a threshold-related performance monitor interrupt.

The 604 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 interrupts, 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 2.1.2.4, “Performance Monitor Registers.” The performance monitor is described in Chapter 9, “Performance Monitor.”

4.5.14 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 (IABR[30] is set). 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.

4.5.15 System Management Interrupt (0x01400)

The 604 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 interrupt vector in the exception table (at offset 0x01400).

Like the external interrupt, a system management interrupt is signaled to the 604 by the assertion of an input signal. The system management interrupt signal ($\overline{\text{SMI}}$) is expected to remain asserted until the interrupt is taken. If the $\overline{\text{SMI}}$ signal is negated early, recognition of the interrupt request is not guaranteed. After the 604 begins execution of the system management interrupt handler, the system can safely negate the $\overline{\text{SMI}}$ signal. After the $\overline{\text{SMI}}$ signal is detected, the 604 stops dispatching instructions and waits for all pending instructions to complete. This allows any instructions in progress that need to take an exception to do so before the system management interrupt is taken.

When the exception is taken, 604 vectors to the system management interrupt vector in the interrupt table. The vector offset of the system management is 0x01400.

4.5.16 Power Management

Nap mode is a simple power-saving mode, in which all internal processing and bus operation is suspended. Software initiates nap mode by setting MSR[POW]. After this bit is set, the 604 suspends instruction dispatch and waits for all activity, including active and pending bus transactions, to complete. It then shuts down the internal chip clocks and enters nap mode state. The 604 indicates the internal idle state by asserting the HALTED output regardless whether the clock is stopped.

Nap mode must be entered by using the following code sequence:

```
naploop:

        sync
        mtmsr <GPR> (modify the POW bit only; at this point the EE bit should
have already been enabled by the software)
isync
ba naploop
```

Since this code sequence creates an infinite loop, the programmer should ensure that the exit routine (one of the exception handler routines listed below) properly updates SRR0 to return to a point outside of this loop.

While the 604 is in nap mode, all internal activity except for decrementer, timebase, and interrupt logic is stopped. During nap mode, the 604 does not snoop; if snooping is required, the system may assert the RUN signal. The clocks run while the RUN signal is asserted, but instruction execution does not resume. The HALTED output is deasserted to indicate any bus activity, including a cache block pushout caused by a snoop request, and is reasserted to indicate that the processor is idle and that the RUN signal can be safely deasserted to stop the clocks. The maximum latency from the RUN signal assertion to the starting of clock is three bus clock cycles.

To ensure proper handling of snoops in a multiprocessor system when a processor is the first to enter nap mode, the system must assert the RUN signal no later than the assertion of $\overline{\text{BG}}$ to another bus master. This constraint is necessary to ensure proper handling of snoops when the first processor is entering nap mode.

Nap mode is exited (clocks resume and MSR[POW] cleared) when an external interrupt is signaled by the assertion of $\overline{\text{INT}}$, $\overline{\text{SRESET}}$, $\overline{\text{MCP}}$, or $\overline{\text{SMI}}$, when a decrementer interrupt occurs, or when a hard reset is sensed.

For more information about the RUN and HALTED signals, refer to Section 7.2.10.4, “Run (RUN)—Input,” and Section 7.2.10.2, “Reservation (RSRV)—Output.”

Chapter 5
Memory Management

This chapter describes the PowerPC 604 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. In addition, the MMU provides access protection on a segment, block or page basis. This chapter describes the specific hardware used to implement the MMU model of the OEA in the 604. 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 segment descriptors, used to generate the interim virtual addresses, are stored as on-chip segment registers on 32-bit implementations (such as the 604). In addition, two translation lookaside buffers (TLBs) are implemented on the 604 to keep recently-used page address translations on-chip. Although the PowerPC OEA describes one MMU (conceptually), the 604 hardware maintains separate TLBs and table search resources for instruction and data accesses that can be performed independently (and simultaneously). Therefore, the 604 is described as having two MMUs, one for instruction accesses (IMMU) and one for data accesses (DMMU).

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 604, 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 4.3, “Exception Processing,” describes the MSR, which controls some of the critical functionality of the MMUs.

5.1 MMU Overview

The 604 implements the memory management specification of the PowerPC OEA for 32-bit implementations. Thus, it provides 4 Gbytes of effective address space accessible to supervisor and user programs with a 4-Kbyte page size and 256-Mbyte segment size. In addition, the MMUs of 32-bit PowerPC processors use an interim virtual address (52 bits) and hashed page tables in the generation of 32-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 604 MMU implementation defined by the OEA are as follows:

- Support for real 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 32-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—If the T bit in the indexed segment register 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 register.
- Segmented address translation—The 32-bit effective address is extended to a 52-bit virtual address by substituting 24 bits of upper address bits from the segment register, for the 4 upper bits of the EA, which are used as an index into the segment register. This 52-bit virtual address space is divided into 4-Kbyte pages, each of which can be mapped to a physical page.

The 604 also provides the following features that are not required by the PowerPC architecture:

- Separate translation lookaside buffers (TLBs)—The 128-entry, two-way set associative ITLBs and DTLBs keep recently-used page address translations on-chip.
- Table search operations performed in hardware—The 52-bit virtual address is formed and the MMU attempts to fetch the PTE, which contains the physical address, from the appropriate TLB on-chip. If the translation is not found in a TLB (that is, a TLB miss occurs), the hardware performs a table search operation (using a hashing function) to search for the PTE.

- TLB invalidation—The 604 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 5.4.3.2, "TLB Invalidation."

Table 5-1 summarizes the 604 MMU features, including those defined by the PowerPC architecture (OEA) for 32-bit processors and those specific to the 604.

Table 5-1. MMU Feature Summary

Feature Category	Architecturally Defined/ 604-Specific	Feature
Address ranges	Architecturally defined	2^{32} bytes of effective address
		2^{52} bytes of virtual address
		2^{32} bytes of physical address
Page size	Architecturally defined	4 Kbytes
Segment size	Architecturally defined	256 Mbytes
Block address translation	Architecturally defined	Range of 128 Kbyte–256 Mbyte sizes
		Implemented with IBAT and DBAT registers in BAT array
Memory protection	Architecturally defined	Segments selectable as no-execute
		Pages selectable as user/supervisor and read-only or guarded
		Blocks selectable as user/supervisor and read-only or guarded
Page history	Architecturally defined	Referenced and changed bits defined and maintained
Page address translation	Architecturally defined	Translations stored as PTEs in hashed page tables in memory
		Page table size determined by mask in SDR1 register
TLBs	Architecturally defined	Instructions for maintaining TLBs (**tlbie** and **tlbsync** instructions in 604)
	604-specific	128-entry, two-way set associative ITLB 128-entry, two-way set associative DTLB LRU replacement algorithm
Segment descriptors	Architecturally defined	Stored as segment registers on-chip (two identical copies maintained)
Page table search support	604-specific	The 604 performs the table search operation in hardware.

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

Figure 5-1 shows the conceptual organization of a PowerPC MMU in a 32-bit implementation; note that it does not describe the specific hardware used to implement the memory management function for a particular processor. Processors may optionally implement on-chip TLBs and may optionally support the automatic search of the page tables for PTEs. In addition, other hardware features (invisible to the system software) not depicted in the figure may be implemented.

The 604 maintains two on-chip TLBs with the following characteristics:

- 128 entries, two-way set associative (64 x 2), LRU replacement
- Data TLB supports the DMMU; instruction TLB supports the IMMU
- Hardware TLB update
- Hardware update of memory access recording bits in the translation table

In the event of a TLB miss, the hardware attempts to load the TLB based on the results of a translation table search operation.

Figure 5-2 and Figure 5-3 show the conceptual organization of the 604 instruction and data MMUs, respectively. The instruction addresses shown in Figure 5-2 are generated by the processor for sequential instruction fetches and addresses that correspond to a change of program flow. Data addresses shown in Figure 5-3 are generated by load and store instructions (both for the memory and the direct-store interfaces) and by cache instructions.

As shown in the figures, after an address is generated, the higher-order bits of the effective address, EA0–EA19 (or a smaller set of address bits, EA0–EA*n*, in the cases of blocks), are translated into physical address bits PA0–PA19. The lower-order address bits, A20–A31 are untranslated and therefore identical for both effective and physical addresses. After translating the address, the MMUs pass the resulting 32-bit physical address to the memory subsystem.

In addition to the higher-order address bits, the MMUs automatically keep 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 MMUs to appropriately direct the address translation and to enforce the protection hierarchy programmed by the operating system. Section 4.3, “Exception Processing,” describes the MSR, which controls some of the critical functionality of the MMUs.

The figures show the way in which the A20–A26 address bits index into the on-chip instruction and data caches to select a cache set. The remaining physical address bits are then compared with the tag fields (comprised of bits PA0–PA19) of the two selected cache blocks to determine if a cache hit has occurred. In the case of a cache miss, the instruction or data access is then forwarded to the bus interface unit which then initiates an external memory access.

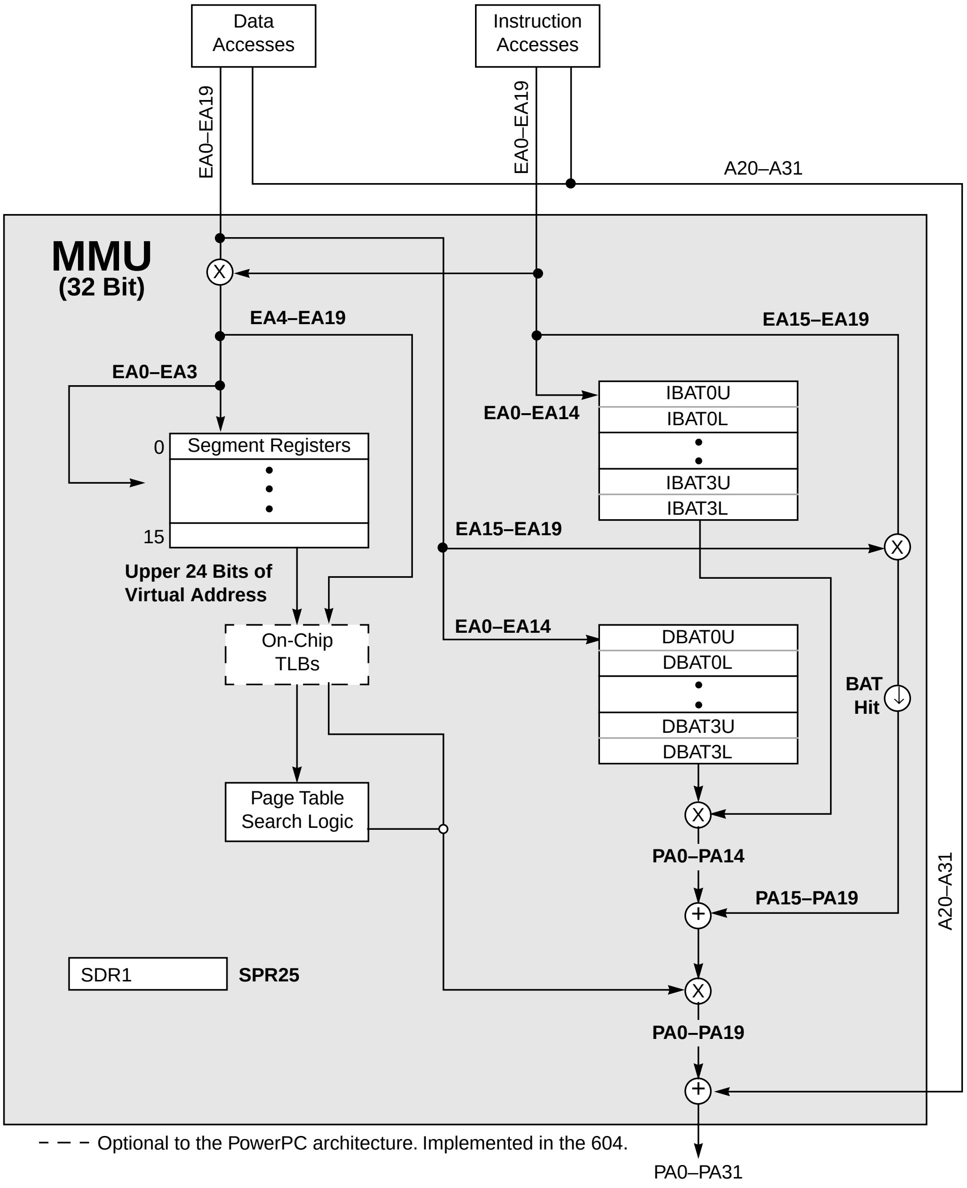


Figure 5-1. MMU Conceptual Block Diagram—32-Bit Implementations

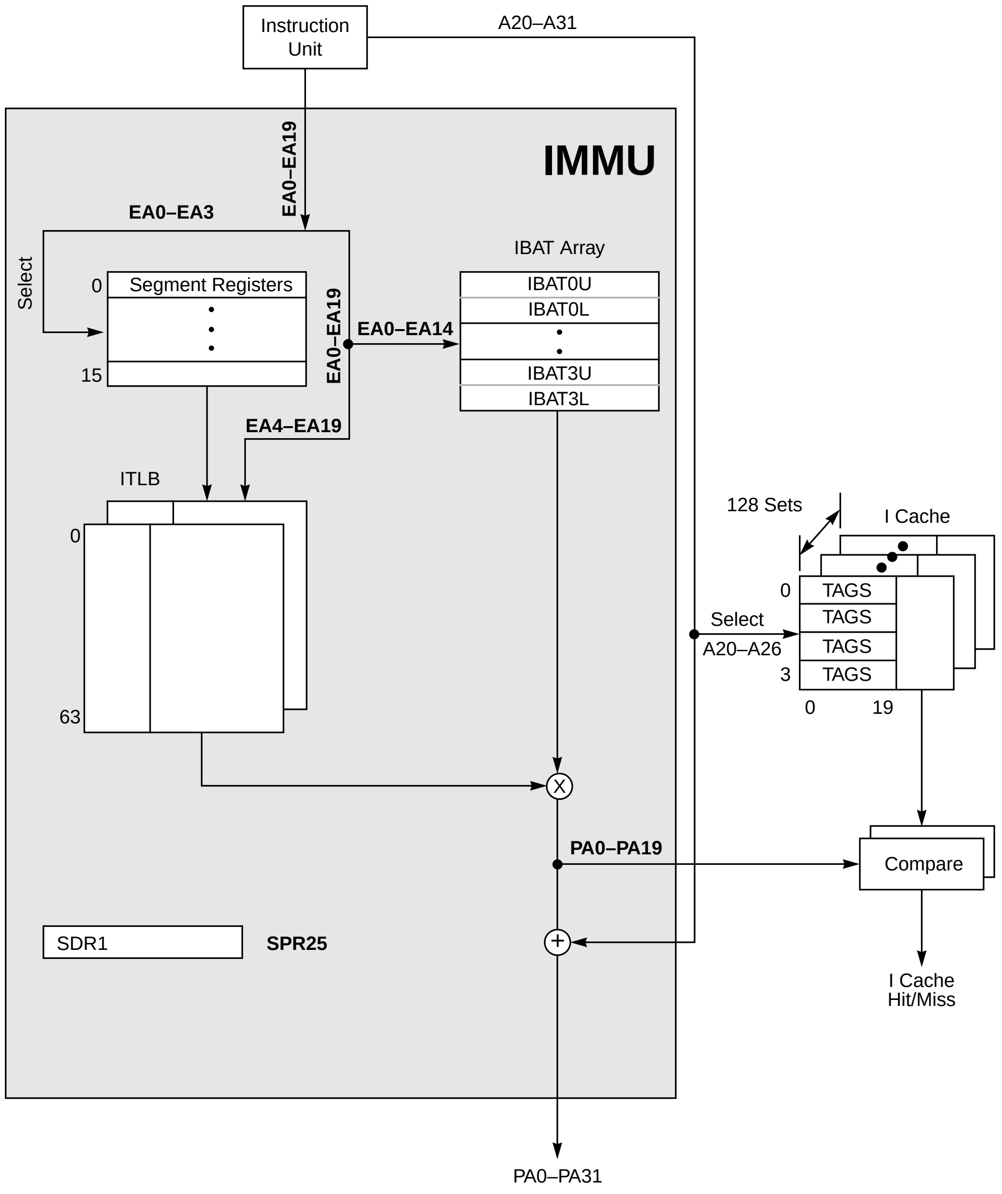


Figure 5-2. PowerPC 604 Microprocessor IMMU Block Diagram

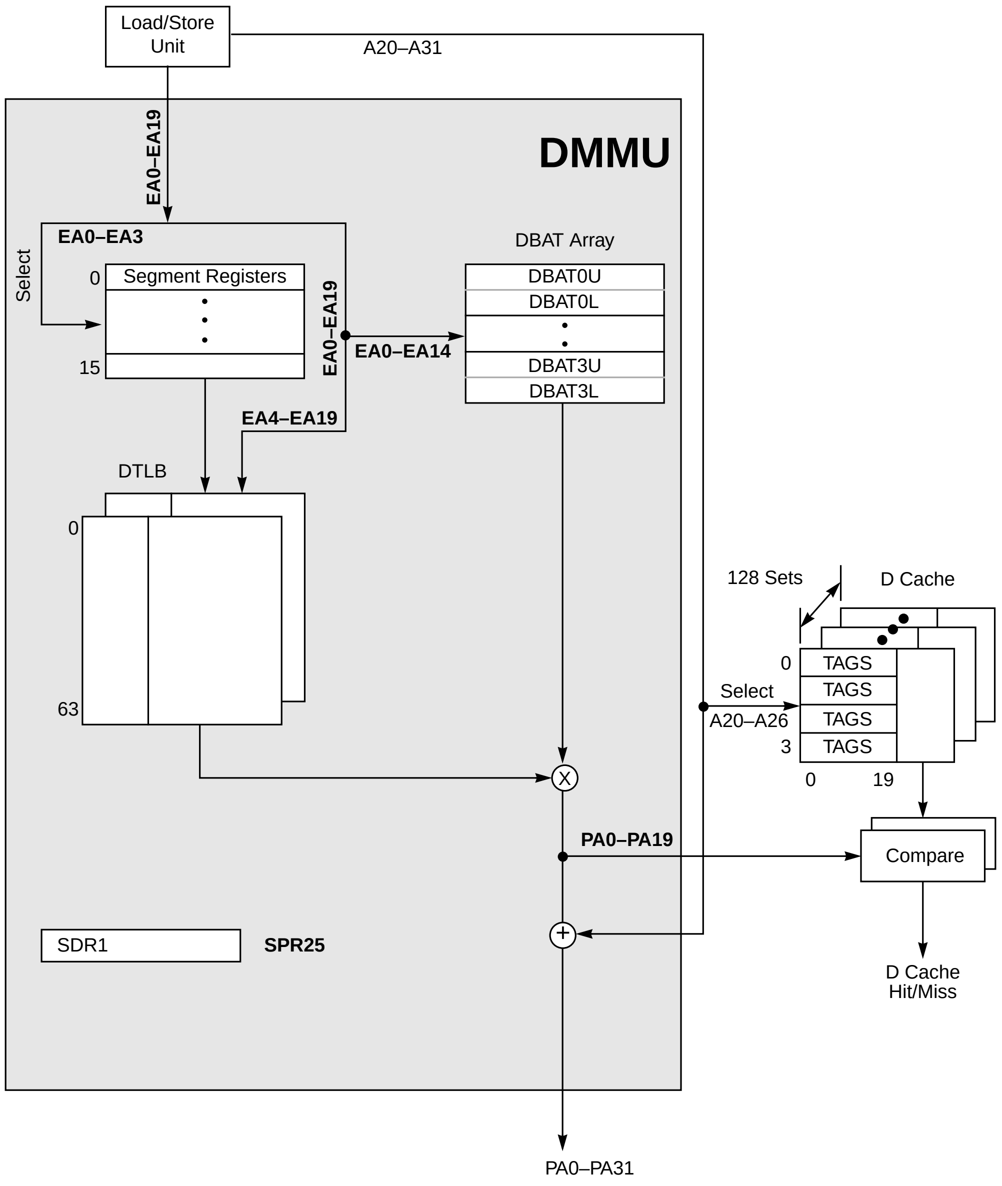


Figure 5-3. PowerPC 604 Microprocessor DMMU Block Diagram

5.1.3 Address Translation Mechanisms

PowerPC processors support the following four types of address translation:

- Page address translation—translates the page frame address for a 4-Kbyte page size
- Block address translation—translates the block number for blocks that range in size from 128 Kbyte to 256 Mbyte.
- Direct-store interface address translation—used to generate direct-store interface accesses on the external bus; not optimized for performance—present for compatibility only.
- Real addressing mode address translation—when address translation is disabled, the physical address is identical to the effective address.

Figure 5-4 shows the four address translation mechanisms provided by the MMUs. The segment descriptors shown in the figure control both the page and direct-store interface address translation mechanisms. When an access uses the page or direct-store interface address translation, the appropriate segment descriptor is required. In 32-bit implementations, one of the 16 on-chip segment registers (which contain segment descriptors) is selected by the four highest-order effective address bits.

A control bit in the corresponding segment descriptor then determines if the access is to memory (memory-mapped) or to the direct-store interface space. Note that the direct-store interface is present only for compatibility with existing I/O devices that used 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 devices 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 32-bit physical address used by the memory subsystem. In most cases, the physical address for the page resides in an on-chip TLB and is available for quick access. However, if the page address translation misses in an on-chip TLB, the MMU causes a search of the page tables in memory (using the virtual address information and a hashing function) to locate the required physical address.

Block address translation occurs in parallel with page and direct-store segment address translation and is similar to page address translation; however, fewer higher-order effective address bits are 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 TLB, 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 and the direct-store translation (occurring in parallel) are ignored.

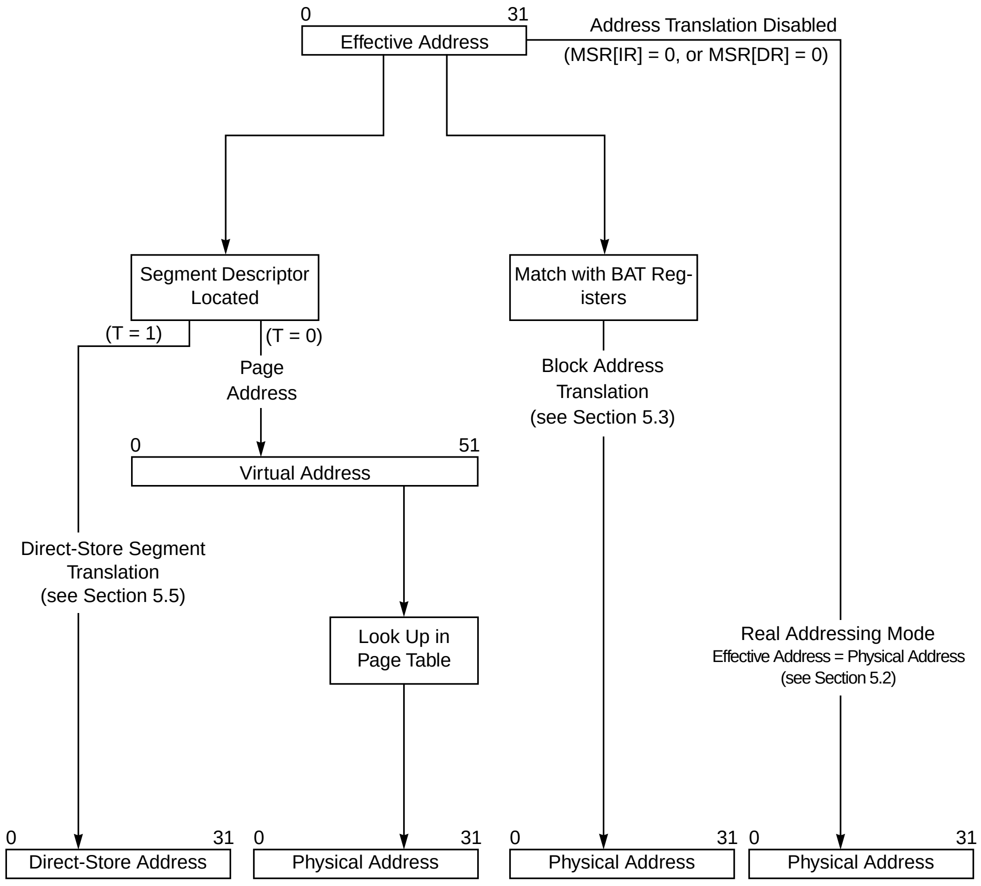


Figure 5-4. Address Translation Types

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 or guarded. Table 5-2 shows the protection options supported by the MMUs for pages.

Table 5-2. 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	—	√	—	—	√	—
Guarded						

√ 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, there is a facility in the VEA and OEA 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 that are 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 Note—In the process of loading the TLB, the 604 checks the state of the changed and referenced bits for the matched PTE. If the referenced bit is not set and the table search operation is initially caused by a load operation or by an instruction fetch, the 604 automatically sets the referenced bit in the translation table. Similarly, if the table search operation is caused by a store operation and either the referenced bit or the changed bit is not set, the hardware automatically sets both bits in the translation table. In addition, during the address translation portion of a store operation that hits in the TLB, the 604 checks the state of the changed bit. If the bit is not already set, the hardware automatically updates the TLB and the translation table in memory to set the changed bit. 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 5-5 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.

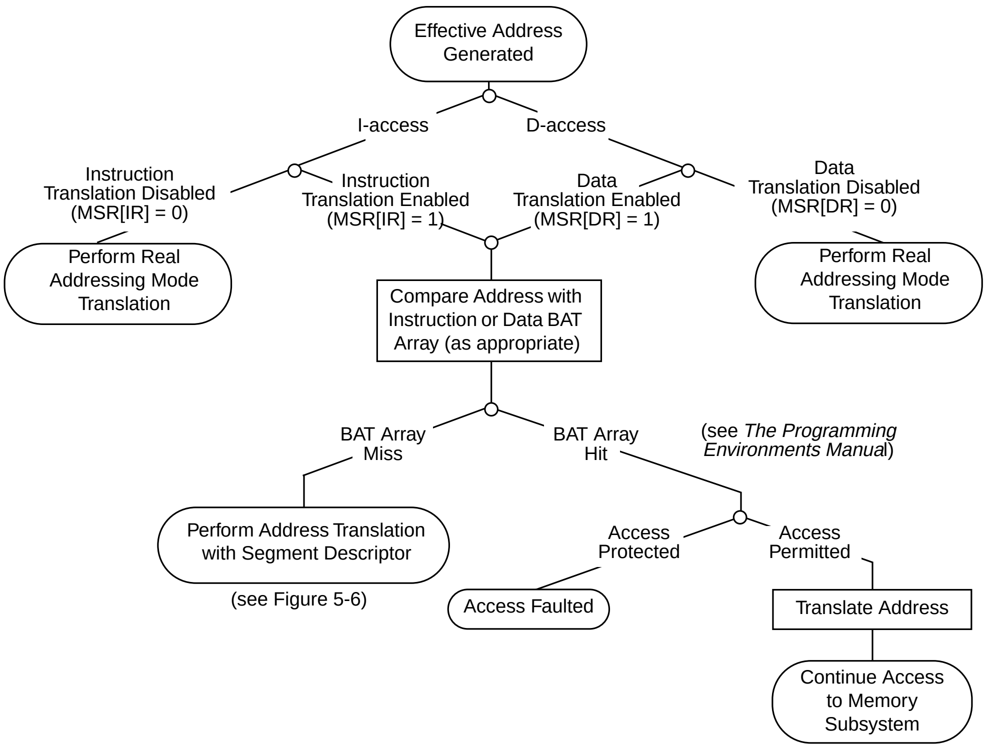

Effective Address Generated
I-access
D-access
Instruction Translation Disabled (MSR[IR] = 0)
Instruction Translation Enabled (MSR[IR] = 1)
Data Translation Enabled (MSR[DR] = 1)
Data Translation Disabled (MSR[DR] = 0)
Perform Real Addressing Mode Translation
Compare Address with Instruction or Data BAT 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 5-6)
Access Protected
Access Permitted
Access Faulted
Translate Address
Continue Access to Memory Subsystem

Figure 5-5. 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—The 604 BAT registers are not initialized by the hardware after the power-up or reset sequence. Consequently, all valid bits in both instruction and data BAT areas must be cleared before setting any BAT area for the first time. This is true regardless of whether address translation is enabled. Also, software must avoid overlapping blocks while updating a BAT area or areas. Even if translation is disabled, multiple BAT area hits are treated as programming errors and can corrupt the BAT registers and produce unpredictable results.

5.1.6.2 Page and Direct-Store Interface Address Translation Selection

If address translation is enabled and the effective address information does not match with a BAT array entry, then the segment descriptor must be located. Once the segment descriptor is 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 5-6. In addition, Figure 5-6 also shows the way in which the 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 604 implements TLBs, these branches are valid, and described in more detail throughout this chapter.

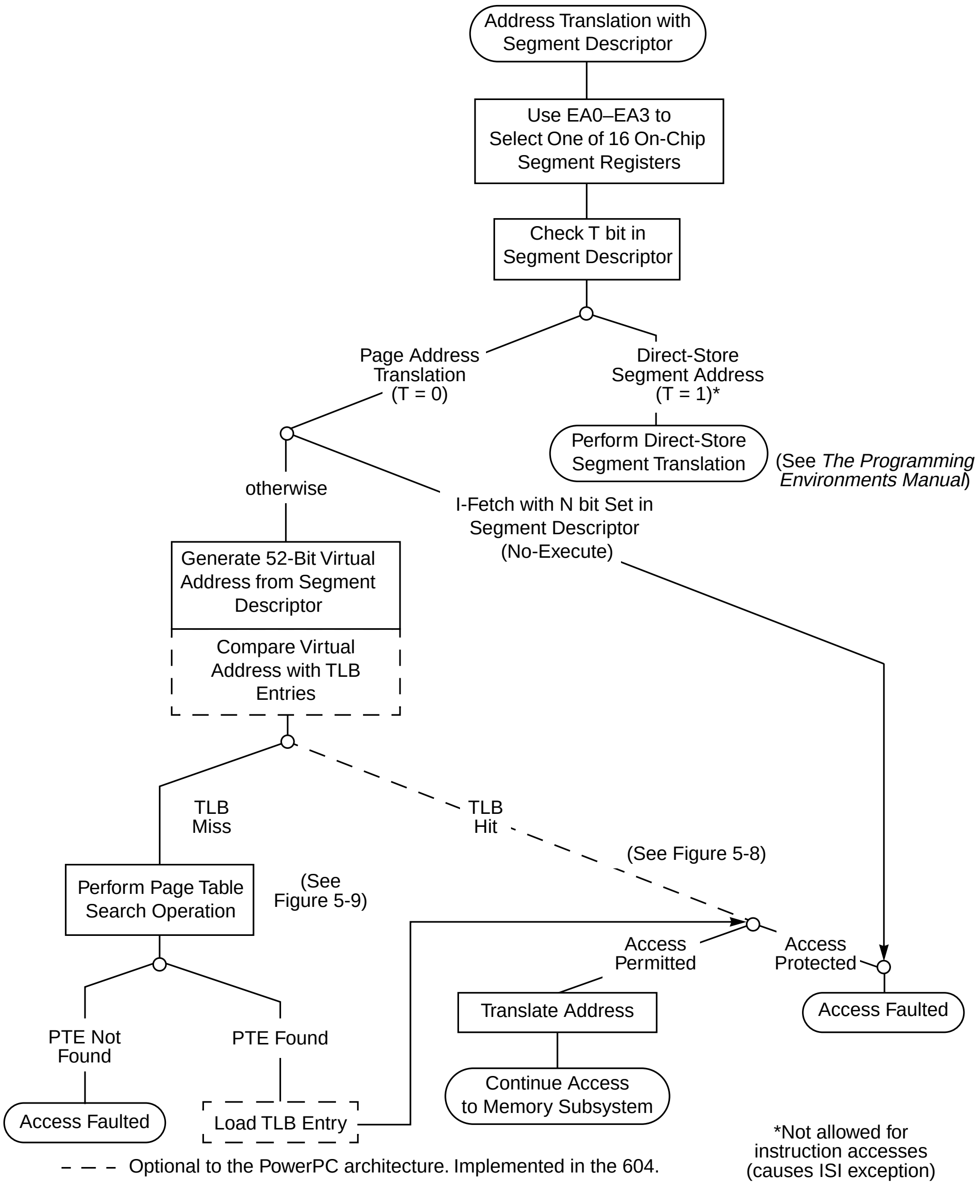

Address Translation with Segment Descriptor
Use EA0–EA3 to Select One of 16 On-Chip Segment Registers
Check T bit in Segment Descriptor
Page Address Translation (T = 0)
Direct-Store Segment Address (T = 1)*
Perform Direct-Store Segment Translation
(See *The Programming Environments Manual*)
otherwise
I-Fetch with N bit Set in Segment Descriptor (No-Execute)
Generate 52-Bit Virtual Address from Segment Descriptor
Compare Virtual Address with TLB Entries
TLB Miss
TLB Hit
(See Figure 5-8)
Perform Page Table Search Operation
(See Figure 5-9)
Access Permitted
Access Protected
Translate Address
Access Faulted
PTE Not Found
PTE Found
Continue Access to Memory Subsystem
Access Faulted
Load TLB Entry
*Not allowed for instruction accesses (causes ISI exception)
Optional to the PowerPC architecture. Implemented in the 604.

Figure 5-6. General Flow of Page and Direct-Store Interface Address Translation

5.1.6.3 Selection of Page Address Translation

If the T bit in the corresponding segment descriptor is 0, page address translation is selected. The information in the segment descriptor is then used to generate the 52-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 604 has two on-chip TLBs to store recently-used PTEs on-chip.

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 requires a search of the page table. In this case, the 604 hardware performs the page table search operation. If the PTE is successfully found, a new TLB entry is created and the page translation is once again attempted. This time, the TLB 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

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 5.5, “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.
- 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 5-3.

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.

Table 5-3. 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[1] = 1
		D access: DSI exception DSISR[1] =1
Block protection violation	Conditions described for block in “Block Memory Protection” in Chapter 7, “Memory Management,” in *The Programming Environments Manual.*“	I access: ISI exception SRR1[4] = 1
		D access: DSI exception DSISR[4] =1
Page protection violation	Conditions described for page in “Block Memory Protection” in Chapter 7, “Memory Management,” in *The Programming Environments Manual.*	I access: ISI exception SRR1[4] = 1
		D access: DSI exception DSISR[4] =1
No-execute protection violation	Attempt to fetch instruction when SR[N] = 1	ISI exception SRR1[3] = 1
Instruction fetch from direct-store segment	Attempt to fetch instruction when SR[T] = 1	ISI exception SRR1[3] =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[3] =1

In addition to the translation exceptions, there are other MMU-related conditions (some of them defined as implementation-specific and therefore, not required by the architecture) that can cause an exception to occur. These exception conditions map to the processor exception as shown in Table 5-4. 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 4.5.6, “Alignment Exception (0x00600).”

Note that some exception conditions depend upon whether the memory area is set up as write-though (W = 1) or cache-inhibited (I = 1). These bits are described fully in “Memory/Cache Access Attributes,” in Chapter 5, “Cache Model and Memory Coherency,” of *The Programming Environments Manual.* Refer to Chapter 4, “Exceptions,” and to Chapter 6, “Exceptions,” in *The Programming Environments Manual* for a complete description of the SRR1 and DSISR bit settings for these exceptions.

Table 5-4. Other MMU Exception Conditions for the PowerPC 604 Processor

Condition	Description	Exception
dcbz with W = 1 or I = 1	**dcbz** instruction to write-through or cache-inhibited segment or block	Alignment exception (not required by architecture for this condition)
dcbz when the data cache is locked	The **dcbz** instruction takes an alignment exception if the data cache is locked (HID0 bits 18 and 19) when it is executed.	Alignment exception
lwarx or **stwcx.** with W = 1	Reservation instruction to write-through segment or block	DSI exception DSISR[5] = 1
lwarx, **stwcx.**, **eciwx**, or **ecowx** instruction to direct-store segment	Reservation instruction or external control instruction when SR[T] =1	DSI exception DSISR[5] = 1
Floating-point load or store to direct-store segment	FP memory access when SR[T] =1	Alignment exception (not required by architecture)
Load or store that results in 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-specific)

5.1.8 MMU Instructions and Register Summary

The MMU instructions and registers provide the operating system with the ability to set up the block address translation areas and the page tables in memory.

Note that because the implementation of TLBs is optional, the instructions that refer to these structures are also optional. However, as these structures serve as caches of the page table, the architecture specifies a software protocol for maintaining coherency between these caches and the tables in memory whenever changes are made to the tables in memory. 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.

Note that the 604 implements all TLB-related instructions except **tlbia**, which is treated as an illegal instruction.

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 5-5 summarizes 604 instructions that specifically control the MMU.

Table 5-5. PowerPC 604 Microprocessor Instruction Summary—Control MMUs

Instruction	Description
mtsr SR,**r**S	Move to Segment Register SR[SR#]← **r**S
mtsrin **r**S,**r**B	Move to Segment Register Indirect SR[**r**B[0–3]]←**r**S
mfsr **r**D,SR	Move from Segment Register **r**D←SR[SR#]
mfsrin **r**D,**r**B	Move from Segment Register Indirect **r**D←SR[**r**B[0–3]]
tlbie rB *	Execution of this instruction causes all entries in the congruence class corresponding to the EA to be invalidated in the processor executing the instruction and in the other processors attached to the same bus. Software must ensure that instruction fetches or memory references to the virtual pages specified by the **tlbie** instruction have been completed prior to executing the **tlbie** instruction.
tlbsync *	The **tlbsync** operation appears on the bus as a distinct operation that causes synchronization of snooped **tlbie** instructions.

* These instructions are defined by the PowerPC architecture, but are optional.

Table 5-6 summarizes the registers that the operating system uses to program the 604 MMUs. These registers are accessible to supervisor-level software only. These registers are described in Chapter 2, “PowerPC 604 Processor Programming Model.”

Table 5-6. PowerPC 604 Microprocessor MMU Registers

Register	Description
Segment registers (SR0–SR15)	The sixteen 32-bit segment registers are present only in 32-bit implementations of the PowerPC architecture. 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 32-bit registers in 32-bit implementations. These are special-purpose registers that are accessed by the **mtspr** and **mfspr** instructions.
SDR1	The SDR1 register specifies the variables used in accessing the page tables in memory. SDR1 is defined as a 32-bit register for 32-bit implementations. This special-purpose register is accessed by the **mtspr** and **mfspr** instructions.

5.1.9 TLB Entry Invalidation

For PowerPC processors such as the 604 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.

Execution of this instruction causes all entries in the congruence class corresponding to the presented EA to be invalidated in the processor executing the instruction and in the other processors attached to the same bus.

The **tlbsync** operation appears on the bus as a distinct operation, that causes synchronization of snooped **tlbie** instructions. Section 5.4.3.2, “TLB Invalidation,” describes the TLB invalidation mechanisms in the 604.

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 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 604 is described in Chapter 7, “Memory Management,” in *The Programming Environments Manual* for 32-bit implementations.

5.4 Memory Segment Model

The 604 adheres to the memory segment model as defined in Chapter 7, “Memory Management,” in *The Programming Environments Manual* for 32-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 (52 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.” If not, 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 604.

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 604 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 604, the referenced and changed bits are updated as follows:

- For TLB hits, the C bit is updated according to Table 5-7.
- 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 5-7. 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 604 initiates a table search operation to update C.
11	No special action for read or write

The table 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.

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 604 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 604 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.** 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 604). 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 604 sets it and a table search operation is performed to also set the C bit in the corresponding PTE in the page table. The 604 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.** 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, 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 604 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 5-8 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 5-8. Model for Guaranteed R and C Bit Settings

Priority	Scenario	Causes Setting of R Bit		Causes Setting of C Bit	
		OEA	604	OEA	604
1	No-execute protection violation	No	No	No	No
2	Page protection violation	Maybe	Yes	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	No
7	Zero-length store (**stswx**)	Maybe[1]	No	Maybe[1]	No
8	Store conditional (**stwcx.**) that does not store	Maybe[1]	Yes	Maybe[1]	Yes
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

Table 5-8. Model for Guaranteed R and C Bit Settings (Continued)

Priority	Scenario	Causes Setting of R Bit		Causes Setting of C Bit	
		OEA	604	OEA	604
12	**icbi**, **dcbt**, **dcbtst**, **dcbst**, or **dcbf** instruction	Maybe	Yes	no	no
13	**dcbi** instruction	Maybe[1]	Yes	Maybe[1]	Yes

[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 604).

For more information, see “Page History Recording” in Chapter 7, “Memory Management,” of *The Programming Environments Manual*.

5.4.2 Page Memory Protection

The 604 implements page memory protection as it is defined in Chapter 7, “Memory Management,” in *The Programming Environments Manual*.

5.4.3 TLB Description

Because the 604 has two MMUs (IMMU and DMMU) that operate in parallel, some of the MMU resources are shared, and some are actually duplicated (shadowed) in each MMU to maximize performance. For example, although the architecture defines a single set of segment registers for the MMU, the 604 maintains two identical sets of segment registers, one for the IMMU and one for the DMMU; when a segment register instruction executes, the 604 automatically updates both sets.

5.4.3.1 TLB Organization

The 604 implements separate 128-entry data and instruction TLBs to support the implementation of separate instruction and data MMUs. This section describes the hardware resources provided in the 604 to facilitate page address translation. Note that the hardware implementation of the MMU is not specified by the architecture, and while this description applies to the 604, it does not necessarily apply to other PowerPC processors.

Each TLB contains 128 entries organized as a two-way set associative array with 64 sets as shown in Figure 5-7 for the DTLB (the ITLB organization is the same). When an address is being translated, a set of two TLB entries is indexed in parallel with the access to a segment register. 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.

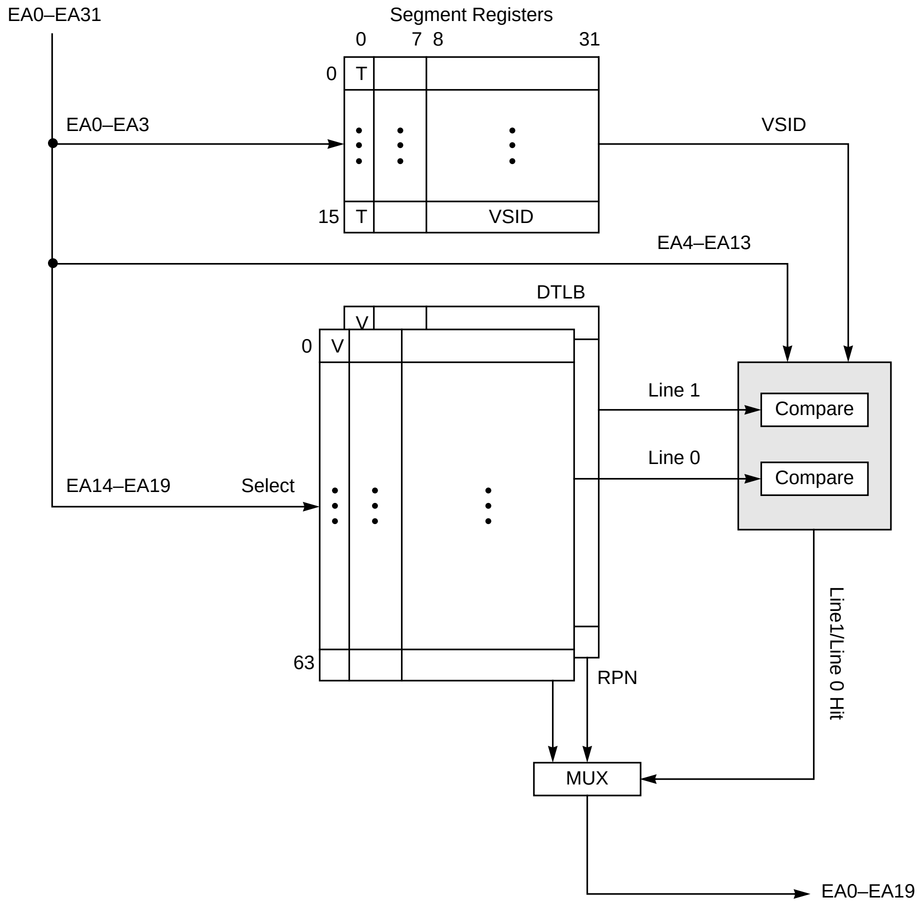


Figure 5-7. Segment Register and DTLB Organization

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 translation process begins again, this time with a TLB hit.

TLB entries are on-chip copies of PTEs in the page tables in memory and are similar in structure. TLB entries consist of two words; the upper-order word contains the VSID and API fields of the upper-order word of the PTE and the lower-order word contains the RPN, the C bit, the WIMG bits and the PP bits (as in the lower-order word of the PTE). To uniquely identify a TLB entry as the required PTE, the PTE also contains four more bits of the page index, EA10–EA13 (in addition to the API bits of the PTE). Formats for the PTE are given in "PTE Format for 32-Bit Implementations," in Chapter 7, "Memory Management," of *The Programming Environments Manual.*

Software does not have direct access to the TLB arrays, 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 MMUs can be accessed simultaneously (both sets of segment registers and TLBs can be accessed in the same clock), when there is an exception condition, 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. Also, note that the segment registers do not have a valid bit, and so they should also be initialized before translation is enabled.

5.4.3.2 TLB Invalidation

The 604 implements the optional **tlbie** and **tlbsync** instructions, which are used to invalidate TLB entries. The execution of the **tlbie** instruction always invalidates four entries—both the ITLB entries indexed by EA14–EA19 and both the indexed entries of the DTLB.

Execution of the **tlbie** instruction causes all entries in the congruence class corresponding to the specified EA to be invalidated in the processor executing the instruction and also in the other processors attached to the same bus by causing a TLB invalidate broadcast operation on the bus as described in Section 7.2.4, "Address Transfer Attribute Signals."

A TLB invalidate broadcast operation is an address-only transaction issued by a processor when it executes a **tlbie** instruction. The address transmitted as part of this transaction contains bits 12–19 of the EA in their correct respective bit positions.

When a snooping 604 detects a TLB invalidate operation on the bus, it accepts the operation only if no TLB invalidation is being performed by this processor and all processors on the bus accept the operation ($\overline{\text{ARTRY}}$ is not asserted). 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 the instruction has completed. Note that a 604 processor can only have one TLB invalidation operation pending internally. Thus if the 604 has a pending TLB invalidate operation, it asserts the $\overline{\text{ARTRY}}$ snoop status in response to another TLB invalidate operation on the bus. Detected TLB invalidate operations on the bus and the execution of the **tlbie** instruction both cause a congruence-class invalidation on both instruction and data TLBs.

The OEA requires that a synchronization instruction be issued to guarantee completion of a **tlbie** instruction across all processors of a system. The 604 implements the **tlbsync** instruction which causes a TLBSYNC broadcast operation to appear on the bus as an address-only transaction, distinct 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.

When the TLBSYNC bus operation is detected by a snooping 604, the 604 asserts the $\overline{\text{ARTRY}}$ snoop status if any operations based on an invalidated TLB are pending.

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.

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.

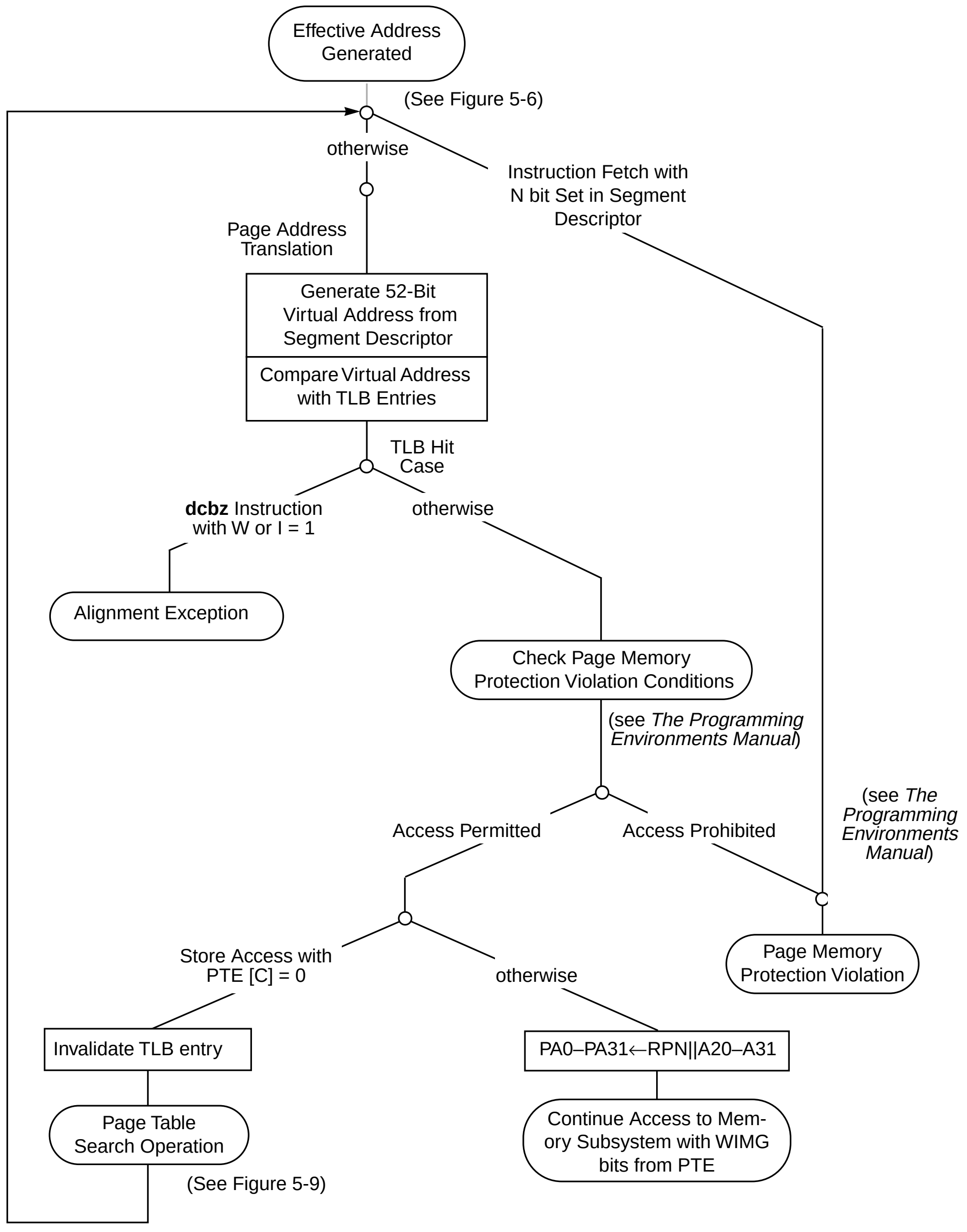
The **tlbia** instruction is optional for an implementation if its effects can be achieved through some other mechanism. As described above, the **tlbie** instruction can be used to invalidate a particular index of the TLB based on EA[14–19]. With that concept in mind, a sequence of 64 **tlbie** instructions followed by a single **tlbsync** instruction would cause all the 604 TLB structures to be invalidated (for EA[14–19] = 0, 1, 2, ..., 63). Therefore the **tlbia** instruction is not implemented on the 604. Execution of a **tlbia** instruction causes an illegal instruction program exception.

The **tlbie** and **tlbsync** instructions are described in detail in Section 2.3.6.3.3, “Translation Lookaside Buffer Management Instructions—(OEA).” For more information about how other processors react to TLB operations broadcast on the system bus of a multiprocessing system, see Section 3.9.6, “Cache Reaction to Specific Bus Operations.”

5.4.4 Page Address Translation Summary

Figure 5-8 provides the detailed flow for the page address translation mechanism.

The figure includes the checking of the N bit in the segment descriptor and then expands on the “TLB Hit” branch of Figure 5-6. The detailed flow for the “TLB Miss” branch of Figure 5-6 is described in Section 5.4.5, “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
(See Figure 5-6)
otherwise
Instruction Fetch with N bit Set in Segment Descriptor
Page Address Translation
Generate 52-Bit Virtual Address from Segment Descriptor
Compare Virtual Address with TLB Entries
TLB Hit Case
dcbz Instruction with W or I = 1
otherwise
Alignment Exception
Check Page Memory Protection Violation Conditions
(see *The Programming Environments Manual*)
Access Permitted
Access Prohibited
(see *The Programming Environments Manual*)
Store Access with PTE [C] = 0
otherwise
Page Memory Protection Violation
Invalidate TLB entry
PA0–PA31←RPN||A20–A31
Page Table Search Operation
(See Figure 5-9)
Continue Access to Memory Subsystem with WIMG bits from PTE

Figure 5-8. Page Address Translation Flow—TLB Hit

5.4.5 Page Table Search Operation

If the translation is not found in the TLBs (a TLB miss), the 604 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 604:

1. The 32-bit physical address of the primary PTEG is generated as described in “Page Table Addresses” in Chapter 7, “Memory Management,” of *The Programming Environments Manual*.
2. The first PTE (PTE0) in the primary PTEG is read from memory. PTE reads should occur with an implied WIM memory/cache mode control bit setting of 0b001. Therefore, they are considered cacheable and read (burst) from memory and placed in the cache.
3. The PTE in the selected PTEG is 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
   — PTE[V] = 1
   — PTE[VSID] = VA[0–23]
   — PTE[API] = VA[24–29]
4. If a match is not found, step 3 is repeated for each of the other seven PTEs in the primary PTEG. If a match is found, the table search process continues as described in step 8. If a match is not found within the 8 PTEs of the primary PTEG, the address of the secondary PTEG is generated.
5. The first PTE (PTE0) in the secondary PTEG is read from memory. Again, because PTE reads typically have a WIM bit combination of 0b001, an entire cache line is read into the on-chip cache.
6. The PTE in the selected secondary PTEG is tested for a match with the virtual page number (VPN) of the access. For a match to occur, the following must be true:
   — PTE[H] = 1
   — PTE[V] = 1
   — PTE[VSID] = VA[0–23]
   — PTE[API] = VA[24–29]
7. If a match is not found, step 6 is repeated for each of the other seven PTEs in the secondary PTEG. If it is never found, an exception is taken (step 9).
8. If a match is found, the PTE is written into the on-chip TLB 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 the access is a write operation) and the table search is complete.

9. If a match is not found within the 8 PTEs of the secondary PTEG, the search fails, and a page fault exception condition occurs (either an ISI exception or a DSI exception).

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 5-9 and Figure 5-10 show how the conceptual model for the primary and secondary page table search operations, described in *The Programming Environments Manual* are realized in the 604.

