PowerPC™ 602

RISC Microprocessor User's Manual

IBM

PowerPC

Ⓜ *MOTOROLA*

CONTENTS

About This Book

Chapter 1
Overview

CONTENTS

Chapter 2
PowerPC 602 Microprocessor Programming Model

CONTENTS

CONTENTS

CONTENTS

Chapter 3
Instruction and Data Cache Operation

CONTENTS

CONTENTS

Chapter 4
Exceptions

CONTENTS

Chapter 5
Memory Management

CONTENTS

Chapter 6
Instruction Timing

CONTENTS

Chapter 7
Signal Descriptions

CONTENTS

CONTENTS

Chapter 8
System Interface Operation

CONTENTS

CONTENTS

Glossary of Terms and Abbreviations

Index

ILLUSTRATIONS

ILLUSTRATIONS

ILLUSTRATIONS

TABLES

TABLES

TABLES

TABLES

TABLES

About This Book

The primary objective of this manual is to help hardware and software designers who are working with the PowerPC 602™ 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. Contact your local sales representative to obtain a copy of *The Programming Environments Manual*.

The *PowerPC 602 RISC Microprocessor User's Manual* summarizes features of the 602 that are not defined by the architecture. This document and *The Programming Environments Manual* distinguish 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.

Note also that the PowerPC architecture does not specify whether certain functionality be implemented in hardware or software. Similarly, a PowerPC implementation may provide functionality that offers alternatives to the PowerPC architecture in addition to that defined by the PowerPC architecture.

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 602-specific registers and progressing to more specialized topics such as 602-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 PowerPC Architecture: A Specification for a New Family of RISC Processors defines the architecture from the perspective of the three programming environments and remains the defining document for the PowerPC architecture.

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 602. 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 602 Microprocessor Programming Model," is useful for software engineers who need to understand the 602-specific registers, operand conventions, and details regarding how PowerPC instructions are implemented on the 602.

- Chapter 3, "Instruction and Data Cache Operation," provides a discussion of the cache and memory model as implemented on the 602.

- Chapter 4, "Exceptions," describes the exception model as implemented on the 602.

- Chapter 5, "Memory Management," provides descriptions of the PowerPC address translation and memory protection mechanism as implemented on the 602.

- Chapter 6, "Instruction Timing," describes instruction timing in the 602.

- Chapter 7, "Signal Descriptions," describes individual signals defined for the 602.

- Chapter 8, "System Interface Operation," describes interface operations on the 602.

- Chapter 9, "Power Management," describes the operation of the power management hardware and software facilities incorporated in the 602.

- Appendix A, "PowerPC Instruction Set Listings," lists all the PowerPC instructions. Instructions are grouped according to mnemonic, opcode, function, and form.

- Appendix B, "Instructions Not Implemented," describes the 32-bit and 64-bit PowerPC instructions that are not implemented in the 602.

- Appendix C, "Boundary-Scan Testing Support," provides a boundary-scan interface for board-level testing.

- This manual also includes a glossary and an index.

In this document, the term "602" is used as an abbreviation for the phrase, "PowerPC 602 Microprocessor." The PowerPC 602 microprocessors are available from IBM as PPC602 and from Motorola as MPC602.

Additional 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 602 RISC Microprocessor Technical Summary,* Rev 1
MPC602/D (Motorola Order Number) and MPR602TSU-02 (IBM Order Number)

- *PowerPC 603™ RISC Microprocessor Technical Summary*, Rev 3
MPC603/D (Motorola order number) and MPR603TSU-03 (IBM order number)

- *PowerPC 603e™ RISC Microprocessor Technical Summary*, Rev 0
MPC603E/D (Motorola order number) and MPR603TSU-04 (IBM order number)

- *PowerPC 603e RISC Microprocessor User's Manual (with Supplement for PowerPC 603 Microprocessor)*, MPC603EUM/AD (Motorola order number) and MPR603EUM-01 (IBM order number)

- *PowerPC 604™ RISC Microprocessor Technical Summary*, Rev 1
MPC604/D (Motorola order number) and MPR604TSU-02 (IBM order number)

- *PowerPC 604 RISC Microprocessor User's Manual*, Rev 0
MPC604UM/AD (Motorola order #) and MPR604UMU-01 (IBM order #)

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

Motorola Electronic Support

Motorola provides electronic support through the following channels. The technical support BBS, known as AESOP (Application Engineering Support through On-Line Productivity), can be reached by modem or the Internet.

Modem: Call 1-800-843-3451 (outside U.S. or Canada, call (512) 891-3650) on a modem that runs at 14,400 bps or slower. Set your software to C/8/1/F emulating a VT100.

Internet: This access is provided via telnet at pirs.aus.sps.mot.com [129.38.233]; or through the world-wide web at http://pirs.aus.sps.mot.com and http://www.mot.com/powerpc/.

Note that the code for implementing the software table search routines can be acquired from Motorola's home page:
http://www.mot.com/pub/SPS/powerpc/library/user_man/602mmu.txt.

Apps FAX Line: You may FAX questions to 1-800-248-8567.

IBM Electronic Support

IBM provides electronic support through the following channels:

Internet: This access is provided through the following world-wide web locations:

   IBM world-wide web home page at http://www.ibm.com

   IBM Microelectronics world-wide web home page at http://www/chips.ibm.com

FAX: IBM Microelectronics FAX service at (415) 855-4121

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.
mtspr(SPR_NAME)	In text, the SPR accessed by an **mtspr** or **mfspr** instruction is identified in parenthesis after the instruction. This should not be confused with the instruction syntax for these instructions.
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

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
BAT	Block address translation
BIST	Built-in self test
BIU	Bus interface unit
BPU	Branch processing unit
COP	Common on-chip processor
CR	Condition register
CTR	Count register
DAR	Data address register
DBAT	Data BAT
DEC	Decrementer (register)

Table i. Acronyms and Abbreviated Terms (Continued)

Term	Meaning
DSISR	Register used for determining the source of a DSI exception
DTLB	Data translation lookaside buffer
EA	Effective address
EAR	External access register
ESASRR	ESA save and restore register
FIFO	First-in-first-out
FPR	Floating-point register
FPSCR	Floating-point status and control register
FPU	Floating-point unit
GPR	General-purpose register
HID0(1)	Hardware implementation dependent (register) 0(1)
IABR	Instruction address breakpoint register
IBAT	Instruction BAT
IBR	Interrupt base register
IEEE	Institute of Electrical and Electronics Engineers
ITLB	Instruction translation lookaside buffer
IQ	Instruction queue
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
LT	Integer tag register
MEI	Modified/exclusive/ invalid
MESI	Modified/exclusive/shared/invalid—cache coherency protocol
MMU	Memory management unit
MSB	Most-significant byte
msb	Most-significant bit
MSR	Machine state register

Table i. Acronyms and Abbreviated Terms (Continued)

Term	Meaning
NaN	Not a number
No-op	No operation
OEA	Operating environment architecture
PLL	Phase-locked loop
POWER	Performance Optimized with Enhanced RISC architecture
PTE	Page table entry
PTEG	Page table entry group
PVR	Processor version register
RISC	Reduced instruction set computing/computer
RPA	Required physical address (register)
RTL	Register transfer language
RWITM	Read with intent to modify
SDR1	Register that specifies the page table base address for virtual-to-physical address translation
SER	ESA enable register
SEBR	ESA enable base register
SIMM	Signed immediate value
SP	Single-precision tag register
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
TCR	Timer control register
TLB	Translation lookaside buffer
UIMM	Unsigned immediate value
UISA	User instruction set architecture
VEA	Virtual environment architecture
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 602™ microprocessor features, including a block diagram showing the major functional components. It provides information about how the 602 implementation complies with the PowerPC™ architecture definition.

1.1 PowerPC 602 Microprocessor Overview

This section describes features of the 602, provides a block diagram showing the major functional units, and describes in a general way how the 602 operates.

The 602 is a low-cost, low-power implementation of the PowerPC microprocessor family of reduced instruction set computer (RISC) microprocessors. The 602 implements the 32-bit portion of the PowerPC architecture, which provides 32-bit effective addresses, integer data types of 8, 16, and 32 bits, and floating-point data types of 32 and 64 bits. Floating-point operations involving either 32- or 64-bit data types in single-precision format are supported; however, floating-point operations involving 64-bit data types in double-precision format are not implemented in hardware and are instead trapped for emulation in software. For more information about how the architecture defines addressing and data types, see *PowerPC Microprocessor Family: The Programming Environments* (also referred to as *The Programming Environments Manual*).

The 602 provides dynamic and static power-saving modes. The three static modes—nap, doze, and sleep—progressively reduce the amount of power required by the processor. Dynamic power management mode allows the processor to reduce power consumption by providing clocking only to those functional units that are active, without affecting operational performance, software execution, or external hardware.

The 602 can simultaneously fold one branch instruction and dispatch one nonbranch instruction per clock cycle to any one of three execution units. Instructions can execute out of order; however, the instructions complete and write back in program order.

The 602 has four execution units—an integer unit (IU), a floating-point unit (FPU), a branch processing unit (BPU), and a load/store unit (LSU). The ability to execute four instructions in parallel and the use of simple instructions with rapid execution times yield high efficiency and throughput for 602-based systems. Most integer instructions execute in

one clock cycle. The FPU is pipelined such that typically when the FPU pipeline is full a single-precision instruction can complete on each clock cycle.

The 602 provides independent on-chip, 4-Kbyte, two-way set-associative, physically addressed caches for instructions and data and on-chip instruction and data memory management units (MMUs). The 602 MMUs contain 32-entry, two-way set-associative, data and instruction translation lookaside buffers (DTLB and ITLB). The TLBs cache the translations for the most recently used pages. The 602 also supports block address translation through the use of two independent instruction and data block address translation (IBAT and DBAT) arrays of four entries each. Effective addresses are compared simultaneously with all four entries in the BAT array during block translation. If an effective address matches any entry in the BATs, the BAT entry takes priority over any potential matches in the TLBs. In real addressing mode, the MMU translation is disabled.

The 602 provides an additional memory protection mechanism not defined by the PowerPC architecture. The 602's protection-only mode can control whether instructions can be fetched from 4-Kbyte instruction pages and whether data can be written to 4-Kbyte data pages. Protection-only mode also controls the ability to execute the 602-specific **esa** (Enable Supervisor Access) instruction on a per-page basis. When this instruction is enabled, its execution causes the processor to enter supervisor mode. In protection-only mode, the effective address is also used as the physical address just as in real addressing mode, but the MMU page addressing mechanism is enabled to enforce this protection. For details, refer to Section 1.1.5.1, "Memory Management Units (MMUs)."

The 602 has a single bus interface for transferring 32-bit addresses and either 32- or 64-bit data. This bus is time-multiplexed, as described in Section 1.2.8, "System Interface." When the bus is in the data phase, it can be configured dynamically to perform as a 32- or 64-bit data bus.

The 602 interface protocol allows multiple masters to compete for system resources through a central external arbiter. The 602 provides a three-state coherency protocol that supports the modified, exclusive, and invalid (MEI) cache states. This protocol is a compatible subset of the MESI (modified/exclusive/shared/invalid) four-state protocol and operates coherently in systems that contain four-state caches.

The amount of data that can be transferred per bus clock cycle depends on whether the bus is configured as a 32- or 64-bit bus. In 64-bit mode, the bus can transfer up to 64 bits in a single-beat (nonburst) transaction and an entire eight-word cache block in a four-beat burst transaction. In 32-bit mode, there are two types of nonburst transactions. A single-beat transaction transfers up to 32 bits and a double-beat transaction transfers 64 bits. In 32-bit mode, the burst transaction requires eight beats to transfer a cache block of data.

The 602 uses an advanced, 3.3-V CMOS process technology and maintains full interface compatibility with TTL devices.

1.1.1 PowerPC 602 Microprocessor Features

This section describes details of the 602's implementation of the PowerPC architecture. Major features of the 602 are as follows:

- High-performance microprocessor with parallel execution units
  - One instruction is fetched from the instruction queue per clock cycle
  - One instruction can be issued and one retired per clock cycle
  - As many as four instructions in execution per clock cycle
  - Single-cycle execution for most instructions
- Four independent execution units and two register files
  - Branch processing unit (BPU)
    - Zero-cycle branch capability (branch folding)
    - Programmable static branch prediction on unresolved conditional branches
    - BPU that performs CR-lookahead operations
  - A 32-bit integer unit (IU)
    - Thirty-two 32-bit general-purpose registers (GPRs) for integer operands
  - A 32-bit floating-point unit (FPU)
    - Fully IEEE 754-compliant FPU for single-precision operations
    - Emulation support for double-precision operations
    - An implementation of the non-IEEE floating-point mode
    - Thirty-two 32-bit floating-point registers (FPRs) for single-precision operands
  - A load/store unit (LSU) for data transfer between data cache and GPRs and FPRs
- Rename registers to allow data-dependent instructions to access source data before it has been written back to architected registers
  - Four GPR rename buffers
  - Four FPR rename buffers
  - One rename buffer each for the condition register (CR), link register (LR), and count register (CTR)
- Instruction pipelining
  - Instruction unit capable of simultaneously folding out a branch instruction and dispatching one instruction per clock cycle from the instruction queue
  - A four-entry instruction queue that provides lookahead capability
  - Independent pipelines with feed-forwarding that reduces data dependencies in hardware

- Separate caches for instructions and data (Harvard architecture)
  - 4-Kbyte data cache—two-way set-associative, physically addressed; LRU replacement algorithm
  - 4-Kbyte instruction cache—two-way set-associative, physically addressed; LRU replacement algorithm
  - Eight-word cache block can be updated by a burst operation
  - Cache write-back or write-through operation programmable on a per-page or per-block basis
- Memory management features
  - Address translation facilities for 4-Kbyte page size, variable block size, and 256-Mbyte segment size
  - A 32-entry, two-way set-associative ITLB
  - A 32-entry, two-way set-associative DTLB
  - 52-bit virtual address; 32-bit physical address
  - Four-entry data and instruction BAT arrays providing 128-Kbyte to 256-Mbyte blocks
  - Efficient software table search operation aided by hardware assistance
  - Ability to inhibit instruction fetching on a page or block basis. This is provided by the 602-specific NE bit that provides the same instruction fetching control as the SR[N] bit for smaller units of memory.
  - Additional MSR bit (MSR[AP]) that can limit supervisor-level software to accessing supervisor-level memory space only
  - Programmable default cache control attributes available (HID0[WIMG]) for use when the processor is in real addressing mode or in protection-only mode
  - Protection-only mode
    - Uses the TLB translation mechanism to control instruction fetching for instruction pages, writing to data pages, and enabling the 602-specific **esa** instruction, which when executed puts the processor in supervisor mode
    - Control is defined for 4-Kbyte pages (32, 4-Kbyte pages defined in each TLB entry)
    - The translation mechanism is not used to determine the physical address (effective address generated by the code is used for the physical address as in real-addressing mode).

- Facilities for enhanced system performance
  - — A 64-bit (address and data multiplexed) external data bus with burst transfers
  - — Dynamic bus sizing allows the data bus to function as either a 32- or 64-bit bus
  - — Support for injected snoops by other devices during burst read operations
  - — Ability to broadcast a line-fill address, during the address phase of a write-back transaction on the bus
- Alternative method for entering supervisor mode without synchronizing the processor. The Enable Supervisor Access (**esa**) instruction is defined by the 602 to cause the processor to enter supervisor mode without taking an exception. The ability to execute this instruction is administered by the MMU on a per block or page basis, depending on the type of address translation used.
- Integrated power management
  - — Low-power 3.3-volt design
  - — Internal processor/bus clock multiplier that provides 2/1 and 3/1 ratios
  - — Three static power-saving modes—doze, nap, and sleep
  - — Automatic dynamic power reduction on an internal subunit level of granularity, on a per clock cycle basis, when the subunits are idle
- In-system testability and debugging features through JTAG boundary-scan capability

1.1.2 Block Diagram

The 602 block diagram in Figure 1-1 illustrates how the execution units—IU, FPU, BPU, and LSU—operate independently and in parallel.

Figure 1-1. PowerPC 602 Microprocessor Block Diagram

PowerPC 602 RISC Microprocessor User's Manual
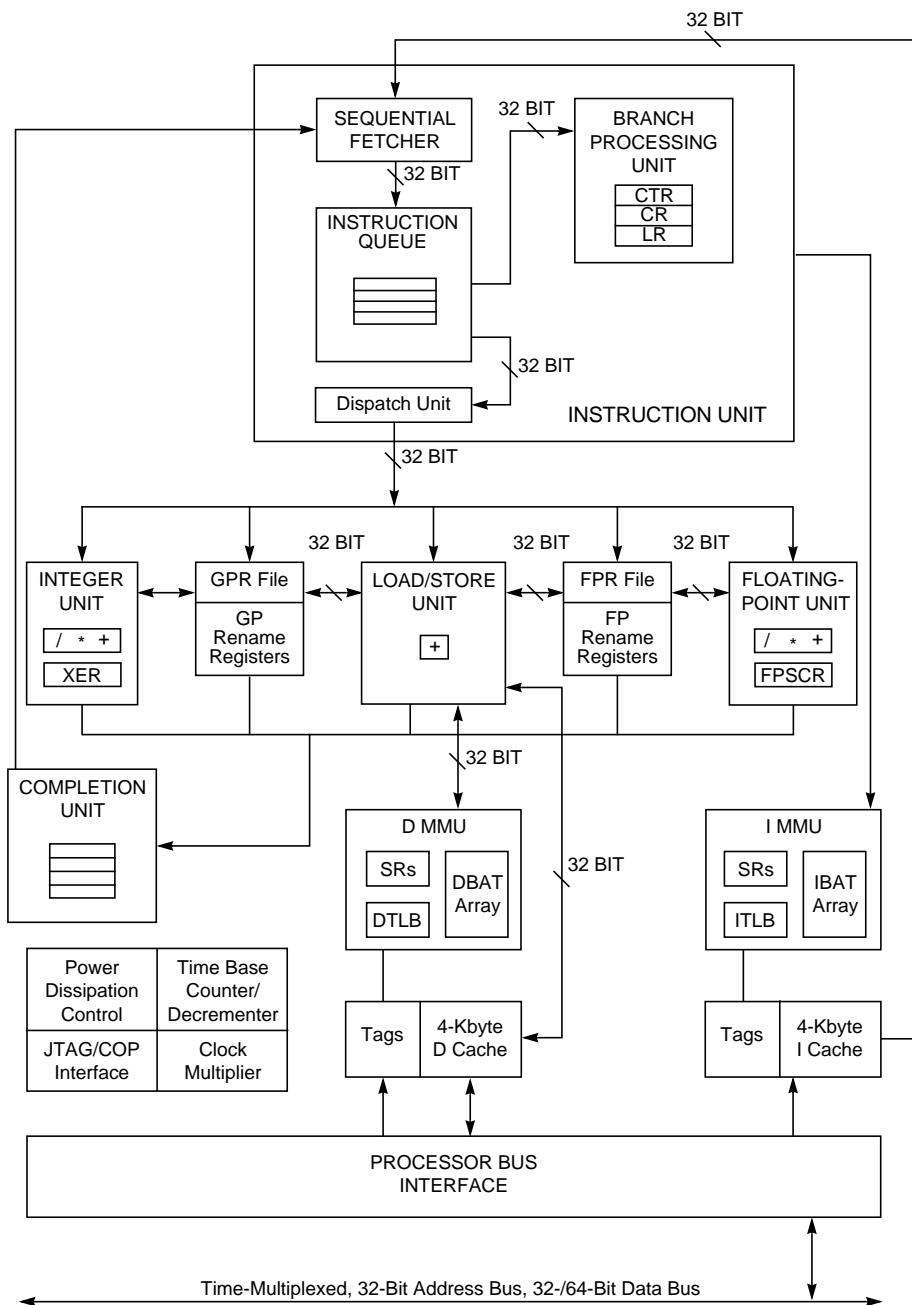
1.1.3 Instruction Pipeline

As shown in Figure 1-2, the instruction pipeline in the 602 has four stages:

- Fetch—During this stage, instructions are fetched from the instruction cache.

- Decode and dispatch—During this stage, instructions are decoded, branch instructions are folded out, and instructions are dispatched for execution once all the resources needed for execution are available.

- Execute—During this stage, instructions are executed in the LSU, IU, or FPU.

- Complete and write-back—During this stage, all results are committed to the architectural registers.

```
                          ┌─────────────────┐
                          │      Fetch      │
                          └────────┬────────┘
                                   │
                                   ▼
One instruction dispatch per clock cycle │ Decode/Dispatch │
                          └────────────────┘
        ┌ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐
                                              Execute Stage
        │     ┌──────┐   ┌──────┐   ┌──────┐            │
              │  IU  │   │ FPU  │   │ LSU  │
        │     └──────┘   └──────┘   └──────┘            │

        └ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘
                       ┌────────────────────┐
                       │ Complete/Write-back │
                       └────────────────────┘
```
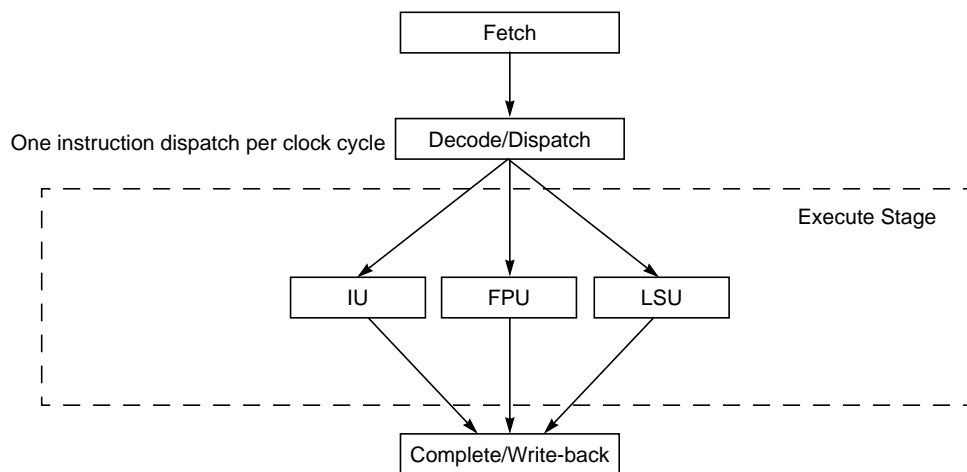
Figure 1-2. Pipeline Diagram

Note that an instruction may remain in a single pipeline stage for multiple processor cycles and may simultaneously occupy more than one pipeline stage.

If an exception is taken during the execute or the complete/write-back stages, all following instructions are flushed, any execution results in rename buffers are discarded, and fetching begins at the appropriate target address. For an overview and definition of exceptions supported in the 602, see Section 1.2.5.1, "PowerPC Exception Model."

The 602 provides address translation and protection facilities, including an ITLB, a DTLB and IBAT and DBAT arrays. Instruction fetching and dispatch are handled by the instruction unit. The MMUs translate effective addresses for fetching instructions and for reading and writing data to and from the physically addressed caches or external memory. For more information, see Sections 1.1.3.1, "Instruction Unit," and 1.1.5.1, "Memory Management Units (MMUs)."

1.1.3.1 Instruction Unit

As shown in Figure 1-1, the 602 instruction unit, which contains a fetch unit, instruction queue, dispatch unit, and BPU, provides centralized control of instruction flow to the execution units. The instruction unit determines the address of the next instruction to be fetched based on information from the sequential fetcher and from the BPU.

The instruction unit fetches the instructions from the instruction cache into the instruction queue (IQ). The BPU extracts branch instructions from the instruction queue and uses the static branch prediction defined by the PowerPC architecture specification on unresolved conditional branches. This allows the instruction unit to fetch instructions from a predicted target instruction stream while a conditional branch is evaluated. The BPU folds out branch instructions for unconditional branches or for conditional branches unaffected by instructions in progress in the execution pipeline.

Instructions issued beyond a predicted branch do not complete execution until the branch is resolved, preserving the programming model of sequential execution. If any of these instructions are to be executed in the BPU, they are decoded but not issued. Instructions to be executed by the FPU, IU, and LSU are issued and allowed to complete up to the register write-back stage. Write-back is allowed when a correctly predicted branch is resolved, and instruction execution continues without interruption along the predicted path. An instruction is dispatched only if there is an entry available for it in the completion unit. If no completion buffers are available, instruction dispatch stalls until an entry is available.

If branch prediction is incorrect, the instruction unit flushes all predicted path instructions and instructions are issued from the correct path.

1.1.3.1.1 Instruction Queue (IQ) and Dispatch Unit

The instruction queue, shown in Figure 1-1, holds as many as four instructions and loads one instruction from the instruction cache during a single cycle. The instruction fetch unit continuously loads as many instructions as space in the IQ allows. If one of the instructions loaded is a branch instruction, it is dispatched to the BPU. One nonbranch instruction per cycle can be dispatched to any one of the three other execution units. Dispatching is facilitated to the IU, FPU, and LSU by the provision of a reservation station at each unit. The dispatch unit checks for source and destination register dependencies, determines dispatch serializations, and inhibits subsequent instruction dispatching as required.

For a more detailed overview of instruction dispatch, see Section 1.2.7, "Instruction Timing."

1.1.3.1.2 Branch Processing Unit (BPU)

The BPU receives branch instructions from the fetch unit and performs CR lookahead operations on conditional branches to resolve them early, achieving the effect of a zero-cycle branch in many cases.

The BPU uses a bit in the instruction encoding to predict the direction of the conditional branch. Therefore, when an unresolved conditional branch instruction is encountered, the 602 fetches instructions from the predicted target stream until the conditional branch is resolved.

The BPU contains an adder for computing branch target addresses and three user-control registers—the link register (LR), the count register (CTR), and the condition register (CR). The BPU calculates the return pointer for subroutine calls and saves it into the LR for certain types of branch instructions. The LR also contains the branch target address for the Branch Conditional to Link Register (**bclr**x) instruction. The CTR contains the branch target address for the Branch Conditional to Count Register (**bcctr**x) instruction. The contents of the LR and CTR can be copied to or from any GPR. Because the BPU uses dedicated registers rather than GPRs or FPRs, execution of branch instructions is largely independent from execution of integer and floating-point instructions.

1.1.3.1.3 Completion Unit

When an instruction is dispatched, a place is reserved in the completion buffer, which ensures that instructions complete in program order. Completing an instruction commits the 602 to any architectural register changes caused by that instruction. In-order completion ensures the correct architectural state when the 602 must recover from a mispredicted branch or an exception.

Instruction state and other information required for completion is kept in a first-in-first-out (FIFO) queue of four completion buffers. A single completion buffer is allocated for each instruction as it enters the dispatch unit. If no completion buffers are available, instruction dispatch stalls. A maximum of one instruction per cycle is completed in order from the queue.

This unit is responsible for ensuring that exceptions are handled in an orderly way.

1.1.4 Independent Execution Units

The PowerPC architecture's support for independent execution units allows the implementation of processors with out-of-order instruction execution. For example, because branch instructions do not depend on GPRs or FPRs, branches can often be resolved early, eliminating stalls caused by taken branches.

Branch instructions do not execute in the same sense that arithmetic, logical, or load/store instructions do. As shown in Figure 1-1, the IU, FPU, and LSU are arranged in parallel to one another. The execution units are described in the following sections.

1.1.4.1 Integer Unit (IU)

The IU executes all integer instructions. The IU executes one integer instruction at a time, performing computations with its arithmetic logic unit (ALU), multiplier, divider, and the XER register. Most integer instructions are single-cycle instructions. Thirty-two 32-bit GPRs are provided to support integer operations. Four rename registers are implemented

for the GPRs. These rename registers allow instructions that have finished execution to make their results available to subsequent instructions before those results can be sent to the architected GPR. Rename registers also eliminate stalls due to contention for GPRs. The 602 writes the contents of the rename registers to the appropriate GPR when integer instructions are retired by the completion unit.

The IU executes all integer arithmetic instructions, condition register logical instructions, synchronization, and move to/from instructions.

1.1.4.2 Floating-Point Unit (FPU)

The FPU contains a single-precision multiply-add array and the floating-point status and control register (FPSCR). The multiply-add array allows the 602 to efficiently implement multiply, add, and multiply-add operations. The FPU is pipelined so that single-precision instructions can be issued back-to-back. Thirty-two 32-bit FPRs support single-precision floating-point operations. Four rename registers are implemented for the FPRs. These rename registers allow instructions that have finished execution to make their results available to subsequent instructions before those results can be sent to the architected FPR. Rename registers also eliminate stalls due to contention for FPRs. The 602 writes the contents of the rename registers to the appropriate FPR when floating-point instructions are retired by the completion unit.

In the 602, all double-precision arithmetic operations, floating-point load or store operations that involve double-precision operands that cannot be expressed as single-precision values, and operations producing denormalized numbers are handled by emulation software. The 602 traps to an exception handler when it encounters these operands or operations.

The 602 can also be operated in the non-IEEE floating-point mode. For a result of divide by zero, invalid, overflow, or underflow, this mode allows the 602 to produce predetermined values that may not conform to IEEE 754. The non-IEEE mode is useful for time-critical applications such as graphics applications.

1.1.4.3 Load/Store Unit (LSU)

The LSU executes all load and store instructions and provides the data transfer interface between the GPRs, FPRs, and the cache/memory subsystem. The LSU calculates effective addresses, performs data alignment, and traps on load/store string instructions. Load/store string instructions are emulated in software.

Load and store instructions are issued and translated in program order; however, the actual memory accesses can occur out of order. Synchronizing instructions are provided to enforce strict ordering.

Cacheable loads, when free of data dependencies, execute with a maximum throughput of one per cycle and a two-cycle total latency. Data returned from the cache is held in a rename register until the completion logic commits the value to a GPR or FPR. Data in a rename register can be used by an executing instruction before that data has been written to a

register file. Store instructions cannot be executed out of order and are held in the store queue until the completion logic signals that the store operation is to be completed to memory. The time required to perform the actual load or store operation depends on whether the operation involves the cache or the system memory.

The LSU executes all load, store, cache control, and memory control instructions.

1.1.5 Memory Subsystem

The 602 provides support for cache and memory management through separate instruction and data memory management units. The 602 also provides separate 4-Kbyte instruction and data caches and an efficient processor bus interface to facilitate access to main memory and other bus subsystems. The memory subsystem support functions are described in the following sections.

1.1.5.1 Memory Management Units (MMUs)

The 602's MMUs support up to 4 Petabytes (2^{52}) of virtual memory and 4 Gigabytes (2^{32}) of physical memory (referred to as real memory in the architecture specification) for instruction and data. The MMUs also control access privileges for these spaces on block and page granularities. Referenced and changed status is maintained by the processor for each page to assist implementation of a demand-paged virtual memory system.

The LSU calculates effective addresses for data load and store operations, and performs data alignment to and from cache memory. The instruction unit calculates the effective addresses for instruction fetching.

After an address is generated, the higher-order bits of the effective address are translated by the appropriate MMU into physical address bits. Simultaneously, the lower-order address bits (which are untranslated and the same for both logical and physical addresses), are directed to the on-chip caches where they form the index into the two-way set-associative tag array. After translating the address, the MMU passes the higher-order bits of the physical address to the cache, and the cache lookup completes. For cache-inhibited accesses or accesses that miss in the cache, the untranslated lower-order address bits are concatenated with the translated higher-order address bits; the resulting 32-bit physical address is then used by the memory unit and the system interface, which accesses external memory.

The MMU also translates addresses and enforces memory protection supervisor/user privilege level of the access in relation to whether the access is a load or store.

For instruction accesses, the MMU performs an address lookup in the 32 entries of the ITLB, and in the IBAT array. If an effective address hits in both the ITLB and the IBAT array, the IBAT array translation takes priority. Data accesses cause a lookup in the DTLB and DBAT array for the physical address translation. In most cases, the physical address translation resides in one of the TLBs or BATs, and the physical address bits are readily available to the on-chip cache.

If an access misses in the BATs, the OEA-defined page address translation is used unless HID0[PO] is set, in which case the 602-specific protection-only mode is used.

The 602-specific protection-only mode enables each TLB to protect up to 128 Kbytes per entry (4 Mbytes per TLB). Effective address translation is not performed for TLBs in protection-only mode. Protection-only mode is used if the effective address misses in the BATs and the protection-only mode is enabled (HID0[PO] = 1), otherwise, page address translation is used. In protection-only mode, the MMU is not used to translate the effective address (the effective address is used for the physical address), and is used only to enforce protection for each 4-Kbyte page. The protection consists of the following:

- Write enabling—The WE bit determines whether data pages can be written to.
- Instruction fetching—The NE bit controls whether instructions (including the **esa** instruction) can be fetched from the current page.
- Enabling execution of the **esa** instruction—The SE bit controls whether the **esa** instruction can be executed from the current page. If fetching is disabled, the SE bit is a don't care.

In either page address translation mode or protection-only mode, when the address misses in the TLBs, the 602 provides hardware assistance for software to perform a search of the translation tables in memory. The hardware assist consists of the following features:

- Separate exception vectors defined for instruction translation miss, data load translation miss, and data store translation miss
- Automatic storage of the missed effective address in the 602-specific IMISS and DMISS registers
- Automatic generation of the primary and secondary hashed real address of the page table entry group (PTEG), which are readable from the HASH1 and HASH2 register locations. The HASH data is generated from the contents of the IMISS or DMISS register. The register selected depends on whether an instruction or data miss was acknowledged last.
- Automatic generation of the first word of the page table entry (PTE) for which the tables are being searched
- A required physical address (RPA) register that matches the format of the lower word of the PTE
- Two 602-specific TLB access instructions (**tlbli** and **tlbld**) that are used to load an address translation into the instruction or data TLBs
- Shadow registers for GPR0–GPR3 that allow missed code to execute without corrupting the state of any of the GPRs. These shadow registers are used only for servicing a TLB miss.

For details about the architecturally-defined translation and protection mechanism, see *The Programming Environments Manual*.

See Section 1.2.6.2, "PowerPC 602 Microprocessor Memory Management," for more information about memory management for the 602.

1.1.5.2 Cache Units

The 602 provides independent 4-Kbyte, two-way set-associative instruction and data caches. The cache block size is 32 bytes (eight words). The caches adhere to a write-back policy, but, as defined by the PowerPC architecture, the 602 allows control of cacheability, write policy, and memory coherency at the page and memory block levels. The caches use an LRU replacement policy.

As shown in Figure 1-1, the caches provide a 32-bit interface to the instruction fetch unit and load/store unit. The surrounding logic selects, organizes, and forwards the requested information to the requesting unit. Store operations to the cache can be performed on a byte basis, and a complete read-modify-write operation to the cache can occur in each cycle.

The load/store unit and instruction fetch unit provide the caches with the address of the data or instruction to be fetched. In the case of a cache hit, the cache returns two words to the requesting unit.

Since the 602 data cache tags are single-ported, simultaneous load or store and snoop accesses cause resource contention. Snoop accesses have the highest priority and are given first access to the tags, unless the snoop access coincides with a tag write. In this case the snoop is retried and must rearbitrate for access to the cache. Load or store operations that are deferred due to snoop accesses are executed on the clock cycle following the snoop.

1.1.6 Processor Bus Interface

The 602 bus interface is a time-multiplexed, 32-bit address, 64-bit data interface. Data transfers consist of two phases—the address phase, during which the address and transfer attributes are broadcast on the bus, and the data phase, during which the bus can function as either a 32-bit or 64-bit data bus. The address phase consists of subphases that are required for the processor to arbitrate and be granted mastership of the bus as well as the actual address transfer itself. The data phase consists of subphases that include the actual data transfer as well as an acknowledgment that each beat of data has been transferred successfully.

The 602 on-chip caches can be configured in the write-through or write-back modes. In the write-back mode, the predominant type of transaction for most applications is burst-read memory operations, followed by burst-write memory operations and single-beat (noncacheable or write-through) operations. Additionally, there can be address-only operations, variants of the burst and nonburst operations, (for example, global memory operations that are snooped and atomic memory operations), and address retry activity (for example, when a snooped read access hits a modified block in the cache).

Access to the system interface is granted through an external arbitration mechanism that allows devices to compete for bus mastership. This arbitration mechanism allows the 602 to be integrated into systems that implement various fairness and bus parking procedures to avoid arbitration overhead.

The amount of data that can be transferred per bus clock cycle depends on whether the bus is configured as a 32- or 64-bit bus. In 64-bit mode, the bus can transfer up to 64 bits in a single-beat (nonburst) transaction and an entire 8-word cache block in a four-beat burst transaction. In 32-bit mode, there are two types of nonburst transactions. A single-beat transaction transfers up to 32 bits and a double-beat transaction transfers 64 bits. In 32-bit mode, the burst transaction requires eight beats to transfer a cache block of data.

Typically, 602 memory accesses are weakly-ordered—sequences of operations, including load/store multiple instructions, do not necessarily complete in the order they begin. This maximizes the efficiency of the bus without sacrificing data integrity. The 602 allows read operations to precede store operations (except when a dependency exists). Because the processor can dynamically optimize run-time ordering of load/store traffic, overall performance is improved.

1.1.7 System Support Functions

The 602 implements several support functions that include power management, time base/ decrementer registers for system timing tasks, a watchdog timer, an IEEE 1149.1(JTAG)/ common on-chip processor (COP) test interface, and a phase-locked loop (PLL) clock multiplier. These system support functions are described in the following subsections.

1.1.7.1 Power Management

The 602 provides four power modes selectable by setting the appropriate control bits in the machine state register (MSR) and hardware implementation register 0 (HID0) registers. The four power modes are as follows:

- Full-power–This is the default power state of the 602. The 602 is fully powered and the internal functional units are operating at the full processor clock speed. If the dynamic power management mode is enabled, functional units that are idle automatically enters a low-power state without affecting performance, software execution, or external hardware.

- Doze–All the functional units of the 602 are disabled except for the time base/ decrementer registers and the bus snooping logic. The 602 returns to the full-power state upon the occurrence of any asynchronous exception. Asynchronous exceptions implemented on the 602 are listed in Table 1-1. The 602 in doze mode maintains the PLL in a fully-powered state and locked to the system external clock input (SYSCLK) so a transition to the full-power state takes only a few processor clock cycles.

- Nap–The nap mode further reduces power consumption by disabling bus snooping, leaving only the time base register and the PLL in a powered state. The 602 returns to the full-power state upon the occurrence of any asynchronous exception. Asynchronous exceptions implemented on the 602 are listed in Table 1-1. A return to full-power state from a nap state takes only a few processor clock cycles.

- Sleep–Sleep mode reduces power consumption to a minimum by disabling all internal functional units, after which external system logic may disable the PLL and SYSCLK. Returning the 602 to the full-power state requires the enabling of the PLL and SYSCLK, followed by any external exception after the time required to relock the PLL.

1.1.7.2 Time Base/Decrementer

The time base (TB) is a 64-bit register (accessed as two 32-bit registers) that is incremented once every four bus clock cycles; external control of the time base is provided through the time base enable (TBEN) signal. The decrementer is a 32-bit register that generates a decrementer exception after a programmable delay. The contents of the decrementer register are decremented once every four bus clock cycles, and the decrementer exception condition is generated as the count passes through zero.

1.1.7.3 IEEE 1149.1 (JTAG)/Common On-Chip Processor (COP) Test Interface

The 602 provides IEEE 1149.1 and COP functions for facilitating board testing and chip debug. The test interface provides a means for boundary-scan testing the 602 and the board to which it is attached. The COP function shares the IEEE 1149.1 test port, provides a means for executing test routines, and facilitates chip and software debugging.

1.1.7.4 Clock Multiplier

The internal clocking of the 602 is generated from and synchronized to the external clock signal, SYSCLK, by means of a voltage-controlled, oscillator-based PLL. The PLL provides programmable internal processor clock rates of either two or three times the externally supplied clock frequency. The bus clock is the same frequency and is synchronous with SYSCLK.