Figure 5-9 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.

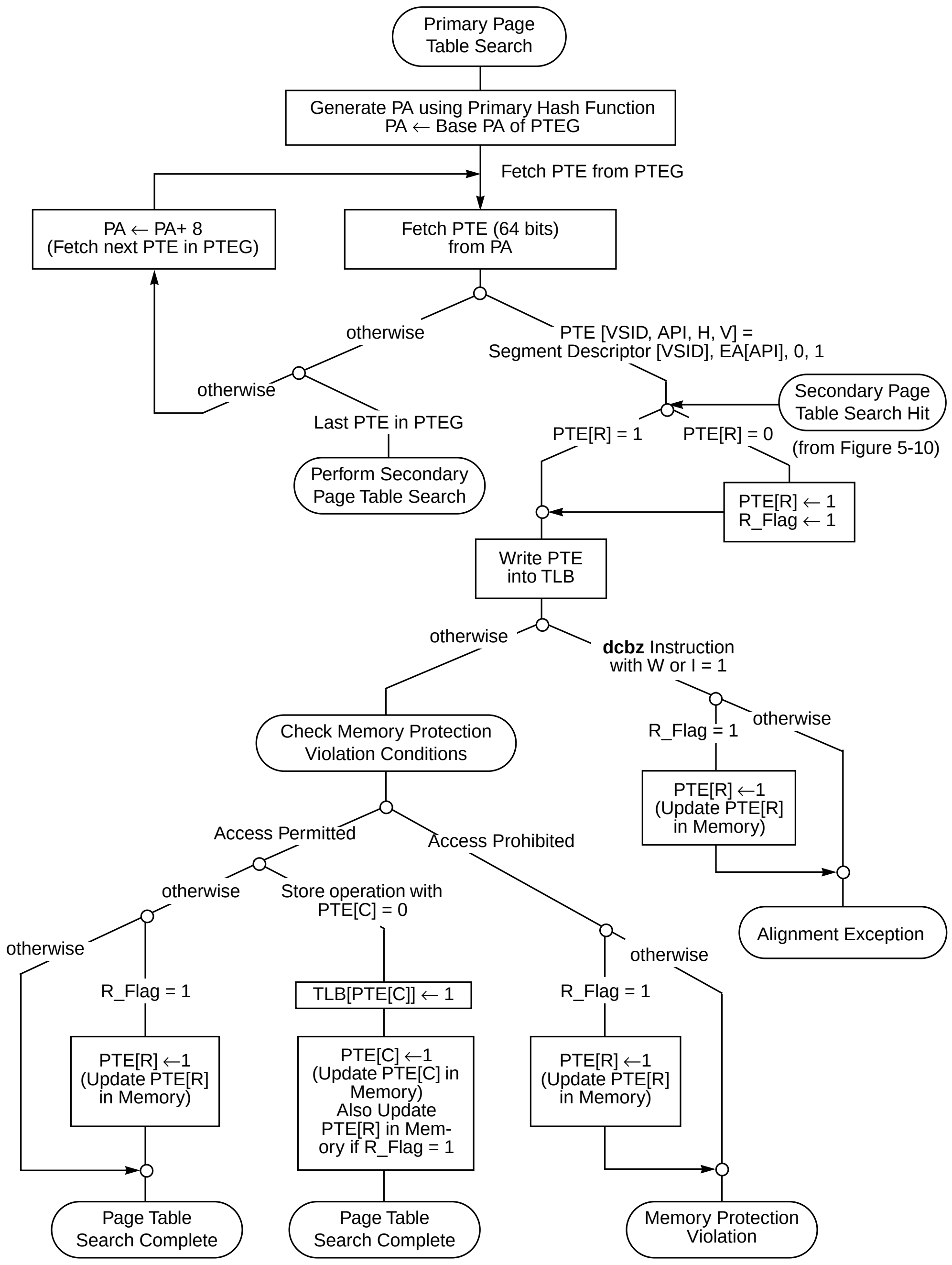

Primary Page Table Search
Generate PA using Primary Hash Function
PA ← Base PA of PTEG
Fetch PTE from PTEG
PA ← PA+ 8
(Fetch next PTE in PTEG)
Fetch PTE (64 bits) from PA
otherwise
PTE [VSID, API, H, V] = Segment Descriptor [VSID], EA[API], 0, 1
otherwise
Last PTE in PTEG
Perform Secondary Page Table Search
Secondary Page Table Search Hit
(from Figure 5-10)
PTE[R] = 1
PTE[R] = 0
PTE[R] ← 1
R_Flag ← 1
Write PTE into TLB
otherwise
dcbz Instruction with W or I = 1
Check Memory Protection Violation Conditions
R_Flag = 1
otherwise
PTE[R] ←1 (Update PTE[R] in Memory)
Alignment Exception
Access Permitted
Access Prohibited
otherwise
Store operation with PTE[C] = 0
otherwise
R_Flag = 1
TLB[PTE[C]] ← 1
PTE[R] ←1 (Update PTE[R] in Memory)
PTE[C] ←1 (Update PTE[C] in Memory) Also Update PTE[R] in Memory if R_Flag = 1
otherwise
R_Flag = 1
PTE[R] ←1 (Update PTE[R] in Memory)
Page Table Search Complete
Page Table Search Complete
Memory Protection Violation

Figure 5-9. Primary Page Table Search

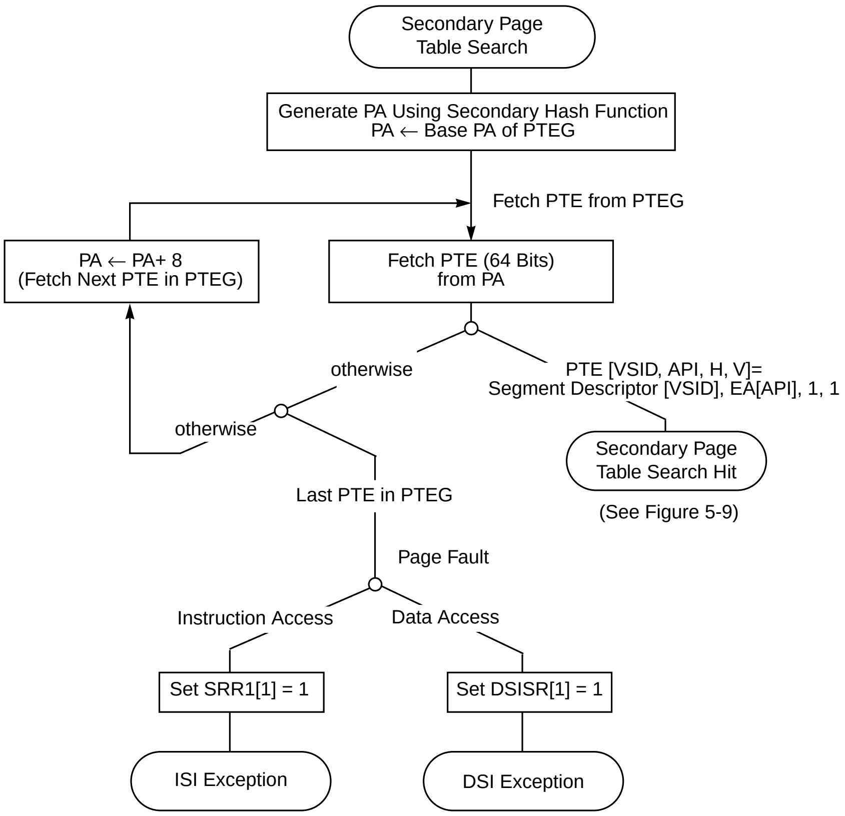

Secondary Page Table Search
Generate PA Using Secondary Hash Function PA ← Base PA of PTEG
Fetch PTE from PTEG
PA ← PA+ 8 (Fetch Next PTE in PTEG)
Fetch PTE (64 Bits) from PA
otherwise
PTE [VSID, API, H, V]= Segment Descriptor [VSID], EA[API], 1, 1
otherwise
Secondary Page Table Search Hit
(See Figure 5-9)
Last PTE in PTEG
Page Fault
Instruction Access
Data Access
Set SRR1[1] = 1
Set DSISR[1] = 1
ISI Exception
DSI Exception

Figure 5-10. Secondary Page Table Search Flow

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.

The LSU initiates out-of-order accesses without knowledge of whether it is legal to do so. Therefore, the MMU does not perform hardware table search due to TLB misses until the request is nonspeculative. In these out-of-order cases, the MMU does detect protection violations and whether a **dcbz** instruction specifies a page marked as write-through or cache-inhibited. The MMU also detects alignment exceptions caused by the **dcbz** instruction, which prevents the changed bit in the PTE from being updated erroneously.

Note that when a TLB miss occurs, the MMU does not begin the table search operation if the access is out of order.

If the MMU registers are being accessed by an instruction in the instruction stream, the IMMU stalls for one translation cycle to perform those operation. The sequencer serializes instructions to ensure the data correctness. For updating the IBATs and SRs, the sequencer classifies those operations as fetch serialization. After such an instruction is dispatched, the instruction buffer is flushed and the fetch stalls until the instruction completes. However, for reading from the IBATs, the operation is classified as execution serialization. As long as the LSU ensures that all previous instructions can be executed, subsequent instructions can be fetched and dispatched.

5.4.6 Page Table Updates

This section describes the requirements on the software when updating page tables in memory via some pseudocode examples. Multiprocessor systems must follow the rules described in this section so that all processors operate with a consistent set of page tables. Even single-processor systems must follow certain rules, because software changes must be synchronized with the other instructions in execution and with automatic updates that may be made by the hardware (referenced and changed bit updates). Updates to the tables include the following operations:

- Adding a PTE
- Modifying a PTE, including modifying the R and C bits of a PTE
- Deleting a PTE

PTEs must be locked on multiprocessor systems. Access to PTEs must be appropriately synchronized by software locking of (that is, guaranteeing exclusive access to) PTEs or PTEGs if more than one processor can modify the table at that time. In the examples below, ‘lock()’ and ‘unlock()’ refer to software locks that must be performed to provide exclusive access to the PTE being updated. See Appendix E, “Synchronization Programming Examples,” in *The Programming Environments Manual*, for more information about the use of the reservation instructions (such as the **lwarx** and **stwcx.** instructions) to perform software locking.

On single-processor systems, PTEs need not be locked. To adapt the examples given below for the single-processor case, simply delete the ‘lock()’ and ‘unlock()’ lines from the examples. The **sync** instructions shown are required even for single-processor systems (to ensure that all previous changes to the page tables and all preceding **tlbie** instructions have completed).

When TLBs are implemented, they are defined as noncoherent caches of the page tables. TLB entries must be invalidated explicitly with the TLB invalidate entry instruction (**tlbie**) whenever the corresponding PTE is modified. In a multiprocessor system, the **tlbie** instruction must be controlled by software locking, so that the **tlbie** is issued on only one processor at a time. The **sync** instruction causes the processor to wait until the TLB invalidate operation in progress by this processor is complete.

The PowerPC OEA defines the **tlbsync** instruction that ensures that TLB invalidate operations executed by this processor have caused all appropriate actions in other processors. In a system that contains multiple processors, the **tlbsync** functionality must be used in order to ensure proper synchronization with the other PowerPC processors. Note that for compatibility with PowerPC 601 microprocessor systems a **sync** instruction must also follow the **tlbsync** to ensure that the **tlbsync** has completed execution on this processor.

Any processor, including the processor modifying the page table, may access the page table at any time in an attempt to reload a TLB entry. An inconsistent page table entry must never accidentally become visible; thus, there must be synchronization between modifications to the valid bit and any other modifications (to avoid corrupted data). This requires as many as two **sync** operations for each PTE update.

Because the V, R, and C bits each reside in a distinct byte of a PTE, programs may update these bits with byte store operations (without requiring any higher-level synchronization). However, extreme care must be taken to ensure that no store overwrites one of these bytes accidentally. Processors write referenced and changed bits with unsynchronized, atomic byte store operations.

Explicitly altering certain MSR bits (using the **mtmsr** instruction), or explicitly altering PTEs, or certain system registers, may have the side effect of changing the effective or physical addresses from which the current instruction stream is being fetched. This kind of side effect is defined as an implicit branch. Implicit branches are not supported and an attempt to perform one causes boundedly undefined results. Therefore, PTEs must not be changed in a manner that causes an implicit branch. Chapter 2, “PowerPC Register Set,” in *The Programming Environments Manual*, lists the possible implicit branch conditions that can occur when system registers and MSR bits are changed.

5.4.7 Segment Register Updates

There are certain synchronization requirements for using the move to segment register instructions. These are described in “Synchronization Requirements for Special Registers and for Lookaside Buffers” in Chapter 2, “PowerPC Register Set,” in *The Programming Environments Manual*.

5.5 Direct-Store Interface Address Translation

As described for memory segments, all accesses generated by the processor map to a segment descriptor in the segment table. 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. 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.

5.5.1 Direct-Store Interface Accesses

When the address translation process determines that the segment descriptor has T = 1, direct-store interface address translation is selected and no reference is made to the page tables and referenced and changed bits are not updated. These accesses are performed as if the WIMG bits were 0b0101; that is, caching is inhibited, the accesses bypass the cache, hardware-enforced coherency is not required, and the accesses are considered guarded.

The specific protocol invoked to perform these accesses involves the transfer of address and data information in packets; however, the PowerPC OEA does not define the exact hardware protocol used for direct-store interface accesses. Some instructions cause multiple address/data transactions to occur on the bus. In this case, the address for each transaction is handled individually with respect to the DMMU.

The following data is sent by the 604 to the memory controller in the protocol (two packets consisting of address-only cycles) described in Section 8.6, “Direct-Store Operation.”

- Packet 0
  - One of the K*x* bits (Ks or Kp) is selected to be the key as follows:
    - For supervisor accesses (MSR[PR] = 0), the Ks bit is used and Kp is ignored.
    - For user accesses (MSR[PR] = 1), the Kp bit is used and Ks is ignored.
  - The contents of bits 3–31 of the segment register, which is the BUID field concatenated with the “controller-specific” field.
- Packet 1—SR[28–31] concatenated with the 28 lower-order bits of the effective address, EA4–EA31.

5.5.2 Direct-Store Segment Protection

Page-level memory protection as described in Section 5.4.2, “Page Memory Protection,” is not provided for direct-store segments. The appropriate key bit (Ks or Kp) from the segment descriptor is sent to the memory controller, and the memory controller implements any protection required. Frequently, no such mechanism is provided; the fact that a direct-store segment is mapped into the address space of a process may be regarded as sufficient authority to access the segment.

5.5.3 Instructions Not Supported in Direct-Store Segments

The following instructions are not supported at all and cause a DSI exception (with DSISR[5] set) when issued with an effective address that selects a segment descriptor that has T = 1 (or when MSR[DR] = 0):

- **lwarx**
- **stwcx.**
- **eciwx**
- **ecowx**

5.5.4 Instructions with No Effect in Direct-Store Segments

The following instructions are executed as no-ops when issued with an effective address that selects a segment where T = 1:

- **dcbt**
- **dcbtst**
- **dcbf**
- **dcbi**
- **dcbst**
- **dcbz**
- **icbi**

5.5.5 Direct-Store Segment Translation Summary Flow

Figure 5-11 shows the flow used by the MMU when direct-store segment address translation is selected. This figure expands the direct-store segment translation stub found in Figure 5-6 for both instruction and data accesses. In the case of a floating-point load or store operation to a direct-store segment, other implementations may not take an alignment exception, as is allowed by the PowerPC architecture. In the case of an **eciwx**, **ecowx**, **lwarx**, or **stwcx.** instruction, the implementation either sets the DSISR register as shown and causes the DSI exception, or causes boundedly undefined results.

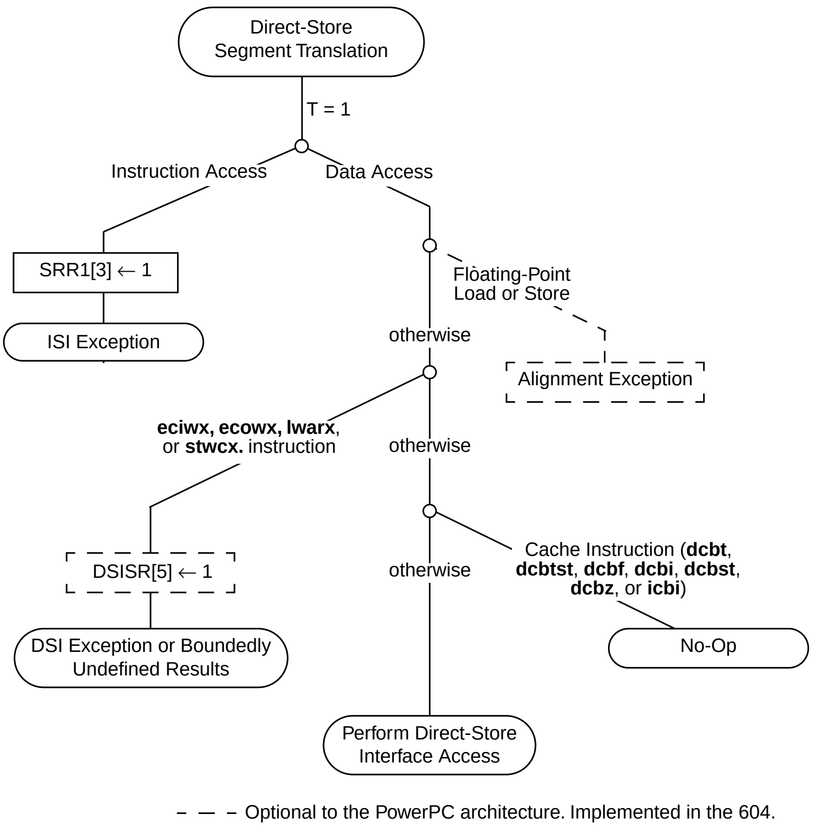

Direct-Store Segment Translation
T = 1
Instruction Access
Data Access
SRR1[3] ← 1
ISI Exception
Floating-Point Load or Store
otherwise
Alignment Exception
eciwx, ecowx, lwarx, or **stwcx.** instruction
otherwise
Cache Instruction (**dcbt**, **dcbtst**, **dcbf**, **dcbi**, **dcbst**, **dcbz**, or **icbi**)
otherwise
DSISR[5] ← 1
DSI Exception or Boundedly Undefined Results
No-Op
Perform Direct-Store Interface Access
Optional to the PowerPC architecture. Implemented in the 604.

Figure 5-11. Direct-Store Segment Translation Flow

Chapter 6
Instruction Timing

This chapter describes instruction prefetch and execution through all of the execution units of the PowerPC 604 microprocessor. It also provides examples of instruction sequences showing concurrent execution and various register dependencies to illustrate timing interactions.

6.1 Terminology and Conventions

This section describes terminology and conventions used in this chapter. This section defines terms used in this chapter.

- 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.

- 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 604 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 604 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.
- 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.
- 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 604, each execution unit has a two-entry reservation station. The 604 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 reservation stations for the other execution units are in-order reservation stations—that is, all noninteger instructions must pass through its assigned 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 604 has been designed to maximize instruction throughput and minimize average instruction execution latency. For many of the instructions in the 604, 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 cache in which case there is a single cycle required for the cache access. If the access misses in the cache, the number of additional cycles required is affected by the 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.

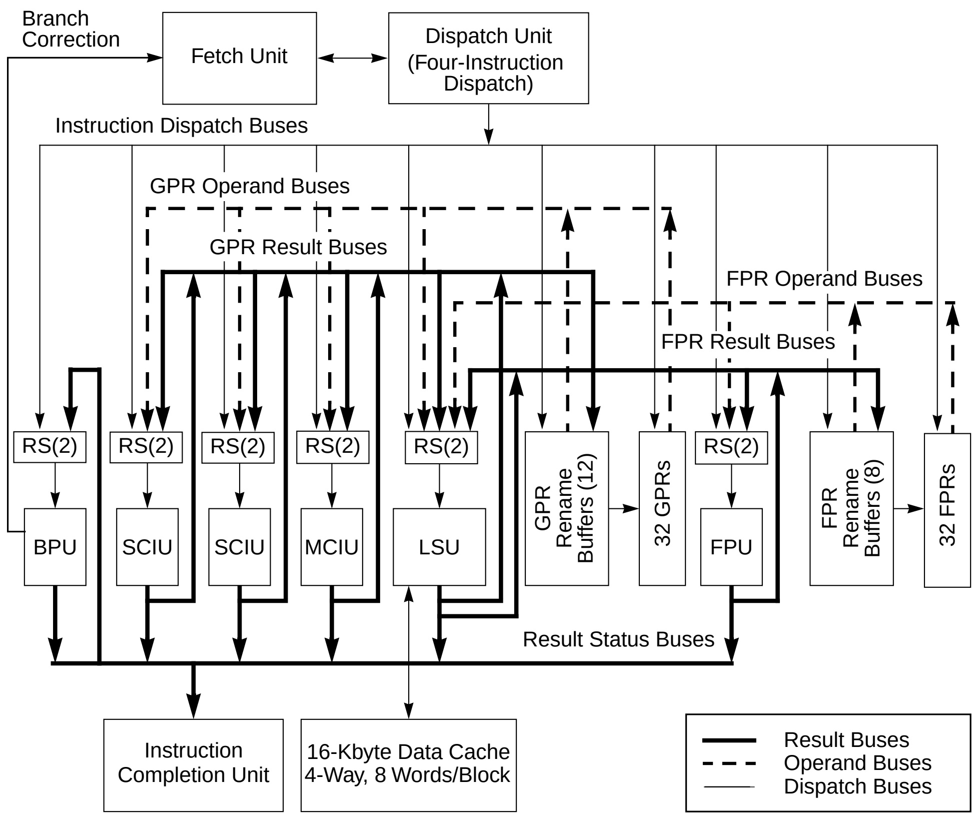


Figure 6-1. PowerPC 604 Microprocessor Block Diagram Showing Data Paths

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 604 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.

The reservation stations and result buses for the GPRs are shown in Figure 6-2

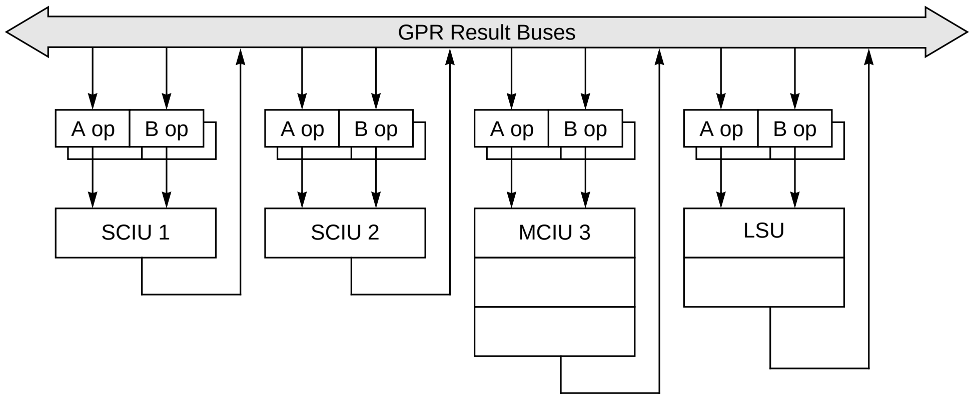


Figure 6-2. GPR Reservation Stations and Result Buses

Although it is not shown in Figure 6-1, the LSU and FPU are pipelined.

The 604’s completion buffer can retire four instructions every clock cycle. In general, instruction processing is accomplished in six stages—fetch stage, decode 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. The decode stage consists of the time it takes to fully decode the instruction. 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 master instruction pipeline of the 604 has six stages. Instructions executed by the machine flow through these stages. Some instructions combine the completion and write-back stages into a single cycle. Some instructions (load, store, and floating-point instructions) flow through additional execution pipeline stages.

The six basic stages of the master instruction pipeline are as follows:

- Fetch (IF)
- Decode (ID)
- Dispatch (DS)
- Execute (E)
- Completion (C)
- Write-back (W)

These stages are shown in Figure 6-3. Some instructions occupy multiple stages simultaneously and some individual execution units, such as the FPU and MCIU, have multiple execution stages.

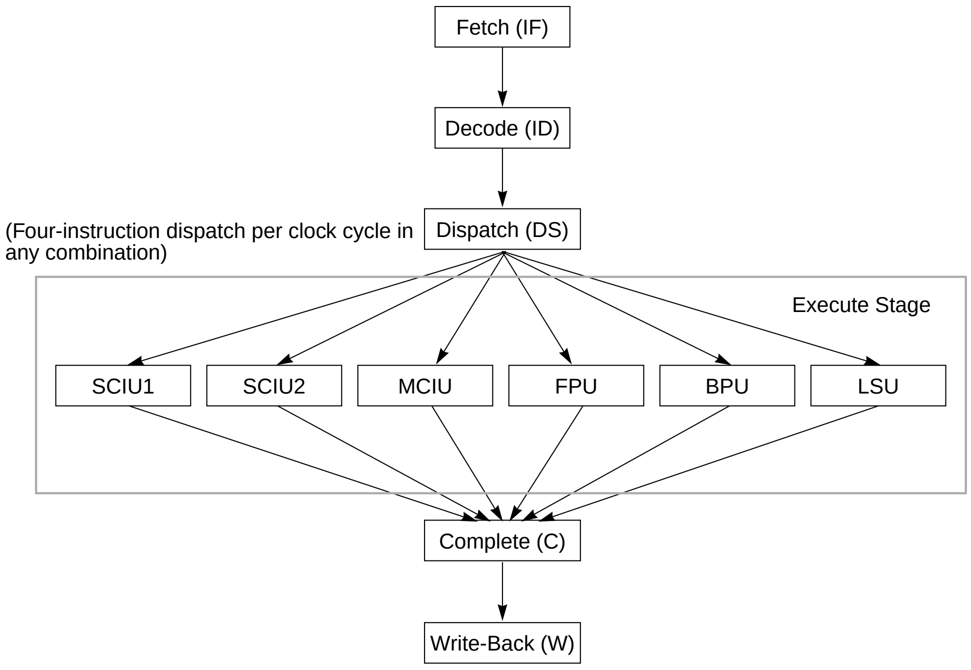


Figure 6-3. Pipeline Diagram

Pipelines for typical instructions for each of the execution units are shown in Figure 6-4. 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.

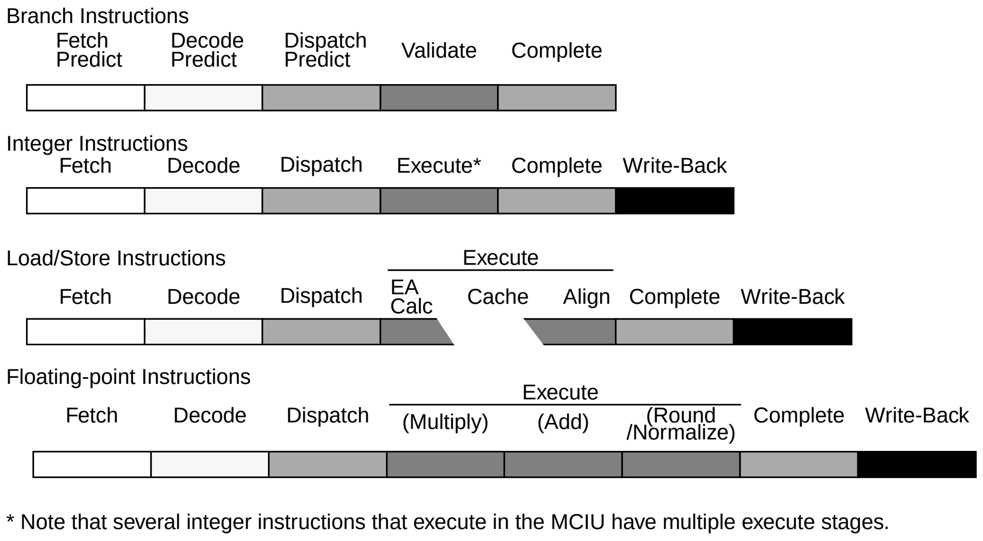


Figure 6-4. PowerPC 604 Microprocessor Pipeline Stages

Table 6-1 lists the latencies and throughputs for general groups of instructions.

Table 6-1. Execution Latencies and Throughputs

Instruction	Latency	Throughput
Most integer instructions	1	1
Integer multiply (32x32)	4	2
Integer multiply (others)	3	1
Integer divide	20	19
Integer load	2	1
Floating-point load	3	1
Floating-point store	3	1
Double-precision floating-point multiply-add	3	1
Single-precision floating-point divide	18	18
Double-precision floating-point divide	31	31

6.2.1.1 Description of Pipeline Stages

This section gives a brief description of each of the six stages of the master instruction pipeline.

6.2.1.1.1 Fetch Stage

The fetch stage primarily is responsible for fetching instructions from the instruction cache and determining the address of the next instruction to be fetched. Instructions fetched from the cache are latched into an instruction buffer for subsequent consideration by the decode stage. The instruction fetching logic is shown in Figure 6-5.

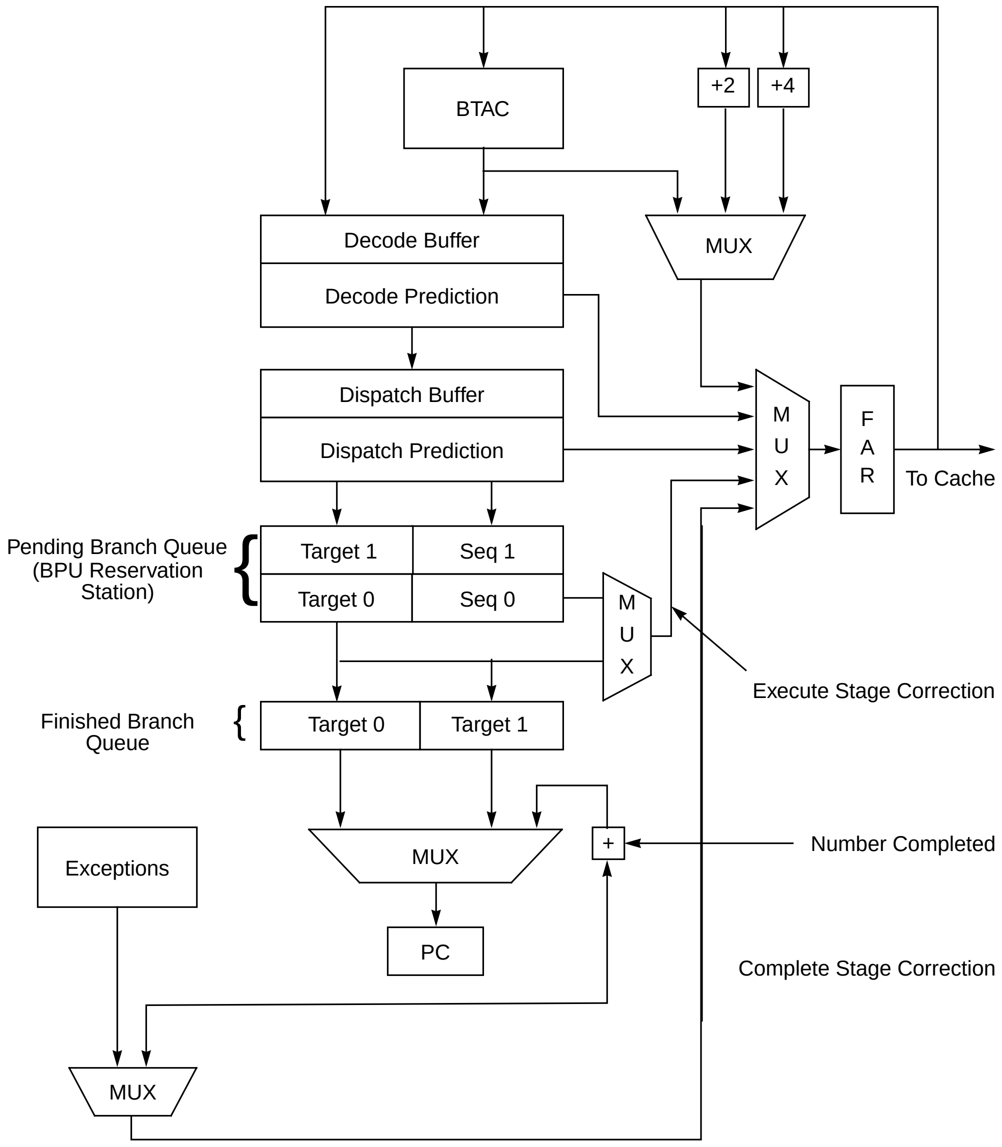


Figure 6-5. Instruction Fetch Address Generation

The fetch unit keeps the instruction buffer (four-entry decode and four-entry dispatch buffer) supplied with instructions for the dispatcher to process. Normally, the fetch unit fetches instructions sequentially, even when the instruction buffer is full because space may become available by the time the instruction cache supplies them. Instructions are fetched from the instruction cache in groups of four along double-word boundaries. Instructions can be fetched from only one cache block at a time, so if only two instructions remain in the cache block, only two instructions are fetched. If fetching is sequential, then it resumes at four instructions per clock from the next cache block.

The next address to be fetched is affected by several different conditions. Each stage offers its own candidate for the next instruction to be fetched, and the latest stage has the highest priority. As a block is prefetched, the branch target address cache (BTAC) and the branch history table (BHT) are searched with the fetch address. If the fetch address is found in the BTAC, it is the fetch stage candidate for being the next instruction address (as shown in Section 6.4.4.1.1, “Timing Example—Branch Timing for a BTAC Hit”); otherwise, the next sequential address is the candidate provided by the fetch stage.

The decode logic may indicate, based on the BHT or an unconditional branch decode, that an earlier BTAC prediction was incorrect. The BPU can indicate that a previous branch prediction, either from the BTAC or the decoder was incorrect and it can supply a new fetch address. In this case, the contents of the instruction buffers are flushed. Exception logic within the completion logic may indicate the need to vector to an exception handler address. From these choices the exception has first priority, the branch unit has second priority, the decode correction of a BTAC prediction has third priority, and the BTAC prediction has the final priority for instruction prefetching.

6.2.1.1.2 Decode Stage

The decode stage handles all time-critical instruction decoding for instructions in the instruction buffer. The decode stage contains a four-instruction buffer that shifts one or two pairs of instructions into the dispatch buffer as space becomes available.

6.2.1.1.3 Dispatch Stage

The dispatch pipeline stage is responsible for non–time-critical decoding of instructions supplied by the decode stage and for determining which of the instructions can be dispatched in the current cycle. Also, the source operands of the instructions are read from the appropriate register file and dispatched with the instruction to the execute stage. At the end of the dispatch stage, the dispatched instructions and their operands are latched into reservation stations or execution unit input latches.

6.2.1.1.4 Execute Stage

As shown in Figure 6-3, after an instruction passes through the common stages of fetch, decode, and dispatch, they are passed to the appropriate execution unit where they are said to be in execute stage. Note that the time that an instruction spends in the execute stage varies depending on the execution unit. For example, the floating-point unit has a fully-pipelined, three-stage execution unit, so most floating-point instructions have a three-cycle execute latency, regardless whether they are single- or double-precision. Some instructions, such as integer divides, must repeat some stages in order to calculate the correct result.

The execute stage executes the instruction selected in the dispatch stage, which may come from the reservation stations or from instructions arriving from dispatch. At the end of execute stage, the execution unit writes the results into the appropriate rename buffer entry, and notifies the complete stage that the instruction has finished execution.

If it is determined that the direction of a branch instruction was mispredicted in an earlier stage, the instructions from the mispredicted path are flushed and fetching resumes at the correct address.

If an instruction causes an exception, the execution unit reports the exception to the complete stage and continues executing instructions regardless of the exception. Under certain conditions, results can write directly into the register file and bypass the rename registers.

Most instructions that execute in the MCIU can finish execution and complete in the same cycle. These include the following:

- Integer divide, multiply when OE = 0
- All **mfspr**
- All **mtspr** instructions except when LR/CTR is involved

Note that all instructions that execute in the MCIU can complete during the same cycle in which they finish executing except for the following:

- Instructions that change OV or CA (OE = 1)
- Move to CTR/LR instructions because they are not execution-serialized

An example of one of these instructions, **mulli**, is shown in the instruction timing examples in Figure 6-9 through Figure 6-12. An instruction can finish execution and complete only if it is the first instruction to complete. Whether an instruction is able to complete in the same cycle in which it finishes execution is also subject to the normal considerations that affect execution and completion.

For more information about individual execution units, see Section 6.5, “Execution Unit Timings.”

6.2.1.1.5 Complete Stage

The complete stage maintains the correct architectural machine state. In doing this it considers a number of instructions residing in the completion buffer and uses the information about the status of instructions provided by the execute stage.

When instructions are dispatched, they are issued a position in the 16-entry completion buffer which they hold until they meet the constraints of completion. When an instruction finishes execution, its status is recorded in its completion buffer entry. The completion buffer is managed as a first-in, first-out (FIFO) buffer; it examines the entries in the order in which the instructions were dispatched. The fact that the completion buffer allows the processor to retain the program order ensures that instructions are completed in order.

The status of four entries are examined during each cycle to determine whether the results can be written back, and therefore, as many as four instructions can complete per clock. If an instruction causes an exception, the status information in the completion buffer reflects this, and this information in the completion buffer is used to generate the exception. In this way the completion buffer is used to ensure a precise exception model. Typically, exceptions are detected in the fetch, decode, or execute stage.

Apart from those restrictions necessary to support a precise exception model, the 604 imposes the following restrictions per each cycle:

- Completion stops before a store since store data is read directly from GPRs or FPRs
- Completion stops after a taken branch instruction to simplify the program counter logic.

Note that the 604 decouples instruction completion from the actual update (write-back) of the register file; therefore, instructions can complete regardless of how many registers they must update, and a few instructions, such as load cache misses can complete before the result is known. The write-back occurs during the complete stage if the ports and results are available; otherwise, the write-back is treated as a separate stage, as shown in the timing examples in Section 6.4.1, “General Instruction Flow.” This provision allows the processor to complete instructions, without concern for the number or presence of results. Note that if a read operation misses in the cache, the instruction can complete (as long as it is certain that the instruction can cause no exceptions) even though the result is not available.

Rename buffer entries for the FPRs, GPRs, and CR act as temporary buffers for instructions that have not completed and as write-back buffers for those that have.

Each of the rename buffers has two read ports for write-back, corresponding to the two ports provided for write-back for the GPRs, FPRs, and CR. As many as two results are copied from each write-back buffer to a register per clock cycle.

If the completion logic detects an instruction containing exception status or an instruction that can cause subsequent instructions to be flushed at completion (such as **mtspr[xer]**, instructions that set the summary overflow (SO) bit, and other instructions listed below), all following instructions are cancelled, their execution results in the rename buffers are discarded, and fetching resumes at the correct stream of instructions. Other architectural registers, such as CTR, LR, and CR, are updated during this stage. A complete list of the affected instructions is as follows:

- **mtspr (xer)**
- **mcrxr**
- **isync**
- Instructions that set the summary overflow, SO, bit
- **lswx** with 0 bytes to load
- Floating-point arithmetic, **frsp**, **fctiw**, and **fctiwz** instructions that cause an exception with FPSCR[VE] = 1
- A floating-point instruction that causes a floating-point zero divide with FPSCR(ZE = 1)

6.2.1.1.6 Write-Back Stage

The write-back stage is used to write back any information from the rename buffers that was not written back by the complete stage.

As mentioned in Section 6.2.1.1.5, “Complete Stage,” each of the rename buffers has two read ports for write-back, corresponding to the two ports provided for write-back for the GPRs, FPRs, and CR. As many as two results are copied from the write-back buffers to a register per clock cycle. To compensate for the extra write-back stage, the GPR rename buffer has 12 entries, which reduces the chances for dispatch stalls for applications that depend heavily on integer instructions.

6.3 Memory Performance Considerations

Due to the 604’s instruction throughput of four instructions per clock cycle, lack of data bandwidth can become a performance bottleneck. In order for the 604 to approach its potential performance levels, it must be able to read and write data quickly and efficiently. If there are many processors in a system environment, one processor may experience long memory latencies while another bus master (for example, a direct memory access controller) is using the external bus.

To reduce this possible contention, the PowerPC architecture provides three memory update modes—write-back, write-through, and cache-inhibit. Each page of memory is specified to be in one of these modes. If a page is in write-back mode, data being stored to that page is written only to the on-chip cache. If a page is in write-through mode, writes to that page update the on-chip cache on hits and always update main memory. If a page is cache-inhibited, data in that page is never stored in the on-chip cache. All three of these modes of operation have advantages and disadvantages. A decision as to which mode to use depends on the system environment as well as the application. Although these modes are described in detail in Chapter 3, “Cache and Bus Interface Unit Operation,” Section 6.3.4, “Memory Operations,” briefly describes how these modes may affect instruction timing.

6.3.1 MMU Overview

The 604 implements separate 128-entry, two-way set-associative TLBs, one each for instruction and data accesses. The TLBs are managed in hardware and adhere to the specifications for segmented page virtual memory provided in the operating environment architecture (OEA). The block address translation (BAT) registers make it possible to easily manage large contiguous areas of memory (128 Kbyte to 256 Mbyte).

The MMUs also control memory protection as well as the cache functions, such as whether a block or page is write-back or write-through, is cacheable/noncacheable, is kept coherent, or is available for speculative execution.

For more information about the 604 MMU implementation, see Chapter 5, “Memory Management.”

6.3.2 Cache Overview

The nonblocking data cache, shown in Figure 6-6, provides continuous load or store access during a cache block reload.

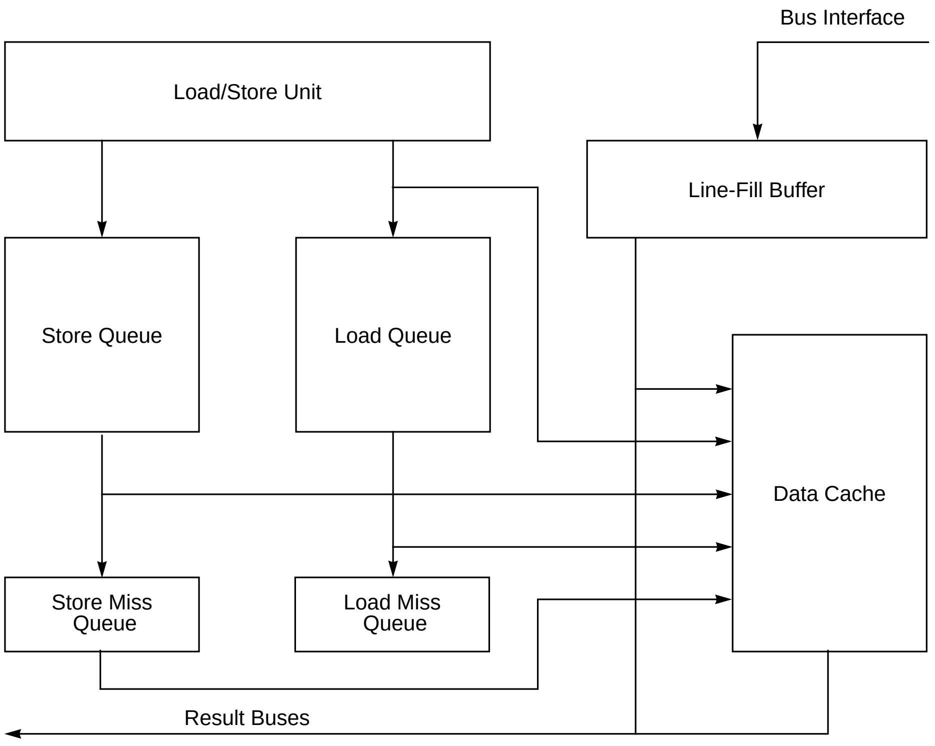


Figure 6-6. Data Caches and Memory Queues

For a load operation, the cache is accessed first by the LSU and data is forwarded to the execution unit and to the rename buffer if the access hits in the cache. Otherwise, the load operation is added to the load queue.

Store operations are added to the store queue after they are successfully translated. As each store operation is completed with respect to the execution unit, it is only marked as completed in the queue so instruction processing can continue without having to wait for the actual store operation to take place either in the cache or in system memory. When the cache is not busy, one completed store can be written to the cache per cycle. In the case of a cache miss on a store operation, that store information is placed in the store miss queue to allow subsequent store operations to continue while the missing cache block is brought in from system memory. The store queue can hold six instructions.

As each load miss completes, the cache is accessed a second time. If it misses again, the instruction is moved to the load miss register while the missing cache block is brought in. This allows a second load miss to begin without having to wait for the first one to complete. The load queue can hold as many as four instructions.

Requests from a mispredicted branch path are selectively removed from the memory queues when the misprediction is corrected, eliminating unnecessary memory accesses and reducing traffic on the system bus. The 604 also implements the cache block touch instructions (**dcbt** and **dcbtst**) which allows the processor to schedule bus activity more efficiently and increase the likelihood of a cache hit.

The data cache is kept coherent using MESI protocol and maintains a separate port so snooping does not interfere with other bus traffic. Note that coherency is not maintained in the instruction cache. Instructions are provided by the PowerPC architecture to ensure coherency in the instruction cache.

Both caches can be disabled, invalidated, or locked by using bits in the HID0 register. For more information, see Section 2.1.2.3, “Hardware Implementation-Dependent Register 0.”

For more information about the 604 cache implementation, see Chapter 3, “Cache and Bus Interface Unit Operation.”

6.3.3 Bus Interface Overview

The bus interface unit (BIU) on the 604 is compatible with that on the PowerPC 601 and 603 processors. The BIU supports both tenured and split-transaction modes and can handle as many as three outstanding pipelined operations. The BIU can complete one or more write transactions between the address and data tenures of a read transaction. The BIU provides critical double word first, so the data in the double word requested by the instruction fetcher or LSU is presented to the cache before the other data in the cache block. The critical double word is forwarded to the fetcher or to the LSU without having to wait for the entire cache block to be updated.

For more information about the BIU, see Chapter 3, “Cache and Bus Interface Unit Operation.”

6.3.4 Memory Operations

The 604 provides features that provide flexible and efficient accesses to memory in both single- and multiple-processor systems.

6.3.4.1 Write-Back Mode

When storing data while in write-back mode, store operations for cacheable data do not necessarily cause an external bus cycle to update memory. Instead, memory updates only occur on modified line replacements, cache flushes, or when another processor attempts to access a specific address for which there is a corresponding modified cache entry. For this reason, write-back mode may be preferred when external bus bandwidth is a potential bottleneck—for example, in a multiprocessor environment. Write-back mode is also well suited for data that is closely coupled to a processor, such as local variables.

If more than one device uses data stored in a page that is in write-back mode, snooping must be enabled to allow write-back operations and cache invalidations of modified data. The 604 implements snooping hardware to prevent other devices from accessing invalid data. When bus snooping is enabled, the processor monitors the transactions of the other devices. For example, if another device accesses a memory location and its memory-coherent (M) bit is set, and the 604's on-chip cache has a modified value for that address, the processor preempts the bus transaction, and updates memory with the cache data. If the cache contents associated with the snooped address are unmodified, the 604 invalidates the cache block. The other device is then free to attempt an access to the updated memory address. See Chapter 3, "Cache and Bus Interface Unit Operation," for complete information about bus snooping.

Write-back mode provides complete cache/memory coherency as well as maximizing available external bus bandwidth.

6.3.4.2 Write-Through Mode

Store operations to memory in write-through mode always update memory as well as the on-chip cache (on cache hits). Write-through mode is used when the data in the cache must always agree with external memory (for example, video memory), or when there is shared (global) data that may be used frequently, or when allocation of a cache block on a cache miss is undesirable. Cached data is not automatically written back if that data is from a memory page marked as write-through mode since valid cache data always agrees with memory.

Stores to memory that are in write-through mode may cause a decrease in performance. Each time a store is performed to memory in write-through mode, the bus remains busy for the extra clock cycles required to update memory; therefore, load operations that miss the cache must wait until the external store operation completes.

6.3.4.3 Cache-Inhibited Mode

If a memory page is specified to be cache-inhibited, data from this page is not cached.

Areas of the memory map can be cache-inhibited by the operating system software. If a cache-inhibited access hits in the on-chip cache, the corresponding cache block is invalidated. If the line is marked as modified, it is written back to memory before being invalidated.

In summary, the write-back mode allows both load and store operations to use the on-chip cache. The write-through mode allows load operations to use the on-chip cache, but store operations cause a memory access and a cache update if the data is already in the cache. Lastly, the cache-inhibited mode causes memory access for both loads and stores.

6.4 Timing Considerations

A superscalar machine is one that can issue multiple instructions concurrently from a conventional linear instruction stream. The 604 is a true superscalar implementation of the PowerPC architecture since a maximum of four instructions can be issued to the execution units during each clock cycle. Although a superscalar implementation complicates instruction timing, these complications are transparent to the functionality of software. While the 604 appears to the programmer to execute instructions in sequential order, the 604 provides increased performance by executing multiple instructions at a time, and by using hardware to manage dependencies.

When an instruction is issued, the register file places the appropriate source data on the appropriate source bus. The corresponding execution unit then reads the data from the bus. The register files and source buses have sufficient bandwidth to allow the dispatching of four instructions per clock. If an operand is unavailable, the instruction is kept in a reservation station until the operand becomes available.

The 604 contains the following execution units that operate independently and in parallel:

- Branch processing unit (BPU)
- Two 32-bit single-cycle integer units (SCIU)
- One 32-bit multiple-cycle integer units (MCIU)
- 64-bit floating-point unit (FPU)
- Load/store unit (LSU)

As shown in Figure 1-1, the BPU directs the program flow with the aid of a dynamic branch prediction mechanism. The instruction unit determines to which of the five other execution units an instruction is dispatched.

6.4.1 General Instruction Flow

When the IU or FPU finishes executing an instruction, it places the resulting data, if any, into one of the GPR, FPR, or condition register rename registers. The results are then stored into the correct register file during the write-back stage. If a subsequent instruction is waiting for this data, it is forwarded from the result buses, directly into the appropriate execution unit for the immediate execution of the waiting instruction. This allows a data-dependent instruction to be executed without waiting for the data to be written into the register file and then read back out again. This feature, known as feed forwarding, significantly shortens the time the machine may stall on data dependencies.

As many as four instructions are fetched from the instruction cache per cycle and placed in the decode buffer. After they are decoded, instructions advance to the dispatch buffers as space becomes available. The 604 tries to keep the IQ full at all times. Although four instructions can be brought in from the on-chip cache in a single clock cycle, if there is a two-instruction vacancy in the IQ, two instructions can be fetched from the cache to fill it. If while filling the IQ, the request for new instructions misses in the on-chip cache, arbitration for a memory access begins. Whenever a pair of positions opens in the queue, the next two instructions are shifted in.

6.4.2 Instruction Fetch Timing

The timing of the instruction fetch mechanism on the 604 depends heavily on the state of the on-chip cache. The speed with which the required instructions are returned to the fetcher depends on whether the instruction being asked for is in the on-chip cache (cache hit) or whether a memory transaction is required to bring the data into the cache (cache miss).

6.4.2.1 Cache Hit Timing Example

Assuming that the instruction fetcher is not blocked from the cache by a cache reload operation and the instructions it needs are in the on-chip cache (a cache hit has occurred), there will only be one clock cycle between the time that the instruction fetcher requests the instructions and the time that the instructions enter the IQ. As previously stated, instructions are fetched in pairs from a single cache block, so usually four instructions are simultaneously fetched from the on-chip cache and loaded into the IQ. If the fetch address points to the last two instructions in the instruction cache block, as is the case in Figure 6-7, only two instructions can be fetched into the IQ.

Figure 6-7 shows the timing for the following simple code sequence for instructions that use the SCIUs and the FPU:

```
and
or
fadd
fsub
addc
subfc
fmadd
fmsub
xor
neg
fadds
fsubs
add
subf
```

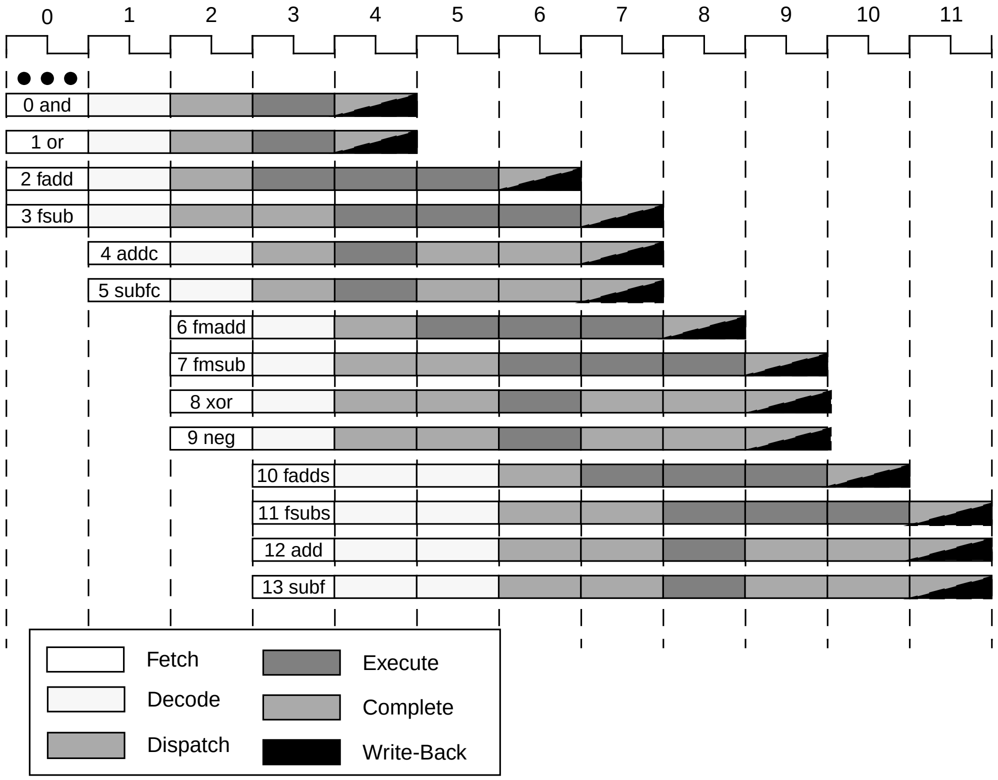


Figure 6-7. Instruction Timing—Cache Hit

The instruction timing for this example is described cycle-by-cycle as follows:

0. Two integer instructions (**and** and **or**) and two floating-point instructions (**fadd** and **fsub**) are fetched in cycle 0. These were fetched from the second double-word boundary in the instruction cache, so only two instructions can be fetched in the next clock cycle.
1. In cycle 1, the last two instructions in the cache block (**addc** and **subfc**) are fetched, while instructions 0–3 pass into the decode stage.
2. In cycle 2, the two integer add instructions (0 and 1) are dispatched, one to each of the SCIUs. The **fadd** instruction (2) is dispatched to the FPU. The **fsub** instruction cannot be dispatched, so is held in the dispatch stage until the next cycle. Instructions 4 and 5 are in the decode stage.

   Instructions 6–9 are fetched from a new cache block. Note that this is the typical, and the most efficient, alignment for instructions fetching, allowing all eight instruction in the cache block to be fetched in two cycles (four instructions per cycle).

3. The following occurs in cycle 3:
   - The first two integer instructions (**and** and **or**) enter the execute stages of the two SCIUs. The two integer instructions decoded in cycle 2 (**addc** and **subfc**) are dispatched without delay to the two SCIUs. The next pair of integer instructions (**xor** and **neg**) is in decode stage and the final pair of integer instructions (**add** and **subf**) is fetched from the second quad word in the instruction cache block.
   - The **fadd** instruction enters execute stage in the FPU, vacating the dispatch stage, allowing the **fsub** instruction to dispatch. The **fmadd** and **fmsub** instructions are in decode stage, and the final pair of floating-point instructions (**fadds** and **fsubs**) is fetched.
4. The following occurs in cycle 4:
   - In the SCIUs, the first two integer instructions complete execution and write back their results, and the second pair of integer instructions (**addc** and **subfc**) enters execute stage. The next pair of integer instructions (**xor** and **neg**) is held in the dispatch stage because the **fmsub** instruction cannot dispatch.
   - The **fadd** instruction is in the second of the three execute stages and **fsub** is in the first. The **fmadd** instruction (6) is in the dispatch stage, which forces **fmsub** to remain in the dispatch stage, similar to the situation in cycle 1 when two floating-point instructions were ready for dispatch. Note that because of in-order dispatch, the integer instructions (8 and 9) are also held in the dispatch stage behind the **fmsub** instruction. The final pair of floating-point instructions enters decode stage.
5. The following occurs in cycle 5:
   - The first two integer instructions have completed, written back their results, and vacated the pipeline. The second pair of integer instructions has executed and vacated the execution stages, but must remain in the completion buffer until the previous floating-point instructions can complete. The third pair of integer instructions is allowed to dispatch, and the final pair of integer instructions is held in the decode stage behind the previous floating-point instructions (10 and 11).
   - In the FPU, **fadd** is in the final execute stage, **fsub** is in the second stage, **fmadd** is in the first, and **fmsub** is allowed to dispatch. Because instructions 7–9 occupy the two available positions for instruction pairs in the dispatch unit, **fadds** and **fsubs** are held in decode, again, forcing subsequent integer instructions to remain in decode.
6. The following occurs in cycle 6:
   - The second pair of integer instructions (4 and 5) remains in the completion buffer waiting for the previous floating-point instructions to complete. The third pair of integer instructions is in execute stage, and the final pair of integer instructions is held in the dispatch stage behind the **fsubs** instruction.

    — In the FPU, **fadd** is in the complete and write-back stages, **fsub** is in the final execute stage, **fmadd** is in the second stage, and **fmsub** is in the first. The **fadds** instruction is in dispatch, causing the final floating-point instruction, **fsubs**, to stall in dispatch.

7. The following occurs in cycle 7:
    — Integer instructions 4 and 5 are allowed to complete and writeback because the previous **fsub** instruction completes. However, the next pair of integer instructions (8 and 9) must wait in the complete stage until **fmadd** and **fmsub** can complete. The **add** and **subf** instructions are in the dispatch stage along with the previous **fsubs** instruction.

    — The **fsub** instruction completes, allowing integer instructions 4 and 5 to complete. Floating-point instructions continue to move through the floating-point pipeline with **fmadd** in the final execute stage, **fmsub** in the second stage, and **fadds** in the first. The final floating-point instruction, **fsubs**, is allowed to dispatch.

8. The following occurs in cycle 8:
    — Integer instructions 8 and 9 continue to wait in the complete stage until **fmsub** can complete. The **add** and **subf** instructions move into execute stage along with the previous **fsubs** instruction, which is in the first stage of execute.

    — The **fmadd** instruction completes and writes back and the subsequent floating-point instructions each move to the next stage in the floating-point pipeline.

9. The following occurs in cycle 9:
    — Integer instructions 8 and 9 are allowed to complete with the **fmsub** instruction. However, the final pair of integer instructions (12 and 13) must wait in the complete stage until **fadds** and **fsubs** can complete and write back.

    — The **fmsub** instruction completes and writes back and the subsequent floating-point instructions each move to the next stage in the floating-point pipeline.

10. The following occurs in cycle 10:
    — The two remaining integer instructions remain in the complete stage until the **fsubs** instruction completes.

    — The **fadds** instruction completes and writes back and the remaining floating-point instruction, **fsubs**, is in the last execute stage in the floating-point pipeline.

11. In cycle 11 all remaining instructions complete.

Note that the double-precision floating-point add instructions each has a latency of three cycles (assuming no register dependencies) but can be fully pipelined and achieve a throughput of one floating-point instruction per clock cycle.

6.4.2.2 Cache Miss Timing Example

Figure 6-8 illustrates the timing for a cache miss using the following code sequence.

```
add
fadd
add
fadd
br
add
fsub
add
fsub
add
fadd
```

Note that this example assumes a best-case scenario.

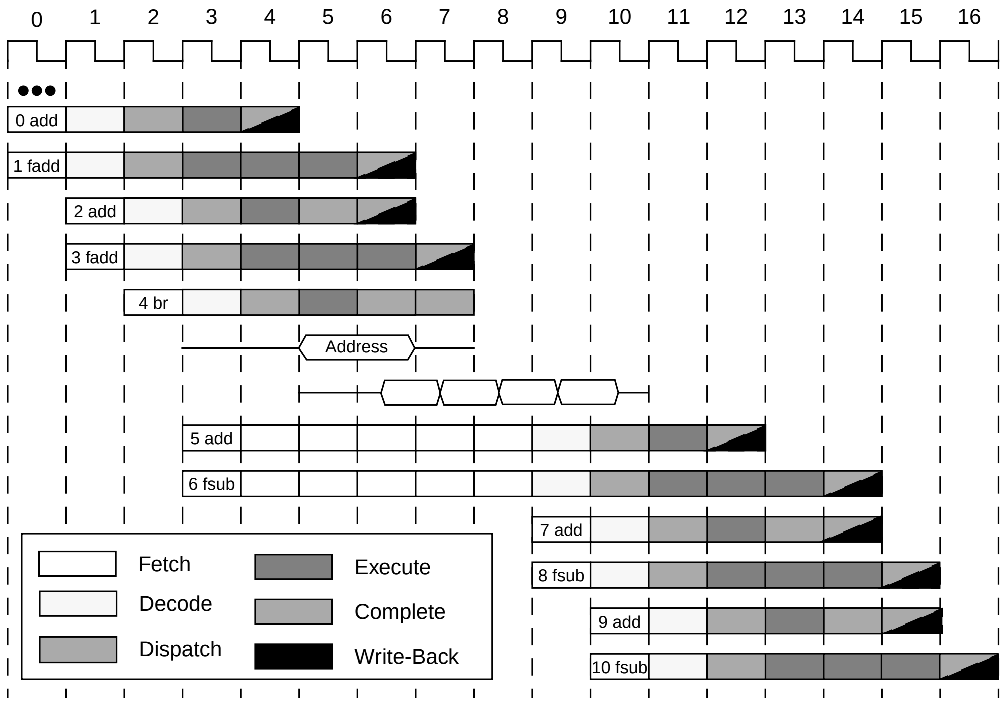


Figure 6-8. Instruction Timing—Instruction Cache Miss (BTAC Hit)

The instruction timing for this example is described cycle-by-cycle as follows:

0. In cycle 0, the first pair of **add** and **fadd** instructions is fetched.
1. In cycle 1, the second pair of **add** and **fadd** instructions is fetched as the first pair is decoded.
2. In cycle 2, the first pair of **add** and **fadd** instructions is dispatched, the second pair is decoded and the **br** instruction is fetched.
3. In cycle 3, the first pair of **add** and **fadd** instructions is in execute, the second pair is in dispatch stage, and the **br** instruction is in decode. By this time the target instruction, **add** (5) was not found in the instruction cache and arbitration for the line fill has begun.
4. In cycle 4, the first **add** instruction completes and writes back, the first **fadd** instruction is in the second execute stage, and the second pair of **add**/**fadd** instructions enter execute stage. The **br** instruction is in dispatch stage and arbitration continues for the line fill. The target instruction, **add** (5), and **fsub** remain in the fetch state.
5. In cycle 5, **fadd** (1) is in the final execute stage in the floating-point pipeline, which prevents the subsequent **add** instruction from completing and writing back. The second **fadd** instruction is in the second cycle of the floating-point execute stage and the **br** instruction is in execute stage. During this cycle, the address for the target instruction is on the address bus and access has been granted for the data bus.
6. In cycle 6, **fadd** (1) completes and writes back, allowing the **add** (2) instruction to complete and write back. The **fadd** (3) instruction is in the final execute stage and the **br** instruction is in complete stage. The first beat of the four-beat burst (which contains the critical double word) is sent over the data bus.
7. In cycle 7, **fadd** (3) completes and writes back, allowing the **br** instruction to complete. The second beat of the burst transfer begins on the data bus.
8. In cycle 8, the two instructions in the critical double word transferred in cycles 6 and 7 (**add** (5) and **fsub** (6)) are placed in the instruction queue. All previous instructions have vacated the completion buffer.
9. In cycle 9, **add** (5) and **fsub** (6) are in decode stage and the pair of instructions loaded in the second beat of the data burst (**add** (7) and **fsub** (8)) are fetched. Note that although there is room in the instruction queue for as many as four instructions, only instructions 7 and 8 are available.
10. In cycle 10, instructions 5 and 6 are in dispatch stage, instructions 7 and 8 are in decode stage, and the third pair of instructions are fetched. The fourth pair of instructions are sent in the fourth and final beat of the four-beat data burst.
11. In the remaining clock cycles, the instructions shown complete processing similarly to instructions 0–3. Note again that although the integer instructions **add** (7) and **add** (9) complete, they cannot write back until the previous floating-point instructions **fsub** (6) and **fsub** (8) write back.

6.4.3 Cache Arbitration

When a cache miss occurs, a line-fill operation is initiated to update the appropriate cache block. When the double word containing the data at the specified address (the critical double word) is available, it is forwarded to the cache and made available to other resources on the 604. Likewise, subsequent double words are also forwarded as they reach the memory unit.

Fetches to different lines can hit in the cache during the line-fill operation; however, if a miss occurs before the cache block has been updated, the line-fill operation must complete before the line-fill operation caused by the subsequent miss can begin.

For more information about the cache implementation in the 604, see Chapter 3, "Cache and Bus Interface Unit Operation."

6.4.4 Branch Prediction

The 604 implements several features to reduce the latencies caused by handling branch instructions. In particular, it provides a means of dynamic branch prediction. This is especially critical for the 604 to take fullest advantage of the possibilities of increased throughput made available from its pipelined and highly parallel organization. Dynamic branch prediction is implemented in the fetch, decode, and dispatch stages, as described in the following:

In the fetch stage, the fetch address is used to access the branch target address cache (BTAC), which contains the target address of previously executed branch instructions that are predicted to be taken. The 64-entry BTAC is fully associative to provide a high hit percentage. If a fetch address is in the BTAC, the target address is used in the next cycle to fetch the instructions from the predicted path. If the address is not present, sequential instruction flow is assumed and the appropriate sequential address is generated based on the number of instructions added to the decode buffer. The fetch address, rather than the first branch address, is sufficient to access the BTAC, since a BTAC entry contains the first predicted taken branch beyond the current fetch address.

In the decode and dispatch stages, the first branch instruction is identified and its outcome is predicted. For an unconditional branch instruction, the instruction prefetch is redirected to the target address if this branch was predicted as not taken by a previous stage. Conditional instructions whose direction depends on the value in the CTR are predicted based on that value. If the prediction differs from the current branch prediction, the prefetch is redirected.

For conditional branch instructions that depend only on a bit in the CR, the BHT is used for the prediction. The BHT is a 512-entry, direct-mapped cache with 2 bits that can indicate four prediction states—strongly taken, taken, not-taken, and strongly not-taken. The entry is updated each time a conditional branch instruction that depends on a bit in the condition register is executed. For example, a BHT entry that predicts "taken" is updated to "strongly taken" after the branch is taken or is updated to "not-taken" if the next branch is not-taken.

6.4.4.1 Branch Timing Examples

This section shows how the timing of a branch is affected depending upon whether the branch hits in the BTAC, or whether correction is required in one of the stages. The following examples use the following code sequence:

```
and
ld
add
bc
or
cmp
ld
mulli
```

6.4.4.1.1 Timing Example—Branch Timing for a BTAC Hit

Figure 6-9 shows the timing for a branch instruction that had a BTAC hit.

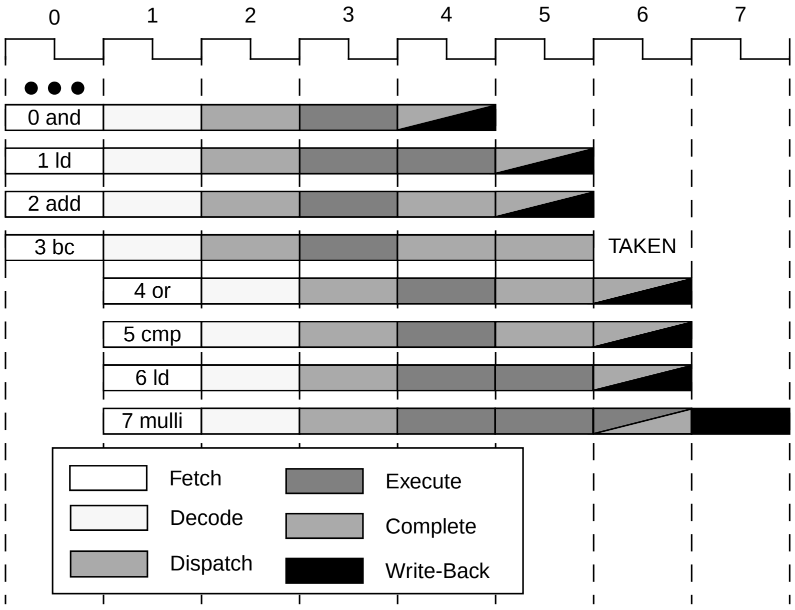


Figure 6-9. Instruction Timing—Branch with BTAC Hit

The timing for this example is described, cycle-by-cycle, as follows:

0. In clock cycle 0, instructions 0–3 are fetched. The target instruction of the **bc** instruction is found in the BTAC.
1. In cycle 1, instructions 0–3 are decoded and instructions 4–7, using the address in the BTAC, are fetched.
2. In cycle 2, instructions 0–3 are dispatched and instructions 4–7 are decoded.
3. In cycle 3, instructions 0–3 are in the execute stage and instructions 4–7 are in the dispatch stage.
4. In cycle 4, instructions 0, 2, and 3 are in the complete stage, but only instruction 0 is allowed to complete and write back because the **ld** instruction (1) is still in the execute stage of the LSU pipeline. Instructions 2and 3 wait in the complete stage. Instructions 4–7 all enter the execute stage.
5. In cycle 5, the **ld** (1) instruction is able to complete and write back, allowing the **add** instruction to write back and vacate the pipeline in the next cycle. The **br** instruction also completes. Because the branch is taken, the **or** (4) instruction, which could otherwise write back in this cycle, stays in the complete stage and completes and writes back in the next cycle. The **cmp** (5) instruction also enters the complete stage; **ld** (6) and **mulli** (7) enter the second stages of the LSU and MCIU pipelines, respectively.
6. In cycle 6, instructions 4–6 complete and write back their results. The **mulli** instruction, which is one of the instructions that can complete and write back during its final cycle in the execute stage, occupies the execute and complete stages, but cannot write back because both GPR write-back ports are occupied by the **or** and **ld** instructions.
7. The **mulli** instruction writes back its results.

6.4.4.1.2 Timing Example—Branch with BTAC Miss/Decode Correction

In the example shown in Figure 6-10, the branch target address is not found in the BTAC during the fetch cycle of the **bc** instruction, as was the case in Figure 6-9. This one-cycle delay causes the second group of instructions to be executed one cycle later than if there is a BTAC hit.

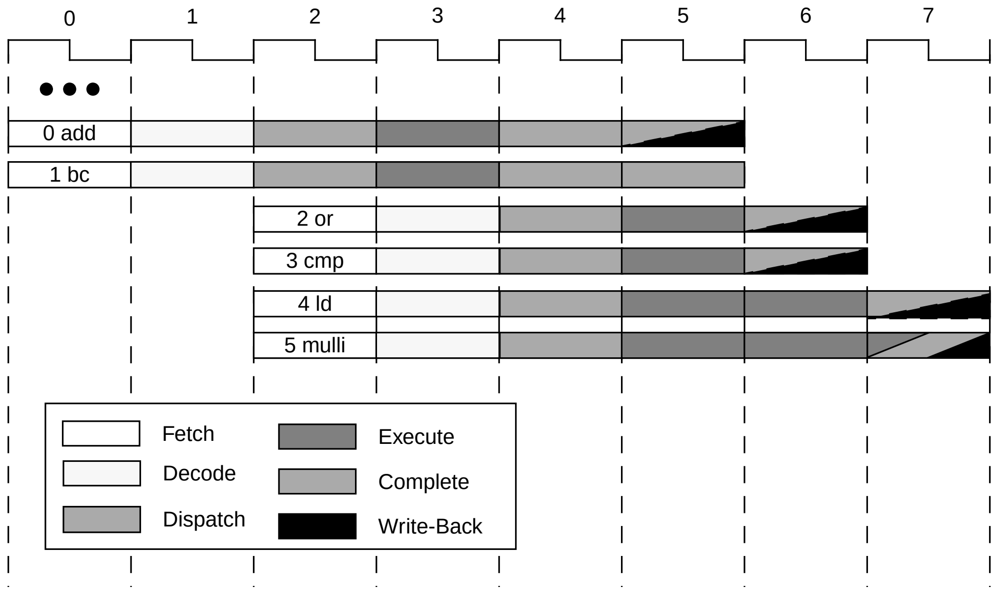


Figure 6-10. Instruction Timing—Branch with BTAC Miss/Decode Correction

A cycle-by-cycle description of this example is as follows:

0. In cycle 2, instructions 0–3 are dispatched and instructions 4–7 are fetched.
1. In cycle 3, instructions 0–3 are in the execute stage and instructions 4–7 are in the decode stage.
2. In cycle 4, instructions 0, 2, and 3 complete, but only instruction 0 is allowed to write back, because the **ld** instruction (1) is still in the execute stage of the LSU pipeline. Instructions 2 and 3 wait in the complete stage. Instructions 4–7 enter the dispatch stage.
3. In cycle 5, the **ld** (1) instruction is able to write back, allowing the following **add** instruction (which completed in the previous cycle) to write back and vacate the pipeline in the next cycle. Instructions 4–7 are in the execute stage.
4. In cycle 6, the **or** and **cmp** (5) instructions complete and write back; **ld** (6) and **mulli** (7) enter the second stages of the LSU and MCIU execute pipelines, respectively.
5. In cycle 7, the **ld** (6) instruction completes and writes back its results. The **mulli** instruction finishes executing, completes, and writes back its results. Note that the **mulli** instruction is able to complete in the same cycle as the **ld** instruction because, unlike in the previous example, the two GPR write-back ports are available.

6.4.4.1.3 Timing Example—Branch with BTAC Miss/Dispatch Correction

Figure 6-11 uses the same code sequence as the example shown in Figure 6-9, and shows the timing when the BTAC miss is corrected in the dispatch stage. The timing in this example is identical to that in Figure 6-10, except that the timings for instructions 4–7 are shifted over by one cycle.

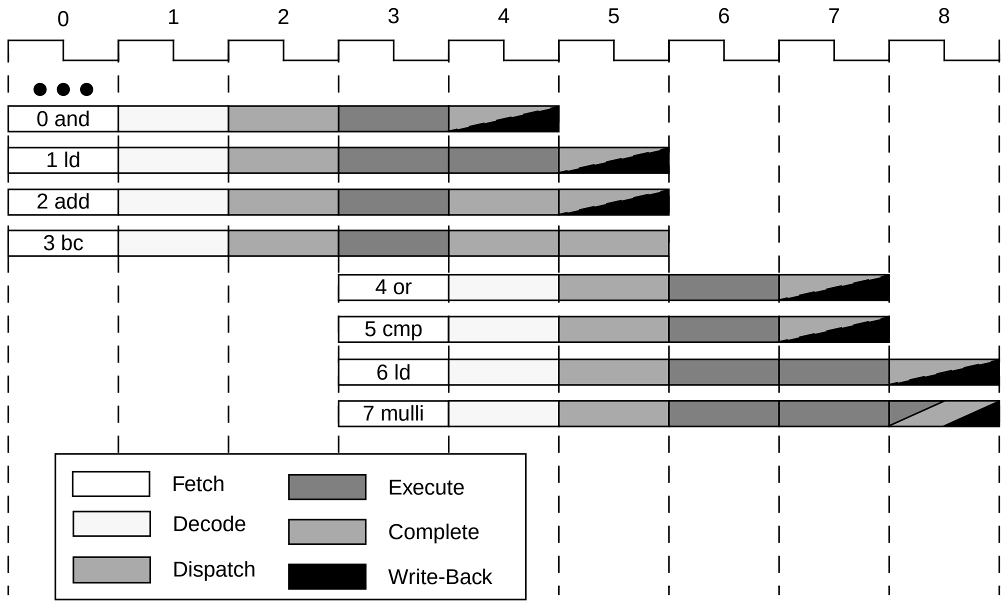


Figure 6-11. Instruction Timing—Branch with BTAC Miss/Dispatch Correction

6.4.4.1.4 Timing Example—Branch with BTAC Miss/Execute Correction

Figure 6-12 uses the same code sequence as the previous examples, and shows the timing when the BTAC miss is corrected in the execute stage. The timing in this example is identical to that in Figure 6-10, except that the timings for instructions 4–7 are shifted over by two cycles (and over one cycle when compared to the timing when correction is provided in the dispatch stage, as shown in Figure 6-11).

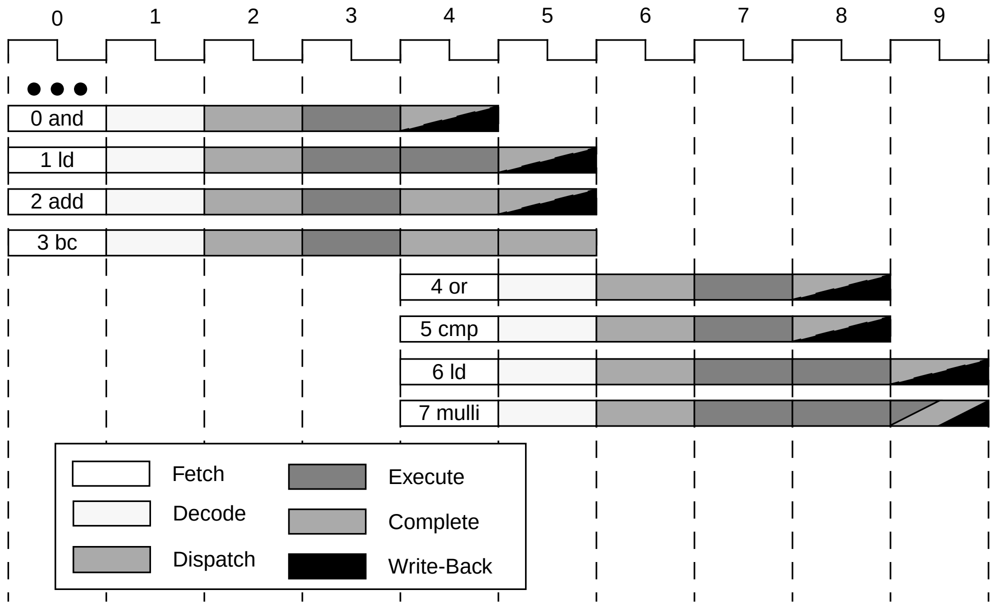


Figure 6-12. Instruction Timing—Branch with BTAC Miss/Execute Correction

6.4.5 Speculative Execution

To take fullest advantage of pipelining and parallelism, the 604 speculatively executes instructions along a predicted path until the branch is resolved. The 604 can handle as many as four dispatched, uncompleted branch instructions (with four more in the instruction queue) and can execute instructions from the predicted path of two unresolved branch instructions. The results of speculatively executed instructions (the predicted state) are kept in temporary locations, such as rename buffers, the completion buffer, and various shadow registers. Architecturally defined resources are updated only after a branch is resolved.

To record the predicted state, the 604 uses many of the same resources (primarily the rename buffers and completion buffer) and logic as the mechanism used to maintain a precise exception model, as is common among superscalar implementations. The 604 design avoids the performance degradation that may come from such a design due to speculative execution of longer latency instructions, by implementing additional logic to record the predicted state whenever a predicted branch instruction is dispatched. This allows the state to be quickly recovered when the branch prediction is incorrect. The recording of these predicted states makes it possible to identify and selectively remove instructions from the mispredicted path.

A shadow register is used with the CTR and LR to accelerate instructions that access these registers. Shadow registers are updated and the old value is saved whenever a branch instruction is dispatched, even if it is from a predicted path for a branch that has not yet been resolved. If the prediction is correct, there is no penalty. If the prediction is incorrect, shadow registers are restored from the saved values so instructions fetched from the correct path can be dispatched and executed. When the branch instruction completes, architected registers are updated.

6.4.6 Instruction Dispatch and Completion Considerations

The 604's ability to dispatch instructions at a peak rate of four per cycle is affected by availability of such resources as execution units, destination rename registers, and completion buffer entries. To avoid dispatch unit stalls due to instruction data dependencies, each execution unit has two reservation stations. If a data dependency could prevent an instruction from beginning execution, that instruction is dispatched to the reservation station associated with its execution unit, clearing the dispatch unit. When the data that the operation depends upon is returned via a cache access or as a result of a previous operation, execution begins during the cycle after the rename register is updated. If the second instruction in the dispatch unit requires the same execution unit, that instruction is not dispatched until the first instruction completes execution.

Instructions are dispatched to reservation stations in order, but from the perspective of the overall program flow, instructions can execute out of order. The following aspects of the 604's support for out-of-order execution should be noted:

- The BPU, FPU, and LSU each have two-entry in-order reservation stations. These stations allow instructions to clear the dispatch stage even though operands may not yet be available for execution to occur. The BPU, FPU, and LSU instructions may execute out of order with respect to one another and to other execution units, but the BPU, FPU, and LSU instructions pass through their respective reservation stations and pipelines in program order.
- Each integer unit has a two-entry out-of-order reservation station which allows integer instructions to execute out-of-order within each execution as well as with respect to instructions in other execution units.

The completion unit can track instructions from dispatch through execution and ensure that they are completed in program order. In-order completion ensures the correct architectural state when the 604 must recover from a mispredicted branch, or any other exception or interrupt.

The rate of instruction completion is unaffected by the 604's ability to write the instruction results from the rename registers to the architecturally defined registers when the instruction is retired. The 604 can perform two write-back operations from each of the rename registers to the register files (CR, GPRs, and FPRs) each clock cycle.

Due to the 604's out-of-order execution capability, the in-order completion of instructions by the completion unit provides a precise exception mechanism. All program-related exceptions are signaled when the instruction causing the exception has reached the last position in the completion buffer. All prior instructions are allowed to complete and write back before the exception is taken.

6.4.6.1 Rename Register Operation

To avoid contention for a given register file location in the course of out-of-order execution, the 604 provides rename registers for the storage of instruction results prior to their commitment (in program order) to the architecturally defined register by the completion unit. Register renaming minimizes architectural resource dependencies, namely the output and antidependencies, that would otherwise limit opportunities for out-of-order execution. Twelve rename registers are provided for the GPRs, eight for the FPRs, and eight for the condition register.

A GPR rename buffer entry is allocated when an instruction that modifies a GPR is dispatched. This entry is marked as allocated but not valid. When the instruction executes, it writes its result to the entry and sets the valid bit. When the instruction completes, its result is copied from the rename buffer entry to the GPR and the entry is freed for reallocation. For load with update instructions that modify two GPRs, one for load data and another for address, two rename buffer entries are allocated.

The rename register for the GPRs is shown in Figure 6-13.

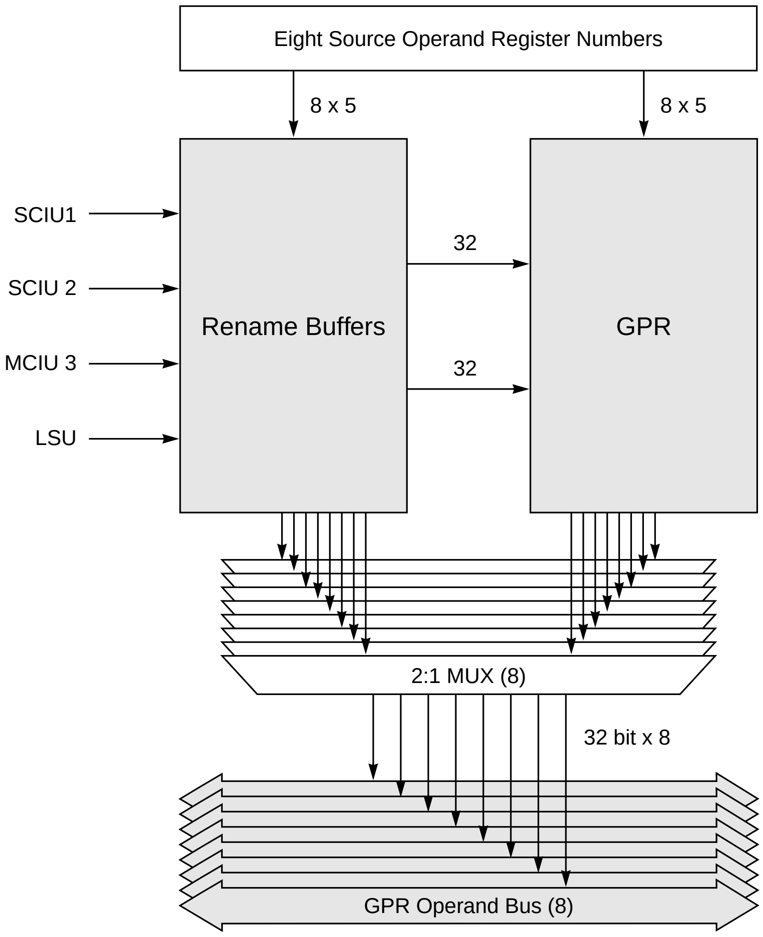


Figure 6-13. GPR Rename Register

When an integer instruction is dispatched, its source operands are searched simultaneously from the GPR file and its rename buffer. If a value is found in the rename buffer, that value is used; otherwise, the value is read from the GPR. However, the rename buffer entry may not yet be valid if the instruction that updates the GPR has not yet executed. In this case, the instruction is dispatched with the rename buffer entry identifier in place of the operand, which will be supplied by the reservation station when the result is produced. The GPR file and its rename buffer have eight read ports for source operands to support dispatching of four integer instructions each cycle.

The FPR file has 32 registers of 64 bits wide and an eight-entry rename buffer. The FPR file and its rename buffer have three read ports for three source operands, which allow one floating-point instruction to be dispatched per cycle.

The 604 treats each of the 4-bit fields in the condition register as a register and applies register renaming for each with an eight-entry rename buffer.

Along with the reorder buffer, the rename buffers provide the basis of the precise exception mechanism, because the 604's architectural state represents, at all times, the results of instructions completed in program order. Precise exceptions greatly simplify the exception model by allowing the appearance of serialized execution.

6.4.6.2 Execution Unit Considerations

As previously noted, the 604 is capable of dispatching and retiring four instructions per clock cycle. One of the factors affecting the peak dispatch rate is the availability of execution units on each clock cycle.

For an instruction to be issued, the required reservation station must be available. The dispatcher monitors the availability of all execution units and suspends instruction dispatch if the required reservation station is not available. An execution unit may not be available if it can accept and execute only one instruction per cycle, or if an execution unit's pipeline becomes full. This situation may occur if instruction execution takes more clock cycles than the number of pipeline stages in the unit, and additional instructions are issued to that unit to fill the remaining pipeline stages.

6.4.7 Instruction Serialization

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 604 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-storage accesses are handled in the same way to ensure that exceptions are precise.

The performance is not degraded since instructions following a serializing instruction are dispatched and executed usually before the serializing instruction is executed. One serialized instruction can complete per clock cycle.

The following sections describe the serialization modes.

6.4.7.1 Dispatch Serialization Mode

Dispatch serialization occurs when an **mtspr** instruction that accesses either the counter or link or a **mtcrf** instruction that accesses multiple bits is dispatched to the MCIU. In these instances, an interlock is set so that no other such instructions or branch unit instructions (branch and CR logical) can dispatch until the original instruction executes and clears the interlock. The interlock is cleared when the instruction that sets the interlock finishes executing. On the next cycle the instruction that is waiting can dispatch.

6.4.7.2 Execution Serialization Mode

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 nonserialization instruction is that the execution serialization instruction cannot be executed until it is the oldest uncompleted instruction in the processor. In other words, the instruction is dispatched into a reservation station, but cannot be executed until the completion block informs the execution unit to execute the instruction. This means it is guaranteed to wait at least one cycle before it can execute.

Instructions causing execution serialization include the following:

- Condition register logical operations (**crand**, **crandc**, **creqv**, **crnand**, **crnor**, **cror**, **crorc**, **crxor**, and **mcrf**)
- **mfspr** and **mfmsr**
- **mtspr** (except count and link registers) and **mtmsr**
- Instructions that use the carry bit (**adde**, **addeo**, **subfe**, **subfeo**, **addme**, **addmeo**, **subfme**, **subfmeo**, **addze**, **addzeo**, **subfze**, and **subfzeo**)

6.4.7.3 Postdispatch Serialization Mode

Postdispatch serialization occurs when the serializing instruction is being completed. All instructions following the postdispatch serialized instruction are flushed, refetched, and re-executed. Instructions causing postdispatch serialization include the following:

- **mtspr (xer)**
- **mcrxr**
- **isync**
- Instructions that set the summary overflow, SO, bit
- **lswx** with 0 bytes to load
- Floating-point arithmetic, **frsp**, **fctiw**, and **fctiwz** instructions that cause an exception with FPSCR[VE] = 1
- Floating-point instructions with the Rc (record bit) set
- FPSCR instructions—**mtfsb0**, **mtfsb1**, **mtfsfi**, **mffs**, **mtfsf**, and **mcrfs**
- A floating-point instruction that causes a floating-point zero divide with FPSCR(ZE = 1)

6.4.7.4 Serialization of String/Multiple Instructions

Serialization is required for all load/store multiple/string instructions. These instructions are broken into a sequence of register-aligned operations. The first operation is dispatched along with any preceding instructions in the dispatch buffer. Subsequent operations are dispatched one-word-per-cycle until the operation is finished. String/multiple instructions remain in the dispatch buffer for at least two cycles even if they only require a single-word–aligned memory operation.

Instructions causing string/multiple serialization include **lmw**, **stmw**, **lswi**, **lswx**, **stswi**, and **stswx.**

6.4.7.5 Serialization of Input/Output

In this serialization mode, all noncacheable loads are performed in order with respect to the **eieio** instruction.

6.5 Execution Unit Timings

The following sections describe instruction timing considerations within each of the respective execution units in the 604. Refer to Table 6-2 for branch instruction execution timing.

6.5.1 Branch Unit Instruction Timings

The 604 can have two unresolved branches in the branch reservation station and two resolved branches that have not yet completed. The branch unit serves to validate branch predictions made in earlier stages. It also verifies that the predicted target matches the actual target address. If a misprediction is detected, it redirects the fetch to the correct address and starts the branch misprediction recovery.

The branch execution unit also executes condition register logical instructions, which the PowerPC architecture provides for calculating complex branch conditions. Other architectures that lack such instructions would need to use a series of branch instructions to resolve complex branching conditions. All execution units can update the CR fields, but only the branch and CR logical operations use CR fields as source operands.

6.5.2 Integer Unit Instruction Timings

The two SCIUs and the MCIU execute all integer and bit-field instructions, and are shown in Figure 6-14 and Figure 6-15, respectively.

The SCIUs consist of three one-cycle subunits:

- A fast adder/comparator subunit
- A logic subunit
- A rotator/shifter/count-leading zero subunit

These subunits handle all of the one-cycle arithmetic instructions. Only one subunit in each SCIU can obtain and execute an instruction at a time.

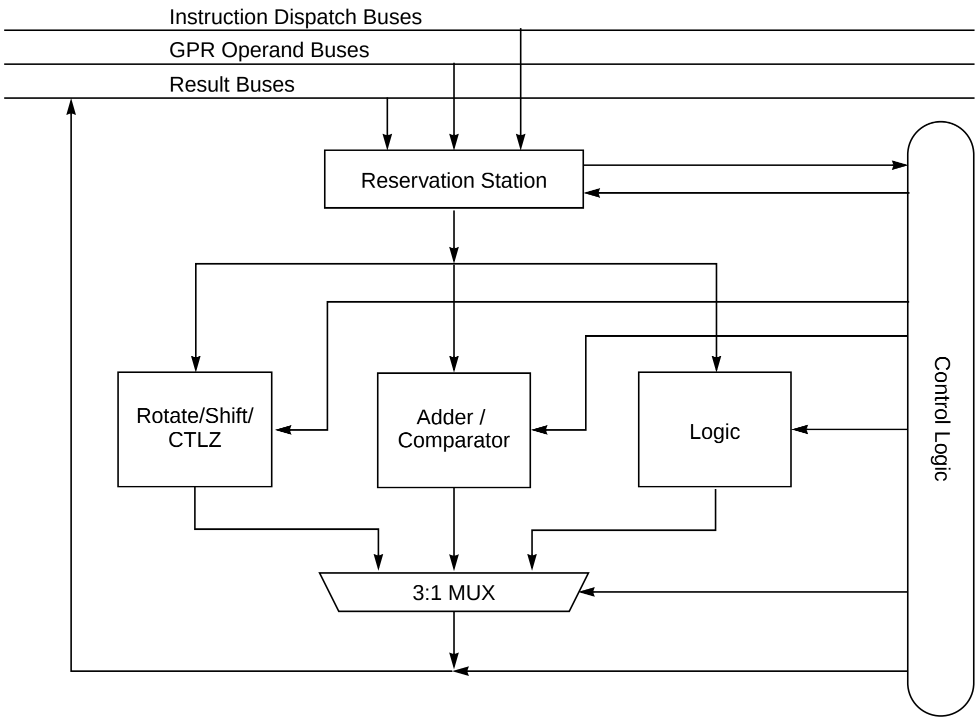


Figure 6-14. SCIU Block Diagram

The MCIU, which handles all integer multiple-cycle integer instructions, consists of a 32-bit integer multiplier/divider subunit. The multiplier supports early exit on 32 x 16-bit operations. In addition the MCIU executes all **mfspr** and **mtspr** instructions.

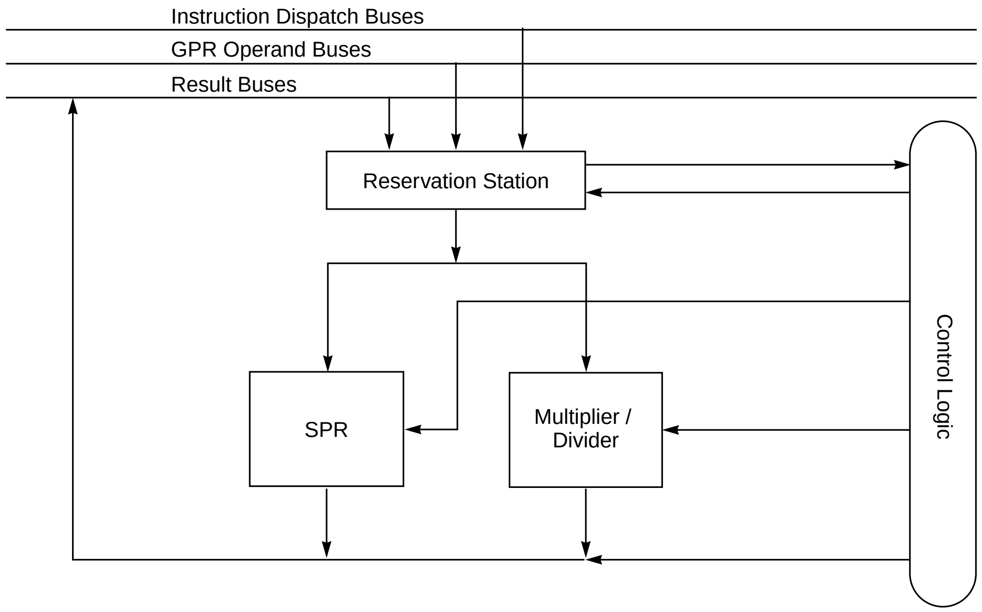


Figure 6-15. MCIU Block Diagram

Most instructions that execute in the MCIU can finish execution and complete in the same cycle. These include the following:

- Integer divide, multiply when OE = 0
- All **mfspr** instructions
- All **mtspr** instructions except when LR/CTR is involved

Note that all instructions that execute in the MCIU can complete during the same cycle in which they finish executing except for the following:

- Instruction that changes OV or CA (OE = 1)
- The move to CTR/LR instructions cannot because they are not execution-serialized

6.5.3 Floating-Point Unit Instruction Timings

The floating-point unit on the 604 executes all floating-point instructions. Execution of most floating-point instructions is pipelined within the FPU, allowing up to three instructions to be executing in the FPU concurrently. While most floating-point instructions execute with three-cycle latency and one-cycle throughput, three instructions (**fdivs**, **fdiv**, and **fres**) execute with latencies of 18 to 33 cycles. The **fdivs**, **fdiv**, **fres**, **mtfsb0**, **mtfsb1**, **mtfsfi**, **mffs**, and **mtfsf** instructions block the floating-point pipeline until they complete execution and thereby inhibit the execution of additional floating-point instructions. With the exception of the **mcrfs** instruction, all floating-point instructions immediately forward their CR results to the BPU for fast branch resolution without waiting for the instruction to be retired by the completion unit and the CR to be updated. Refer to Table 6-2 for floating-point instruction execution timing.

As shown in Figure 6-16, The FPU on the 604 is a single-pass, double-precision unit. This means that both single- and double-precision floating-point operations require one-pass/one-cycle throughput with a latency of three cycles. This hardware implementation supports the IEEE 754-1985 standard for floating-point arithmetic, including support for the NaNs and denormalized data types.

Instructions are obtained from the instruction dispatcher and placed in the reservation station queue. The operand sources are the FPR, the floating-point rename buffers, and the result buses. The result of an FPU operation is written to the floating-point rename buffers and to the reservation stations. Instructions are executed from the reservation station queue in the order they were originally dispatched.

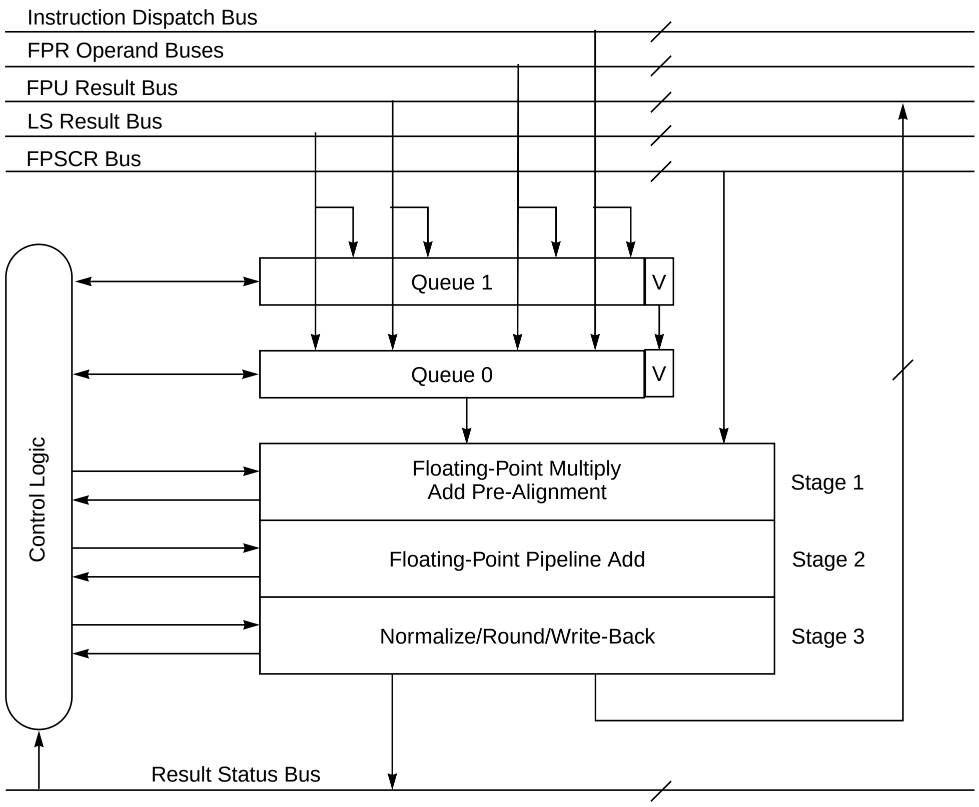


Figure 6-16. FPU Block Diagram

6.5.4 Load/Store Unit Instruction Timings

The execution of most load and store instructions is pipelined. The LSU has two pipeline stages; the first stage is for effective address calculation, and MMU translation, and the second stage is for accessing the data in the cache. Load instructions have a two-cycle latency and one-cycle throughput, and store instructions have a two-cycle latency and single-cycle throughput.

The primary function of the LSU is to transfer 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. As shown in Table 6-2, the LSU also executes special instructions such as string transfers and cache control.

To improve execution performance, the LSU allows a load operation to be executed ahead of pending store operations. All data dependencies introduced by this out-of-order execution are resolved by the LSU. These dependencies arise when, in the instruction stream, a store is followed by a load from the same address. If the load instruction is speculatively executed before the store has modified the cache, incorrect data is loaded into the rename registers. If the low-order 12 bits of the effective addresses are equal, the two effective addresses may be aliases for the same physical address, in which case the load instruction waits until the store data is written back to the cache, guaranteeing that the load operation retrieves the correct data.

The LSU provides hardware support for denormalization of floating-point numbers. Within the 604, all floating-point numbers are represented as double-precision numbers. Denormalization can occur during a store floating-point single instruction, when the double-precision number is converted to a single-precision number.

A block diagram of the load/store unit is shown in Figure 6-17. The unit is composed of: reservation stations, an address calculation block, data alignment blocks, load queues, and store queues.

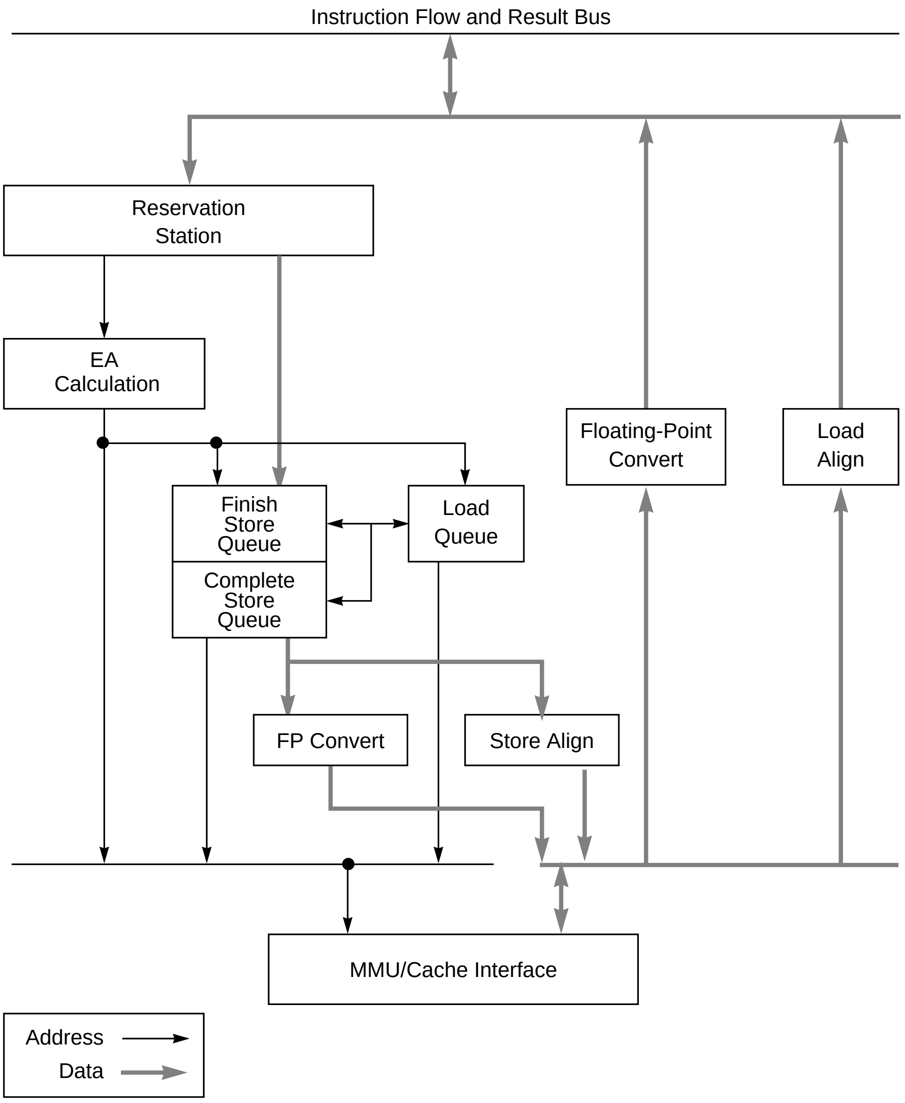


Figure 6-17. LSU Block Diagram

The reservation stations are used as temporary storage of dispatched instructions that cannot be executed until all of the instruction operands are valid. The address calculation block includes a 32-bit adder that computes the effective address for all operations. The data alignment blocks manage the necessary byte manipulations to support aligned or unaligned data transfers to and from the data cache. The load and store queues are used for temporary storage of instructions for which the effective addresses have been translated and are waiting to be completed by the sequencer unit.

Figure 6-18 shows the structure of the store queue. There are four regions that identify the state of the store instructions.

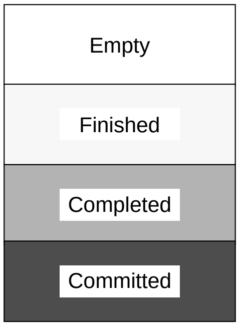


Figure 6-18. Store Queue Structure

When a store instruction finishes execution, it is placed in the finished state. When it is completed, the finish pointer advances to place it in the completed state. When the store data is committed to memory, the completion pointer advances to place it in the committed state. If the store operation hits in the cache, the commit pointer advances to effectively remove the instruction from the queue. Otherwise, the commit pointer does not advance until the cache block is reloaded and the store operation can occur. During this time, the next store instruction pointed to by the completion pointer can access the cache. If this second store instruction hits in the cache, it is removed from the queue. If not, another cache block reload begins.

6.5.5 isync, rfi, and sc Instruction Timings

The **isync**, **rfi**, and **sc** instructions do not execute in one of the execution units. These instructions decode to branch unit instructions, as specified by the PowerPC architecture, but they do not actually execute in the BPU in the same sense that other branch instructions do. The completion unit treats the **rfi** and **sc** instructions as exceptions, and handles them precisely. When an **isync** instruction reaches the top of the completion buffer, subsequent instructions are flushed from the pipeline and are refetched during the next clock cycle.

Although the **rfi** and **sc** are dispatched to the branch reservation stations, these instructions do not execute in the ordinary sense, and do not occupy a position in an execute stage in one of the BPU. Instead, these instructions are given a position in the completion buffer at dispatch. When the **sc** instruction reaches the top of the completion buffer, the system call exception is taken. When the **rfi** instruction reaches the top of the completion buffer, the necessary operations required for restoring the machine state upon returning from an exception are performed.

The **isync** instruction causes instructions to be flushed when it is completed. This means that the decode buffers, dispatch buffers, and execution pipeline are all flushed. Fetching resumes from the instruction following the **isync**.

6.6 Instruction Scheduling Guidelines

The performance of the 604 can be improved by avoiding resource conflicts and promoting parallel utilization of execution units through efficient instruction scheduling. Instruction scheduling on the 604 can be improved by observing the following guidelines:

- Schedule instructions such that they can maximize the dispatch rate.
- Schedule instructions to minimize execution-unit-busy stalls
- Avoid using serializing instructions
- Schedule instructions to avoid dispatch stalls due to renamed resource limitations

6.6.1 Instruction Dispatch Rules

The following list provides limitations on instruction dispatch that should be kept in mind in order to ensure stalls:

- At most, four instructions can be dispatched per cycle.
- 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 SCIUs are identical and either can be used to execute any integer arithmetic, logical, shift/rotate, trap, and **mtcrf** instructions that update only one field.
  - The MCIU executes all integer multiply, divide and move to/from instructions except **mtcrf** instructions that update only one field, which are executed in either of the SCIUs.
  - The load/store unit executes load, store, and cache control instructions
  - The FPU executes all floating-point instructions including move to/from FPSCR

  Table 6-2 indicates which execution unit executes each instruction.
- Each instruction must have an entry in the 16-entry reorder buffer. The dispatch unit stalls when the reorder buffer is full. Reorder buffer entries become available on the cycle after the instruction has completed.
- An instruction that modifies a GPR is assigned one of the 12 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 one cycle after the result is written to the GPR.
- Any floating-point instruction except **mcrfs**, **mtfsfi**, **mtfsfi**., **mtfsf**, **mtfsf**., **mtfsb0**, **mtfsb0.**, **mtfsb1**, and **mtfsb1.** gets one entry in the eight-entry FPR rename buffer. When the FPR rename buffer is full, dispatch stalls on the next floating-point instruction. A rename buffer entry can become available one cycle after the result is written to the FPR.