1.1.7.5 Watchdog Timer

The 602-specific watchdog timer can be used to generate a periodic exception based on the operation of the time base register. The watchdog timer is enabled and programmed through the timer control register (TCR), which is a supervisor-level SPR specific to the 602. Supervisor-level software can set bits in the TCR to select one of four time periods for the interrupts, and other aspects of the watchdog timer operations. When a watchdog timer exception occurs, instruction fetching begins at vector offset 0x1500 (as shown in Table 1-2). The watchdog timer can be programmed such that if the exception handler does not reset the timer, a second watchdog timer interrupt condition will cause a soft reset (system reset exception).

1.2 PowerPC 602 Microprocessor: Implementation

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 PowerPC architecture design facilitates parallel instruction execution and is scalable to take advantage of future technological gains.

This section describes the PowerPC architecture in general, and gives specific details about the implementation of the 602 as a low-power, 32-bit member of the PowerPC processor family.

- Features—Section 1.2.1, "Features," describes general features that the 602 shares with the PowerPC microprocessor family.

- Registers and programming model—Section 1.2.2, "PowerPC Registers and Programming Model," describes the registers for the operating environment architecture common among PowerPC processors and describes the programming model. It also describes the additional registers that are unique to the 602.

- Instruction set and addressing modes—Section 1.2.3, "Instruction Set and Addressing Modes," describes the PowerPC instruction set and addressing modes for the PowerPC operating environment architecture, and defines and describes the PowerPC instructions implemented in the 602.

- Cache implementation—Section 1.2.4, "Cache Implementation," describes the cache model that is defined generally for PowerPC processors by the virtual environment architecture. It also provides specific details about the 602 cache implementation.

- Exception model—Section 1.2.5, "Exception Model," describes the exception model of the PowerPC operating environment architecture and the differences in the 602 exception model.

- Memory management—Section 1.2.6, "Memory Management," describes generally the conventions for memory management among the PowerPC processors. This section also describes the 602's implementation of the 32-bit PowerPC memory management specification.

- Instruction timing—Section 1.2.7, "Instruction Timing," provides a general description of the instruction timing provided by the parallel execution supported by the PowerPC architecture and the 602.

- System interface—Section 1.2.8, "System Interface," describes the signals implemented on the 602.

1.2.1 Features

The 602 is a high-performance, low-cost microprocessor for consumer electronics and computers. It is designed for use in advanced home entertainment and educational devices with audio/video, multimedia, and complex graphics requirements. The 602 is also applicable for low-power business and commercial devices with speech recognition and synthesis, wireless communications, or handwriting recognition.

The following sections summarize the features of the 602, including both those that are defined by the architecture and those that are unique to the 602 implementation.

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:

- PowerPC user instruction set architecture (UISA)—Defines the base user-level instruction set, user-level registers, data types, floating-point exception model, memory models for a uniprocessor environment, and programming model for a uniprocessor environment.

- PowerPC virtual environment architecture (VEA)—Describes the memory model for a multiprocessor environment, defines cache control instructions, and describes other aspects of virtual environments. Implementations that conform to the VEA also adhere to the UISA, but may not necessarily adhere to the OEA.

- PowerPC operating environment architecture (OEA)—Defines the memory management model, supervisor-level registers, synchronization requirements, and the exception model. Implementations that conform to the OEA also adhere to the UISA and the VEA.

The 602 does not implement the double-precision floating-point instructions and the load/ store string instructions in hardware. Barring these exceptions, the 602 implements the levels of architecture as mentioned above. Specific features of the 602 are listed in Section 1.1.1, "PowerPC 602 Microprocessor Features."

For more information, see *The Programming Environments Manual*.

1.2.2 PowerPC 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.

PowerPC processors have two levels of privilege—supervisor level (typically used by the operating system) and user level (used by the application software). The programming models incorporate 32 GPRs, 32 FPRs, special-purpose registers (SPRs), and several miscellaneous registers. Each PowerPC microprocessor may also have its own unique set of implementation-specific registers. The registers implemented in the 602 are shown in

Figure 1-3 and described in the following sections. Registers defined by the PowerPC architecture are described in *The Programming Environments Manual*.

Access to supervisor-level instructions, registers, and other resources 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 operating in supervisor mode.

Figure 1-3 shows the registers implemented in the 602 and indicates whether these registers are accessible to user- or supervisor-level software. 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.

SUPERVISOR MODEL

Configuration Registers

PLL Configuration Register[1]	Machine State Register	Checkstop Enables Register[1]	Processor Version Register
HID1 — SPR 1009	MSR	HID0 — SPR 1008	PVR — SPR 287

USER MODEL

General-Purpose Registers

GPR0
GPR1
⋮
GPR31

Single-Precision Floating-Point Registers

FPR0
FPR1
⋮
FPR31

Condition Register

CR

Floating-Point Status and Control Register

FPSCR

XER

XER — SPR 1

Link Register

LR — SPR 8

Count Register

CTR — SPR 9

Time Base Facility (For Reading)

TBL — TBR 268
TBU — TBR 269

Memory Management Registers

Instruction BAT Registers

IBAT0U	SPR 528
IBAT0L	SPR 529
IBAT1U	SPR 530
IBAT1L	SPR 531
IBAT2U	SPR 532
IBAT2L	SPR 533
IBAT3U	SPR 534
IBAT3L	SPR 535

Data BAT Registers

DBAT0U	SPR 536
DBAT0L	SPR 537
DBAT1U	SPR 538
DBAT1L	SPR 539
DBAT2U	SPR 540
DBAT2L	SPR 541
DBAT3U	SPR 542
DBAT3L	SPR 543

Software Table Search Registers[1]

DMISS	SPR 976
DCMP	SPR 977
HASH1	SPR 978
HASH2	SPR 979
IMISS	SPR 980
ICMP	SPR 981
RPA	SPR 982

SDR1 — SPR 25

Segment Registers

SR0
SR1
⋮
SR15

ESA Access Registers

ESA Enable Register[1]

SER — SPR 991

ESA Enable Base Register1

SEBR — SPR 990

ESA Save/Restore Register[1]

ESASRR — SPR 987

Exception Handling Registers

Data Address Register

DAR — SPR 19

DSISR

DSISR — SPR 18

SPRGs

SPRG0	SPR 272
SPRG1	SPR 273
SPRG2	SPR 274
SPRG3	SPR 275

Save and Restore Registers

| SRR0 | SPR 26 |
| SRR1 | SPR 27 |

Miscellaneous Registers

Time Base Facility (For Writing)

| TBL | SPR 284 |
| TBU | SPR 285 |

Instruction Address Breakpoint Register[1]

IABR — SPR 1010

Single-Precision Tag Register[1]

SP — SPR 1021

Integer Tag Register[1]

LT — SPR 1022

Decrementer

DEC — SPR 22

Timer Control Register[1]

TCR — SPR 984

Interrupt Base Register[1]

IBR — SPR 986

[1] These registers are 602–specific registers. They may not be supported by other PowerPC processors.

Figure 1-3. PowerPC 602 Microprocessor Programming Model—Registers

The following sections summarize the PowerPC registers that are implemented in the 602.

1.2.2.1 General-Purpose Registers (GPRs)

The PowerPC architecture defines 32 user-level, general-purpose registers (GPRs) that serve as the data source or destination for all integer instructions. The 602 also implements four GPR rename registers, which are used to make the results of executing an instruction available to subsequent instructions before they are written back to the architected GPR (during the complete/write-back stage).

The 602 also defines four GPR shadow registers to support the software table search operations. MSR[TGPR] is set whenever the 602 takes an instruction TLB miss, data read miss, or data write miss exception. When MSR[TGPR] is set, all instruction accesses to GPR0–GPR3 are to be mapped to TGPR0–TGPR3. The contents of GPR0–GPR3 remain unchanged while MSR[TGPR] is set. Attempts to use GPR4–GPR31 with MSR[TGPR] set yields undefined results. MSR[TGPR] is cleared when an **rfi** instruction is executed.

1.2.2.2 Floating-Point Registers (FPRs)

The UISA portion of the PowerPC architecture defines 32 user-level, 64-bit floating-point registers (FPRs). The FPRs serve as the data source or destination for floating-point instructions. These registers can contain data objects of either single- or double-precision floating-point formats. However, because the 602 is optimized for systems that perform single- and not double-precision floating-point arithmetic, the 602 implements 32 user-level, 32-bit FPRs. Double-precision arithmetic instructions and operations that employ double-precision operands that cannot be represented in single-precision are trapped for emulation in software. Single-precision operations can execute instructions using double-precision operands without taking an exception as long as the double-precision operand can be represented as a single-precision value without loss of accuracy.

The 602 also implements four FPR rename registers, which are used to make the results of executing a floating-point instruction available to subsequent instructions before those results are written back to the architected FPR (during the complete/write-back stage).

1.2.2.3 Condition Register (CR)

The CR is a 32-bit, user-level register that consists of eight 4-bit fields that reflect the results of certain operations, such as move, integer and floating-point compare, arithmetic, and logical instructions, and provide a mechanism for testing and branching.

1.2.2.4 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.

1.2.2.5 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 typically restored when the exception handling completes. The 602 implements the MSR as a 32-bit register. The 602 defines additional bits in the MSR to support 602-specific functionality. For example, MSR[SA] indicates whether the **esa** instruction has been executed to put the processor in supervisor mode. MSR[AP] controls whether supervisor-level software can access memory at user level or at supervisor level.

1.2.2.6 Segment Registers (SRs)

For memory management, 32-bit PowerPC microprocessors implement sixteen 32-bit segment registers (SRs). To speed access, the 602 implements the segment registers as two arrays—a main array (for data memory accesses) and a shadow array (for instruction memory accesses). Loading a segment entry with the Move to Segment Register (**mtsr**) instruction loads both arrays.

1.2.2.7 Special-Purpose Registers (SPRs)

The PowerPC operating environment architecture defines numerous SPRs that serve a variety of functions, such as providing controls, indicating status, configuring the processor, and performing special operations. During normal execution, a program can access the registers, shown in Figure 1-3, depending on the program's access privilege (supervisor or user, determined by the privilege level (PR) bit in the MSR). Note that registers such as the GPRs and 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 implicit, as the part of the execution of an instruction. Some registers are accessed both explicitly and implicitly.

1.2.2.7.1 User-Level SPRs

The following SPRs are accessible by user-level software:

- Link register (LR)—The 32-bit link register can be used to provide the branch target address and to hold the return address after branch and link instructions.

- Count register (CTR)—The 32-bit CTR is decremented and tested automatically as a result of branch and count instructions.

- The XER register—The 32-bit XER contains the summary overflow bit, integer carry bit, and overflow bit.

1.2.2.7.2 Supervisor-Level SPRs

The 602 contains SPRs that can be accessed only by supervisor-level software. See Figure 1-3 for a list of the SPR numbers. The 602 implements both those supervisor-level SPRs defined by the PowerPC architecture as well as several additional SPRs required for supporting 602-specific features such as the software table search, the watchdog timer, and the **esa** supervisor access.

The following registers are defined by the PowerPC architecture, although some of these registers implement additional bits to support 602 functionality:

- The 32-bit DSISR defines the cause of data access 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.

- Decrementer register (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay.

- The 32-bit SDR1 specifies the page table format used in virtual-to-physical address translation for pages. (Note that physical address is referred to as real address in the architecture specification.)

- The machine status save/restore register 0 (SRR0) is a 32-bit register that is used by the 602 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. The 602 defines additional bits in the MSR to support 602-specific functionality.

- The 32-bit SPRG0–SPRG3 registers are provided for operating system use.

- 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 processor version register (PVR) is a 32-bit, read-only register that identifies the version (model) and revision level of the PowerPC processor.

- 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 IBAT registers implement two additional bits—the NE bit controls whether instructions can be fetched from the block and the SE bit controls whether an **esa** instruction fetched from the block can be executed.

The following supervisor-level SPRs are 602-specific:

- The hardware implementation-dependent register 0 (HID0) provides means for enabling the 602's checkstops and features, such as protection-only mode.

- Hardware implementation-dependent register 1 (HID1) is a read-only register that stores the PLL configuration bits.

- The DMISS and IMISS registers are read-only registers that are loaded automatically upon an instruction or data TLB miss.

- The HASH1 and HASH2 registers contain the physical addresses of the primary and secondary page table entry groups (PTEGs).

- The ICMP and DCMP registers contain a duplicate of the first word in the page table entry (PTE) for which the table search is looking.

- The required physical address (RPA) register is loaded by the processor with the second word of the correct PTE during a page table search.
- The instruction address breakpoint register (IABR) is loaded with an instruction address that is compared to instruction addresses in the dispatch queue. When an address match occurs, an instruction address breakpoint exception is generated.
- The ESA enable register (SER) is a 32-bit register used in the protection-only mode. Each bit controls **esa** execution privileges for a 4-Kbyte page.
- The ESA enable base register (SEBR) contains the base address of the 128-Kbyte region that is protected by the special execute (SE) bits in SER.
- The ESA save and restore register (ESASRR) is a 32-bit register that saves selected MSR bits (PR, AP, SA, EE) when the **esa** instruction is executed. Executing the **dsa** instruction restores these bits to the MSR.
- The single-precision tag register (SP) is a 32-bit register for which each bit (SP0–SP31) corresponds to one of the 32, 32-bit FPRs (FPR0–FPR31). An SP bit is set if the corresponding FPR holds a single-precision value.
- The integer tag register (LT) is a 32-bit register for which each bit (LT0–LT31) corresponds to one of the 32, 32-bit FPRs (FPR0–FPR31). An LT bit is set if the corresponding FPR holds an integer value.
- The timer control register (TCR) provides bits for enabling and programming the watchdog timer.
- The interrupt base register (IBR) contains an exception vector offset address used by exception handlers if the MSR[IP] is cleared when an exception occurs; if MSR[IP] is set, the default offset address 0xFFF00000 is used.

1.2.3 Instruction Set and Addressing Modes

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

1.2.3.1 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.

1.2.3.1.1 PowerPC Instruction Set

The PowerPC instructions are divided into the following categories:

- Integer instructions—These include computational and logical instructions.
  - Integer arithmetic instructions
  - Integer compare instructions
  - Integer 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 arithmetic instructions
  - — Floating-point multiply/add instructions
  - — Floating-point rounding and conversion instructions
  - — Floating-point compare instructions
  - — Floating-point status and control instructions
- Load/store instructions—These include integer and floating-point load and store instructions.
  - — Integer load and store instructions
  - — Integer load and store multiple instructions
  - — Floating-point load and store instructions
  - — Primitives used to construct atomic memory operations (**lwarx** and **stwcx.** instructions)
- 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
  - — Condition register logical instructions
- Processor control instructions—These instructions are used for synchronizing memory accesses and management of caches, TLBs, and the segment registers.
  - — Move to/from SPR instructions
  - — Move to/from MSR instructions
  - — Synchronize instruction
  - — Instruction synchronize
- Memory control instructions—These instructions provide control of caches, TLBs, and segment registers.
  - — Supervisor-level cache management instructions
  - — User-level cache instructions
  - — Segment register manipulation instructions
  - — Translation lookaside buffer management instructions

Note that this grouping of the instructions does not indicate the execution unit that 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) operands. Floating-point double-precision operations are trapped for emulation in software. The PowerPC architecture uses instructions that are four bytes long and word-aligned. It provides for byte, half-word, and

word operand load and store operations between memory and 32 GPRs. It also provides for word and double-word operand loads and stores between memory and 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 distinct 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 causing an exception, or by an asynchronous event. Either kind of exception may cause one of several components of the system software to be invoked.

1.2.3.1.2 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 memory 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 in 32-bit implementations.

1.2.3.2 PowerPC 602 Microprocessor Instruction Set

The 602 instruction set is defined as follows:

- The 602 provides hardware support for most 32-bit PowerPC instructions. The 602 does not support double-precision floating-point, load/store string, **eciwx**, and **ecowx** instructions in hardware.
- The 602 provides two implementation-specific instructions used for software table search operations following TLB misses:
    - TLB Load Data (**tlbld**)
    - TLB Load Instruction (**tlbli**)

- The 602 provides two implementation-specific instructions that enable/disable access into the supervisor mode (without having to branch or invoking a system call):
  – Enable Supervisor Access (**esa**) instruction. When the **esa** instruction is executed, the processor enters supervisor mode without incurring the additional latency due to synchronizing the processor and refetching from the exception vector. The ability to execute the **esa** instruction on a block or page is controlled by the SE bit, which is configured through the IBATs and the ITLBs. This functionality is not available in real addressing mode.
  – Disable Supervisor Access (**dsa**) instruction. Executing the **dsa** instruction restores the processor to the state it was in prior to the execution of the **esa** instruction.
- The 602 implements the following instructions that are defined as optional by the PowerPC architecture:
  – Floating Select (**fsel**) instruction
  – Floating Reciprocal Estimate Single-Precision (**fres**) instruction. On the 602, **fres** is implemented as a divide rather than an estimate.
  – Floating Convert to Integer Word with Round toward Zero (**fctiwz**) instruction

1.2.4 Cache Implementation

The following subsections describe the PowerPC architecture's treatment of cache in general, and the 602-specific implementation, respectively.

1.2.4.1 PowerPC Cache Characteristics

The PowerPC architecture does not define hardware aspects of cache implementations. For example, some PowerPC processors, including the 602, have separate instruction and data caches (Harvard architecture), while others, such as the PowerPC 601™ microprocessor, implement a unified cache.

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

- Write-back/write-through mode
- Caching-inhibited mode
- Memory coherency

Note that in the 602, a cache block is defined as being eight words wide. The VEA defines cache management instructions that application programmers can use to affect the cache contents and coherency state.

1.2.4.2 PowerPC 602 Microprocessor Cache Implementation

The 602 has two 4-Kbyte, two-way set-associative (instruction and data) caches. The caches are physically addressed, and the data cache can operate in either write-back or write-through mode as specified by the PowerPC architecture. Cache organization is shown in Figure 1-4.
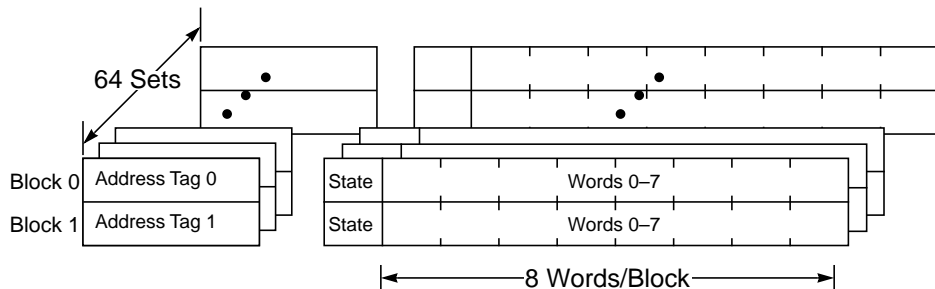
Figure 1-4. Cache Organization

The data cache is configured as 64 sets of two cache blocks each. Each cache block consists of eight 32-bit words, two state bits, and an address tag. The two state bits implement the three-state MEI (modified/exclusive/invalid) protocol. Note that the PowerPC architecture defines the term block as the cacheable unit. For the 602, the block size is equivalent to a cache block.

The instruction cache differs in that it maintains only one state bit that indicates only whether the data is valid, because the instruction caches do not support the MEI coherency protocol. Because the instruction cache may not be written to except through a line-fill operation it is not snooped, and cache coherency must be maintained by software. A fast hardware invalidation capability is provided to support cache maintenance.

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); thus, a cache block never crosses a page boundary. Accesses that cross a page boundary can incur a performance penalty.

The 602's cache blocks are loaded in four beats of 64 bits each when the bus is in 64-bit mode (and in eight 32-bit beats in 32-bit mode). The burst load is performed as critical double word first and the requested data in that double word is forwarded to the requesting processor unit as an instruction or as an operand. The caches are nonblocking, so additional data or instructions can be accessed by the requesting unit as soon as it arrives in the cache.

To ensure coherency among caches in a multiprocessor (or multiple caching-device) implementation, the 602 implements the MEI protocol in the data caches. These three states, modified, exclusive, and invalid, indicate the state of the cache block as follows:

- Modified—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—This cache block holds valid data that is identical to the data at this address in system memory. No other device has this data.

- Invalid—This cache block does not hold valid data.

Cache coherency is enforced by on-chip bus snooping logic. Since the 602's data cache tags are single-ported, a simultaneous load or store and snoop access represent a resource contention. The snoop access is given first access to the tags. The load or store then occurs on the clock following the snoop. All read operations on the bus are treated as read-with-intent-to-modify operations, as are all burst read operations from a snooping perspective. Injected snooping, that is snooping between beats in a burst read operation, provides an additional snooping opportunity.

1.2.5 Exception Model

The following subsections describe the PowerPC exception model and the 602 implementation, respectively.

1.2.5.1 PowerPC Exception Model

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, and differ from the arithmetic exceptions defined by the IEEE for floating-point operations. 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 occurs 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 FPSCR. Additionally, some exception conditions can be explicitly enabled or disabled by software.

The PowerPC architecture requires that a processor be able to handle instruction-caused exceptions in program order; therefore, although a particular implementation may recognize exception conditions out of order, they are presented strictly in order. When an instruction-caused exception is recognized, any unexecuted instructions that appear in the instruction stream prior to the instruction causing the interrupt are required to complete before the exception is taken. Any exceptions caused by those instructions are handled first. Likewise, exceptions that are asynchronous and precise are recognized when they occur, but are not handled until the instruction currently in the completion stage successfully completes execution or generates an exception, and the completed store queue is emptied.

Unless a catastrophic condition causes a system reset or machine check exception, only one exception is handled at a time. If, for example, an instruction encounters multiple exception conditions, those conditions are processed sequentially. After the exception handler handles an exception, the instruction execution continues until the next exception condition is encountered. However, in many cases, there is no attempt to re-execute the instruction. 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 and machine check exception or to an instruction-caused exception in the exception handler, and before enabling external interrupts.

The PowerPC architecture supports four types of exceptions:

- Synchronous, precise—These are caused by instructions. All instruction-caused exceptions are handled precisely; that is, the machine state at the time the exception occurs is known and can be completely restored. This means that (excluding the trap and system call exceptions) the address of the faulting instruction is provided to the exception handler and that neither the faulting instruction nor subsequent instructions in the code stream completes execution before the exception is taken. Once the exception is processed, execution resumes at the address of the faulting instruction (or at an alternate address provided by the exception handler). When an exception is taken due to a trap or system call instruction, execution resumes at an address provided by the handler.

- Synchronous, imprecise—The PowerPC architecture defines two imprecise floating-point exception modes, recoverable and nonrecoverable. Even though the 602 provides a means to enable the imprecise modes, it implements these modes identically to the precise mode (that is, all enabled floating-point exceptions are always precise on the 602).

- Asynchronous, maskable—The external, system management, and decrementer interrupts are maskable asynchronous exceptions. When these exceptions occur, their handling is postponed until the next instruction, and any exceptions associated with that instruction, completes execution. If there are no instructions in the execution units, the exception is taken immediately upon determination of the correct restart address (for loading SRR0).

- Asynchronous, nonmaskable—There are two nonmaskable asynchronous exceptions, system reset and the machine check exception. These exceptions may not be recoverable, or may provide a limited degree of recoverability. All exceptions report recoverability through the MSR[RI] bit.

1.2.5.2 PowerPC 602 Microprocessor Exception Model

As specified by the PowerPC architecture, all 602 exceptions can be described as either precise or imprecise and either synchronous or asynchronous. Asynchronous exceptions (some of which are maskable) are caused by events external to the processor's execution; synchronous exceptions, which are all handled precisely by the 602, are caused by instructions. The 602 exception classes are shown in Table 1-1.

Table 1-1. PowerPC 602 Microprocessor Exception Classifications

Synchronous/Asynchronous	Precise/Imprecise	Exception Type
Asynchronous, nonmaskable	Imprecise	Machine check System reset
Asynchronous, maskable	Precise	External interrupt Decrementer interrupt System management interrupt Watchdog timer interrupt
Synchronous	Precise	Instruction-caused exceptions

Although exceptions have other characteristics as well, such as whether they are maskable or nonmaskable, the distinctions shown in Table 1-1 define categories of exceptions that the 602 handles uniquely. Note that Table 1-1 includes no synchronous imprecise instructions. Although the PowerPC architecture supports imprecise handling of floating-point exceptions, the 602 implements these exception modes as precise exceptions.

The 602's exceptions, and conditions that cause them, are listed in Table 1-2. The vector column indicates how to determine the vector address from the vector offset and a combination of the setting of MSR[IP] and the contents of the interrupt base register (IBR).

Exceptions that are specific to the 602 are indicated.

Table 1-2. Exceptions and Conditions

Exception Type	Vector (hexadecimal) Prefix IP = 0	Vector (hexadecimal) Prefix IP = 1	Offset	Causing Conditions
Reserved	—	—	0000	—
System reset (Hard reset)	FFF0	FFF0	0100	Assertion of $\overline{\text{HRESET}}$
System reset (Soft reset)	0000	FFF0	0100	Assertion of $\overline{\text{SRESET}}$
Machine check	0000	FFF0	0200	Assertion of $\overline{\text{TEA}}$ during a data transaction; assertion of $\overline{\text{MCP}}$.

Table 1-2. Exceptions and Conditions (Continued)

Exception Type	Vector (hexadecimal)			Causing Conditions
	Prefix		Offset	
	IP = 0	IP = 1		
DSI	IBR	FFF0	0300	Determined by the bit settings in the DSISR, as follows: 4 Set if a memory access is not permitted by the page or DBAT protection mechanism; otherwise cleared. 5 Set if memory access is attempted and SR[T] = 1; otherwise cleared. The 602 does not support direct-store memory. 6 Set for a store operation and cleared for a load operation.
ISI	IBR	FFF0	0400	An instruction cannot be fetched for one of the following reasons: • The EA cannot be translated and an ISI exception must be taken to load the PTE (and possibly the page) into memory. • The fetch access violates memory protection. If SR[Ks] and SR[Kp] and PTE[PP] are set to prohibit read access, instructions cannot be fetched from this location.
External interrupt	IBR	FFF0	0500	MSR[EE] = 1 and the $\overline{INT}$ signal is asserted.
Alignment	IBR	FFF0	0600	Memory cannot be accessed for one of the following reasons: • The operand of a floating-point load or store is not word-aligned. • The operand of **lmw**, **stmw**, **lwarx**, or **stwcx.** is not word-aligned. • The operand of **dcbz** is in a page marked write-through or caching-inhibited, for a virtual mode access. • A little-endian access is misaligned, or a multiple access is attempted with the little-endian bit set.
Program	IBR	FFF0	0700	The following conditions correspond to bit settings in SRR1 and arise during execution of an instruction: • Floating-point enabled exception—The following is met: (MSR[FE0] | MSR[FE1]) & FPSCR[FEX] is 1. FPSCR[FEX] is set by a floating-point instruction that causes an enabled exception or by the execution of one of the "move to FPSCR" instructions that results in both an exception condition bit and its corresponding enable bit being set in the FPSCR. • Illegal instruction—Execution of an instruction is attempted with an illegal opcode or combination of opcode and extended opcode (including PowerPC instructions not implemented in the 602 but not including those optional instructions treated as no-ops). • Privileged instruction—Execution of a privileged instruction is attempted and MSR[PR] = 1. In the 602, this exception is generated for **mtspr** or **mfspr** with an invalid SPR field if SPR[0] = 1 and MSR[PR] = 1. This may not be true for all PowerPC processors. • Trap— Generated when a trap instruction condition is met.
Floating-point unavailable	IBR	FFF0	0800	An attempt to execute a floating-point instruction (including floating-point load, store, or move instructions) when the floating-point available bit is disabled, (MSR[FP] = 0)

Table 1-2. Exceptions and Conditions (Continued)

Exception Type	Vector (hexadecimal)			Causing Conditions
	Prefix		Offset	
	IP = 0	IP = 1		
Decrementer	IBR	FFF0	0900	The most significant bit of the decrementer (DEC) register changes from 0 to 1. Must be enabled with the MSR[EE] bit.
Reserved	IBR	FFF0	0A00–0BFF	—
System call	IBR	FFF0	0C00	Execution of the System Call (**sc**) instruction
Trace	IBR	FFF0	0D00	MSR[SE] =1 or when a completing instruction is a branch and MSR[BE] =1
Floating-point assist	IBR	FFF0	0E00	Not implemented in the 602
Reserved	—	—	0E10–0FFF	—
Instruction translation miss	IBR	FFF0	1000	The ITLB cannot translate the EA for an instruction fetch.
Data load translation miss	IBR	FFF0	1100	An EA for a data load cannot be translated by the DTLB.
Data store translation miss	IBR	FFF0	1200	An EA for a data store cannot be translated by the DTLB; or when a DTLB hit occurs and the change bit in the PTE must be set due to a data store operation.
Instruction address breakpoint	0000	FFF0	1300	The address (bits 0–29) in the instruction address breakpoint register (IABR) matches the next instruction to complete in the completion unit and the IABR enable bit (bit 30) is set.
System management interrupt	IBR	FFF0	1400	MSR[EE] = 1 and the $\overline{SMI}$ input signal is asserted.
Watchdog timer	IBR	FFF0	1500	A carry occurs out of a bit specified by the user. If the watchdog timer is not reset by the interrupt service routine, a second watchdog timer exception forces an internal reset.
Emulation trap	IBR	FFF0	1600	Either a double-precision floating-point instruction or a load/store string instruction is encountered.
Reserved	—	—	1700–2FFF	—

1.2.6 Memory Management

The following subsections describe the memory management features of the PowerPC architecture, and the 602 implementation, respectively.

1.2.6.1 PowerPC Memory Management

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

There are two types of accesses generated by the 602 that require address translation—instruction accesses and data accesses to memory generated by load and store instructions.

The PowerPC MMU 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.

The page table contains a number of page table entry groups (PTEGs). A PTEG contains eight page table entries (PTEs) of 8 bytes each; therefore, each PTEG is 64 bytes long. PTEG addresses are entry points for table search operations.

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

1.2.6.2 PowerPC 602 Microprocessor Memory Management

Instruction and data TLBs provide address translation in parallel with the on-chip cache access, incurring no additional time penalty in the event of a TLB hit. A TLB is a cache of the most recently used page table entries. Software is responsible for maintaining the consistency of the TLB with memory. The 602's TLBs are 32-entry, two-way set-associative caches that contain instruction and data address translations. The 602 provides hardware assist for software table search operations through the hashed page table on TLB misses. Supervisor software can invalidate TLB entries selectively.

The instruction and data memory management units provide 4 Gbytes of logical address space accessible to supervisor and user programs with a 4-Kbyte page size and 256-Mbyte segment size. Block sizes range from 128 Kbytes to 256 Mbytes and are software-selectable. In addition, the 602 uses an interim 52-bit virtual address and hashed page tables for generating 32-bit physical addresses. The MMUs in the 602 rely on the exception processing mechanism for the implementation of the paged virtual memory environment and for enforcing protection of designated memory areas.

As specified by the PowerPC architecture, 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.

Also as specified by the PowerPC architecture, the page table contains a number of page table entry groups (PTEGs). A PTEG contains eight page table entries (PTEs) of 8 bytes each; therefore, each PTEG is 64 bytes long. PTEG addresses are entry points for table search operations.

For applications requiring protection for areas of memory larger than 256 Kbytes, the 602 provides an optional configuration of its TLBs. Upon reset, the operating system can configure the TLBs in the protection-only mode, which is described in detail in Section 1.2.6.2.1, "Protection-Only Mode."

The 602 also provides independent four-entry BAT arrays for instructions and data that maintain address translations for blocks of memory. These entries define blocks that can vary from 128 Kbytes to 256 Mbytes. The BAT arrays are maintained by system software. An address match with an entry in the BATs takes priority over a hit in the TLBs for the same effective address.

1.2.6.2.1 Protection-Only Mode

Protection-only mode is provided for special-purpose implementations that do not require the more extensive paging functionality required for multipurpose personal computers, but need memory protection not offered by the OEA-defined real addressing mode.

Protection-only mode is described as follows:

- Protection-only mode is used when an effective address misses in the BAT registers and HID0[PO] is set.
- When HID0[PO] is set, the TLBs are configured differently than the configuration for the OEA-defined page address translation. In protection-only mode, each TLB is configured to provide protection for 32, 4-Kbyte pages per TLB entry. A total of 4 Mbytes of memory can be protected in each TLB at one time. Protection consists of 1 bit per 4-Kbyte page to control execution (NE bit) in instruction pages and control write access (WE bit) in the data pages.
- This protection is provided by the NE bit, which provides no-execute protection on a page level, the SE bit, which controls the use of **esa**/**dsa** supervisor access, and the WE bit, which controls whether memory can be written to on a page basis. In protection-only mode, these additional bits are defined in the TLBs and are propagated and managed through portions of the architecturally-defined page translation mechanism.
- Although the page translation mechanism is used to enforce memory protection, it is not used to determine the physical address (that is, the effective address is used as the physical address). In protection-only mode only the 24-bit virtual segment ID (VSID) in SR0 is used. This VSID also functions as a process ID in protection-only mode.
- Only the settings for the page from segment register 0 are used in this mode. Other entries can be written to, but are not used.

- The 602 provides programmable default cache control bits (WIMG) in the HID0 register to be used when the processor is running in real addressing mode or protection-only mode.

- The SEBR and SER registers control the execution of the 602-specific **esa** instruction for each of the 32, 4-Kbyte pages of a 128-Kbyte block of memory at any one time.

1.2.7 Instruction Timing

The 602 is a pipelined processor with parallel execution units. A pipelined processor is one in which the processing of an instruction is reduced into discrete stages. Because the processing of an instruction is broken into a series of stages, an instruction does not require the entire resources of an execution unit. For example, after an instruction completes the decode stage, it can pass on to the next stage, while the subsequent instruction can advance into the decode stage. This improves the throughput of the instruction flow. For example, it may take three cycles for a floating-point instruction to complete, but if there are no stalls in the floating-point pipeline, a series of floating-point instructions can have a throughput of one instruction per cycle.

The instruction pipeline in the 602 has four major pipeline stages, described as follows:

- The fetch pipeline stage primarily involves retrieving instructions from the memory system and determining the location of the next instruction fetch. Additionally, the BPU decodes branches during the fetch stage and folds out branch instructions before the dispatch stage if possible.

- The decode and dispatch pipeline stage is responsible for decoding the instructions supplied by the instruction fetch stage and determining which of the instructions are eligible to be dispatched in the current cycle. In addition, the source operands of the instructions are read from the appropriate register file and dispatched with the instruction to the execute pipeline stage. At the end of the dispatch pipeline stage, the dispatched instructions and their operands are latched by the appropriate execution unit.

- During the execute pipeline stage each execution unit that has an executable instruction executes the selected instruction (perhaps over multiple cycles), writes the instruction's result into the appropriate rename register, and notifies the completion stage that the instruction has finished execution. In the case of an internal exception, the execution unit reports the exception to the completion/write-back pipeline stage and discontinues instruction execution until the exception is handled. The exception is not signaled until that instruction is the next to be completed. Execution of most floating-point instructions is pipelined within the FPU allowing up to three instructions to be executing in the FPU concurrently. The pipeline stages for the floating-point unit are multiply, add, and round-convert. Execution of most load/store instructions is also pipelined. The load/store unit 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.

- The complete/write-back pipeline stage maintains the correct architectural machine state and transfers the contents of the rename registers to the GPRs and FPRs as instructions are retired. If the completion logic detects an instruction causing an exception, all following instructions are flushed, any execution results in rename registers are discarded, and instructions are fetched from the correct instruction stream.

The 602 has four independent execution units, one each for integer instructions, floating-point instructions, branch instructions, and load/store instructions. The IU and the FPU each have dedicated register files for maintaining operands (GPRs and FPRs, respectively), allowing integer calculations and floating-point calculations to occur simultaneously without interference.

Because the PowerPC architecture can be applied to such a wide variety of implementations, instruction timing among various PowerPC processors varies accordingly.

1.2.8 System Interface

The system interface is specific for each PowerPC microprocessor implementation.

The 602 interface includes a time-multiplexed, 32-bit address and 64-bit data bus, and 56 control and information signals (see Figure 1-5). The system interface allows for address-only transactions as well as address and data transactions. The 602 control and information signals include the bus arbitration, address start, address transfer, transfer attribute, address termination, data transfer, data termination, and processor state signals. Test and control signals provide diagnostics for selected internal circuits.
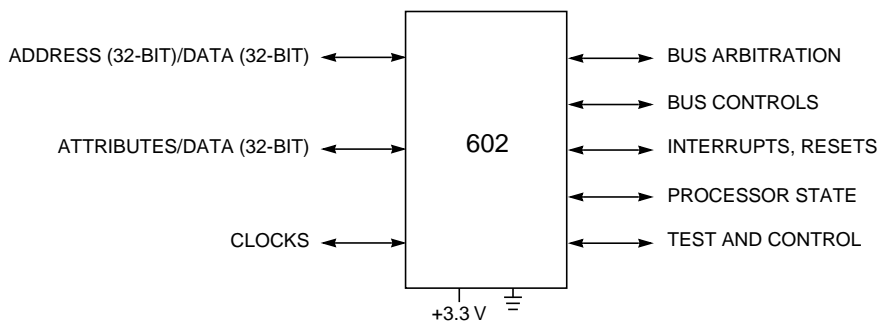
Figure 1-5. System Interface

The 602 supports multiple masters through a bus arbitration scheme that allows various devices to compete for the shared bus resource. The arbitration logic can implement priority protocols, such as fairness, and can park masters to avoid arbitration overhead. The MEI protocol ensures coherency among multiple devices and system memory. Also, the 602's on-chip caches and TLBs and optional second-level caches can be controlled externally.

The 602's clocking structure allows the bus to operate at integer multiples of the processor cycle time.

The following sections describe the 602 bus support for memory. Note that some signals perform different functions depending upon the addressing protocol used.

1.2.8.1 Memory Accesses

The 602 memory accesses allow transfer sizes of 8, 16, 24, 32, or 64 bits in one bus clock cycle. In 64-bit mode, data transfers occur in either single-beat transactions (up to 64 bits) or four-beat burst (8 words) transactions. In 64-bit mode, nonburst data transfers occur in either single- or double-beat (up to 32 or 64 bits, respectively) transactions or eight-beat burst transactions.

Nonburst transactions are caused by noncached accesses that access memory directly (that is, read and write operations when caching is disabled, caching-inhibited accesses, and stores in write-through mode). Four-beat burst transactions, which always transfer an entire 32-byte cache block, are initiated when a block is read from or written to memory. The 602 initiates two distinct bus transactions in cases of misaligned accesses.

1.2.8.2 PowerPC 602 Microprocessor Signals

The 602 signals are grouped as follows:

- Bus arbitration signals—The 602 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 consist of the address, and prefetch line-fill address buses.
- Transfer attribute signals—These signals provide information about the type of transfer, such as the transfer size and whether the transaction is a burst, write-through, or caching-inhibited transaction.
- 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 transfer signals—These signals consist of the 64-bit data bus.
- Data transfer termination signals—Data termination signals are required after each data beat in a data transfer. In a nonburst transaction, the data termination signals also indicate the end of the phase, while in burst accesses, the data termination signals apply to individual beats and indicate the end of the data phase only after the final data beat.
- 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 signals enable the time base and control the ability to put the 602 in quiescent mode.
- IEEE 1149.1(JTAG)/COP interface signals—The IEEE 1149.1 test unit and the common on-chip processor (COP) unit are accessed through a shared set of input, output, and clocking signals. The IEEE 1149.1/COP interface provides a means for boundary-scan testing and internal debugging of the 602.
- Test interface signals—These signals are used for production testing.
- 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, for example, TT0–TT4 (transfer type signals), are referred to as asserted when they are high and negated when they are low.