- The eight-entry CR rename buffer is similar to the GPR rename buffer in that an instruction that modifies a CR field gets one entry. This includes, for example, all condition register logical instructions and **mtcrf** instructions that update only one CR field. When the CR rename buffer is full, dispatch stalls when the next instruction to be dispatched needs a CR entry. A rename buffer entry becomes available one cycle after the result is written to the CR.
- Each execution unit has a two-entry reservation station that holds instructions until they are ready for execution. Instructions cannot be dispatched if the reservation station is full.
- No following instruction can dispatch in the same cycle as a branch instruction.
- Since instructions are dispatched in program order, a later instruction cannot be dispatched until all earlier ones have.
- There is an interlock mechanism between CTR and LR. After dispatching a move to CTR/LR or **mtcrf** with multiple field update, the dispatch stalls on the first branch, CR logical, move to CTR/LR, or **mtcrf** that update multiple fields until one cycle after the dispatched move to CTR/LR or **mtcrf** instruction executes. Those **mtcrf** instructions that update multiple fields are execution-serialized.
- The 604 can handle as many as four branch instructions in the execute and complete stages. The dispatch stalls on the first instruction after the fourth branch until the first branch completes.
- An instruction cannot be dispatched until all destination registers for the instruction have been assigned to a rename register.
- An instruction may not be dispatched if a serialization mode is in effect for the instruction.

6.6.2 Additional Programming Tips for the PowerPC 604 Processor

The following guidelines should be followed when writing assembly code for the 604.

- **Interleave memory instructions with integer and floating-point operations.**

  The 604 has a dedicated LSU that does not require the use of the integer or floating-point units to process memory operations. As a result, when scheduling code for the 604, interleaving memory operations with integer or floating-point instructions typically result in better performance.

- **Interleave integer operations.**

  Because the 604 has three IUs, it is also possible to interleave multiple, independent integer operations. Two of these integer units support simple integer operations, while the third supports complex integer operations such as bit-field manipulation.

- **Avoid using instructions that write to multiple registers.**

  The 604's dynamic register renaming permits instructions to execute out of order with respect to their original program sequence, which increases overall throughput. However, in other PowerPC processors, certain instructions including the load/store multiple/string operations, monopolize these internal hardware resources, which can affect performance. For software portability, such instructions should be avoided, even though they do not suffer the performance degradation in the 604 that they might in other PowerPC processors. The most common use of such instructions is in subroutine prologues or epilogues The following alternatives are typically more efficient:

  — Expanding the register save/restore code in-line
  — Branching to special save/restore functions (sometimes called millicode) that use in-line sequences of save and restore instructions.

- **Use the load with update instruction judiciously**.

  Another frequently used set of instructions that are subject to this multiple register usage effect are the load with update instructions. While use of such instructions is usually desirable from a performance standpoint (they eliminate a dependent integer operation), care must still be taken to not issue too many of these instructions consecutively.

- **Schedule code to take advantage of rename registers**.

  As discussed previously, the 604 provides register renaming as a means of improving execution speed. Since there are a limited number of rename buffers implemented in hardware, it is always desirable to minimize pressure on this resource. One relatively simple means of doing this is to use immediate addressing when the option exists. For example, an integer register copy can be performed in a single cycle using a number of different instructions. However, using an **ori** instruction (with an immediate operand of zero) uses only one source register operand; whereas, the register indirect form of the **or** instruction uses two source registers.

- **Minimize use of instructions that serialize execution.**

  Some operations, such as memory synchronization primitives and trap instructions, have well-known serialization properties that are intended when used by a programmer. Other instructions, however, have more subtle serialization effects that may affect performance. For example, if operations that manipulate condition register fields are used frequently, they can significantly hinder performance, particularly when multiple condition fields are being accessed by a single instruction, described in the following:

- **Avoid using the mtcrf instruction to update multiple fields.**

  Note that the performance of the **mtcrf** instruction depends greatly on whether only one field is 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 set 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.

- **Minimize branching.**

  The 604 supports dynamic branch prediction and other mechanisms that reduce the impact of branching; nevertheless, changing control flow in a program is relatively expensive, in that fullest advantage cannot be taken of resources that can improve throughput. such as superscalar instruction dispatch and execution. In some cases, branches can be minimized by simply rewriting an algorithm. In other cases, special PowerPC instructions, such as **fsel**, can be used to eliminate a conditional branch altogether.

- Note that the **fsel** instruction is optional to the PowerPC architecture and may not be implemented on all PowerPC implementations, so use of this instruction to improve performance in the 604 should be weighed against portability considerations.

6.7 Instruction Latency Summary

Table 6-2 summarizes the execution cycle time of each instruction. Note that the latencies themselves provide limited insight as to the actual behavior of an instruction. The following list summarizes some aspects of instruction behavior:

- For a store operation, availability means data is visible to the following loads from the same address. Misaligned load or store operations require one additional cycle, assuming cache hits.

  — Floating-point stores that require denormalization take an additional cycle for each bit of shifting that is needed up to a maximum of 23.

  — Store multiple instructions are taken in pairs and take one additional cycle if an odd number of registers is stored.

- — Misaligned load string operations require two cycles per register plus two additional cycles.
- — Misaligned store string operations take six cycles per register being stored (although the final store may only take three cycles if it does not cross a word boundary).

- For instructions with both a CR result and either a GPR or an FPR result, the cycle count shown is for the GPR or FPR result. CR results from logical or bit field instructions that execute in the SCIU and CR results from instructions that execute in the FPU take one additional cycle.
- Integer multiplies that detect an early exit condition finish a cycle earlier than others. For signed multiplies, if the top 15 bits of the RB operand are all the same it is an early out condition. For unsigned multiplies, if the top 15 bits are all zeros it is an early out condition.
- All instructions are fully pipelined except for divides and some integer multiplies. The integer multiplier is a three-stage pipeline. Integer multiplies other than those that can exit early (described in the previous bullet) stall for one cycle in the first stage of the pipeline. Integer divide instructions iterate in stage two of the multiplier. Special-purpose register operations can execute in the MCIU in parallel with multiplies and divides.
  - — The FPU unit is a three-stage pipeline. Floating-point divides iterate in the floating-point pipeline. The floating-point unit also has some data-dependent delays not shown inTable 6-2. If the rounder has a carry out, that is, 1.11...111 rounds to 2.00...000, the FPU takes an additional cycle. If the final normalization of the result requires a shift of more than 63, the FPU takes an additional cycle. Underflow and overflow take an additional cycle. Denormalization to zero takes an additional cycle. Massive cancellation resulting in zero takes an additional cycle.

Table 6-2. Instruction Execution Timing

Instruction	Unit	Cycle (cycle)	Serialization
add	SCIU	1	—
addc	SCIU	1	—
adde	SCIU	1	Execute
addi	SCIU	1	—
addic	SCIU	1	—
addic.	SCIU	1	—
addis	SCIU	1	—
addme	SCIU	1	Execute
addze	SCIU	1	Execute
and	SCIU	1	—

Table 6-2. Instruction Execution Timing (Continued)

Instruction	Unit	Cycle (cycle)	Serialization
andc	SCIU	1	—
andi.	SCIU	1	—
andis.	SCIU	1	—
b	BPU	1	—
bc	BPU	1	—
bcctr	BPU	1	—
bclr	BPU	1	—
cmp	SCIU	1	—
cmpi	SCIU	1	—
cmpl	SCIU	1	—
cmpli	SCIU	1	—
cntlzw	SCIU	1	—
crand	BPU	1	Execute
crandc	BPU	1	Execute
creqv	BPU	1	Execute
crnand	BPU	1	Execute
crnor	BPU	1	Execute
cror	BPU	1	Execute
crorc	BPU	1	Execute
crxor	BPU	1	Execute
dcbf	LSU	—	Execute
dcbi	LSU	3	Execute
dcbst	LSU	—	Execute
dcbt	LSU	—	Execute
dcbtst	LSU	—	Execute
dcbz	LSU	3	Execute
divw	MCIU	20	—
divwu	MCIU	20	—
eciwx	LSU	2 + bus	Execute
ecowx	LSU	3 + bus	Execute
eieio	LSU	—	I/O
eqv	SCIU	1	—

Table 6-2. Instruction Execution Timing (Continued)

Instruction	Unit	Cycle (cycle)	Serialization
extsb	SCIU	1	—
extsh	SCIU	1	—
fabs	FPU	3	—
fadd	FPU	3	—
fadds	FPU	3	—
fcmpo	FPU	3	—
fcmpu	FPU	3	—
fctiw	FPU	3	—
fctiwz	FPU	3	—
fdiv	FPU	32	FP empty[1]
fdivs	FPU	18	FP empty[1]
fmadd	FPU	3	—
fmadds	FPU	3	—
fmr	FPU	3	—
fmsub	FPU	3	—
fmsubs	FPU	3	—
fmul	FPU	3	—
fmuls	FPU	3	—
fnabs	FPU	3	—
fneg	FPU	3	—
fnmadd	FPU	3	—
fnmadds	FPU	3	—
fnmsub	FPU	3	—
fnmsubs	FPU	3	—
fres	FPU	18	FP empty[1]
frsp	FPU	3	—
frsqrte	FPU	3	—
fsel	FPU	3	—
fsub	FPU	3	—
fsubs	FPU	3	—
icbi	LSU	—	—
isync	Completion	1	Postdispatch

Table 6-2. Instruction Execution Timing (Continued)

Instruction	Unit	Cycle (cycle)	Serialization
lbz	LSU	2	—
lbzu	LSU	2	—
lbzux	LSU	2	—
lbzx	LSU	2	—
lfd	LSU	3	—
lfdu	LSU	3	—
lfdux	LSU	3	—
lfdx	LSU	3	—
lfs	LSU	3	—
lfsu	LSU	3	—
lfsux	LSU	3	—
lfsx	LSU	3	—
lha	LSU	2	—
lhau	LSU	2	—
lhaux	LSU	2	—
lhax	LSU	2	—
lhbrx	LSU	2	—
lhz	LSU	2	—
lhzu	LSU	2	—
lhzux	LSU	2	—
lhzx	LSU	2	—
lmw	LSU	#regs + 2	String/multiple
lswi	LSU	2(#regs) + 2	String/multiple
lswx	LSU	2(#regs) + 2	String/multiple
lwarx	LSU	3+bus	Execute
lwbrx	LSU	2	—
lwz	LSU	2	—
lwzu	LSU	2	—
lwzux	LSU	2	—
lwzx	LSU	2	—
mcrf	BPU	1	Execute
mcrfs	FPU	3	—

Table 6-2. Instruction Execution Timing (Continued)

Instruction	Unit	Cycle (cycle)	Serialization
mcrxr	MCIU	3	Execute
mfcr	MCIU	3	Execute
mffs	FPU	3	—
mfmsr	MCIU	3	Execute
mftb	MCIU	3	Execute
mfspr LR/CTR	MCIU	3	Execute
mfspr (others)	MCIU	3	Execute
mtcrf (0/multiple bit)	MCIU	1	Dispatch/Execute
mtcrf (single bit)	SCIU	1	—
mtfsb0	FPU	3	—
mtfsb1	FPU	3	—
mtfsf	FPU	3	—
mtfsfi	FPU	3	—
mtmsr	MCIU	1	Execute
mtspr (LR/CTR)	MCIU	1	Dispatch
mtspr (XER)	MCIU	1	Complete [2]
mtspr (others)	MCIU	1	Execute
mulhw	MCIU	4(3)	—
mulhwu	MCIU	4(3)	—
mulli	MCIU	3	—
mullw	MCIU	4(3)	—
nand	SCIU	1	—
neg	SCIU	1	—
nor	SCIU	1	—
or	SCIU	1	—
orc	SCIU	1	—
ori	SCIU	1	—
oris	SCIU	1	—
rfi	Completion	—	Postdispatch
rlwimi	SCIU	1	—
rlwinm	SCIU	1	—
rlwnm	SCIU	1	—

Table 6-2. Instruction Execution Timing (Continued)

Instruction	Unit	Cycle (cycle)	Serialization
sc	Completion	—	Postdispatch
slw	SCIU	1	—
sraw	SCIU	1	—
srawi	SCIU	1	—
srw	SCIU	1	—
stb	LSU	3	Execute
stbu	LSU	3	Execute
stbux	LSU	3	Execute
stbx	LSU	3	Execute
stfd	LSU	3	Execute
stfdu	LSU	3	Execute
stfdux	LSU	3	Execute
stfdx	LSU	3	Execute
stfiwx	LSU	3	Execute
stfs	LSU	3	Execute
stfsu	LSU	3	Execute
stfsux	LSU	3	Execute
stfsx	LSU	3	Execute
sth	LSU	3	Execute
sthbrx	LSU	3	Execute
sthu	LSU	3	Execute
sthux	LSU	3	Execute
sthx	LSU	3	Execute
stmw	LSU	#regs + 2	String/multiple
stswi	LSU	#regs + 2	String/multiple
stswx	LSU	#regs + 2	String/multiple
stw	LSU	3	Execute
stwbrx	LSU	3	Execute
stwcx.	LSU	3	Execute
stwu	LSU	3	Execute
stwux	LSU	3	Execute
stwx	LSU	3	Execute

Table 6-2. Instruction Execution Timing (Continued)

Instruction	Unit	Cycle (cycle)	Serialization
subf	SCIU	1	—
subfc	SCIU	1	—
subfe	SCIU	1	Execute
subfic	SCIU	1	—
subfme	SCIU	1	Execute
subfze	SCIU	1	Execute
sync	LSU	—	—
tlbie	LSU	—	Execute
tlbsync	LSU	—	—
tw	SCIU	1	—
twi	SCIU	1	—
xor	SCIU	1	—
xori	SCIU	1	—
xoris	SCIU	1	—

[1] These instructions are not pipelined. They cannot be executed until the previous instruction in the FPU completes; subsequent FPU instructions cannot begin execution until these instructions complete.

[2] The **mtspr** (XER) instruction causes instructions to be flushed when it executes.

Chapter 7
Signal Descriptions

This chapter describes the PowerPC 604 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, $\overline{\text{ARTRY}}$ (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 AP0–AP3 (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.

The 604 signals are grouped as follows:

- Address arbitration signals—The 604 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 cache-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 604 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 external 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.
- JTAG/COP interface signals—The JTAG (IEEE 1149.1) interface and common on-chip processor (COP) unit provides a serial interface to the system for performing monitoring and boundary tests.
- Processor configuration signals—These signals include the memory reservation signal, machine quiesce control signals, time base enable signal, driver mode signal, L2 intervention signal, the run and halted signals, and the analog VDD signal.
- Clock signals—These signals provide for system clock input and frequency control.

7.1 Signal Configuration

Figure 7-1 illustrates the pin configuration of the 604, showing how the signals are grouped.

NOTE

A pinout showing actual pin numbers is included in the 604 hardware specifications.

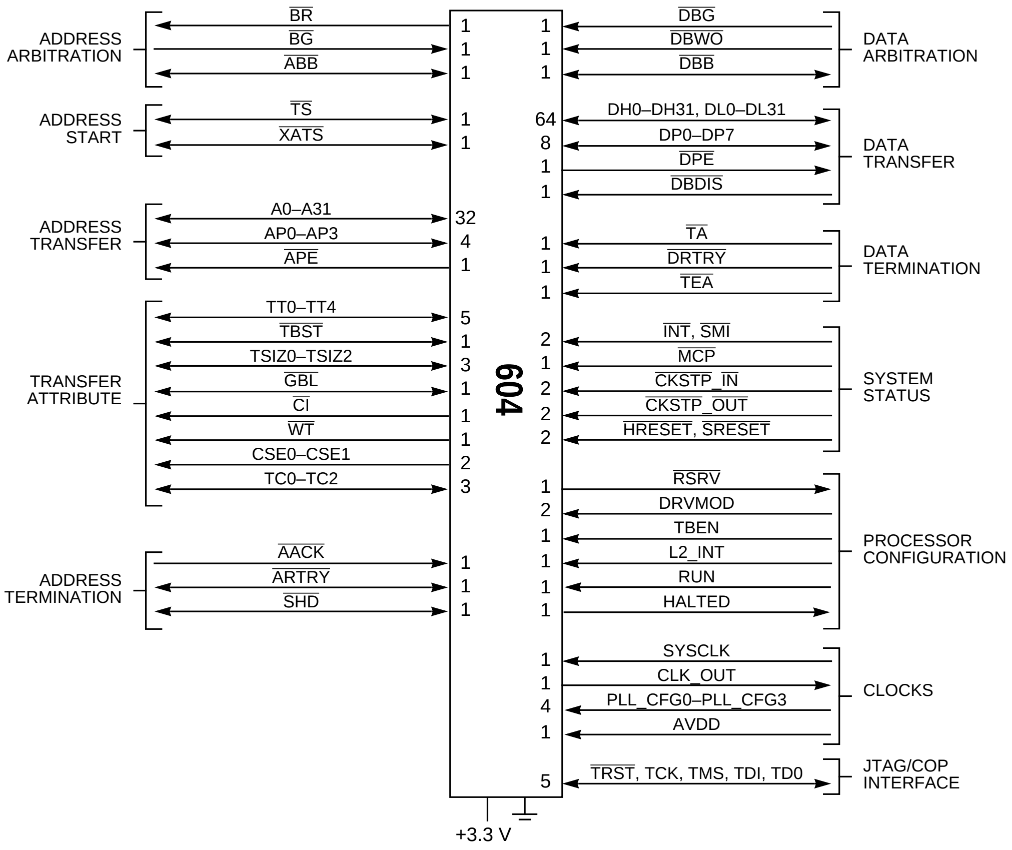


Figure 7-1. PowerPC 604 Microprocessor Signal Groups

7.2 Signal Descriptions

This section describes individual 604 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 604 uses to request the address bus, recognize when the request is granted, and indicate to other devices when mastership is granted. For a detailed description of how these signals interact, see Section 8.3.1, “Address Bus Arbitration.”

7.2.1.1 Bus Request ($\overline{\text{BR}}$)—Output

The bus request ($\overline{\text{BR}}$) signal is an output signal on the 604. Following are the state meaning and timing comments for the $\overline{\text{BR}}$ signal.

State Meaning Asserted—Indicates that the 604 is requesting mastership of the address bus. Note that $\overline{\text{BR}}$ may be asserted for one or more cycles, and then deasserted due to an internal cancellation of the bus request (for example, due to the loss of a memory reservation). See Section 8.3.1, “Address Bus Arbitration.”

Negated—Indicates that the 604 is not requesting the address bus. The 604 may have no bus operation pending, it may be parked, or the $\overline{\text{ARTRY}}$ input was asserted on the previous bus clock cycle.

Timing Comments Assertion—Occurs when a bus transaction is needed and the 604 does not have a qualified bus grant. This may occur even if the three possible pipeline accesses have occurred.

Negation—Occurs for at least one bus clock cycle after an accepted, qualified bus grant (see $\overline{\text{BG}}$ and $\overline{\text{ABB}}$), even if another transaction is pending. It is also negated for at least one bus clock cycle when the assertion of $\overline{\text{ARTRY}}$ is detected on the bus, with the exception of the bus master that asserted $\overline{\text{ARTRY}}$ due to the need to perform a cache line push.

7.2.1.2 Bus Grant ($\overline{\text{BG}}$)—Input

The bus grant ($\overline{\text{BG}}$) signal is an input signal on the 604. Following are the state meaning and timing comments for the $\overline{\text{BG}}$ signal.

State Meaning Asserted—Indicates that the 604 may, with the proper qualification, assume mastership of the address bus. A qualified bus grant occurs when $\overline{\text{BG}}$ is asserted, $\overline{\text{ABB}}$ and $\overline{\text{ARTRY}}$ are not asserted, and $\overline{\text{ARTRY}}$ has been negated on the previous cycle. The $\overline{\text{ABB}}$ and $\overline{\text{ARTRY}}$ signals are driven by the 604 or other bus masters. If the 604 is parked, $\overline{\text{BR}}$ need not be asserted for the qualified bus grant. See Section 8.3.1, “Address Bus Arbitration.”

Negated— Indicates that the 604 is not the next potential address bus master.

Timing Comments Assertion—May occur at any time to indicate the 604 is free to use the address bus. After the 604 assumes bus mastership, it does not check for a qualified bus grant again until the cycle during which the address bus tenure is completed (assuming it has another transaction to run). The 604 does not accept a $\overline{\text{BG}}$ in the cycles between the assertion of any $\overline{\text{TS}}$ or $\overline{\text{XATS}}$ through to the assertion of $\overline{\text{AACK}}$.

Negation—May occur at any time to indicate the 604 cannot use the bus. The 604 may still assume bus mastership on the bus clock cycle of the negation of $\overline{\text{BG}}$ because during the previous cycle $\overline{\text{BG}}$ indicated to the 604 that it was free to take mastership (if qualified).

7.2.1.3 Address Bus Busy ($\overline{\text{ABB}}$)

The address bus busy ($\overline{\text{ABB}}$) signal is both an input and an output signal.

7.2.1.3.1 Address Bus Busy ($\overline{\text{ABB}}$)—Output

Following are the state meaning and timing comments for the $\overline{\text{ABB}}$ output signal.

State Meaning Asserted—Indicates that the 604 is the address bus master. See Section 8.3.1, “Address Bus Arbitration.”

Negated—Indicates that the 604 is not using the address bus. If $\overline{\text{ABB}}$ is negated during the bus clock cycle following a qualified bus grant, the 604 did not accept mastership, even if $\overline{\text{BR}}$ was asserted. This can occur if a potential transaction is aborted internally before the transaction is started.

Timing Comments Assertion—Occurs on the bus clock cycle following a qualified $\overline{\text{BG}}$ that is accepted by the processor (see Negated).

Negation—Occurs on the bus clock cycle following the assertion of $\overline{\text{AACK}}$. If $\overline{\text{ABB}}$ is negated during the bus clock cycle following a qualified bus grant, the 604 did not accept mastership, even if $\overline{\text{BR}}$ was asserted.

High Impedance—Occurs one-half bus cycle (two-thirds bus cycle when using 3:1 clock mode, and one-third bus cycle when using 3:2 bus ratio) after $\overline{\text{ABB}}$ is negated.

7.2.1.3.2 Address Bus Busy ($\overline{\text{ABB}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{ABB}}$ input signal.

State Meaning Asserted—Indicates that the address bus is in use. This condition effectively blocks the 604 from assuming address bus ownership, regardless of the $\overline{\text{BG}}$ input; see Section 8.3.1, “Address Bus Arbitration.” Note that the 604 will not take the address bus for the sequence of cycles beginning with $\overline{\text{TS}}$ and ending with $\overline{\text{AACK}}$; thus effectively making the use of $\overline{\text{ABB}}$ optional, provided that other bus masters respond in the same way.

Negated—Indicates that the address bus is not owned by another bus master and that it is available to the 604 when accompanied by a qualified bus grant.

Timing Comments Assertion—May occur when the 604 must be prevented from using the address bus (and the processor is not currently asserting $\overline{\text{ABB}}$).

Negation—May occur whenever the 604 can use the address bus.

7.2.2 Address Transfer Start Signals

Address transfer start signals are input and output signals that indicate that an address bus transfer has begun. The transfer start ($\overline{\text{TS}}$) signal identifies the operation as a memory transaction; extended address transfer start ($\overline{\text{XATS}}$) identifies the transaction as a direct-store operation.

For detailed information about how $\overline{\text{TS}}$ and $\overline{\text{XATS}}$ interact with other signals, refer to Section 8.3.2, “Address Transfer,” and Section 8.6, “Direct-Store Operation,” respectively.

7.2.2.1 Transfer Start ($\overline{\text{TS}}$)

The $\overline{\text{TS}}$ signal is both an input and an output signal on the 604.

7.2.2.1.1 Transfer Start ($\overline{\text{TS}}$)—Output

Following are the state meaning and timing comments for the $\overline{\text{TS}}$ output signal.

State Meaning	Asserted—Indicates that the 604 has begun a memory bus transaction and that the address-bus and transfer-attribute signals are valid. When asserted with the appropriate TT0–TT4 signals it is also an implied data bus request for a memory transaction (unless it is an address-only operation). Negated—Is negated during a direct-store operation.
Timing Comments	Assertion—Coincides with the assertion of $\overline{\text{ABB}}$. Negation—Occurs one bus clock cycle after $\overline{\text{TS}}$ is asserted. High Impedance—Occurs one bus clock cycle after $\overline{\text{TS}}$ is negated.

7.2.2.1.2 Transfer Start ($\overline{\text{TS}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{TS}}$ input signal.

State Meaning	Asserted—Indicates that another master has begun a bus transaction and that the address bus and transfer attribute signals are valid for snooping (see $\overline{\text{GBL}}$). Negated—Indicates that no bus transaction is occurring.
Timing Comments	Assertion—May occur during the assertion of $\overline{\text{ABB}}$. Negation—Must occur one bus clock cycle after $\overline{\text{TS}}$ is asserted.

7.2.2.2 Extended Address Transfer Start ($\overline{\text{XATS}}$)

The $\overline{\text{XATS}}$ signal is both an input and an output signal on the 604.

7.2.2.2.1 Extended Address Transfer Start ($\overline{\text{XATS}}$)—Output

Following are the state meaning and timing comments for the $\overline{\text{XATS}}$ output signal.

State Meaning	Asserted—Indicates that the 604 has begun a direct-store operation and that the first address cycle is valid. When asserted with the appropriate XATC signals it is also an implied data bus request for certain direct-store operation (unless it is an address-only operation). Negated—Is negated during an entire memory transaction.

Timing Comments Assertion—Coincides with the assertion of $\overline{\text{ABB}}$.
Negation—Occurs one bus clock cycle after the assertion of $\overline{\text{XATS}}$.

High Impedance—Occurs one bus clock cycle after the negation of $\overline{\text{XATS}}$.

7.2.2.2.2 Extended Address Transfer Start ($\overline{\text{XATS}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{XATS}}$ input signal.

State Meaning Asserted—Indicates that the 604 must check for a direct-store operation reply.

Negated—Indicates that there is no need to check for a direct-store operation reply.

Timing Comments Assertion—May occur while $\overline{\text{ABB}}$ is asserted.
Negation—Must occur one bus clock cycle after $\overline{\text{XATS}}$ 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 8.3.2, “Address Transfer.”

7.2.3.1 Address Bus (A0–A31)

The address bus (A0–A31) consists of 32 signals that are both input and output signals.

7.2.3.1.1 Address Bus (A0–A31)—Output (Memory Operations)

Following are the state meaning and timing comments for the A0–A31 output signals.

State Meaning Asserted/Negated—Represents the physical address (real address in the architecture specification) of the data to be transferred. On burst transfers, the address bus presents the double-word–aligned address containing the critical code/data that missed the cache on a read operation, or the first double word of the cache line on a write operation. Note that the address output during burst operations is not incremented. See Section 8.3.2, “Address Transfer.”

Timing Comments Assertion/Negation—Occurs on the bus clock cycle after a qualified bus grant (coincides with assertion of $\overline{\text{ABB}}$ and $\overline{\text{TS}}$).

High Impedance—Occurs one bus clock cycle after $\overline{\text{AACK}}$ is asserted.

7.2.3.1.2 Address Bus (A0–A31)—Input (Memory Operations)

Following are the state meaning and timing comments for the A0–A31 input signals.

State Meaning Asserted/Negated—Represents the physical address of a snoop operation.

Timing Comments Assertion/Negation—Must occur on the same bus clock cycle as the assertion of $\overline{\text{TS}}$; is sampled by 604 only on this cycle.

7.2.3.1.3 Address Bus (A0–A31)—Output (Direct-Store Operations)

Following are the state meaning and timing comments for the address bus signals (A0 to A31) for output direct-store operations on the 604.

State Meaning Asserted/Negated—For direct-store operations where the 604 is the master, the address tenure consists of two packets (each requiring a bus cycle). For packet 0, these signals convey control and tag information. For packet 1, these signals represent the physical address of the data to be transferred. For reply operations, the address bus contains control, status, and tag information.

Timing Comments Assertion/Negation—Address tenure consists of two beats. The first beat occurs on the bus clock cycle after a qualified bus grant, coinciding with $\overline{\text{XATS}}$. The address bus transitions to the second beat on the next bus clock cycle.

High Impedance—Occurs on the bus clock cycle after $\overline{\text{AACK}}$ is asserted.

7.2.3.1.4 Address Bus (A0–A31)—Input (Direct-Store Operations)

Following are the state meaning and timing comments for input direct-store operations on the 604.

State Meaning Asserted/Negated—When the 604 is not the master, it snoops (and checks address parity) on the first address beat only of all direct-store operations for an I/O reply operation with a receiver tag that matches its PID tag. See Section 8.6, "Direct-Store Operation."

Timing Comments Assertion/Negation—The first beat of the I/O transfer address tenure coincides with $\overline{\text{XATS}}$, with the second address bus beat on the following cycle.

7.2.3.2 Address Bus Parity (AP0–AP3)

The address bus parity (AP0–AP3) signals are both input and output signals reflecting one bit of odd-byte parity for each of the four bytes of address when a valid address is on the bus.

7.2.3.2.1 Address Bus Parity (AP0–AP3)—Output

Following are the state meaning and timing comments for the AP0–AP3 output signal on the 604.

State Meaning Asserted/Negated—Represents odd parity for each of four bytes of the physical address for a 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	A0–A7
AP1	A8–A15
AP2	A16–A23
AP3	A24–A31

For more information, see Section 8.3.2.1, “Address Bus Parity.”

Timing Comments Assertion/Negation—The same as A0–A31.
High Impedance—The same as A0–A31.

7.2.3.2.2 Address Bus Parity (AP0–AP3)—Input

Following are the state meaning and timing comments for the AP0–AP3 input signal on the 604.

State Meaning Asserted/Negated—Represents odd parity for each of four bytes of the physical address for snooping and direct-store operations. 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.” (See also the $\overline{\text{APE}}$ signal description.)

Timing Comments Assertion/Negation—The same as A0–A31.

7.2.3.3 Address Parity Error ($\overline{\text{APE}}$)—Output

The address parity error ($\overline{\text{APE}}$) signal is an output signal on the 604. Note that the ($\overline{\text{APE}}$) signal is an open-drain type output, and requires an external pull-up resistor (for example, 10 kΩ to Vdd) to assure proper deassertion of the $\overline{\text{APE}}$ signal). Following are the state meaning and timing comments for the $\overline{\text{APE}}$ signal on the 604. For more information, see Section 8.3.2.1, “Address Bus Parity.”

State Meaning Asserted—Indicates that incorrect address bus parity has been detected by the 604 on a snoop that the 604 recognizes. This includes the first address beat of a direct-store operation.

Negated—Indicates that the 604 has not detected a parity error (even parity) on the address bus.

Timing Comments Assertion—Occurs on the second bus clock cycle after $\overline{\text{TS}}$ or $\overline{\text{XATS}}$ is asserted.

High Impedance—Occurs on the third bus clock cycle after $\overline{\text{TS}}$ or $\overline{\text{XATS}}$ is asserted.

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 8.3.2, “Address Transfer.”

Note that some signal functions vary depending on whether the transaction is a memory access or an I/O access. For a description of how these signals function for direct-store operations, see Section 8.6, “Direct-Store Operation.”

7.2.4.1 Transfer Type (TT0–TT4)

The transfer type (TT0–TT4) signals consist of five input/output signals on the 604. For a complete description of TT0–TT4 signals and for transfer type encodings, see Table 7-1.

7.2.4.1.1 Transfer Type (TT0–TT4)—Output

Following are the state meaning and timing comments for the TT0–TT4 output signals on the 604.

State Meaning Asserted/Negated—Indicates the type of transfer in progress.

For direct-store operations these signals are part of the extended address transfer code (XATC) along with TSIZ and $\overline{\text{TBST}}$:

XATC(0–7)=TT(0–3)||$\overline{\text{TBST}}$||TSIZ(0–2).

Timing Comments Assertion/Negation/High Impedance—The same as A0–A31.

7.2.4.1.2 Transfer Type (TT0–TT4)—Input

Following are the state meaning and timing comments for the TT0–TT4 input signals on the 604.

State Meaning Asserted/Negated—Indicates the type of transfer in progress (see Table 7-1). For direct-store operations, the TT0–TT3 signals form part of the XATC and are snooped by the 604 if $\overline{\text{XATS}}$ is asserted.

Timing Comments Assertion/Negation—The same as A0–A31.

Table 7-1 describes the transfer encodings for a 604 bus master and the 60x bus specification.

Table 7-1. Transfer Encoding for PowerPC 604 Processor Bus Master

TT0	TT1	TT2	TT3	TT4	604 Bus Master Transaction	Transaction	Transaction Source
0	0	0	0	0	Clean block	Address only	Cache operation
0	0	1	0	0	Flush block	Address only	Cache operation
0	1	0	0	0	SYNC	Address only	Cache operation
0	1	1	0	0	Kill block	Address only	Store hit/shared or cache operation
1	0	0	0	0	Ordered I/O operation	Address only	**eieio**
1	0	1	0	0	External control word write	Single-beat write	**ecowx**
1	1	0	0	0	TLB invalidate	Address only	**tlbie**
1	1	1	0	0	External control word read	Single-beat read	**eciwx**
0	0	0	0	1	**lwarx** Reservation set	Address only	**lwarx** with cache hit
0	0	1	0	1	Reserved	Address only	N/A

Table 7-1. Transfer Encoding for PowerPC 604 Processor Bus Master (Continued)

TT0	TT1	TT2	TT3	TT4	604 Bus Master Transaction	Transaction	Transaction Source
0	1	0	0	1	TLBSYNC	Address only	**tlbsync** or **tlbie**
0	1	1	0	1	ICBI	Address only	N/A
1	X	X	0	1	Reserved	—	N/A
0	0	0	1	0	Write-with-flush	Single-beat write or burst	Caching-inhibited or write-through store
0	0	1	1	0	Write-with-kill	Single-beat write or burst	Cast-out, or snoop copyback
0	1	0	1	0	Read	Single-beat read or burst	Caching-inhibited load
0	1	1	1	0	Read-with-intent-to-modify	Burst	Load miss, or store miss
1	0	0	1	0	Write-with-flush-atomic	Single-beat write	**stwcx.**
1	0	1	1	0	Reserved	N/A	N/A
1	1	0	1	0	Read-atomic	Single-beat read or burst	**lwarx** (caching-inhibited load)
1	1	1	1	0	Read-with-intent-to-modify-atomic	Burst	**lwarx** (load miss)
0	0	0	1	1	Reserved	—	N/A
0	0	1	1	1	Reserved	—	N/A
0	1	0	1	1	Read-with-no-intent-to-cache	Single-beat read or burst	N/A
0	1	1	1	1	Reserved	—	N/A
1	X	X	1	1	Reserved	—	N/A

7.2.4.2 Transfer Size (TSIZ0–TSIZ2)

The transfer size (TSIZ0–TSIZ2) signals consist of three input/output signals on the 604.

7.2.4.2.1 Transfer Size (TSIZ0–TSIZ2)—Output

Following are the state meaning and timing comments for the TSIZ0–TSIZ2 output signals on the 604.

State Meaning Asserted/Negated—For memory accesses, these signals along with $\overline{\text{TBST}}$, indicate the data transfer size for the current bus operation, as shown in Table 7-2. Table 8-4 shows how the TSIZ signals are used with the address signals for aligned transfers. Table 8-5 shows how the TSIZ signals are used with the address signals for misaligned transfers. For I/O transfer protocol, these signals form part of the I/O transfer code; see the description in Section 7.2.4.1, “Transfer Type (TT0–TT4).”

For external control instructions (**eciwx** and **ecowx**), TSIZ0–TSIZ2 are used to output bits 29–31 of the external access register (EAR), which are used to form the resource ID ($\overline{\text{TBST}}$||TSIZ0–TSIZ2).

Timing Comments Assertion/Negation—The same as A0–A31.
High Impedance—The same as A0–A31.

Table 7-2. Data Transfer Size

$\overline{\text{TBST}}$	TSIZ0–TSIZ2	Transfer Size
Asserted	010	Burst (32 bytes)
Negated	000	8 bytes
Negated	001	1 byte
Negated	010	2 bytes
Negated	011	3 bytes
Negated	100	4 bytes
Negated	101	5 bytes
Negated	110	6 bytes
Negated	111	7 bytes

7.2.4.2.2 Transfer Size (TSIZ0–TSIZ2)—Input

Following are the state meaning and timing comments for the TSIZ0–TSIZ2 input signals on the 604.

State Meaning Asserted/Negated— For the direct-store protocol, these signals form part of the I/O transfer code; see Section 7.2.4.1, “Transfer Type (TT0–TT4).”

Timing Comments Assertion/Negation—The same as A0–A31.

7.2.4.3 Transfer Burst ($\overline{\text{TBST}}$)

The transfer burst ($\overline{\text{TBST}}$) signal is an input/output signal on the 604.

7.2.4.3.1 Transfer Burst ($\overline{\text{TBST}}$)—Output

Following are the state meaning and timing comments for the $\overline{\text{TBST}}$ output signal.

State Meaning Asserted—Indicates that a burst transfer is in progress.

Negated—Indicates that a burst transfer is not in progress. Also, part of I/O transfer code; see Section 7.2.4.1, “Transfer Type (TT0–TT4).”

For external control instructions (**eciwx** and **ecowx**), $\overline{\text{TBST}}$ is used to output bit 28 of the EAR, which is used to form the resource ID ($\overline{\text{TBST}}$||TSIZ0–TSIZ2).

Timing Comments Assertion/Negation—The same as A0–A31.
High Impedance—The same as A0–A31.

7.2.4.3.2 Transfer Burst ($\overline{\text{TBST}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{TBST}}$ input signal.

State Meaning Asserted/Negated— For the I/O transfer protocol, this signal forms part of the I/O transfer code; see Section 7.2.4.1, “Transfer Type (TT0–TT4).”

Timing Comments Assertion/Negation—The same as A0–A31.

7.2.4.4 Transfer Code (TC0–TC2)—Output

The transfer code (TC0–TC2) consists of three output signals on the 604 that, when combined with the $\overline{\text{WT}}$ signal, provide additional information about the transaction in progress. Following are the state meaning and timing comments for the TC0–TC2 signals.

State Meaning Asserted/Negated—Represents a special encoding for the transfer in progress (see Table 7-3).

Timing Comments Assertion/Negation—The same as A0–A31.
High Impedance—The same as A0–A31.

Table 7-3. Encodings for TC0–TC2 Signals

Transfer Type	$\overline{\text{WT}}$	TC0	TC1	TC2	Transaction
Write-with-kill	1	1	0	0	Cache copyback
Write-with-kill	0	1	0	0	Block invalidate (**dcbf**)
Write-with-kill	0	0	0	0	Block clean (**dcbst**)
Write-with-kill	0	0	1	0	Snoop push (read operation)

Table 7-3. Encodings for TC0–TC2 Signals (Continued)

Transfer Type	$\overline{WT}$	TC0	TC1	TC2	Transaction
Write-with-kill	0	1	0	0	Snoop push (read-with-intent-to-modify)
Write-with-kill	0	0	0	0	Snoop push (clean operation)
Write-with-kill	0	1	0	0	Snoop push (flush operation)
Kill block	x	1	0	0	Kill block de-allocate (**dcbi**)
Kill block	1	0	0	0	Kill block & allocate with no cast-out (**dcbz**)
Kill block	1	0	0	1	Kill block & allocate with cast-out (**dcbz**)
Kill block	1	0	0	0	Kill block Write to shared block
Read[1]	$\overline{W}$[3]	0	x	0	Data read with no cast-out
Read	$\overline{W}$	0	x	1	Data read with cast-out
Read	$\overline{W}$	1	x	0	Instruction read
ICBI	x	1	0	0	Kill block and de-allocate (**icbi**)[2]

Note: 1. Includes both ordinary and atomic read and read-with-intent-to-modify operations.

2. ICBI operation is distinguished from kill block by assertion of TT4 bit.

3. $\overline{W}$ = write-through bit from translation.

The value shown in the $\overline{WT}$ column reflects the actual logic value seen on the $\overline{WT}$ input signal.

7.2.4.5 Cache Inhibit ($\overline{CI}$)—Output

The cache inhibit ($\overline{CI}$) signal is an output signal on the 604. Following are the state meaning and timing comments for the $\overline{CI}$ signal.

State Meaning Asserted—Indicates that a single-beat transfer will not be cached, reflecting the setting of the I bit for the block or page that contains the address of the current transaction.

Negated—Indicates that a burst transfer will allocate a line in the 604 data cache.

Timing Comments Assertion/Negation—The same as A0–A31.
High Impedance—The same as A0–A31.

7.2.4.6 Write-Through ($\overline{\text{WT}}$)—Output

The write-through ($\overline{\text{WT}}$) signal is an output signal on the 604. Following are the state meaning and timing comments for the $\overline{\text{WT}}$ signal.

State Meaning	Asserted—Indicates that a single-beat transaction is write-through, reflecting the value of the W bit for the block or page that contains the address of the current transaction. Negated—Indicates that a transaction is not write-through.
Timing Comments	Assertion/Negation—The same as A0–A31. High Impedance—The same as A0–A31.

7.2.4.7 Global ($\overline{\text{GBL}}$)

The global ($\overline{\text{GBL}}$) signal is an input/output signal on the 604.

7.2.4.7.1 Global ($\overline{\text{GBL}}$)—Output

Following are the state meaning and timing comments for the $\overline{\text{GBL}}$ output signal.

State Meaning	Asserted—Indicates that a transaction is global, reflecting the setting of the M bit for the block or page that contains the address of the current transaction (except in the case of copy-back operations, which are nonglobal.) Negated—Indicates that a transaction is not global.
Timing Comments	Assertion/Negation—The same as A0–A31. High Impedance—The same as A0–A31.

7.2.4.7.2 Global ($\overline{\text{GBL}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{GBL}}$ input signal.

State Meaning	Asserted—Indicates that a transaction may be snooped by the 604. The 604 will not snoop, regardless of $\overline{\text{GBL}}$ signal assertion, reserved transaction types, bus operations associated with the **eieio**, **eciwx**, **ecowx** instructions, or the address-only bus transaction associated with a **lwarx** reservation set. Negated—Indicates that a transaction is not snooped by the 604.
Timing Comments	Assertion/Negation—The same as A0–A31.

7.2.4.8 Cache Set Element (CSE0–CSE1)—Output

Following are the state meaning and timing comments for the CSE0–CSE1 signals.

State Meaning	Asserted/Negated—Represents the cache replacement set element for the current transaction reloading into or writing out of the cache. Can be used with the address bus and the transfer attribute signals to externally track the state of each cache line in the 604's cache.
Timing Comments	Assertion/Negation—The same as A0–A31. High Impedance—The same as A0–A31.

7.2.5 Address Transfer Termination 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. For detailed information about how these signals interact, see Section 8.3.3, “Address Transfer Termination.”

7.2.5.1 Address Acknowledge ($\overline{\text{AACK}}$)—Input

The address acknowledge ($\overline{\text{AACK}}$) signal is an input signal (input-only) on the 604. Following are the state meaning and timing comments for the $\overline{\text{AACK}}$ signal.

State Meaning Asserted—Indicates that the address phase of a transaction is complete. The address bus will go to a high impedance state on the next bus clock cycle. The 604 samples $\overline{\text{ARTRY}}$ on the bus clock cycle following the assertion of $\overline{\text{AACK}}$.

Negated—Indicates that the address bus and the transfer attributes must remain driven, if negated during $\overline{\text{ABB}}$.

Timing Comments Assertion—May occur as early as the bus clock cycle after $\overline{\text{TS}}$ or $\overline{\text{XATS}}$ is asserted; assertion can be delayed to allow adequate address access time for slow devices. For example, if an implementation supports slow snooping devices, an external arbiter can postpone the assertion of $\overline{\text{AACK}}$.

Negation—Must occur one bus clock cycle after the assertion of $\overline{\text{AACK}}$.

7.2.5.2 Address Retry ($\overline{\text{ARTRY}}$)

The address retry ($\overline{\text{ARTRY}}$) signal is both an input and output signal on the 604.

7.2.5.2.1 Address Retry ($\overline{\text{ARTRY}}$)—Output

Following are the state meaning and timing comments for the $\overline{\text{ARTRY}}$ output signal.

State Meaning Asserted—Indicates that the 604 detects a condition in which a snooped address tenure must be retried. If the 604 needs to update memory as a result of the snoop that caused the retry, the 604 asserts $\overline{\text{BR}}$ the cycle after the $\overline{\text{ARTRY}}$ is asserted.

High Impedance—Indicates that the 604 does not need the snooped address tenure to be retried.

Timing Comments Assertion—Asserted the third bus cycle following the assertion of $\overline{\text{TS}}$ if a retry is required.

Negation—Occurs the second bus cycle after the assertion of $\overline{\text{AACK}}$. Since this signal may be simultaneously driven by multiple devices, it is driven negated in the following ways:

- 1:1 and 2:1 bus ratio—high-impedance for 1/2 bus clock cycle, deasserted for 1 bus clock cycle, then high-impedance.

- 3:1 bus ratio—high-impedance for 1/3 bus clock cycle, deasserted for 2/3 bus clock cycle, then high-impedance.
- 3:2 bus ratio—high-impedance for 1/3 system clock cycle, deasserted for 1 bus clock cycle, then high-impedance.

This special method of negation may be disabled by setting the disable snoop response high state restore bit (bit 7) in HID0.

7.2.5.2.2 Address Retry ($\overline{\text{ARTRY}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{ARTRY}}$ input signal.

State Meaning Asserted—If the 604 is the address bus master, $\overline{\text{ARTRY}}$ indicates that the 604 must retry the preceding address tenure and immediately negate $\overline{\text{BR}}$ (if asserted). If the associated data tenure has already started, the 604 will also abort the data tenure immediately, even if the burst data has been received. If the 604 is not the address bus master, this input indicates that the 604 should immediately negate $\overline{\text{BR}}$ for one bus clock cycle following the assertion of $\overline{\text{ARTRY}}$ by the snooping bus master to allow an opportunity for a copy-back operation to main memory. Note that the subsequent address presented on the address bus may not be the same one associated with the assertion of the $\overline{\text{ARTRY}}$ signal.

Negated/High Impedance—Indicates that the 604 does not need to retry the last address tenure.

Timing Comments Assertion—May occur as early as the second cycle following the assertion of $\overline{\text{TS}}$ or $\overline{\text{XATS}}$, and must occur by the bus clock cycle immediately following the assertion of $\overline{\text{AACK}}$ if an address retry is required.

Negation—Must occur during the second cycle after the assertion of $\overline{\text{AACK}}$.

7.2.5.3 Shared ($\overline{\text{SHD}}$)

The shared ($\overline{\text{SHD}}$) signal is both an input and output signal on the 604.

7.2.5.3.1 Shared ($\overline{\text{SHD}}$)—Output

Following are the state meaning and timing comments for the $\overline{\text{SHD}}$ output signal.

State Meaning Asserted—Indicates that the 604 had a cache hit on a shared block, or, if asserted with $\overline{\text{ARTRY}}$, a snoop push of modified data is required.

Negated/High Impedance—Indicates that the 604 did not have a cache hit on the snooped address.

Timing Comments Assertion/Negation—Same as $\overline{\text{ARTRY}}$.

High Impedance—Same as $\overline{\text{ARTRY}}$.

7.2.5.3.2 Shared ($\overline{\text{SHD}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{SHD}}$ input signal.

State Meaning	Asserted—If $\overline{\text{ARTRY}}$ is not asserted, indicates that for a self-generated transaction the 604 must allocate the incoming cache block as shared-unmodified. Negated—If $\overline{\text{ARTRY}}$ is not asserted, indicates that the address for the current transaction is not in any other cache.
Timing Comments	Assertion/Negation—The same as $\overline{\text{ARTRY}}$.

7.2.6 Data Bus Arbitration Signals

Like the address bus arbitration signals, data bus arbitration signals maintain an orderly process for determining data bus mastership. Note that there is no data bus arbitration signal equivalent to the address bus arbitration signal $\overline{\text{BR}}$ (bus request), because, except for address-only transactions, $\overline{\text{TS}}$ and $\overline{\text{XATS}}$ imply data bus requests. For a detailed description on how these signals interact, see Section 8.4.1, "Data Bus Arbitration."

One special signal, $\overline{\text{DBWO}}$, allows the 604 to be configured dynamically to write data out of order with respect to read data. For detailed information about using $\overline{\text{DBWO}}$, see Section 8.11, "Using Data Bus Write Only."

7.2.6.1 Data Bus Grant ($\overline{\text{DBG}}$)—Input

The data bus grant ($\overline{\text{DBG}}$) signal is an input signal (input-only) on the 604. Following are the state meaning and timing comments for the $\overline{\text{DBG}}$ signal.

State Meaning	Asserted—Indicates that the 604 may, with the proper qualification, assume mastership of the data bus. The 604 derives a qualified data bus grant when $\overline{\text{DBG}}$ is asserted and $\overline{\text{DBB}}$, $\overline{\text{DRTRY}}$, and $\overline{\text{ARTRY}}$ are negated; that is, the data bus is not busy ($\overline{\text{DBB}}$ is negated), there is no outstanding attempt to retry the current data tenure ($\overline{\text{DRTRY}}$ is negated), and there is no outstanding attempt to perform an $\overline{\text{ARTRY}}$ of the associated address tenure. Negated—Indicates that the 604 must hold off its data tenures.
Timing Comments	Assertion—May occur any time to indicate the 604 is free to take data bus mastership. It is not sampled until $\overline{\text{TS}}$ or $\overline{\text{XATS}}$ is asserted. Negation—May occur at any time to indicate the 604 cannot assume data bus mastership.

7.2.6.2 Data Bus Write Only ($\overline{\text{DBWO}}$)—Input

The data bus write only ($\overline{\text{DBWO}}$) signal is an input signal (input-only) on the 604. Following are the state meaning and timing comments for the $\overline{\text{DBWO}}$ signal.

State Meaning Asserted—Indicates that the 604 may run the data bus tenure for an outstanding write address even if a read address is pipelined before the write address. Refer to Section 8.11, “Using Data Bus Write Only,” for detailed instructions for using $\overline{\text{DBWO}}$.

Negated—Indicates that the 604 must run the data bus tenures in the same order as the address tenures.

Timing Comments Assertion—Must occur no later than a qualified $\overline{\text{DBG}}$ for an outstanding write tenure. $\overline{\text{DBWO}}$ is only recognized by the 604 on the clock of a qualified $\overline{\text{DBG}}$. If no write requests are pending, the 604 will ignore $\overline{\text{DBWO}}$ and assume data bus ownership for the next pending read request.

Negation—May occur any time after a qualified $\overline{\text{DBG}}$ and before the next assertion of $\overline{\text{DBG}}$.

7.2.6.3 Data Bus Busy ($\overline{\text{DBB}}$)

The data bus busy ($\overline{\text{DBB}}$) signal is both an input and output signal on the 604.

7.2.6.3.1 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—Indicates that the 604 is the data bus master. The 604 always assumes data bus mastership if it needs the data bus and is given a *qualified* data bus grant (see $\overline{\text{DBG}}$).

Negated—Indicates that the 604 is not using the data bus.

Timing Comments Assertion—Occurs during the bus clock cycle following a qualified $\overline{\text{DBG}}$.

Negation—Occurs a fractional bus clock cycle following the assertion of the final $\overline{\text{TA}}$.

High Impedance—Occurs one-half bus cycle (two-thirds bus cycle when using 3:1 clock mode, and one-third bus cycle when using 3:2 bus ratio) after $\overline{\text{DBB}}$ is negated.

7.2.6.3.2 Data Bus Busy ($\overline{\text{DBB}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{DBB}}$ input signal. Note that the $\overline{\text{DBB}}$ input signal cannot be used in systems that use read data streaming.

State Meaning Asserted—Indicates that another device is bus master.
Negated—Indicates that the data bus is free (with proper qualification, see $\overline{\text{DBG}}$) for use by the 604.

Timing Comments Assertion—Must occur when the 604 must be prevented from using the data bus.

Negation—May occur whenever the data bus is available.

7.2.7 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 8.4.3, “Data Transfer.”

7.2.7.1 Data Bus (DH0–DH31, DL0–DL31)

The data bus (DH0–DH31 and DL0–DL31) consists of 64 signals that are both input and output on the 604. Following are the state meaning and timing comments for the DH and DL 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. Direct-store operations use DH exclusively (that is, there are no 64-bit, I/O transfers).

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
DH0–DH7	0
DH8–DH15	1
DH16–DH23	2
DH24–DH31	3
DL0–DL7	4
DL8–DL15	5
DL16–DL23	6
DL24–DL31	7

7.2.7.1.1 Data Bus (DH0–DH31, DL0–DL31)—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—Initial beat coincides with $\overline{\text{DBB}}$ and, for bursts, transitions on the bus clock cycle following each assertion of $\overline{\text{TA}}$.

High Impedance—Occurs on the bus clock cycle after the final assertion of $\overline{\text{TA}}$.

7.2.7.1.2 Data Bus (DH0–DH31, DL0–DL31)—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 $\overline{\text{TA}}$ is asserted.

7.2.7.2 Data Bus Parity (DP0–DP7)

The eight data bus parity (DP0–DP7) signals on the 604 are both output and input signals.

7.2.7.2.1 Data Bus Parity (DP0–DP7)—Output

Following are the state meaning and timing comments for the DP output signals.

State Meaning Asserted/Negated—Represents odd parity for each of eight 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 DL0–DL31.
High Impedance—The same as DL0–DL31.

Table 7-5. DP0–DP7 Signal Assignments

Signal Name	Signal Assignments
DP0	DH0–DH7
DP1	DH8–DH15
DP2	DH16–DH23
DP3	DH24–DH31
DP4	DL0–DL7
DP5	DL8–DL15
DP6	DL16–DL23
DP7	DL24–DL31

7.2.7.2.2 Data Bus Parity (DP0–DP7)—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. Parity is checked on all data byte lanes during data read operations, regardless of the size of the transfer. During direct-store read operations, only the DP0-DP3 signals (corresponding to byte lanes DH0–DH31) are checked for odd parity. Detected even parity causes a checkstop or a machine check exception (and assertion of $\overline{\text{DPE}}$) if data parity errors are enabled in the HID register. (The DP0–DP7 signals function in the same way as the AP0-AP3 signals.)

Timing Comments	Assertion/Negation—The same as DL0–DL31.

7.2.7.3 Data Parity Error ($\overline{\text{DPE}}$)—Output

The data parity error ($\overline{\text{DPE}}$) signal is an output signal (output-only) on the 604. Note that the ($\overline{\text{DPE}}$) signal is an open-drain type output, and requires an external pull-up resistor (for example, 10 kΩ to Vdd) to assure proper deassertion of the ($\overline{\text{DPE}}$) signal. Following are the state meaning and timing comments for the $\overline{\text{DPE}}$ signal.