1.2.8.3 Signal Configuration

Figure 1-6 illustrates the 602's logical pin configuration, showing how the signals are grouped.
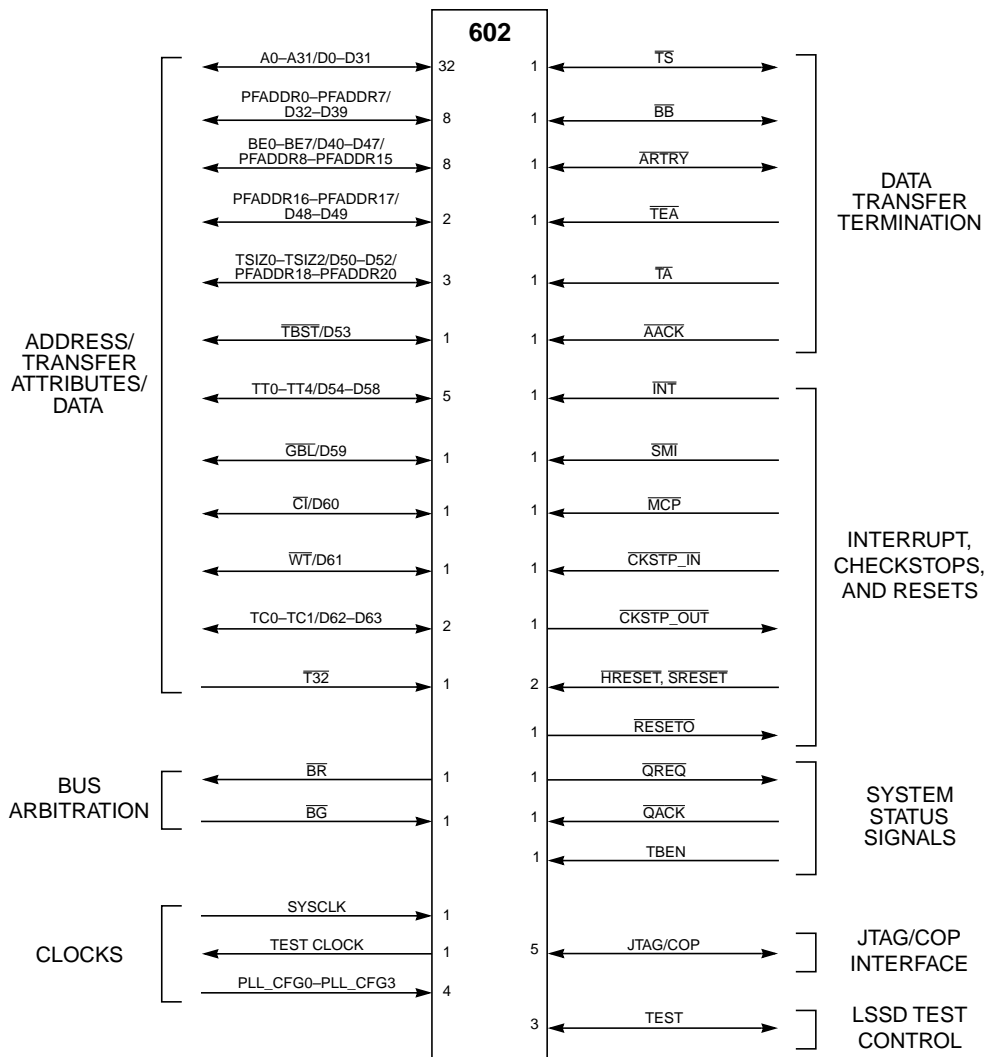
Figure 1-6. PowerPC 602 Microprocessor Signal Groups

PowerPC 602 RISC Microprocessor User's Manual

Chapter 2
PowerPC 602 Microprocessor Programming Model

This chapter describes the PowerPC programming model with respect to the PowerPC 602 microprocessor. It consists of three major sections that describe the following:

- Registers implemented in the 602
- Operand conventions
- The 602 instruction set

2.1 PowerPC 602 Processor Register Set

This section describes the registers implemented in the 602. These registers can be grouped into three types:

- Registers that are implemented as they are defined by the PowerPC architecture. These registers are identified in this chapter, but are described more fully in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual*.

- Registers that are defined by the PowerPC architecture that have been altered somewhat from the PowerPC architecture definition. Typically this is the result of implementing additional bits in bit locations reserved for use by individual PowerPC processors. This chapter describes the differences.

- Registers that are defined for the 602 that are not defined by the PowerPC architecture. For example, the timer control register (TCR) is used to configure and control the 602's watchdog timer facility, and the ESA save and restore register (ESASRR) provides a place to save and restore state information when the 602-specific **esa** instruction is used to put the processor in supervisor mode.

Some registers are updated as the result of instruction execution. The PowerPC architecture defines register-to-register operations for all computational instructions. Source operands are accessed from the on-chip registers—primarily the 32 general-purpose registers (GPRs) and the 32 floating-point registers (FPRs)—or is provided as an immediate value 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. Data is transferred between memory and registers with explicit load and store instructions only. In addition to the GPRs

and FPRs, other registers can be affected directly by instructions; for example, the counter register (CTR) and condition registers (CRs). Special conditions and errors are reflected in the XER register and the floating-point status and control register (FPSCR).

Some registers are used for a variety of specific purposes, such as time keeping, configuration, and support for exception handling. These registers are called special-purpose registers (SPRs). The SPRs can be read by using the Move from Special-Purpose Register (**mfspr**) instruction and written to by using the Move to Special-Purpose Register (**mfspr**) instruction. Some SPRs are also affected by other operations as well.

When the 602 detects SPR encodings other than those defined in this document, it takes an illegal instruction-type program exception. (Note that the term, 'exception,' is also referred to as 'interrupt' in the architecture specification.) Conversely, some SPRs in the 602 may not be implemented in other PowerPC processors, or may not be implemented in the same way in other PowerPC processors. In general, for registers with reserved bits, implementations return zeros or return the value last written to those bits.

The PowerPC UISA registers, shown in Figure 2-1, can be accessed by either user- or supervisor-level instructions (the architecture specification refers to user- and supervisor-level as problem state and privileged state, respectively). The number to the right of the register name indicates the number used in the syntax of the instruction operands to access the register (for example, the number used to access the XER is SPR1).

SUPERVISOR MODEL

Configuration Registers

PLL Configuration Register[1]
| HID1 | SPR 1009 |

Machine State Register
| MSR |

Checkstop Enables Register[1]
| HID0 | SPR 1008 |

Processor Version Register
| PVR | SPR 287 |

USER MODEL

General-Purpose Registers
| GPR0 |
| GPR1 |
| ⋮ |
| GPR31 |

Single-Precision Floating-Point Registers
| FPR0 |
| FPR1 |
| ⋮ |
| FPR31 |

Condition Register
| CR |

Floating-Point Status and Control Register
| FPSCR |

XER
| XER | SPR 1 |

Link Register
| LR | SPR 8 |

Count Register
| CTR | SPR 9 |

Time Base Facility (For Reading)
| TBL | TBR 268 |
| TBU | TBR 269 |

Memory Management Registers

Instruction BAT Registers
IBAT0U	SPR 528
IBAT0L	SPR 529
IBAT1U	SPR 530
IBAT1L	SPR 531
IBAT2U	SPR 532
IBAT2L	SPR 533
IBAT3U	SPR 534
IBAT3L	SPR 535

Data BAT Registers
DBAT0U	SPR 536
DBAT0L	SPR 537
DBAT1U	SPR 538
DBAT1L	SPR 539
DBAT2U	SPR 540
DBAT2L	SPR 541
DBAT3U	SPR 542
DBAT3L	SPR 543

SDR1
| SDR1 | SPR 25 |

Software Table Search Registers[1]
DMISS	SPR 976
DCMP	SPR 977
HASH1	SPR 978
HASH2	SPR 979
IMISS	SPR 980
ICMP	SPR 981
RPA	SPR 982

Segment Registers
| SR0 |
| SR1 |
| ⋮ |
| SR15 |

ESA Access Registers

ESA Enable Register[1]
| SER | SPR 991 |

ESA Enable Base Register[1]
| SEBR | SPR 990 |

ESA Save/Restore Register[1]
| ESASRR | SPR 987 |

Exception Handling Registers

Data Address Register
| DAR | SPR 19 |

DSISR
| DSISR | SPR 18 |

SPRGs
SPRG0	SPR 272
SPRG1	SPR 273
SPRG2	SPR 274
SPRG3	SPR 275

Save and Restore Registers
| SRR0 | SPR 26 |
| SRR1 | SPR 27 |

Miscellaneous Registers

Time Base Facility (For Writing)
| TBL | SPR 284 |
| TBU | SPR 285 |

Instruction Address Breakpoint Register[1]
| IABR | SPR 1010 |

Single-Precision Tag Register[1]
| SP | SPR 1021 |

Integer Tag Register[1]
| LT | SPR 1022 |

Decrementer
| DEC | SPR 22 |

Timer Control Register[1]
| TCR | SPR 984 |

Interrupt Base Register[1]
| IBR | SPR 986 |

[1] These registers are 602–specific registers. They may not be supported by other PowerPC processors.

Figure 2-1. PowerPC 602 Processor Programming Model

The 602'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 general-purpose register file consists of thirty-two 32-bit GPRs designated as GPR0–GPR31. This register file serves as the data source or destination for all integer instructions and provides data for generating addresses.

  — Floating-point registers (FPRs). The floating-point register file implemented in hardware in the 602 consists of thirty-two 32-bit FPRs designated as FPR0–FPR31, which serve as the data source or destination for all floating-point instructions. The UISA specifies that FPRs be 64 bits wide, to accommodate double-precision operands, however, because the 602 does not support double-precision arithmetic in hardware, the architected 64-bit FPRs are emulated in software for double-precision instructions that require them.

  The smaller single-precision registers need status bits to recognize a valid floating-point operand in the hardware (rather than in a memory image) or an integer value moved from the FPSCR or generated by the **fctiwz** instruction. These status bits are implemented in SPRs (the SP and LT registers) and are accessed and reloaded using the **mfspr**/**mtspr** instructions. For information about the SP and LT registers, see Section 2.1.2.4.1, "Floating-Point Tag Registers (SP and LT)."

  For information on saving and restoring the contents of the FPRs, see Section 2.1.3, "Saving and Restoring FPRs and the FPSCR."

  — Condition register (CR). The CR consists of eight 4-bit fields, CR0–CR7, that reflect the results of certain arithmetic operations and are used for testing and branching.

  — Floating-point status and control register (FPSCR). The FPSCR is used to configure how floating-point operations are handled and to register the results of certain floating-point operations. 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.

The remaining user-level registers are SPRs. These instructions are typically used to explicitly access certain registers, while other SPRs may be more commonly accessed as the side effect of executing other instructions.

  — XER register. Bits in the 32-bit XER are set as the result of specific integer conditions, such as underflows and carries.

  — Link register (LR). The 32-bit link register 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 (referred to as the effective address in the architecture specification) of the instruction that follows a branch and link instruction, typically used for linking to subroutines.

PowerPC 602 RISC Microprocessor User's Manual

— Count register (CTR). The CTR is a 32-bit register for holding 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.

- **User-level registers (VEA)**—The PowerPC VEA introduces the time base facility (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 registers.

- **Supervisor-level registers (OEA)**—The OEA defines the registers that are typically used by an operating system for such operations as memory management, configuration, and exception handling. The 602 implements the supervisor-level registers defined by the PowerPC architecture 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 Interrupt (**rfi**) instructions. It can be read by the Move from Machine State Register (**mfmsr**) instruction. The 602 implements additional bits in the MSR for configuring functionality not defined by the PowerPC architecture. Section 2.1.1.1, "Machine State Register," of this manual describes 602-specific MSR bits.

    – Processor version register (PVR). This read-only register identifies the version (model) and revision level of the PowerPC processor. Section 2.1.1.3, "Processor Version Register," describes how the PVR is used to show the processor version number for 602.

  — **Memory management registers**

    – Block-address translation (BAT) registers. The 602 includes the four pairs of instruction BATs (IBAT0U–IBAT3U and IBAT0L–IBAT3L) and four pairs of data BATs (DBAT0U–DBAT3U and DBAT0L–DBAT3L) defined by the OEA. The 602 implements two additional bits in the lower BAT registers to support 602-specific functionality. These bits are described in Section 2.1.1.4, "BAT Registers."

    – SDR1. The SDR1 register specifies the page table base address used in virtual-to-physical address translation.

    – Segment registers (SR). The PowerPC OEA defines sixteen 32-bit segment registers (SR0–SR15) for 32-bit implementations only.

— **Exception-handling registers**

  – Data address register (DAR). After a data access or an alignment exception, the DAR is set to the effective address generated by the faulting instruction.

  – SPRG0–SPRG3. The SPRG0–SPRG3 registers are provided for operating system use.

  – DSISR. The DSISR defines the cause of data access and alignment exceptions.

  – Machine status save/restore register 0 (SRR0). The SRR0 is used to save the address of the instruction that should be executed after an **rfi** instruction is executed. The instruction address saved to the SRR0 depends on the exception taken.

  – Machine status save/restore register 1 (SRR1). The SRR1 is used to save machine status on exceptions and to restore machine status when an **rfi** instruction is executed. The bits saved to SRR1 depend on the exception taken. See Section 2.1.1.2, "Machine Status Save/Restore Register 1," for information on how SRR1 is implemented in the 602.

— **Miscellaneous registers**

  – The time base facility (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 registers.

    The 602's time base is incremented once every four bus clocks. Additional time base control is achieved through the time base enable (TBEN) signal which serves as a count enable.

  – 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.

  – External access register (EAR). The EAR is an optional 32-bit register that is defined by the PowerPC architecture but not implemented in the 602.

For more information about PowerPC architecture-defined registers refer to Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual*.

2.1.1 PowerPC Registers with Implementation-Specific Bits

A number of registers defined by the PowerPC architecture have additional implementation-specific bits defined for the 602. These registers are described in the following sections.

2.1.1.1 Machine State Register

The 602's implementation of the MSR includes bits described by the PowerPC architecture as well as additional bits that support 602-specific functionality. The MSR is shown in Figure 2-2.
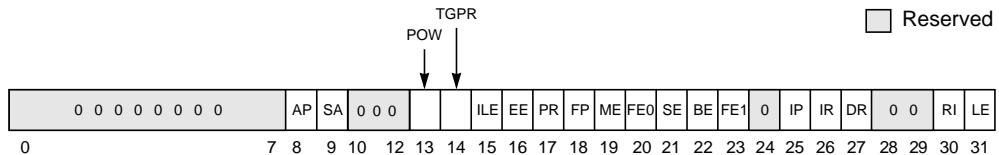
Figure 2-2. Machine State Register (MSR)

The following additional 602-specific bits (shown in Table 2-1) are implemented in the MSR.

Table 2-1. Machine State Register—Implementation-Specific Bits

Bit	Name	Function
8	AP	Access privilege state. This bit is checked only when MSR[PR] = 0; if AP is set, the processor has user-level access to instruction and data space. If AP is cleared, the processor has supervisor-level access to memory.
9	SA	Supervisor access mode. If set, this bit allows execution of supervisor instructions without entering supervisor mode.
13	POW	Activates power management. This bit is defined by the PowerPC architecture, but may not be implemented in all processors. MSR[POW] may be altered with an **mtmsr** instruction only. Also, when altering the POW bit, software may alter only this bit in the MSR and no others. The **mtmsr** instruction must be followed by a context-synchronizing instruction. See Chapter 9, "Power Management," for more information about power management.
14	TGPR	Temporarily replaces TGPR0–TGPR3 with GPR0–GPR3 for use by TLB miss routines. When this bit is set, all instruction accesses to GPR0–GPR3 are mapped to TGPR0–TGPR3, respectively. The contents of GPR0–GPR3 are unchanged as long as this bit remains set. Attempts to use GPR4–GPR31 when this bit is set yields undefined results. The TGPR bit is set when either an instruction TLB miss, data read miss, or data write miss exception is taken. The TGPR bit is cleared by an **rfi** instruction.
20	FE0	IEEE floating-point exception mode. This bit is implemented as defined by the PowerPC architecture; however, if either FE0 or FE1 are set, the 602 operates in precise mode. As defined by the architecture, if both bits are cleared, floating-point exceptions are disabled. For more information, see Section 4.5.7, "Program Exception (0x0700)." These modes operate regardless of the setting of FPSCR[NI].

Bit	Name	Function
23	FE1	IEEE floating-point exception mode. This bit is implemented as defined by the PowerPC architecture; however, if either FE0 or FE1 are set, the 602 operates in precise mode. As defined by the architecture, if both bits are cleared, floating-point exceptions are disabled. For more information, see Section 4.5.7, "Program Exception (0x0700)." These modes operate regardless of the setting of FPSCR[NI].
25	IP	The IP bit, defined by the PowerPC architecture, is implemented differently in the 602. How the IP bit is interpreted depends on the exception. • If a soft reset, machine check, or instruction address breakpoint exception is taken, the IP is used as it is defined by the PowerPC architecture. That is, if IP = 0, the vector address is determined by prefixing 0's to the vector offset. If IP is set, the vector address is determined by prefixing the vector offset with 0xFFF. • If a hard reset is taken, the vector address is always 0XFFF0_0100. • For all other exceptions, if the IP bit is cleared, the vector address is determined by prefixing the contents of the IBR to the vector offset. If IP is set, the vector address is determined by prefixing 0xFFF to the vector offset. The 602-specific interrupt base register can be used to program the top 16 bits or exception addresses. See Section 2.1.2.4.3, "Interrupt Base Register (IBR)."
26	IR	This bit is implemented as defined by the PowerPC architecture. Turns on instruction address translation, protections, and cache control. The DR and IR bits operate as defined by the PowerPC architecture. If IR or DR bits are set, the BAT/TLB hit mechanisms take priority.
27	DR	This bit is implemented as defined by the PowerPC architecture. Turns on data address translation, protections, and cache control. The DR and IR bits operate as defined by the PowerPC architecture. If IR or DR bits are set, the BAT/TLB hit mechanisms take priority.

2.1.1.2 Machine Status Save/Restore Register 1

Table 2-3 shows the 602-specific bits implemented in SRR1 as implemented for table search operations.

Table 2-2. SRR1—PowerPC 602-Specific Bits for Software Table Search Operations

Bit(s)	Name	Function
0–3	CRF0	Condition register field 0 bits
12	KEY	TLB miss protection key
13	I/D	Instruction TLB miss
14	WAY	Specifies which TLB set should be replaced
15	S/L	TLB miss was on a store or load operation

When the 602 takes a machine check exception, it sets one or more error bits in SRR1. Table 2-3 shows the 602-specific bits implemented in SRR1 as implemented for machine check handling.

Table 2-3. SRR1—PowerPC 602-Specific Bits for Machine Check Handling

Bit	Name	Function
12	MCPIN	If set, the exception was caused by the assertion of the machine check interrupt ($\overline{MCP}$) signal.
13	TEA	If set, the exception was caused by the assertion of the transfer error acknowledge ($\overline{TEA}$) signal.

2.1.1.3 Processor Version Register

The processor version number is 0x0005 for the 602. The processor revision level starts at 0x0100 and is incremented for each revision of the processor. The PVR is described in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual.*

2.1.1.4 BAT Registers

The PowerPC OEA defines the BAT registers as eight instruction block-address translation (IBAT) registers, consisting of four pairs of instruction BATs, or IBATs (IBAT0U–IBAT3U and IBAT0L–IBAT3L) and eight data BATs, or DBATs, (DBAT0U–DBAT3U and DBAT0L–DBAT3L). The BAT registers (BATs) maintain the address translation information for four instruction blocks and four data blocks in memory. BAT registers define the starting addresses and sizes of BAT areas as well as other characteristics of each block.

Figure 2-3 and Figure 2-4 show the format of the upper and lower BAT registers for 32-bit PowerPC processors.
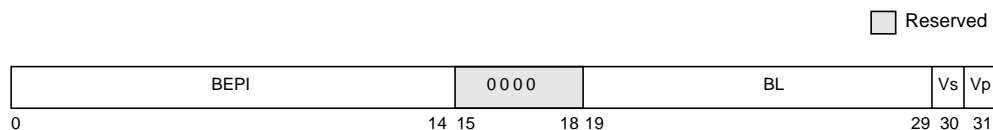
Figure 2-3. Format of Upper BAT Registers—32-Bit Implementations
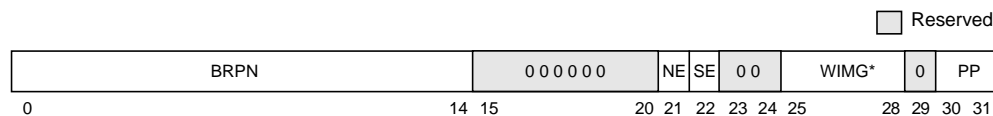
*W and G bits are reserved (not defined) for IBAT registers

Figure 2-4. Format of Lower BAT Registers—32-Bit Implementations

Table 2-4 describes the bits in the BAT registers.

Table 2-4. BAT Registers—Field and Bit Descriptions

Register	Bit(s)	Name	Description
Upper BAT register	0–14	BEPI	Block effective page index. This field is compared with high-order bits of the logical address to determine if there is a hit in that BAT array entry. (Note that the architecture specification refers to logical address as effective address.)
	15–18	—	Reserved
	19–29	BL	Block length. BL is a mask that encodes the size of the block. Values for BL are listed in Table 2-5.
	30	Vs	Supervisor mode valid bit. This bit interacts with MSR[PR] to determine if there is a match with the logical address. For more information, see Section 5.3, "Block Address Translation."
	31	Vp	User mode valid bit. This bit and MSR[PR] determine if there is a match with the logical address. See Section 5.3, "Block Address Translation."
Lower BAT register	0–14	BRPN	This field is used with BL to generate high-order bits of the physical address of the block.
	15–20	—	Reserved
	21	NE	No execute. This bit controls execute privileges for the block. When this bit is set, instructions cannot be fetched from this block. Note that setting SR[N] also inhibits execute privileges on a 256-Mbyte basis and overrides a setting of zero for the NE bit. The NE bit is valid only in the IBATs and is specific to the 602.
	22	SE	ESA enable. This bit controls whether the **esa** instruction, which puts the processor in supervisor mode, can execute from this block. The SE bit is valid only in the IBATs and is specific to the 602.
	23–24	—	Reserved
	25–28	WIMG	Memory/cache access mode bits W Write-through I Caching-inhibited M Memory coherence G Guarded For detailed information about the WIMG bits, see Section 3.6, "Memory Management/Cache Access Mode Bits—W, I, M, and G."
	29	—	Reserved
	30–31	PP	Protection bits for block. This field determines the protection for the block as described in Section 5.3, "Block Address Translation."

The value loaded into BL determines both the length and alignment of the BAT area in both logical and physical address space. The values loaded into BEPI and BRPN must have at least as many low-order zeros as there are ones in BL. Table 2-5 lists the BAT area lengths encoded in BAT[BL].

For more information on the BAT registers, refer to *The Programming Environments Manual*. Use of BAT registers is described in Chapter 5, "Memory Management."

Table 2-5. BAT Area Lengths

BAT Area Length	BL Encoding	BAT Area Length	BL Encoding
128 Kbytes	000_0000_0000	8 Mbytes	000_0011_1111
256 Kbytes	000_0000_0001	16 Mbytes	000_0111_1111
512 Kbytes	000_0000_0011	32 Mbytes	000_1111_1111
1 Mbyte	000_0000_0111	64 Mbytes	001_1111_1111
2 Mbytes	000_0000_1111	128 Mbytes	011_1111_1111
4 Mbytes	000_0001_1111	256 Mbytes	111_1111_1111

2.1.2 PowerPC 602 Processor-Specific Registers

The 602 includes several implementation-specific, supervisor-level SPRs not defined by the PowerPC architecture, as shown in Table 2-6.

Table 2-6. PowerPC 602 Processor-Specific SPRs

Register Name	Function	SPR (Decimal)	SPR spr_{5-9}	spr_{0-4}	R/W
HID0	Checkstop/miscellaneous enables	1008	11111	10000	R/W
HID1	PLL configuration values	1009	11111	10001	Read-only
IABR	Instruction address breakpoint register	1010	11111	10010	R/W
SP	FPU single-precision tags	102	11111	11101	R/W
LT	FPU integer tags	1022	11111	11110	R/W
DMISS	DTLB miss address register	976	11110	10000	R/W
DCMP	DTLB miss compare register	977	11110	10001	R/W
HASH1	Primary hash address	978	11110	10010	Read-only
HASH2	Secondary hash address	979	11110	10011	Read-only
IMISS	ITLB miss address register	980	11110	10100	R/W
ICMP	ITLB miss compare register	981	11110	10101	R/W
RPA	Required physical address register	982	11110	10110	R/W
TCR	Timer control register	984	11110	11000	R/W
IBR	Interrupt base register	986	11110	11010	R/W
ESASRR	ESA save and restore register	987	11110	11011	R/W
SER	ESA enable register	991	11110	11111	R/W
SEBR	ESA enable base register	990	11110	11110	R/W

DMISS, IMISS, DCMP, ICMP, HASH1, HASH2, and RPA are used for software table search operations and should be accessed only when address translation is disabled (that is, MSR[IR] = 0 and MSR[DR] = 0). For a complete discussion of software table search operations, refer to Section 5.5, "Page Table Search Operation."

2.1.2.1 Configuration Registers

The 602 provides additional configuration registers for enabling and disabling 602-specific functions, such as power management, cache control, protection-only mode, and PLL configuration. These registers are described in the following sections.

2.1.2.1.1 Hardware Implementation Register 0 (HID0)

The hardware implementation register 0 (HID0), shown in Figure 2-5, defines enable bits for various 602-specific features.
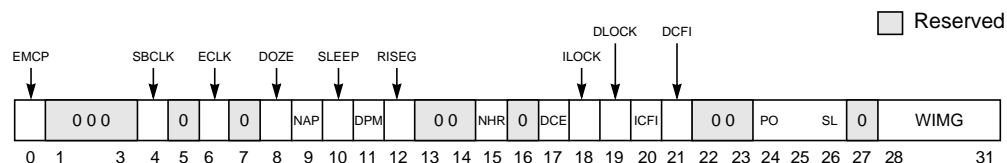
Figure 2-5. Hardware Implementation Register 0 (HID0)

Table 2-7 shows the bit definitions for HID0.

Table 2-7. HID0 Bit Settings

Bit(s)	Name	Description
0	EMCP	Enable machine check pin. EMCP is used to mask machine check interrupts caused by the assertion of $\overline{\text{MCP}}$. Setting EMCP enables the $\overline{\text{MCP}}$ signal to cause a checkstop if MSR[ME] is cleared or a machine check interrupt if MSR[ME] is set. Clearing EMCP prevents the $\overline{\text{MCP}}$ signal from causing either a machine check interrupt or a checkstop.
1–3	—	Not used
4	SBCLK	Select bus clock for test clock pin. 1 The test clock, CLK_OUT, runs at the bus frequency. 0 The test clock, CLK_OUT, runs at the processor frequency. Used in combination with SBCLK to determine configure the CLK_OUT signal. SeeTable 2-8.
5	—	Not used
6	ECLK	Enable external test clock pin. Used in combination with SBCLK to configure the CLK_OUT signal. SeeTable 2-8.
7	—	Not used
8	DOZE	Doze mode—PLL, time base, and snooping active. See Chapter 9, "Power Management."
9	NAP	Nap mode—PLL and time base active. See Chapter 9, "Power Management."
10	SLEEP	Sleep mode—no external clock required. See Chapter 9, "Power Management."
11	DPM	Enable dynamic power management. See Chapter 9, "Power Management."

Table 2-7. HID0 Bit Settings (Continued)

Bit(s)	Name	Description
12	RISEG	Reserved for test.
13–14	—	Not used
15	NHR	Not hard reset. Software can set this bit at start up to indicate that soft reset can be distinguished from hard reset. This bit is subsequently cleared by a hard reset.
16	—	Not used
17	DCE	Data cache enable. To prevent a cache from being disabled in the middle of an access the setting of this bit must be preceded by a **sync** instruction. To guarantee that instructions are cleared from the instruction queue after the cache is disabled, an **isync** instruction should follow the **mtspr** instruction that updates HID0.
18	ILOCK	Instruction cache lock. A locked cache can supply data normally on a hit, but a miss operation is treated as a caching-inhibited transaction. The setting of the ILOCK must be preceded by an **isync** instruction to prevent the cache from being locked while it is being accessed.
19	DLOCK	Data cache lock. A locked cache can supply data normally on a hit, but a miss operation is treated as a caching-inhibited transaction. A snoop hit to a locked data cache performs as if the cache were not locked. A cache block invalidated by a snoop remains invalid until the cache is unlocked. The setting of the DLOCK bit must be preceded by a **sync** instruction to prevent the cache from being locked while it is being accessed.
20	ICFI	Instruction cache flash invalidate. Setting this bit causes the instruction cache to be invalidated. Both caches are invalidated automatically upon power-up (hard reset). Soft reset does not invalidate the caches automatically, so ICFI must be set if invalidation is desired after a soft reset. Proper use of this bit is to set it and clear it in two consecutive **mtspr** operations. This creates an adequate window for the operation to be performed. Between the two stores, the tags are continuously invalidated.
21	DCFI	Data cache flash invalidate. Setting this bit causes the data cache to be invalidated after a soft reset. Both caches are invalidated automatically upon power-up (hard reset). Soft reset does not invalidate the caches automatically, so DCFI must be set if invalidation is desired after a soft reset. Proper use of this bit is to set it and clear it in two consecutive **mtspr** operations. This creates an adequate window for the operation to be performed. Between the two stores, the tags are continuously invalidated.
22–23	—	Not used
24	PO	Protection-only mode. Setting PO enables the 602-specific protection-only mode to be used after a BAT miss. See Section 5.6, "Protection-Only Mode."
25	—	Not used
26	SL	Enable out-of-order loads on the bus.
27	—	Not used
28–31	WIMG	Default WIMG settings used in real addressing mode and protection-only mode.

Table 2-8 shows how the HID0[ECLK] and HID0[SBCLK] are used to configure the CLK_OUT signal.

Table 2-8. CLK_OUT Signal Configuration

HID0[ECLK]	HID0[SBCLK]	CLK_OUT
0	0	High impedance
0	1	High impedance
1	0	Processor clock
1	1	Bus clock (= SYS_CLK)

For more information on the CLK_OUT, see Section 7.2.11.2, "Test Clock (CLK_OUT)—Output."

The HID0 register is accessed as SPR 1008.

2.1.2.1.2 Hardware Implementation Register 1 (HID1)—PLL Configuration

In the 602, the HID1 register is used to configure the PLL. The HID1 register is shown in Figure 2-6.
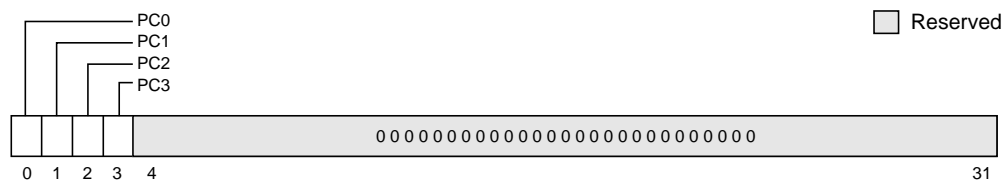
Figure 2-6. HID1—PLL Configuration Register

Table 2-9 describes the bits in the HID1 implemented in the 602.

Table 2-9. HID1 Bit Settings

Bit(s)	Name	Function
0	PC0	PLL configuration bit 0 (read only)
1	PC1	PLL configuration bit 1 (read only)
2	PC2	PLL configuration bit 2 (read only)
3	PC3	PLL configuration bit 3 (read only)
4–31	Not used	—

The HID1 register is accessed as SPR 1009.

2.1.2.2 PowerPC 602 Processor Memory Management Registers

The 602 implements additional registers not defined by the PowerPC architecture for memory management. These registers are implemented primarily to support the 602's software table search operations. These registers are described in the following sections. For more detailed information about how these registers are used with the 602's MMU, see Chapter 5, "Memory Management."

2.1.2.2.1 Data and Instruction TLB Miss Address Registers (DMISS and IMISS)

The DMISS and IMISS registers have the same format, as shown in Figure 2-7. They are loaded automatically upon a data or instruction TLB miss. The DMISS and IMISS contain the effective page address of the access that caused the TLB miss exception. The contents are used by the 602 when calculating the values of HASH1 and HASH2, and by the **tlbld** and **tlbli** instructions when loading a new TLB entry. Note that the 602 always loads the DMISS register with a big-endian address, even when MSR[LE] is set. These registers are read-only to the software.

Effective Page Address
0   31
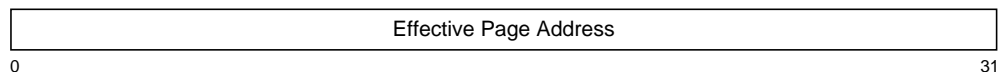
Figure 2-7. DMISS and IMISS Registers

The DMISS, IMISS, DCMP, ICMP, HASH1, HASH2, and RPA registers should be accessed with translation disabled (MSR[IR] = 0 and MSR[DR] = 0).

2.1.2.2.2 Data and Instruction PTE Compare Registers (DCMP and ICMP)

The DCMP and ICMP registers, shown in Figure 2-8, contain the first word in the required PTE. The contents are constructed automatically from the contents of the segment registers and the effective address (DMISS or IMISS) when a TLB miss exception occurs. Each PTE read from the tables during the table search process should be compared with this value to determine whether or not the PTE is a match. Upon execution of a **tlbld** or **tlbli** instruction, the DCMP or ICMP register is loaded into the first word of the selected TLB entry.

V	VSID	H	API
0 1	24	25 26	31
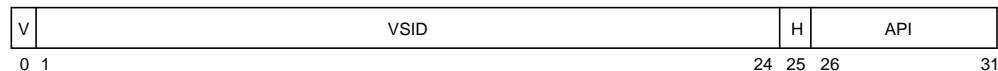
Figure 2-8. DCMP and ICMP Registers

Table 2-10 describes the bit settings for the DCMP and ICMP registers.

Table 2-10. DCMP and ICMP Bit Settings

Bit(s)	Name	Description
0	V	Valid bit. Set by the processor on a TLB miss exception.
1–24	VSID	Virtual segment ID. Copied from VSID field of the corresponding segment register.
25	H	Hash function identifier. Cleared by the processor on a TLB miss exception.
26–31	API	Abbreviated page index. Copied from API of effective address.

Note that DMISS, IMISS, DCMP, ICMP, HASH1, HASH2, and RPA should be accessed with translation disabled (MSR[IR] = 0 and MSR[DR] = 0).

The DCMP register can be accessed as SPR 977; the ICMP register can be accessed as SPR 981.

2.1.2.2.3 Primary and Secondary Hash Address Registers (HASH1 and HASH2)

The HASH1 and HASH2 registers are read-only, supervisor-level SPRs that contain the physical addresses of the primary and secondary PTEGs for the access that caused the TLB miss exception. Only bits 7–25 differ between them. For convenience, the 602 automatically constructs the full physical address by routing bits 0–6 of SDR1 into HASH1 and HASH2 and clearing the lower-order 6 bits. These registers are read-only and are constructed from the contents of the DMISS or IMISS register. The format for the HASH1 and HASH2 registers is shown in Figure 2-9.

Reserved

HTABORG[0–6]	Hashed Page Address	0 0 0 0 0 0

0　　　　　　6　7　　　　　　　　　　　　　　　　　25　26　　　　　31
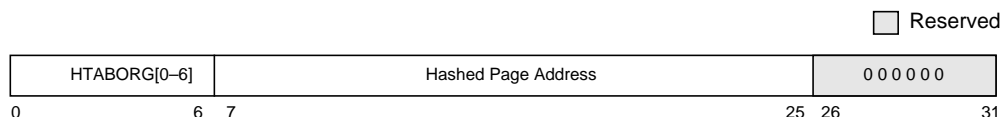
Figure 2-9. HASH1 and HASH2 Registers

Table 2-11 describes the bit settings of the HASH1 and HASH2 registers.

Table 2-11. HASH1 and HASH2 Bit Settings

Bits	Name	Description
0–6	HTABORG[0–6]	Copy of the upper 7 bits of the HTABORG field from SDR1
7–25	Hashed page address	Address bits 7–25 of the PTEG to be searched
26–31	—	Reserved

Note that DMISS, IMISS, DCMP, ICMP, HASH1, HASH2, and RPA should be accessed with translation disabled (MSR[IR] = 0 and MSR[DR] = 0).

The HASH1 register can be read as SPR 978; the HASH2 register can be read as SPR 979.

2.1.2.2.4 Required Physical Address Register (RPA)

The RPA register, shown in Figure 2-10, is used to hold the physical address and is used in conjunction with page table search operations performed in software on the 602. During a page table search operation, the software must load the RPA with the second word of the correct PTE. When the **tlbld** or **tlbli** instruction is executed, the contents of the RPA register and the DMISS or IMISS register are merged and loaded into the selected TLB entry.

☐ Reserved

RPN	0	NE	SE	R	C	WIMG	0	PP

0   19 20   21 22   23 24 25   28 29 30 31
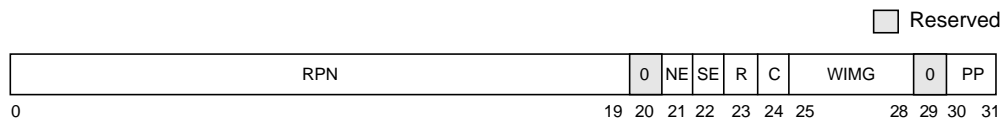
Figure 2-10. Required Physical Address Register (RPA)—Default Configuration

Table 2-12 describes the bit settings of the RPA register.

Table 2-12. RPA Bit Settings—Default Configuration

Bit(s)	Name	Description
0–19	RPN	Physical page number from PTE.
20	—	Reserved
21	NE	No execute. Controls execute privileges for that page. If set, instructions cannot be fetched from that 4-Kbyte page. NE is a don't care if SR[N] is set.
22	SE	ESA enable. The SE bit is used to control whether the **esa** instruction can be executed. Executing **esa** puts the processor in supervisor mode without taking an exception.
23	R	Referenced bit from PTE
24	C	Changed bit from PTE
25–28	WIMG	Memory/cache access attribute bits
29	—	Reserved
30–31	PP	Page protection bits from PTE

Note that the DMISS, IMISS, DCMP, ICMP, HASH1, HASH2, and RPA registers should be accessed with translation disabled (MSR[IR] = 0 and MSR[DR] = 0).

The RPA register can be accessed as SPR 982.

2.1.2.2.5 RPA Register in Protection-Only Mode

The RPA register should be loaded by the processor with the second word of the correct PTE during a page table search. In protection-only mode, the format of the PTE, the TLB entries, and the RPA register are different in that each contains 32 protection bits for the 128-Kbyte region they define.

For ITLB loads, the RPA register should contain 32 no-execute (NE) bits, that control whether instructions from the corresponding pages can be executed. The organization of the RPA for ITLB loads is shown in Figure 2-11.

NE_0	NE_1	NE_2	NE_3	NE_4	NE_5	NE_6	NE_7	NE_8	NE_9	NE_{10}	NE_{11}	NE_{12}	NE_{13}	NE_{14}	NE_{15}	NE_{16}	NE_{17}	NE_{18}	NE_{19}	NE_{20}	NE_{21}	NE_{22}	NE_{23}	NE_{24}	NE_{25}	NE_{26}	NE_{27}	NE_{28}	NE_{29}	NE_{30}	NE_{31}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Figure 2-11. RPA for ITLB Loads—Protection-Only Mode

Before a TLB Load Instruction (**tlbli**) is executed, the RPA register should be loaded with 32 NE bits.