State Meaning	Asserted—Indicates incorrect data bus parity. Negated—Indicates correct data bus parity.
Timing Comments	Assertion—Occurs on the second bus clock cycle after $\overline{\text{TA}}$ is asserted to the 604. High Impedance—Occurs on the third bus clock cycle after $\overline{\text{TA}}$ is asserted to the 604.

7.2.7.4 Data Bus Disable ($\overline{\text{DBDIS}}$)—Input

The Data Bus Disable ($\overline{\text{DBDIS}}$) signal is an input signal (input-only) on the 604. Following are the state meanings and timing comments for the $\overline{\text{DBDIS}}$ signal.

State Meaning	Asserted—Indicates (for a write transaction) that the 604 must release data bus and the data bus parity to high impedance during the following cycle. The data tenure will remain active, $\overline{\text{DBB}}$ will remain driven, and the transfer termination signals will still be monitored by the 604. Negated—Indicates the data bus should remain normally driven. $\overline{\text{DBDIS}}$ is ignored during read transactions.
Timing Comments	Assertion/Negation—May be asserted on any clock cycle when the 604 is driving, or will be driving the data bus; may remain asserted multiple cycles.

7.2.8 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.

For a detailed description of how these signals interact, see Section 8.4.4, “Data Transfer Termination.”

7.2.8.1 Transfer Acknowledge ($\overline{\text{TA}}$)—Input

The transfer acknowledge ($\overline{\text{TA}}$) signal is an input signal (input-only) on the 604. Following are the state meaning and timing comments for the $\overline{\text{TA}}$ signal.

State Meaning	Asserted— Indicates that a single-beat data transfer completed successfully or that a data beat in a burst transfer completed successfully (unless $\overline{\text{DRTRY}}$ is asserted on the next bus clock cycle). Note that $\overline{\text{TA}}$ must be asserted for each data beat in a burst transaction. For more information, see Section 8.4.4, “Data Transfer Termination.” Negated—(During $\overline{\text{DBB}}$) indicates that, until $\overline{\text{TA}}$ is asserted, the 604 must continue to drive the data for the current write or must wait to sample the data for reads.
Timing Comments	Assertion—When the bus is configured for normal operation, must not occur earlier than one bus clock cycle before the beginning of the valid $\overline{\text{ARTRY}}$ window, or when the bus is configured for fast-L2 mode, must not be asserted earlier than the first cycle of a valid $\overline{\text{ARTRY}}$ window; otherwise, assertion may occur at any time during the assertion of $\overline{\text{DBB}}$. The system can withhold assertion of $\overline{\text{TA}}$ to indicate that the 604 should insert wait states to extend the duration of the data beat. Negation—Must occur after the bus clock cycle of the final (or only) data beat of the transfer. For a burst transfer, the system can assert $\overline{\text{TA}}$ for one bus clock cycle and then negate it to advance the burst transfer to the next beat and insert wait states during the next beat.

7.2.8.2 Data Retry ($\overline{\text{DRTRY}}$)—Input

The data retry ($\overline{\text{DRTRY}}$) signal is input only on the 604. Following are the state meaning and timing comments for the $\overline{\text{DRTRY}}$ signal.

State Meaning	Asserted—Indicates that the 604 must invalidate the data from the previous read operation. Negated—Indicates that data presented with $\overline{\text{TA}}$ on the previous read operation is valid. This is essentially a late $\overline{\text{TA}}$ to allow speculative forwarding of data (with $\overline{\text{TA}}$) during reads. Note that $\overline{\text{DRTRY}}$ is ignored for write transactions.

Timing Comments Assertion—Must occur during the bus clock cycle immediately after $\overline{\text{TA}}$ is asserted if a retry is required. The $\overline{\text{DRTRY}}$ signal may be held asserted for multiple bus clock cycles. When $\overline{\text{DRTRY}}$ is negated, data must have been valid on the previous clock with $\overline{\text{TA}}$ asserted.

Negation—Must occur during the bus clock cycle after a valid data beat. This may occur several cycles after $\overline{\text{DBB}}$ is negated, effectively extending the data bus tenure.

Startup—$\overline{\text{DRTRY}}$ is sampled at the negation of $\overline{\text{HRESET}}$; if $\overline{\text{DRTRY}}$ is asserted, fast-L2 mode is selected. If $\overline{\text{DRTRY}}$ is negated at startup, $\overline{\text{DRTRY}}$ is enabled. $\overline{\text{DRTRY}}$ must be negated during normal operation (following $\overline{\text{HRESET}}$) if fast-L2/data streaming mode is selected.

7.2.8.3 Transfer Error Acknowledge ($\overline{\text{TEA}}$)—Input

The transfer error acknowledge ($\overline{\text{TEA}}$) signal is input only on the 604. Following are the state meaning and timing comments for the $\overline{\text{TEA}}$ signal.

State Meaning Asserted—Indicates that a bus error occurred. Causes a machine check exception (and possibly causes the processor to enter checkstop state if machine check enable bit is cleared (MSR[ME] = 0)). For more information, see Section 4.5.2.2, “Checkstop State (MSR[ME] = 0).” Assertion terminates the current transaction; that is, assertion of $\overline{\text{TA}}$ and $\overline{\text{DRTRY}}$ are ignored. The assertion of $\overline{\text{TEA}}$ causes the negation/high impedance of $\overline{\text{DBB}}$ in the next clock cycle. However, data entering the GPR or the cache are not invalidated. Note that the architecture specification refers to all exceptions as interrupts.

Negated—Indicates that no bus error was detected.

Timing Comments Assertion—May be asserted while $\overline{\text{DBB}}$ is asserted, or during valid $\overline{\text{DRTRY}}$ window. In fast-L2/data streaming mode, the 604 will not recognize $\overline{\text{TEA}}$ the cycle after $\overline{\text{TA}}$ during a read operation due to the absence of a $\overline{\text{DRTRY}}$ assertion opportunity. The $\overline{\text{TEA}}$ signal should be asserted for one cycle only.

Negation— The $\overline{\text{TEA}}$ signal must be negated no later than the negation of $\overline{\text{DBB}}$ or the last $\overline{\text{DRTRY}}$. The 604 deasserts $\overline{\text{DBB}}$ within one bus clock cycle following the assertion of $\overline{\text{TEA}}$.

7.2.9 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 604 must be reset. The 604 generates the output signal, $\overline{\text{CKSTP\_OUT}}$, when it detects a checkstop condition. For a detailed description of these signals, see Section 8.8, “Interrupt, Checkstop, and Reset Signals.”

7.2.9.1 Interrupt ($\overline{\text{INT}}$)—Input

The interrupt ($\overline{\text{INT}}$) signal is input only. Following are the state meaning and timing comments for the $\overline{\text{INT}}$ signal.

State Meaning Asserted—The 604 initiates an interrupt if MSR[EE] is set; otherwise, the 604 ignores the interrupt. To guarantee that the 604 will take the external interrupt, the $\overline{\text{INT}}$ signal must be held active until the 604 takes the interrupt; otherwise, the 604 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. See Section 8.8.1, “External Interrupts.”

Timing Comments Assertion—May occur at any time and may be asserted asynchronously to the input clocks. The $\overline{\text{INT}}$ input is level-sensitive.

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 $\overline{\text{INT}}$ signal should be asserted and negated synchronously with the SYSCLK signal.

7.2.9.2 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—The 604 initiates a system management interrupt operation if the MSR[EE] is set; otherwise, the 604 ignores the interrupt condition. The system must hold the $\overline{\text{SMI}}$ signal active until the interrupt is taken.

Negated—Indicates that normal operation should proceed. See Section 8.8.1, “External Interrupts.”

Timing Comments Assertion—May occur at any time and may be asserted asynchronously to the input clocks. The $\overline{\text{SMI}}$ input is level-sensitive.

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.

7.2.9.3 Machine Check Interrupt ($\overline{\text{MCP}}$)—Input

The machine check interrupt ($\overline{\text{MCP}}$) signal is input only on the 604. Following are the state meaning and timing comments for the $\overline{\text{MCP}}$ signal.

State Meaning Asserted—The 604 initiates a machine check interrupt operation if MSR[EE] and HID0[EMCP] are set; if MSR[EE] is cleared and HID0[EMCP] is set, the 604 must terminate operation by internally gating off all clocks, and releasing all outputs (except $\overline{\text{CKSTP\_OUT}}$) to the high impedance state. If HID0[EMCP] is cleared, the 604 ignores the interrupt condition. The $\overline{\text{MCP}}$ signal must be held asserted for two bus clock cycles.

Negated—Indicates that normal operation should proceed. See Section 8.8.1, "External Interrupts."

Timing Comments Assertion—May occur at any time and may be asserted asynchronously to the input clocks. The $\overline{\text{MCP}}$ input is negative edge-sensitive.

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.

7.2.9.4 Checkstop Input($\overline{\text{CKSTP\_IN}}$)—Input

The checkstop input ($\overline{\text{CKSTP\_IN}}$) signal is input only on the 604. Following are the state meaning and timing comments for the $\overline{\text{CKSTP\_IN}}$ signal.

State Meaning Asserted—Indicates that the 604 must terminate operation by internally gating off all clocks, and release all outputs (except $\overline{\text{CKSTP\_OUT}}$) to the high impedance state. Once $\overline{\text{CKSTP\_IN}}$ has been asserted it must remain asserted until the system has been reset.

Negated—Indicates that normal operation should proceed. See Section 8.8.2, "Checkstops."

Timing Comments Assertion—May occur at any time and may be asserted asynchronously to the input clocks.

Negation—May occur any time after the $\overline{\text{CKSTP\_OUT}}$ output signal has been asserted.

7.2.9.5 Checkstop Output ($\overline{\text{CKSTP\_OUT}}$)—Output

The checkstop ($\overline{\text{CKSTP\_OUT}}$) signal is output only on the 604. Note that the ($\overline{\text{CKSTP\_OUT}}$) signal is an open-drain type output, and requires an external pull-up resistor (for example, 10 kΩ to Vdd) to assure proper deassertion of the ($\overline{\text{CKSTP\_OUT}}$) signal. Following are the state meaning and timing comments for the $\overline{\text{CKSTP\_OUT}}$ signal.

State Meaning Asserted—Indicates that the 604 has detected a checkstop condition and has ceased operation.

Negated—Indicates that the 604 is operating normally.
See Section 8.8.2, "Checkstops."

Timing Comments Assertion—May occur at any time and may be asserted asynchronously to the 604 input clocks.

Negation—Is negated upon assertion of $\overline{\text{HRESET}}$.

7.2.9.6 Reset Signals

There are two reset signals on the 604—hard reset ($\overline{\text{HRESET}}$) and soft reset ($\overline{\text{SRESET}}$). Descriptions of the reset signals are as follows:

7.2.9.6.1 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 4.5.1, “System Reset Exception (0x00100).” Output drivers are released to high impedance within five clocks after the assertion of $\overline{\text{HRESET}}$.

Negated—Indicates that normal operation should proceed. See Section 8.8.3, “Reset Inputs.”

Timing Comments Assertion—May occur at any time and may be asserted asynchronously to the 604 input clock; must be held asserted for a minimum of 255 clock cycles.

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 SYSCLK signal. The $\overline{\text{HRESET}}$ signal has additional functionality in certain test modes.

7.2.9.6.2 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— Initiates processing for a reset exception as described in Section 4.5.1, “System Reset Exception (0x00100).”

Negated—Indicates that normal operation should proceed. See Section 8.8.3, “Reset Inputs.”

Timing Comments Assertion—May occur at any time and may be asserted asynchronously to the 604 input clock. The $\overline{\text{SRESET}}$ input is negative edge-sensitive.

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 $\overline{\text{SRESET}}$ signal should be asserted and negated synchronously with the SYSCLK signal. The $\overline{\text{SRESET}}$ signal has additional functionality in certain test modes.

7.2.10 Processor Configuration Signals

The signals described in this section provide inputs for controlling the 604's timebase, signal drive capabilities, L2 cache access, bus snooping while in nap mode, and PLL configuration, along with output signals to indicate that a storage reservation has been set, and that the 604's internal clocking has stopped.

7.2.10.1 Timebase Enable (TBEN)—Input

The timebase enable (TBEN) signal is input only on the 604. Following are the state meanings and timing comments for the TBEN signal.

State Meaning	Asserted—Indicates that the timebase should continue clocking. This input is essentially a "count enable" control for the timebase counter.
	Negated—Indicates the timebase should stop clocking.
Timing Comments	Assertion/Negation—May occur on any cycle.

7.2.10.2 Reservation ($\overline{\text{RSRV}}$)—Output

The reservation ($\overline{\text{RSRV}}$) signal is output only on the 604. 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. See Section 8.9.1, "Support for the lwarx/stwcx. Instruction Pair."
Timing Comments	Assertion/Negation—Occurs synchronously one bus clock cycle after the execution of an **lwarx** instruction that sets the internal reservation condition.

7.2.10.3 L2 Intervention (L2_INT)—Input

The L2 intervention (L2_INT) signal is input only on the 604. Following are the state meanings and timing comments for the L2_INT signal.

State Meaning	Asserted— Indicates that the current data transaction requires intervention from other bus masters.
	Negated—Indicates that the current data transaction requires no intervention from other bus masters.
Timing Comments	Assertion/Negation—The L2_INT signal is sampled by the 604 concurrently with the first assertion of $\overline{\text{TA}}$ for a given data tenure.

7.2.10.4 Run (RUN)—Input

The run (RUN) signal is input only on the 604. Following are the state meanings and timing comments for the RUN signal.

State Meaning	Asserted— Forces the internal clocks to continue running during nap mode, allowing bus snooping to occur.
	Negated—Internal clocks are inhibited from running when 604 is in nap mode.

For additional information regarding the nap mode, refer to Section 4.5.16, “Power Management.”

Timing Comments	Assertion/Negation—Assertion may occur asynchronously to the 604 input clock; and must be held asserted for a minimum of 3 bus clock cycles before snoop activity.

7.2.10.5 Halted (HALTED) —Output

The halted (HALTED) signal is output only on the 604. Following are the state meaning and timing comments for the HALTED signal.

State Meaning	Asserted—Indicates that the internal clocks have stopped due to the 604 entering nap mode, or a JTAG/COP request.
	Negated—Indicates that internal clocks are running.
Timing Comments	Assertion/Negation—Occurs synchronously with internal processor clock.

For additional information regarding the nap mode, refer to Section 4.5.16, “Power Management.”

7.2.11 COP/Scan Interface

The 604 has extensive on-chip test capability including the following:

- Built-in instruction and data cache self test (BIST)
- Debug control/observation (COP)
- Boundary scan (IEEE 1149.1 compliant interface)

The BIST hardware is not exercised as part of the POR sequence. The COP and boundary scan logic are not used under typical operating conditions.

Detailed discussion of the 604 test functions is beyond the scope of this document; however, sufficient information has been provided to allow the system designer to disable the test functions that would impede normal operation.

The COP/scan interface is shown in Figure 7-2. For more information, see Section 8.10.1, “IEEE 1149.1 Interface Description.”

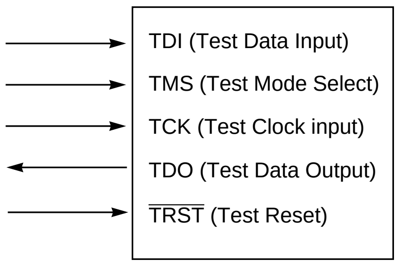


Figure 7-2. IEEE 1149.1-Compliant Boundary Scan Interface

7.2.12 Clock Signals

The clock signal inputs of the 604 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.

Refer to the 604 hardware specifications for exact timing relationships of the clock signals.

7.2.12.1 System Clock (SYSCLK)—Input

The 604 requires a single system clock (SYSCLK) input. This input sets the frequency of operation for the bus interface. Internally, the 604 uses a phase-lock loop (PLL) circuit to generate a master clock for all of the CPU circuitry (including the bus interface circuitry) which is phase-locked to the SYSCLK input. The master clock may be set to a multiple (x1, x1.5, x2, or x3) of the SYSCLK frequency allowing the CPU core to operate at an equal or greater frequency than the bus interface.

State Meaning	Asserted/Negated—The SYSCLK input is the primary clock input for the 604, and represents the bus clock frequency for 604 bus operation. Internally, the 604 may be operating at a multiple of the bus clock frequency.
Timing Comments	Duty cycle—Refer to the 604 hardware specifications for timing comments. **Note**: SYSCLK is used as the frequency reference for the internal PLL clock generator, and must not be suspended or varied during normal operation to ensure proper PLL operation.

7.2.12.2 Test Clock (CLK_OUT)—Output

The Test Clock (CLK_OUT) signal is an output signal (output-only) on the 604. Following are the state meaning and timing comments for the CLK_OUT signal.

State Meaning	Asserted/Negated—Provides PLL clock output for PLL testing and monitoring. CLK_OUT clocks at the processor clock frequency. The CLK_OUT signal is provided for testing purposes only.
Timing Comments	Assertion/Negation—Refer to the 604 hardware specifications for timing comments.

7.2.12.3 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 604 electrical specification.

7.2.12.4 PLL Configuration (PLL_CFG0–PLL_CFG3)—Input

The PLL (phase-lock loop) is configured by the PLL_CFG0–PLL_CFG3 pins. For a given SYSCLK (bus) frequency, the PLL configuration pins set the internal CPU frequency of operation.

Following are the state meaning and timing comments for the PLL_CFG0–PLL_CFG3 signals.

State Meaning	Asserted/Negated— Configures the operation of the PLL and the internal processor clock frequency. Settings are based on the desired bus and internal frequency of operation.
Timing Comments	Assertion/Negation—Must remain stable during operation.

Table 7-6. PLL Configuration

PLL_CFG 0–3	Bus, CPU and PLL Frequencies							
	CPU/ SYSCLK Ratio	Bus 16.6 MHz	Bus 20 MHz	Bus 25 MHz	Bus 33.3 MHz	Bus 40 MHz	Bus 50 MHz	Bus 66.6 MHz
00 00	1:1	—	—	—	—	—	50 (100)	66.6 (133)
0001	1:1	—	—	25 (100)	33.3 (133)	40 (160)	50 (200)	—
0010	1:1	16.6 (133)	20 (160)	25 (200)	—	—	—	—
0100	2:1	—	—	50 (100)	66.6 (133)	80 (160)	100 (200)	—
0101	2:1	33.3 (133)	40 (160)	50 (200)	—	—	—	—

Table 7-6. PLL Configuration (Continued)

PLL_CFG 0–3	Bus, CPU and PLL Frequencies							
	CPU/ SYSCLK Ratio	**Bus 16.6 MHz**	**Bus 20 MHz**	**Bus 25 MHz**	**Bus 33.3 MHz**	**Bus 40 MHz**	**Bus 50 MHz**	**Bus 66.6 MHz**
1000	3:1	—	—	75 (150)	100 (200)	—	—	—
1100	1.5:1	—	—	—	50 (100)	60 (120)	75 (150)	100 (200)
0011	PLL Bypass							

Notes:
1. Some PLL configurations may select bus, CPU, or PLL frequencies which are not useful, not supported, or not tested for by the 604. For complete information, see the 604 hardware specifications for timing comments. PLL frequencies (shown in parenthesis in the table above) should not fall below 100 MHz, and should not exceed 200 MHz.
2. In PLL-bypass mode, the SYSCLK input signal clocks the internal processor directly, and the bus is set for 1:1 mode operation. The PLL-bypass mode is for test only, and is not intended for functional use. In clock-off mode, no clocking occurs inside the 604 regardless of the SYSCLK input.
3. PLL_CFG(0:1) selects the CPU-to-bus ratio (1:1,1.5:1, 2:1, 3:1), PLL_CFG(2:3) selects the CPU-to-PLL multiplier (x2, x4, x8).

Chapter 8
System Interface Operation

This chapter describes the PowerPC 604 microprocessor bus interface and its operation. It shows how the 604 signals, defined in Chapter 7, “Signal Descriptions,” interact to perform address and data transfers.

8.1 PowerPC 604 Microprocessor System Interface Overview

The system interface prioritizes requests for bus operations from the instruction and data caches, and performs bus operations per the 604 bus protocol. It includes address register queues, prioritization logic, and the bus control unit. The system interface latches snoop addresses for snooping in the data cache and in the address register queues, and snoops for direct-store reply operations and for reservations controlled by the Load Word and Reserve Indexed (**lwarx**) and Store Word Conditional Indexed (**stwcx.**) instructions. The interface allows two level of pipelining; that is, with certain restrictions discussed later, there can be three outstanding transactions at any given time. Accesses are prioritized with load operations preceding store operations.

Instructions are automatically fetched from the memory system into the instruction unit where they are dispatched to the execution units at a peak rate of four instructions per clock. Conversely, load and store instructions explicitly specify the movement of operands to and from the integer and floating-point register files and the memory system.

When the 604 encounters an instruction or data access, it calculates the logical address (effective address in the architecture specification) and uses the low-order address bits to check for a hit in the on-chip, 16-Kbyte instruction and data caches. During cache lookup, the instruction and data memory management units (MMUs) use the higher-order address bits to calculate the virtual address, from which they calculate the physical address (real address in the architecture specification). The physical address bits are then compared with the corresponding cache tag bits to determine if a cache hit occurred. If the access misses in the corresponding cache, the physical address is used to access system memory.

In addition to the loads, stores, and instruction fetches, the 604 performs hardware table search operations following TLB misses, cache cast-out operations when least-recently used cache lines are written to memory after a cache miss, and cache-line snoop push-out operations when a modified cache line experiences a snoop hit from another bus master.

Figure 8-1 shows the address path from the execution units and instruction fetcher, through the translation logic to the caches and system interface logic.

The 604 uses separate address and data buses and a variety of control and status signals for performing reads and writes. The address bus is 32 bits wide and the data bus is 64 bits wide. The interface is synchronous—all 604 inputs are sampled at and all outputs are driven from the rising edge of the bus clock. The bus can run at the full processor-clock frequency, or at 1/2, 1/3 or 2/3 the frequency of the processor clock. While the 604 operates at 3.3 Volts, all the I/O signals are 5.0-Volt TTL-compatible.

8.1.1 Operation of the Instruction and Data Caches

The 604 provides independent instruction and data caches. Each cache is a physically-addressed, 16-Kbyte cache with four-way set associativity. Both caches consist of 128 sets of four cache lines, with eight words in each cache line.

Because the data cache on the 604 is an on-chip, write-back primary cache, the predominant type of transaction for most applications is burst-read memory operations, 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, variants of the burst and single-beat operations (global memory operations that are snooped, and atomic memory operations, for example), and address retry activity (for example, when a snooped read access hits a modified line in the cache).

The 604 data cache tags are dual-ported to facilitate efficient coherency checking. This allows data cache accesses to occur concurrently with snooping operations. Data cache accesses are only interrupted when the snoop control logic detects a situation where snoop push of modified data is required to maintain memory coherency.

The 604 supports a four-state coherency protocol that supports the modified, exclusive, shared and invalid (MESI) cache states. The MESI protocol ensures that the 604 operates coherently in systems that contain multiple four-state caches, provided that all bus participants employ similar snooping and coherency control mechanisms.

Cache lines in the 604 are loaded in four beats of 64 bits each. The burst load is performed as critical-double-word-first. The cache that is being loaded allows internal accesses until the load completes (that is, the 604 supports cache hits under misses). The critical double word is simultaneously written to the cache and forwarded to the requesting unit, thus minimizing stalls due to load delays. If consecutive double words are required from the same cache line following a cache line miss, the LSU stalls until the entire cache line has been loaded into the cache,

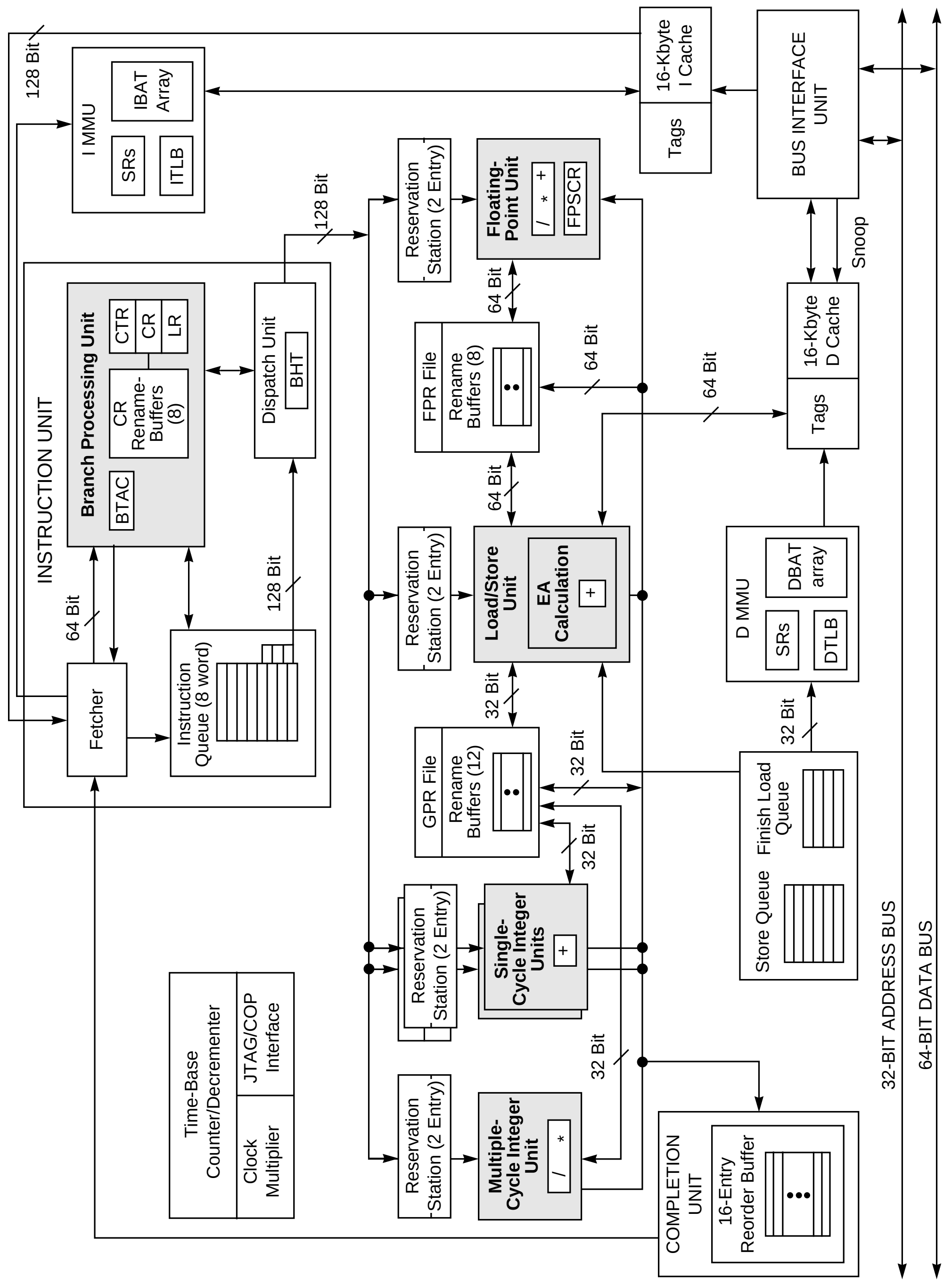


Figure 8-1. PowerPC 604 Microprocessor Block Diagram

Cache lines are selected for replacement based on an LRU (least recently used) algorithm. Each time a cache line is accessed, it is tagged as the most recently used line of the set. When a miss occurs, if all lines in the set are marked as valid, the least recently used line is replaced with the new data. When data to be replaced is in the modified state, the modified data is written into a write-back buffer while the missed data is being read from memory. When the load completes, the 604 then pushes the replaced line from the write-back buffer to main memory in a burst write operation if the memory queue is idle, or at a later time if other transactions are pending.

8.1.2 Operation of the System Interface

Memory accesses can occur in single-beat (1–8 bytes) and four-beat (32 bytes) burst data transfers. The address and data buses are independent for memory accesses to support pipelining and split transactions. The 604 can pipeline as many as three transactions and has limited support for out-of-order split-bus transactions.

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 604 to be integrated into systems that implement various fairness and bus-parking procedures to avoid arbitration overhead.

Typically, memory accesses are weakly ordered—sequences of operations, including load/store string and multiple instructions, do not necessarily complete in the order they begin—maximizing the efficiency of the bus without sacrificing coherency of the data. The 604 allows read operations to precede store operations (except when a dependency exists). In addition, the 604 performs snoop push operations ahead of all other bus operations. Because the processor can dynamically optimize run-time ordering of load/store traffic, overall performance is improved.

Note that the Synchronize (**sync**) or Enforce In-Order Execution of I/O (**eieio**) instructions can be used to enforce strong ordering.

The following sections describe how the 604 interface operates, providing detailed timing diagrams that illustrate how the signals interact. A collection of more general timing diagrams are included as examples of typical bus operations.

Figure 8-2 is a legend of the conventions used in the timing diagrams.

This is a synchronous interface—all 604 input signals are sampled and output signals are driven on the rising edge of the bus clock cycle (see the 604 hardware specifications for exact timing information).

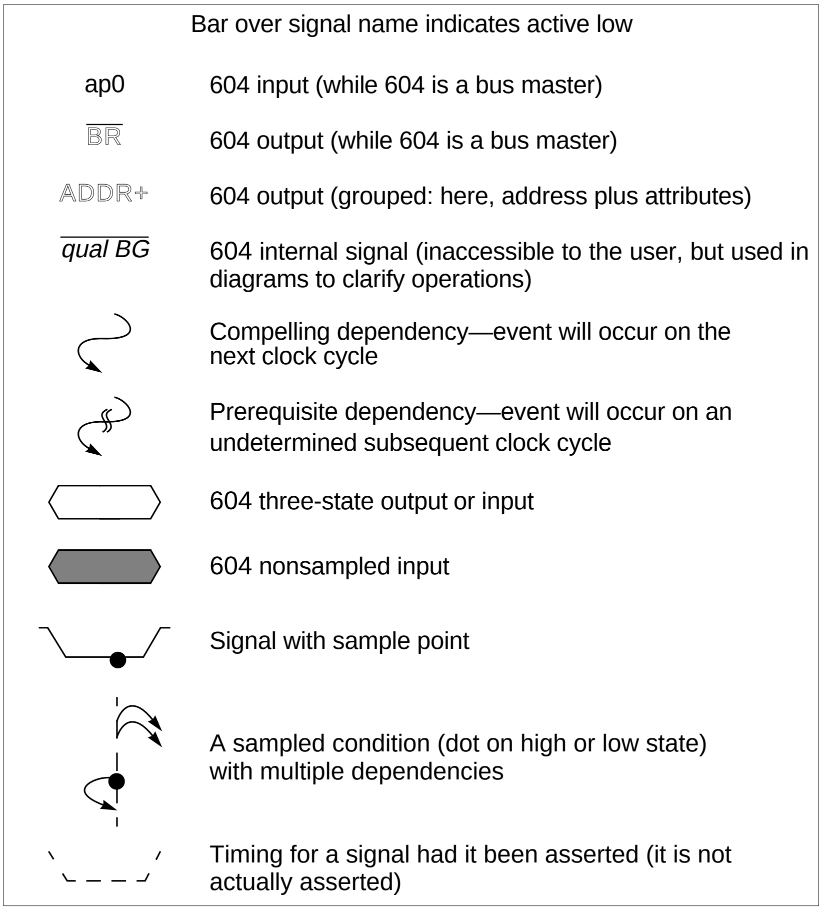

Bar over signal name indicates active low
ap0
604 input (while 604 is a bus master)
$\overline{\text{BR}}$
604 output (while 604 is a bus master)
ADDR+
604 output (grouped: here, address plus attributes)
$\overline{\textit{qual BG}}$
604 internal signal (inaccessible to the user, but used in diagrams to clarify operations)
Compelling dependency—event will occur on the next clock cycle
Prerequisite dependency—event will occur on an undetermined subsequent clock cycle
604 three-state output or input
604 nonsampled input
Signal with sample point
A sampled condition (dot on high or low state) with multiple dependencies
Timing for a signal had it been asserted (it is not actually asserted)

Figure 8-2. Timing Diagram Legend

8.1.3 Direct-Store Accesses

Memory and direct-store accesses use the 604 signals differently.

The 604 defines separate memory and I/O address spaces, or segments, distinguished by the segment register T bit in the address translation logic of the 604. If the T bit is cleared, the memory reference is a normal memory access and uses the paged virtual memory management mechanism of the 604. If the T bit is set, the memory reference is a direct-store access.

The function and timing of some address transfer and attribute signals (such as TT0–TT3, $\overline{\text{TBST}}$, and TSIZ0–TSIZ2) are changed for direct-store accesses. Additional controls are required to facilitate transfers between the 604 and the specific I/O devices that use this interface. Direct-store and memory transfers are distinguished from one another by their address transfer start signals—$\overline{\text{TS}}$ indicates that a memory transfer is starting and $\overline{\text{XATS}}$ indicates that a direct-store transaction is starting.

Direct-store accesses are strongly ordered—each access occurs in strict program order and completes before another access can begin. For this reason, direct-store accesses are less efficient than memory accesses. The direct-store extensions also allow for additional bus pacing and multiple transaction operations for variably-sized data transfers (1 to 128 bytes), and they support a tagged, split request/response protocol. The direct-store access protocol also requires the slave device to function as a bus master.

8.2 Memory Access Protocol

Memory accesses are divided into address and data tenures. Each tenure has three phases—bus arbitration, transfer, and termination. The 604 also supports address-only transactions. Note that address and data tenures can overlap, as shown in Figure 8-3.

Figure 8-3 shows that the address and data tenures are distinct from one another and that both consist of three phases—arbitration, transfer, and termination. Address and data tenures are independent (indicated in Figure 8-3 by the fact that the data tenure begins before the address tenure ends), which allows split-bus transactions to be implemented at the system level in multiprocessor systems. Figure 8-3 shows a data transfer that consists of a single-beat transfer of as many as 64 bits. Four-beat burst transfers of 32-byte cache lines require data transfer termination signals for each beat of data.

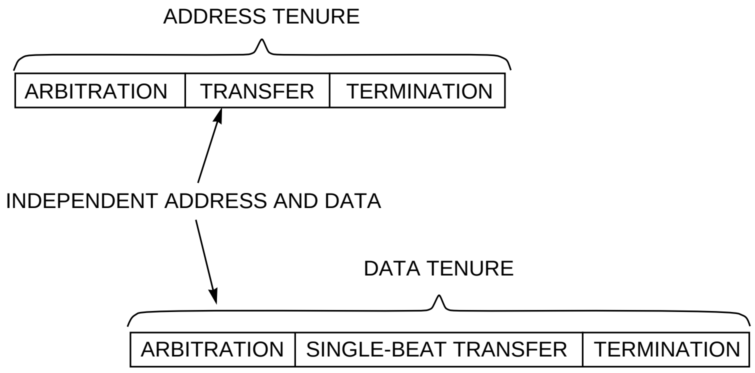


Figure 8-3. Overlapping Tenures on the PowerPC 604 Microprocessor Bus for a Single-Beat Transfer

The basic functions of the address and data tenures are as follows:

- **Address tenure**
  — Arbitration: During arbitration, address bus arbitration signals are used to gain mastership of the address bus.
  — Transfer: After the 604 is the address bus master, it transfers the address on the

address bus. The address signals and the transfer attribute signals control the address transfer. The address parity and address parity error signals ensure the integrity of the address transfer.

  - Termination: After the address transfer, the system signals that the address tenure is complete or that it must be repeated.

- Data tenure
  - Arbitration: To begin the data tenure, the 604 arbitrates for mastership of the data bus.
  - Transfer: After the 604 is the data bus master, it samples the data bus for read operations or drives the data bus for write operations. The data parity and data parity error signals ensure the integrity of the data transfer.
  - Termination: 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.

The 604 generates an address-only bus transfer during the execution of **dcbz**, **sync**, **eieio**, **tlbie**, **tlbsync**, and **lwarx** instructions, which use only the address bus with no data transfer involved. Additionally, the 604's retry capability provides an efficient snooping protocol for systems with multiple memory systems (including caches) that must remain coherent.

8.2.1 Arbitration Signals

Arbitration for both address and data bus mastership is performed by a central, external arbiter and, minimally, by the arbitration signals shown in Section 8.3.1, "Address Bus Arbitration." Most arbiter implementations require additional signals to coordinate bus master/slave/snooping activities. Note that address bus busy ($\overline{\text{ABB}}$) and data bus busy ($\overline{\text{DBB}}$) are bidirectional signals. These signals are inputs unless the 604 has mastership of one or both of the respective buses; they must be connected high through pull-up resistors so that they remain negated when no devices have control of the buses.

The following list describes the address arbitration signals:

- **$\overline{\text{BR}}$ (bus request)**—Assertion indicates that the 604 is requesting mastership of the address bus.
- **$\overline{\text{BG}}$ (bus grant)**—Assertion indicates that the 604 may, with the proper qualification, assume mastership of the address bus. A qualified bus grant occurs when $\overline{\text{BG}}$ is asserted, $\overline{\text{ABB}}$ is negated, and $\overline{\text{ARTRY}}$ is negated during the current and previous bus cycle.

  If the 604 is parked, $\overline{\text{BR}}$ need not be asserted for the qualified bus grant.
- **$\overline{\text{ABB}}$ (address bus busy)**— Assertion by the 604 indicates that the 604 is the address bus master.

The following list describes the data arbitration signals:

- **$\overline{\text{DBG}}$ (data bus grant**)—Indicates that the 604 may, with the proper qualification, assume mastership of the data bus. A qualified data bus grant occurs when $\overline{\text{DBG}}$ is asserted while $\overline{\text{DBB}}$, $\overline{\text{DRTRY}}$, and $\overline{\text{ARTRY}}$ are negated (although $\overline{\text{ARTRY}}$ may actually be asserted at the time $\overline{\text{DBG}}$ is asserted due to the snoop of a later address tenure).

  The $\overline{\text{DBB}}$ signal is driven by the current bus master, $\overline{\text{DRTRY}}$ is only driven from the bus, and $\overline{\text{ARTRY}}$ is from the bus, but only for the address bus tenure associated with the current data bus tenure (that is, not from another address tenure).

- **$\overline{\text{DBWO}}$ (data bus write only**)—Assertion indicates that the 604 may perform the data bus tenure for an outstanding write address even if a read address is pipelined before the write address. If $\overline{\text{DBWO}}$ is asserted, the 604 will assume data bus mastership for a pending data bus write operation; the 604 will take the data bus for a pending read operation if this input is asserted along with $\overline{\text{DBG}}$ and no write is pending. Care must be taken with $\overline{\text{DBWO}}$ to ensure the desired write is queued (for example, a cache-line snoop push-out operation).

- **$\overline{\text{DBB}}$ (data bus busy)**—Assertion by the 604 indicates that the 604 is the data bus master. The 604 always assumes data bus mastership if it needs the data bus and is given a qualified data bus grant (see $\overline{\text{DBG}}$).

  For more detailed information on the arbitration signals, refer to Section 8.3.1, “Address Bus Arbitration,” and Section 8.4.1, “Data Bus Arbitration.”

Note that while operating in fast-L2/data streaming mode, $\overline{\text{DBB}}$ becomes a 604 output-only signal and is driven in the same manner as before. If systems using the 604 in fast-L2/data streaming mode also implement data streaming across multiple masters, the $\overline{\text{DBB}}$ signal must not be common among processors to avoid contention problems when one processor is negating $\overline{\text{DBB}}$ while another is asserting $\overline{\text{DBB}}$. Table 8-1 describes the bus arbitration signals provided by the 604.

Table 8-1. PowerPC 604 Microprocessor Bus Arbitration Signals

Signal Name	Mnemonic	Signal Type	Signal Connection Requirements
Bus request	$\overline{\text{BR}}$	Output	One per processor
Bus grant	$\overline{\text{BG}}$	Input	One per processor
Address bus busy	$\overline{\text{ABB}}$	Input/output	Common among processors
Data bus grant	$\overline{\text{DBG}}$	Input	One per processor
Data bus busy	$\overline{\text{DBB}}$	Input/output	Common among processors (One per processor if in fast-L2/data streaming mode, and data streaming across multiple processors is implemented.)

8.2.2 Address Pipelining and Split-Bus Transactions

The 604 protocol provides independent address and data bus capability to support pipelined and split-bus transaction system organizations. Address pipelining allows the address tenure of a new bus transaction to begin before the data tenure of the current transaction has finished. Split-bus transaction capability allows other bus activity to occur (either from the same master or from different masters) between the address and data tenures of a transaction.

While this capability does not inherently reduce memory latency, support for address pipelining and split-bus transactions can greatly improve effective bus/memory throughput. For this reason, these techniques are most effective in shared-memory multiprocessor implementations where bus bandwidth is an important measurement of system performance.

External arbitration is required in systems in which multiple devices must compete for the system bus. The design of the external arbiter affects pipelining by regulating the $\overline{\text{BG}}$, $\overline{\text{DBG}}$, and $\overline{\text{AACK}}$ signals. For example, a one-level pipeline is enabled by asserting $\overline{\text{AACK}}$ to the current address bus master and granting mastership of the address bus to the next requesting master before the current data bus tenure has completed. Three address tenures can occur before the current data bus tenure completes.

The 604 can pipeline its own transactions to a depth of two levels (intraprocessor pipelining); however, the 604 bus protocol does not constrain the maximum number of levels of pipelining that can occur on the bus between multiple masters (interprocessor pipelining). The external arbiter must control the pipeline depth and synchronization between masters and slaves.

In a pipelined implementation, data bus tenures are kept in strict order with respect to address tenures. However, external hardware can further decouple the address and data buses, allowing the data tenures to occur out of order with respect to the address tenures. This requires some form of system tag to associate the out-of-order data transaction with the proper originating address transaction (not defined for the 604 interface). Individual bus requests and data bus grants from each processor can be used by the system to implement tags to support interprocessor, out-of-order transactions.

The 604 supports a limited intraprocessor out-of-order, split-transaction capability via the $\overline{\text{DBWO}}$ signal. For more information about using $\overline{\text{DBWO}}$, see Section 8.11, “Using Data Bus Write Only.”

8.3 Address Bus Tenure

This section describes the three phases of the address tenure—address bus arbitration, address transfer, and address termination.

8.3.1 Address Bus Arbitration

When the 604 needs access to the external bus and does not have a qualified bus grant, it asserts bus request ($\overline{\text{BR}}$) until it is granted mastership of the bus and the bus is available (see Figure 8-4). The external arbiter must grant master-elect status to the potential master by asserting the bus grant ($\overline{\text{BG}}$) signal. The 604 requesting the bus determines that the bus is available when the $\overline{\text{ABB}}$ input is negated. When the address bus is not busy ($\overline{\text{ABB}}$ input is negated), $\overline{\text{BG}}$ is asserted and the address retry ($\overline{\text{ARTRY}}$) input is negated, and was negated the previous cycle, the 604 has what is referred to as a qualified bus grant. The 604 assumes address bus mastership by asserting $\overline{\text{ABB}}$ when it receives a qualified bus grant.

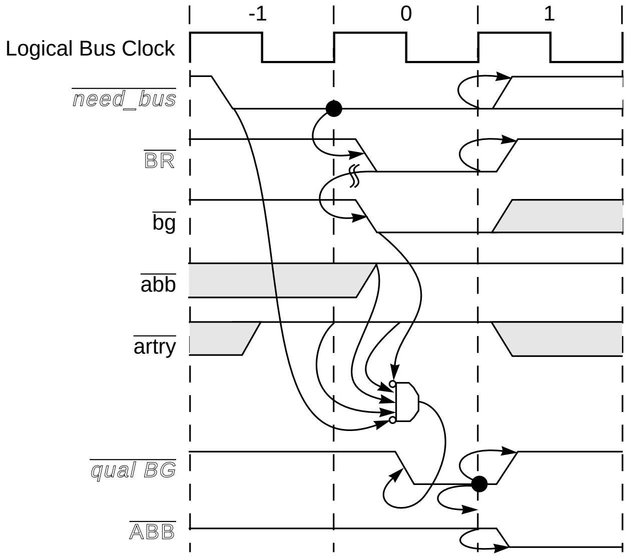


Figure 8-4. Address Bus Arbitration

External arbiters must allow only one device at a time to be the address bus master. Implementations in which no other device can be a master, $\overline{\text{BG}}$ can be grounded (always asserted) to continually grant mastership of the address bus to the 604.

If the 604 asserts $\overline{\text{BR}}$ before the external arbiter asserts $\overline{\text{BG}}$, the 604 is considered to be unparked, as shown in Figure 8-4. Figure 8-5 shows the parked case, where a qualified bus grant exists on the clock edge following a need_bus condition. Notice that the two bus clock cycles required for arbitration are eliminated if the 604 is parked, reducing overall memory latency for a transaction. The 604 always negates $\overline{\text{ABB}}$ for at least one bus clock cycle after $\overline{\text{AACK}}$ is asserted, even if it is parked and has another transaction pending.

Typically, bus parking is provided to the device that was the most recent bus master; however, system designers may choose other schemes such as providing unrequested bus grants in situations where it is easy to correctly predict the next device requesting bus mastership.

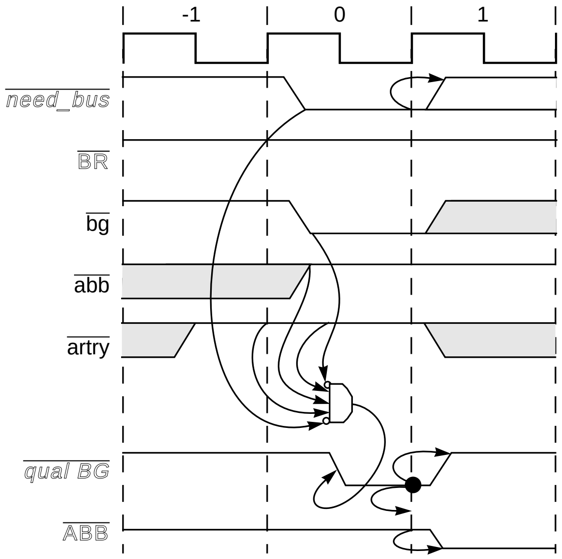


Figure 8-5. Address Bus Arbitration Showing Bus Parking

When the 604 receives a qualified bus grant, it assumes address bus mastership by asserting $\overline{\text{ABB}}$ and negating the $\overline{\text{BR}}$ output signal. Meanwhile, the 604 drives the address for the requested access onto the address bus and asserts $\overline{\text{TS}}$ to indicate the start of a new transaction.

When designing external bus arbitration logic, note that the 604 may assert $\overline{\text{BR}}$ without using the bus after it receives the qualified bus grant. For example, in a system using bus snooping, if the 604 asserts $\overline{\text{BR}}$ to perform a queued read-with-intent-to-modify-atomic (RWITMA), and the 604 snoops an access which cancels the reservation associated with the RWITMA. Once the 604 is granted the bus, it no longer needs to perform the RWITMA; therefore, the 604 does not assert $\overline{\text{ABB}}$ and does not use the bus for the read operation. Note that the 604 asserts $\overline{\text{BR}}$ for at least one clock cycle in these instances.

8.3.2 Address Transfer

During the address transfer, the physical address and all attributes of the transaction are transferred from the bus master to the slave device(s). Snooping logic may monitor the transfer to enforce cache coherency; see discussion about snooping in Section 8.3.3, "Address Transfer Termination."

The signals used in the address transfer include the following signal groups:

- Address transfer start signal: Transfer start ($\overline{\text{TS}}$)

  Note that extended address transfer start ($\overline{\text{XATS}}$) signal is used for direct-store operations and has no function for memory-mapped accesses; see Section 8.6, "Direct-Store Operation."
- Address transfer signals: Address bus (A0–A31), address parity (AP0–AP3), and address parity error ($\overline{\text{APE}}$)
- Address transfer attribute signals: Transfer type (TT0–TT4), transfer code (TC0–TC2), transfer size (TSIZ0–TSIZ2), transfer burst ($\overline{\text{TBST}}$), cache inhibit ($\overline{\text{CI}}$), write-through ($\overline{\text{WT}}$), global ($\overline{\text{GBL}}$), and cache set element (CSE0–CSE1)

Figure 8-6 shows that the timing for all of these signals, except $\overline{\text{TS}}$ and $\overline{\text{APE}}$ is identical. All of the address transfer and address transfer attribute signals are combined into the ADDR+ grouping in Figure 8-6. The $\overline{\text{TS}}$ signal indicates that the 604 has begun an address transfer and that the address and transfer attributes are valid (within the context of a synchronous bus). The 604 always asserts $\overline{\text{TS}}$ (or $\overline{\text{XATS}}$ for direct-store operations) coincident with $\overline{\text{ABB}}$. As an input, $\overline{\text{TS}}$ need not coincide with the assertion of $\overline{\text{ABB}}$ on the bus (that is, either $\overline{\text{TS}}$ or $\overline{\text{XATS}}$ can be asserted with, or on a subsequent clock cycle after $\overline{\text{ABB}}$ is asserted; the 604 tracks this transaction correctly).

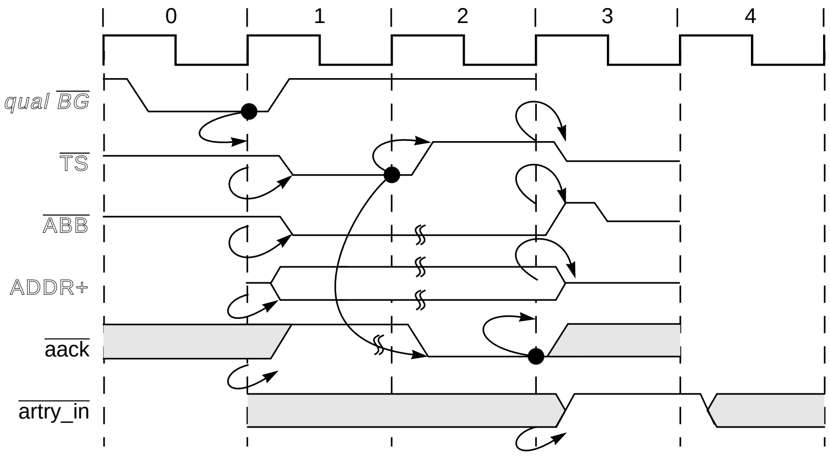


Figure 8-6. Address Bus Transfer

In Figure 8-6, the address transfer occurs during bus clock cycles 1 and 2 (arbitration occurs in bus clock cycle 0 and the address transfer is terminated in bus clock 3). In this diagram, the address bus termination input, $\overline{\text{AACK}}$, is asserted to the 604 on the bus clock following assertion of $\overline{\text{TS}}$ (as shown by the dependency line). This is the minimum duration of the address transfer for the 604; the duration can be extended by delaying the assertion of $\overline{\text{AACK}}$ for one or more bus clocks.

8.3.2.1 Address Bus Parity

The 604 always generates one bit of correct odd-byte parity for each of the four bytes of address when a valid address is on the bus. The calculated values are placed on the AP0–AP3 outputs when the 604 is the address bus master. If the 604 is not the master, $\overline{\text{TS}}$ and $\overline{\text{GBL}}$ are asserted together, and the transaction type is one that the 604 snoops (qualified condition for snooping memory operations), the calculated values are compared with the AP0–AP3 inputs. If there is an error, the $\overline{\text{APE}}$ output is asserted. If HID0[2] is set to 1, a parity error will cause a machine check if the MSR[ME] bit is set, or will cause a checkstop if the MSR[ME] bit is cleared. If HID0[2] is cleared to 0, then no action is taken. In either case, the $\overline{\text{APE}}$ signal will be asserted if even parity is detected. For more information about checkstop conditions, see Chapter 4, “Exceptions.”

8.3.2.2 Address Transfer Attribute Signals

The transfer attribute signals include several encoded signals such as the transfer type (TT0–TT4) signals, transfer burst ($\overline{\text{TBST}}$) signal, transfer size (TSIZ0–TSIZ2) signals, and transfer code (TC0–TC2) signals. Section 7.2.4, “Address Transfer Attribute Signals,” describes the encodings for the address transfer attribute signals. Note that TT0–TT4, $\overline{\text{TBST}}$, and TSIZ0–TSIZ2 have alternate functions for direct-store operations; see Section 8.6, “Direct-Store Operation.”

8.3.2.2.1 Transfer Type (TT0–TT4) Signals

Snooping logic should fully decode the transfer type signals if the $\overline{\text{GBL}}$ signal is asserted. Slave devices can sometimes use the individual transfer type signals without fully decoding the group. For a complete description of the encoding for TT0–TT4 signals, refer to Table 7-1.

8.3.2.2.2 Transfer Size (TSIZ0–TSIZ2) Signals

The transfer size signals (TSIZ0–TSIZ2) indicate the size of the requested data transfer as shown in Table 8-2. The TSIZ0–TSIZ2 signals may be used along with $\overline{\text{TBST}}$ and A29–A31 to determine which portion of the data bus contains valid data for a write transaction or which portion of the bus should contain valid data for a read transaction. Note that for a burst transaction (as indicated by the assertion of $\overline{\text{TBST}}$) TSIZ0–TSIZ2 are always set to 0b010. Therefore, if the $\overline{\text{TBST}}$ signal is asserted (except in cases of direct-store operations, or operations involving the use of **eciwx** or **ecowx** instructions), the memory system should transfer a total of eight words (32 bytes), regardless of the TSIZ0–TSIZ2 encoding.

Table 8-2. Transfer Size Signal Encodings

$\overline{\text{TBST}}$	TSIZ0	TSIZ1	TSIZ2	Transfer Size
Asserted	0	1	0	Eight-word burst
Negated	0	0	0	Eight bytes
Negated	0	0	1	One byte
Negated	0	1	0	Two bytes
Negated	0	1	1	Three bytes
Negated	1	0	0	Four bytes
Negated	1	0	1	Five bytes
Negated	1	1	0	Six bytes
Negated	1	1	1	Seven bytes

The basic coherency size of the bus is defined to be 32 bytes (corresponding to one cache line). Data transfers that cross an aligned, 32-byte boundary either must present a new address onto the bus at that boundary (for coherency consideration) or must operate as noncoherent data with respect to the 604.

8.3.2.3 Burst Ordering During Data Transfers

During burst data transfer operations, 32 bytes of data (one cache line) are transferred to or from the cache in order. Burst write transfers are always performed zero-double-word-first, but since burst reads are performed critical-double-word-first, a burst read transfer may not start with the first double word of the cache line, and the cache line fill may wrap around the end of the cache line. Table 8-3 describes the various burst orderings for the 604.

Table 8-3. PowerPC 604 Microprocessor Burst Ordering

Data Transfer	For Starting Address:			
	A27–A28 = 00	A27–A28 = 01	A27–A28 = 10	A27–A28 = 11
First data beat	DW0	DW1	DW2	DW3
Second data beat	DW1	DW2	DW3	DW0
Third data beat	DW2	DW3	DW0	DW1
Fourth data beat	DW3	DW0	DW1	DW2

Note: A29–A31 are always 0b000 for burst transfers by the 604.

8.3.2.4 Effect of Alignment in Data Transfers

Table 8-4 lists the aligned transfers that can occur on the 604 bus. These are transfers in which the data is aligned to an address that is an integer multiple of the size of the data. For example, Table 8-4 shows that one-byte data is always aligned; however, for a four-byte word to be aligned, it must be oriented on an address that is a multiple of four.

Table 8-4. Aligned Data Transfers

Transfer Size	TSIZ0	TSIZ1	TSIZ2	A29–A31	Data Bus Byte Lane(s)							
					0	1	2	3	4	5	6	7
Byte	0	0	1	000	√	—	—	—	—	—	—	—
	0	0	1	001	—	√	—	—	—	—	—	—
	0	0	1	010	—	—	√	—	—	—	—	—
	0	0	1	011	—	—	—	√	—	—	—	—
	0	0	1	100	—	—	—	—	√	—	—	—
	0	0	1	101	—	—	—	—	—	√	—	—
	0	0	1	110	—	—	—	—	—	—	√	—
	0	0	1	111	—	—	—	—	—	—	—	√
Half word	0	1	0	000	√	√	—	—	—	—	—	—
	0	1	0	010	—	—	√	√	—	—	—	—
	0	1	0	100	—	—	—	—	√	√	—	—
	0	1	0	110	—	—	—	—	—	—	√	√
Word	1	0	0	000	√	√	√	√	—	—	—	—
	1	0	0	100	—	—	—	—	√	√	√	√
Double word	0	0	0	000	√	√	√	√	√	√	√	√

The 604 supports misaligned memory operations, although their use may substantially degrade performance. Misaligned memory transfers address memory that is not aligned to the size of the data being transferred (such as, a word read of an odd byte address). Although most of these operations hit in the primary cache (or generate burst memory operations if they miss), the 604 interface supports misaligned transfers within a word (32-bit aligned) boundary, as shown in Table 8-5. Note that the four-byte transfer in Table 8-5 is only one example of misalignment. As long as the attempted transfer does not cross a word boundary, the 604 can transfer the data on the misaligned address (for example, a half-word read from an odd byte-aligned address). An attempt to address data that crosses a word boundary requires two bus transfers to access the data.

Due to the performance degradations associated with misaligned memory operations, they are best avoided. In addition to the double-word straddle boundary condition, the address translation logic can generate substantial exception overhead when the load/store multiple and load/store string instructions access misaligned data. It is strongly recommended that software attempt to align code and data where possible.

Table 8-5. Misaligned Data Transfers (Four-Byte Examples)

Transfer Size (Four Bytes)	TSIZ(0–2)	A29–A31	Data Bus Byte Lanes							
			0	1	2	3	4	5	6	7
Aligned	1 0 0	0 0 0	A	A	A	A	—	—	—	—
Misaligned—first access	0 1 1	0 0 1		A	A	A	—	—	—	—
second access	0 0 1	1 0 0	—	—	—	—	A	—	—	—
Misaligned—first access	0 1 0	0 1 0	—	—	A	A	—	—	—	—
second access	0 1 0	1 0 0	A	A	—	—	—	—	—	—
Misaligned—first access	0 0 1	0 1 1	—	—	—	A	—	—	—	—
second access	0 1 1	1 0 0	—	—	—	—	A	A	A	—
Aligned	1 0 0	1 0 0	A	A	A	A	—	—	—	—
Misaligned—first access	0 1 1	1 0 1	—	—	—	—	—	A	A	A
second access	0 0 1	0 0 0	A	—	—	—	—	—	—	—
Misaligned—first access	0 1 0	1 1 0	—	—	—	—	—	—	A	A
second access	0 1 0	0 0 0	A	A	—	—	—	—	—	—
Misaligned—first access	0 0 1	1 1 1	—	—	—	—	—	—	—	A
second access	0 1 1	0 0 0	A	A	A	—	—	—	—	—

A: Byte lane used
—: Byte lane not used

8.3.2.4.1 Alignment of External Control Instructions

The size of the data transfer associated with the **eciwx** and **ecowx** instructions is always four bytes. However, if the **eciwx** or **ecowx** instruction is misaligned and crosses any word boundary, the 604 will generate two bus operations, each with a size of fewer than four bytes. For the first bus operation, bits A29–A31 equals bits 29–31 of the data, which will be 0b101, 0b110, or 0b111. The size associated with the first bus operation will be 3, 2, or 1 bytes, respectively. For the second bus operation, bits A29–A31 equal 0b000, and the size associated with the operation will be 1, 2, or 3 bytes, respectively. For both operations, $\overline{\text{TBST}}$ and TSIZ0–TSIZ2 are redefined to specify the resource ID (RID). The resource ID is copied from bits 28–31 of the external access register (EAR). For **eciwx**/**ecowx** operations, the state of bit 28 of the EAR is presented by the $\overline{\text{TBST}}$ signal without inversion (if EAR[28] = 1, $\overline{\text{TBST}}$ = 1). The size of the second bus operation cannot be deduced from the operation itself; the system must determine how many bytes were transferred on the first bus operation to determine the size of the second operation.

Furthermore, the two bus operations associated with such a misaligned external control instruction are not atomic. That is, the 604 may initiate other types of memory operations between the two transfers. Also, the two bus operations associated with a misaligned **ecowx** may be interrupted by an **eciwx** bus operation, and vice versa. The 604 does guarantee that the two operations associated with a misaligned **ecowx** will not be interrupted by another **ecowx** operation; and likewise for **eciwx**.

Because a misaligned external control address is considered a programming error, the system may choose some means to cause an exception, typically by asserting $\overline{\text{TEA}}$ to cause a machine check exception or $\overline{\text{INT}}$ to cause an external interrupt, when a misaligned external control bus operation occurs.

8.3.2.5 Transfer Code (TC0–TC2) Signals

The TC0–TC2 signals provide supplemental information about the corresponding address. Note that the TC*x* signals can be used with the $\overline{\text{WT}}$, TT0–TT4 and $\overline{\text{TBST}}$ signals to further define the current transaction. When asserted, the transfer codes have the following meanings:

- TC0
  - Read cycle: indicates code fetch
  - Write cycle: de-allocation from L1 cache
- TC1
  - Write cycle: indicates new cache state is shared
- TC2
  - Read and write cycle: indicates allocation cycle utilized a copy-back buffer

Table 8-6 shows the supplemental information provided by the TC0–TC2 and $\overline{\text{WT}}$ signals.

Table 8-6. Transfer Code Encoding

TT Type Code	$\overline{\text{WT}}$	TC0	TC1	TC2	Operation
Write with kill	1	1	0	0	Cache copyback
Write with kill	0	1	0	0	Block invalidate (**dcbf)**
Write with kill	0	0	0	0	Block clean (**dcbst**)
Write with kill	0	0	1	0	Snoop push (read operation)
Write with kill	0	1	0	0	Snoop push (read-with-intent-to-modify)
Write with kill	0	0	0	0	Snoop push (clean operation)
Write with kill	0	1	0	0	Snoop push (flush operation)
Kill block	x	1	0	0	Kill block de-allocate (**dcbi**)

Table 8-6. Transfer Code Encoding (Continued)

TT Type Code	$\overline{WT}$	TC0	TC1	TC2	Operation
Kill block	1	0	0	0	Kill block and allocate, no cast out required (**dcbz**)
Kill block	1	0	0	1	Kill block and allocate, cast out required (**dcbz**)
Kill block	1	0	0	0	Kill block, write to shared block
Read[1]	W[3]	0	x	0	Data read, cast out required
Read	W[3]	0	x	1	Data read, cast out required
Read	W[3]	1	x	0	Instruction read
Instruction cache block invalidate	x	1	0	0	Kill block de-allocate (**icbi**)[2]

Note: 1. Read encompasses all of the read or read-with-intent-to-modify operations, both normal and atomic.
2. The **icbi** instruction is distinguished from kill block by assertion of the TT4 bit.
3. Value determined by write-through bit from translation.

8.3.3 Address Transfer Termination

The address tenure of a bus operation is terminated when completed with the assertion of $\overline{\text{AACK}}$, or retried with the assertion of $\overline{\text{ARTRY}}$. The $\overline{\text{SHD}}$ signal may also be asserted either coincident with the $\overline{\text{ARTRY}}$ signal, or alone to indicate that a copy of the requested data exists in one of the devices on the bus, and that the requesting device should mark the data as shared in its cache. The 604 does not terminate the address transfer until the $\overline{\text{AACK}}$ (address acknowledge) input is asserted; therefore, the system can extend the address transfer phase by delaying the assertion of $\overline{\text{AACK}}$ to the 604. $\overline{\text{AACK}}$ can be asserted as early as the bus clock cycle following $\overline{\text{TS}}$ (see Figure 8-7), which allows a minimum address tenure of two bus cycles. As shown in Figure 8-7, these signals are asserted for one bus clock cycle, three-stated for half of the next bus clock cycle, driven high till the following bus cycle, and finally three-stated. Note that $\overline{\text{AACK}}$ must be asserted for only one bus clock cycle.

The address transfer can be terminated with the requirement to retry if $\overline{\text{ARTRY}}$ is asserted anytime during the address tenure and through the cycle following $\overline{\text{AACK}}$. The assertion causes the entire transaction (address and data tenure) to be rerun. As a snooping device, the 604 asserts $\overline{\text{ARTRY}}$ for a snooped transaction that hits modified data in the data cache that must be written back to memory, or if the snooped transaction could not be serviced. As a bus master, the 604 responds to an assertion of $\overline{\text{ARTRY}}$ by aborting the bus transaction and re-requesting the bus. Note that after recognizing an assertion of $\overline{\text{ARTRY}}$ and aborting the transaction in progress, the 604 is not guaranteed to run the same transaction the next time it is granted the bus.

If an address retry is required, the $\overline{\text{ARTRY}}$ response will be asserted by a bus snooping device as early as the second cycle after the assertion of $\overline{\text{TS}}$. Once asserted, $\overline{\text{ARTRY}}$ must remain asserted through the cycle after the assertion of $\overline{\text{AACK}}$. The assertion of $\overline{\text{ARTRY}}$ during the cycle after the assertion of $\overline{\text{AACK}}$ is referred to as a qualified $\overline{\text{ARTRY}}$. An earlier assertion of $\overline{\text{ARTRY}}$ during the address tenure is referred to as an early $\overline{\text{ARTRY}}$.

As a bus master, the 604 recognizes either an early or qualified $\overline{\text{ARTRY}}$ and prevents the data tenure associated with the retried address tenure. If the data tenure has already begun, the 604 aborts and terminates the data tenure immediately even if the burst data has been received. If the assertion of $\overline{\text{ARTRY}}$ is received up to or on the bus cycle following the first (or only) assertion of $\overline{\text{TA}}$ for the data tenure, the 604 ignores the first data beat, and if it is a load operation, does not forward data internally to the cache and execution units.

If the 604 is in fast-L2/data streaming mode, $\overline{\text{TA}}$ should not be asserted prior to the qualified $\overline{\text{ARTRY}}$ cycle. If $\overline{\text{ARTRY}}$ is asserted after the first (or only) assertion of $\overline{\text{TA}}$, improper operation of the bus interface may result.

During the clock of a qualified $\overline{\text{ARTRY}}$, the 604 also determines if it should negate $\overline{\text{BR}}$ and ignore $\overline{\text{BG}}$ on the following cycle. On the following cycle, only the snooping master that asserted $\overline{\text{ARTRY}}$ and needs to perform a snoop copy-back operation is allowed to assert $\overline{\text{BR}}$. This guarantees the snooping master an opportunity to request and be granted the bus before the just-retried master can restart its transaction.

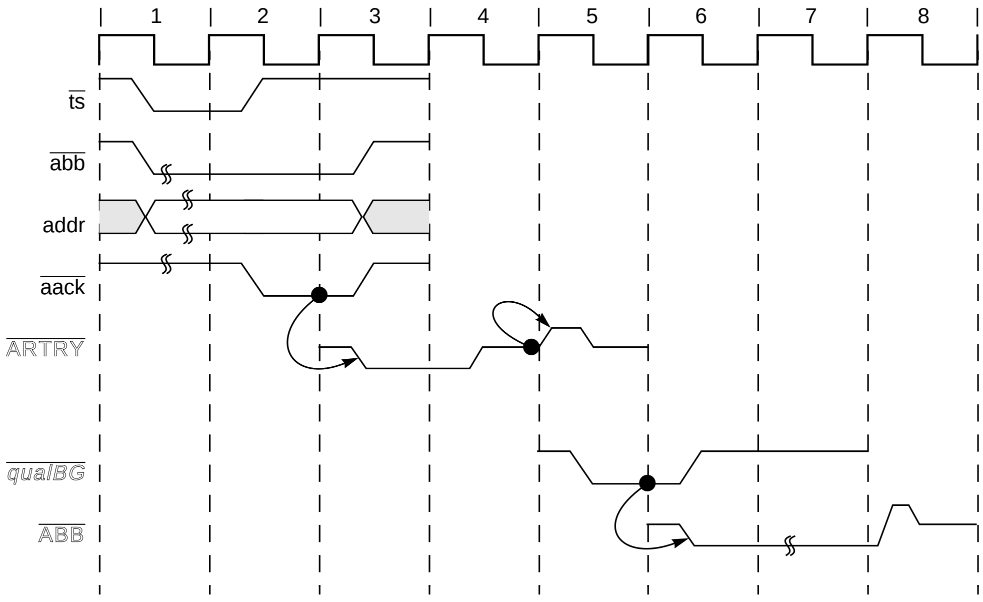


Figure 8-7. Snooped Address Cycle with $\overline{\text{ARTRY}}$

8.4 Data Bus Tenure

This section describes the data bus arbitration, transfer, and termination phases defined by the 604 memory access protocol. The phases of the data tenure are identical to those of the address tenure, underscoring the symmetry in the control of the two buses.

8.4.1 Data Bus Arbitration

Data bus arbitration uses the data arbitration signal group—$\overline{\text{DBG}}$, $\overline{\text{DBWO}}$, and $\overline{\text{DBB}}$. Additionally, the combination of $\overline{\text{TS}}$ or $\overline{\text{XATS}}$ and TT0–TT4 provides information about the data bus request to external logic.

The $\overline{\text{TS}}$ signal is an implied data bus request from the 604; the arbiter must qualify $\overline{\text{TS}}$ with the transfer type (TT) encodings to determine if the current address transfer is an address-only operation, which does not require a data bus transfer (see Figure 8-7). If the data bus is needed, the arbiter grants data bus mastership by asserting the $\overline{\text{DBG}}$ input to the 604. As with the address-bus arbitration phase, the 604 must qualify the $\overline{\text{DBG}}$ input with a number of input signals before assuming bus mastership, as shown in Figure 8-8.

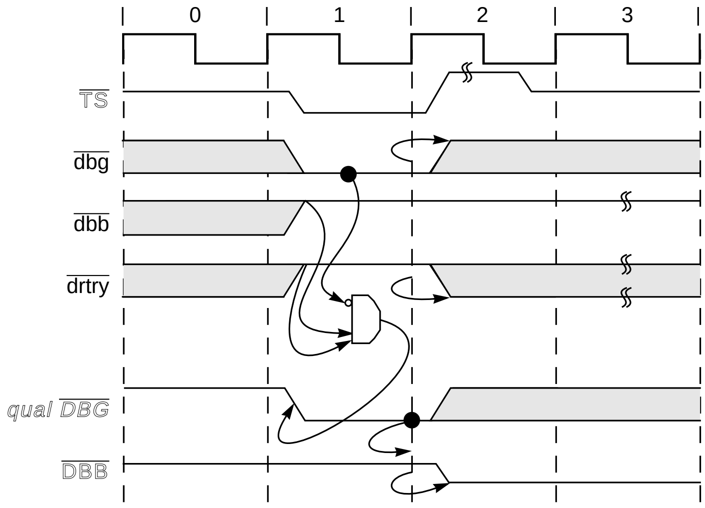


Figure 8-8. Data Bus Arbitration

A qualified data bus grant can be expressed as the following:

QDBG = $\overline{\text{DBG}}$ asserted while $\overline{\text{DBB}}$, $\overline{\text{DRTRY}}$, and $\overline{\text{ARTRY}}$ (associated with the data bus operation) are negated.

When a data tenure overlaps with its associated address tenure, a qualified $\overline{\text{ARTRY}}$ assertion coincident with a data bus grant signal does not result in data bus mastership ($\overline{\text{DBB}}$ is not asserted). Otherwise, the 604 always asserts $\overline{\text{DBB}}$ on the bus clock cycle after recognition of a qualified data bus grant. Since the 604 can pipeline transactions, there may be an outstanding data bus transaction when a new address transaction is retried. In this case, the 604 becomes the data bus master to complete the previous transaction.

8.4.1.1 Effect of $\overline{\text{ARTRY}}$ Assertion on Data Transfer and Arbitration

The system designer must define the qualified snoop response window, and ensure that data is not transferred prior to one cycle before the end of that window in non–fast-L2/data streaming mode, or prior to the same cycle as the end of that window in fast-L2/data streaming mode. The 604 supports a snoop response window as early as two cycles after assertion of $\overline{\text{TS}}$. Operation of the 604 in fast-L2/data streaming mode requires that data be transferred no earlier than the first cycle of the $\overline{\text{ARTRY}}$ window, not the cycle earlier. The system may assert $\overline{\text{TA}}$ for a data transaction prior to the termination of an address tenure; in this case note that the snoop response window is closed either on the clock that $\overline{\text{TA}}$ is asserted (if in fast-L2/data streaming mode), or the clock after the assertion of $\overline{\text{TA}}$ (if in non–fast-L2/data streaming mode).

An asserted $\overline{\text{ARTRY}}$ can invalidate a previous or current data transfer and terminate the data cycle, invalidate a qualified data bus grant, or cancel a future data transfer. The possible scenarios are described below:

- If data is transferred (via assertion of $\overline{\text{TA}}$) two or more cycles before the beginning of the snoop window in non–fast-L2/data streaming mode, or one or more cycles before the beginning of the snoop window in fast-L2/data streaming, then data is transferred too early to be cancelled by $\overline{\text{ARTRY}}$. Therefore, systems in which $\overline{\text{ARTRY}}$ can be asserted must not attempt data transfers (assert $\overline{\text{TA}}$) prior to this cycle.
- If data is transferred in the cycle before the beginning of the snoop response window, assertion of $\overline{\text{ARTRY}}$ invalidates the data transfer, in a similar fashion to assertion of $\overline{\text{DRTRY}}$, except that the data tenure is aborted, not extended. If the fast-L2/data streaming mode is active, data may not be transferred in this cycle.
- If data is transferred in the first cycle of the snoop response window, assertion of $\overline{\text{ARTRY}}$ invalidates the data transfer. This is similar to deasserting $\overline{\text{TA}}$ except that the data tenure is aborted, instead of continued.
- If $\overline{\text{DBG}}$ has been asserted, the system must not attempt to transfer data in cycles following the assertion of $\overline{\text{ARTRY}}$. The 604 negates $\overline{\text{DBB}}$ the cycle following $\overline{\text{ARTRY}}$, and expects no more data to be transferred. However, note that the data related to a previous address tenure must not be affected, and that the system must distinguish this case.
- If a $\overline{\text{DBG}}$ has not been asserted, an $\overline{\text{ARTRY}}$ assertion effectively negates the implied data bus request that was associated with the address transfer, and the 604 will not expect a transfer. The system must not assert $\overline{\text{DBG}}$ for this transfer if any other 604 data transfers are pending.
- If $\overline{\text{ARTRY}}$ assertion occurs while a data transfer is in progress, the 604 will terminate data transfers following the first cycle of $\overline{\text{ARTRY}}$ assertion. This means that a burst transfer may be cut short.

- If an ARTRY assertion occurs the same cycle as its corresponding DBG, the ARTRY will disqualify the data bus grant in that cycle and the 604 will not initiate any data transaction on the following cycle regardless of whether any other data transactions are queued. However, on the following cycle (the cycle after the ARTRY assertion) the 604 processor will respond to a qualified data bus grant if it has previously queued data transactions. Figure 8-9 shows an example where a write address tenure receives an ARTRY snoop response in the same cycle the system asserts DBWO and DBG (cycle 6) to grant the write data tenure before a previously requested read data tenure. Following the ARTRY assertion, the qualified DBG assertion to the 604 in cycle 7 will be accepted for the read data tenure.

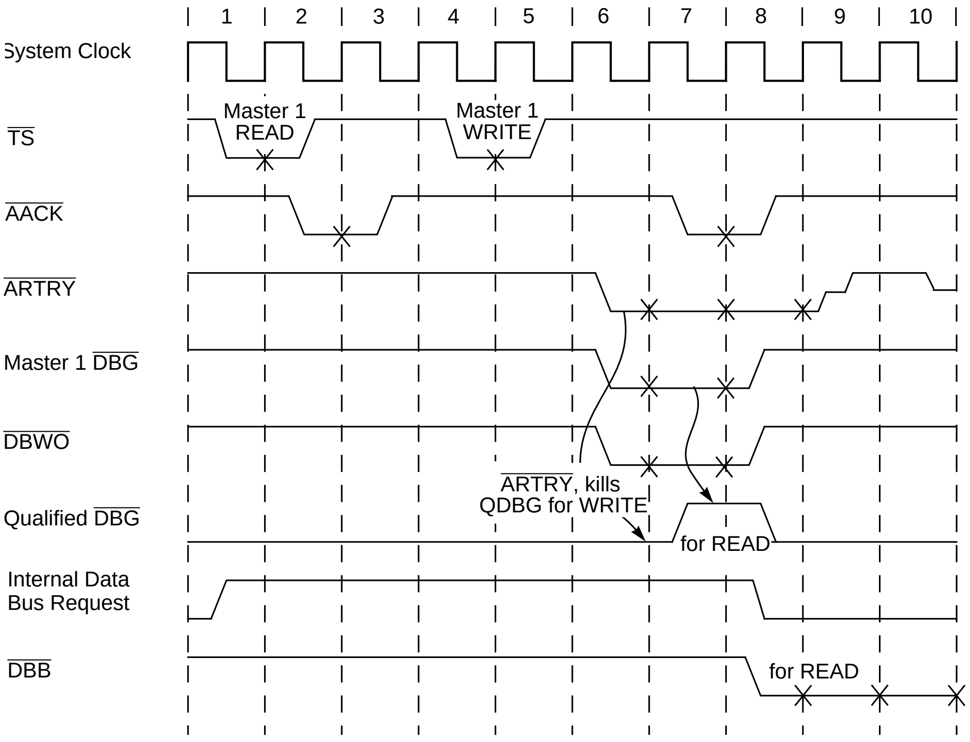


Figure 8-9. Qualified DBG Generation Following ARTRY

8.4.1.2 Using the DBB Signal

The DBB signal should be connected between masters if data tenure scheduling is left to the masters. Optionally, the memory system can control data tenure scheduling directly with DBG. However, it is possible to ignore the DBB signal in the system if the DBB input is not used as the final data bus allocation control between data bus masters, and if the memory system can track the start and end of the data tenure. In non–fast-L2/data streaming mode, if $\overline{\text{DBB}}$ is not used to signal the end of a data tenure, $\overline{\text{DBG}}$ is only asserted to the next bus master the cycle before the cycle that the next bus master may actually begin its data tenure, rather than asserting it earlier (usually during another master's data tenure) and allowing the negation of $\overline{\text{DBB}}$ to be the final gating signal for a qualified data bus grant. If the 604 is in fast-L2/data streaming mode, the $\overline{\text{DBB}}$ signal is an output only, and is not sampled by the 604. Even if $\overline{\text{DBB}}$ is ignored in the system, the 604 always recognizes its own assertion of $\overline{\text{DBB}}$ (except when in fast-L2/data streaming mode), and requires one cycle after data tenure completion to negate its own $\overline{\text{DBB}}$ before recognizing a qualified data bus grant for another data tenure. If the $\overline{\text{DBB}}$ signal is not used by the system, $\overline{\text{DBB}}$ must still be connected to a pull-up resistor on the 604 to ensure proper operation. If the 604 is in fast-L2/data streaming mode, and data streaming is to be performed across multiple processors, the $\overline{\text{DBB}}$ signal for each processor should be connected directly to the memory arbiter.

8.4.2 Data Bus Write Only

As a result of address pipelining, the 604 may have up to three data tenures queued to perform when it receives a qualified $\overline{\text{DBG}}$. Generally, the data tenures should be performed in strict order (the same order) as their address tenures were performed. The 604, however, also supports a limited out-of-order capability with the data bus write only ($\overline{\text{DBWO}}$) input. The $\overline{\text{DBWO}}$ capability exists to alleviate deadlock conditions that are possible in certain system topologies. When recognized on the clock of a qualified $\overline{\text{DBG}}$, $\overline{\text{DBWO}}$ may direct the 604 to perform the next pending data write tenure even if a pending read tenure would have normally been performed first. For more information on the operation of $\overline{\text{DBWO}}$, refer to Section 8.11, "Using Data Bus Write Only."

If the 604 has any data tenures to perform, it always accepts data bus mastership to perform a data tenure when it recognizes a qualified $\overline{\text{DBG}}$. If $\overline{\text{DBWO}}$ is asserted with a qualified $\overline{\text{DBG}}$ and no write tenure is queued to run, the 604 still takes mastership of the data bus to perform the next pending read data tenure. If the 604 has multiple queued writes, the assertion of $\overline{\text{DBWO}}$ causes the reordering of the write operation whose address was sent first.

Generally, $\overline{\text{DBWO}}$ should only be used to allow a copy-back operation (burst write) to occur before a pending read operation. If $\overline{\text{DBWO}}$ is used for single-beat write operations, it may negate the effect of the **eieio** instruction by allowing a write operation to precede a program-scheduled read operation. If $\overline{\text{DBWO}}$ is asserted when the 604 does not have write data available, bus operations occur as if $\overline{\text{DBWO}}$ had not been asserted.

8.4.3 Data Transfer

The data transfer signals include DH0–DH31, DL0–DL31, DP0–DP7 and $\overline{\text{DPE}}$. For memory accesses, the DH and DL signals form a 64-bit data path for read and write operations.

The 604 transfers data in either single- or four-beat burst transfers. Single-beat operations can transfer from one to eight bytes at a time and can be misaligned; see Section 8.3.2.4, “Effect of Alignment in Data Transfers.” Burst operations always transfer eight words and are aligned on eight-word address boundaries. Burst transfers can achieve significantly higher bus throughput than single-beat operations.

The type of transaction initiated by the 604 depends on whether the code or data is cacheable and, for store operations whether the cache is considered in write-back or write-through mode, which software controls on either a page or block basis. Burst transfers support cacheable operations only; that is, memory structures must be marked as cacheable (and write-back for data store operations) in the respective page or block descriptor to take advantage of burst transfers.

The 604 output $\overline{\text{TBST}}$ indicates to the system whether the current transaction is a single- or four-beat transfer (except during **eciwx**/**ecowx** transactions, when it signals the state of EAR[28]). A burst transfer has an assumed address order. For load or store operations that missed in the cache (and are marked as cacheable and, for stores, write-back in the MMU), the 604 uses the double-word–aligned address associated with the critical code or data that initiated the transaction. This minimizes latency by allowing the critical code or data to be forwarded to the processor before the rest of the cache line is filled. For all other burst operations, however, the cache line write operations are transferred beginning with the oct-word–aligned data, and burst reads begin on double-word boundaries.

The 604 does not directly support dynamic interfacing to subsystems with less than a 64-bit data path (except for direct-store operations discussed in Section 8.6, “Direct-Store Operation”).

8.4.4 Data Transfer Termination

Four signals are used to terminate data bus transactions—$\overline{\text{TA}}$, $\overline{\text{DRTRY}}$ (data retry), $\overline{\text{TEA}}$ (transfer error acknowledge), and $\overline{\text{ARTRY}}$. The $\overline{\text{TA}}$ signal indicates normal termination of data transactions. It must always be asserted on the bus cycle coincident with the data that it is qualifying. It may be withheld by the slave for any number of clocks until valid data is ready to be supplied or accepted. $\overline{\text{DRTRY}}$ indicates invalid read data in the previous bus clock cycle. $\overline{\text{DRTRY}}$ extends the current data beat and does not terminate it. If it is asserted after the last (or only) data beat, the 604 negates $\overline{\text{DBB}}$ but still considers the data beat active and waits for another assertion of $\overline{\text{TA}}$. $\overline{\text{DRTRY}}$ is ignored on write operations. $\overline{\text{TEA}}$ indicates a nonrecoverable bus error event. Upon receiving a final (or only) termination condition, the 604 always negates $\overline{\text{DBB}}$ for one cycle, except when data streaming in fast-L2/data streaming mode.

If $\overline{\text{DRTRY}}$ is asserted by the memory system to extend the last (or only) data beat past the negation of $\overline{\text{DBB}}$, the memory system should three-state the data bus on the clock after the final assertion of $\overline{\text{TA}}$, even though it will negate $\overline{\text{DRTRY}}$ on that clock. This is to prevent a potential momentary data bus conflict if a write access begins on the following cycle.

The $\overline{\text{TEA}}$ signal is used to signal a nonrecoverable error during the data transaction. The $\overline{\text{TEA}}$ signal will be recognized anytime during the assertion of $\overline{\text{DBB}}$ or when a valid $\overline{\text{DRTRY}}$ could be sampled. The assertion of $\overline{\text{TEA}}$ terminates the data tenure immediately even if in the middle of a burst; however, it does not prevent incorrect data that has just been acknowledged with $\overline{\text{TA}}$ from being written into the 604's cache or GPRs. The assertion of $\overline{\text{TEA}}$ initiates either a machine check exception or a checkstop condition based on the setting of the MSR.

An assertion of $\overline{\text{ARTRY}}$ causes the data tenure to be terminated immediately if the $\overline{\text{ARTRY}}$ is for the address tenure associated with the data tenure in operation (the data tenure may not be terminated due to address pipelining). If $\overline{\text{ARTRY}}$ is connected for the 604, the earliest allowable assertion of $\overline{\text{TA}}$ to the 604 is directly dependent on the earliest possible assertion of $\overline{\text{ARTRY}}$ to the 604; see Section 8.3.3, "Address Transfer Termination."

8.4.4.1 Normal Single-Beat Termination

Normal termination of a single-beat data read operation occurs when $\overline{\text{TA}}$ is asserted by a responding slave. The $\overline{\text{TEA}}$ and $\overline{\text{DRTRY}}$ signals must remain negated during the transfer (see Figure 8-10).

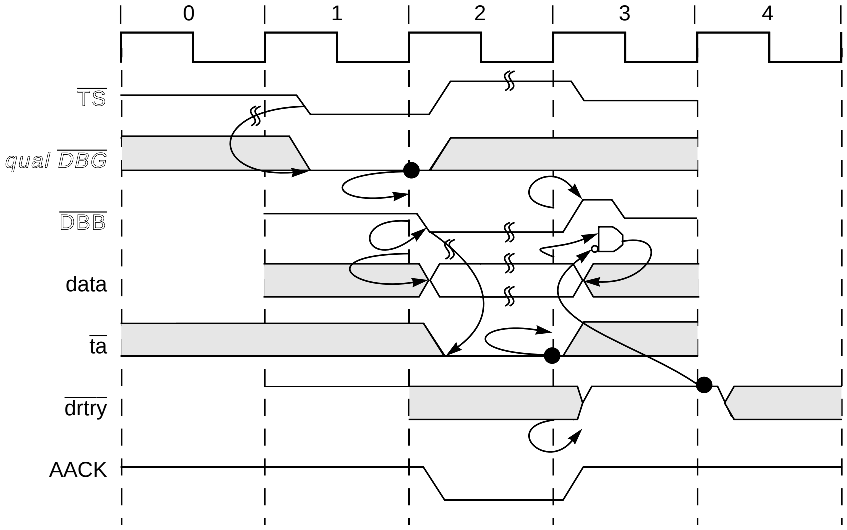


Figure 8-10. Normal Single-Beat Read Termination

The $\overline{\text{DRTRY}}$ signal is not sampled during data writes, as shown in Figure 8-11.

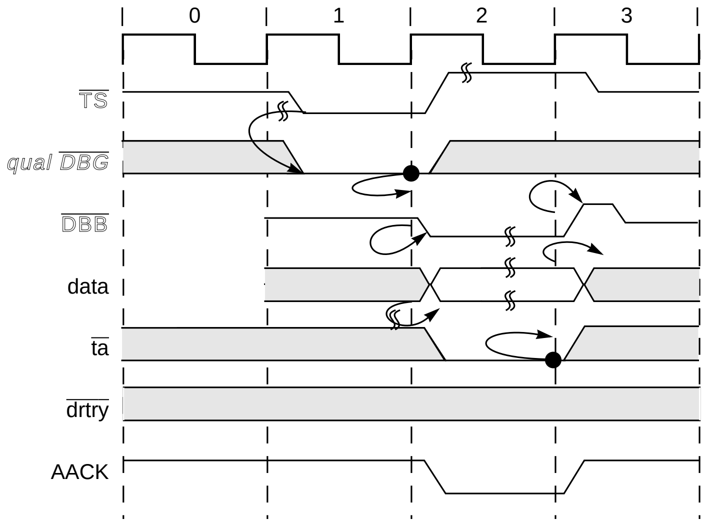


Figure 8-11. Normal Single-Beat Write Termination

Normal termination of a burst transfer occurs when $\overline{\text{TA}}$ is asserted for four bus clock cycles, as shown in Figure 8-12. The bus clock cycles in which $\overline{\text{TA}}$ is asserted need not be consecutive, thus allowing pacing of the data transfer beats. For read bursts to terminate successfully, $\overline{\text{TEA}}$ and $\overline{\text{DRTRY}}$ must remain negated during the transfer. For write bursts, $\overline{\text{TEA}}$ must remain negated for a successful transfer. $\overline{\text{DRTRY}}$ is ignored during data writes.

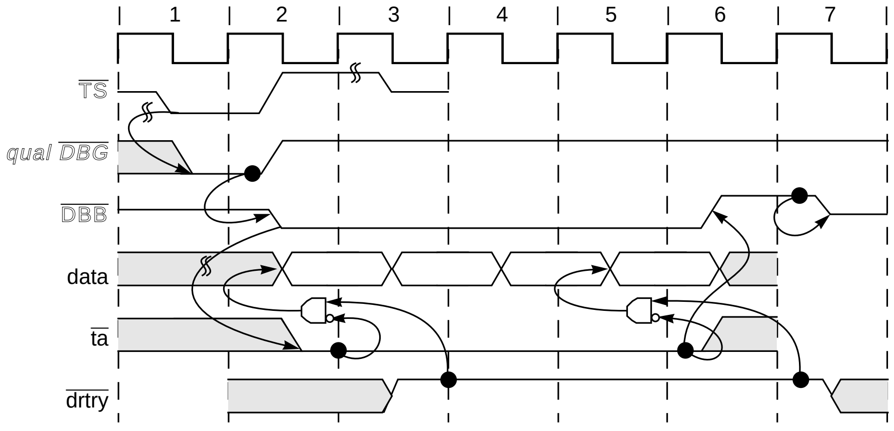


Figure 8-12. Normal Burst Transaction

For read bursts, $\overline{\text{DRTRY}}$ may be asserted one bus clock cycle after $\overline{\text{TA}}$ is asserted to signal that the data presented with $\overline{\text{TA}}$ is invalid and that the processor must wait for the negation of $\overline{\text{DRTRY}}$ before forwarding data to the processor (see Figure 8-13). Thus, a data beat can be speculatively terminated with $\overline{\text{TA}}$ and then one bus clock cycle later confirmed with the negation of $\overline{\text{DRTRY}}$. The $\overline{\text{DRTRY}}$ signal is valid only for read transactions. $\overline{\text{TA}}$ must be asserted on the bus clock cycle before the first bus clock cycle of the assertion of $\overline{\text{DRTRY}}$; otherwise the results are undefined.

The $\overline{\text{DRTRY}}$ signal extends data bus mastership such that other processors cannot use the data bus until $\overline{\text{DRTRY}}$ is negated. Therefore, in the example in Figure 8-13, $\overline{\text{DBB}}$ cannot be asserted until bus clock cycle 5. This is true for both read and write operations even though $\overline{\text{DRTRY}}$ does not extend bus mastership for write operations.

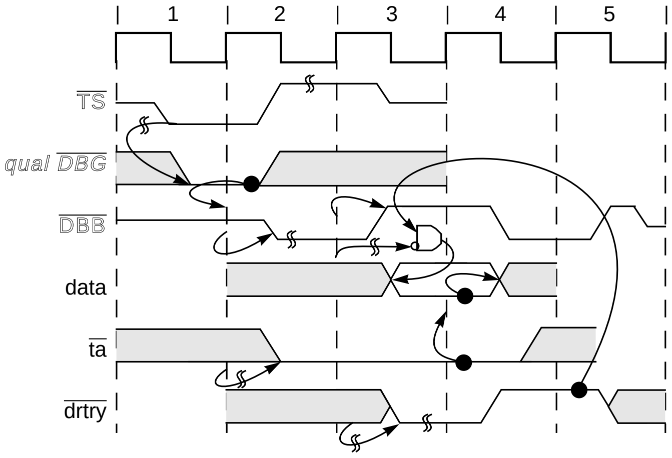


Figure 8-13. Termination with $\overline{\text{DRTRY}}$

Figure 8-14 shows the effect of using $\overline{\text{DRTRY}}$ during a burst read. It also shows the effect of using $\overline{\text{TA}}$ to pace the data transfer rate. Notice that in bus clock cycle 3 of Figure 8-14, $\overline{\text{TA}}$ is negated for the second data beat. The 604 data pipeline does not proceed until bus clock cycle 4 when the $\overline{\text{TA}}$ is reasserted.

Note that $\overline{\text{DRTRY}}$ is useful for systems that implement speculative forwarding of data such as those with direct-mapped, second-level caches where hit/miss is determined on the following bus clock cycle, or for parity- or ECC-checked memory systems.

Note that $\overline{\text{DRTRY}}$ may not be implemented on other PowerPC processors.

8.4.4.2 Data Transfer Termination Due to a Bus Error

The $\overline{\text{TEA}}$ signal indicates that a bus error occurred. It may be asserted while $\overline{\text{DBB}}$ is asserted or when a valid $\overline{\text{DRTRY}}$ could be recognized by the 604. Asserting $\overline{\text{TEA}}$ to the 604 terminates the transaction; that is, further assertions of $\overline{\text{TA}}$ and $\overline{\text{DRTRY}}$ are ignored and $\overline{\text{DBB}}$ is negated. If the system asserts $\overline{\text{TEA}}$ for a data transaction on the same cycle or before $\overline{\text{ARTRY}}$ is asserted for the corresponding address transaction, the 604 will ignore the effects of $\overline{\text{ARTRY}}$ on the address transaction and will consider it successfully completed.

Note that from a bus standpoint, the assertion of $\overline{\text{TEA}}$ causes nothing worse than the early termination of the data tenure in progress. All the system logic involved in processing the data transfer prior to the $\overline{\text{TEA}}$ must return to the normal nonbusy state following the $\overline{\text{TEA}}$ so that the bus operations associated with a machine check exception can proceed. Due to bus pipelining in the 604, all outstanding bus operations, including all queued requests, are completed in the normal fashion following the $\overline{\text{TEA}}$. The machine check exception can be taken while these transactions are in progress.

If the $\overline{\text{TEA}}$ signal is asserted during a direct-store access, the action of the $\overline{\text{TEA}}$ is delayed until all data transfers from the direct store access have been completed. The device causing assertion of the $\overline{\text{TEA}}$ signal is responsible for maintaining assertion of the $\overline{\text{TEA}}$ signal until the last direct-store data tenure is complete. The direct store reply, in cases of $\overline{\text{TEA}}$ assertion, is not required, and will be ignored by the 604. The 604 will recognize the assertion of the $\overline{\text{TEA}}$ signal at the completion of the last direct-store data tenure, and not before.

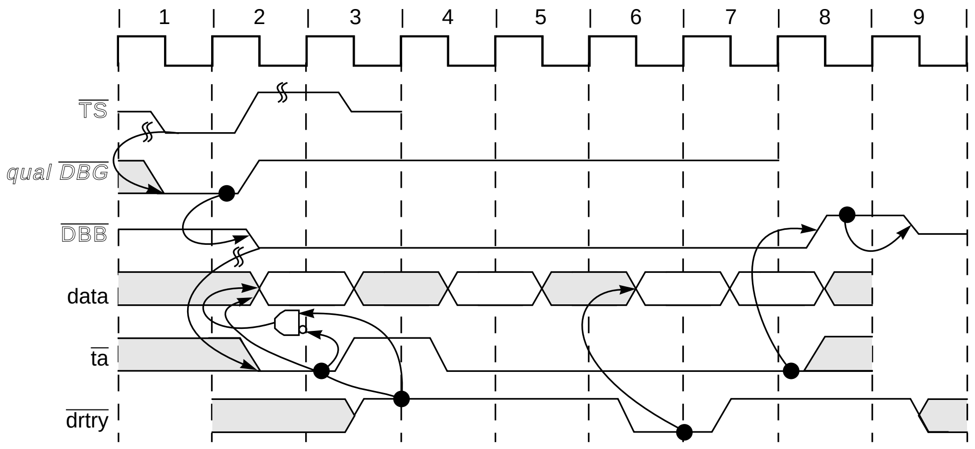


Figure 8-14. Read Burst with $\overline{\text{TA}}$ Wait States and $\overline{\text{DRTRY}}$

Assertion of the $\overline{\text{TEA}}$ signal causes a machine check exception (and possibly a checkstop condition within the 604). For more information, see Section 4.5.2, “Machine Check Exception (0x00200).” Note also that the 604 does not implement a synchronous error capability for memory accesses. This means that the exception instruction pointer does not point to the memory operation that caused the assertion of $\overline{\text{TEA}}$, but to the instruction about to be executed (perhaps several instructions later). However, assertion of $\overline{\text{TEA}}$ does not invalidate data entering the GPR or the cache. Additionally, the corresponding address of the access that caused $\overline{\text{TEA}}$ to be asserted is not latched by the 604. To recover, the exception handler must determine and remedy the cause of the $\overline{\text{TEA}}$, or the 604 must be reset; therefore, this function should only be used to flag fatal system conditions to the processor (such as parity or uncorrectable ECC errors).

After the 604 has committed to run a transaction, that transaction must eventually complete. Address retry causes the transaction to be restarted; $\overline{\text{TA}}$ wait states and $\overline{\text{DRTRY}}$ assertion for reads delay termination of individual data beats. Eventually, however, the system must either terminate the transaction or assert the $\overline{\text{TEA}}$ signal (and vector the 604 into a machine check exception.) For this reason, care must be taken to check for the end of physical memory and the location of certain system facilities to avoid memory accesses that result in the generation of machine check exceptions.

Note that $\overline{\text{TEA}}$ generates a machine check exception depending on the ME bit in the MSR. Clearing the machine check exception enable control bit leads to a true checkstop condition (instruction execution halted and processor clock stopped); a machine check exception occurs if the ME bit is set.

8.4.5 Memory Coherency—MESI Protocol

The 604 provides dedicated hardware to provide memory coherency by snooping bus transactions. The address retry capability enforces the four-state, MESI cache-coherency protocol (see Figure 8-15). In addition to the hardware required to monitor bus traffic for coherency, the 604 has a cache port dedicated to snooping so that comparing cache entries to address traffic on the bus does not tie up the 604's on-chip data cache.

The global ($\overline{\text{GBL}}$) signal output, indicates whether the current transaction must be snooped by other snooping devices on the bus. Address bus masters assert $\overline{\text{GBL}}$ to indicate that the current transaction is a global access (that is, an access to memory shared by more than one processor/cache). If $\overline{\text{GBL}}$ is not asserted for the transaction, that transaction is not snooped. When other devices detect the $\overline{\text{GBL}}$ input asserted, they must respond by snooping the broadcast address.

Normally, $\overline{\text{GBL}}$ reflects the M-bit value specified for the memory reference in the corresponding translation descriptor(s). Note that care must be taken to minimize the number of pages marked as global, because the retry protocol discussed in the previous section is used to enforce coherency and can require significant bus bandwidth.

When the 604 is not the address bus master, $\overline{\text{GBL}}$ is an input. The 604 snoops a transaction if $\overline{\text{TS}}$ and $\overline{\text{GBL}}$ are asserted together in the same bus clock cycle (this is a *qualified* snooping condition). No snoop update to the 604 cache occurs if the snooped transaction is not marked global. This includes invalidation cycles.

When the 604 detects a qualified snoop condition, the address associated with the $\overline{\text{TS}}$ is compared against the data cache tags through a dedicated cache tag port. Snooping completes if no hit is detected. If, however, the address hits in the cache, the 604 reacts according to the MESI protocol shown in Figure 8-15, assuming the WIM bits are set to write-back mode, caching allowed, and coherency enforced (WIM = 001).

Note that write hits to clean lines of nonglobal pages do not generate invalidate broadcasts. There are several types of bus transactions that involve the movement of data that can no longer access the TLB M-bit (for example, replacement cache block copy-back, or a snoop push). In these cases, the hardware cannot determine whether the cache block was originally marked global; therefore, the 604 marks these transactions as nonglobal to avoid retry deadlocks.

The 604's on-chip data cache is implemented as a four-way set-associative cache. To facilitate external monitoring of the internal cache tags, the cache set element (CSE0–CSE1) signals indicate which sector of the cache set is being replaced on read operations (including RWITM). Note that these signals are valid only for 604 burst operations; for all other bus operations, the CSE0–CSE1 signals should be ignored.

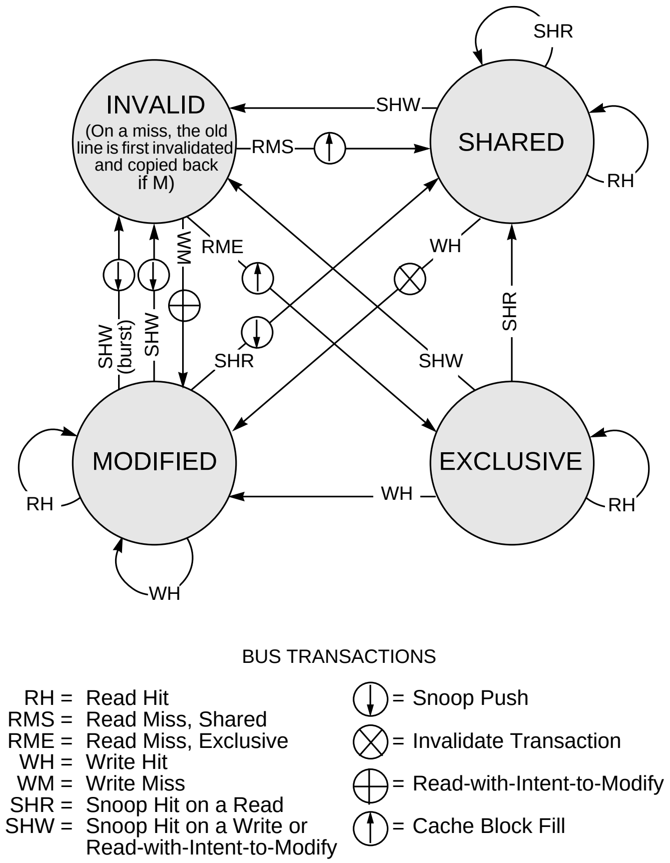


Figure 8-15. MESI Cache Coherency Protocol—State Diagram (WIM = 001)

Table 8-7 shows the CSE0–CSE1 encodings.

Table 8-7. CSE0–CSE1 Signals

CSE0–CSE1	Cache Set Element
00	Set 0
01	Set 1
10	Set 2
11	Set 3

8.5 Timing Examples

This section shows timing diagrams for various scenarios. Figure 8-16 illustrates the fastest single-beat reads possible for the 604. This figure shows both minimal latency and maximum single-beat throughput. By delaying the data bus tenure, the latency increases, but, because of split-transaction pipelining, the overall throughput is not affected unless the data bus latency causes the fourth address tenure to be delayed.

Note that all bidirectional signals are three-stated between bus tenures.

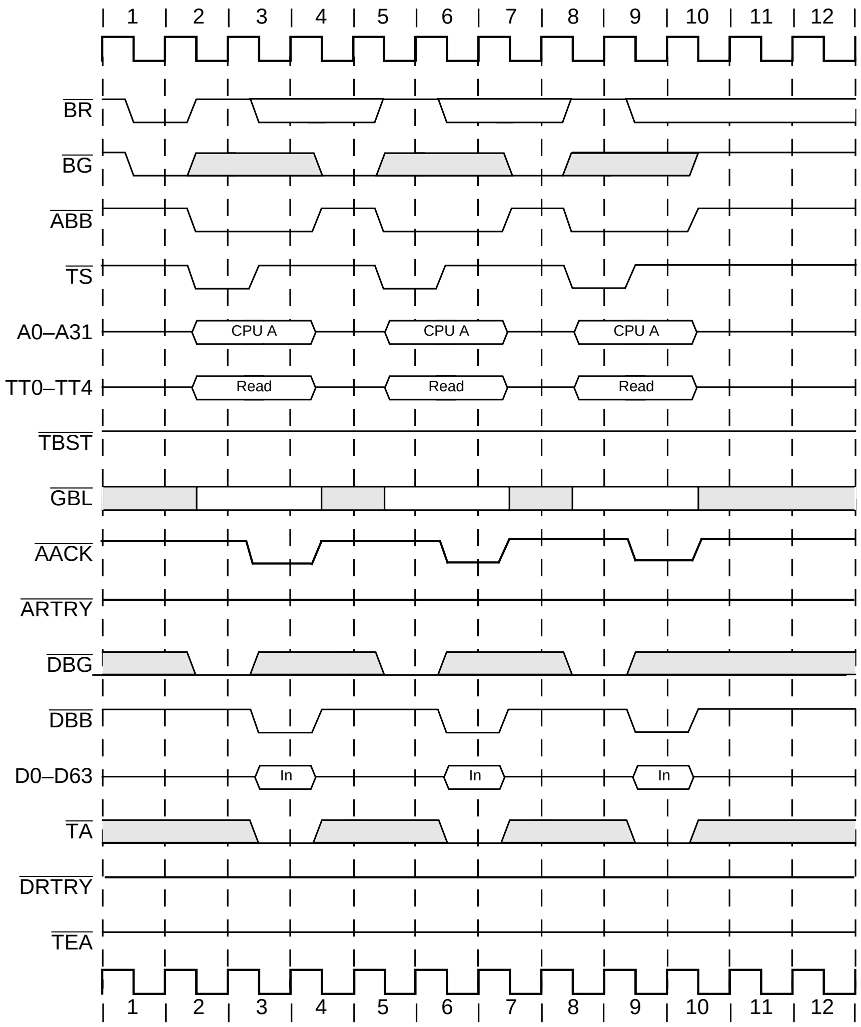


Figure 8-16. Fastest Single-Beat Reads

Figure 8-17 illustrates the fastest single-beat writes supported by the 604. Note that all bidirectional signals are three-stated between bus tenures. The TT1–TT4 signals are binary encoded 0bx0010, and TT0 can be either 0 or 1.

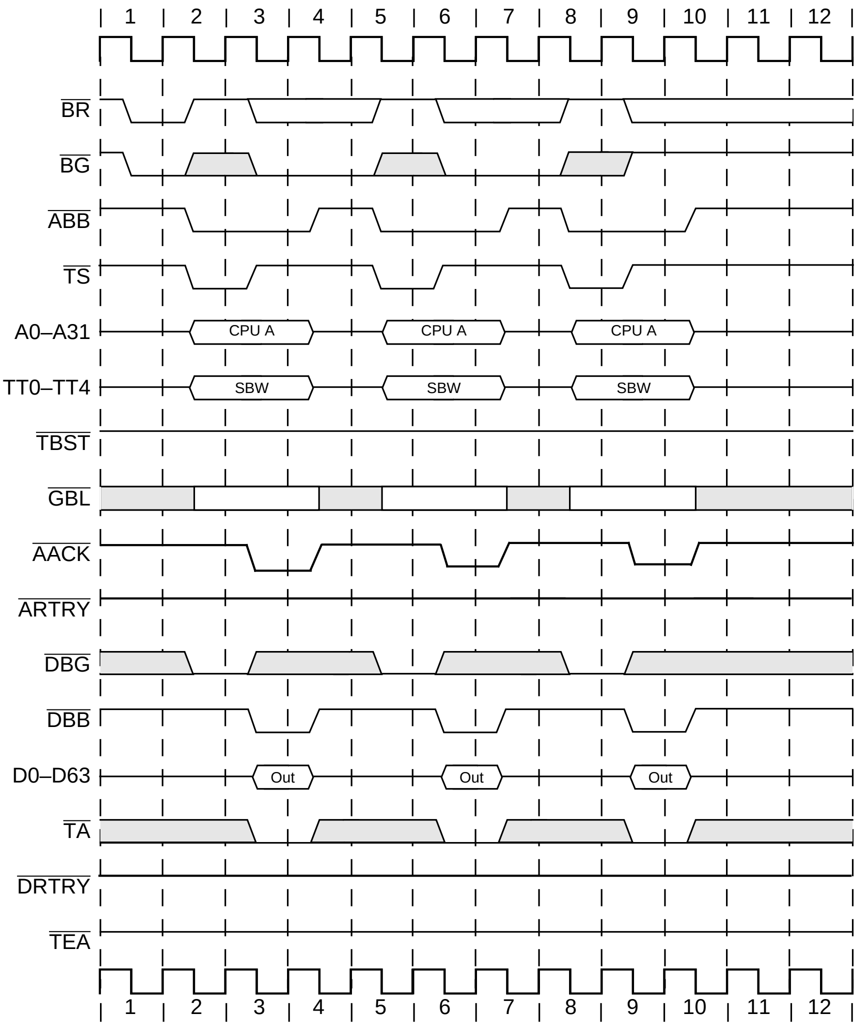


Figure 8-17. Fastest Single-Beat Writes

Figure 8-18 shows three ways to delay single-beat reads showing data-delay controls:

- The $\overline{\text{TA}}$ signal can remain negated to insert wait states in clock cycles 3 and 4.
- For the second access, $\overline{\text{DBG}}$ could have been asserted in clock cycle 6.
- In the third access, $\overline{\text{DRTRY}}$ is asserted in clock cycle 11 to flush the previous data.