For DTLB loads, the RPA register should contain 32 write-enable (WE) bits, that control whether instructions from the corresponding pages can be executed. For DTLB loads, the RPA organization is shown in Figure 2-12.

WE_0	WE_1	WE_2	WE_3	WE_4	WE_5	WE_6	WE_7	WE_8	WE_9	WE_{10}	WE_{11}	WE_{12}	WE_{13}	WE_{14}	WE_{15}	WE_{16}	WE_{17}	WE_{18}	WE_{19}	WE_{20}	WE_{21}	WE_{22}	WE_{23}	WE_{24}	WE_{25}	WE_{26}	WE_{27}	WE_{28}	WE_{29}	WE_{30}	WE_{31}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Figure 2-12. RPA for DTLB Loads—Protection-Only Mode

Before a TLB Load Data (**tlbld**) instruction is executed, the RPA register should be loaded with 32 WE bits.

2.1.2.3 ESA Supervisor Access Registers

The 602 defines a set of resources that allow the processor to access supervisor instructions, registers, and memory resources without taking an exception. This supervisor access is signaled by the execution of the 602-specific **esa** instruction. Execution of this instruction is allowed only if it is enabled for page or block in which it resides.

There are three registers that are 602-specific that support this functionality:

- ESASRR, which is used to save information about the context of the processor when the **esa** instruction is executed

The remaining two registers are used to control access to the **esa** instruction only when the processor is running in protection-only mode. They are:

- SEBR, which contains the base address for the 128-Kbyte region which is broken into 32, 4-Kbyte pages
- Each of the 32 bits in the SER control whether the **esa** instruction can be executed from the corresponding 4-Kbyte page.

All three registers are described in the following sections.

2.1.2.3.1 ESA Save and Restore Register (ESASRR)

The ESA save and restore register (ESASRR) is a supervisor-level register that provides a means for automatically saving and restoring aspects of the machine state for use with the enable/disable supervisor access instructions (**esa** and **dsa**).

☐ Reserved

| 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 0 0 0 | PR | AP | SA | EE |

0   27   28   29   30   31
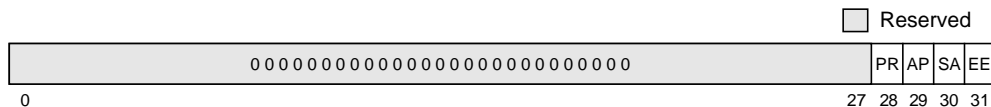
Figure 2-13. ESASRR—ESA Save and Restore Register

The bits in the ESASRR are described in Table 2-13.

Table 2-13. ESASRR Bit Settings

Bit(s)	Name	Function
0–27	—	—
28	PR	Copy of MSR[PR] when **esa** is executed
29	AP	Copy of MSR[AP] when **esa** is executed
30	SA	Copy of MSR[SA] when **esa** is executed
31	EE	Copy of MSR[EE] when **esa** is executed

When an **esa** instruction is executed, MSR[SA, EE, PR, AP] are updated and the previous values are automatically saved in the ESASRR. When a **dsa** instruction is executed, the contents of these bits are automatically restored to the MSR.

The ESASRR can be accessed explicitly using SPR number 987.

2.1.2.3.2 ESA Enable Base Register (SEBR) (Protection-Only Mode)

The ESA enable base register (SEBR) and ESA enable register (SER) are used to control whether the **esa** instruction can be executed for each of the 32 pages of a 128-Kbyte region of memory when the processor is operating in protection-only mode (MSR[PO] = 0).

☐ Reserved

| Base Address | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |

0   14   15   31
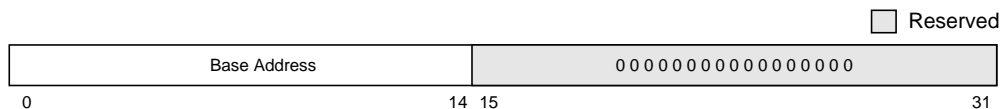
Figure 2-14. ESA Enable Base Register (SEBR)

SEBR[0–14] contains the base address of the 128-Kbyte region that is protected by the 32 SE bits in SER (each bit in the SER configures a 4-Kbyte page). SEBR[0–14] are compared against the EA[0–14]. If a match occurs, EA[15–19] indicate which of the 32 SE bits in the SER is examined to determine whether the **esa** instruction can be executed from the corresponding 4-Kbyte page. If there is no match, SE = 0. The matching requirement of the SEBR is similar to the BAT register.

The SER and SEBR registers do not affect protection checking unless the processor is operating in protection-only mode. If HID0[PO] = 0, these registers can be read and written to, but are not used by the MMU.

The SEBR register is accessed as SPR 990.

2.1.2.3.3 ESA Enable Register (SER) (Protection-Only Mode)

The ESA enable register (SER), shown in Figure 2-15, contains 32 SE bits that control the ability to execute the **esa** instruction on a per-page basis when the processor is operating in protection-only mode (MSR[PO] = 0).

SE$_0$	SE$_1$	SE$_2$	SE$_3$	SE$_4$	SE$_5$	SE$_6$	SE$_7$	SE$_8$	SE$_9$	SE$_{10}$	SE$_{11}$	SE$_{12}$	SE$_{13}$	SE$_{14}$	SE$_{15}$	SE$_{16}$	SE$_{17}$	SE$_{18}$	SE$_{19}$	SE$_{20}$	SE$_{21}$	SE$_{22}$	SE$_{23}$	SE$_{24}$	SE$_{25}$	SE$_{26}$	SE$_{27}$	SE$_{28}$	SE$_{29}$	SE$_{30}$	SE$_{31}$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Figure 2-15. ESA Enable Register (SER)

Each SE bit corresponds to a 4-Kbyte page. To execute an **esa** instruction, the 602 must first determine whether the page on which the **esa** instruction resides allows it to be executed. This is done by comparing the high-order 15 bits of the EA against the same bits in the SEBR. If a match occurs, EA[15–19] indicate the 4-Kbyte page, providing an index to the SER from which the appropriate SE bit is read.

If SE = 1, an **esa** instruction residing on that page can be executed, putting the processor in supervisor mode. If SE = 0, the **esa** instruction is disabled for this page. SER[0] corresponds to the lowest page in memory, SER[1] corresponds with the next higher page in memory.

The SER and SEBR registers do not affect protection checking unless the processor is in protection-only mode. If the processor is not in protection-only mode, these registers can be read and written to, but are not used by the MMU.

The SER is accessed as SPR 991.

2.1.2.4 Miscellaneous PowerPC 602 Processor–Specific Registers

The following sections describe 602-specific registers that support a variety of functions, such as software support for double-precision floating-point operations and the watchdog timer facility.

2.1.2.4.1 Floating-Point Tag Registers (SP and LT)

Because the 602 does not support double-precision arithmetic in hardware, the 602 provides two 32-bit, SPRs that characterize the contents of the 32 FPRs. The SP tag register holds tags that identify single-precision values and the LT tag register holds tags that identify integer values.

Each bit of each register corresponds to a single 32-bit FPR. An SP or LT bit being set indicates that the corresponding register contains valid data—SP designating single-precision floating-point data and LT designating integer data. If neither bit is set, the data resides in memory in the associated double-precision emulated FPR.

During power-on reset, the bits in the SP and LT registers are not automatically cleared to all zeros and must be cleared by using **mtspr** instructions in the reset routine.

Care should be taken to ensure that the SP/LT bits for an associated FPR are not inadvertently altered by the **mtspr** instruction; valid data residing in an FPR that has its SP/LT bits changed causes erroneous results when the FPR is used as an operand.

Instructions requiring single-precision values as operands cause an emulation trap exception if any of the operand's associated SP bits are not set. The SP register is accessed as SPR 1021; The LT register is accessed as SPR 1022.

2.1.2.4.2 Timer Control Register (TCR)

The timer control register (TCR) is a supervisor-level SPR used to program the watchdog timer, which is an implementation-specific feature of the 602 and is not defined by the PowerPC architecture. The TCR is shown in Figure 2-16.
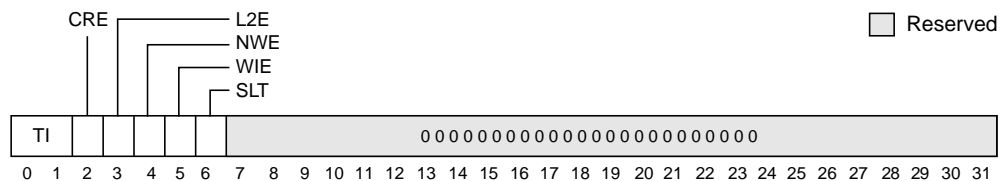
Figure 2-16. Timer Control Register (TCR)

The bits in the TCR are described in Table 2-14.

Table 2-14. Timer Control Register Bit Settings

Bit(s)	Name	Description
0–1	TI	The timer interval bits indicate the number of clock cycles that should occur before the watchdog timer interrupt exception is taken. 00 2e23 clock cycles (ca. 0.25 s) 01 2e24 clock cycles (ca. 0.50 s) 10 2e25 clock cycles (ca. 1.00 s) 11 2e26 clock cycles (ca. 2.00 s) Approximate durations assume 33 MHz bus running in 2:1 mode. For example, if the TI bit is set as 0b00, as soon as bit 8 is set (that is, after 2e23 clock cycles) a carry-out occurs.
2	CRE	Timer core reset enable 0 Timer core reset disabled 1 Timer core reset enabled
3	L2E	Level 2 watchdog timer interrupt enable. Enables the watchdog timer level 2 interrupt after a carry-out occurs from the bit in the time base register specified by the user. 0 Timer level 2 interrupt disabled 1 Timer level 2 interrupt enabled
4	NWE	Next watchdog timer interrupt enable 0 Enable next interrupt 1 Disable next interrupt
5	WIE	Watchdog timer interrupt enable 0 Interrupt disabled 1 Interrupt enabled
6	SLT	Second-level exception taken. This bit is used by software to determine if the watchdog timer caused the soft reset. 0 Second-level soft reset not taken 1 Second-level soft reset taken
7–31	—	—

For information about the watchdog timer utility, see Section 4.5.17, "Watchdog Timer Interrupt (0x1500)."

The TCR is accessed as SPR 984.

2.1.2.4.3 Interrupt Base Register (IBR)

The IBR is used to store a 16-bit base address used to determine the exception vector prefix for certain exceptions and under certain conditions. The 16-bit base address is concatenated with the exception vector offset to form the address for the exception handler. The IBR can be read and written to by the processor. See Figure 2-17 for the format of this register.

Interrupt Base Address	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 15	16 31
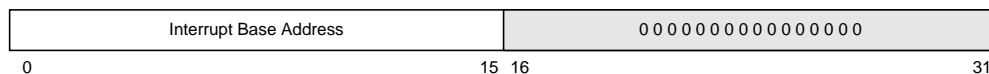
Figure 2-17. Interrupt Base Register

The exception vector is determined as follows:

- For all exceptions, if MSR[IP] is set, the prefix is 0xFFF0.

- For all exceptions except system reset on a hard reset, machine check, and instruction address breakpoint exceptions, if MSR[IP] is cleared, the value of the IBR is used as the 16-bit prefix. For a hard reset, a machine check, or an instruction address breakpoint exception, the prefix is 0x0000 if MSR[IP] is clear.

The IBR is cleared and MSR[IP] is set on a power-on reset; therefore, the system reset exception vector on a power-on reset is 0xFFF0_0100.

Table 2-15 shows which exceptions use the IBR to determine the vector address.

Table 2-15. Determining the Exception Vector Address

Exception Type	Vector (hexadecimal)		
	Prefix		Offset
	IP = 0	IP = 1	
System reset (hard reset)	FFF0		0100
System reset (soft reset)	0000	FFF0	0100
Machine check	0000	FFF0	0200
DSI	IBR	FFF0	0300
ISI	IBR	FFF0	0400
External interrupt	IBR	FFF0	0500
Alignment	IBR	FFF0	0600
Program	IBR	FFF0	0700
Floating-point unavailable	IBR	FFF0	0800
Decrementer	IBR	FFF0	0900
System call	IBR	FFF0	0C00
Trace	IBR	FFF0	0D00
Floating-point assist	IBR	FFF0	0E00
Instruction translation miss	IBR	FFF0	1000

Table 2-15. Determining the Exception Vector Address (Continued)

Exception Type	Vector (hexadecimal)		
	Prefix		Offset
	IP = 0	IP = 1	
Data load translation miss	IBR	FFF0	1100
Data store translation miss	IBR	FFF0	1200
Instruction address breakpoint	0000	FFF0	1300
System management interrupt	IBR	FFF0	1400
Watchdog timer	IBR	FFF0	1500
Emulation trap	IBR	FFF0	1600

For soft system reset exceptions or machine check exceptions, if MSR[IP] is cleared the IBR is not used for the interrupt prefix. In these cases, the offset is 0x0000.

If a soft reset-type system reset interrupt or machine check interrupt occurs, the 602 does not use the value of IBR, but reverts to the value for the interrupt prefix specified by MSR[IP].

The 602 generates a system reset exception if the $\overline{\text{SRESET}}$ signal is asserted. Unlike a hard reset, latches are not initialized and the instruction cache is disabled. The $\overline{\text{SRESET}}$ signal must be asserted for at least two bus clock cycles. After $\overline{\text{SRESET}}$ is deasserted, the 602 vectors to the system reset exception handler at 0xFFF0_0100. The IBR is not used as a vector offset for soft reset.

The IBR register is accessed as SPR 986.

2.1.2.4.4 Instruction Address Breakpoint Register (IABR)

The IABR, shown in Figure 2-18, is used in conjunction with the instruction address breakpoint exception. IABR[CEA] holds an effective address to which the address of each instruction is compared. The exception is enabled by setting IABR[IE]. The exception is taken when the instruction breakpoint address matches the next instruction to complete. The instruction tagged with the match is not completed before the breakpoint exception is taken.

☐ Reserved

CEA	IE	0

0   29 30 31
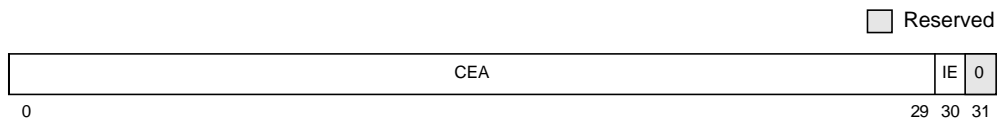
Figure 2-18. Instruction Address Breakpoint Register (IABR)

The fields in the IABR are described in Table 2-16.

Table 2-16. Instruction Address Breakpoint Register Bit Settings

Bit	Name	Description
0–29	CEA	This field holds an effective address to which the address of each instruction is compared.
30	IE	Setting the IE bit enables the instruction address breakpoint exception.
31	—	Reserved

For information about the instruction address breakpoint register, see Section 4.5.15, "Instruction Address Breakpoint Exception (0x1300)."

The IABR is accessed as SPR 1010.

2.1.3 Saving and Restoring FPRs and the FPSCR

The Store Floating-Point Double (**stfd**) and Load Floating-Point Double (**lfd**) instructions can be used to save and restore the 32-bit hardware FPRs. As long as the SP bit is set and the data in the hardware FPR is not an infinity, NaN, or denormalized number, the data in the FPR will be stored with the **stfd** instruction as a double-precision number with data in the single-precision range, and the store instruction does not trap.

For data residing in the 64-bit emulated FPRs, and for the cases of infinities, NaNs, denormalized numbers, and integers residing in the hardware FPRs, the **stfd** instruction traps to 0x1600 and is emulated. The **lfd** instruction does not trap if the operand data is within the single-precision range (with regard to the exponent and fraction). If an operand is outside that range, or if the 64-bit operand value is an infinity, NaN, or single-precision format denormalized number, the instruction traps to 0x1600 and is emulated.

To save the contents of the FPSCR, an **mffs** (Move from FPSCR) instruction can be issued followed by an **stfd** instruction. To restore the FPSCR, an **lfd** instruction can be issued on the data that was previously stored with the **stfd** instruction followed by an **mtfsf** (Move to FPSCR Fields) instruction. The **stfd** instruction traps on the integer data and it is left to the emulation code to expand the integer to its architected value while clearing the high-order 32 bits of the architected value, and stores the 64-bit value. The load instruction acting on the data that was stored by the **stfd** instruction also takes an emulation trap exception. It is left to the emulation code to place bits 32–63 of the operand into the hardware FPR, clear the corresponding SP bit, and set the corresponding LT bits. The **mtfsf** executes on the data in the FPR, placing it in the FPSCR.

2.1.4 Synchronization Requirements for SPRs

As specified by the OEA portion of the PowerPC architecture, altering certain registers requires software synchronization to honor register dependencies for subsequent instructions. A context-synchronizing operation must follow any instruction that affects instruction fetching or data access dependencies by altering any of the following registers:

- Instruction fetch dependencies
  - — MMU control register
  - — IBATs
  - — MSR[AP, FP, FE0, FE1, LE, TE, PE, SA]
  - — IABR
- Data access dependencies
  - — MMU control register
  - — DBATs
  - — MSR[AP, LE, TE, PE, SA]
- Other
  - — MSR[POW, TGPR, FP]
  - — HID0—Context-synchronizing operations that may be used are **isync**, **sc**, **rfi**, and any exception other than system reset or machine check.

Note that MSR[POW] and MSR[LE] may not be altered concurrently with any other MSR bit. Software must alter only one bit in the MSR when altering either of these, and the alteration must be followed by a context-synchronizing operation.

2.2 Operand Conventions

This section describes the operand conventions as they are represented in two levels of the PowerPC architecture. It also provides detailed descriptions of conventions used for storing values in registers and memory, accessing the 602's registers, and representation of data in these registers.

2.2.1 Floating-Point Execution Models—UISA

The IEEE 754 standard includes 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. The 602 implements single-precision instructions in hardware and double-precision instructions in software. Section 6.8.4, "FPU Instruction Timings," indicates which instructions are implemented in hardware and which take the emulation trap exception.

The PowerPC UISA follows these guidelines:

- 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, conversions from double- to single-precision must be done explicitly by software, while conversions from single- to double-precision are done implicitly.

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 1:

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

Because all double-precision arithmetic instructions take an emulation trap exception, single-precision instructions always operate faster than their double-precision equivalents.

Single-precision instructions with operands residing in hardware FPRs with their associated SP bit set (LT is a don't care) execute in hardware as defined by the architecture placing the result in the hardware **fr**D, setting the associated SP bit, and clearing the associated LT bit. If any of the operands have their associated SP bits cleared, the instruction causes an emulation trap exception (0x1600).

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 instructions, a sequence of 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 2-17. (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.

Note that the 602 provides hardware support for misaligned memory accesses; however, a misaligned access suffers a slight performance degradation compared to an aligned access of the same type. The 602 does not provide hardware support for floating-point store operations that are not word-aligned. Instead, an alignment exception is taken.

Floating-point single-word accesses should be word-aligned and floating-point double word accesses should be double-word-aligned. Frequent use of misaligned accesses degrades system performance.

Any memory access that crosses an alignment boundary must be broken into multiple discrete accesses. This includes half-word, word, double-word, and multiple-word references. Multiple-word accesses are architecturally required to be aligned. The resulting performance degradation depends upon how well each individual access behaves with respect to the memory hierarchy. At a minimum, additional cache access cycles are required. More dramatically, for the case of access to a noncacheable page, each discrete access involves an individual bus operation which will reduce the effective bandwidth of the bus. The effect that misalignment and cache misses have on instruction timing is described in Chapter 6, "Instruction Timing."

The casual use of misaligned accesses is discouraged since they can compromise the overall performance of the processor.

2.2.4 Floating-Point Operand

The 602 provides hardware support for all single-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*.

Table 2-17. Memory Operands

Operand	Length	Addr[60–63] (If Aligned)
Byte	8 bits	xxxx
Half word	2 bytes	xxx0
Word	4 bytes	xx00
Double word	8 bytes	x000
Quad word	16 bytes	0000

Note: An "x" in an address bit position indicates that the bit can be 0 or 1 independent of the state of other address bits.

2.2.5 Effect of Operand Placement on Performance

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

2.3 Instruction Set Summary

This section describes instructions and addressing modes defined for the PowerPC 602 microprocessor. 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," and Section 2.3.4.5, "Trap Instructions."

- System linkage instructions—For more information, see Section 2.3.6.1, "System Linkage Instructions."

- Processor control instructions—These instructions are used for synchronizing memory accesses and managing caches, TLBs, and segment registers. For more information, see Sections 2.3.4.6, 2.3.5.1, and 2.3.6.2.

- Memory synchronization instructions—These instructions are used for memory synchronizing. See Sections 2.3.4.7 and 2.3.5.2 for more information.

- Memory control instructions—These instructions provide control of caches, TLBs, and segment registers. For more information, see Sections 2.3.5.3 and 2.3.6.3.

- External control instructions—These include instructions for use with special input/output devices. The optional external instructions (**eciwx** and **ecowx**) defined by the PowerPC architecture, are not implemented in the 602.

  For information about instructions specific to the 602, see Section 2.3.7, "PowerPC 602 Implementation-Specific 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 602's parallel instruction execution, is provided in Chapter 8, "Instruction Set," in *The Programming Environments Manual*.

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 4 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 (referred to as extended mnemonics in the architecture specification) 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 mnemonic examples.

2.3.1 Classes of Instructions

The 602 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, an instruction that is specific to 64-bit implementations is considered defined for 64-bit implementations but illegal for 32-bit implementations such as the 602.

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.

In future versions of the PowerPC architecture, instruction codings 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 602 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.

A defined instruction can have invalid forms, as described in the following subsection.

2.3.1.3 Illegal Instruction Class

Illegal instructions can be grouped into the following categories:

- Instructions that are not implemented in the PowerPC architecture. These opcodes are available for future extensions of the PowerPC architecture; that is, future versions of the PowerPC architecture may define any of these instructions to perform new functions.

  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

- Instructions that are implemented in the PowerPC architecture but are 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.

The following primary opcodes are defined for 64-bit implementations only and are illegal on the 602:

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 that are 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 entirely of 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. This is further described in Section 2.3.1.4, "Reserved Instruction Class."

An attempt to execute an illegal instruction invokes the illegal instruction error handler (a program exception) but has no other effect. See Section 4.5.7, "Program Exception (0x0700)," 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 Section 4.5.7, "Program Exception (0x0700)," for additional information about illegal and invalid instruction exceptions.

The following types of instructions are included in this class:

- Implementation-specific instructions (for example, TLB Load Data (**tlbld**) and TLB Load Instruction (**tlbli**) instructions)
- Optional instructions defined by the PowerPC architecture but not implemented by the 602 (for example, Floating Square Root (**fsqrt**) and Floating Square Root Single (**fsqrts**) instructions)

PowerPC 602 RISC Microprocessor User's Manual

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.

2.3.2.2 Memory Operands

Bytes in memory are numbered consecutively starting with zero. 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 instructions, a sequence of 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.

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 byte ordering and memory operands, see Chapter 3, "Operand Conventions," in *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 2.3.4.3.1, "Integer Load and Store Address Generation," for further discussion 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

Refer to Section 2.3.4.4.1, "Branch Instruction Address Calculation," for further discussion of branch instruction effective address generation.

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 PowerPC architecture defines the System Call (**sc**) and the Return From Interrupt (**rfi**) instructions to be context synchronizing. Exceptions (external, internal and taken traps) are also context synchronizing. Context synchronization requires the following:

- No higher priority exception exists.

- The instruction cannot complete until all previous instructions have completed to a point where they can no longer cause an exception.

- Instructions that precede this instruction complete in the context (including privilege, protection and translation) under which they were issued.

- The instruction following this instruction executes in the context established by this instruction.

In the 602, these instructions guarantee context synchronization by performing the following steps:

1. Working their way through the pipeline (clearing any previous instructions).

2. Performing the appropriate context updates while at the final stage of the pipeline.

3. Redirecting the instruction fetcher to refetch the instructions from the address specified by the instruction.

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 the Synchronize (**sync**) and Instruction Synchronize (**isync**) instructions, 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 602—those caused directly by the execution of an instruction and those caused by an asynchronous event. 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 602 provides the following supervisor-level instructions—**dcbi**, **mfmsr**, **mfspr**, **mfsr**, **mfsrin**, **mtmsr**, **mtspr**, **mtsr**, **mtsrin**, **rfi**, **tlbie**, **tlbsync, tlbld**, and **tlbli**. 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."

2.3.2.4.4 Self-Modifying Code Requirements

The following sequence of instructions will synchronize the instruction stream.

```
dcbst

sync

icbi

isync
```

2.3.3 Instruction Set Overview

This section provides a brief overview of the PowerPC instructions implemented in the 602 and highlights any special information with respect to how the 602 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 provided for the convenience of the programmer and do not necessarily reflect the PowerPC architecture specification.

Note that some of the instructions have the following optional features:

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

2.3.4 PowerPC UISA Instructions

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

2.3.4.1 Integer Instructions

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

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

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

2.3.4.1.1 Integer Arithmetic Instructions

Table 2-18 lists the integer arithmetic instructions for the 602.

Table 2-18. 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
Subtract from Immediate Carrying	**subfic**	**r**D,**r**A,SIMM

Table 2-18. Integer Arithmetic Instructions (Continued)

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

Although there is no Subtract Immediate instruction, its effect can be achieved by using an **addi** instruction with the immediate operand negated. Simplified mnemonics are provided that include this negation. The **subf** instructions subtract the second operand (**r**A) from the third operand (**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.

2.3.4.1.2 Integer Compare Instructions

The integer compare instructions algebraically or logically compare the contents of **r**A with either the UIMM operand, the SIMM operand, or the contents of **r**B. The comparison is signed for the **cmpi** and **cmp** instructions, and unsigned for the **cmpli** and **cmpl** instructions. Table 2-19 lists the integer compare instructions.

Table 2-19. Integer Compare Instructions

Name	Mnemonic	Operand Syntax
Compare Immediate	**cmpi**	**crf**D,L,**rA**,SIMM
Compare	**cmp**	**crf**D,L,**rA**,**rB**
Compare Logical Immediate	**cmpli**	**crf**D,L,**rA**,UIMM
Compare Logical	**cmpl**	**crf**D,L,**rA**,**rB**

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.

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 2-20 perform bit-parallel operations. Logical instructions with the CR update enabled and instructions **andi.** and **andis.** set CR field CR0 to characterize the result of the logical operation. These fields are set as if the sign-extended low-order 32 bits of the result were algebraically compared to zero. Logical instructions without CR update and the remaining logical instructions do not modify the CR. Logical instructions do not affect the XER[SO], XER[OV], and XER[CA] bits.

For simplified mnemonics examples for the integer logical operations see Appendix F, "Simplified Mnemonics," in *The Programming Environments Manual*.

Table 2-20. Integer Logical Instructions

Name	Mnemonic	Operand Syntax	602 Comments
AND Immediate	**andi.**	rA,rS,UIMM	—
AND Immediate Shifted	**andis.**	rA,rS,UIMM	—
OR Immediate	**ori**	rA,rS,UIMM	**ori** r0,r0,0 is the preferred form for the no-op instruction. This acts as a 'branch never' instruction in the 602 and is folded-out by the BPU.
OR Immediate Shifted	**oris**	rA,rS,UIMM	—
XOR Immediate	**xori**	rA,rS,UIMM	—
XOR Immediate Shifted	**xoris**	rA,rS,UIMM	—
AND	**and** (**and.**)	rA,rS,rB	—
OR	**or** (**or.**)	rA,rS,rB	—
XOR	**xor** (**xor.**)	rA,rS,rB	—
NAND	**nand** (**nand.**)	rA,rS,rB	—
NOR	**nor** (**nor.**)	rA,rS,rB	—
Equivalent	**eqv** (**eqv.**)	rA,rS,rB	—
AND with Complement	**andc** (**andc.**)	rA,rS,rB	—
OR with Complement	**orc** (**orc.**)	rA,rS,rB	—
Extend Sign Byte	**extsb** (**extsb.**)	rA,rS	—
Extend Sign Half Word	**extsh** (**extsh.**)	rA,rS	—
Count Leading Zeros Word	**cntlzw** (**cntlzw.**)	rA,rS	—

2.3.4.1.4 Integer Rotate and Shift Instructions

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

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

The integer rotate instructions are listed in Table 2-21.

Table 2-21. Integer Rotate Instructions

Name	Mnemonic	Operand Syntax
Rotate Left Word Immediate then AND with Mask	**rlwinm** (**rlwinm.**)	rA,rS,SH,MB,ME
Rotate Left Word then AND with Mask	**rlwnm** (**rlwnm.**)	rA,rS,rB,MB,ME
Rotate Left Word Immediate then Mask Insert	**rlwimi** (**rlwimi.**)	rA,rS,SH,MB,ME

The integer shift instructions perform left and right shift operations. 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 listed in Table 2-22.

Table 2-22. Integer Shift Instructions

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

2.3.4.2 Floating-Point Instructions

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

- Floating-point arithmetic instructions
- Floating-point multiply-add instructions
- Floating-point rounding and conversion instructions
- Floating-point compare instructions
- Floating-point status and control register instructions
- Floating-point move instructions
- Floating-point special instructions—Because the 602 hardware supports only single-precision operations, the smaller single-precision FPRs need status bits to recognize the following:
    - A valid floating-point operand in the hardware (rather than in a memory image)
    - An integer value moved from the FPSCR or generated by an **fctiwz** instruction. See Section 6.4.3, "Floating-Point Unit," for a complete description of these status bits.
- These status bits are implemented as SPR registers (SP and LT) and are accessed and reloaded using the **mfspr**/**mtspr** instructions. See Section 2.1.2.4.1, "Floating-Point Tag Registers (SP and LT)," for more information.

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

If the SP bit is set for all of its operands and those operands reside in the 602's 32-bit FPRs, a single-precision floating-point instruction executes in hardware. That is, by placing the result in the target hardware FPR, setting the associated SP bit, and clearing the associated LT bit. If any of the operands have their associated SP bits cleared, the instruction takes an emulation trap exception (0x1600).

All double-precision arithmetic instructions take an emulation trap exception.

The PowerPC architecture supports a floating-point system as defined in the IEEE 754 standard, but the 602 provides software support to conform with that standard. All floating-point operations conform to the IEEE 754 standard, unless the non-IEEE mode bit (FPSCR[NI]) is set, in which case the 602 is in nondenormalized mode. Floating-point modes are described in Section 2.3.4.2.2, "IEEE Mode (FPSCR[NI] = 0)," and Section 2.3.4.2.3, "Non-IEEE Mode (FPSCR[NI] = 1)."

2.3.4.2.1 Denormalized Number Support

The 602 hardware accepts denormalized numbers as operands; however, in IEEE-mode, when underflow exceptions are disabled, any underflow condition causes a trap to 0x1600 where emulation software produces the proper IEEE result. In non-IEEE mode (FPSCR[NI] = 1), the hardware underflows to zero instead of producing a denormalized number.

Some instructions, however, require the denormalized number to be preserved, such as when executing a Floating Move Register (**fmr)** or Floating Select (**fsel)** instruction, or when the sign bit of a denormalized operand is changed—as is the case when a Floating Absolute Value (**fabs)**, Floating Negative Absolute Value (**fnabs)**, or Floating Negate (**fneg)** instruction is executed. In such cases, the hardware produces the desired result.

2.3.4.2.2 IEEE Mode (FPSCR[NI] = 0)

When the processor is in full IEEE compatibility mode, the following conditions will cause a trap to 0x1600 where emulation code produces the proper IEEE results or conditions:

- Invalid operation exceptions when such exceptions are enabled (FPSCR[VE] = 1).
- Zero divide exceptions when such exceptions are enabled (FPSCR[ZE]).
- Underflow exceptions when such exceptions are disabled (FPSCR[UE] = 0).
- All double-precision operations or instructions having double-precision operands. The instructions that cause an emulation trap exception are listed in Section 6.8, "Instruction Latency Summary."
- For some instructions, the setting of an operand's SP or LT can generate an exception.

2.3.4.2.3 Non-IEEE Mode (FPSCR[NI] = 1)

The 602 supports a non-IEEE mode that is useful for time-critical operations where IEEE compliance is not useful. This mode is enabled by setting FPSCR[NI]. Table 2-23 describes the operation of non-IEEE mode. These results are always produced when FPSCR[NI] is set regardless of the settings of the exception enable bits in the FPSCR.

Table 2-23. Non-IEEE Mode Results

Result	Output
Divide by zero	± infinity
Invalid	QNaN for arithmetic or round-to-single-precision operations • Most-positive integer if convert-to-integer and positive overflow or positive infinity operand • Most-negative integer if convert-to-integer and negative overflow, negative infinity, or NaN operand
Overflow	± infinity when round-to-nearest • Format's largest representable finite number with sign of the intermediate result when round-towards-zero • Most negative number for negative overflow and +infinity for positive overflow when round-towards-positive-infinity • –infinity for negative overflow or largest finite number for positive overflow when round-towards-negative-infinity
Underflow	Zero

Note that, as defined by the IEEE model, the 602 presents a QNaN regardless of whether the input is an SNaN or a QNaN.

The exception enable bits in the FPSCR and the floating-point exception mode bits in the MSR determine whether an exception condition is generated when the results in Table 2-23 occur regardless of the setting of FPSCR[NI].

When traps are disabled, the IEEE and non-IEEE modes differ only with respect to how underflows are handled.

2.3.4.2.4 Time-Critical Floating-Point Operations

For time-critical applications, the FPSCR bits must be set such that the non-IEEE mode is enabled (FPSCR[NI] = 1) and all floating-point exceptions are disabled. With these settings, the 602 does not cause floating-point enabled program exceptions or generate denormalized numbers, either of which would slow performance.

When the 602 is in non-IEEE mode, all floating-point operations should involve only single-precision operands (or integer operands, in a few instructions). See Section 6.8, "Instruction Latency Summary," for instructions that trap to 0x1600; such instructions should be avoided in time-critical operations.

See Section 2.2, "Operand Conventions," for more information about the 602 support for the nondenormalized mode.

2.3.4.2.5 Floating-Point Arithmetic Instructions

The floating-point arithmetic instructions are listed in Table 2-24.

Table 2-24. Floating-Point Arithmetic Instructions

Name	Mnemonic	Operand Syntax	602 Notes
Floating Add (Double-Precision)	**fadd** (**fadd.**)	**fr**D,**fr**A,**fr**B	This instruction is not supported in hardware on the 602; causes an emulation trap exception (0x1600).
Floating Add Single	**fadds** (**fadds.**)	**fr**D,**fr**A,**fr**B	If the SP bits are set for source operands, the instruction is executed as defined by the architecture; otherwise, it takes an emulation trap exception.
Floating Subtract (Double-Precision)	**fsub** (**fsub.**)	**fr**D,**fr**A,**fr**B	This instruction is not supported in hardware on the 602; causes an emulation trap exception (0x1600).
Floating Subtract Single	**fsubs** (**fsubs.**)	**fr**D,**fr**A,**fr**B	If the SP bits are set for source operands, the instruction is executed as defined by the architecture; otherwise, it takes an emulation trap exception.
Floating Multiply (Double-Precision)	**fmul** (**fmul.**)	**fr**D,**fr**A,**fr**C	This instruction is not supported in hardware on the 602; causes an emulation trap exception (0x1600).
Floating Multiply Single	**fmuls** (**fmuls.**)	**fr**D,**fr**A,**fr**C	If the SP bits are set for source operands, the instruction is executed as defined by the architecture; otherwise, it takes an emulation trap exception.

Table 2-24. Floating-Point Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax	602 Notes
Floating Divide (Double-Precision)	**fdiv** (**fdiv.**)	**fr**D,**fr**A,**fr**B	This instruction is not supported in hardware on the 602; causes an emulation trap exception (0x1600).
Floating Divide Single	**fdivs** (**fdivs.**)	**fr**D,**fr**A,**fr**B	If the SP bits are set for source operands, the instruction is executed as defined by the architecture; otherwise, it takes an emulation trap exception.
Floating Reciprocal Estimate Single	**fres** (**fres.**)	**fr**D,**fr**B	This instruction is implemented as a single-precision divide instruction; it is not an estimate.
Floating Reciprocal Square Root Estimate	**frsqrte** (**frsqrte.**)	**fr**D,**fr**B	The estimate is accurate to 1 part in 32 of the reciprocal of the square root of **fr**B. The target operand is single-precision if it is in a hardware FPR and its SP bit is set. Otherwise, the instruction traps to 0x1600.
Floating Select	**fsel**	**fr**D,**fr**A,**fr**C,**fr**B	Traps if the SP bit associated with **fr**A is "OFF" or if the SP bits is cleared for the selected **fr**B or **fr**C.

2.3.4.2.6 Floating-Point Multiply-Add Instructions

These instructions combine multiply and add operations without an intermediate rounding operation. The fractional part of the intermediate product is 48 bits wide for single-precision values and 106 bits wide for double-precision values. All intermediate fractional bits take part in the add/subtract portion of the instruction. Note that double-precision instructions take an emulation trap exception.

The floating-point multiply-add instructions are listed in Table 2-25.

Table 2-25. Floating-Point Multiply-Add Instructions

Name	Mnemonic	Operand Syntax	602 Notes
Floating Multiply-Add (Double-Precision)	**fmadd** (**fmadd.**)	**fr**D,**fr**A,**fr**C,**fr**B	This instruction is not supported in hardware on the 602; causes an emulation trap exception (0x1600).
Floating Multiply-Add Single	**fmadds** (**fmadds.**)	**fr**D,**fr**A,**fr**C,**fr**B	If the SP bits are set for source operands, the instruction is executed as defined by the architecture; otherwise, it takes an emulation trap exception.
Floating Multiply-Subtract (Double-Precision)	**fmsub** (**fmsub.**)	**fr**D,**fr**A,**fr**C,**fr**B	This instruction is not supported in hardware on the 602; causes an emulation trap exception (0x1600).
Floating Multiply-Subtract Single	**fmsubs** (**fmsubs.**)	**fr**D,**fr**A,**fr**C,**fr**B	If the SP bits are set for source operands, the instruction is executed as defined by the architecture; otherwise, it takes an emulation trap exception.

Table 2-25. Floating-Point Multiply-Add Instructions (Continued)

Name	Mnemonic	Operand Syntax	602 Notes
Floating Negative Multiply-Add (Double-Precision)	**fnmadd** (**fnmadd.**)	**fr**D,**fr**A,**fr**C,**fr**B	This instruction is not supported in hardware on the 602; causes an emulation trap exception (0x1600).
Floating Negative Multiply-Add Single	**fnmadds** (**fnmadds.**)	**fr**D,**fr**A,**fr**C,**fr**B	If the SP bits are set for source operands, the instruction is executed as defined by the architecture; otherwise, it takes an emulation trap exception.
Floating Negative Multiply-Subtract (Double-Precision)	**fnmsub** (**fnmsub.**)	**fr**D,**fr**A,**fr**C,**fr**B	This instruction is not supported in hardware on the 602; causes an emulation trap exception (0x1600).
Floating Negative Multiply-Subtract Single	**fnmsubs** (**fnmsubs**)	**fr**D,**fr**A,**fr**C,**fr**B	If the SP bits are set for source operands, the instruction is executed as defined by the architecture; otherwise, it takes an emulation trap exception.

2.3.4.2.7 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 conversion instructions convert a 64-bit double-precision floating-point number to a 32-bit signed integer number.

On the 602, if the operand resides in the hardware FPR and has its SP bit "ON", the **frsp** instruction simply moves the data from the source FPR to the target FPR, and sets the corresponding SP and LT bits to 1 and 0 respectively. If the data resides in the emulated, double-precision FPR or is an integer value in the hardware FPR, the instruction traps to the emulation trap exception vector and is emulated. If the operand is a denormalized number, an underflow condition occurs and the data is manipulated as required by the mode of operation and by the setting of FPSCR[UE].

The PowerPC architecture defines bits 0–31 of floating-point register **fr**D as undefined when executing the Floating Convert to Integer Word (**fctiw**) and Floating Convert to Integer Word with Round toward Zero (**fctiwz**) instructions. Note that of the convert-to-integer instructions, only the **fctiwz** instruction is supported in hardware in the 602. The **fctiw** instruction causes an emulation trap exception. Executing the **fctiwz** instruction produces an integer in bits 0–31 of the target register. The target is also flagged as an integer by loading its associated SP and LT bits with 0 and 1, respectively.

Examples of uses of these instructions to perform various conversions can be found in Appendix D, "Floating-Point Models," in *The Programming Environments Manual*. The floating-point rounding instructions are shown in Table 2-26.

Table 2-26. Floating-Point Rounding and Conversion Instructions

Name	Mnemonic	Operand Syntax	602 Notes
Floating Round to Single-Precision	**frsp** (**frsp.**)	**fr**D,**fr**B	—
Floating Convert to Integer Word	**fctiw** (**fctiw.**)	**fr**D,**fr**B	This instruction is not supported in hardware on the 602; causes an emulation trap exception (0x1600).
Floating Convert to Integer Word with Round toward Zero	**fctiwz** (**fctiwz.**)	**fr**D,**fr**B	If the operand's associated SP bit is set, the instruction produces an integer in the target hardware register (**fr**D) with its associated SP‖LT set to 0b01; otherwise it takes an emulation trap exception. This instruction is dispatch serialized.

2.3.4.2.8 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 listed in Table 2-27.

Table 2-27. Floating-Point Compare Instructions

Name	Mnemonic	Operand Syntax	602 Notes
Floating Compare Unordered	fcmpu	**crf**D,**fr**A,**fr**B	If the SP bits are set for both operands, the instruction is executed as defined by the architecture; otherwise, it takes an emulation trap exception.
Floating Compare Ordered	fcmpo	**crf**D,**fr**A,**fr**B	If the SP bits are set for both operands, the instruction is executed as defined by the architecture; otherwise, it takes an emulation trap exception.

2.3.4.2.9 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 listed in Table 2-28.

Table 2-28. Floating-Point Status and Control Register Instructions

Name	Mnemonic	Operand Syntax	602 Notes
Move from FPSCR	**mffs** (**mffs.**)	**fr**D	This instruction is implemented using the 32-bit hardware FPRs. For the **mffs** instruction, the target is a 32-bit FPR (all bits 0–31), the corresponding SP bit is cleared, and the corresponding LT bit is set.
Move to CR 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	This instruction is implemented using the 32-bit hardware FPRs. The **fr**B operand is a 32-bit FPR (bits 0–31) masked as long as **fr**B has its associated LT bit set; otherwise, the instruction is trapped to 0x1600.
Move to FPSCR Bit 0	**mtfsb0** (**mtfsb0.**)	**crb**D	—
Move to FPSCR Bit 1	**mtfsb1** (**mtfsb1.**)	**crb**D	—

2.3.4.2.10 Floating-Point Move Instructions

Floating-point move instructions copy data from one floating-point register 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. Floating-point move instructions are listed in Table 2-28.

Table 2-29. Floating-Point Move Instructions

Name	Mnemonic	Operand Syntax	602 Notes
Floating Move Register	**fmr** (**fmr.**)	**fr**D,**fr**B	If the SP bit is set for **fr**B, the instruction is executed as defined by the architecture; otherwise, it takes an emulation trap exception.
Floating Negate	**fneg** (**fneg.**)	**fr**D,**fr**B	If the SP bit is set for **fr**B, the instruction is executed as defined by the architecture; otherwise, it takes an emulation trap exception.
Floating Absolute Value	**fabs** (**fabs.**)	**fr**D,**fr**B	If the SP bit is set for **fr**B, the instruction is executed as defined by the architecture; otherwise, it takes an emulation trap exception.
Floating Negative Absolute Value	**fnabs** (**fnabs.**)	**fr**D,**fr**B	If the SP bit is set for **fr**B, the instruction is executed as defined by the architecture; otherwise, it takes an emulation trap exception.

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 of the 602, 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

2.3.4.3.1 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 the 602 is optimized for load and store operations that are aligned on natural boundaries, and operations that are not naturally aligned may suffer performance degradation. Refer to Section 4.5.6.1, "Integer Alignment Exceptions," for additional information about load and store address alignment exceptions.

2.3.4.3.2 Register Indirect Integer Load Instructions

For integer load instructions, the byte, half word, word, or double word addressed by the EA 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, the EA is placed into **r**A and the memory element (byte, half word, word, or double word) addressed by EA is loaded into **r**D.

Note that in some implementations of the architecture, the load word algebraic instructions (**lha** and **lhax**) and the load with update (**lbzu**, **lbzux**, **lhzu**, **lhzux**, **lhau**, and **lhaux**) instructions may execute with greater latency than other types of load instructions. The load with update instructions may take longer to execute in some implementations than the corresponding pair of a nonupdate load followed by an **add**x instruction. In the 602, these instructions operate with the same latency as other load instructions.

Table 2-30 lists the integer load instructions.

Table 2-30. Integer Load Instructions

Name	Mnemonic	Operand Syntax
Load Byte and Zero	**lbz**	rD,d**(rA)**
Load Byte and Zero Indexed	**lbzx**	rD,rA,rB
Load Byte and Zero with Update	**lbzu**	rD,d**(rA)**
Load Byte and Zero with Update Indexed	**lbzux**	rD,rA,rB
Load Half Word and Zero	**lhz**	rD,d**(rA)**
Load Half Word and Zero Indexed	**lhzx**	rD,rA,rB
Load Half Word and Zero with Update	**lhzu**	rD,d**(rA)**
Load Half Word and Zero with Update Indexed	**lhzux**	rD,rA,rB
Load Half Word Algebraic	**lha**	rD,d**(rA)**
Load Half Word Algebraic Indexed	**lhax**	rD,rA,rB
Load Half Word Algebraic with Update	**lhau**	rD,d**(rA)**
Load Half Word Algebraic with Update Indexed	**lhaux**	rD,rA,rB
Load Word and Zero	**lwz**	rD,d**(rA)**
Load Word and Zero Indexed	**lwzx**	rD,rA,rB
Load Word and Zero with Update	**lwzu**	rD,d**(rA)**
Load Word and Zero with Update Indexed	**lwzux**	rD,rA,rB

Note that the PowerPC architecture cautions programmers that some implementations of the architecture may run the **lha**, **lhax**, **lbzu**, **lbzux**, **lhzu**, **lhzux**, **lhau**, and **lhaux**, instructions with greater latency than other types of load instructions. This is not the case in the 602; these instructions have the same latency as other load instructions.

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

The 602 defines store with update instructions with **r**A = 0 and integer store instructions with the CR update option enabled (Rc field, bit 31, in the instruction encoding = 1) to be invalid forms. Table 2-31 provides a list of the integer store instructions for the 602.

Table 2-31. Integer Store Instructions

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

2.3.4.3.4 Integer Load and Store with Byte-Reverse Instructions

Table 2-32 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 "Byte Ordering" in Chapter 3, "Operand Conventions," in *The Programming Environments Manual*.

Note that the PowerPC architecture cautions programmers that in some PowerPC implementations, load byte-reverse instructions (**lhbrx** and **lwbrx**) may have greater latency than other load instructions; however, these instructions operate with the same latency as other load instructions in the 602.

Table 2-32. Integer Load and Store with Byte-Reverse Instructions

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

2.3.4.3.5 Integer Load and Store Multiple Instructions

The integer load and 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 602 implementation of the load and store multiple instructions:

- 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. In this case, the 602 performs some or all of the memory references from the first page, and none of the memory references from the second page before taking the exception. On return from the DSI exception, the load or store multiple instruction will re-execute from the beginning. For additional information, refer to "DSI Exception (0x0300)" in Chapter 6, "Exceptions," in *The Programming Environments Manual*.

- For the 602, there are no preferred forms for load and store multiple instructions.

- In some PowerPC processors, 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.

- 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. It defines the load multiple and store multiple instructions with misaligned operands (that is, the EA is not a multiple of 4) to be an invalid form. The 602 defines the load multiple word (**lmw**) instruction with **r**A in the range of registers to be loaded as an invalid form.

Table 2-33 lists the integer load and store multiple instructions for the 602.

Table 2-33. Integer Load and Store Multiple Instructions

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

2.3.4.3.6 Integer Load and Store String Instructions

The integer load and store string instructions are defined by the architecture to allow movement of data from memory to registers or from registers to memory. When the 602 encounters a load or store string instruction, an emulation trap exception is taken.

Table 2-34 lists the integer load and store string instructions.

Table 2-34. Integer Load and Store String Instructions

Name	Mnemonic	Operand Syntax	602 Notes
Load String Word Immediate	**lswi**	**r**D,**r**A,NB	This instruction is not supported in hardware on the 602; causes an emulation trap exception (0x1600).
Load String Word Indexed	**lswx**	**r**D,**r**A,**r**B	This instruction is not supported in hardware on the 602; causes an emulation trap exception (0x1600).
Store String Word Immediate	**stswi**	**r**S,**r**A,NB	This instruction is not supported in hardware on the 602; causes an emulation trap exception (0x1600).
Store String Word Indexed	**stswx**	**r**S,**r**A,**r**B	This instruction is not supported in hardware on the 602; causes an emulation trap exception (0x1600).

If **r**A is in the range of registers to be loaded for an **lswi** instruction or if either **r**A or **r**B is in the range of registers to be loaded for an **lswx** instruction, the PowerPC architecture defines the instruction to be of an invalid form. In addition, the **lswx** and **stswx** instructions that specify a string length of zero are defined to be invalid by the PowerPC architecture.

2.3.4.3.7 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, the details of which are described in the following sections.

2.3.4.3.8 Floating-Point Load Instructions

There are two forms of the floating-point load instruction—single-precision and double-precision operand formats. Note that the PowerPC architecture defines load with update instructions with **r**A = 0 as an invalid form.

Table 2-35 provides a list of the floating-point load instructions.

Table 2-35. Floating-Point Load Instructions

Name	Mnemonic	Operand Syntax	602 Notes
Load Floating-Point Single	**lfs**	**fr**D,d(**r**A)	The 602 loads the single-precision operand as a single-precision value into the hardware FPR and sets its corresponding SP and LT bits to 1 and 0, respectively.
Load Floating-Point Single Indexed	**lfsx**	**fr**D,**r**A,**r**B	The 602 loads the single-precision operand as a single-precision value into the hardware FPR and sets its corresponding SP and LT bits to 1 and 0, respectively.
Load Floating-Point Single with Update	**lfsu**	**fr**D,d(**r**A)	The 602 loads the single-precision operand as a single-precision value into the hardware FPR and sets its corresponding SP and LT bits to 1 and 0, respectively.
Load Floating-Point Single with Update Indexed	**lfsux**	**fr**D,**r**A,**r**B	The 602 loads the single-precision operand as a single-precision value into the hardware FPR and sets its corresponding SP and LT bits to 1 and 0, respectively.

Table 2-35. Floating-Point Load Instructions (Continued)

Name	Mnemonic	Operand Syntax	602 Notes
Load Floating-Point Double	**lfd**	**fr**D,d(**r**A)	If the 64-bit operand fits in single-precision format and is not a NaN, infinity, or single-precision format denormalized number, the operand is compressed to single-precision format and placed in the hardware FPR with its corresponding SP and LT bits set to 1 and 0, respectively. Otherwise, **lfd** takes an emulation trap exception.
Load Floating-Point Double Indexed	**lfdx**	**fr**D,**r**A,**r**B	If the 64-bit operand fits in single-precision format and is not a NaN, infinity, or single-precision format denormalized number, the operand is compressed to single-precision format and placed in the hardware FPR with its corresponding SP and LT bits set to 1 and 0, respectively. Otherwise, **lfdx** takes an emulation trap exception.
Load Floating-Point Double with Update	**lfdu**	**fr**D,d(**r**A)	If the 64-bit operand fits in single-precision format and is not a NaN, infinity, or single-precision format denormalized number, the operand is compressed to single-precision format and placed in the hardware FPR with its corresponding SP and LT bits set to 1 and 0, respectively. Otherwise, **lfdu** takes an emulation trap exception.
Load Floating-Point Double with Update Indexed	**lfdux**	**fr**D,**r**A,**r**B	If the 64-bit operand fits in single-precision format and is not a NaN, infinity, or single-precision format denormalized number, the operand is compressed to single-precision format and placed in the hardware FPR with its corresponding SP and LT bits set to 1 and 0, respectively. Otherwise, **lfdux** takes an emulation trap exception.

Note that the 602 performs the **lfs**, **lfsx**, **lfsu**, and **lfsux** instructions by saving the single-precision operand as a single-precision value in the 32-bit hardware target FPR.

2.3.4.3.9 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** (Store Floating-Point as Integer Word Indexed) instruction.

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

Table 2-36 provides a list of the floating-point store instructions.

Table 2-36. Floating-Point Store Instructions

Name	Mnemonic	Operand Syntax	602 Notes
Store Floating-Point Single	**stfs**	**fr**S,d(**r**A)	If the operand's SP bit is set, the operand is copied directly from the hardware FPR to memory; otherwise, the instruction takes an emulation trap exception.
Store Floating-Point Single Indexed	**stfsx**	**fr**S,**r**A,**r**B	If the operand's SP bit is set, the operand is copied directly from the hardware FPR to memory; otherwise, the instruction takes an emulation trap exception.
Store Floating-Point Single with Update	**stfsu**	**fr**S,d(**r**A)	If the operand's SP bit is set, the operand is copied directly from the hardware FPR to memory; otherwise, the instruction takes an emulation trap exception.

Table 2-36. Floating-Point Store Instructions (Continued)

Name	Mnemonic	Operand Syntax	602 Notes
Store Floating-Point Single with Update Indexed	**stfsux**	**frS,rA,rB**	If the operand's SP bit is set, the operand is copied directly from the hardware FPR to memory; otherwise, the instruction takes an emulation trap exception.
Store Floating-Point Double	**stfd**	**frS,d(rA)**	If the operand's SP bit is set and the operand is not a NaN, infinity, or denormalized number, it is expanded to the double-precision format and stored; otherwise, the instruction takes an emulation trap exception.
Store Floating-Point Double Indexed	**stfdx**	**frS,rA,rB**	If the operand's SP bit is set and the operand is not a NaN, infinity, or denormalized number, it is expanded to the double-precision format and stored; otherwise, the instruction takes an emulation trap exception.
Store Floating-Point Double with Update	**stfdu**	**frS,d(rA)**	If the operand's SP bit is set and the operand is not a NaN, infinity, or denormalized number, it is expanded to the double-precision format and stored; otherwise, the instruction takes an emulation trap exception.
Store Floating-Point Double with Update Indexed	**stfdux**	**frS,rA,rB**	If the operand's SP bit is set and the operand is not a NaN, infinity, or denormalized number, it is expanded to the double-precision format and stored; otherwise, the instruction takes an emulation trap exception.
Store Floating-Point as Integer Word Indexed	**stfiwx**	**frS,rA,rB**	If the operand's LT bit is set, the value is stored directly; otherwise, an emulation trap exception is taken.

2.3.4.4 Branch and Flow Control Instructions

Branch instructions are executed by the branch processing unit (BPU). The BPU receives branch instructions from the fetch unit and performs condition register (CR) lookahead operations on conditional branches to resolve them early, achieving the effect of a zero-cycle branch in many cases.

Some branch instructions can redirect instruction execution conditionally based on the value of bits in the CR. When the branch 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.

If an interlock is detected, the branch is considered unresolved and the direction of the branch is predicted using static branch prediction as described in "Conditional Branch Control" in Chapter 4, "Addressing Modes and Instruction Set Summary," in *The Programming Environments Manual*. The interlock is monitored while instructions are fetched for the predicted branch. When the interlock is cleared, the branch processor determines whether the prediction was correct based on the value of the CR bit. If the prediction is correct, the branch is considered completed and instruction fetching continues. If the prediction is incorrect, the fetched instructions are purged, and instruction fetching

continues along the alternate path. See Chapter 8, "Instruction Timing," in *The Programming Environments Manual* for more information about how branches are executed.

Note that when the 602 predicts a branch path, prefetching is allowed only from the internal instruction cache. Prefetching from external memory is blocked until the branch instruction resolves.

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 processor ignores 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

2.3.4.4.2 Branch Instructions

Table 2-37 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 2-37. 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 2-38, and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions, although they are executed by the system register unit (SRU). Most instructions executed by the SRU are completion-serialized to maintain system state; that is, the instruction is held for execution in the SRU until all prior instructions issued have completed.

Table 2-38. 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, these forms of the instructions are invalid in the 602.

2.3.4.5 Trap Instructions

The trap instructions shown in Table 2-39 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 2-39. Trap Instructions

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

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

2.3.4.6 Processor Control Instructions

Processor control instructions are used to read from and write to the condition register (CR), machine state register (MSR), and special-purpose registers (SPRs), and to read from the time base register (TBU or TBL).

2.3.4.6.1 Move to/from Condition Register Instructions

Table 2-46 lists the instructions provided by the 602 for reading from or writing to the CR.

Table 2-40. Move to/from Condition Register Instructions

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

Note that the PowerPC architecture cautions programmers that in some implementations, the **mtcrf** instruction may perform more slowly when only a portion of the fields are updated as opposed to all of the fields. This is not the case in the 602.

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, "Instruction and Data Cache Operation," for additional information about these instructions and about related aspects of memory synchronization.

The **sync** instruction delays execution of subsequent instructions until previous instructions have completed to the point that they can no longer cause an exception and until all previous memory accesses are performed globally; the **sync** operation is not broadcast onto the 602 bus interface. Additionally, all load and store cache/bus activities initiated by prior instructions are completed. Touch load operations (**dcbt** and **dcbtst**) are required to complete at least through address translation, but not required to complete on the bus.

The functions performed by the **sync** instruction normally takes a significant amount of time to complete. Because the latency of the **sync** instruction depends on the state of the processor when the instruction is issued as well as various system-level factors, frequent use of this instruction may cause some performance degradation.

The 602 treats the Enforce In-Order Execution of I/O (**eieio**) instruction as a no-op, since it enforces that all loads and stores to caching-inhibited memory and stores to write-through memory execute in order on the external bus.

The proper paired use of the **lwarx** and **stwcx.** instructions allows programmers to emulate common semaphore operations such as "test and set," "compare and swap," "exchange memory," and "fetch and add." Examples of these semaphore operations can be found in Appendix E, "Synchronization Programming Examples," in *The Programming Environments Manual*. 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 32 bytes.

The concept behind the use of the **lwarx** and **stwcx.** instructions is that a processor may load a semaphore from memory, compute a result based on the value of the semaphore, and conditionally store it back to the same location (only if that location has not been modified since it was first read), and determine if the store was successful. The conditional store is performed based upon the existence of a reservation established by the preceding **lwarx** instruction. If the reservation exists when the store is executed, the store is performed and a bit is set in the CR. If the reservation does not exist when the store is executed, the target memory location is not modified and a bit is cleared in the CR.

If the store was successful, the sequence of instructions from the read of the semaphore to the store that updated the semaphore appear to have been executed atomically (that is, no other processor or mechanism modified the semaphore location between the read and the update), thus providing the equivalent of a real atomic operation. However, in reality, other processors may have read from the location during this operation. In the 602, the reservations are made on behalf of aligned 32-byte sections of the memory address space.

The **lwarx** and **stwcx.** instructions require the EA to be aligned. Exception handling software should not attempt to emulate a misaligned **lwarx** or **stwcx.** instruction, because there is no correct way to define the address associated with the reservation.

In general, the **lwarx** and **stwcx.** instructions should be used only in system programs, which can be invoked by application programs as needed.

At most, one reservation exists simultaneously on any processor. The address associated with the reservation can be changed by a subsequent **lwarx** instruction. The conditional store is performed based upon the existence of a reservation established by the preceding **lwarx** regardless of whether the address generated by the **lwarx** matches that generated by the **stwcx.** instruction. A reservation held by the processor is cleared by one of the following:

- Executing an **stwcx.** instruction to any address
- Attempt by some other device to modify a location in the reservation granularity (32 bytes)

The **lwarx** and **stwcx.** instructions in write-through access mode do not cause a DSI exception.

The **stwcx.** instruction always broadcasts on the external bus and thus operates with slightly less performance characteristics as compared to normal store operations.

Table 2-41 lists the UISA memory synchronization instructions for the 602.

Table 2-41. Memory Synchronization Instructions—UISA

Name	Mnemonic	Operand Syntax	602 Notes
Load Word and Reserve Indexed	**lwarx**	rD,rA,rB	—
Store Word Conditional Indexed	**stwcx.**	rS,rA,rB	—
Synchronize	**sync**	—	This instruction delays subsequent bus activity until previous instructions and bus operations (except queued touch load operations and instruction fetches) have completed.

In the 602, reservations for **lwarx** and **stwcx.** instructions are made on behalf of aligned 32-byte sections of the memory address space. Using these instructions when the 602 is in write-through mode does not cause a DSI exception. Because the **stwcx.** instruction is broadcast on the external bus, it typically does not perform as efficiently as normal store operations.

2.3.4.8 Preferred No-Op Instruction

The PowerPC architecture defines the instruction "**ori** r0,r0,0" as the preferred form for the no-op instruction. This preferred form acts as a 'branch never' instruction in the 602 and is folded out by the BPU.

2.3.5 PowerPC VEA Instructions

The PowerPC 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.

2.3.5.1 Processor Control Instructions

In addition to the move to condition register instructions specified by the UISA, the VEA defines the Move from Time Base (**mftb**) instruction for reading the contents of the time base register. The **mftb** is a user-level instruction; it is shown in Table 2-42.

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. 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. Simplified mnemonics are also provided for Move from Time Base Upper (**mftbu**), which is a variant of the **mftb** instruction rather than of **mfspr**. For more information, refer to "Simplified Mnemonics for Special Purpose Registers," in Appendix F, "Simplified Mnemonics," in *The Programming Environments Manual*.

Table 2-42. Move from Time Base Instruction

Name	Mnemonic	Operand Syntax	602 Notes
Move from Time Base	**mftb**	rD, TBR	The 602 time base is incremented every four bus clocks. The time base enable (TBEN) signal enables the count. The 602 ignores bit 25 of **mftb** and treats it like an **mfspr** instruction.

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, "Instruction and Data Cache Operation," for additional information about these instructions and about related aspects of memory synchronization.

Implementation Notes—The following list describes how the 602 handles memory synchronization in the VEA.

- The Instruction Synchronize (**isync**) instruction causes a refetch serialization, which waits for all prior instructions to complete and then executes the next sequential instruction. Execution of subsequent instructions is held until all previous instructions have completed until they can no longer cause an exception and all store queues have completed translation. Any instruction after an **isync** see all effects of prior instructions

- The Enforce In-Order Execution of I/O (**eieio**) instruction is used to ensure memory reordering. Since the 602 does not reorder noncacheable memory accesses, the **eieio** instruction is treated as a no-op.

Table 2-41 lists the VEA memory synchronization instructions for the 602.

Table 2-43. Memory Synchronization Instructions—VEA

Name	Mnemonic	Operand Syntax	602 Notes
Enforce In-Order Execution of I/O	**eieio**	—	The 602 does not reorder noncacheable memory accesses; therefore, **eieio** is treated as a no-op.
Instruction Synchronize	**isync**	—	—

2.3.5.3 Memory Control Instructions—VEA

The memory control instructions defined by the VEA provide user-level programs the ability to manage on-chip caches when they exist. Memory control instructions include the following types:

- Cache control instructions
- Segment register manipulation instructions
- Translation lookaside buffer management instructions

User-level cache instructions are listed in Table 2-44. See Section 2.3.6.3, "Memory Control Instructions—OEA," for information on supervisor-level cache, segment register manipulation, and TLB management instructions.

Note that the 602 interprets cache control instructions (**icbi**, **dcbi**, **dcbf**, and **dcbst**) as if they pertain only to the 602 cache and does not broadcast these instructions The **dcbz** instruction is broadcast to maintain coherency. Any bus activity caused by these instructions results from the operation on the 602 cache.

As with other memory-related instructions, the effect of the cache management instructions on memory is 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.

Table 2-44 lists the cache instructions that are accessible to user-level programs.

Table 2-44. User-Level Cache Instructions

Name	Mnemonic	Operand Syntax	602 Notes
Data Cache Block Touch	**dcbt**	**rA,rB**	The EA is computed, translated, and checked for protection violations. • If the EA violates page protection or misses in the MMU, the operation is a no-op. • If the address hits in the cache, no action is taken. • If the address misses in the cache and the tag is in the modified (M) state, the cache block is written back to memory and the new cache block is brought in and placed in the exclusive (E) state. • If the address misses in the cache and tag is in the E state, the new cache block is brought in and placed in the E state. Address-only transfers are not generated on the external bus (other cache operations).
Data Cache Block Touch for Store	**dcbtst**	**rA,rB**	This instruction is treated like a **dcbt** instruction with respect to the MEI cache coherency protocol.

Table 2-44. User-Level Cache Instructions (Continued)

Name	Mnemonic	Operand Syntax	602 Notes
Data Cache Block Set to Zero	**dcbz**	**r**A,**r**B	The EA is computed, translated, and checked for protection violations. If the EA hits in the cache, zeros are burst into the cache. Also, if M = 1 (coherency enforced), the address is broadcast onto the bus before the zero-line-fill operation. The **dcbz** instruction is the only cache instruction that the 602 broadcasts on the bus. The exception priorities (from highest to lowest) for **dcbz** are as follows: 1 BAT protection violation—DSI exception 2 MMU miss—DTLB exception 3 Cache disabled—Alignment exception 4 Page marked write-through or caching-inhibited—Alignment exception 5 TLB protection violation—DSI exception When data address translation is disabled (MSR[DR] = 0), the **dcbz** instruction allocates a cache block but may not verify that the physical address is valid. If a cache block is created for an invalid physical address, a machine check condition may result when an attempt is made to write that cache block back to memory. The cache block could be written back as a result of the execution of an instruction that causes a cache miss and the invalid addressed cache block is the target for replacement or a **dcbst** instruction.
Data Cache Block Store	**dcbst**	**r**A,**r**B	The EA is computed, translated, and checked for protection violations. • If the address hits in the cache and the tag is in the E state, no further action is taken. • If the address hits in the cache and is in the M state, the cache block is written back to memory and the cache block is put in the E state. Address-only transfers are not generated on the external bus. The exception priorities (from highest to lowest) are as follows: 1 BAT protection violation—DSI exception 2 MMU miss—DTLB exception 3 TLB protection violation—DSI exception
Data Cache Block Flush	**dcbf**	**r**A,**r**B	The EA is computed, translated, and checked for protection violations. • If the address hits in the cache and the block is marked M, the block is written back to memory and the cache entry is invalidated. • If the address hits in the cache, and the cache block is marked E, the cache entry is invalidated. • If the address misses in the cache, no further action is taken. The exception priorities (from highest to lowest) are as follows: 1 BAT protection violation—DSI exception 2 MMU miss—DTLB exception 3 TLB protection violation—DSI exception
Instruction Cache Block Invalidate	**icbi**	**r**A,**r**B	The **icbi** instruction performs a virtual lookup into the instruction cache (index only). The address is not translated and cannot generate an exception. Both ways of the selected set are invalidated. This instruction is not broadcast onto the external bus.

2.3.5.4 External Control Instructions

The optional external control instructions, **eciwx** and **ecowx**, defined by the PowerPC architecture are not implemented in the 602.

2.3.6 PowerPC OEA Instructions

The PowerPC OEA includes the structure of the memory management model, supervisor-level registers, and the exception model.

2.3.6.1 System Linkage Instructions

This section describes the system linkage instructions (see Table 2-45). 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 Return from Interrupt (**rfi**) instruction is a supervisor-level instruction used for returning from an exception handler.

Table 2-45. System Linkage Instructions

Name	Mnemonic	Operand Syntax	602 Notes
System Call	**sc**	—	—
Return from Interrupt	**rfi**	—	—
Enable Supervisor Access	**esa**	—	602-specific. Provides the entry point for supervisor access. The **esa** instruction is a nonserialized instruction that saves MSR[SA, EE, PR, AP] to the ESASRR and sets appropriate MSR bits (SA = 1, EE = 0, PR = 0, AP = 0). For more information, see Section 2.3.9, "Using the esa Instruction for Supervisor-Level Access."
Disable Supervisor Access	**dsa**	—	602-specific. Provides the exit point for supervisor access. The **dsa** instruction is a nonserialized instruction that restores MSR[SA, EE, PR, AP] from the ESASRR to the MSR. For more information, see Section 2.3.9, "Using the esa Instruction for Supervisor-Level Access."

2.3.6.2 Processor Control Instructions—OEA

Processor control instructions are used to read from and write to the condition register (CR), machine state register (MSR), and special-purpose registers (SPRs), and to read from the time base register (TBU or TBL).

2.3.6.2.1 Move to/from Machine State Register Instructions

Table 2-46 lists the instructions provided by the 602 for reading from or writing to the MSR.

Table 2-46. Move to/from Machine State Register Instructions

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

2.3.6.2.2 Move to/from Special-Purpose Register Instructions

Simplified mnemonics are provided for the **mtspr** and **mfspr** instructions so they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand. See Appendix F, "Simplified Mnemonics," in *The Programming Environments Manual* for

simplified mnemonic examples. The **mtspr** and **mfspr** instructions are shown in Table 2-47.

Table 2-47. Move to/from Special-Purpose Register Instructions

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

The following describes exception conditions associated with the **mtspr** and **mfspr** instructions:

- The 602 treats **mtspr** and **mfspr** instructions that specify SPRs defined for POWER and not for PowerPC as illegal.
- Any **mtspr** or **mfspr** instruction that references privileged SPRs while not in the supervisor mode (MSR[PR]=1) causes a privileged instruction type program exception.
- Any **mtspr** instruction with an invalid SPR causes a program exception.
- Any **mfspr** with an invalid SPR causes an illegal opcode type program exception.

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 encoding, with the high-order 5 bits appearing in bits 16–20 of the instruction encoding and the low-order 5 bits in bits 11–15.

If the SPR field contains any value other than one of the values shown in Table 2-6, either the program exception handler is invoked or the results are boundedly undefined.

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.

Note that the updating of some registers requires additional synchronization to ensure that data access and instruction fetching dependencies are handled properly. For more information, see Section 2.1.4, "Synchronization Requirements for SPRs."

2.3.6.3 Memory Control Instructions—OEA

This section describes memory control instructions, which include the following types:

- Cache management instructions
- Segment register manipulation instructions
- TLB management instructions

2.3.6.3.1 Supervisor-Level Cache Management Instruction

Table 2-48 lists the only supervisor-level cache management instruction. See Section 2.3.5.3, "Memory Control Instructions—VEA," for a description of cache instructions that provide user-level programs the ability to manage the on-chip caches.

When data translation is disabled, MSR[DR] = 0, the **dcbz** instruction establishes a block in the cache and may not verify that the physical address is valid. If a block is created for an invalid real address, a machine check exception may result when an attempt is made to write that block back to memory. The block could be written back as the result of the execution of an instruction that causes a cache miss and the invalid address block is the target for replacement or as the result of a **dcbst** instruction.

Table 2-48. Supervisor-Level Cache Management Instruction

Name	Mnemonic	Operand Syntax	602 Notes
Data Cache Block Invalidate	**dcbi**	rA,rB	The EA is computed, translated, and checked for protection violations. If the addressed block is in the cache, it is marked 'I' regardless of whether the data is in the M or E state. The exception priorities (from highest to lowest) are as follows: 1 BAT protection violation—DSI exception 2 MMU Miss—DTLB exception 3 TLB protection violation—DSI exception

2.3.6.3.2 Segment Register Manipulation Instructions

The instructions listed in Table 2-49 provide access to the segment registers for the 602. These instructions operate completely independently of the MSR[IR] and MSR[DR] bit settings. Refer to Section 2.1.4, "Synchronization Requirements for SPRs," for serialization requirements and other recommended precautions to observe when manipulating the segment registers.

Table 2-49. Segment Register Manipulation Instructions

Name	Mnemonic	Operand Syntax
Move to Segment Register	**mtsr**	SR,rS
Move to Segment Register Indirect	**mtsrin**	rS,rB
Move from Segment Register	**mfsr**	rD,SR
Move from Segment Register Indirect	**mfsrin**	rD,rB

2.3.6.3.3 Translation Lookaside Buffer Management Instructions

The address translation mechanism is defined in terms of segment descriptors and page table entries (PTEs) used by PowerPC processors to locate the effective-to-physical address mapping for a particular access. The PTEs reside in page tables in memory. As defined for 32-bit implementations by the PowerPC architecture, segment descriptors reside in 16 on-chip segment registers.

The 602 provides the ability to invalidate a TLB entry. The TLB Invalidate Entry (**tlbie**) instruction invalidates the TLB entry indexed by the EA, and operates on both the instruction and data TLBs simultaneously invalidating four TLB entries (both sets in each TLB). The index corresponds to bits 15–19 of the EA. To invalidate all entries within both TLBs, 32 **tlbie** instructions should be issued, incrementing this field by one each time.

The 602 provides two implementation-specific instructions, TLB Load Data and TLB Load Instruction (**tlbld** and **tlbli**), that are used by software table search operations to load TLB entries on-chip following TLB misses. For a complete description of these instructions, see Section 2.3.7, "PowerPC 602 Implementation-Specific Instructions."

Refer to Chapter 5, "Memory Management" for more information about the TLB operations for the 602. Table 2-50 lists the TLB instructions.

Table 2-50. Translation Lookaside Buffer Management Instructions

Name	Mnemonic	Operand Syntax	602 Notes
T LB Invalidate Entry	**tlbie**	rB	This instruction invalidates both ways in the ITLBs and DTLBs at the index provided within the EA. Executes without regard to translation setting. To invalidate all TLB entries, the **tlbie** instruction should be executed 16 times, each time incrementing the index (EA[16–19]).
TLB Synchronize	**tlbsync**	—	This instruction is implemented and treated as a no-op
TLB Load Data	**tlbld**	rB	This is a 602-specific instruction. It loads the contents of the DCMP and RPA registers into the first word of the DTLB entry selected by the EA and the SRR1[WAY] bit. See Section 2.3.7, "PowerPC 602 Implementation-Specific Instructions."
TLB Load Instruction	**tlbli**	rB	This is a 602-specific instruction. It loads the contents of the ICMP and required physical address (RPA) registers into the first word of the ITLB entry selected by the EA and the SRR1[WAY] bit. See Section 2.3.7, "PowerPC 602 Implementation-Specific Instructions."

Because the presence and exact semantics of the TLB management instructions are implementation-dependent, system software should incorporate uses of the instruction into subroutines to maximize compatibility with programs written for other processors.

For more information on the PowerPC instruction set, refer to Chapter 4, "Addressing Modes and Instruction Set Summary," and Chapter 8, "Instruction Set," in *The Programming Environments Manual*.

2.3.7 PowerPC 602 Implementation-Specific Instructions

This section describes the instructions that are specific to the 602. However, some of these instructions may be supported by other PowerPC processors.

dsa

Disable Supervisor Access

dsa

Integer Unit

dsa

31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	628	0
0 5	6 10	11 15	16 20	21 30	31
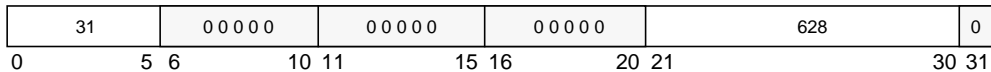
This instruction is not part of the PowerPC architecture.

The **dsa** instruction is the last instruction executed before returning from a routine in supervisor-access region. The **dsa** instruction does the following:

- Restores MSR[PR, AP, EE, SA] from ESASRR.
- If MSR[SA] = 0, a program exception is taken.

The following rules should be followed for using the **dsa** and **esa** instructions:

- Supervisor-access routines must begin with an **esa** instruction.

- Execution of an **esa** or **dsa** instruction cannot alter the instruction stream in any way. Make sure that consistency is maintained in the following:

  — Protections for fetching (NE, etc.)

  — Translation method (BAT, TLB)

- Unprivileged store addresses must be checked before being passed to supervisor-access routines.

- Data is not allowed in supervisor-access areas.

- Supervisor-access routines cannot call other supervisor-access routines. Executing an **esa** instruction when MSR[SA] = 1 causes a program exception.

esa
Enable Supervisor Access

<div align="right">

esa
Integer Unit

</div>

esa

31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	596	0
0 5	6 10	11 15	16 20	21 30	31
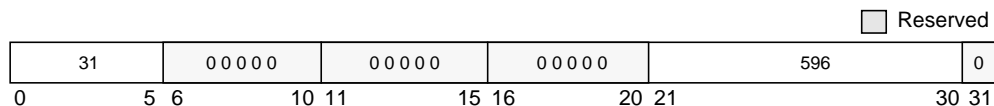
This instruction is not part of the PowerPC architecture.

The **esa** instruction is the entry point for routines in supervisor access regions. The **esa** instruction is a nonserialized instruction and does the following:

- Saves MSR[SA, EE, PR, AP] to the ESASRR
- Sets appropriate MSR bits (SA = 1, EE = 0, PR = 0, AP = 0)
- If the memory space is not designated as a supervisor-access region (that is, if the SE bit in the corresponding TLB or BAT is cleared), a program exception occurs.
- If MSR[SA] = 1, a program exception occurs. A second **esa** instruction cannot be executed until the SA bit has been cleared, preferably by a **dsa** instruction.

The following rules should be followed for using the **esa** and **dsa** instructions:

- Supervisor access routines must begin with an **esa** instruction.
- Execution of **esa** or **dsa** cannot alter the instruction stream in any way:
  — Protections for fetch (NE, etc.)
  — Translation method (BAT, TLB, or protection-only mode) must be consistent
- Store addresses passed to supervisor access routines must be checked.
- Data is not allowed in supervisor access areas.
- Supervisor access routines cannot call other supervisor access routines. Nesting supervisor access routines causes a program exception (that is, issuing an **esa** instruction with MSR[SA] set).

mfrom
Move from ROM

mfrom   **r**D**,r**A

31	D	A	0 0 0 0 0	265	0

☐ Reserved

0   5 6   10 11   15 16   20 21   30 31
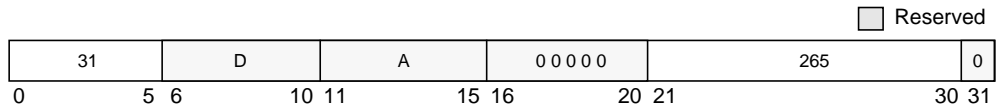
This instruction is not part of the PowerPC architecture.

The content of the internal ROM addressable by (**r**A) is moved to the GPR addressed by **r**D. The ROM contains a 7-bit value that is zero-extended to 32 bits. The ROM contains 602 entries (addressed by the least significant 10 bits of **r**A); if addressed out of range, the ROM returns a zero value. The ROM contents are derived by the following:

$ROM(I) = 256 * Log_{10}(1 + 10^{-I/256}) + 0.5$

The **mfrom** instruction is a supervisor-level instruction, Attempting to execute this instruction when MSR[PR] = 1 (user level), causes an illegal instruction program exception to be taken.

Application note: For speech or handwriting recognition applications using Markov models where single-precision operations are not adequate, the integer unit can be used as long as the input data is in logarithmic form. For multiplication operations, such as
$C = A \ x \ B$, the operation can be performed as the addition of logarithms as the following:

$log(C) = log(A)+log(B)$

For addition operations such as $C = A + B$, the **mfrom** instruction solves this problem by helping to implement the following computation:

$$log(C) = log(10^{log(A)} + 10^{log(B)})$$

It can be seen that, for a given finite binary implementation, all numbers have minimum and maximum boundaries and that if *A* is very large and *B* is very small, then *C* will be equal to *A*. The format for integer values is *256 x log(value)*. With this precision, the *log(C)* can be computed as an adjustment to the greater of *log(A)* or *log(B)*. The scaling factor, *256 x log(value)*, ensures that there are only 602 values that allow *log(A)* and *log(B)* to be close enough to each other to cause *log(C)* to be other than *max(log(A), log(B))*.

The adjustment needed to determine log(C) can be computed with **mfrom** instruction and a lookup table indexed by the difference of |log(A) - log(B)|. The instruction returns a 7-bit value from a 602-entry table through the following formula

$$\chi_i = 256 \times log(1 + 10^{-(i/256)}) + 0.5$$

where *i* is the address (or index) into the table determined by *|log(A)-log(B)|* and *x* is the returned 7-bit value needed to adjust *max(log(A), log(B))* to determine *log(C)*.

tlbld   tlbld
TLB Load Data   Integer Unit

tlbld   **r**B

31	0 0 0 0 0	0 0 0 0 0	B	978	0
0 5	6 10	11 15	16 20	21 30	31
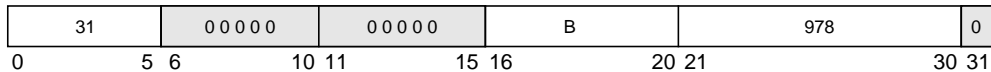
This instruction is not part of the PowerPC architecture.

> EA ← (**r**B)
> TLB entry created from DCMP and RPA
> DTLB entry selected by EA[15-19] and SRR1[WAY] ← created TLB entry

The EA is the contents of rB. The **tlbld** instruction loads the contents of the DCMP and RPA registers into the first word of the selected data TLB entry. The specific DTLB entry to be loaded is selected by <ea> and the SRR1[WAY] bit.

The **tlbld** instruction should only be executed when address translation is disabled. MSR[IR] = 0 and MSR[DR] = 0.

This is a 602-specific, supervisor-level instruction.

Note that if the processor is in HID0[PO] = 1 mode, bits 11–14 of the EA are used to index the TLB instead of bits 15–19.

Other registers altered:

- None

Load the data TLB entry indexed by EA with DCMP and RPA.

tlbli

tlbli

TLB Load Instruction

tlbli

Integer Unit

tlbli   **r**B

☐ Reserved

31	0 0 0 0 0	0 0 0 0 0	B	1010	0

0   5 6   10 11   15 16   20 21   30 31
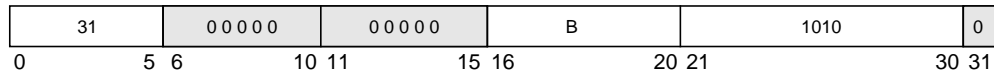
This instruction is not part of the PowerPC architecture.

> EA ← (**r**B)
> TLB entry created from ICMP and RPA
> ITLB entry selected by EA[15-19] and SRR1[WAY] ← created TLB entry

The EA is the contents of **r**B. The **tlbli** instruction loads the contents of the instruction PTE compare (ICMP) and required physical address (RPA) registers into the first word of the selected instruction TLB entry. The specific ITLB entry to be loaded is selected by <ea> and the SRR1[WAY] bit.

For code compatibility the **tlbli** instruction should be limited to TLB miss handlers and should only be executed when address translation is disabled. MSR[IR] = 0 and MSR[DR] = 0. If it is desired to use these instructions with translation enabled, use care. It is required that a context-synchronizing instruction follow these instructions if translation is on. It is considered a programming error to perform a **tlbli** that modifies the current prefetch addresses.

Note that if the processor is in HID0[PO] = 1 mode, bits 11–14 of the EA are used to index the TLB instead of bits 15–19.

This is a supervisor-level instruction; it is also a 602-specific instruction, and not part of the PowerPC instruction set.

Other registers altered:

- None

2.3.8 Recommended Simplified Mnemonics

To simplify assembly language programs, a set of simplified mnemonics is provided for some of the most frequently used operations (such as no-op, load immediate, load address, move register, and complement register). PowerPC-compliant assemblers provide the simplified mnemonics listed in "Recommended Simplified Mnemonics" in Appendix F, "Simplified Mnemonics," in *The Programming Environments Manual* and listed with some of the instruction descriptions in this chapter. 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*.

2.3.9 Using the esa Instruction for Supervisor-Level Access

The 602 can be made to operate in supervisor mode either by taking an exception or by executing the 602-specific Enable Supervisor Access (**esa**) instruction. Executing the **esa** allows the processor to access supervisor-level instructions, registers, and memory without encountering the latencies associated with the kind of exception handling required for processors used in multipurpose personal computers. Such latencies include synchronization to ensure precise operation and the pipeline and memory access latencies associated with having to refetch from a new instruction path.

Note that after the **esa** instruction has been successfully executed, the program can fetch from any page defined as instruction space for which fetching is enabled regardless of the setting of SE.

When the **esa** instruction is executed, several bits from the MSR (MSR[SA, EE, PR, AP] to the ESASRR) and those bits are automatically set as follows (SA = 1, EE = 0, PR = 0, AP = 0). Clearing MSR[EE] disables external interrupts, clearing MSR[PR] puts the processor in supervisor mode, and clearing MSR[AP] gives the processor supervisor-level access to memory locations. MSR[SA] indicates that the processor is operating in this **esa**-initiated supervisor mode. This bit is cleared when the Disable Supervisor Access (**dsa**) instruction is executed. If MSR[SA] is not set, attempting to execute **dsa** causes a program exception.

The processor remains in supervisor mode until the **dsa** instruction is executed. Note that the **dsa** instruction can be executed from any memory location for which instruction fetching is enabled—that is, the **dsa** instruction can be executed regardless of the setting of SE for the page on which it resides.

Implementation of the ESA supervisor-access feature affects the 602's MMU implementation in the following ways:

- The **esa** instruction is enabled on a page or block basis, so the MMU translation mechanism must be used to configure memory to allow or disallow this functionality. An additional SE bit is provided in the TLBs and BATs to enable the **esa** instruction.The ESA enable register (SER) and the ESA enable base register (SEBR) control supervisor execute privileges for each of the 32 pages of a 128-Kbyte block of memory. Configuration of memory space defined by TLBs is handled by using the 602-defined TLB Load Instruction (**tlbli**) and TLB Load Data (**tlbld**) instructions to set the SE bit in the TLBs. BAT[SE] can be written by using the **mtspr** instruction.

- This facility can be used regardless of whether the processor uses one of the architecturally-defined translation mechanisms or the 602-specific protection-only mode. When the **esa** instruction is enabled in protection-only mode (for which the translation mechanism is not used to form the real address, EA = RA), resources such as the RPA and TLBs that are otherwise defined for translation are redefined to support memory protection.

- Note that translation must be enabled for **esa** to be executed; therefore, **esa** cannot be executed when the processor is in real addressing mode.

2.3.9.1 esa/dsa Instructions

The two instructions are described as follows:

- Enable Supervisor Access (**esa**)—The **esa** instruction is the entry point for routines in **esa** supervisor-access regions.

  If the memory space is not designated as a supervisor access region (that is, if the SE bit in the corresponding TLB or BAT is cleared), a program exception occurs.

- Disable Supervisor Access (**dsa**)—The **dsa** instruction is the last instruction executed before returning from a routine initiated by the **esa** instruction. The **dsa** instruction restores MSR[PR, AP, EE, SA] from ESASRR. Note that if MSR[SA] = 0, a program exception is taken when the **dsa** instruction is executed.

The following rules should be followed for using the **dsa** and **esa** instructions:

- Begin all supervisor-access routines with an **esa** instruction.

- Execution of an **esa** or **dsa** instruction cannot alter the instruction stream in any way. Protections for fetching (NE, etc.) and the translation method (BAT, TLB) must be consistent.

- Check unprivileged store address passed to supervisor-access routines.

- Supervisor access routines cannot call other supervisor-access routines. Executing an **esa** instruction when MSR[SA] = 1 causes a program exception.

- No data should be updated in supervisor access areas.

PowerPC 602 RISC Microprocessor User's Manual

2.3.9.2 ESA Supervisor-Access Registers

The 602 defines a set of resources that allow the processor to access supervisor-level instructions, registers, and memory resources without taking an exception. This supervisor access is signaled by the execution of the 602-specific **esa** instruction. Execution of this instruction is allowed only if it is enabled for the 4-Kbyte page in which it resides.

There are three registers that are 602-specific that support this functionality as described in the following list. The ESASRR register is used to save and restore the four MSR bits that are saved when the **esa** instruction is executed. The SEBR and SER are used to enable the **esa** instruction on a 4-Kbyte page basis when the processor is in protection-only mode.

- ESA Save and Restore Register (ESASRR)—ESASRR is a supervisor-level register that provides a means for automatically saving and restoring aspects of the machine state for use with the enable/disable supervisor access instructions (**esa** and **dsa**). When an **esa** instruction is executed, MSR[SA, EE, PR, AP] bits are automatically saved in the ESASRR. When a **dsa** instruction is executed, the contents of these bits are automatically restored to the MSR. The ESASRR is described more fully in Section 2.1.2.3.1, "ESA Save and Restore Register (ESASRR)."

- ESA Enable Base Register (SEBR)—The SEBR is used when the processor is in protection-only mode to determine whether the **esa** instruction is enabled for the 4-Kbyte page in which it resides. SEBR[0–14] contains the base address of the 128-Kbyte region that is protected by the 32 SE bits in SER (each bit in the SER configures a 4-Kbyte page). The SEBR is described more fully in Section 2.1.2.3.2, "ESA Enable Base Register (SEBR) (Protection-Only Mode)."

- ESA Enable Register (SER)—The SER contains 32 SE bits, each of which corresponds to a 4-Kbyte page when the processor is in protection-only mode. If a match occurs when SEBR[0–14] are compared against the EA[0–14], EA[15–19] indicate which of the 32 SE bits in the SER is examined to determine whether the **esa** instruction can be executed. If there is no match, SE = 0. The matching requirement of the SEBR is similar to the BAT register. The SER is described more fully in Section 2.1.2.3.3, "ESA Enable Register (SER) (Protection-Only Mode)."

2.3.9.2.1 Enabling the esa Instruction

For the **esa** instruction to be executed it must first be enabled for the memory region in which it resides. This ability is enabled by setting the SE bit, which is read either from the IBAT, the ITLB, or the ESA register depending on the memory translation protection mode chosen. These are described as follows:

- Block address translation—If a BAT hit occurs, whether the **esa** instruction can be executed depends on the setting of the corresponding IBAT[SE].

- Page address translation—If page address translation is used (that is a BAT miss occurs and the processor is not in protection-only mode), whether the **esa** instruction can be executed depends on the setting of the SE bit in the corresponding ITLB.

- Protection-only mode—If the processor is in protection-only mode (HID0[PO] = 1) and a BAT miss occurs, whether the **esa** instruction can be executed depends on the setting of the corresponding SE bit in the SER register.

These translation/protection mechanisms are described in Chapter 5, "Memory Management."

2.3.9.2.2 Executing the esa Instruction

The appropriate SE bit is not checked when the **esa** instruction is fetched. That is, the **esa** instruction can be fetched even though it may not be enabled by the SE bit. Logic is provided in the 602 instruction pipeline that allows the completion unit to detect whether the **esa** instruction has been enabled. If **esa** has not been enabled, an illegal instruction program exception is taken. If it has been enabled, the following occurs:

— MSR[SA, EE, PR, AP] are saved to the ESASRR before those bits are reset for operation in supervisor mode.

— MSR[SA] is set, to indicate that the processor is in **esa** supervisor mode. This bit is cleared by the **dsa** instruction; attempting to execute another **esa** instruction when this bit is set causes a program exception.

— MSR[EE] is cleared to disable external interrupt exceptions.

— MSR[PR] is cleared to give supervisor level access to registers and instructions.

— MSR[AP] is cleared to provide access to supervisor-level memory regions.

Note that when the **esa** instruction is enabled, it is not context-synchronizing, There is no change in program flow and subsequent instructions in the program do not encounter stalls due to saving the MSR bits.

After the **esa** instruction has executed, the processor is in supervisor mode and can access supervisor-level instructions, registers, and memory space (depending on the setting of MSR[AP]). Note that after the MSR[SA] bit has been set, no more **esa** instructions can be executed. Attempting to do so causes a program exception.

2.3.9.2.3 Returning to User-Level Operation

After the **esa** instruction has been successfully executed, the processor remains in supervisor mode as long as the MSR[SA] bit remains set. Although this bit can be cleared explicitly by using an **mtspr** instruction, the recommended way to clear this bit is by executing the Disable Supervisor Access (**dsa**) instruction. Executing this instruction is analogous to executing an **rfi** instruction in an exception handler in that the processor returns to user mode and the four bits that were saved and reset when the **esa** instruction was executed (MSR[SA, EE, PR, AP]) are restored to the MSR.

2.3.10 Differences between Using the esa Instruction and Taking a System Call Exception

The Enable Supervisor Access (**esa**) instruction can be executed only if the SE bit is set for the block or the page in which the **esa** instruction resides. If an **esa** instruction is fetched from any other region, a program exception is taken. Successful execution of the **esa** instruction places the processor in supervisor mode, and can be compared with the system call exception in the following ways:

- When the **esa** instruction is executed, there is no change in program flow as is the case with fetching from the address of the target exception vector. This eliminates the latency required for synchronizing to maintain a precise exception model and the stalls that may occur during those cycles when the exception handler instructions have not reached the instruction queue.

- Because there is no change in program flow, the SRR0 register is not used to save a return address. The SRR1 register is not used to save the machine state. Instead, only four MSR bits are affected. The status of MSR[SA, EE, PR, AP] are saved to the ESASRR.

- After the **esa** instruction is executed, four bits in the MSR are updated:
  — MSR[SA] is set to indicate that the processor is in **esa** supervisor mode.
  — MSR[EE] is cleared to disable external interrupt exceptions.
  — MSR[PR] is cleared to give supervisor-level access to registers and instructions.
  — MSR[AP] is cleared to provide access to supervisor-level memory regions.

- Translation must be enabled; therefore, **esa** supervisor access is not available when the processor is running in real addressing mode.

If translation remains active, all effective addresses must map in the TLBs or BATs. MSR[PR, AP, SA, EE] bits are updated upon entry into the **esa** routine; the state of these bits prior to the execution of **esa** are saved in ESASRR until the **dsa** instruction restores them. Note that MSR[SA] must be set when the **dsa** instruction is executed to return the processor to user-level operation; if this bit is cleared, a program exception occurs. Note that any changes to the MSR (other than to the four saved bits) are retained after the **dsa** instruction is executed.

PowerPC 602 RISC Microprocessor User's Manual

Chapter 3
Instruction and Data Cache Operation

This chapter describes the organization of the PowerPC 602 microprocessor's on-chip instruction and data caches, the MEI cache coherency protocol, cache control instructions, various cache operations, and the interaction between the cache, load/store unit, and the bus interface unit.

3.1 PowerPC 602 Processor Cache Implementation Overview

The 602 provides separate 4-Kbyte, two-way set-associative caches for instructions and data. Both the instruction and data caches are tightly coupled to the 602's bus interface unit (BIU) to allow efficient access to the system memory controller and other bus masters. The 602's load/store unit (LSU) is also directly coupled to the data cache to move data to and from the GPRs and FPRs efficiently. Figure 3-1 shows the organization of the 602 caches, which is essentially the same for both the instruction and data caches.
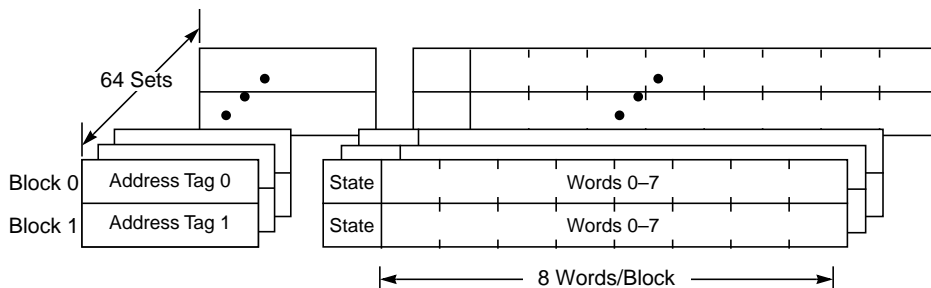
Figure 3-1. PowerPC 602 Processor Instruction and Data Cache Organization

Both caches have a cache block (cache line) size of 32 bytes, and the data cache blocks can be snooped, or cast out when the cache block is reloaded. The 602 provides bits (WIM) for control of write-back policy, cacheability, and memory coherency. These bits are implemented differently depending on the type of address translation used, as described in Chapter 5, "Memory Management."

Note that in the PowerPC architecture, the term "cache block" refers to the unit of memory at which coherency is maintained. For the 602, this is eight words. This size of a cache block may be different for other PowerPC implementations.

Typically, when an instruction or data address is not found in the cache, the cache is updated. Because it is extremely likely that the data or instructions at adjacent addresses will also be needed, the entire cache block of data that includes the data at the requested address is updated from memory by means of a burst transaction—an automatic series of bus transactions that transfers an entire eight-word cache block.

Read misses and instruction fetches may generate burst read operations that transfer the entire cache block from memory. A write miss may also cause a burst transaction from memory—after the cache block is updated from memory, the write operation that caused the burst read updates the memory at the write address within the cache block. Whether and when this new data is passed on to external memory depends on how coherency attributes are configured.

Burst operations can also occur as the result of other operations. For example, both caches use a least-recently used (LRU) replacement policy, so if both cache blocks to which an address can be mapped are valid, the new data replaces the older of the two cache blocks. If a data cache block that is being replaced is modified (that is, it has the correct contents for that address but those contents have not been written back to memory), the data in that cache block must be written back (cast out) to memory. Basic data cache operations are described in Section 3.4, "Basic Data Cache Operations."

As shown in Figure 3-1, the data cache is organized into 64 sets of two cache blocks apiece. Each block has an address tag and two state bits. The two state bits implement the three-state MEI (modified-exclusive-invalid) protocol, a coherent subset of the standard four-state MESI protocol. The 602's on-chip data cache tags are single-ported, and load or store operations must be arbitrated with snoop accesses to the data cache tags. Load or store operations can be performed to the cache on the clock cycle immediately following a snoop access if the snoop misses; snoop hits may block the data cache for two or more cycles, depending on whether a copyback to main memory is required.

The 602 supports an additional snooping mechanism, known as injected snooping. While the 602 as a bus master performs a burst read transaction, the read target device can inject the snoop address onto the bus between data beats. Injected snooping is described in Section 8.4.2, "Qualified Snoop Conditions," and is illustrated in Section 8.5.4.7, "Injected Snoop Timings." Information about the MEI protocol is provided in Section 3.7, "Cache Coherency—MEI Protocol."

The instruction cache is also organized into 64 sets of two 32-byte cache blocks. Each block has an address tag, but unlike the data cache which has two bits that identify the MEI state, the instruction cache has only one state bit that indicates whether the contents of the cache block are valid. Because the instruction cache is updated only as a result of a line-fill operation on a cache miss, it is not snooped.

The load/store unit (LSU) provides the data transfer interface between the data cache and the GPRs and the FPRs. The LSU provides all logic required to calculate effective addresses and handles data alignment to and from the data cache. As shown in Figure 1-1, the caches provide a 32-bit interface to the instruction fetcher and LSU. Write operations to the data cache can be performed on a byte, half-word, word, or double-word basis. The bus can be dynamically configured to function as a 32-bit bus in the data phase (using only D0–D31 for the data transfer).

The 602's bus interface unit (BIU), described in Section 3.10, "Bus Interface", receives requests for bus operations from the instruction and data caches, and uses the 602 bus protocol to direct transfers to and from external memory. The BIU provides address queues, prioritization, and bus control logic. The BIU also captures snoop addresses for data cache, address queue, and memory reservation (**lwarx** and **stwcx.** instruction) operations.

On a burst-read operation, corresponding cache block is filled in four beats of 64 bits each when the bus is in 64-bit mode or in eight beats of 32 bits each in 32-bit mode. The requested instruction or data arrives first as part of the critical double word, and is simultaneously written to the cache and forwarded to the requesting unit.

For example, if the instruction cache is performing a cache line reload, the requested instruction is forwarded to the BPU at the same time that is made available in the cache. If the bus is in 64-bit mode, the instruction that arrives as part of the first double beat goes to the cache only, and must be fetched from there.

Additionally, if a branch instruction is fetched from the cache block that is being filled, it may point either to another address within same cache block or to another location as follows:

- If the target instruction is elsewhere in the same cache block, that instruction can be fetched as soon as it becomes available in the instruction cache.
- If the target instruction is elsewhere in the cache, the instruction can be fetched without having to wait for the entire cache block to be updated.

The 602 supports a fully-coherent 4-Gbyte physical memory address space. How that memory is organized and defined is largely controlled by the memory management model, and this in turn has an effect on how the caches operate. Caching attributes are defined by four mode control bits—W, I, M, and G. The W (write-through) and I (caching-inhibited) bits control how the processor executing the access uses its own cache. The M (memory coherence) bit specifies whether the processor executing the access must use the MEI (modified, exclusive, or invalid) cache coherence protocol to ensure all copies of the addressed memory location are kept consistent. The G (guarded memory) bit controls whether out-of-order data and instruction fetching is permitted.

The method by which the WIMG bits are configured depends on the type of memory translation used. If block address translation is used, the bits are defined in the BAT registers. If page address translation is used, the bits are configured through the translation lookaside buffers (TLBs), which in turn are kept in page table entries (PTEs) which also contain translation information and are maintained by the MMU. If real addressing mode is used (that is, the translation is disabled) or if the processor is in the 602-specific protection-only mode, the WIMG bits are read from the HID0 register. Address translation is described in Chapter 5, "Memory Management."

The 602 maintains data cache coherency in hardware by coordinating activity between the data cache, the memory system, and the bus interface logic. As bus operations are performed on the bus by other bus masters, the 602 bus snooping logic monitors the addresses that are referenced. These addresses are compared with the addresses resident in the data cache. If there is a snoop hit, the 602's bus snooping logic responds to the bus interface with the appropriate snoop status (for example, an $\overline{\text{ARTRY}}$). Additional snoop action may be forwarded to the cache as a result of a snoop hit in some cases (a cache push of modified data or a cache block invalidation).

3.2 Instruction Cache Organization and Control

Each block of the instruction cache can hold eight 32-bit words (that is, eight instructions), an address tag, and a valid bit. The instruction cache may not be written to except through a line-fill operation. In addition, this cache is not snooped. Instruction cache coherency must be maintained by software and is supported by a fast hardware invalidation capability.

The instruction fetcher accesses the instruction cache frequently in order to sustain the high throughput provided by the six-entry instruction dispatch queue.

3.2.1 Instruction Cache Organization

The organization of the instruction 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 logical (effective) addresses are zero); as a result, cache blocks are aligned with page boundaries.

Note that address bits A21–A26 provide an index to select a set. Bits A27–A31 select a byte within a block. The tags consists of bits PA0–PA20. Address translation occurs in parallel, such that higher-order bits (the tag bits in the cache) are physical. Note that the replacement algorithm is strictly an LRU algorithm; that is, the least-recently used block is filled with new instructions on a cache miss.

3.2.2 Instruction Cache Fill Operations

In 64-bit mode, the 602's instruction cache blocks are loaded in four beats of 64 bits each, with the critical double word loaded first. In 32-bit mode, this operation takes eight beats.

The 602's caches are nonblocking during line-fill operations. If the instruction cache is performing a cache-line-reload, the requested instruction is forwarded to the BPU at the same time that is made available in the cache. In 32-bit mode, the instruction that arrives as part of the first double beat goes to the cache only and must be fetched from there.

Additionally, if a branch instruction is fetched from the cache block that is being filled it may point either to another address within the same block or to another location as follows:

- If the target instruction is elsewhere in the same cache block, that instruction can be fetched as soon as it becomes available in the instruction cache.
- If the target instruction is elsewhere in the cache, the instruction can be fetched without having to wait for the entire cache block to be updated.

Note, however, that in both of these cases, instructions can be fetched from the cache only while it is not being written to as part of the cache-line-refill.

3.2.3 Instruction Cache Control

In addition to instruction cache control instructions, the 602 provides three control bits in the HID0 register for the control of invalidating, disabling, and locking the instruction cache. The HID0 register is described in Section 2.1.2.1.1, "Hardware Implementation Register 0 (HID0)."

3.2.3.1 Instruction Cache Invalidation

While the 602's instruction cache is automatically invalidated during a power-on or hard reset, assertion of the soft reset signal does not cause instruction cache invalidation. Software may invalidate the contents of the instruction cache using the instruction cache flash invalidate control bit, HID0[ICFI]. Flash invalidation of the instruction cache is accomplished by setting and clearing the ICFI bit with two consecutive move to SPR operations to the HID0 register.

3.2.3.2 Locking the Instruction Cache

The contents of instruction cache may be locked through the use of the instruction cache lock control bit, HID0[ILOCK]. A locked instruction cache supplies instructions normally on a cache hit, but cache misses are treated as caching-inhibited accesses. The cache inhibited ($\overline{\text{CI}}$) signal is asserted if a cache access misses into a locked cache. The setting of the ILOCK bit in HID0 must be preceded by an **isync** instruction to prevent the instruction cache from being locked during an instruction access.

3.3 Data Cache Organization and Control

The data cache supplies data to the GPRs and FPRs by means of the load/store unit, and provides buffers for load and store bus operations. The data cache also provides storage for the cache tags required for memory coherency and performs the cache block replacement LRU function.

3.3.1 Data Cache Organization

The organization 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 logical (effective) addresses are zero); as a result, cache blocks are aligned with page boundaries.

Note that address bits A21–A26 provide an index to select a set. Bits A27–A31 select a byte within a block. The tags consists of bits PA0–PA20. Address translation occurs in parallel, such that higher-order bits (the tag bits in the cache) are physical. Note that the replacement algorithm is strictly an LRU algorithm; that is, the least recently used block is filled with new data on a cache miss.

3.3.2 Data Cache Fill Operations

The 602's data cache blocks are filled in four beats of 64 bits each, in 64-bit mode or eight beats of 32 bits each in 32-bit mode, with the critical double word loaded first in either mode. The contents of the critical double word are forwarded to the register files at the same time that it arrives in the cache. The data cache is nonblocking and can be accessed by the processor core as subsequent data arrives in the cache.

3.3.3 Data Cache Control

The 602 provides several means of data cache control through the use of the WIMG bits (implemented in the page tables for page addressing mode, the BATs for block address translation mode, and in HID0 for real addressing mode and protection-only mode), control bits in the HID0 register, and user- and supervisor-level cache control instructions. While memory page level cache control is provided by the WIMG bits, the on-chip data cache can be invalidated, disabled, or locked by the three control bits in the HID0 register described in this section. The HID0 register is described in Section 2.1.2.1.1, "Hardware Implementation Register 0 (HID0)." (Note that, user- and supervisor-level are referred to as problem state and privileged state, respectively, in the architecture specification.)

3.3.3.1 Data Cache Invalidation

While the data cache is automatically invalidated when the 602 is powered up and during a hard reset, assertion of the soft reset signal does not cause data cache invalidation. Software may invalidate the contents of the data cache using the data cache flash invalidate control bit, HID0[DCFI]. Flash invalidation of the data cache is accomplished by setting and clearing the DCFI bit with two consecutive **mtspr**[HID0] operations.

3.3.3.2 Disabling the Data Cache

The data cache may be disabled through the use of the data cache enable (DCE) control bit in the HID0 register. When the data cache is in the disabled state, the cache tag state bits are ignored, and all accesses are propagated to the bus as nonburst transactions. The DCE bit is cleared on power-up, causing the data cache to be disabled. The setting of the DCE bit must be preceded by a **sync** instruction to prevent the cache from being enabled or disabled in the middle of a data access. An **isync** instruction should follow this **mtspr** instruction to guarantee that instructions after the disabling of the cache are cleared from the prefetch buffer.

Disabling the caches does not affect the translation logic; translation is still controlled with MSR[DR].

Note that while snooping is not performed when the data cache is disabled, cache operations (caused by the **dcbz**, **dcbf**, **dcbst**, and **dcbi** instructions) are not affected by disabling the cache, causing potential coherency errors. An example of this would be a **dcbf** instruction that hits a modified cache block in the disabled cache, causing a copyback to memory of potentially stale data.

3.3.3.3 Locking the Data Cache

The contents of the data cache may be locked through by setting HID0[DLOCK]. A locked data cache supplies data normally on a cache hit, but cache misses are treated as caching-inhibited accesses. The caching-inhibited ($\overline{\text{CI}}$) signal is asserted if a cache access misses into a locked cache. A snoop hit to a locked data cache performs as if the cache were not locked. A line invalidated by a snoop will remain invalid until the cache is unlocked.

The setting of HID0[DLOCK] must follow a **sync** instruction to prevent the data cache from being locked during a data access.

3.4 Basic Data Cache Operations

This section describes the three types of operations that can occur to the data cache, and how these operations are implemented in the 602.

3.4.1 Data Cache Line-Fill Operation

A cache block is filled after a read miss or write miss (read-with-intent-to-modify) occurs in the cache. The cache block that corresponds to the missed address is updated by a burst transfer of the data from system memory. Note that if a read miss occurs in a system with multiple bus masters, and the data is modified in another cache, the modified data is first written to external memory before the cache line-fill occurs.

3.4.2 Data Cache Cast-Out Operation

The 602 uses an LRU replacement algorithm to determine which of the two possible cache locations should be used for a cache update on a cache miss. Adding a new block to the cache causes any modified data associated with the least recently used element to be written back, or cast out, to system memory to maintain memory coherence.

3.4.3 Cache Block Push Operation

When a cache block in the 602 is snooped and hit by another bus master and the data is modified, the cache block must be written to memory and made available to the snooping device. The cache block that is hit is said to be pushed out onto the bus.

3.5 Data Cache Transactions on Bus

The 602 transfers data to and from the data cache in burst and nonburst transactions. In 32-bit mode, data can be transferred in single-beat (one word), double-beat (double word) or eight-beat burst transactions that transfer an eight-word cache block. In 64-bit mode, data is transferred in single-beat (double word) or four-beat burst transactions, that transfer the contents of an entire cache block.

3.5.1 Nonburst Transactions

Nonburst transactions (single-beat when the bus is in 64-bit mode and double-beat when the bus is in 32-bit mode) can transfer from one to eight bytes to or from the 602. Nonburst transactions can be cache write-through or caching-inhibited, and can be misaligned.

3.5.2 Burst Transactions

Burst transactions on the 602 always transfer eight words of data at a time and are aligned to a double-word boundary. When the bus is in 32-bit mode, eight beats are required; when it is in 64-bit mode, four beats are required. The 602 transfer burst ($\overline{\text{TBST}}$) output signal indicates to the system whether the current transaction is a nonburst or four-beat burst transfer. The target data bus 32 ($\overline{\text{T32}}$) signal indicates whether the bus is in 32- or 64-bit mode. Burst transactions have an assumed address order. For cacheable read operations or cacheable, non-write-through write operations that miss the cache, the 602 presents the double-word-aligned address associated with the load or store instruction that initiated the transaction.

As shown in Figure 3-2, this quad word contains the address of the load or store that missed the cache. This minimizes latency by allowing the critical code or data to be forwarded to the processor before the rest of the block is filled. For all other burst operations, however, the entire block is transferred in order (oct-word aligned). After an instruction or data cache miss, the critical double-word is forwarded first.

602 Cache Address
Bits (27..28)

0 0	0 1	1 0	1 1
A	B	C	D

If the address requested is in double word A (bits 27, 28 are 00), the address placed on the bus is that of double-word A, and the four data beats are ordered in the following manner:

Beat

0	1	2	3
A	B	C	D

If the address requested is in double word C (bits 27, 28 are 10), the address placed on the bus will be that of double-word C, and the four data beats are ordered in the following manner:

Beat

0	1	2	3
C	D	A	B

Figure 3-2. Double-Word Address Ordering—Critical Double Word First

3.5.3 Access to Direct-Store Segments

The 602 does not provide support for access to direct-store segments. Operations attempting to access a direct-store segment will invoke a DSI exception. For additional information about DSI exceptions, refer to Section 4.5.3, "DSI Exception (0x0300)."

3.6 Memory Management/Cache Access Mode Bits— W, I, M, and G

Some memory characteristics can be set on either a block or page basis by using the WIMG bits in the BAT registers or page table entry (PTE) respectively. The 602 also implements these bits in the HID0 register to provide control when the processor is in real addressing mode (that is, translation is disabled) or in the 602-specific protection-only mode. The WIMG attributes control the following functionality:

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

These bits allow both single- and multiprocessor-system designs to exploit numerous system-level performance optimizations.

Careless specification and use of these bits may create situations where coherency paradoxes are observed by the processor. In particular, this can happen when the state of these bits is changed without appropriate precautions being taken (for example, when flushing the pages that correspond to the changed bits from the caches of all processors in the system is required, or when the address translations of aliased physical addresses (referred to as real addresses in the architecture specification) specify different values for any of the WIM bits). The 602 treats either of these cases as a programming error that may compromise the coherency. These paradoxes can occur within a single processor or across several devices, as described in Section 3.7.7.1, "Internal Coherency Paradoxes."

The WIMG 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 out-of-order loading and prefetching from the addressed memory location.

When an access requires coherency, the processor performing the access must inform the coherency mechanisms throughout the system that the access requires memory coherency. The M attribute determines the kind of access performed on the bus (global or local).

The WIMG attributes occupy four bits in the BAT registers for block address translation, in the PTEs for page address translation, and in HID0 for real addressing and protection-only modes. The WIMG bits are programmed as follows:

- The operating system uses the **mtspr** instruction to program the WIMG bits in the BAT registers for block address translation. The IBAT register pairs do not have a G bit and all accesses that use the IBAT register pairs are considered not guarded.
- The operating system writes the WIMG bits for each page into the PTEs in system memory as it sets up the page tables.

3.6.1 Write-Through Attribute (W)

When an access is designated as write-through (W = 1), if the data is in the cache, a store operation updates the cached copy of the data. In addition, the update is written to the external memory location (as described below).

While the PowerPC architecture permits multiple store instructions to be combined for write-through accesses except when the store instructions are separated by a **sync** or **eieio** instruction, the 602 does not implement this "combined store" capability. Note that a store operation that uses the write-through attribute may cause any part of valid data in the cache to be written back to main memory.

The definition of the external memory location to be written to in addition to the on-chip cache depends on the implementation of the memory system but can be illustrated by the following examples:

- RAM—The store is sent to the RAM controller to be written into the target RAM.
- I/O device—The store is sent to the memory-mapped I/O control hardware to be written to the target register or memory location.

In systems with multilevel caching, the store must be written to at least a depth in the memory hierarchy that is seen by all processors and devices.

Accesses that correspond to $W = 0$ are considered write-back. For this case, although the store operation is performed to the cache, it is only made to external memory when a copy-back operation is required. Use of the write-back mode ($W = 0$) can improve overall performance for areas of the memory space that are seldom referenced by other masters in the system.

3.6.2 Caching-Inhibited Attribute (I)

If $I = 1$, the memory access is completed by referencing the location in main memory, bypassing the on-chip cache. During the access, the addressed location is not loaded into the cache nor is the location allocated in the cache. It is considered a programming error if a copy of the target location of an access to caching-inhibited memory is resident in the cache. Software must ensure that the location has not been previously loaded into the cache, or, if it has, that it has been flushed from the cache.

The PowerPC architecture permits data accesses from more than one instruction to be combined for caching-inhibited operations, except when the accesses are separated by a **sync** instruction, or by an **eieio** instruction when the page or block is also designated as guarded. This "combined access" capability is not implemented on the 602.

3.6.3 Memory Coherency Attribute (M)

This attribute is provided to allow improved performance in systems where hardware-enforced coherency is relatively slow, and software is able to enforce the required coherency. When $M = 0$, the processor does not enforce data coherency. When $M = 1$, the processor enforces data coherency and the corresponding access is considered to be a global access.

When the M attribute is set, and the access is performed, the global signal ($\overline{GBL}$)is asserted to indicate that the access is global. Snooping devices affected by the access must then respond to this global access if their data is modified by asserting $\overline{ARTRY}$, and updating the memory location.

Because instruction memory does not have to be consistent with data memory, the 602 ignores the M attribute for instruction accesses.

3.6.4 Guarded Attribute (G)

When the guarded bit is set, the memory area is designated as guarded, meaning that the processor will perform out-of-order accesses to this area of memory, only as follows:

- Out-of-order load operations from guarded memory areas are performed only if the corresponding data is resident in the cache.
- The processor prefetches from guarded areas, but only when required, and only within the memory boundary dictated by the cache block. That is, if an instruction is certain to be required for execution by the program, it is fetched and the remaining instructions in the block may be prefetched, even if the area is guarded.

This setting can be used to protect certain memory areas from read accesses made by the processor that are not dictated directly by the program. If there are areas of memory that are not fully populated (in other words, there are holes in the memory map within this area), this setting can protect the system from undesired accesses caused by out-of-order load operations or instruction prefetches that could lead to the generation of the machine check exception. Also, the guarded bit can be used to prevent out-of-order load operations or prefetches from occurring to certain peripheral devices that produce undesired results when accessed in this way.

3.6.5 W, I, and M Bit Combinations

Table 3-1 summarizes the six combinations of the WIM bits. Note that either a '0' or '1' setting for the G bit is allowed for each of these WIM bit combinations.

Table 3-1. Combinations of W, I, and M Bits

WIM Setting	Meaning
000	Data may be cached. Loads or stores whose target hits in the cache use that entry in the cache. Memory coherency is not enforced by hardware.
001	Data may be cached. Loads or stores whose target hits in the cache use that entry in the cache. Memory coherency is enforced by hardware.
010	Caching is inhibited. The access is performed to external memory, completely bypassing the cache. Memory coherency is not enforced by hardware.
011	Caching is inhibited. The access is performed to external memory, completely bypassing the cache. Memory coherency must be enforced by external hardware (processor provides hardware indication that access is global).
100	Data may be cached. Load operations whose target hits in the cache use that entry in the cache. Stores are written to external memory. The target location of the store may be cached and is updated on a hit. Memory coherency is not enforced by hardware.

Table 3-1. Combinations of W, I, and M Bits (Continued)

WIM Setting	Meaning
101	Data may be cached. Load operations whose target hits in the cache use that entry in the cache. Stores are written to external memory. The target location of the store may be cached and is updated on a hit. Memory coherency is enforced by hardware.
11x	Not supported

3.6.5.1 Out-of-Order Execution and Guarded Memory

Out-of-order execution occurs when the 602 performs operations in advance in case the result is needed. Typically, these operations are performed by otherwise idle resources; thus if a result is not required, it is ignored and the out-of-order operation incurs no time penalty (typically).

Supervisor-level programs designate memory as guarded on a block or page level. Memory is designated as guarded if it may not be "well-behaved" with respect to out-of-order operations.

For example, the memory area that contains a memory-mapped I/O device may be designated as guarded if an out-of-order load or instruction fetch performed to such a device might cause the device to perform unexpected or incorrect operations. Another example of memory that should be designated as guarded is the area that corresponds to a device that resides at the highest implemented physical address (as it has no successor and out-of-order sequential operations such as instruction prefetching can cause a machine check exception). In addition, areas that contain holes in the physical memory space may be designated as guarded.

3.6.5.2 Effects of Out-of-Order Data Accesses

Most data operations may be performed out-of-order, as long as the machine appears to follow a simple sequential model. However, the following out-of-order operations do not occur:

- Out-of-order loading from guarded memory (G = 1) does not occur unless the requested data is in the cache. However, when a load or store operation is required by the program, the entire cache block(s) containing the referenced data may be loaded into the cache.
- Out-of-order store operations that alter the state of the target location do not occur.
- No errors except machine check exceptions are reported due to the out-of-order execution of an instruction until it is known that execution of the instruction is required.