Note that all bidirectional signals are three-stated between bus tenures.

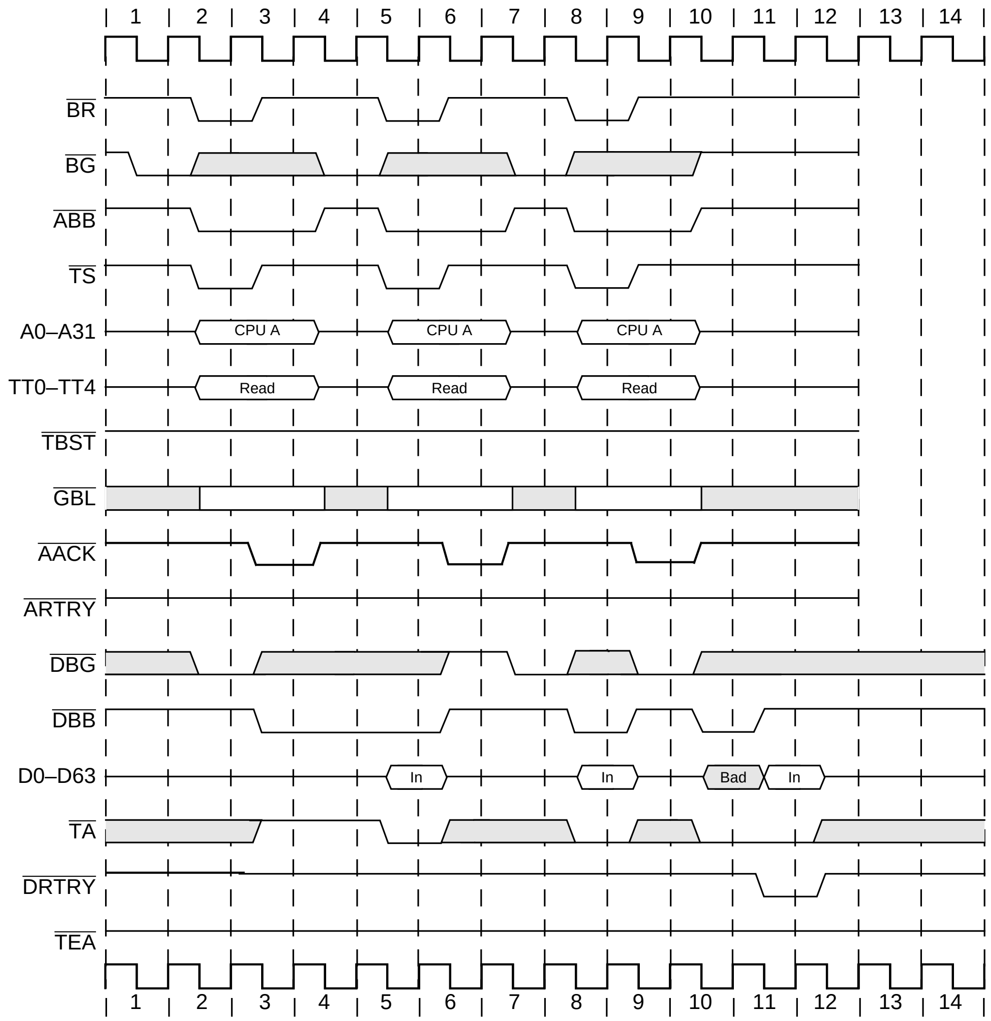


Figure 8-18. Single-Beat Reads Showing Data-Delay Controls

Figure 8-19 shows data-delay controls in a single-beat write operation. Note that all bidirectional signals are three-stated between bus tenures. Data transfers are delayed in the following ways:

- The $\overline{\text{TA}}$ signal is held negated to insert wait states in clocks 3 and 4.
- In clock 6, $\overline{\text{DBG}}$ is held negated, delaying the start of the data tenure.

The last access is not delayed ($\overline{\text{DRTRY}}$ is valid only for read operations).

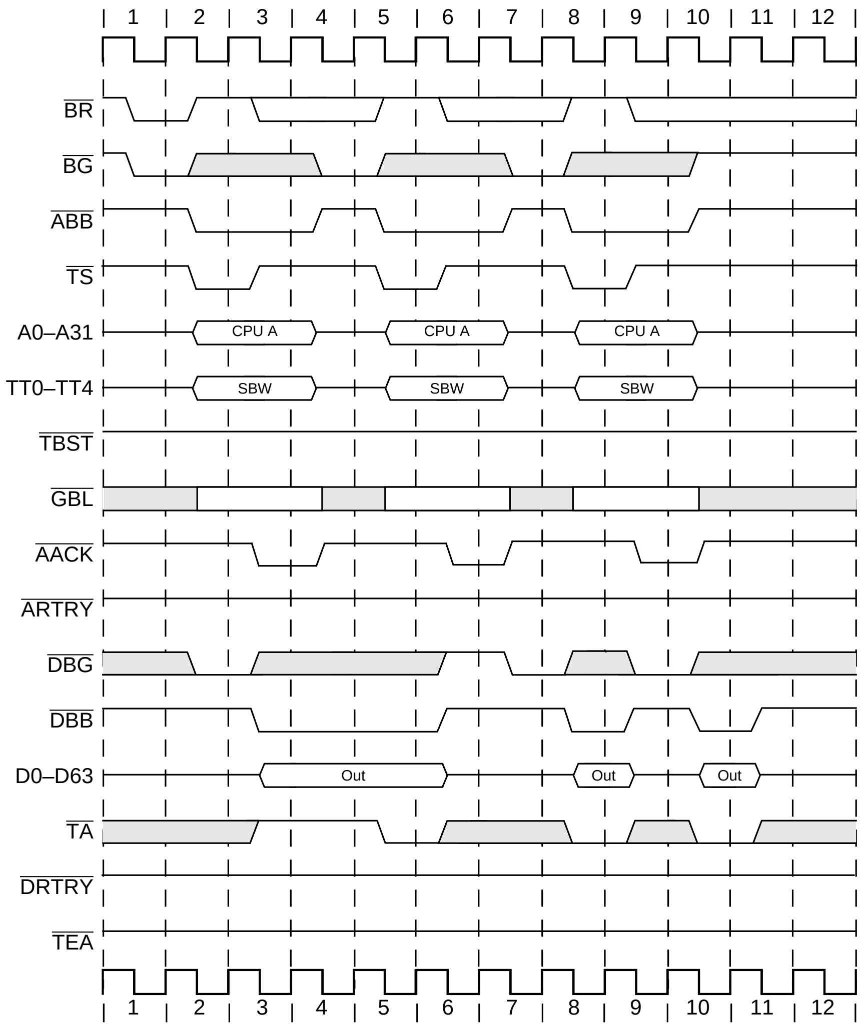


Figure 8-19. Single-Beat Writes Showing Data Delay Controls

Figure 8-20 shows the use of data-delay controls with burst transfers. Note that all bidirectional signals are three-stated between bus tenures. Note the following:

- The first data beat of bursted read data (clock 3) is the critical quad word.
- The write burst shows the use of $\overline{\text{TA}}$ signal negation to delay the third data beat.
- The final read burst shows the use of $\overline{\text{DRTRY}}$ on the third data beat.
- The address for the third transfer is delayed until the first transfer completes.

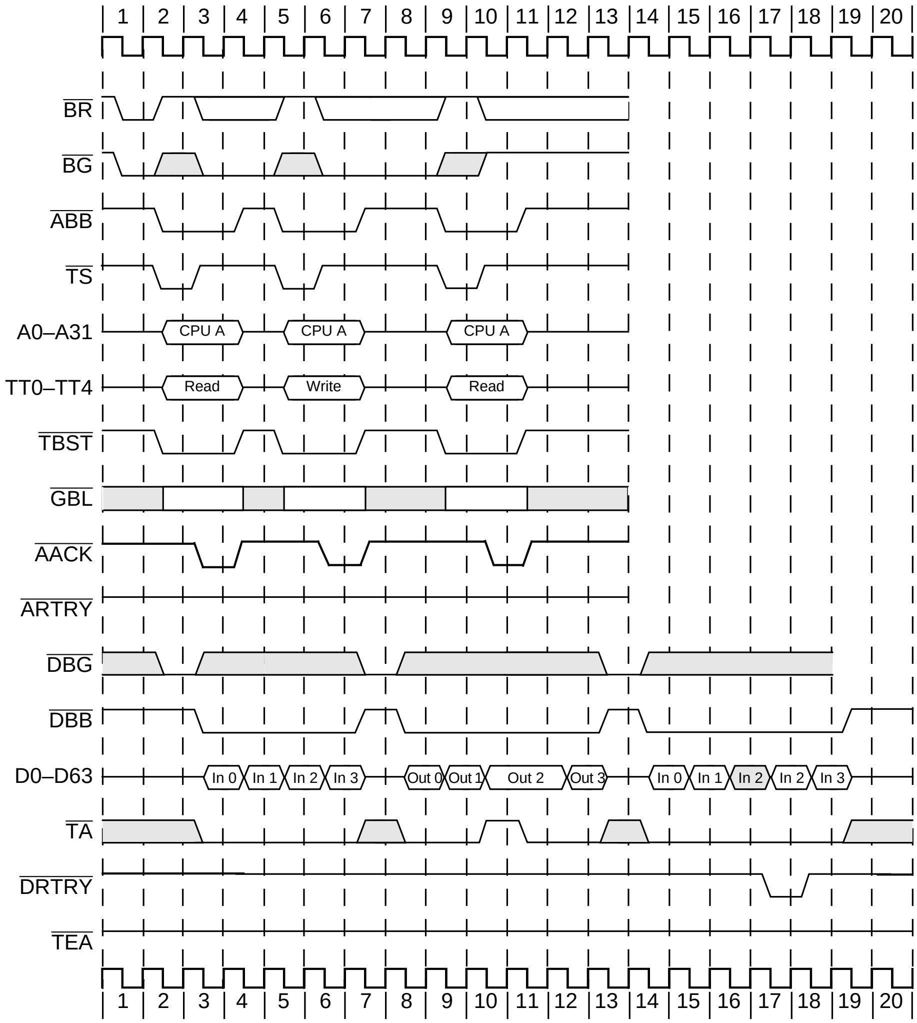


Figure 8-20. Burst Transfers with Data Delay Controls

Figure 8-21 shows the use of the $\overline{\text{TEA}}$ signal. Note that all bidirectional signals are three-stated between bus tenures. Note the following:

- The first data beat of the read burst (in clock 0) is the critical quad word.
- The $\overline{\text{TEA}}$ signal truncates the burst write transfer on the third data beat.
- The 604 eventually causes an exception to be taken on the $\overline{\text{TEA}}$ event.

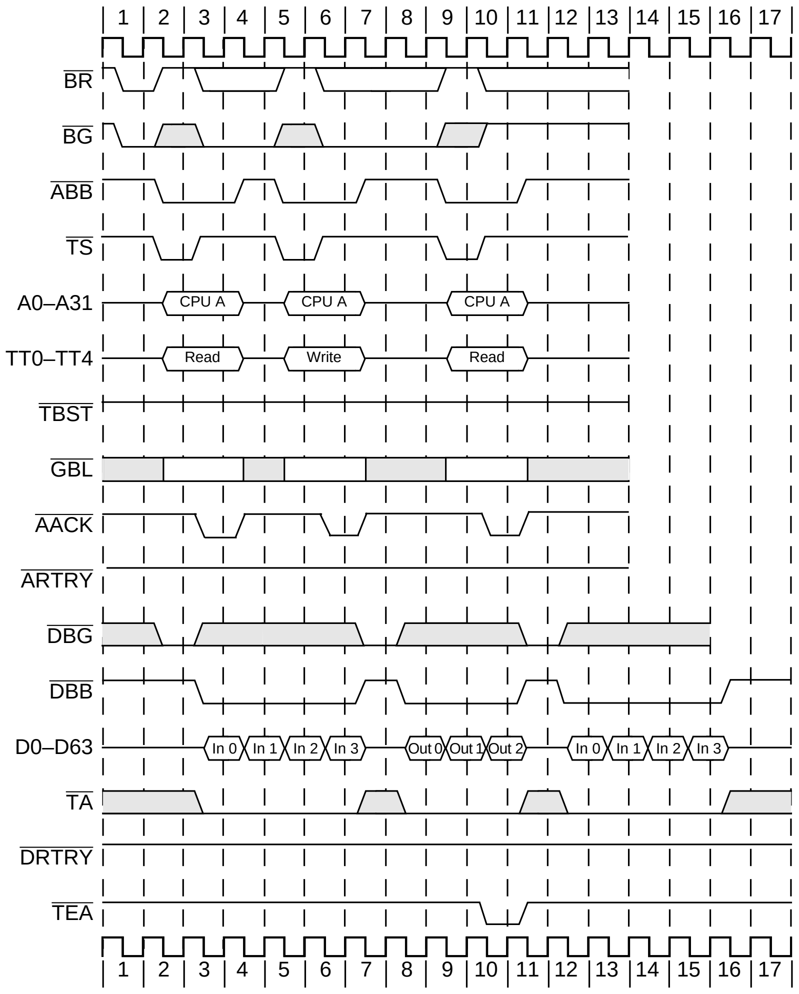


Figure 8-21. Use of Transfer Error Acknowledge ($\overline{\text{TEA}}$)

8.6 Direct-Store Operation

The 604 defines separate memory-mapped and I/O address spaces, or segments, distinguished by the corresponding segment register T bit in the address translation logic of the 604. If the T bit is cleared, the memory reference is a normal memory-mapped access and can use the virtual memory management hardware of the 604. If the T bit is set, the memory reference is a direct-store access.

The following points should be considered for direct-store accesses:

- The use of direct-store segment (referred to as direct-store segments in the architecture specification) accesses may have a significant impact on the performance of the 604. The provision of direct-store segment access capability by the 604 is to provide compatibility with earlier hardware I/O controllers and may not be provided in future derivatives of the 604 family.
- Direct-store accesses must be strongly ordered; for example, these accesses must run on the bus strictly in order with respect to the instruction stream.
- Direct-store accesses must provide synchronous error reporting. Chapter 3, "Cache and Bus Interface Unit Operation," describes architectural aspects of direct-store segments, as well as an overview of the segmented address space management of PowerPC processors.

The 604 has a single bus interface to support both memory accesses and direct-store segment accesses.

The direct-store protocol for the 604 allows for the transfer of 1 to 128 bytes of data between the 604 and the bus unit controller (BUC) for each single load or store request issued by the program. The block of data is transferred by the 604 as multiple single-beat bus transactions (individual address and data tenure for each transaction) until completion. The program waits for the sequence of bus transactions to be completed so that a final completion status (error or no error) can be reported precisely with respect to the program flow. The completion status is snooped by the 604 from a bus transaction run by the BUC.

The system recognizes the assertion of the $\overline{\text{TS}}$ signal as the start of a memory-mapped access. The assertion of $\overline{\text{XATS}}$ indicates a direct-store access. This allows memory-mapped devices to ignore direct-store transactions. If $\overline{\text{XATS}}$ is asserted, the access is to a direct-store space and the following extensions to the memory access protocol apply:

- A new set of bus operations are defined. The transfer type, transfer burst, and transfer size signals are redefined for direct-store operations; they convey the opcode for the I/O transaction (see Table 8-8).
- There are two beats of address for each direct-store transfer. The first beat (packet 0) provides basic address information such as the segment register and the sender tag and several control bits; the second beat (packet 1) provides additional addressing bits from the segment register and the logical address.

- The TT0–TT3, $\overline{\text{TBST}}$, and TSIZ0–TSIZ2 signals are remapped to form an 8-bit extended transfer code (XATC) which specifies a command and transfer size for the transaction. The XATC field is driven and snooped by the 604 during direct-store transactions.
- Only the data signals such as DH0–DH31 and DP0–DP3 are used. The lower half of the data bus and parity is ignored.
- The sender that initiated the transaction must wait for a reply from the receiver bus unit controller (BUC) before starting a new operation.
- The 604 does not burst direct-store transactions. All direct-store transactions generated by the 604 are single-beat transactions of four bytes or less (single data beat tenure per address tenure).

Direct-store transactions use separate arbitration for the split address and data buses and define address-only and single-beat transactions. The address-retry vehicle is identical, although there is no hardware coherency support for direct-store transactions. The $\overline{\text{ARTRY}}$ signal is useful, however, for pacing 604 transactions, effectively indicating to the 604 that the BUC is in a queue-full condition and cannot accept new data.

In addition to the extensions noted above, there are fundamental differences between memory-mapped and direct-store operations. For example, only half of the 64-bit data path is available for 604 direct-store transactions. This lowers the pin count for I/O interfaces but generally results in substantially less bandwidth than memory-mapped accesses. Additionally, load/store instructions that address direct-store segments cannot complete successfully without an error-free reply from the addressed BUC. Because normal direct-store accesses involve multiple I/O transactions (streaming), they are likely to be very long latency instructions; therefore, direct-store operations usually stall 604 instruction issue.

Figure 8-22 shows a direct-store tenure. Note that the I/O device response is an address-only bus transaction.

It should be noted that in the best case, the use of the 604 direct-store protocol degrades performance and requires the addressed controllers to implement 604 bus master capability to generate the reply transactions.

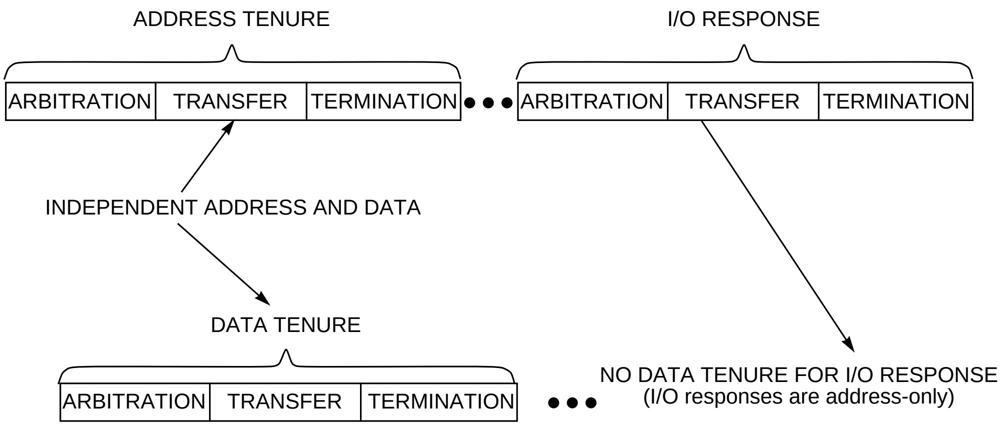


Figure 8-22. Direct-Store Tenures

8.6.1 Direct-Store Transactions

The 604 defines seven direct-store transaction operations, as shown in Table 8-8. These operations permit communication between the 604 and BUCs. A single 604 store or load instruction (that translates to a direct-store access) generates one or more direct-store operations (two or more direct-store operations for loads) from the 604 and one reply operation from the addressed BUC.

Table 8-8. Direct-Store Bus Operations

Operation	Address Only	Direction	XATC Encoding
Load start (request)	Yes	604 ⇒ IO	0100 0000
Load immediate	No	604 ⇒ IO	0101 0000
Load last	No	604 ⇒ IO	0111 0000
Store immediate	No	604 ⇒ IO	0001 0000
Store last	No	604 ⇒ IO	0011 0000
Load reply	Yes	IO ⇒ 604	1100 0000
Store reply	Yes	IO ⇒ 604	1000 0000

For the first beat of the address bus, the extended address transfer code (XATC), contains the I/O opcode as shown in Table 8-8; the opcode is formed by concatenating the transfer type, transfer burst, and transfer size signals defined as follows:

XATC = TT[0:3]||$\overline{\text{TBST}}$||TSIZ[0:2]

8.6.1.1 Store Operations

There are three operations defined for direct-store store operations from the 604 to the BUC, defined as follows:

1. Store immediate operations transfer up to 32 bits of data each from the 604 to the BUC.
2. Store last operations transfer up to 32 bits of data each from the 604 to the BUC.
3. Store reply from the BUC reveals the success/failure of that direct-store access to the 604.

A direct-store store access consists of one or more data transfer operations followed by the I/O store reply operation from the BUC. If the data can be transferred in one 32-bit data transaction, it is marked as a store last operation followed by the store reply operation; no store immediate operation is involved in the transfer, as shown in the following sequence:

STORE LAST (from 604)

•

•

STORE REPLY (from BUC)

However, if more data is involved in the direct-store access, there will be one or more store immediate operations. The BUC can detect when the last data is being transferred by looking for the store last opcode, as shown in the following sequence:

STORE IMMEDIATE(s)

•

•

STORE LAST

•

•

STORE REPLY

8.6.1.2 Load Operations

Direct-store load accesses are similar to store operations, except that the 604 latches data from the addressed BUC rather than supplying the data to the BUC. As with memory accesses, the 604 is the master on both load and store operations; the external system must provide the data bus grant to the 604 when the BUC is ready to supply the data to the 604.

The load request direct-store operation has no analogous store operation; it informs the addressed BUC of the total number of bytes of data that the BUC must provide to the 604 on the subsequent load immediate/load last operations. For direct-store load accesses, the simplest, 32-bit (or fewer) data transfer sequence is as follows:

LOAD REQUEST

•

•

LOAD LAST

•

•

LOAD REPLY(from BUC)

However, if more data is involved in the direct-store access, there will be one or more load immediate operations. The BUC can detect when the last data is being transferred by looking for the load last opcode, as seen in the following sequence:

LOAD REQUEST

•

•

LOAD IMM(s)

•

•

LOAD LAST

•

•

LOAD REPLY

Note that three of the seven defined operations are address-only transactions and do not use the data bus. However, unlike the memory transfer protocol, these transactions are not broadcast from one master to all snooping devices. The direct-store address-only transaction protocol strictly controls communication between the 604 and the BUC.

8.6.2 Direct-Store Transaction Protocol Details

As mentioned previously, there are two address-bus beats corresponding to two packets of information about the address. The two packets contain the sender and receiver tags, the address and extended address bits, and extra control and status bits. The two beats of the address bus (plus attributes) are shown at the top of Figure 8-23 as two packets. The first packet, packet 0, is then expanded to depict the XATC and address bus information in detail.

8.6.2.1 Packet 0

Figure 8-23 shows the organization of the first packet in a direct-store transaction.

The XATC contains the I/O opcode, as discussed earlier and as shown in Table 8-8. The address bus contains the following:

Key bit || segment register || sender tag

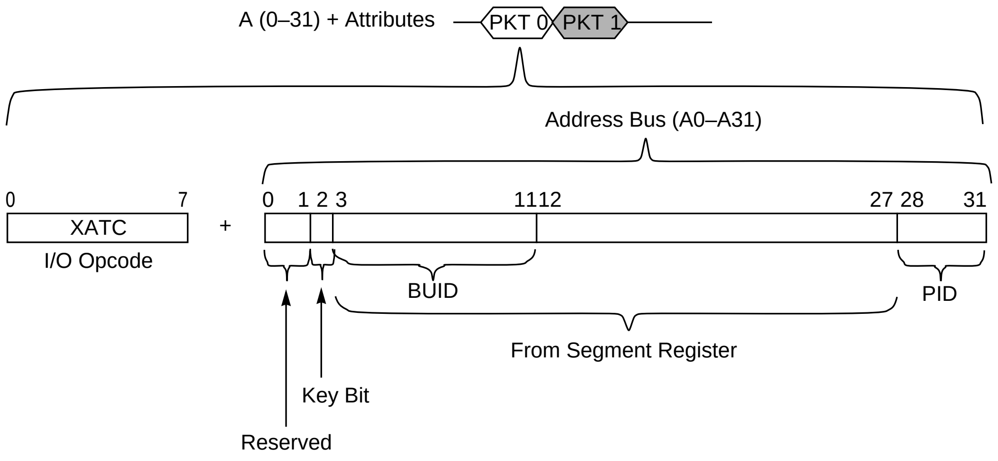


Figure 8-23. Direct-Store Operation—Packet 0

This information is organized as follows:

- Bits 0 and 1 of the address bus are reserved—the 604 always drives these bits to zero.
- Key bit—Bit 2 is the key bit from the segment register (either SR[Kp] or SR[Ks]). Kp indicates user-level access and Ks indicate supervisor-level access. The 604 multiplexes the correct key bit into this position according to the current operating context (user or supervisor). (Note that user- and supervisor-level refer to problem and privileged state, respectively, in the architecture specification.)
- Segment register—Address bits 3–27 correspond to bits 3–27 of the selected segment register. Note that address bits 3–11 form the 9-bit receiver tag. Software must initialize these bits in the segment register to the ID of the BUC to be addressed; they are referred to as the BUID (bus unit ID) bits.
- PID (sender tag)—Address bits 28–31 form the 4-bit sender tag. The 604 PID (processor ID) comes from bits 28-31 of the 604's processor ID register. The 4-bit PID tag allows a maximum of 16 processor IDs to be defined for a given system. If more bits are needed for a very large multiprocessor system, for example, it is envisioned that the second-level cache (or equivalent logic) can append a larger processor tag as needed. The BUC addressed by the receiver tag should latch the sender address required by the subsequent I/O reply operation.

8.6.2.2 Packet 1

The second address beat, packet 1, transfers byte counts and the physical address for the transaction, as shown in Figure 8-24.

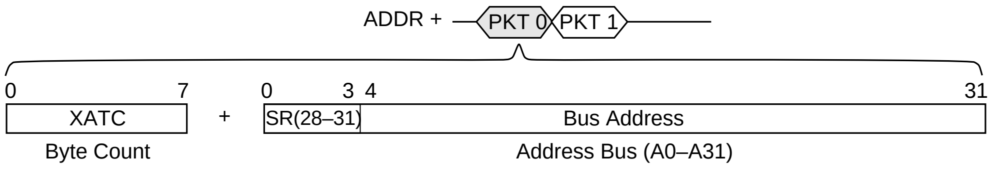


Figure 8-24. Direct-Store Operation—Packet 1

For packet 1, the XATC is defined as follows:

- Load request operations—XATC contains the total number of bytes to be transferred (128 bytes maximum for 604).
- Immediate/last (load or store) operations—XATC contains the current transfer byte count (1 to 4 bytes).

Address bits 0–31 contain the physical address of the transaction. The physical address is generated by concatenating segment register bits 28–31 with bits 4–31 of the effective address, as follows:

Segment register (bits 28–31) || effective address (bits 4–31)

While the 604 provides the address of the transaction to the BUC, the BUC must maintain a valid address pointer for the reply.

8.6.3 I/O Reply Operations

BUCs must respond to 604 direct-store transactions with an I/O reply operation, as shown in Figure 8-25. The purpose of this reply operation is to inform the 604 of the success or failure of the attempted direct-store access. This requires the system direct-store to have 604 bus mastership capability—a substantially more complex design task than bus slave implementations that use memory-mapped I/O access.

Reply operations from the BUC to the 604 are address-only transactions. As with packet 0 of the address bus on 604 direct-store operations, the XATC contains the opcode for the operation (see Table 8-8). Additionally, the I/O reply operation transfers the sender/receiver tags in the first beat.

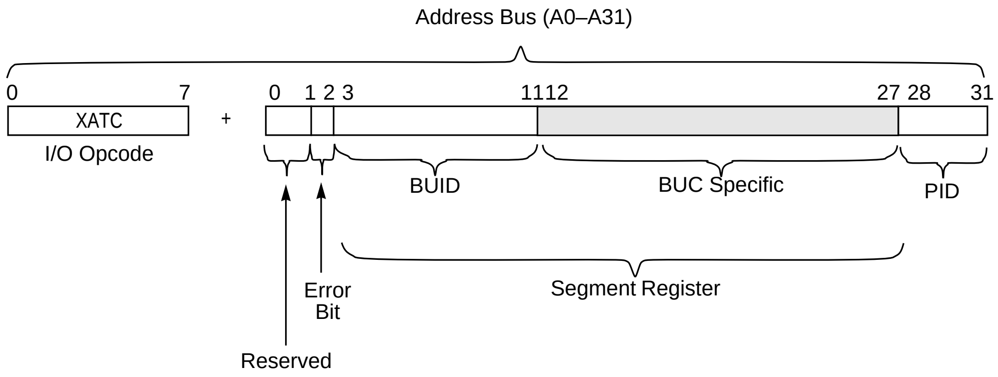


Figure 8-25. I/O Reply Operation

The address bits are described in Table 8-9.

Table 8-9. Address Bits for I/O Reply Operations

Address Bits	Description
0–1	Reserved. These bits should be cleared for compatibility with future PowerPC microprocessors.
2	Error bit. It is set if the BUC records an error in the access.
3–11	BUID. Sender tag of a reply operation. Corresponds with bits 3–11 of one of the 604 segment registers.
12–27	Address bits 12–27 are BUC-specific and are ignored by the 604.
28–31	PID (receiver tag). The 604 effectively snoops operations on the bus and, on reply operations, compares this field to bits 28–31 of the PID register to determine if it should recognize this I/O reply.

The second beat of the address bus is reserved; the XATC and address buses should be driven to zero to preserve compatibility with future protocol enhancements.

The following sequence occurs when the 604 detects an error bit set on an I/O reply operation:

1. The 604 completes the instruction that initiated the access.
2. If the instruction is a load, the data is forwarded onto the register file(s)/sequencer.
3. A direct-store error exception is generated, which transfers 604 control to the direct-store error exception handler to recover from the error.

If the error bit is not set, the 604 instruction that initiated the access completes and instruction execution resumes.

System designers should note the following:

- “Misplaced” reply operations (that match the processor tag and arrive unexpectedly) are ignored by the 604.
- External logic must assert $\overline{\text{AACK}}$ for the 604, even though it is the receiver of the reply operation. $\overline{\text{AACK}}$ is an input-only signal to the 604.
- The 604 monitors address parity when enabled by software and $\overline{\text{XATS}}$ and reply operations (load or store).

8.6.4 Direct-Store Operation Timing

The following timing diagrams show the sequence of events in a typical 604 direct-store load access (Figure 8-26) and a typical 604 direct-store store access (Figure 8-27). All arbitration signals except for $\overline{\text{ABB}}$ and $\overline{\text{DBB}}$ have been omitted for clarity, although they are still required as described earlier in this chapter. Note that, for either case, the number of immediate operations depends on the amount and the alignment of data to be transferred. If no more than 4 bytes are being transferred, and the data is double-word–aligned (that is, does not straddle an 8-byte address boundary), there will be no immediate operation as shown in the figures.

The 604 can transfer as many as 128 bytes of data in one load or store instruction (requiring more than 33 immediate operations in the case of misaligned operands).

In Figure 8-26, $\overline{\text{XATS}}$ is asserted with the same timing relationship as $\overline{\text{TS}}$ in a memory access. Notice, however, that the address bus (and XATC) transition on the next bus clock cycle. The first of the two beats on the address bus is valid for one bus clock cycle window only, and that window is defined by the assertion of $\overline{\text{XATS}}$. The second address bus beat, however, can be extended by delaying the assertion of $\overline{\text{AACK}}$ until the system has latched the address.

The load request and load reply operations, shown in Figure 8-26, are address-only transactions as denoted by the negated TT3 signal during their respective address tenures. Note that other types of bus operations can occur between the individual direct-store operations on the bus. The 604 involved in this transaction, however, does not initiate any other direct-store load or store operations once the first direct-store operation has begun address tenure; however, if the I/O operation is retried, other higher-priority operations can occur.

Notice that, in this example (zero wait states), 13 bus clock cycles are required to transfer no more than 8 bytes of data.

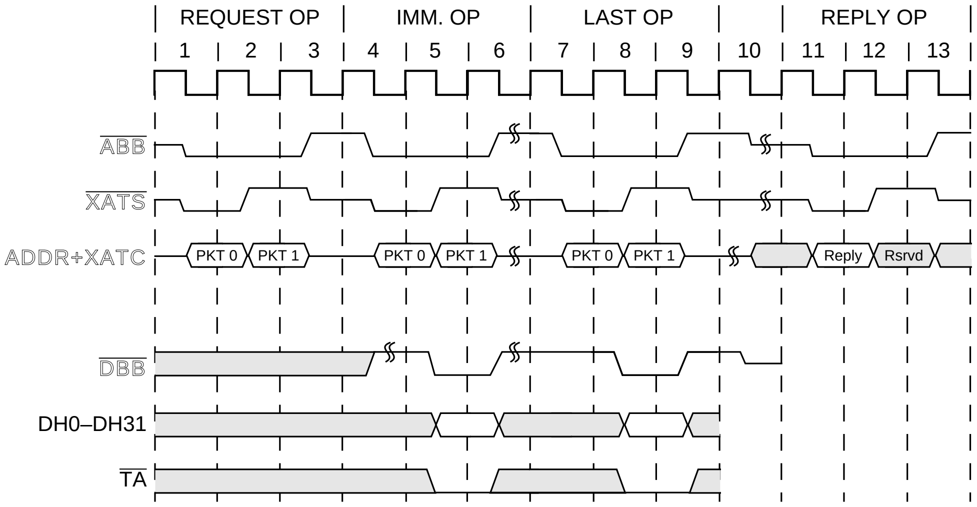


Figure 8-26. Direct-Store Interface Load Access Example

Figure 8-27 shows a direct-store store access, comprised of three direct-store operations. As with the example in Figure 8-26, notice that data is transferred only on the 32 bits of the DH bus. As opposed to Figure 8-26, there is no request operation since the 604 has the data ready for the BUC.

The assertion of the $\overline{\text{TEA}}$ signal during a direct-store operation indicates that an unrecoverable error has occurred. If the $\overline{\text{TEA}}$ signal is asserted during a direct-store operation, the $\overline{\text{TEA}}$ action will be delayed and following direct-store transactions will continue until all data transfers from direct store segment had been completed. The bus agent that asserts $\overline{\text{TEA}}$ is responsible to assert $\overline{\text{TEA}}$ for every direct-store transaction tenure including the last one. The direct-store reply, under this case, is not required and will be ignored by the processor. The processor will take a machine check exception after the last direct-store data tenure has been terminated by the assertion of $\overline{\text{TEA}}$, and not before.

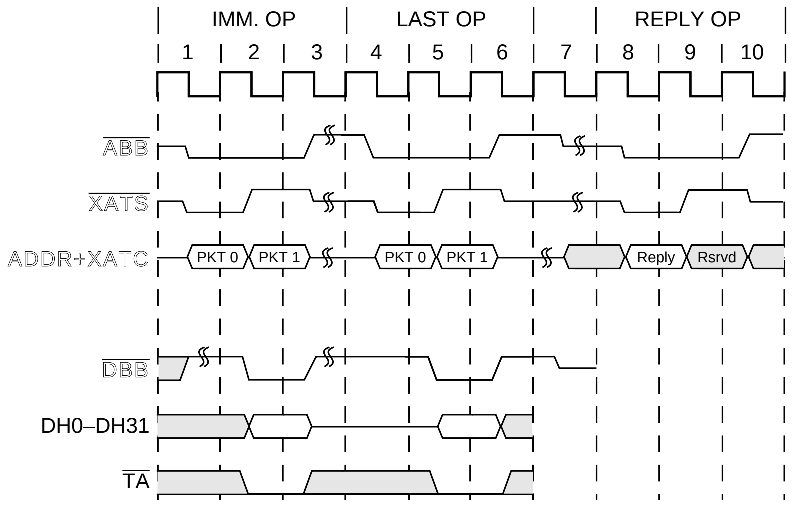


Figure 8-27. Direct-Store Interface Store Access Example

8.7 Optional Bus Configuration

The 604 supports an optional bus configuration that is selected by the assertion or negation of the $\overline{\text{DRTRY}}$ signal during the negation of the $\overline{\text{HRESET}}$ signal. The operation and selection of the optional bus configuration is described in the following section.

8.7.1 Fast-L2/Data Streaming Mode

The 604 supports an optional mode (described as the fast-L2/data streaming mode), that disables the use of the data retry function provided through the $\overline{\text{DRTRY}}$ signal. Although this bus interface mode implies its suitability for use in interfacing to a second-level cache, the fast-L2/data streaming mode allows the forwarding of data during load operations to the internal CPU one bus cycle sooner than in the normal bus protocol. The PowerPC bus protocol specifies that, during load operations, the memory system normally has the capability to cancel data that was read by the master on the bus cycle after $\overline{\text{TA}}$ was asserted. In the 604 implementation, this late cancellation protocol requires the 604 to hold any loaded data at the bus interface for one additional bus clock to verify that the data is valid before forwarding it to the internal CPU. The use of the optional fast-L2/data streaming mode eliminates the one-cycle stall during all load operations, and allows for the forwarding of data to the internal CPU immediately when $\overline{\text{TA}}$ is recognized, thereby increasing maximum read bandwidth.

When the 604 is following normal bus protocol, data may be cancelled the bus cycle after $\overline{\text{TA}}$ by either of two means—late cancellation by $\overline{\text{DRTRY}}$, or late cancellation by $\overline{\text{ARTRY}}$. When the fast-L2/data streaming mode is selected, both cancellation cases must be disallowed in the system design for the bus protocol.

When the fast-L2/data streaming mode is selected for the 604, the system must ensure that $\overline{\text{DRTRY}}$ will not be asserted to the 604. If it is asserted, it may cause improper operation of the bus interface. The system must also ensure that an assertion of $\overline{\text{ARTRY}}$ by a snooping device must occur before or coincident with the first assertion of $\overline{\text{TA}}$ to the 604, but not on the cycle after the first assertion of $\overline{\text{TA}}$.

The 604 selects the desired $\overline{\text{DRTRY}}$ mode at startup by sampling the state of the $\overline{\text{DRTRY}}$ signal at the negation of the $\overline{\text{HRESET}}$ signal. If the $\overline{\text{DRTRY}}$ signal is negated at the negation of $\overline{\text{HRESET}}$, normal operation is selected. If the $\overline{\text{DRTRY}}$ signal is asserted at the negation of $\overline{\text{HRESET}}$, fast-L2/data streaming mode is selected. To select the fast-L2/data streaming mode, the system designer may connect the $\overline{\text{DRTRY}}$ signal to the $\overline{\text{HRESET}}$ signal. This asserts $\overline{\text{DRTRY}}$ during startup for fast-L2/data streaming mode selection, and holds the $\overline{\text{DRTRY}}$ signal negated during operation.

When the 604 is in fast-L2/data streaming mode, the bus protocol is modified to disable the ability to cancel data that was read by the master on the bus cycle after $\overline{\text{TA}}$ was asserted. Also, $\overline{\text{DBB}}$ is an output-only signal, and is not a term in generating a qualified data bus grant. When in fast-L2/data streaming mode, the system is not allowed to assert $\overline{\text{DBG}}$ earlier than one cycle before the data tenure is to commence, to park $\overline{\text{DBG}}$, or to assert $\overline{\text{DBG}}$ for multiple consecutive cycles. In all other respects, the bus protocol for the 604 is identical to that for the basic and extended transfer bus protocols described in this chapter.

8.7.1.1 Fast-L2/Data Streaming Mode Design Considerations

It is recommended that use of fast-L2/data streaming mode be accompanied by two other system design practices.

The first recommendation is not to use the $\overline{\text{ABB}}$ signal. If the system is designed so that an address tenure is defined by $\overline{\text{TS}}$ and $\overline{\text{AACK}}$ assertion, (which the 604 is designed to support), the $\overline{\text{ABB}}$ signal is unnecessary, and should be pulled high at the 604. Because the $\overline{\text{ABB}}$ signal has an inherently short “restore high” time, it is desirable that the $\overline{\text{ABB}}$ signal not be used in systems that try to achieve a short cycle time.

The second recommendation is not to use the $\overline{\text{DBB}}$ signal. This signal is restored high in the same way as $\overline{\text{ABB}}$, and therefore has the same problems in a system with short cycle time. To avoid the use of the $\overline{\text{DBB}}$ signal, the system arbiter must assert t $\overline{\text{DBG}}$ for a single cycle, one cycle before the 604 is supposed to begin its data tenure. The $\overline{\text{DBB}}$ signal should be pulled high. The additional system cost of operating in this manner is that it must count the number of data transfers, and assert $\overline{\text{DBG}}$ only on the last cycle in a data tenure.

8.7.1.2 Data Streaming in the Fast-L2/Data Streaming Mode

Data streaming is the ability to commence a data tenure after a previous data tenure with no dead cycles between. The 604 only supports data streaming for consecutive burst read data transfers. This does include support for data streaming consecutive burst read data transfers between two separate masters. For instance, in a multi-604 system, data streaming is allowed on consecutive burst read data transfers from different 604s.

To cause data streaming to take place, the system asserts $\overline{\text{DBG}}$ during the last data transfer of the first data tenure as shown in Figure 8-28. To fully realize the performance gain of data streaming, the system should be prepared to, but is not required to, supply an uninterrupted sequence of $\overline{\text{TA}}$ assertions.

Figure 8-28 shows the operation of the $\overline{\text{DBG}}$ signal when data streaming operations are taking place on the data bus

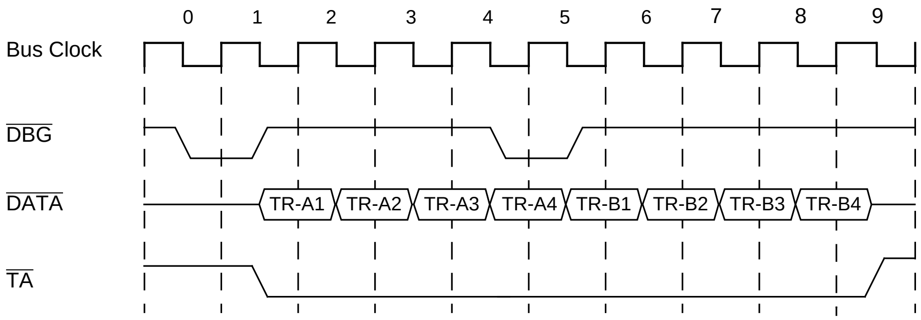


Figure 8-28. Data Transfer in Fast-L2/Data Streaming Mode

8.7.1.3 Data Valid Window in the Fast-L2/Data Streaming Mode

Standard bus mode operations allow data to be transferred no earlier than the cycle before the ARTRY window that the system defines. In some cases, an asserted $\overline{\text{ARTRY}}$ signal invalidates the data that was transferred the previous cycle, in the same way $\overline{\text{DRTRY}}$ cancels data from the previous cycle.

In fast-L2/data streaming mode, the data buffering that allows late cancellation of a data transfer does not exist, so late cancellation with $\overline{\text{ARTRY}}$ is also impossible. Therefore, the earliest that data can be transferred in fast-L2/data streaming mode is the first cycle of the ARTRY window, not the cycle before that.

8.8 Interrupt, Checkstop, and Reset Signals

This section describes external interrupts, checkstop operations, and hard and soft reset inputs.

8.8.1 External Interrupts

The external interrupt input signals ($\overline{\text{INT}}$, $\overline{\text{SMI}}$ and $\overline{\text{MCP}}$) to the 604 eventually force the processor to take the external interrupt vector, the system management interrupt vector, or the machine check interrupt if enabled by the MSR[EE] bit (and the HID0[EMCP] bit in the case of a machine check interrupt).

8.8.2 Checkstops

The 604 has two checkstop input signals—$\overline{\text{CKSTP\_IN}}$ and $\overline{\text{MCP}}$ (when MSR[ME] is cleared, and HID0[EMCP] is set), and a checkstop output ($\overline{\text{CKSTP\_OUT}}$). If $\overline{\text{CKSTP\_IN}}$ or $\overline{\text{MCP}}$ is asserted, the 604 halts operations by gating off all internal clocks. The 604 asserts $\overline{\text{CKSTP\_OUT}}$ if $\overline{\text{CKSTP\_IN}}$ is asserted.

If $\overline{\text{CKSTP\_OUT}}$ is asserted by the 604, it has entered the checkstop state, and processing has halted internally. The $\overline{\text{CKSTP\_OUT}}$ signal can be asserted for various reasons including receiving a $\overline{\text{TEA}}$ signal and detection of external parity errors. For more information about checkstop state, see Section 4.5.2.2, “Checkstop State (MSR[ME] = 0).”

8.8.3 Reset Inputs

The 604 has two reset inputs, described as follows:

- $\overline{\text{HRESET}}$ (hard reset)—The $\overline{\text{HRESET}}$ signal is used for power-on reset sequences, or for situations in which the 604 must go through the entire cold-start sequence of internal hardware initializations.
- $\overline{\text{SRESET}}$ (soft reset)—The soft reset input provides warm reset capability. This input can be used to avoid forcing the 604 to complete the cold start sequence.

When either reset input is negated, the processor attempts to fetch code from the system reset exception vector. The vector is located at offset 0x00100 from the exception prefix (all zeros or ones, depending on the setting of the exception prefix bit in the machine state register (MSR[IP]). The IP bit is set for $\overline{\text{HRESET}}$.

8.8.4 PowerPC 604 Microprocessor Configuration during $\overline{\text{HRESET}}$

The 604’s bus interface can be configured into one of two modes during a hard reset, as described in Table 8-10.

Table 8-10. PowerPC 604 Microprocessor Mode Configuration during $\overline{\text{HRESET}}$

604 Mode	Input Signal Used	Timing Requirements	Notes
Normal bus mode	$\overline{\text{DRTRY}}$	Must be negated throughout the duration of the $\overline{\text{HRESET}}$ assertion. After $\overline{\text{HRESET}}$ negation, $\overline{\text{DRTRY}}$ can be used normally.	
Fast-L2/data streaming mode	$\overline{\text{DRTRY}}$	Must be asserted and negated coincidentally with $\overline{\text{HRESET}}$ and remain negated during normal operation.	Can be implemented by tying $\overline{\text{DRTRY}}$ to $\overline{\text{HRESET}}$.

8.9 Processor State Signals

This section describes the 604's support for atomic update and memory through the use of the **lwarx**/**stwcx.** opcode pair.

8.9.1 Support for the lwarx/stwcx. Instruction Pair

The Load Word and Reserve Indexed (**lwarx**) and the Store Word Conditional Indexed (**stwcx.**) instructions provide a means for atomic memory updating. Memory can be updated atomically by setting a reservation on the load and checking that the reservation is still valid before the store is performed. In the 604, the reservations are made on behalf of aligned, 32-byte sections of the memory address space.

The reservation ($\overline{\text{RSRV}}$) output signal is driven synchronously with the bus clock and reflects the status of the reservation coherency bit in the reservation address register (see Chapter 3, "Cache and Bus Interface Unit Operation," for more information). See Section 7.2.10.2, "Reservation (RSRV)—Output," for information about timing.

8.10 IEEE 1149.1-Compliant Interface

The 604 boundary-scan interface is a fully-compliant implementation of the IEEE 1149.1 standard. This section describes the 604 IEEE 1149.1(JTAG) interface.

8.10.1 IEEE 1149.1 Interface Description

The 604 has five dedicated JTAG signals which are described in Table 8-11. The TDI and TDO scan ports are used to scan instructions as well as data into the various scan registers for JTAG operations. The scan operation is controlled by the test access port (TAP) controller which in turn is controlled by the TMS input sequence. The scan data is latched in at the rising edge of TCK.

Table 8-11. IEEE Interface Pin Descriptions

Signal Name	Input/Output	Weak Pullup Provided	IEEE 1149.1 Function
TDI	Input	Yes	Serial scan input pin
TDO	Output	No	Serial scan output pin
TMS	Input	Yes	TAP controller mode pin
TCK	Input	Yes	Scan clock
$\overline{\text{TRST}}$	Input	Yes	TAP controller reset

$\overline{\text{TRST}}$ is a JTAG optional signal which is used to reset the TAP controller asynchronously. The $\overline{\text{TRST}}$ signal assures that the JTAG logic does not interfere with the normal operation of the chip, and should be held asserted during normal operation. The remaining JTAG signals are provided with internal pullup resistors, and may be left unconnected.

Boundary scan description language (BSDL) files for the 604 and other PowerPC microprocessors are available in the RISC support area of the Motorola Freeware Data Services bulletin board system. The bulletin board system, located in Austin, Texas, can be reached at (512) 891-3733; the connecting terminal or terminal emulator should be configured with 8-bit data, no parity, and one start and one stop bit. Asynchronous transmission rates to 14.4K bits per second are supported.

8.11 Using Data Bus Write Only

The 604 supports split-transaction pipelined transactions. It supports a limited out-of-order capability for its own pipelined transactions through the data bus write only ($\overline{\text{DBWO}}$) signal. When recognized on the clock of a qualified $\overline{\text{DBG}}$, the assertion of $\overline{\text{DBWO}}$ directs the 604 to perform the next pending data write tenure (if any), even if a pending read tenure would have normally been performed because of address pipelining. The $\overline{\text{DBWO}}$ does not change the order of write tenures with respect to other write tenures from the same 604. It only allows that a write tenure be performed ahead of a pending read tenure from the same 604.

In general, an address tenure on the bus is followed strictly in order by its associated data tenure. Transactions pipelined by the 604 complete strictly in order. However, the 604 can run bus transactions out of order only when the external system allows the 604 to perform a cache line snoop push out operation (or other write transaction, if pending in the 604 write queues) between the address and data tenures of a read operation through the use of $\overline{\text{DBWO}}$. This effectively envelopes the write operation within the read operation. Figure 8-29 shows how the $\overline{\text{DBWO}}$ signal is used to perform an enveloped write transaction.

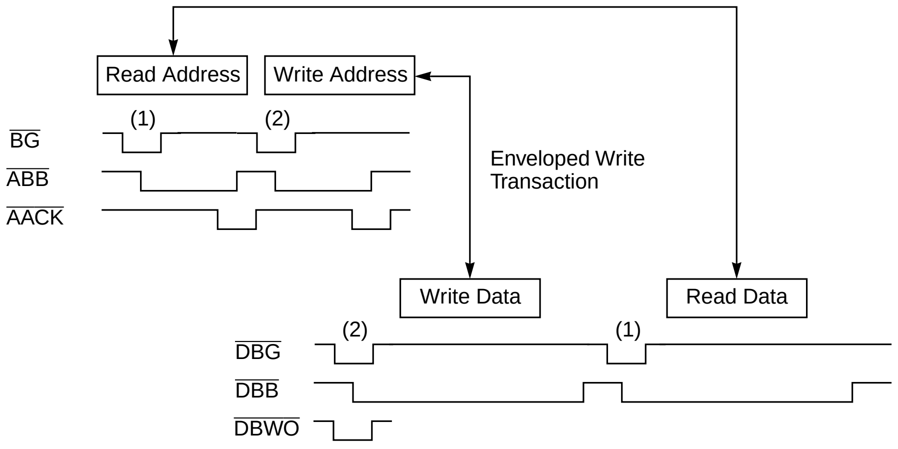


Figure 8-29. Data Bus Write Only Transaction

Note that although the 604 can pipeline any write transaction behind the read transaction, special care should be used when using the enveloped write feature. It is envisioned that most system implementations will not need this capability; for these applications $\overline{\text{DBWO}}$ should remain negated. In systems where this capability is needed, $\overline{\text{DBWO}}$ should be asserted under the following scenario:

1. The 604 initiates a read transaction (either single-beat or burst) by completing the read address tenure with no address retry.
2. Then, the 604 initiates a write transaction by completing the write address tenure, with no address retry.
3. At this point, if $\overline{\text{DBWO}}$ is asserted with a qualified data bus grant to the 604, the 604 asserts $\overline{\text{DBB}}$ and drives the write data onto the data bus, out of order with respect to the address pipeline. The write transaction concludes with the 604 negating $\overline{\text{DBB}}$.
4. The next qualified data bus grant signals the 604 to complete the outstanding read transaction by latching the data on the bus. This assertion of $\overline{\text{DBG}}$ should not be accompanied by an asserted $\overline{\text{DBWO}}$.

Any number of bus transactions by other bus masters can be attempted between any of these steps.

Note the following regarding $\overline{\text{DBWO}}$:

- The $\overline{\text{DBWO}}$ signal can be asserted if no data bus read is pending, but it has no effect on write ordering.
- The ordering and presence of data bus writes is determined by the writes in the write queues at the time $\overline{\text{BG}}$ is asserted for the write address (not $\overline{\text{DBG}}$). A cache-line snoop push-out operation has the highest priority, and takes precedence over other queued write operations.
- Because more than one write may be in the write queue when $\overline{\text{DBG}}$ is asserted for the write address, more than one data bus write may be enveloped by a pending data bus read.

The arbiter must monitor bus operations and coordinate the various masters and slaves with respect to the use of the data bus when $\overline{\text{DBWO}}$ is used. Individual $\overline{\text{DBG}}$ signals associated with each bus device should allow the arbiter to synchronize both pipelined and split-transaction bus organizations. Individual $\overline{\text{DBG}}$ and $\overline{\text{DBWO}}$ signals provide a primitive form of source-level tagging for the granting of the data bus.

Note that use of the $\overline{\text{DBWO}}$ signal allows some operation-level tagging with respect to the 604 and the use of the data bus.

Chapter 9
Performance Monitor

The PowerPC 604 microprocessor provides a performance monitor facility to monitor and count predefined events such as processor clocks, misses in either the instruction cache or the data cache, instructions dispatched to a particular execution unit, mispredicted branches, and other occurrences. The count of such events (which may be an approximation) can be used to trigger the performance monitor exception. The performance monitor facility is not defined by the PowerPC architecture.

The performance monitor can be used for the following:

- To increase system performance with efficient software, especially in a multiprocessing system. Memory hierarchy behavior must be monitored and studied in order to develop algorithms that schedule tasks (and perhaps partition them) and that structure and distribute data optimally.
- To improve processor architecture, the detailed behavior of the 604's structure must be known and understood in many software environments. Some environments may not easily be characterized by a benchmark or trace.
- To help system developers bring up and debug their systems.

The performance monitor uses the following 604-specific special-purpose registers (SPRs):

- Performance monitor counters 1 and 2 (PMC1 and PMC2)—two 32-bit counters used to store the number of times a certain event has been detected.
- The monitor mode control register (MMCR0), which establishes the function of the counters.
- Sampled instruction address and sampled data address registers (SIA and SDA). Depending on how the performance monitor is configured, these registers point to the data or instruction that caused a threshold-related performance monitor interrupt.

The 604 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 interrupts, the performance monitor interrupt follows the normal PowerPC exception model with a defined exception vector offset (0x00F00). The priority of the performance monitor interrupt is below the external interrupt and above the decrementer interrupt. The contents of the SIA and SDA are described in Section 9.1.1.2.1, “Sampled Instruction Address Register (SIA),” and Section 9.1.1.2.2, “Sampled Data Address Register (SDA),” respectively. The performance monitor counter registers are described in Section 9.1.1.1, “Performance Monitor Counter Registers (PMC1 and PMC2).”