Machine check exceptions resulting solely from out-of-order execution (from nonguarded memory) may be reported. When an out-of-order instruction's result is abandoned, only one side effect (other than a possible machine check) may occur—the referenced bit (R) in the corresponding page table entry can be set due to an out-of-order load operation. See Chapter 4, "Exceptions," for more information on the machine check exception.

Thus an out-of-order load or store instruction will not access guarded memory unless one of the following conditions exist:

- The target memory item is resident in an on-chip cache. In this case, the location may be accessed from the cache or main memory.
- The target memory item is cacheable (I = 0) and it is guaranteed that the load or store is in the execution path (assuming there are no intervening exceptions). In this case, the entire cache block containing the target may be loaded into the cache.
- The target memory is caching-inhibited (I = 1), the load or store instruction is in the execution path, and it is guaranteed that no prior instructions can cause an exception.

3.6.5.3 Effects of Out-of-Order Instruction Fetches

To avoid instruction fetch delay, the processor typically fetches instructions ahead of those currently being executed. Such instruction prefetching is said to be out-of-order in that prefetched instructions may not be executed due to intervening branches or exceptions.

During instruction prefetching, no errors except machine check exceptions are reported due to the out-of-order fetching of an instruction until it is known that execution of the instruction is required.

Machine check exceptions resulting solely from out-of-order execution (from nonguarded memory) may be reported. When an out-of-order instruction's result is abandoned, only one side effect (other than a possible machine check) may occur—the referenced bit (R) in the corresponding page table entry can be set due to an out-of-order load operation. See Chapter 4, "Exceptions," for more information on the machine check exception.

3.7 Cache Coherency—MEI Protocol

The primary objective of a coherent memory system is to provide the same image of memory to all devices using the system. Coherency allows synchronization and cooperative use of shared resources. Otherwise, multiple copies of a memory location, some containing stale values, could exist in a system resulting in errors when the stale values are used. Each potential bus master must follow rules for managing the state of its cache.

The 602 cache coherency protocol is a coherent subset of the standard MESI four-state cache protocol that omits the shared state. Since data cannot be shared, the 602 signals all cache block fills as if they were write misses (read-with-intent-to-modify), which flushes the corresponding copies of the data in all caches external to the 602 prior to the 602's cache-block-fill operation. Following the cache-block-load, the 602 is the exclusive owner of the data and may write to it without a bus broadcast transaction.

To maintain this coherency, all global reads observed on the bus by the 602 are snooped as if they were writes, causing the 602 either to write a modified cache block back to memory and invalidate the cache block, or to simply invalidate the cache block if it is unmodified. If the cache block is modified, the block is written back to memory, and the cache block is marked exclusive. If the cache block is marked exclusive when snooped, no bus action is taken, and the cache block remains in the exclusive state. This treatment of caching-inhibited reads decreases the possibility of data thrashing by allowing noncaching devices to read data without invalidating the entry from the 602's data cache.

3.7.1 MEI State Definitions

The 602's data cache characterizes each 32-byte block it contains as being in one of three MEI states. Addresses presented to the cache are indexed into the cache directory with bits A21–A26, and the upper-order 21 bits from the physical address translation (PA0–PA20) are compared against the indexed cache directory tags. If neither of the indexed tags matches, the result is a cache miss. If a tag matches, a cache hit occurred and the directory indicates the state of the cache block through two state bits kept with the tag. The three possible states for a cache block in the cache are the modified state (M), the exclusive state (E), and the invalid state (I). The three MEI states are defined in Table 3-2.

Table 3-2. MEI State Definitions

MEI State	Definition
Modified (M)	The addressed cache block is valid in the cache and only in the cache. The cache block is modified with respect to system memory—that is, the modified data in the cache block has not been written back to memory.
Exclusive (E)	The addressed block is in this cache only. The data in this cache block is consistent with system memory.
Invalid (I)	This state indicates that the addressed cache block is not resident in the cache.

3.7.2 MEI State Diagram

The 602 provides dedicated hardware to provide memory coherency by snooping bus transactions. The address retry capability of the 602 enforces the MEI protocol, as shown in Figure 3-3. Figure 3-3 assumes that the WIM bits for the page or block are set to 001; that is, write-back, caching-not-inhibited, and memory coherency enforced.

Section 3.11, "MEI State Transactions" provides a detailed list of MEI transitions for various operations and WIM bit settings.

BUS TRANSACTIONS

SH =Snoop Hit
RH =Read Hit
RM =Read Miss
WH =Write Hit
WM=Write Miss
SH/CRW=Snoop Hit, Cacheable Read/Write
SH/CIR=Snoop Hit, Cache Inhibited Read

⟨↓⟩= Snoop Push

⟨↑⟩ = Cache Line Fill
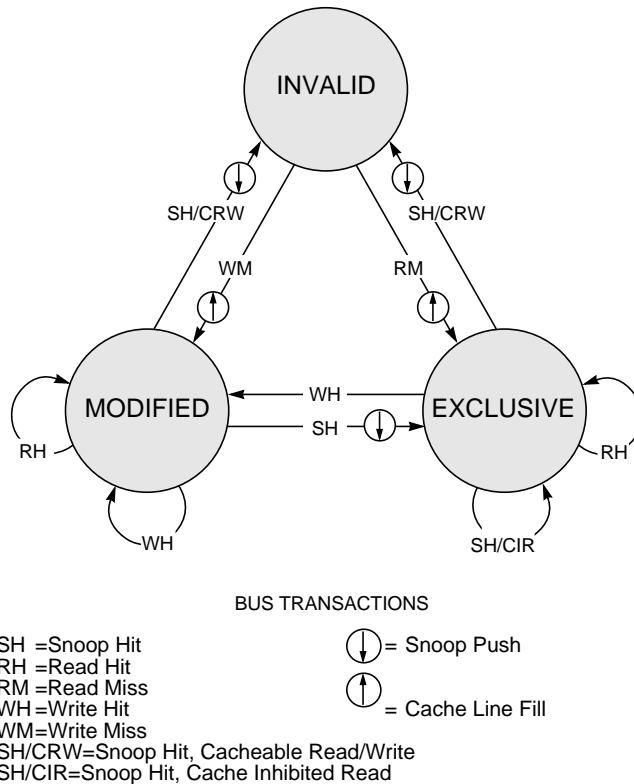
Figure 3-3. MEI Cache Coherency Protocol—State Diagram (WIM = 001)

3.7.3 Compatibility with MESI Protocol

The 602 cache coherency protocol is a coherent subset of the standard MESI four-state protocol. Sharing data with another 602 is not supported. To ensure this, the 602 marks all line-fill operations as if they were write misses (read-with-intent-to-modify). This flushes all external caches prior to the line-fill operation. Following the line-fill operation, the 602 is the exclusive owner of the data and may write to it without a bus broadcast transaction.

Also, all burst reads on the bus are snooped as if they were writes; they flush the 602 cache block. This prevents the 602 from storing a cache block that another cache is trying to fill.

However, if the snooped transaction is a caching-inhibited-read, the 602 does not invalidate its copy of the data. In this case, when the cache block is in the modified state prior to the snoop, the modified data is pushed and the cache block is then marked exclusive. If the cache block is exclusive, the snoop causes no bus action and the cache block remains in the exclusive state. This reduces data thrashing by allowing noncaching devices to read data without completely flushing the entry from the cache. Note that caching-inhibited reads may be either burst or nonburst operations.

3.7.4 Resource Collisions and Retries

Because the 602 data cache tags are single-ported, a resource collision occurs when a snoop is attempted at the same time as a load or store operation. Except for the following cases, the snoop access has highest priority and is given first access to the tags. The load or store access occurs on the clock cycle following the snoop.

However, a snoop is not given priority into the tags when the snoop coincides with a tag write (that is, during validation after a line-fill operation, transition to modified after a first write, etc.). In these situations, the snoop is retried and must re-arbitrate before the lookup is possible.

Occasionally, snoops that hit modified data, which requires a copyback operation, cannot be serviced and must be retried. These retries occur if the cache is busy with a burst read or write at the moment a snoop copyback should begin.

Note that a snoop can hit a modified cache block that is already in the copyback process. If this happens, the 602 retries the snoop.

3.7.5 Page Table Aliasing

If a store operation hits to a page marked write-through and the entry is modified, the page has probably been aliased through another page entry that is marked as write-back. If this occurs, the 602 ignores the modified bit in the cache tag. The cache is updated during the write-through operation and the cache block remains in the modified state.

3.7.6 MEI Hardware Considerations

While the 602 provides the hardware required to monitor bus traffic for coherency, the 602 data cache tags are single ported, and a simultaneous load or store and snoop access represent a resource conflict. In general, the snoop access has highest priority and is given first access to the tags. The load or store access will then occur on the clock following the snoop. The snoop is not given priority into the tags when the snoop coincides with a tag write (that is, validation after a cache block load, transition to E state after a first write, etc.). In these situations, the snoop is retried and must re-arbitrate before the lookup is possible.

Occasionally, cache snoops cannot be serviced and must be retried. These retries occur if the cache is busy with a burst read or write when the snoop operation takes place.

Note that it is possible for a snoop to hit a modified cache block that is already in the process of being written to the copyback buffer for replacement purposes. If this happens, the 602 retries the snoop, and raises the priority of the cast-out operation to allow it to go to the bus before the cache block fill.

The global ($\overline{\text{GBL}}$) signal, asserted as part of the address attribute field during a bus transaction, enables the snooping hardware of the 602. 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 device). If $\overline{\text{GBL}}$ is not asserted for the transaction, that transaction is not

snooped by the 602. Note that the $\overline{\text{GBL}}$ signal is not asserted for instruction fetches, and that $\overline{\text{GBL}}$ is asserted for all data read or write operations when using direct address translation. (Note that direct address translation is referred to as the real addressing mode, not the direct-store segment, in the architecture specification.)

Normally, $\overline{\text{GBL}}$ reflects the M-bit value specified for the memory reference in the corresponding translation descriptor(s). Care must be taken to minimize the number of pages marked as global, because the retry protocol enforces coherency and can use considerable bus bandwidth if much data is shared. Therefore, available bus bandwidth can decrease as more traffic is marked global.

The 602 snoops a transaction if the transfer start ($\overline{\text{TS}}$) and $\overline{\text{GBL}}$ signals are asserted together in the same bus clock (this is a *qualified* snooping condition). No snoop update to the 602 cache occurs if the snooped transaction is not marked global. Also, because cache block cast-outs and snoop pushes do not require snooping, the $\overline{\text{GBL}}$ signal is not asserted for these operations.

When the 602 detects a qualified snoop condition, the address associated with the $\overline{\text{TS}}$ signal is compared with the cache tags. Snooping finishes if no hit is detected. If, however, the address hits in the cache, the 602 reacts according to the MEI protocol shown in Figure 3-3.

3.7.7 Coherency Precautions

The 602 supports a three-state coherency protocol that supports the modified, exclusive, and invalid (MEI) cache states. This protocol is a compatible subset of the MESI four-state protocol and operates coherently in systems that contain four-state caches. In addition, the 602 does not broadcast cache operations caused by cache instructions. They are intended for the management of the local cache but not for other caches in the system.

3.7.7.1 Internal Coherency Paradoxes

The following situations concerning coherency can be encountered within a single-processor system:

- Load or store to a caching-inhibited page (WIM = 0bX1X) and a cache hit occurs

  Caching is inhibited for this page (I = 1)—Load or store operations to a caching-inhibited page that hit in the cache cause a paradox. The 602 ignores any hit to a cache block in memory space marked caching-inhibited. The access is performed on the bus as if there were no hit. The data is not pushed and the cache block is not invalidated.

Note that when WIM bits are changed, it is critical that the cache contents should reflect the new WIM bit settings. For example, if a block or page that had allowed caching becomes caching-inhibited, software should ensure that the appropriate cache blocks are flushed to memory and invalidated.

3.7.8 Load and Store Coherency Summary

Table 3-3 provides a summary of memory coherency actions performed by the 602 on load operations. Noncacheable cases are not part of this table.

Table 3-3. Memory Coherency Actions on Load Operations

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

Table 3-4 provides an overview of memory coherency actions on store operations. This table does not include noncacheable or write-through cases. The read-with-intent-to-modify (RWITM) examples involve selecting a replacement class and casting-out modified data that may have resided in that replacement class.

Table 3-4. Memory Coherency Actions on Store Operations

Cache State	Bus Operation	$\overline{\text{ARTRY}}$	Action
M	None	Don't care	Modify cache
E	None	Don't care	Modify cache, mark M
I	RWITM	Negated	Load data, modify it, mark M
I	RWITM	Asserted	Retry the RWITM

3.7.9 Atomic Memory References

The Load Word and Reserve Indexed (**lwarx**) and Store Word Conditional Indexed (**stwcx.**) instructions provide an atomic update function for a single, aligned word of memory. While an **lwarx** instruction will normally be paired with an **stwcx.** instruction with the same effective address, an **stwcx.** instruction to any address will cancel the reservation. For detailed information on these instructions, refer to Chapter 2, "PowerPC 602 Microprocessor Programming Model," and Chapter 8, "Instruction Set," in *The Programming Environments Manual*.

3.7.10 Cache Reaction to Specific Bus Operations

There are several bus transaction types defined for the 602 bus. The 602 must snoop these transactions and perform the appropriate action to maintain memory coherency as shown in Table 3-5. A processor may assert $\overline{\text{ARTRY}}$ for any bus transaction due to internal conflicts that prevent the appropriate snooping. The transactions in Table 3-5 correspond to the transfer type signals TT0–TT4, which are described in Section 7.2.4.1, "Transfer Type (TT0–TT4)."

Table 3-5. Response to Bus Transactions

Snooped Transaction	602 Response
Clean block	No action is taken.
Flush block	No action is taken.
Write-with-flush Write-with-flush-atomic	Write-with-flush and write-with-flush-atomic operations occur after the processor issues a store or **stwcx.** instruction, respectively. • If the addressed block is in the exclusive state, the address snoop forces the state of the addressed block to invalid. • If the addressed block is in the modified state, the address snoop causes $\overline{\text{ARTRY}}$ to be asserted and initiates a push of the modified block out of the cache and changes the state of the block to invalid.
Kill block	The kill block operation is an address-only bus transaction initiated when a **dcbz** instruction is executed; when snooped by the 602, the addressed cache block is invalidated, and any associated reservation is canceled.
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 cache block is forced to the I state, killing modified data that may have been in the block. Any reservation associated with the block is also cancelled.
Read Read-atomic	The read operation is used by most nonburst and burst read operations on the bus. All burst reads observed on the bus are snooped as if they were writes, causing the addressed cache block to be flushed. A read on the bus with the $\overline{\text{GBL}}$ signal asserted causes the following responses: • If the addressed block in the cache is invalid, the 602 takes no action. • If the addressed block in the cache is in the exclusive state, the block is invalidated. • If f the addressed block in the cache is in the modified state, the block is flushed to memory and the block is invalidated. • If the snooped transaction is a caching-inhibited read, and the block in the cache is in the exclusive state, the snoop causes no bus activity and the block remains in the exclusive state. If the block is in the cache in the modified state, the 602 initiates a push of the modified block out to memory and marks the cache block as exclusive. Read atomic operations appear on the bus in response to **lwarx** instructions and generate the same snooping responses as read operations.
Read-with-intent-to-modify (RWITM) RWITM-atomic	A RWITM operation is issued to acquire exclusive use of a memory location for the purpose of modifying it. • If the addressed block is invalid, the 602 takes no action. • If the addressed cache block is in exclusive state, the 602 initiates an additional snoop action to change the state of the cache block to invalid. • If the addressed cache block is in modified state it is flushed to memory and the block is invalidated. The RWITM atomic operations appear on the bus in response to **stwcx.** instructions and are snooped like RWITM instructions.
sync	No action is taken.
TLB invalidate	No action is taken.

3.7.11 Operations Causing $\overline{\text{ARTRY}}$ Assertion

The following scenarios cause the 602 to assert the $\overline{\text{ARTRY}}$ signal:

- Snoop hits to a block in the M state (optional on kill requests)
- Snoop attempt while the cache is being accessed by a load or store operation
- Snoop hit during a burst load operation
- Snoop hits while a cast-out request is pending during this or the next clock cycle

3.8 Cache Control Instructions

Software must use the appropriate cache management instructions to ensure that caches are kept consistent when data is modified by the processor. When a processor alters a memory location that may be contained in an instruction cache, software must ensure that updates to memory are visible to the instruction fetching mechanism. Although the instructions to enforce coherency vary among implementations and hence operating systems should provide a system service for this function, the following sequence is typical:

1. **dcbst** (update memory)
2. **sync** (wait for update)
3. **icbi** (invalidate copy in cache)
4. **isync** (invalidate copy in own instruction buffer)

These operations are necessary because the processor does not maintain instruction memory coherent with data memory. Software is responsible for enforcing coherency of instruction caches and data memory. Since instruction fetching may bypass the data cache, changes made to items in the data cache may not be reflected in memory until after the instruction fetch completes.

The PowerPC architecture defines instructions for controlling both the instruction and data caches when they exist. The 602 interprets the cache control instructions (**icbi**, **dcbi**, **dcbt**, and **dcbst**) as if they pertain only to the 602's caches. They are not intended for use in managing other caches in the system.

The **dcbz** instruction causes an address-only broadcast on the bus if the contents of the block are from a page marked global through the WIMG bits. This broadcast is performed for coherency reasons; the **dcbz** instruction is the only cache control instruction that can allocate and take new ownership of a cache block. The other instructions do not broadcast either for the purpose of invalidating or flushing other caches in the system or for managing system resources. Any bus activity caused by these instructions is the direct result of performing the operation in the 602 cache. Note that a DSI exception is generated if the effective address of a **dcbi**, **dcbst**, **dcbf**, or **dcbz** instruction cannot be translated due to the lack of a TLB entry. (Note that exceptions are referred to as interrupts in the architecture specification.)

Note that in the PowerPC architecture, the term cache block, or simply block when used in the context of cache implementations, refers to the unit of memory at which coherency is maintained. For the 602 this is the eight-word cache line. This value may be different for other PowerPC implementations. In-depth descriptions of coding these instructions is provided in Section 2.3.6.3, "Memory Control Instructions—OEA."

3.8.1 Data Cache Block Touch (dcbt) Instruction

When a **dcbt** instruction is executed, the effective address is computed, translated, and checked for protection violations as defined in the PowerPC architecture. The behavior of the **dcbt** depends on several circumstances:

- If the effective address of the operation violates page protection or misses in the MMU, the operation is a no-op.
- If the address hits in the cache, no further action is taken.
- If the address misses in the cache and tag is in the modified state, the cache block is written back to memory and the new cache block is brought in and placed in the exclusive state.
- If the address misses in the cache and tag is in the exclusive state, the new cache block is brought in and placed in the exclusive state.
- Address-only transfers are not generated on the external bus (unlike other cache operations).

3.8.2 Data Cache Block Touch for Store (dcbtst) Instruction

The **dcbtst** instruction, like the data cache block touch instruction (**dcbt**), allows software to prefetch a cache block in anticipation of a store operation (read-with-intent-to-modify).

3.8.3 Data Cache Block Set to Zero (dcbz) Instruction

When a **dcbz** instruction is executed, the effective address is computed, translated, and checked for protection violations as defined in the PowerPC architecture. If the EA hits in the cache, four beats of zeros are written into the cache. Also, if M = 1 (coherency enforced), the address is broadcast onto the bus prior to the zero line fill.

The exception priorities (from highest to lowest) for **dcbz** are as follows;

1. Attempt to execute **dcbz** while in user mode—program exception
2. BAT protection violation—DSI exception
3. MMU miss—DTLB exception
4. Cache disabled—Alignment exception
5. Page marked write-through or caching-inhibited—Alignment exception
6. TLB protection violation—DSI exception

The **dcbz** instruction is the only cache instruction that the 602 broadcasts. This is done to maintain coherency with other cache devices in the system.

3.8.4 Data Cache Block Invalidate (dcbi) Instruction

When a **dcbi** instruction is executed, the effective address is computed, translated, and checked for protection violations as defined in the PowerPC architecture. If the addressed line is present in the cache, then the 602 marks this data as invalid regardless of whether the data is exclusive or modified.

The exception priorities (from highest to lowest) for **dcbi** are as follows;

1. BAT protection violation—DSI exception
2. MMU miss—DTLB exception
3. TLB protection violation—DSI exception

3.8.5 Data Cache Block Store (dcbst) Instruction

When a **dcbst** instruction is executed, the effective address is computed, translated and checked for protection violations as defined in the PowerPC architecture. The resulting actions are as follows:

*   If the address hits in the cache and the tag is in the exclusive state, no further action is taken.
*   If the address hits in the cache and is in the modified state, the contents of the cache block are written back to memory and the cache block is placed in the exclusive state.
*   Address-only transfers are not generated on the external bus.

The exception priorities (from highest to lowest) for **dcbst** are as follows;

1. BAT protection violation—DSI exception
2. MMU miss—DTLB exception
3. TLB protection violation—DSI exception

3.8.6 Data Cache Block Flush (dcbf) Instruction

When the **dcbf** instruction is executed, the effective address is computed, translated, and checked for protection violations as defined in the PowerPC architecture. The action taken depends on the memory mode associated with the target, and on the state of the cache block. The following list describes the action taken for the various cases. The actions described are executed regardless of whether the page containing the addressed byte is in caching-inhibited or caching-allowed mode. The following actions occur in both coherency-required mode (WIM = 0bXX1) and coherency-not-required mode (WIM = 0bXX0).

The effect of this instruction is as follows:

*   If the address hits in the cache, and the cache block is marked modified, the contents are written back to memory and the cache entry is invalidated.
*   If the address hits in the cache, and the line is marked exclusive, the cache entry is invalidated.
*   If the address misses in the cache, no further action is taken.

The exception priorities (from highest to lowest) for **dcbf** are as follows;

1. BAT protection violation—DSI exception
2. MMU miss—DTLB exception
3. TLB protection violation—DSI exception

3.8.7 Enforce In-Order Execution of I/O Instruction (eieio)

The **eieio** instruction is used to order memory accesses. Since the 602 instruction does not reorder noncacheable memory accesses, the **eieio** instruction is treated as a no-op instruction.

3.8.8 Instruction Cache Block Invalidate (icbi) Instruction

The **icbi** instruction performs a virtual lookup into the instruction cache (index only). The address is not translated and as such, cannot generate an exception. Both ways of the selected set are invalidated. This instruction is not broadcast onto the external bus.

3.8.9 Instruction Synchronize (isync) Instruction

The **isync** instruction waits for all previous instructions to complete and then discards any previously-fetched instructions, causing subsequent instructions to be fetched (or refetched) from memory and to execute in the context established by the previous instructions. This instruction has no effect on other processors or on their caches. Any instruction after an **isync** will see all effects of prior instructions.

3.8.10 Synchronize (sync) Instruction

The **sync** instruction waits for all previous instructions and all previous bus operations (except already queued instruction fetches) to complete before allowing any following bus activity to be initiated.

3.9 Bus Operations Caused by Cache Control Instructions

Table 3-6 provides an overview of the bus operations initiated by cache control instructions. The cache control, TLB management, and synchronization instructions supported by the 602 may affect or be affected by the operation of the bus. None of the instructions will actively broadcast through address-only transactions on the bus (except for **dcbz**), and no broadcasts by other masters are snooped by the 602 (except for kills). The operation of the instructions, however, may indirectly cause bus transactions to be performed, or their completion may be linked to the bus. The following table summarizes how these instructions may operate with respect to the bus.

Note that Table 3-6 assumes that the WIM bits are set to 001; that is, the cache is operating in write-back mode, caching is permitted and coherency is enforced.

Table 3-6. Bus Operations Caused by Cache Control Instructions (WIM = 001)

Operation	Cache State	Next State	Bus Operations	Comment
sync	Don't care	No change	None	Waits for memory queues to complete bus activity
icbi	Don't care	I	None	—
dcbi	Don't care	I	None	—
dcbf	I, E	I	None	—
	M	I	Write-with-kill	Block is pushed
dcbst	I, E	No change	None	—
	M	E	Write	Block is pushed
dcbz	I	M	Write-with-kill	—
	E, M	M	Kill block	Writes over modified data
dcbt, **dcbtst**	I	E	Read-with-intent-to-modify	—
	E	E	Read-with-intent-to-modify	Replace old data
	M	E	Write, Read-with-intent-to-modify	Block is pushed then updated

Table 3-6 does not include noncacheable or write-through cases, nor does it completely describe the mechanisms for the operations described. For more information, see Section 3.11, "MEI State Transactions."

For detailed information on the cache control instructions, refer to Section 2.3.6.3, "Memory Control Instructions—OEA," and to Chapter 8, "Instruction Set," in *The Programming Environments Manual*. The 602 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 602, see Chapter 8, "System Interface Operation."

3.10 Bus Interface

The bus interface buffers bus requests from the instruction and data caches, and executes the requests per the 602 bus protocol. It includes address register queues, prioritization logic, and bus control logic. The bus interface also captures snoop addresses for snooping in the cache and in the address register queues, and snoops for reservations. All data storage for the address register buffers (load and store data buffers) are located in the cache section. The data buffers are considered temporary storage for the cache and not part of the bus interface.

The general functions and features of the bus interface are as follows:

- Four address register queues:
  — Instruction cache load address
  — Data cache load address
  — Data cache nonburst store address
  — Data cache cast-out/store address (associated data cache block buffer located in cache)
- Reservation bit for **lwarx**/**stwcx.** instructions
- Prefetch line-fill address during copy-back transaction.

A conceptual block diagram of the bus interface is shown in Figure 3-4. The address register queues in the figure hold transaction requests that the bus interface may issue on the bus independently of the other requests. Only one transaction may appear on the bus at a time.
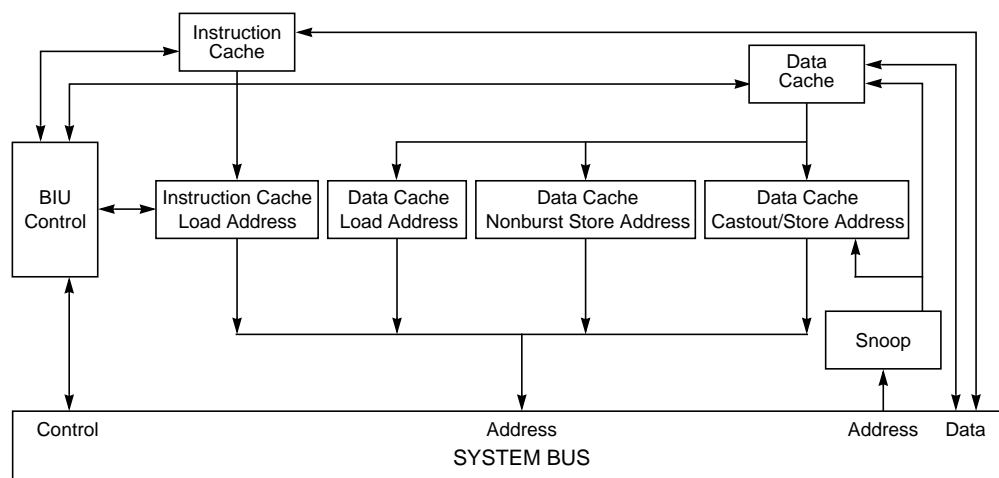
Figure 3-4. Bus Interface Address Buffers

For additional information about the 602 bus interface and the bus protocols, refer to Chapter 8, "System Interface Operation."

3.11 MEI State Transactions

Table 3-7 shows MEI state transitions for various operations. Bus operations are described in Table 3-5.

Table 3-7. MEI State Transitions

Operation	Cache Operation	Bus sync	WIM	Current State	Next State	Cache Actions	Bus Operation
Load (T = 0)	Read	No	x0x	I	Same	1 Cast out of modified block (as required)	Write-with-kill
						2 Pass four-beat read to memory queue	Read
		No	x0x	E, M	Same	Read data from cache	—
		No	x1x	I	Same	Pass nonburst read to memory queue	Read
		No	x1x	E	I	CRTRY read	—
		No	x1x	M	I	CRTRY read (push sector to write queue)	Write-with-kill
lwarx	Read	Acts like other reads but bus operation uses special encoding.					
Store (T = 0)	Write	No	00x	I	Same	1 Cast out of modified block (if necessary)	Write-with-kill
						2 Pass RWITM to memory queue	RWITM
		No	00x	E, M	M	Write data to cache	—
Store ≠ **stwcx** (T = 0)	Write	No	10x	I	Same	Pass nonburst write to memory queue	Write-with-flush
		No	10x	E	Same	1 Write data to cache	—
						2 Pass nonburst write to memory queue	Write-with-flush
		No	10x	M	E	1 CRTRY write	—
						2 Push block to write queue	Write-with-kill
Store (T = 0) or **stwcx.** (WIM = 10x)	Write	No	x1x	I	Same	Pass nonburst write to memory queue	Write-with-flush
		No	x1x	E	I	CRTRY write	—
		No	x1x	M	I	1 CRTRY write	—
						2 Push block to write queue	Write-with-kill
stwcx.	Conditional write	If the reserved bit is set, this operation is like other writes except the bus operation uses a special encoding.					

Table 3-7. MEI State Transitions (Continued)

Operation	Cache Operation	Bus sync	WIM	Current State	Next State	Cache Actions	Bus Operation
dcbf	Data cache block flush	No	xxx	I, E	Same	1 CRTRY **dcbf**	—
						2 Pass flush	Flush
				Same	I	3 State change only	—
		No	xxx	M	I	Push block to write queue	Write-with-kill
dcbst	Data cache block store	No	xxx	I, E	Same	1 CRTRY **dcbst**	—
						2 Pass clean	Clean
				Same	Same	3 No action	—
		No	xxx	M	E	Push block to write queue	Write-with-kill
dcbz	Data cache block set to zero	No	x1x	x	x	Alignment trap	—
		No	10x	x	x	Alignment trap	—
		Yes	00x	I	Same	1 CRTRY **dcbz**	—
						2 Cast out of modified block	Write-with-kill
						3 Pass kill	Kill
				Same	M	4 Clear block	—
		No	00x	E, M	M	Clear block	—
dcbt	Data cache block touch	No	x1x	I	Same	Pass nonburst read to memory queue	Read
		No	x1x	E	I	CRTRY read	—
		No	x1x	M	I	1 CRTRY read	—
						2 Push block to write queue	Write-with-kill
		No	x0x	I	Same	1 Cast out of modified block (as required)	Write-with-kill
						2 Pass burst read to memory queue	Read
		No	x0x	E,M	Same	No action	—
Nonburst read	Reload dump 1	No	xxx	I	Same	Forward data_in	—
Burst read (double-word-aligned)	Reload dump	No	xxx	I	E	Write data_in to cache	—
Burst write (double-word-aligned)	Reload dump	No	xxx	I	M	Write data_in to cache	—

Table 3-7. MEI State Transitions (Continued)

Operation	Cache Operation	Bus sync	WIM	Current State	Next State	Cache Actions	Bus Operation
E→I	Snoop write or kill	No	xxx	E	I	State change only (committed)	—
M→I	Snoop kill	No	xxx	M	I	State change only (committed)	—
Push M→I	Snoop flush	No	xxx	M	I	Conditionally push	Write-with-kill
Push M→E	Snoop clean	No	xxx	M	E	Conditionally push	Write-with-kill
tlbie	TLB invalidate	No	xxx	x	x	1 CRTRY TLBI	—
						2 Pass TLBI	—
						3 No action	—
sync	Synchroni-zation	No	xxx	x	x	1 CRTRY **sync**	—
						2 Pass **sync**	—
						3 No action	—

Chapter 4
Exceptions

The PowerPC exception mechanism allows the processor to change to supervisor state (referred to as privileged state in the architecture specification) as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When exceptions (referred to as interrupts in the architecture specification) 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 occurs in supervisor mode.

The PowerPC 602 microprocessor also provides an additional mechanism, not defined by the PowerPC architecture, for entering and exiting supervisor mode without taking an exception. Two user-level instructions, Enable Supervisor Access (**esa**) and Disable Supervisor Access (**dsa**) and several registers—ESA enable base register (SEBR), ESA enable register (SER), and ESA save and restore register (ESASRR)—are implemented to support this functionality. For information about the **esa** and **dsa** instructions, see Section 2.3.7, "PowerPC 602 Implementation-Specific Instructions," and for information about the SER, SEBR, and ESASRR registers, see Section 2.1.2, "PowerPC 602 Processor-Specific Registers."

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 or the FPSCR. Additionally, certain 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 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. An instruction is said to have "completed" when the results of that instruction's execution have been committed to the registers defined by the architecture (for example, the GPRs or FPRs, rather than rename buffers). If a single instruction encounters multiple exception conditions, those exceptions are taken in order of priority. Likewise, exceptions that are asynchronous are recognized when they occur, but are not handled until the next instruction to complete in program order successfully

completes. Throughout this chapter, the term "next instruction" implies the next instruction to complete in program order.

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 to allow control to ultimately return to the original excepting program.

In many cases, after an 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.

Exception handlers should 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. The information should be saved before enabling any exception that is automatically disabled when an exception is 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 routing is executed in supervisor mode.
Handling	Exception handling is performed by the software linked to the appropriate vector offset. Exception handling is performed at supervisor-level.

4.1 Exception Classes

The PowerPC architecture supports four types of exceptions:

- Synchronous, precise—These are caused by instructions. All instruction-caused exceptions are handled precisely; that is, the machine state at the time the exception occurs is known and can be completely restored. This means that (excluding the system call exceptions) the address of the faulting instruction is provided to the exception handler and that neither the faulting instruction nor subsequent instructions in the code stream will complete before the exception is taken. Once the exception is processed, execution resumes at the address of the faulting instruction (or at an alternate address provided by the exception handler). When an exception is taken due to a trap or system call instruction, execution resumes at an address provided by the handler.

- Synchronous, imprecise—The PowerPC architecture defines two imprecise floating-point exception modes, recoverable and nonrecoverable. Even though the 602 provides a means to enable the imprecise modes, it implements these modes identically to the precise mode (that is, floating-point enabled exceptions are always precise on the 602).

- Asynchronous, maskable—The external interrupt, system management interrupt (SMI), decrementer interrupt, and watchdog timer interrupt are maskable asynchronous exceptions. When these exceptions occur, their handling is postponed until the next instruction, and any exceptions associated with that instruction, completes. If there are no instructions in the execution units, the exception is taken immediately upon determination of the correct restart address (for loading SRR0). These exceptions are maskable in that the processor can be disabled by setting MSR[EE]. This is described in Section 4.2.1, "Enabling and Disabling Exceptions."

- Asynchronous, nonmaskable—There are two nonmaskable, asynchronous exceptions: system reset and the machine check exception. These exceptions may not be recoverable, or may provide a limited degree of recoverability. All exceptions report recoverability through the MSR[RI] bit.

The 602 exception classes are shown in Table 4-1.

Table 4-1. PowerPC 602 Microprocessor Exception Classifications

Type	Exception
Asynchronous, nonmaskable	Machine check System reset
Asynchronous, maskable	External interrupt Decrementer System management interrupt Watchdog timer interrupt
Synchronous, precise	Instruction-caused exceptions

Note that Table 4-1 includes no synchronous imprecise instructions. While the PowerPC architecture supports imprecise handling of floating-point exceptions, the 602 implements these exception modes as precise exceptions.

Although the PowerPC architecture specifies that the recognition of the machine check exception is nonmaskable, on the 602 the stimuli that cause this exception are maskable. For example, the machine check exception is caused by assertion of $\overline{\text{TEA}}$ or $\overline{\text{MCP}}$. However, the $\overline{\text{MCP}}$ signal can be disabled by HID0[0]. Therefore, the machine check caused by $\overline{\text{TEA}}$ is the only truly nonmaskable machine check exception.

The 602's exceptions, and conditions that cause them, are listed in Table 4-2. Exceptions that are specific to the 602 are indicated.

Table 4-2. Exceptions and Conditions

Exception Type	Vector (hexadecimal)			Causing Conditions
	Prefix		Offset	
	IP = 0	IP = 1		
Reserved	—	—	0000	—
System reset (Hard reset)	FFF0		0100	Assertion of $\overline{\text{HRESET}}$
System reset (Soft reset)	0000	FFF0	0100	Assertion of $\overline{\text{SRESET}}$
Machine check	0000	FFF0	0200	Assertion of $\overline{\text{TEA}}$ during a data transaction; assertion of $\overline{\text{MCP}}$.
DSI	IBR	FFF0	0300	Determined by the bit settings in the DSISR, as follows: 4 Set if a memory access is not permitted by the page or DBAT protection mechanism; otherwise cleared. 5 Set only if memory access is attempted and SR[T] = 1. The 602 does not support direct-store memory. 6 Set for a store operation and cleared for a load operation.
ISI	IBR	FFF0	0400	An instruction cannot be fetched for one of the following reasons: • The EA cannot be translated and an ISI exception must be taken to load the PTE (and possibly the page) into memory. • The fetch access violates memory protection. If SR[Ks] and SR[Kp] and PTE[PP] are set to prohibit read access, instructions cannot be fetched from this location.
External interrupt	IBR	FFF0	0500	MSR[EE] = 1 and the $\overline{\text{INT}}$ signal is asserted.
Alignment	IBR	FFF0	0600	Memory cannot be accessed for one of the following reasons: • The operand of a floating-point load or store is not word-aligned. • The operand of **lmw**, **stmw**, **lwarx**, or **stwcx.** is not word-aligned. • The operand of **dcbz** is in a page marked write-through or caching-inhibited, for a virtual mode access. • A little-endian access is misaligned, or a multiple access is attempted with the little-endian bit set.

Table 4-2. Exceptions and Conditions (Continued)

Exception Type	Vector (hexadecimal)			Causing Conditions
	Prefix		Offset	
	IP = 0	IP = 1		
Program	IBR	FFF0	0700	The following conditions correspond to bit settings in SRR1 and arise during execution of an instruction: • Floating-point enabled exception—The following is met: (MSR[FE0] | MSR[FE1]) & FPSCR[FEX] is 1. FPSCR[FEX] is set by a floating-point instruction that causes an enabled exception or by the execution of one of the "move to FPSCR" instructions that results in both an exception condition bit and its corresponding enable bit being set in the FPSCR. • Illegal instruction—Execution of an instruction is attempted with an illegal opcode or combination of opcode and extended opcode (including PowerPC instructions not implemented in the 602 but not including those optional instructions treated as no-ops). • Privileged instruction—Execution of a privileged instruction is attempted and MSR[PR] = 1. In the 602, this exception is generated for **mtspr** or **mfspr** with an invalid SPR field if SPR[0] = 1 and MSR[PR] = 1. This may not be true for all PowerPC processors. • Trap— Generated when a trap instruction condition is met.
Floating-point unavailable	IBR	FFF0	0800	An attempt to execute a floating-point instruction (including floating-point load, store, or move instructions) when the floating-point available bit is disabled, (MSR[FP] = 0).
Decrementer	IBR	FFF0	0900	The most significant bit of the decrementer (DEC) register changes from 0 to 1. Must be enabled with the MSR[EE] bit.
Reserved	IBR	FFF0	0A00–0BFF	—
System call	IBR	FFF0	0C00	Execution of the System Call (**sc**) instruction
Trace	IBR	FFF0	0D00	MSR[SE] =1 or when a completing instruction is a branch and MSR[BE] =1.
Floating-point assist	IBR	FFF0	0E00	Not implemented in the 602
Reserved	—	—	0E10–0FFF	—
Instruction translation miss	IBR	FFF0	1000	The ITLB cannot translate the EA for an instruction fetch.
Data load translation miss	IBR	FFF0	1100	An EA for a data load cannot be translated by the DTLB.
Data store translation miss	IBR	FFF0	1200	An EA for a data store cannot be translated by the DTLB; or when a DTLB hit occurs and the change bit in the PTE must be set due to a data store operation.

Table 4-2. Exceptions and Conditions (Continued)

Exception Type	Vector (hexadecimal)			Causing Conditions
	Prefix		Offset	
	IP = 0	IP = 1		
Instruction address breakpoint	0000	FFF0	1300	The address (bits 0–29) in the IABR matches the next instruction to complete in the completion unit and the IABR enable bit (bit 30) is set.
System management interrupt	IBR	FFF0	1400	MSR[EE] =1 and the $\overline{\text{SMI}}$ input signal is asserted.
Watchdog timer	IBR	FFF0	1500	A carry occurs out of a bit specified by the user. If the watchdog timer is not reset by the interrupt service routine, a second watchdog timer exception forces an internal reset.
Emulation trap	IBR	FFF0	1600	A double-precision floating-point instruction or a load/store string instruction is encountered.
Reserved	—	—	1700–2FFF	—

Exceptions are roughly prioritized by exception class, as follows:

1. Nonmaskable, asynchronous exceptions have priority over all other exceptions—system reset and machine check exceptions (although the machine check exception condition can be disabled so the condition causes the processor to go directly into the checkstop state). These exceptions cannot be delayed, and do not wait for the completion of any precise exception handling.

2. Synchronous, precise exceptions are caused by instructions and are taken in strict program order.

3. Imprecise exceptions (imprecise mode floating-point enabled exceptions) are caused by instructions and they are delayed until higher priority exceptions are taken.

4. Maskable asynchronous exceptions (external interrupt and decrementer exceptions) are delayed until higher priority exceptions are taken.

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

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 4, "Exceptions," in *The Programming Environments Manual*.

4.1.1 Exception Priorities

The exceptions are listed in Table 4-3 in order of highest to lowest priority.

Table 4-3. Exception Priorities

Category	Priority	Exception	Cause
Asynchronous	0	System reset	$\overline{\text{HRESET}}$ (hard reset) or power-on reset
	1	Machine check	$\overline{\text{TEA}}$ or $\overline{\text{MCP}}$ (soft reset)
	2	System reset	$\overline{\text{SRESET}}$
	3	System management interrupt	$\overline{\text{SMI}}$
	4	External interrupt	$\overline{\text{INT}}$
	5	Decrementer	Decrementer passed through 0x0000_0000
	6	Watchdog timer	See Section 4.5.17, "Watchdog Timer Interrupt (0x1500)."
Instruction fetch	0	ITLB miss	See Section 4.5.12, "Instruction TLB Miss Exception (0x1000)."
	1	ISI	See Section 4.5.4, "ISI Exception (0x0400)."
Instruction dispatch/ execution	0	IABR	See Section 4.5.15, "Instruction Address Breakpoint Exception (0x1300)."
	1	Program	Illegal, privileged, or trap instruction
	2	System call	See Section 4.5.10, "System Call Exception (0x0C00)."
	3	Floating-point unavailable	See Section 4.5.8, "Floating-Point Unavailable Exception (0x0800)."
	4	Program	Floating-point enabled exception condition
	5	Alignment	• Floating-point operand not word-aligned • **lmw**, **stmw**, **lwarx**, or **stwcx.** not word-aligned • Little-endian access is misaligned • Multiple access with little-endian bit set • **dcbz** to W=1 or I =1 space
	6	DSI	• BAT page protection violation • An attempt to access memory for which SR[T] = 1.
	7	DTLB miss	Store or load miss. A store miss can jump to a DSI routine.
	8	DSI	TLB page protection violation
	9	DTLB miss	Change bit not set on a store operation
Post-instruction execution	0	Trace	See Section 4.5.11, "Trace Exception (0x0D00)." • MSR[SE] = 1 • MSR[BE] = 1 for branches

4.1.2 Summary of Front-End Exception Handling

The following list of interrupt categories describes how the 602 handles exceptions up to the point of signaling the appropriate exception to occur. Note that a recoverable state is reached if the completed store queue is empty (drained, not canceled) and any instruction that is next in program order and has been signaled to complete has completed. If MSR[RI] is clear, the 602 is in a nonrecoverable state by default. Also, completion of an instruction is defined as performing all architectural register writes associated with that instruction, and then removing that instruction from the completion buffer queue.

- Asynchronous nonmaskable nonrecoverable—(System reset caused by the assertion of either HRESET or internally during power-on reset (POR)). These interrupts have highest priority and are taken immediately regardless of other pending exceptions or recoverability. An address of an instruction that will not take an exception is guaranteed.

- Asynchronous maskable nonrecoverable—(Machine check). A machine check exception takes priority over any other pending exception except a nonrecoverable system reset caused. A machine check exception is taken immediately regardless of recoverability. An address of an instruction that will not take an exception is guaranteed.

- Asynchronous nonmaskable recoverable—(System reset caused by the assertion of SRESET). This interrupt takes priority over any other pending exception except nonrecoverable exceptions listed above. It is taken immediately when a recoverable state is reached.

- Asynchronous maskable recoverable—(System management interrupt, external interrupt, decrementer interrupt). Before handling this type of exception, the next instruction in program order must complete or except. If this action causes another type of exception, that exception is taken and the asynchronous maskable recoverable exception remains pending. Once an instruction can complete without causing an exception, further instruction completion is halted while the untaken exception remains pending. The exception is taken when a recoverable state is reached.

- Instruction fetch–(ITLB, ISI). When this type of exception is detected, dispatch is halted and the current instruction stream is allowed to drain. If completing any instructions in this stream causes an exception, that exception is taken and the instruction fetch exception is forgotten. Otherwise, as soon as the machine is empty and a recoverable state is reached, the instruction fetch exception is taken.

- Instruction dispatch/execution—(Program, DSI, alignment, emulation trap, system call, DTLB miss on load or store, IABR). This type of exception is determined at dispatch or execution of an instruction. The exception remains pending until all instructions in program order, before the exception-causing instruction, are completed. The exception is then taken without completing the exception-causing instruction. If any other exception condition is created in completing these previous instructions in the machine, that exception takes priority over the pending instruction dispatch/execution exception, which will then be forgotten.

- Post-instruction execution—(Trace). This type of exception is generated following execution and completion of an instruction while a trace mode is enabled. If executing the instruction produces conditions for another type of interrupt, that exception is taken and the post-instruction execution exception is forgotten for that instruction.

4.2 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 (MSR) for user-level mode (referred to as problem mode in the architecture specification) and to identify where instruction execution should resume after the exception is handled.

When an exception occurs, SRR0 is set to point to the instruction at which instruction processing should resume when the exception handler returns control to the interrupted process. All instructions in the program flow preceding this one will have completed and no subsequent instruction will have completed. This may be the address of the instruction that caused the exception or the next one (as in the case of a system call exception). The instruction addressed can be determined from the exception type and status bits. This address is used to resume instruction processing in the interrupted process, typically when an **rfi** instruction is executed. The SRR0 register is shown in Figure 4-1.

SRR0 (holds EA for resuming program execution)
0   31
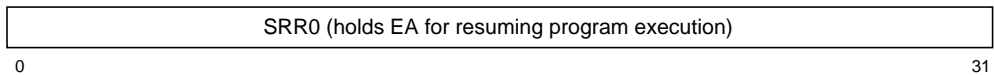
Figure 4-1. Machine Status Save/Restore Register 0

The save/restore register 1(SRR1) is used to save machine status (the contents of the MSR) on exceptions and to restore those values when **rfi** is executed. SRR1 is shown in Figure 4-2.

Exception specific information and MSR bit values
0   31
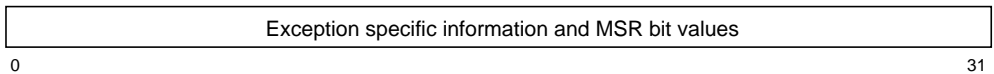
Figure 4-2. Machine Status Save/Restore Register 1

Typically, when an exception occurs, bits 0–15 of SRR1 are loaded with exception-specific information and bits 16–31 of MSR are placed into the corresponding bit positions of SRR1. The 602 loads SRR1 with specific bits for handling machine check exceptions, as shown in Table 4-4.

Table 4-4. SRR1 Bit Settings for Machine Check Exceptions

Bits	Name	Description
0	MSR[0]	Copy of MSR bit 0
1–4	—	Reserved
5–9	MSR[5–9]	Copy of MSR bits 5–9
10–11	—	Reserved
12	MCP	Machine check signal
13	TEA	TEA error
14–15	—	Reserved
16–31	MSR[16–31]	Copy of MSR bits16–31

The 602 loads SRR1 with specific bits for handling the three TLB miss exceptions, as shown in Table 4-5.

Table 4-5. SRR1 Bit Settings for Software Table Search Operations

Bits	Name	Description
0–3	CRF0	Copy of condition register field 0 (CR0)
4	—	Reserved
5–9	MSR[5–9]	Copy of MSR bits 5–9
10–12	—	Reserved
13	I/D	Instruction/data TLB miss 0 DTLB miss 1 ITLB miss
14	WAY	Which TLB associativity set should be replaced 0 Set 0 1 Set 1
15	S/L	Store/load protection instruction 0 Load miss 1 Store miss
16–31	MSR[16–31]	Copy of MSR bits16–31

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

The MSR is shown in Figure 4-3. When an exception occurs, MSR bits, as described in Table 4-6, are altered as determined by the exception.

TGPR
POW

☐ Reserved

| 0 0 0 0 0 0 0 0 | AP | SA | 0 0 | | | ILE | EE | PR | FP | ME | FE0 | SE | BE | FE1 | 0 | IP | IR | DR | 0 0 | RI | LE |

0   7   8   9   10 12 13 14 15   16   17   18   19   20   21   22   23   24 25 26 27 28 29 30 31
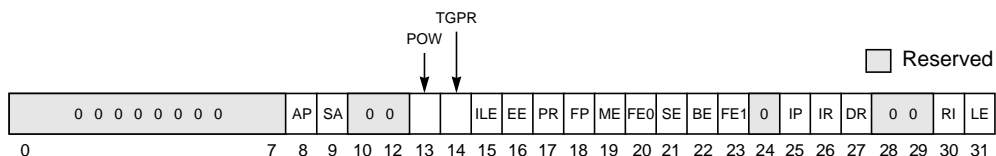
Figure 4-3. Machine State Register (MSR)

Table 4-6 shows the bit definitions for the MSR.

Table 4-6. MSR Bit Settings

Bit(s)	Name	Description
0	—	Reserved, but saved in SRR1 when an exception occurs
1–4	—	Reserved
5–7	—	Reserved, but saved in SRR1 when an exception occurs
8	AP	Access privilege state. (602-specific—not defined by the PowerPC architecture). Like MSR[PR], but only affects access permission if PR = 0. This bit is checked if and only if MSR[PR] = 0. MSR[AP] restricts access privilege if PR = 0. Setting MSR[AP] affects access to the instruction and data locations as if it were PR.
9	SA	Supervisor access mode. (602-specific—not defined by the PowerPC architecture). If this field is set, it allows execution of supervisor instructions without entering supervisor mode.
10–12	—	Reserved
13	POW	Power management enable (602-specific—not defined by the PowerPC architecture) 0 Disables programmable power modes (normal operation mode). 1 Enables programmable power modes (Nap, Doze, or Sleep mode). This bit controls the programmable power modes only, it has no effect on dynamic power management (DPM). MSR[POW] may be altered with an **mtmsr** instruction only. Also, when altering the POW bit, software may alter only this bit in the MSR and no others. The **mtmsr** instruction must be followed by a context-synchronizing instruction. See Chapter 9, "Power Management," for more information.
14	TGPR	Temporary GPR remapping (602-specific—not defined by the PowerPC architecture) 0 Normal operation 1 TGPR mode. GPR0–GPR3 are remapped to TGPR0–TGPR3 for use by TLB miss routines. The contents of GPR0–GPR3 remain unchanged while MSR[TGPR] = 1. Attempts to use GPR4–GPR31 with MSR[TGPR] = 1 yield undefined results. Overlays TGPR0–TGPR3 over GPR0–GPR3 for use by TLB miss routines. When this bit is set, all instruction accesses to GPR0–GPR3 are mapped to TGPR0–TGPR3, respectively. The contents of GPR0–GPR3 are unchanged while as long as this bit remains set. Attempts to use GPR4–GPR31 when this bit is set yields undefined results.The TGPR bit is set when either an instruction TLB miss, data read miss, or data write miss exception is taken. The TGPR bit is cleared by an **rfi** instruction. This bit is 602-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.

Table 4-6. MSR Bit Settings (Continued)

Bit(s)	Name	Description
16	EE	External interrupt enable 0 The processor ignores external interrupts, system management interrupts, and decrementer interrupts. 1 The processor is enabled to take an external interrupt, system management interrupt, or decrementer interrupt.
17	PR	Privilege level 0 The processor can execute both user- and supervisor-level instructions. 1 The processor can only execute user-level instructions.
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	Floating-point exception mode 0 (see Table 4-7).
21	SE	Single-step trace enable 0 The processor executes instructions normally. 1 The processor generates a trace exception upon the successful completion of the next instruction.
22	BE	Branch trace enable 0 The processor executes branch instructions normally. 1 The processor generates a trace exception upon the successful completion of a branch instruction.
23	FE1	Floating-point exception mode 1 (see Table 4-7)
24	—	Reserved, but saved in SRR1 when an exception occurs
25	IP	Interrupt prefix. The functionality of this bit is enhanced in the 602. How this bit is interpreted depends on the exception that is taken. • If a soft reset, machine check, or instruction address breakpoint exception is taken, the IP is used as it is defined by the PowerPC architecture. That is, if IP = 0, the vector address is determined by prepending 0x0000 to the vector offset. If IP is set, the vector address is determined by prepending the vector offset with 0xFFF0. • If a hard reset is taken, the vector address is always 0xFFF0_0100. • For all other exceptions, if the IP bit is cleared, the vector address is determined by prepending the contents of the IBR to the vector offset. If IP is set, the vector address is determined by prepending 0xFFF0 to the vector offset.
26	IR	Instruction address translation 0 Instruction address translation is disabled. 1 Instruction address translation is enabled. The 602 implements this bit as defined by the PowerPC architecture. Turns on instruction address translation, protections, and cache control. The DR and IR bits operate as on PowerPC; however, if HID0[SB] is set, the BAT arrays are to be used for translation, cache control, and protection checking if IR or DR are cleared for the specific access. If IR or DR are set, the BAT/TLB hit mechanisms take priority. For more information, see Chapter 5, "Memory Management."

Table 4-6. MSR Bit Settings (Continued)

Bit(s)	Name	Description
27	DR	Data address translation 0 Data address translation is disabled. 1 Data address translation is enabled. As defined by the PowerPC architecture. Turns on data address translation, protections, and cache control. The DR and IR bits operate as defined on PowerPC architecture; however, if HID0[SB] is set, the BAT arrays are to be used for translation, cache control, and protection checking if IR or DR are cleared for the specific access. If IR or DR are set, the BAT/TLB hit mechanisms take priority. For more information see Chapter 5, "Memory Management."
28–29	—	Reserved, but saved in SRR1 when an exception occurs

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 602 are shown in Table 4-7. For further details, see Chapter 6, "Exceptions," of *The Programming Environments Manual*.

Table 4-7. IEEE Floating-Point Exception Mode Bits

FE0	FE1	Mode
0	0	Floating-point exceptions disabled
0	1	Floating-point imprecise nonrecoverable. In the 602, this bit setting causes the PowerPC 604™ microprocessor to operate in floating-point precise mode.
1	0	Floating-point imprecise recoverable. In the 602, 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.

The vector address can be affected by the value in the interrupt base register (IBR). The IBR is used to store a 16-bit base address for most 602 exception vectors. The 16-bit base address is concatenated with the exception vector offset to form the address for the exception handler. The IBR can be read and written to by the processor. See Figure 2-17 for the format of this register. If MSR[IP] is set, the exception vector prefix is 0xFFF0 (following the format of the IBR). If MSR[IP] is not set, the value in the IBR is used as the 16-bit prefix. The IBR is cleared and MSR[IP] is set on a power-on reset; therefore, the system reset exception vector on a power-on reset is 0xFFF00100. For more information, see Section 2.1.2.4.3, "Interrupt Base Register (IBR)."

4.2.1 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 as follows:

- 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 Section 2.1.2.1.1, "Hardware Implementation Register 0 (HID0)."

- System reset exceptions cannot be masked.

4.2.2 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 determined by the exception. See the individual exception description for details about how SRR0 is used.

2. Bits 1–4 and 10–15 of SRR1 are loaded with information specific to the exception type. Additional bits may also be loaded.

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 4-6. 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 the exception that is taken, the value in MSR[IP], and the interrupt base register (IBR).

   — Soft reset, machine check, or instruction address breakpoint exception—The IP bit is used as it is defined by the PowerPC architecture. That is, if IP = 0, the

vector address is determined by prepending 0x0000 to the vector offset. If IP is set, the vector address is determined by prepending the vector offset with 0xFFF0.

— Hard reset—The vector address is always 0xFFF0_0100.

— For all other exceptions, if IP is cleared, the vector address is determined by prepending the contents of the IBR to the vector offset. If IP is set, the vector address is determined by prepending 0xFFF0 to the vector offset.

4.2.3 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**.

Note 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.

4.2.4 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.

- 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.

- Branch to the address contained in SRR0.

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

4.3 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 more information about using 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 4.2.3, "Setting MSR[RI]."

4.4 Exception Latencies

Latencies for taking various exceptions depend on the state of the machine when the exception conditions occur. This latency may be as short as one cycle, in which case an exception is signaled in the cycle following the appearance of the exception condition. The latencies are as follows:

- Hard reset and machine check—In most cases, a hard reset or machine check exception will have a single-cycle latency. A two-to-three-cycle delay may occur only when an instruction on a predicted branch is next to complete, and the branch prediction associated with this instruction was resolved as incorrect.

- Soft reset—The latency of a soft reset exception is affected by recoverability. The time to reach a recoverable state may depend on the time needed to complete or except an instruction at the point of completion, the time needed to drain the completed store queue, and the time waiting for a correct empty state so that a valid MSR[IP] may be saved. For lower-priority externally-generated interrupts, a delay may be incurred waiting for another interrupt generated while reaching a recoverable state, to be serviced.

Further delays are possible for other types of exceptions depending on the number and type of instructions that must be completed before that exceptions may be serviced. See Section 4.1.2, "Summary of Front-End Exception Handling," to determine possible maximum latencies for different exceptions.

4.5 Exception Definitions

Table 4-8 shows all the types of exceptions that can occur with the 602 and the MSR bit settings when the processor transitions to supervisor mode. The state of these bits prior to the exception is typically stored in SRR1.

Table 4-8. MSR Setting Due to Exception

Exception Type	MSR Bit																
	AP	SA	POW	TGPR	ILE	EE	PR	FP	ME	FE0	SE	BE	FE1	IR	DR	RI	LE
System reset	0	0	0	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	0	0	0	ILE
DSI	0	0	0	0	—	0	0	0	—	0	0	0	0	0	0	0	ILE
ISI	0	0	0	0	—	0	0	0	—	0	0	0	0	0	0	0	ILE
External	0	0	0	0	—	0	0	0	—	0	0	0	0	0	0	0	ILE
Alignment	0	0	0	0	—	0	0	0	—	0	0	0	0	0	0	0	ILE
Program	0	0	0	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	0	0	0	ILE
Decrementer	0	0	0	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	0	0	0	ILE
Trace exception	0	0	0	0	—	0	0	0	—	0	0	0	0	0	0	0	ILE
ITLB miss	0	0	0	1	—	0	0	0	—	0	0	0	0	0	0	0	ILE
DTLB miss on load	0	0	0	1	—	0	0	0	—	0	0	0	0	0	0	0	ILE
DTLB miss on store	0	0	0	1	—	0	0	0	—	0	0	0	0	0	0	0	ILE
Instruction address breakpoint	0	0	0	0	—	0	0	0	—	0	0	0	0	0	0	0	ILE
System management interrupt	0	0	0	0	—	0	0	0	—	0	0	0	0	0	0	0	ILE
Watchdog timer	0	0	0	0	—	0	0	0	—	0	0	0	0	0	0	0	ILE
Emulation trap	0	0	0	0	—	0	0	0	—	0	0	0	0	0	0	0	ILE

0   Bit is cleared
1   Bit is set
ILE   Bit is copied from the ILE bit in the MSR.
—   Bit is not altered
Reserved bits are read as if written as 0.

4.5.1 Reset Exceptions (0x0100)

The system reset exception is a nonmaskable, asynchronous exception signaled to the 602 either through the assertion of the reset signals ($\overline{\text{SRESET}}$ or $\overline{\text{HRESET}}$) or internally during the power-on reset (POR) process. The assertion of the soft reset signal, $\overline{\text{SRESET}}$, as described in Section 7.2.9.6.2, "Soft Reset (SRESET)—Input," causes the soft reset exception to be taken, and the physical base address of the handler is determined by the MSR[IP] bit. The assertion of the hard reset signal, $\overline{\text{HRESET}}$, as described in Section 7.2.9.6.1, "Hard Reset (HRESET)—Input," causes the hard reset exception to be taken, and the physical address of the handler is always 0xFFF0_0100.

The reset sequence is shown in Figure 4-4.
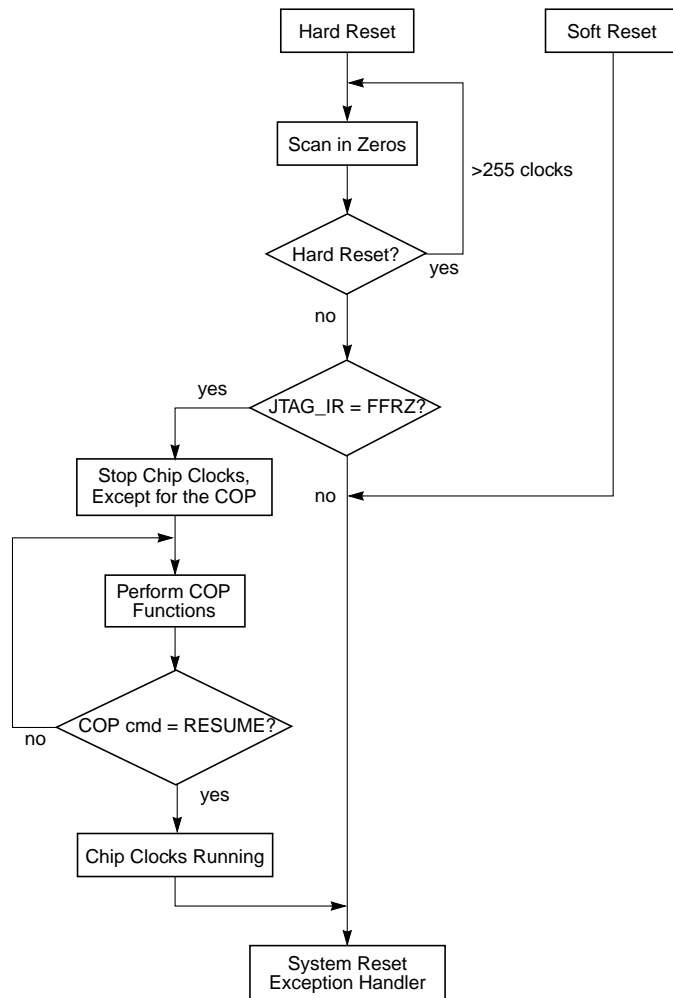
Figure 4-4. Reset Sequence

4.5.1.1 Hard Reset and Power-On Reset

As described in 4.1.2, "Summary of Front-End Exception Handling," the hard reset exception is a nonrecoverable, nonmaskable asynchronous exception (maskable interrupt). When $\overline{\text{HRESET}}$ is asserted or at power-on reset (POR), the 602 immediately branches to 0xFFF0_0100 without attempting to reach a recoverable state. A hard reset has the highest priority of any exception. It is always nonrecoverable. Table 4-9 shows the state of the machine just before it fetches the first instruction of the system reset handler after a hard reset.

The $\overline{\text{HRESET}}$ signal can be asserted for the following reasons:

- System power-on reset
- System reset from a panel switch

For information on the $\overline{\text{HRESET}}$ signal, see Section 7.2.9.6.1, "Hard Reset (HRESET)— Input."

Table 4-9. Settings Caused by Hard Reset

Register	Setting	Register	Setting
GPRs	All 0s	PVR	0005010n
FPRs	All 0s	HID0	00000000
FPSCR	00000000	HID1	00000000
CR	All 0s	DMISS and IMISS	All 0s
SRs	Unknown	DCMP and ICMP	All 0s
MSR	00000040	RPA	00000000
XER	00000000	IABR	00000000
TBU	00000000	ESARR	00000000
TBL	00000000	SER	00000000
LR	00000000	SEBR	00000000
CTR	00000000	IBR	00000000
DSISR	00000000	HASH1	00000000
DAR	00000000	HASH2	00000000
DEC	FFFFFFFF	SP	Unknown
SDR1	00000000	LT	Unknown
SRR0	00000000	TLBs	Unknown

Table 4-9. Settings Caused by Hard Reset (Continued)

Register	Setting	Register	Setting
SRR1	00000000	Cache	All cache blocks invalidated
SPRGs	00000000	BATs	Unknown
Tag directory	All 0s. (However, LRU bits are initialized so each side of the cache has a unique LRU value.)		

The following is also true after a hard reset operation:

- External checkstops are enabled.
- The on-chip test interface has given control of the I/Os to the rest of the chip for functional use.
- Since the reset exception has data and instruction translation disabled (MSR[DR] and MSR[IR] both cleared), the chip operates in direct address translation mode (referred to as the real addressing mode in the architecture specification).

4.5.1.2 Soft Reset

As described in Section 4.1.2, "Summary of Front-End Exception Handling," the soft reset exception is a type of system reset exception that is a recoverable, nonmaskable, and asynchronous. When $\overline{\text{SRESET}}$ is asserted, the processor attempts to reach a recoverable state by allowing the next instruction to either complete or cause an exception, blocking the completion of subsequent instructions, and allowing the completed store queue to drain.

Unlike a hard reset, the latches are not initialized. The $\overline{\text{SRESET}}$ signal must be asserted for at least two bus clock cycles. After the $\overline{\text{SRESET}}$ signal is deasserted, the 602 vectors to the system reset routine at 0xFFF00100. The IBR is not used to determine the vector offset for soft reset.

When a soft reset occurs, registers are set as shown in Table 4-10.

Table 4-10. Soft Reset Exception—Register Settings

Register	Setting Description
SRR0	Set to the effective address of the instruction that the processor would have attempted to complete next if no exception conditions were present.
SRR1	0–15 Cleared 16–31 Loaded from bits 16–31 of the MSR. Note that if the processor state is corrupted to the extent that execution cannot be reliably restarted, SRR1[30] is cleared.
MSR	AP 0 EE 0 FE0 0 IR 0 SA 0 PR 0 SE 0 DR 0 POW 0 FP 0 BE 0 RI 0 ILE — ME — FE1 0 LE Set to value of ILE

The vector address for a soft reset is 0x0000_0100 if MSR[IP] is cleared or 0xFFF0_0100 if MSR[IP] is set (the IBR is not used). A soft reset is recoverable provided that attaining the recoverable state does not cause a machine check exception. This interrupt case is third in priority, following hard reset and machine check. Soft resets are recoverable provided that attaining a recoverable state does not cause a machine check exception.

4.5.2 Machine Check Exception (0x0200)

The 602 conditionally initiates a machine check exception after detecting the assertion of the $\overline{\text{TEA}}$ or $\overline{\text{MCP}}$ signals on the 602 bus (assuming the machine check is enabled, MSR[ME] = 1). The assertion of one of these signals indicates that a bus error occurred and the system terminates the current transaction. One clock cycle after the signal is asserted, the data bus signals go to the high-impedance state; however, data entering the GPR or the cache is not invalidated. Note that if HID0[EMCP] is cleared, the processor ignores the assertion of the $\overline{\text{MCP}}$ signal.

Note that the 602 makes no attempt to force recoverability; however, it does guarantee that the machine check exception is always taken immediately upon request, with an address guaranteed not to be behind one that can take an exception saved in SRR0, regardless of the current machine state. Any pending stores in the completed store queue are cancelled when the exception is taken. Software can use the machine check exception in a recoverable mode for checking bus configuration. For this case, a **sync**, load, **sync** instruction sequence is used. A subsequent machine check exception at the load address indicates a bus configuration problem and the processor is in a recoverable state.

If the MSR[ME] bit is set, the exception is recognized and handled; otherwise, the 602 attempts to enter an internal checkstop. Note that the resulting machine check exception has priority over any exceptions caused by the instruction that generated the bus operation.

Machine check exceptions are only 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 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 4-11.

Table 4-11. Machine Check Exception—Register Settings

Register	Setting Description
SRR0	Set to the address of the next instruction that would have been completed in the interrupted instruction stream. Neither this instruction nor any others beyond it will have been completed. All preceding instructions will have been completed.
SRR1	0–11 Cleared 12 $\overline{\text{MCP}}$—Machine check interrupt signal caused exception 13 $\overline{\text{TEA}}$—Transfer error acknowledge signal caused exception 14–15 Cleared 16–31 Loaded from MSR[16–31].
MSR	AP 0 EE 0 FE0 0 IR 0 SA 0 PR 0 SE 0 DR 0 POW 0 FP 0 BE 0 RI 0 TGPR 0 ME — FE1 0 LE Set to value of ILE 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 $\overline{\text{TEA}}$ assertion. Otherwise, subsequent $\overline{\text{TEA}}$ assertions cause the processor to automatically enter the checkstop state.

When a machine check exception is taken, instruction execution for the handler begins at offset 0x0200. Table 4-2 shows how the vector address is determined. Note that the 602-specific IBR is not used to determine the vector of a machine check exception.

In order to return to the main program, the exception handler should do the following:

1. SRR0 and SRR1 should be given the values to be used by the **rfi** instruction.
2. Execute **rfi**.

4.5.2.2 Checkstop State (MSR[ME] = 0)

When the 602 enters the checkstop state, it asserts the checkstop output signal, $\overline{\text{CKSTP\_OUT}}$. The following events cause the 602 to enter the checkstop state:

- Machine check exception occurs with MSR[ME] cleared.
- External checkstop input signal, $\overline{\text{CKSTP\_IN}}$, is asserted.

When a processor is in the checkstop state, instruction processing is suspended and generally cannot be restarted without resetting the processor. The contents of all latches are frozen within two cycles upon entering the checkstop state so that the state of the processor can be analyzed as an aid in problem determination.

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 (0x0300)

A DSI exception occurs when no higher priority exception exists and a data memory access cannot be performed. The condition that caused the DSI exception can be determined by reading the DSISR register, a supervisor-level SPR (SPR18) that can be read by using the **mfspr** instruction. Bit settings are provided in Table 4-12. Table 4-12 also indicates which memory element is saved to the DAR. DSI exceptions can occur for any of the following reasons:

- The instruction is not supported for the type of memory addressed.
- Any attempt to access memory for which $SR[T] = 1$. Direct-store accesses are not supported on the 602.
- The access violates memory protection. Access is not permitted by the key (Ks and Kp) and PP bits, which are set in the segment register and PTE for page protection and in the BATs for block protection.

Note that the OEA specifies an additional case that may cause a DSI exception—when an effective address for a load, store, or cache operation cannot be translated by the TLBs. On the 602, this condition causes a TLB miss exception instead.

DSI exceptions can be generated by load/store instructions, and the cache control instructions (**dcbi**, **dcbz**, **dcbst**, and **dcbf**).

The 602 supports the crossing of page boundaries. However, if the second page has a translation error or protection violation associated with it, the 602 will take the DSI exception in the middle of the instruction. In this case, the data address register (DAR) always points to the first byte address of the offending page.

If an **stwcx.** instruction has an effective address for which a normal store operation would cause a DSI exception, the 602 will take the DSI exception without checking for the reservation.

If the XER indicates that the byte count for an **lswi** or **stswi** instruction is zero, a DSI exception does not occur, regardless of the effective address.

The condition that caused the exception is defined in the DSISR. These conditions also use the data address register (DAR) as shown in Table 4-12.

Table 4-12. DSI Exception—Register Settings

Register	Setting Description
SRR0	Set to the effective address of the instruction that caused the exception.
SRR1	0–15 Cleared 16–31 Loaded with bits 16–31 of the MSR
MSR	AP 0 EE 0 FE0 0 IR 0 SA 0 PR 0 SE 0 DR 0 POW 0 FP 0 BE 0 RI 0 TGPR 0 ME — FE1 0 LE Set to value of ILE ILE —
DSISR	0 Cleared. 1 Set by the data TLB miss exception handler if the translation of an attempted access is not found in the primary hash table entry group (HTEG), or in the rehashed secondary HTEG, or in the range of a DBAT register; otherwise cleared. 2–3 Cleared 4 Set if a memory access is not permitted by the page or BAT protection mechanism; otherwise cleared. 5 Set if the **lwarx** or **stwcx.** instruction is attempted to direct-store space. 6 Set for a store operation and cleared for a load operation. 7–31 Cleared
DAR	Set to the effective address of a memory element as described in the following list: • A byte in the first word accessed in the page that caused the DSI exception, for a byte, half word, or word memory access. • A byte in the first word accessed in the BAT area that caused the DSI exception for a byte, half word, or word access to a BAT area. • A byte in the block that caused the exception for **icbi**, **dcbz**, **dcbst**, **dcbf**, or **dcbi** instructions. • Any EA in the memory range addressed (for direct-store exceptions).

When a DSI exception is taken, instruction execution for the handler begins at offset 0x0300. Table 4-2 shows how the vector address can be determined.

The architecture permits certain instructions to be partially executed when they cause a DSI exception. These are as follows:

- Load multiple instructions—Some registers in the range of registers to be loaded may have been loaded.

- Store multiple instructions—Some bytes of memory in the range addressed may have been updated.

In these cases, the number of registers and amount of memory altered are instruction- and boundary-dependent. For update forms, the update register (**r**A) is not altered.

4.5.4 ISI Exception (0x0400)

The ISI exception is implemented as it is defined by the PowerPC architecture. An ISI exception occurs when no higher priority exception exists and an attempt to fetch the next instruction fails for any of the following reasons:

- If an instruction TLB miss fails to find the desired PTE, then a page fault is synthesized. The ITLB miss handler branches to the ISI exception handler to retrieve the translation from a storage device.

- The fetch access violates memory protection.

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

When an ISI exception is taken, instruction execution for the handler begins at offset 0x0400. Table 4-2 shows how the vector address can be determined.

4.5.5 External Interrupt (0x0500)

An external interrupt is signaled to the 602 by the assertion of the $\overline{INT}$ signal as described in Section 7.2.9.1, "Interrupt (INT)—Input." The interrupt may not be recognized if a higher priority exception is detected simultaneously or if the MSR[EE] bit is cleared when $\overline{INT}$ is asserted.

After the $\overline{INT}$ assertion is detected (and assuming that MSR[EE] is set), the 602 generates a recoverable halt to instruction completion. The 602 requires the next instruction in program order to complete (although it may cause an exception before doing so), block completion of any following instructions, and allow the completed store queue to drain. If any other exceptions are encountered in this process, they are taken first and the external interrupt is delayed until a recoverable halt is achieved. At this time the 602 saves the state information and takes the external interrupt as defined in the PowerPC architecture.

The register settings for the external interrupt are shown in Table 4-13.

Table 4-13. External Interrupt Exception—Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.
SRR1	0–15 Cleared 16–31 Loaded from bits 16–31 of the MSR
MSR	AP 0 EE 0 FE0 0 IR 0 SA 0 PR 0 SE 0 DR 0 POW 0 FP 0 BE 0 RI 0 TGPR 0 ME — FE1 0 LE Set to value of ILE ILE —

When an external interrupt is taken, instruction execution for the handler begins at offset 0x0500. Table 4-2 shows how the vector address is determined.

The 602 only recognizes the interrupt condition ($\overline{\text{INT}}$ asserted) if the MSR[EE] bit is set; it ignores the interrupt condition if the MSR[EE] bit is cleared. To guarantee that the external interrupt is taken, the $\overline{\text{INT}}$ signal must be held active until the 602 takes the interrupt. If the $\overline{\text{INT}}$ signal is negated before the interrupt is taken, the 602 is not guaranteed to take an external interrupt. The exception handler must send a command to the device that asserted $\overline{\text{INT}}$, acknowledging the interrupt and instructing the device to negate $\overline{\text{INT}}$.

Table 4-2 shows how the vector address is determined.

4.5.6 Alignment Exception (0x0600)

This section describes conditions that can cause alignment exceptions in the 602. Similar to DSI exceptions, alignment exceptions use the SRR0 and SRR1 to save the machine state and the DSISR to determine the source of the exception. The 602 will initiate an alignment exception when it detects any of the following conditions:

- The operand of a floating-point load or store operation is not word-aligned.
- The operand of an **lmw**, **stmw**, **lwarx**, or **stwcx.** instruction is not word-aligned.
- A little-endian access (MSR[LE] = 1) is misaligned.
- A multiple load or store operation is attempted with the MSR[LE] bit set.
- The operand of a **dcbz** instruction is in a page that is write-through or caching-inhibited.

The register settings for alignment exceptions are shown in Table 4-13.

Table 4-14. Alignment Exception—Register Settings

Register	Setting			
SRR0	Set to the effective address of the instruction that caused the exception.			
SRR1	0–15　　Cleared 16–31　Loaded from bits 16–31 of the MSR			
MSR	AP　　0 SA　　0 POW　0 TGPR　0 ILE　　—	EE　　0 PR　　0 FP　　0 ME　　—	FE0　0 SE　　0 BE　　0 FE1　0	IR　　0 DR　　0 RI　　0 LE　　Set to value of ILE

Table 4-14. Alignment Exception—Register Settings (Continued)

Register	Setting

DSISR	0–11 Cleared
	12–13 Cleared. (Can be set by several 64-bit PowerPC instructions not supported in the 602.)
	14 Cleared
	15–16 For instruction with register indirect with index addressing—set to instruction bits 29–30
	         For instruction with register indirect with immediate index addressing—cleared
	17 For instruction with register indirect with index addressing—set to instruction bit 25
	         For instruction with register indirect with immediate index addressing— set to instruction bit 5
	18–21 For instruction with register indirect with index addressing—set to bits 21–24 of the instruction
	         For instruction with register indirect with immediate index addressing—set to instruction bits 1–4
	22–26 Set to bits 6–10 (identifying the source or destination) of the instruction—undefined for **dcbz**
	27–31 Set to bits 11–15 of the instruction (**r**A)
	         Set to either bits 11–15 of the instruction or to any register number not in the range of registers
	         loaded by a valid form instruction, for **lmw**, **lswi**, and **lswx** instructions. Otherwise undefined.

DAR	Set to the EA of the data access as computed by the instruction causing the alignment exception

The architecture does not support the use of an unaligned EA by **lwarx** or **stwcx.** instructions. If one of these instructions specifies an unaligned EA, the exception handler should not emulate the instruction, but should treat the occurrence as a programming error. Table 4-2 shows how the vector address can be determined.

4.5.6.1 Integer Alignment Exceptions

The 602 is optimized for load and store operations that are aligned on natural boundaries. Operations that are not naturally aligned may suffer performance degradation, depending on the type of operation, the boundaries crossed, and the mode that the processor is in during execution. More specifically, these operations may either cause an alignment exception or they may cause the processor to break the memory access into multiple, smaller accesses with respect to the cache and the memory subsystem.

The 602 can initiate alignment exception for the accesses as shown in Table 4-15. In all of these cases, the appropriate range check is performed before the instruction begins execution. As a result, if an alignment exception is taken, it is guaranteed that no portion of the instruction has been executed.

Table 4-15. Access Types

MSR[DR]	SR[T]	Access Type
0	0	Direct translation access
x	1	Direct-store access—not supported on the 602. Any attempt to access a