9.1 Performance Monitor Interrupt

The 604 performance monitor is a software-accessible mechanism that provides detailed information concerning the dispatch, execution, completion, and memory access of PowerPC instructions. A performance monitor interrupt (PMI) can be triggered by a negative counter (most significant bit set to one) condition. If the interrupt signal condition occurs while MSR[EE] is cleared, the interrupt is delayed until the MSR[EE] bit is set. A PMI may also occur when certain bits in the time base register change from 0 to 1; this provides a way to generate interrupts based on a time reference.

Depending on the type of event that causes the PMI condition to be signaled, the performance monitor responds in one of two ways:

- When a threshold event causes a PMI to be signaled, the exact addresses of the instruction and data that caused the counter to become negative are saved in the sampled instruction address (SIA) register and the sampled data address (SDA) register, respectively. For more information, see Section 9.1.2.2, “Threshold Events.”
- For all other programmable events that cause a PMI, the address of the last completed instruction during that cycle is saved in the SIA, which allows the user to determine the part of the code being executed when a PMI was signaled. Likewise, the effective address of an operand being used is saved in the SDA. Typically, the operands in the SDA and SIA are unrelated. For more information, see Section 9.1.2.3, “Nonthreshold Events.”

When the performance monitor interrupt is signaled, the hardware clears MMCR0[ENINT] and prevents the changing of the values in the SIA and SDA until ENINT is set by software. The MMCR0 is described in the Section 9.1.1.3, “Monitor Mode Control Register 0 (MMCR0).”

The following section describes the SPRs used with the performance monitor.

9.1.1 Special-Purpose Registers Used by Performance Monitor

The performance monitor incorporates the SPRs listed in Table 9-1. The SIA register is located in the sequencer unit and the SDA register is located in the LSU. All of these supervisor-level registers are accessed through **mtspr** and **mfspr** instructions. The following table shows more information about all performance monitor SPRs.

Table 9-1. Performance Monitor SPRs

SPR Number	spr[5–9] || spr[0–4]	Register Name	Access Level
952	0b11101 11000	MMCR0	Supervisor
953	0b11101 11001	PMC1	Supervisor
954	0b11101 11010	PMC2	Supervisor
955	0b11101 11011	SIA	Supervisor
959	0b11101 11111	SDA	Supervisor

9.1.1.1 Performance Monitor Counter Registers (PMC1 and PMC2)

PMC1 and PMC2 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; they reach the value 0x8000_0000, that is, all zeros with the most significant bit, or sign bit, set. However, an interrupt is not signaled unless both MMCR0[INTCONTROL] and MMCR0[ENINT] 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[DISCOUNT] forces the counters to stop counting when a counter interrupt occurs.

PMC1 and PMC2 are SPRs 953 and 954, 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 nonnegative values. If software sets a negative value, an erroneous interrupt may occur. For example, if both MMCR0[INTCONTROL] and MMCR0[ENINT] 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 9-2 and Table 9-3 list the selectable events for the PMC1 and PMC2 registers, respectively, with their appropriate MMCR0 encodings.

9.1.1.1.1 PMC1 Selectable Events

The events counted by PMC1 can be divided into two groups.

- Events that can occur only once per cycle. These are the most common.
- Events can have as many as four occurrences per cycle, such as instructions dispatched per clock.

Events selectable for counting by PMC1 are listed, along with their MMCR0[19–25] encodings, in Table 9-2.

Table 9-2. PMC1 Events—MMCR0 [19–25] Select Encodings

Encoding	Description
000 0000	Nothing. Register counter holds current value.
000 0001	Processor cycles are counted
000 0010	Count the number of instructions completed per cycle. Legal values are 000, 001, 010, 011, 100.
000 0011	RTCSELECT bit transition. (0 = 47, 1 = 51, 2 = 55, 3 = 63) Bits from the time-base lower register (TBL).
000 0100	Number of instructions dispatched. From zero to four instructions per cycle
000 0101	Instruction cache misses (speculative (Instruction cache line-fill))
000 0110	**dtlb** misses (not speculative)
000 0111	Branch incorrectly predicted
000 1000	Number of reservations requested
000 1001	Number of load data cache misses that exceeded the threshold value with lateral L2 cache intervention. For more information on L2 cache intervention, see Section 7.2.10.3, “L2 Intervention (L2_INT)—Input.”
000 1010	Number of store data cache misses that exceeded the threshold value with lateral L2 cache intervention
000 1011	Number of **mtspr** instructions dispatched
000 1100	Number of **sync** instructions completed
000 1101	Number of **eieio** instructions completed
000 1110	Number of integer instructions completed every cycle (no loads or stores)
000 1111	Number of floating-point instructions completed every cycle (no loads or stores)
001 0000	LSU produced result without an exception condition
001 0001	SCIU1 unit produced result. (add, subtract, compare, rotate, shift, or logical instructions)
001 0010	FPU produced result
001 0011	Number of instructions dispatched to the LSU
001 0100	Number of instructions dispatched to the SCIU1 unit
001 0101	Number of instructions dispatched to the floating-point unit
001 0110	Snoop requests received. Valid snoops from outside the 604. Does not know if it is a hit or miss.
001 0111	Number of marked load data cache misses that exceeded the threshold value without lateral L2 intervention.
001 1000	Number of marked store data cache misses that exceeded the threshold value without lateral L2 intervention

9.1.1.1.2 PMC2 Selectable Events

The events counted by PMC2 follow the same groupings explained in the previous section. The differences between PMC1 and PMC2 event selection are as follows:

- Different bits of the MMCR0 register are decoded in order to select events.
- PMC2 has fewer events.

Events selectable for counting by PMC2 are listed, along with their MMCR0[26–31] encodings, are listed in Table 9-3.

Table 9-3. PMC2 Events—MMCR0 [26–31] Select Encoding

Encoding	Description
00 0000	Nothing. Register counter holds current value
00 0001	Processor cycles
00 0010	Number of instructions completed. Legal values are 000, 001, 010, 011, and 100.
00 0011	RTCSELECT bit transition. 0 = 47, 1 = 51, 2 = 55, 3 = 63 bits from the time base lower register.
00 0100	Number of instructions dispatched
00 0101	Number of cycles a load miss takes
00 0110	Data cache misses (data cache line fill)
00 0111	Number of **itlb** misses
00 1000	Number of branches completed. Indicates the number of branch instructions completed every cycle. 00 None 01 Illegal value 10 One 11 Two
00 1001	Number of reservations successfully obtained
00 1010	Number of **mfspr** instructions dispatched (speculative)
00 1011	Number of **icbi** instructions. The **icbi** instruction may not hit in the cache.
00 1100	Number of pipeline-flushing operations (**sc**, **isync**, **mtspr[xer]**, floating-point operations with divide by 0 or invalid operand when the 604 is in precise mode, branch when MSR[BE] is set, **lswx** with XER = 0 and SO set).
00 1101	Branch unit produced result (branch or CR-logical instruction finished)
00 1110	SCIU0 unit produced result (add, subtract, compare, rotate, shift, or logical instruction)
00 1111	MCIU unit produced result (multiply/divide or SPR instruction)
01 0000	Number of instructions dispatched to the branch unit
01 0001	Number of instructions dispatched to the SCIU0 unit
01 0010	Number of loads completed. From 0 to 4 instructions per cycle. Indicates the number of load instructions being completed every cycle. These include all cache operations, **tlbie**, **tlbsync**, **sync**, **eieio**, and **icbi**.
01 0011	Number of instructions dispatched to the MCIU
01 0100	Number of snoop hits occurred

9.1.1.2 SIA and SDA Registers

The two address registers contain the addresses of the data or the instruction that caused a threshold-related performance monitor interrupt. For more information on threshold-related interrupts, see Section 9.1.2.2, “Threshold Events.”

9.1.1.2.1 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.

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 SDA contains an effective address that is not guaranteed to match the instruction in the SIA. The SIA and SDA are supervisor-level SPRs.

The SIA can be read by using the **mfspr** instruction and written to by using the **mtspr** instruction (SPR 955).

9.1.1.2.2 Sampled Data Address Register (SDA)

The SDA contains the effective address of an operand of an instruction executing at or around the time that the processor signals the performance monitor interrupt condition. In this case the SDA is not meant to have any connection with the value in the SIA. 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 SIA and SDA are supervisor-level SPRs.

The SDA can be read by using the **mfspr** instruction and written to by using the **mtspr** instruction (SPR 959).

9.1.1.2.3 Updating SIA and SDA

The values of the SIA and SDA registers depend on the type of event being monitored. These registers have predicted values after a PMI is signaled. A PMI may be signaled, but not serviced because the exception is masked by the MSR(EE) bit. Programmers must make sure that this bit is set active in order to take the PMI.

9.1.1.3 Monitor Mode Control Register 0 (MMCR0)

The monitor mode control register 0 (MMCR0) is a 32-bit SPR (SPR 952) 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. Control fields in the MMCR0 select the events to be counted, can enable a counter overflow to initiate a performance monitor interrupt, and specify the conditions under which counting is enabled.

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 is cleared at power up. Reading this register does not change its contents. The fields of the register are defined in Table 9-4.

Table 9-4. MMCR0 Bit Settings

Bit	Name	Description
0	DIS	Disable counting unconditionally 0 The values of the PMCn counters can be changed by hardware. 1 The values of the PMCn counters cannot be changed by hardware.
1	DP	Disable counting while in supervisor mode 0 The PMCn 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	DU	Disable counting while in user mode 0 The PMCn 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	DMS	Disable counting while MSR[PM] is set 0 The PMCn counters can be changed by hardware. 1 If MSR[PM] is set, the PMCn counters are not changed by hardware.
4	DMR	Disable counting while MSR[PM] is zero. 0 The PMCn counters can be changed by hardware. 1 If MSR[PM] is cleared, the PMCn counters are not changed by hardware.
5	ENINT	Enable performance monitor interrupt signaling. 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. This bit is cleared before passing control to the operating system.
6	DISCOUNT	Disable counting of PMC1 and PMC2 when a performance monitor interrupt is signaled (that is, ((PMCnINTCONTROL = 1) & (PMCn[0] = 1) & (ENINT = 1)) or the occurrence of an enabled time base transition with ((INTONBITTRANS =1) & (ENINT = 1)). 0 Signaling a performance monitor interrupt has no effect on the counting status of PMC1 and PMC2. 1 Signaling a performance monitor interrupt prevents the PMC1 counter from changing. The PMC2 counter does not change if PMC2COUNTCTL = 0. Because, a time-base signal could have occurred along with an enabled counter negative condition, software should always reset INTONBITTRANS to zero, if the value in INTONBITTRANS was a one.

Table 9-4. MMCR0 Bit Settings (Continued)

Bit	Name	Description
7–8	RTCSELECT	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	INTONBITTRANS	Cause interrupt signaling on bit transition (identified in RTCSELECT) 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 INTONBITTRANS.
10–15	THRESHOLD	Threshold value. All 6 bits are supported by the 604 processor; allowing threshold values from 0 to 63. The intent of the THRESHOLD support is to be able to characterize L1 data cache misses.
16	PMC1INTCONTROL	Enable interrupt signaling due to PMC1 counter negative. 0 Disable PMC1 interrupt signaling due to PMC1 counter negative. 1 Enable PMC1 Interrupt signaling due to PMC1 counter negative.
17	PMC2INTCONTROL	Enable interrupt signaling due to PMC2 counter negative. This signal overrides the setting of DISCOUNT. 0 Disable PMC2 interrupt signaling due to PMC2 counter negative. 1 Enable PMC2 Interrupt signaling due to PMC2 counter negative.
18	PMC2COUNTCTL	May be used to trigger counting of PMC2 after PMC1 has become negative or after a performance monitor interrupt is signaled. 0 Enable PMC2 counting 1 Disable PMC2 counting until PMC1 bit 0 is set or until a performance monitor interrupt is signaled. This signal can be used to trigger counting of PMC2 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	PMC1SELECT	PMC1 input selector, 128 events selectable; 25 defined. See Table 9-2.
26–31	PMC2SELECT	PMC2 input selector, 64 events selectable; 21 defined. See Table 9-3.

9.1.2 Event Counting

Counting can be enabled if conditions in the processor state match a software-specified condition. Because 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. The performance monitor (PM) bit, MSR[29] is used for this purpose. System software may set this bit when a 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[PM] together define a state that the processor (supervisor or program) and the process (marked or unmarked) may be in at any time. If this state matches a state specified by the MMCR, the state for which monitoring is enabled, counting is enabled.

The following are states that can be monitored:

- (Supervisor) only
- (User) only
- (Marked and user) only
- (Not marked and user) only
- (Marked and supervisor) only
- (Not marked and supervisor) only
- (Marked) only
- (Not marked) only

In addition, one of two unconditional counting modes may be specified:

- Counting is unconditionally enabled regardless of the states of MSR[PM] and MSR[PR]. This can be accomplished by clearing MMCR0[0–4].
- Counting is unconditionally disabled regardless of the states of MSR[PM] and MSR[PR]. This is done by setting MMCR0[0].

The performance monitor counters track how often a selected event occurs and are used to generate performance monitor exceptions when an overflow (most significant bit is a 1) situation occurs. The 604 performance monitor contains two counters. This register is cleared at startup and can be updated through an **mtspr** instruction.

The 32-bit registers can count up to 0x7FFFFFFF (2,147,483,648 in decimal) before becoming negative. The most significant bit (bit 0) of both registers is used to determine if an interrupt condition exists.

9.1.2.1 Event Selection

Event selection is handled through PMC1 and PMC2, described in Table 9-2 and Table 9-3, respectively. Event selection is described as follows:

- The event select fields are located in MMCR0. There are 7 bits associated with PMC1 and 6 bits associated with PMC2. Only the low order 5 bits are used for selection. The higher order bits are reserved for future applications.
- In the tables, a correlation is established between each counter, the events to be traced, and the pattern required for the desired selection.
- The first five events are common to both counters. These are considered to be reference events.
- Some events can have multiple occurrences per cycle, and therefore need two or three bits to represent them. These events are number 2, 4, 14, 15 for PMC1 and 2, 4, 8, 18 for PMC2.

9.1.2.2 Threshold Events

These PMC1 events are numbers 9, 10, 23, and 24. These events monitor load and store misses (with and without lateral L2 intervention). Only “marked” loads and stores (loads and stores at queue position 0) are monitored. See Section 9.1.2.2.1, “Threshold Conditions,” for more information.

When a marked operation is detected, the SDA is updated with the effective address. When the marked instruction finishes executing, the SIA will be updated with the address of that instruction. Thus, when a PMI is signaled (as a result of a threshold event) the SIA and SDA contains the exact SIA and SDA belonging to the instruction that caused PMC1 to become negative; see Section 9.1.2.2.3, “Warnings,” for further information.

9.1.2.2.1 Threshold Conditions

The ability to generate a PMI based on a threshold condition makes it possible to characterize L1 data cache misses. Specifically, the programmer should be able to identify (through repeated runs and sampling) the time distribution required to satisfy L1 cache misses. For example, if PMC1 is counting load misses and the threshold is set to two (cycles), only load misses taking more than two cycles are counted. Repeated runs with different threshold values would allow construction of a load-miss distribution chart.

When a load (or store) miss arrives in the load/store queue, the threshold control logic begins decrementing. For each cycle that passes, the threshold value in a shadow register (obtained from MMCR0[10–15]) is decremented. The threshold is exceeded when this value reaches 0, at which point the PMC1 count is updated.

While servicing the load/store misses, the SIA and SDA registers are updated to the exact instruction and data addresses at the time an interrupt condition occurs. Thus, at the end of each threshold load or store operation, the SIA contains the address of the instruction that was last monitored, and the SDA contains the address of the data of the same instruction.

9.1.2.2.2 Lateral L2 Cache Intervention

A load or store operation that misses in the L1 cache can receive its data from one of several memory devices. In a uniprocessor system, the data would likely come an L2 cache, or from main memory if no L2 cache is present. In a multiprocessor system, the data can originate from the L2 cache connected to another 604 (that is, a lateral L2 cache), in which case, the L2 controller asserts an intervention signal (L2_INT) used by the performance monitor. This signal is useful when tracking memory latencies in a SMP system. For information about the L2_intervention signal, see Section 7.2.10.3, “L2 Intervention (L2_INT)—Input.”

9.1.2.2.3 Warnings

The following warnings should be noted:

- Not all load and store operations are monitored. Only those in queue position 0 of their respective load/store queues are monitored.
- The 604 cannot accurately track threshold events with respect to the following types of loads and stores:
  - Unaligned load and store operations that cross a word boundary
  - Load and store multiple operations
  - Load and store string operations
- The lateral L2 cache intervention signal is controlled by the L2 cache controller being used. If the L2 cache controller does not provide this functionality, the events that use this signal (PMC1 events 9 and 10) become obsolete.

9.1.2.3 Nonthreshold Events

Nonthreshold events are all events except for PMC1 events 9, 10, 23, or 24. Any PMI signaled from nonthreshold events operate the same way. There is no distinction (in the SIA and SDA registers) between an interrupt generated by a time-base register bit transition or from PMC2 or PMC1 becoming negative. In these cases the SIA contains the address of the last instruction completed during the cycle the PMI was signaled. The SDA contains an effective address of some instruction currently being processed.

Under these events the SIA and SDA does not contain information belonging to the same instruction.

Appendix A
PowerPC Instruction Set Listings

This appendix lists the PowerPC 604 microprocessor instruction set as well as PowerPC instructions not implemented in the 604. 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 604 in alphabetical order by mnemonic.

Key:

▢	Reserved bits
▨	Instruction not implemented in the 604

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* [4]	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
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* [4]	31	D			A	B	OE	489		Rc
divdu*x* [4]	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
eieio	31	0 0 0 0 0			0 0 0 0 0	0 0 0 0 0	854			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
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_[4]	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_[4]	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_[4]	63	D		0 0 0 0 0	B	814		Rc
fctidz_x_[4]	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_[5]	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_[5]	63	D		0 0 0 0 0	B	0 0 0 0 0	26	Rc
fsel_x_[5]	63	D		A	B	C	23	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
fsqrt*x*[5]	63	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsqrts*x*[5]	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[4]	58	D	A	ds			0
ldarx[4]	31	D	A	B	84		0
ldu[4]	58	D	A	ds			1
ldux[4]	31	D	A	B	53		0
ldx[4]	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
lmw[3]	46	D	A	d			

Name	0	6	7	9	11	12	14	15	16	20	21	22	30	31
lswi [3]	31	D			A				NB		597			0
lswx [3]	31	D			A				B		533			0
lwa [4]	58	D			A				ds				2	
lwarx	31	D			A				B		20			0
lwaux [4]	31	D			A				B		373			0
lwax [4]	31	D			A				B		341			0
lwbrx	31	D			A				B		534			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
mcrf	19	crfD		0 0	crfS		0 0		0 0 0 0 0		0			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
mfspr [2]	31	D			spr						339			0
mfsr [1]	31	D			0	SR			0 0 0 0 0		595			0
mfsrin [1]	31	D			0 0 0 0 0				B		659			0
mftb	31	D			tbr						371			0
mtcrf	31	S			0	CRM				0	144			0
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*	63	0	FM					0	B		711			Rc
mtfsfi*x*	63	crfD		0 0	0 0 0 0 0				IMM	0	134			Rc
mtmsr [1]	31	S			0 0 0 0 0				0 0 0 0 0		146			0
mtspr [2]	31	S			spr						467			0
mtsr [1]	31	S			0	SR			0 0 0 0 0		210			0
mtsrin [1]	31	S			0 0 0 0 0				B		242			0
mulhd*x* [4]	31	D			A				B		0	73		Rc
mulhdu*x* [4]	31	D			A				B		0	9		Rc
mulhw*x*	31	D			A				B		0	75		Rc

Name	0	6	11	16	21	22	26	27	30	31
mulhwu*x*	31	D	A	B	0	11				Rc
mulld*x* [4]	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	0 0 0 0 0	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]	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	50					0
rldcl*x* [4]	30	S	A	B	mb			8		Rc
rldcr*x* [4]	30	S	A	B	me			9		Rc
rldic*x* [4]	30	S	A	sh	mb			2	sh	Rc
rldicl*x* [4]	30	S	A	sh	mb			0	sh	Rc
rldicr*x* [4]	30	S	A	sh	me			1	sh	Rc
rldimi*x* [4]	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	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0					1	0
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* [4]	31	S	A	B	27					Rc
slw*x*	31	S	A	B	24					Rc
srad*x* [4]	31	S	A	B	794					Rc
sradi*x* [4]	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* [4]	31	S	A	B	539					Rc
srw*x*	31	S	A	B	536					Rc
stb	38	S	A	d						

Name	0	6	11	16	21	22	30	31
stbu	39	S	A	d				
stbux	31	S	A	B	247			0
stbx	31	S	A	B	215			0
std [4]	62	S	A	ds			0	
stdcx. [4]	31	S	A	B	214			1
stdu [4]	62	S	A	ds			1	
stdux [4]	31	S	A	B	181			0
stdx [4]	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 [5]	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
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

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
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 [4]	31	TO	A	B	68		0
tdi [4]	02	TO	A	SIMM			
tlbia [1,5]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	370		0
tlbie [1,5]	31	0 0 0 0 0	0 0 0 0 0	B	306		0
tlbsync[1,5]	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] 64-bit instruction
[5] Optional instruction

A.2 Instructions Sorted by Opcode

Table A-2 lists the 603 instruction set sorted in numeric order by opcode, including those PowerPC instructions not implemented by the 604.

Key:

☐	Reserved bits
▨	Instruction not implemented in the 604

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 [4]	0 0 0 0 1 0	TO			A		SIMM			
twi	0 0 0 0 1 1	TO			A		SIMM			
mulli	0 0 0 1 1 1	D			A		SIMM			
subfic	0 0 1 0 0 0	D			A		SIMM			
cmpli	0 0 1 0 1 0	crfD	0	L	A		UIMM			
cmpi	0 0 1 0 1 1	crfD	0	L	A		SIMM			
addic	0 0 1 1 0 0	D			A		SIMM			
addic.	0 0 1 1 0 1	D			A		SIMM			
addi	0 0 1 1 1 0	D			A		SIMM			
addis	0 0 1 1 1 1	D			A		SIMM			
bc*x*	0 1 0 0 0 0	BO			BI		BD		AA	LK
sc	0 1 0 0 0 1	0 0 0 0 0			0 0 0 0 0		0 0 0 0 0 0 0 0 0 0 0 0 0 0		1	0
b*x*	0 1 0 0 1 0	LI							AA	LK
mcrf	0 1 0 0 1 1	crfD	0 0		crfS	0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0		0
bclr*x*	0 1 0 0 1 1	BO			BI		0 0 0 0 0	0 0 0 0 0 1 0 0 0 0		LK
crnor	0 1 0 0 1 1	crbD			crbA		crbB	0 0 0 0 1 0 0 0 0 1		0
rfi	0 1 0 0 1 1	0 0 0 0 0			0 0 0 0 0		0 0 0 0 0	0 0 0 0 1 1 0 0 1 0		0
crandc	0 1 0 0 1 1	crbD			crbA		crbB	0 0 1 0 0 0 0 0 0 1		0
isync	0 1 0 0 1 1	0 0 0 0 0			0 0 0 0 0		0 0 0 0 0	0 0 1 0 0 1 0 1 1 0		0
crxor	0 1 0 0 1 1	crbD			crbA		crbB	0 0 1 1 0 0 0 0 0 1		0
crnand	0 1 0 0 1 1	crbD			crbA		crbB	0 0 1 1 1 0 0 0 0 1		0
crand	0 1 0 0 1 1	crbD			crbA		crbB	0 1 0 0 0 0 0 0 0 1		0
creqv	0 1 0 0 1 1	crbD			crbA		crbB	0 1 0 0 1 0 0 0 0 1		0
crorc	0 1 0 0 1 1	crbD			crbA		crbB	0 1 1 0 1 0 0 0 0 1		0
cror	0 1 0 0 1 1	crbD			crbA		crbB	0 1 1 1 0 0 0 0 0 1		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*	0 1 0 0 1 1	BO	BI	0 0 0 0 0	1 0 0 0 0 1 0 0 0 0	LK
rlwimi*x*	0 1 0 1 0 0	S	A	SH	MB ME	Rc
rlwinm*x*	0 1 0 1 0 1	S	A	SH	MB ME	Rc
rlwnm*x*	0 1 0 1 1 1	S	A	B	MB ME	Rc
ori	0 1 1 0 0 0	S	A	UIMM		
oris	0 1 1 0 0 1	S	A	UIMM		
xori	0 1 1 0 1 0	S	A	UIMM		
xoris	0 1 1 0 1 1	S	A	UIMM		
andi.	0 1 1 1 0 0	S	A	UIMM		
andis.	0 1 1 1 0 1	S	A	UIMM		
rldicl*x*[4]	0 1 1 1 1 0	S	A	sh	mb 0 0 0 sh	Rc
rldicr*x*[4]	0 1 1 1 1 0	S	A	sh	me 0 0 1 sh	Rc
rldic*x*[4]	0 1 1 1 1 0	S	A	sh	mb 0 1 0 sh	Rc
rldimi*x*[4]	0 1 1 1 1 0	S	A	sh	mb 0 1 1 sh	Rc
rldcl*x*[4]	0 1 1 1 1 0	S	A	B	mb 0 1 0 0 0	Rc
rldcr*x*[4]	0 1 1 1 1 0	S	A	B	me 0 1 0 0 1	Rc
cmp	0 1 1 1 1 1	crfD 0 L	A	B	0 0 0 0 0 0 0 0 0 0	0
tw	0 1 1 1 1 1	TO	A	B	0 0 0 0 0 0 0 1 0 0	0
subfc*x*	0 1 1 1 1 1	D	A	B	OE 0 0 0 0 0 0 1 0 0 0	Rc
mulhdu*x*[4]	0 1 1 1 1 1	D	A	B	0 0 0 0 0 0 0 1 0 0 1	Rc
addc*x*	0 1 1 1 1 1	D	A	B	OE 0 0 0 0 0 0 1 0 1 0	Rc
mulhwu*x*	0 1 1 1 1 1	D	A	B	0 0 0 0 0 0 0 1 0 1 1	Rc
mfcr	0 1 1 1 1 1	D	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0 1 0 0 1 1	0
lwarx	0 1 1 1 1 1	D	A	B	0 0 0 0 0 1 0 1 0 0	0
ldx[4]	0 1 1 1 1 1	D	A	B	0 0 0 0 0 1 0 1 0 1	0
lwzx	0 1 1 1 1 1	D	A	B	0 0 0 0 0 1 0 1 1 1	0
slw*x*	0 1 1 1 1 1	S	A	B	0 0 0 0 0 1 1 0 0 0	Rc
cntlzw*x*	0 1 1 1 1 1	S	A	0 0 0 0 0	0 0 0 0 0 1 1 0 1 0	Rc
sld*x*[4]	0 1 1 1 1 1	S	A	B	0 0 0 0 0 1 1 0 1 1	Rc
and*x*	0 1 1 1 1 1	S	A	B	0 0 0 0 0 1 1 1 0 0	Rc
cmpl	0 1 1 1 1 1	crfD 0 L	A	B	0 0 0 0 1 0 0 0 0 0	0
subf*x*	0 1 1 1 1 1	D	A	B	OE 0 0 0 0 1 0 1 0 0 0	Rc

Name	0 … 5	6 … 10	11 … 15	16 … 20	21 … 30	31
ldux[4]	0 1 1 1 1 1	D	A	B	0 0 0 0 1 1 0 1 0 1	0
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*[4]	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[4]	0 1 1 1 1 1	TO	A	B	0 0 0 1 0 0 0 1 0 0	0
mulhd*x*[4]	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
mfmsr	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[4]	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
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	CRM | 0	0 0 1 0 0 1 0 0 0 0	0
mtmsr	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[4]	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
stdux[4]	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	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.[4]	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[4]	0 1 1 1 1 1	D	A	B	OE | 0 0 1 1 1 0 1 0 0 1	Rc
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

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
mullw*x*	0 1 1 1 1 1	D	A	B	OE	0 0 1 1 1 0 1 0 1 1	Rc
mtsrin	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 [1,5]	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 [4]	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 [1,5]	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 [4]	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* [4]	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* [4]	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	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* [4]	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*	0 1 1 1 1 1	D	A	B	OE | 0 1 1 1 1 0 1 0 1 1	Rc
slbia [1,4,5]	0 1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 1 1 1 1 1 0 0 1 0	0
mcrxr	0 1 1 1 1 1	crfD | 0 0	0 0 0 0 0	0 0 0 0 0	1 0 0 0 0 0 0 0 0 0	0
lswx [3]	0 1 1 1 1 1	D	A	B	1 0 0 0 0 1 0 1 0 1	0
lwbrx	0 1 1 1 1 1	D	A	B	1 0 0 0 0 1 0 1 1 0	0
lfsx	0 1 1 1 1 1	D	A	B	1 0 0 0 0 1 0 1 1 1	0
srw*x*	0 1 1 1 1 1	S	A	B	1 0 0 0 0 1 1 0 0 0	Rc
srd*x* [4]	0 1 1 1 1 1	S	A	B	1 0 0 0 0 1 1 0 1 1	Rc
tlbsync [1,5]	0 1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	1 0 0 0 1 1 0 1 1 0	0
lfsux	0 1 1 1 1 1	D	A	B	1 0 0 0 1 1 0 1 1 1	0
mfsr	0 1 1 1 1 1	D	0 | SR	0 0 0 0 0	1 0 0 1 0 1 0 0 1 1	0
lswi [3]	0 1 1 1 1 1	D	A	NB	1 0 0 1 0 1 0 1 0 1	0
sync	0 1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	1 0 0 1 0 1 0 1 1 0	0
lfdx	0 1 1 1 1 1	D	A	B	1 0 0 1 0 1 0 1 1 1	0
lfdux	0 1 1 1 1 1	D	A	B	1 0 0 1 1 1 0 1 1 1	0
mfsrin [1]	0 1 1 1 1 1	D	0 0 0 0 0	B	1 0 1 0 0 1 0 0 1 1	0
stswx [3]	0 1 1 1 1 1	S	A	B	1 0 1 0 0 1 0 1 0 1	0
stwbrx	0 1 1 1 1 1	S	A	B	1 0 1 0 0 1 0 1 1 0	0
stfsx	0 1 1 1 1 1	S	A	B	1 0 1 0 0 1 0 1 1 1	0
stfsux	0 1 1 1 1 1	S	A	B	1 0 1 0 1 1 0 1 1 1	0
stswi [3]	0 1 1 1 1 1	S	A	NB	1 0 1 1 0 1 0 1 0 1	0
stfdx	0 1 1 1 1 1	S	A	B	1 0 1 1 0 1 0 1 1 1	0
stfdux	0 1 1 1 1 1	S	A	B	1 0 1 1 1 1 0 1 1 1	0
lhbrx	0 1 1 1 1 1	D	A	B	1 1 0 0 0 1 0 1 1 0	0
sraw*x*	0 1 1 1 1 1	S	A	B	1 1 0 0 0 1 1 0 0 0	Rc
srad*x* [4]	0 1 1 1 1 1	S	A	B	1 1 0 0 0 1 1 0 1 0	Rc
srawi*x*	0 1 1 1 1 1	S	A	SH	1 1 0 0 1 1 1 0 0 0	Rc
eieio	0 1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	1 1 0 1 0 1 0 1 1 0	0
sthbrx	0 1 1 1 1 1	S	A	B	1 1 1 0 0 1 0 1 1 0	0
extsh*x*	0 1 1 1 1 1	S	A	0 0 0 0 0	1 1 1 0 0 1 1 0 1 0	Rc
extsb*x*	0 1 1 1 1 1	S	A	0 0 0 0 0	1 1 1 0 1 1 1 0 1 0	Rc
icbi	0 1 1 1 1 1	0 0 0 0 0	A	B	1 1 1 1 0 1 0 1 1 0	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 [5]	0 1 1 1 1 1	S	A	B	1 1 1 1 0 1 0 1 1 1		0
extsw [4]	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 [4]	1 1 1 0 1 0	D	A	ds			0 0
ldu [4]	1 1 1 0 1 0	D	A	ds			0 1
lwa [4]	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

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
fadds*x*	1 1 1 0 1 1	D		A		B		0 0 0 0 0	1 0 1 0 1		Rc
fsqrts*x*[5]	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*[5]	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[4]	1 1 1 1 1 0	S		A		ds				0 0	
stdu[4]	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*[5]	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*[5]	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

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
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
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* [4]	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* [4]	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* [4]	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] 64-bit instruction
[5] Optional instruction

A.3 Instructions Grouped by Functional Categories

Table A-3 through Table A-30 list the 604 instructions grouped by function, as well as the PowerPC instructions not implemented in the 604.

Key:

□	Reserved bits
▨	Instruction not implemented in the 604

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*[4]	31	D	A	B	OE	489	Rc
▨ **divdu***x*[4]	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*[4]	31	D	A	B	0	73	Rc
▨ **mulhdu***x*[4]	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**[4]	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			

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
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*[4]	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*[4]	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 22 23 24 25 26	27 28 29	30	31
rldcl*x*[4]	30	S	A	B	mb	8		Rc
rldcr*x*[4]	30	S	A	B	me	9		Rc
rldic*x*[4]	30	S	A	sh	mb	2	sh	Rc
rldicl*x*[4]	30	S	A	sh	mb	0	sh	Rc
rldicr*x*[4]	30	S	A	sh	me	1	sh	Rc
rldimi*x*[4]	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 22 23 24 25 26 27 28 29	30	31
sld*x*[4]	31	S	A	B	27		Rc
slw*x*	31	S	A	B	24		Rc
srad*x*[4]	31	S	A	B	794		Rc
sradi*x*[4]	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*[4]	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	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
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
fres*x*[5]	59	D	0 0 0 0 0	B	0 0 0 0 0	24	Rc
frsqrte*x*[5]	63	D	0 0 0 0 0	B	0 0 0 0 0	26	Rc
fsub*x*	63	D	A	B	0 0 0 0 0	20	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
fsubs*x*	59	D	A	B	0 0 0 0 0	20	Rc
fsel*x* [5]	63	D	A	B	C	23	Rc
fsqrt*x* [5]	63	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsqrts*x* [5]	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* [4]	63	D	0 0 0 0 0	B	846	Rc
fctid*x* [4]	63	D	0 0 0 0 0	B	814	Rc
fctidz*x* [4]	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	11–13	14	15	16–19	20	21–30	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	6–10	11–15	16–20	21–30	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 [4]	58	D	A	ds		0
ldu [4]	58	D	A	ds		1
ldux [4]	31	D	A	B	53	0
ldx [4]	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 [4]	58	D	A	ds		2
lwaux [4]	31	D	A	B	373	0
lwax [4]	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[4]	62	S	A	ds		0
stdu[4]	62	S	A	ds		1
stdux[4]	31	S	A	B	181	0
stdx[4]	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 nstructions

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	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	854	0
isync	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	150	0
ldarx [4]	31	D	A	B	84	0
lwarx	31	D	A	B	20	0
stdcx.[4]	31	S	A	B	214	1
stwcx.	31	S	A	B	150	1
sync	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	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

Name	0–5	6–10	11–15	16–20	21–30	31
stfiwx [5]	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	6–10	11–15	16–20	21–30	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	6–10	11–15	16–20	21–29	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	6–8	9–10	11–13	14–15	16–20	21–30	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	6–10	11–15	16–20	21–29	30	31
rfi [1]	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	50		0
sc	17	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		1	0

Table A-25. Trap Instructions

Name	0–5	6–10	11–15	16–20	21–30	31
td [4]	31	TO	A	B	68	0
tdi [4]	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	6–8	9–10	11	12–15	16–19	20	21–30	31
mcrxr	31	crfS	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
mfmsr [1]	31	D		0 0 0 0 0		0 0 0 0 0		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]	31	S		0 0 0 0 0		0 0 0 0 0		146	0
mtspr [2]	31	D		spr				467	0

Table A-27. Cache Management Instructions

Name	0–5	6–10	11–15	16–20	21–30	31
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
icbi	31	0 0 0 0 0	A	B	982	0

Table A-28. Segment Register Manipulation Instructions

Name	0 … 5	6 … 10	11	12 … 15	16 … 20	21 … 30	31
mfsr [1]	31	D	0	SR	0 0 0 0 0	595	0
mfsrin [1]	31	D	0 0 0 0 0		B	659	0
mtsr [1]	31	S	0	SR	0 0 0 0 0	210	0
mtsrin [1]	31	S	0 0 0 0 0		B	242	0

Table A-29. Lookaside Buffer Management Instructions

Name	0 … 5	6 … 10	11 … 15	16 … 20	21 … 30	31
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
tlbia [1,5]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	370	0
tlbie [1,5]	31	0 0 0 0 0	0 0 0 0 0	B	306	0
tlbsync[1,5]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	566	0

Table A-30. External Control Instructions

Name	0 … 5	6 … 10	11 … 15	16 … 20	21 … 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] 64-bit instruction
[5] Optional instruction

A.4 Instructions Sorted by Form

Table A-31 through Table A-45 list the 604 instructions grouped by form, including those PowerPC instructions not implemented in the 604.

Key:

▢	Reserved bits
▨	Instruction not implemented in the 604

Table A-31. I-Form

0–5	6–29	30	31
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

0–5	6–10	11–15	16–29	30	31
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

0–5	6–10	11–15	16–29	30	31
OPCD	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	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	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1	0

Table A-34. D-Form

0–5	6–8	9	10	11–15	16–31
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

Specific Instructions (Continued)

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
stw	36	S	A	d
stwu	37	S	A	d
subfic	08	D	A	SIMM
tdi [4]	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 [4]	58	D	A	ds	0
ldu [4]	58	D	A	ds	1
lwa [4]	58	D	A	ds	2
std [4]	62	S	A	ds	0
stdu [4]	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

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
cmp	31	crfD	0	L	A	B	0	0
cmpl	31	crfD	0	L	A	B	32	0
cntlzd*x* [4]	31	S			A	0 0 0 0 0	58	Rc
cntlzw*x*	31	S			A	0 0 0 0 0	26	Rc
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

Specific Instructions (Continued)

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
extsw_x_[4]	31	S		A	0 0 0 0 0	986	Rc
fabs_x_	63	D		0 0 0 0 0	B	264	Rc
fcfid_x_[4]	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_[4]	63	D		0 0 0 0 0	B	814	Rc
fctidz_x_[4]	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[4]	31	D		A	B	84	0
ldux[4]	31	D		A	B	53	0
ldx[4]	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[4]	31	D		A	B	373	0
lwax[4]	31	D		A	B	341	0

Specific Instructions (Continued)

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
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]	31	D	0 SR	0 0 0 0 0	595	0
mfsrin [1]	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]	31	S	0 0 0 0 0	0 0 0 0 0	146	0
mtsr [1]	31	S	0 SR	0 0 0 0 0	210	0
mtsrin [1]	31	S	0 0 0 0 0	B	242	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* [4]	31	S	A	B	27	Rc
slw*x*	31	S	A	B	24	Rc
srad*x* [4]	31	S	A	B	794	Rc
sraw*x*	31	S	A	B	792	Rc
srawi*x*	31	S	A	SH	824	Rc
srd*x* [4]	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. [4]	31	S	A	B	214	1
stdux [4]	31	S	A	B	181	0

Specific Instructions (Continued)

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
stdx [4]	31	S	A	B	149	0
stfdux	31	S	A	B	759	0
stfdx	31	S	A	B	727	0
stfiwx[5]	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 [4]	31	TO	A	B	68	0
tlbia [1,5]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	370	0
tlbie [1,5]	31	0 0 0 0 0	0 0 0 0 0	B	306	0
tlbsync[1,5]	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

Specific Instructions (Continued)

Name	0–5	6–10	11–15	16–20	21–30	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
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]	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	50	0

Table A-38. XFX-Form

0–5	6–10	11–20	21–30	31
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	6–10	11–20	21–30	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

0–5	6	7–14	15	16–20	21–30	31
OPCD	0	FM	0	B	XO	Rc

Specific Instructions

Name	0–5	6	7–14	15	16–20	21–30	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*[4]	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*[4]	31	D	A	B	OE	489	Rc
divdu*x*[4]	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*[4]	31	D	A	B	0	73	Rc
mulhdu*x*[4]	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*[4]	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* [5]	59	D	0 0 0 0 0	B	0 0 0 0 0	24	Rc
frsqrte*x* [5]	63	D	0 0 0 0 0	B	0 0 0 0 0	26	Rc
fsel*x* [5]	63	D	A	B	C	23	Rc
fsqrt*x* [5]	63	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsqrts*x* [5]	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 … 10	11 … 15	16 … 20	21 … 25	26 … 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 … 10	11 … 15	16 … 20	21 … 26	27 … 29	30	31
rldic*x* [4]	30	S	A	sh	mb	2	sh	Rc
rldicl*x* [4]	30	S	A	sh	mb	0	sh	Rc
rldicr*x* [4]	30	S	A	sh	me	1	sh	Rc
rldimi*x* [4]	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 … 10	11 … 15	16 … 20	21 … 26	27 … 30	31
rldcl*x* [4]	30	S	A	B	mb	8	Rc
rldcr*x* [4]	30	S	A	B	me	9	Rc

[1] Supervisor-level instruction
[2] Supervisor- and user-level instruction
[3] Load and store string or multiple instruction
[4] 64-bit instruction
[5] Optional instruction

A.5 Instruction Set Legend

Table A-46 provides general information on the 604 instruction set (such as the architectural level, privilege level, and form), including instructions not implemented in the 604.

Key:

▨	Instruction not implemented in the 604

Table A-46. PowerPC Instruction Set Legend

	UISA	VEA	OEA	Supervisor Level	64-Bit	Optional	Form
add*x*	√						XO
addc*x*	√						XO
adde*x*	√						XO
addi	√						D
addic	√						D
addic.	√						D
addis	√						D
addme*x*	√						XO
addze*x*	√						XO
and*x*	√						X
andc*x*	√						X
andi.	√						D
andis.	√						D
b*x*	√						I
bc*x*	√						B
bcctr*x*	√						XL
bclr*x*	√						XL
cmp	√						X
cmpi	√						D
cmpl	√						X
cmpli	√						D
▨ **cntlzd***x*	√				√		X
cntlzw*x*	√						X
crand	√						XL
crandc	√						XL

	UISA	VEA	OEA	Supervisor Level	64-Bit	Optional	Form
creqv	√						XL
crnand	√						XL
crnor	√						XL
cror	√						XL
crorc	√						XL
crxor	√						XL
dcbf		√					X
dcbi			√	√			X
dcbst		√					X
dcbt		√					X
dcbtst		√					X
dcbz		√					X
divd*x*	√				√		XO
divdu*x*	√				√		XO
divw*x*	√						XO
divwu*x*	√						XO
eciwx		√				√	X
ecowx		√				√	X
eieio		√					X
eqv*x*	√						X
extsb*x*	√						X
extsh*x*	√						X
extsw*x*	√				√		X
fabs*x*	√						X
fadd*x*	√						A
fadds*x*	√						A
fcfid*x*	√				√		X
fcmpo	√						X
fcmpu	√						X
fctid*x*	√				√		X
fctidz*x*	√				√		X
fctiw*x*	√						X
fctiwz*x*	√						X

	UISA	VEA	OEA	Supervisor Level	64-Bit	Optional	Form
fdiv*x*	√						A
fdivs*x*	√						A
fmadd*x*	√						A
fmadds*x*	√						A
fmr*x*	√						X
fmsub*x*	√						A
fmsubs*x*	√						A
fmul*x*	√						A
fmuls*x*	√						A
fnabs*x*	√						X
fneg*x*	√						X
fnmadd*x*	√						A
fnmadds*x*	√						A
fnmsub*x*	√						A
fnmsubs*x*	√						A
fres*x*	√					√	A
frsp*x*	√						X
frsqrte*x*	√					√	A
fsel*x*	√					√	A
fsqrt*x*	√					√	A
fsqrts*x*	√					√	A
fsub*x*	√						A
fsubs*x*	√						A
icbi		√					X
isync		√					XL
lbz	√						D
lbzu	√						D
lbzux	√						X
lbzx	√						X
ld	√				√		DS
ldarx	√				√		X
ldu	√				√		DS
ldux	√				√		X

	UISA	VEA	OEA	Supervisor Level	64-Bit	Optional	Form
ldx	√				√		X
lfd	√						D
lfdu	√						D
lfdux	√						X
lfdx	√						X
lfs	√						D
lfsu	√						D
lfsux	√						X
lfsx	√						X
lha	√						D
lhau	√						D
lhaux	√						X
lhax	√						X
lhbrx	√						X
lhz	√						D
lhzu	√						D
lhzux	√						X
lhzx	√						X
lmw [2]	√						D
lswi [2]	√						X
lswx [2]	√						X
lwa	√				√		DS
lwarx	√						X
lwaux	√				√		X
lwax	√				√		X
lwbrx	√						X
lwz	√						D
lwzu	√						D
lwzux	√						X
lwzx	√						X
mcrf	√						XL
mcrfs	√						X
mcrxr	√						X

	UISA	VEA	OEA	Supervisor Level	64-Bit	Optional	Form
mfcr	√						X
mffs*x*	√						X
mfmsr			√	√			X
mfspr [1]	√		√	√			XFX
mfsr			√	√			X
mfsrin			√	√			X
mftb		√					XFX
mtcrf	√						XFX
mtfsb0*x*	√						X
mtfsb1*x*	√						X
mtfsf*x*	√						XFL
mtfsfi*x*	√						X
mtmsr			√	√			X
mtspr [1]	√		√	√			XFX
mtsr			√	√			X
mtsrin			√	√			X
mulhd*x*	√				√		XO
mulhdu*x*	√				√		XO
mulhw*x*	√						XO
mulhwu*x*	√						XO
mulld*x*	√				√		XO
mulli	√						D
mullw*x*	√						XO
nand*x*	√						X
neg*x*	√						XO
nor*x*	√						X
or*x*	√						X
orc*x*	√						X
ori	√						D
oris	√						D
rfi			√	√			XL
rldcl*x*	√				√		MDS
rldcr*x*	√				√		MDS

	UISA	VEA	OEA	Supervisor Level	64-Bit	Optional	Form
rldic*x*	√				√		MD
rldicl*x*	√				√		MD
rldicr*x*	√				√		MD
rldimi*x*	√				√		MD
rlwimi*x*	√						M
rlwinm*x*	√						M
rlwnm*x*	√						M
sc	√		√				SC
slbia			√	√	√	√	X
slbie			√	√	√	√	X
sld*x*	√				√		X
slw*x*	√						X
srad*x*	√				√		X
sradi*x*	√				√		XS
sraw*x*	√						X
srawi*x*	√						X
srd*x*	√				√		X
srw*x*	√						X
stb	√						D
stbu	√						D
stbux	√						X
stbx	√						X
std	√				√		DS
stdcx.	√				√		X
stdu	√				√		DS
stdux	√				√		X
stdx	√				√		X
stfd	√						D
stfdu	√						D
stfdux	√						X
stfdx	√						X
stfiwx	√					√	X
stfs	√						D

	UISA	VEA	OEA	Supervisor Level	64-Bit	Optional	Form
stfsu	√						D
stfsux	√						X
stfsx	√						X
sth	√						D
sthbrx	√						X
sthu	√						D
sthux	√						X
sthx	√						X
stmw 2	√						D
stswi 2	√						X
stswx 2	√						X
stw	√						D
stwbrx	√						X
stwcx.	√						X
stwu	√						D
stwux	√						X
stwx	√						X
subf*x*	√						XO
subfc*x*	√						XO
subfe*x*	√						XO
subfic	√						D
subfme*x*	√						XO
subfze*x*	√						XO
sync	√						X
td	√				√		X
tdi	√				√		D
tlbia			√	√		√	X
tlbie			√	√		√	X
tlbsync			√	√		√	X
tw	√						X
twi	√						D

	UISA	VEA	OEA	Supervisor Level	64-Bit	Optional	Form
xor*x*	√						X
xori	√						D
xoris	√						D

[1] Supervisor- and user-level instruction
[2] Load and store string or multiple instruction

Appendix B
Invalid Instruction Forms

This appendix describes how invalid instructions are treated by the PowerPC 604 microprocessor.

B.1 Invalid Forms Excluding Reserved Fields

The following table 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				

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
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	
fsub[**.**]																	X	
fsubs[**.**]																	X	
fmul[**.**]														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
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
mtsr									X					X				X
mfsr									X					X				X
mtsrin										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
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 604 treats the bits listed above as causing an invalid form as “don’t cares.”

Glossary of Terms and Abbreviations

The glossary contains an alphabetical list of terms, phrases, and abbreviations used in this book. Some of the terms and definitions included in the glossary are reprinted from *IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, copyright ©1985 by the Institute of Electrical and Electronics Engineers, Inc. with the permission of the IEEE.

A

Atomic. A bus access that attempts to be part of a read-write operation to the same address uninterrupted by any other access to that address (the term refers to the fact that the transactions are indivisible). The PowerPC architecture implements atomic accesses through the **lwarx**/**stwcx.** instruction pair.

B

Biased exponent. The sum of the exponent and a constant (bias) chosen to make the biased exponent's range non-negative.

Big-endian. A byte-ordering method in memory where the address n of a word corresponds to the most significant byte. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the most significant byte.

Boundedly undefined. The results of attempting to execute a given instruction are said to be *boundedly undefined* if they could have been achieved by executing an arbitrary sequence of defined instructions, in valid form, starting in the state the machine was in before attempting to execute the given instruction. Boundedly undefined results for a given instruction may vary between implementations, and between execution attempts in the same implementation.

C

Cache. High-speed memory containing recently accessed data and/or instructions (subset of main memory).

Cache block. The cacheable unit for a PowerPC processor. The size of a cache block may vary among processors.

Cache coherency. Caches are coherent if a processor performing a read from its cache is supplied with data corresponding to the most recent value written to memory or to another processor's cache.

Cast-outs. Cache blocks that must be written to memory when a snoop miss causes the least recently used section with modified data to be replaced.

Context synchronization. Context synchronization as the result of specific instructions (such as **isync** or **rfi**) or when certain events occur (such as an exception). During context synchronization, all instructions in execution complete past the point where they can produce an exception; all instructions in execution complete in the context in which they began execution; all subsequent instructions are fetched and executed in the new context.

D

Denormalized number. A nonzero floating-point number whose exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero.

E

E**xception**. A condition encountered by the processor that requires special processing.

Exception handler. A software routine that executes when an exception occurs. Normally, the exception handler corrects the condition that caused the exception, or performs some other meaningful task (such as aborting the program that caused the exception). The addresses of the exception handlers are defined by a two-word exception vector that is branched to automatically when an exception occurs.

Execution synchronization. All instructions in execution are architecturally complete before beginning execution (appearing to begin execution) of the next instruction. Similar to context synchronization but doesn't force the contents of the instruction buffers to be deleted and refetched.

Exponent. The component of a binary floating-point number that normally signifies the integer power to which two is raised in determining the value of the represented number. Occasionally the exponent is called the signed or unbiased exponent.

F

Floating-point register (FPR). Any of the 32 registers in the floating-point register file. These registers provide the source operands and destination results for floating-point instructions. Load instructions move data from memory to FPRs, and store instructions move data from FPRs to memory.

Fraction. The field of the significand that lies to the right of its implied binary point.

G

General-purpose register (GPR). Any of the 32 registers in the register file. These registers provide the source operands and destination results for all data manipulation instructions. Load instructions move data from memory to registers, and store instructions move data from registers to memory.

I

IEEE 754. A standard written by the Institute of Electrical and Electronics Engineers that defines operations of binary floating-point arithmetic and representations of binary floating-point numbers.

Interrupt. An asynchronous exception.

K

Kill. An operation that causes a cache block to be invalidated.

L

Latency. The number of clock cycles necessary to execute an instruction and make ready the results of that instruction.

Little-endian. A byte-ordering method in memory where the address *n* of a word corresponds to the least significant byte. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 being the most significant byte.

M

Mantissa. The decimal part of logarithm.

Memory-mapped accesses. Accesses whose addresses use the segmented or block address translation mechanisms provided by the MMU and that occur externally with the bus protocol defined for memory.

Memory coherency. Refers to memory agreement between caches in a multiple processor and system memory (for example, MESI cache coherency).

Memory consistency. Refers to agreement of levels of memory with respect to a single processor and system memory (e.g. on-chip cache, secondary cache, and system memory).

Memory management unit. The functional unit that translates the effective address bits to physical address bits.

N

NaN. An abbreviation for Not a number; a symbolic entity encoded in floating-point format. There are two types of NaNs—signaling NaNs and quiet NaNs.

No-op. No-operation. A single-cycle operation that does not affect registers or generate bus activity.

O

Overflow. An error condition that occurs during arithmetic operations when the result cannot be stored accurately in the destination register(s). For example, if two 32-bit numbers are added, the sum may require 33 bits due to carry.

P

Page. A 4-Kbyte area of memory, aligned on a 4-Kbyte boundary.

Pipelining. A technique that breaks instruction execution into distinct steps so that multiple steps can be performed at the same time.

Precise exceptions. The pipeline can be stopped so the instructions that preceded the faulting instruction can complete, and subsequent instructions can be executed from scratch. The system is precise unless one of the imprecise modes for invoking the floating-point enabled exception is in effect.

Q

Quiet NaNs. Propagate through almost every arithmetic operation without signaling exceptions. These are used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when invalid.

S

Signaling NaNs. Signal the invalid operation exception when they are specified as arithmetic operands

Significand. The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

Static branch prediction. Mechanism by which software (for example, compilers) can give a hint to the machine hardware about the direction the branch is likely to take.

Sticky bit. A bit that when set must be cleared explicitly.

Superscalar machine. A machine that can issue multiple instructions concurrently from a conventional linear instruction stream.

Supervisor mode. The privileged operation state of the a processor. In supervisor mode, software can access all control registers and can access the supervisor memory space, among other privileged operations.

U

Underflow. An error condition that occurs during arithmetic operations when the result cannot be represented accurately in the destination register. For example, underflow can happen if two floating-point fractions are multiplied and the result is a single-precision number. The result may require a larger exponent and/or mantissa than the single-precision format makes available. In other words, the result is too small to be represented accurately.

Unified cache. Combined data and instruction cache.

User mode. The unprivileged operating state of a processor. In user mode, software can only access certain control registers and can only access user memory space. No privileged operations can be performed.

W

Write-through. A memory update policy in which all processor write cycles are written to both the cache and memory.

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX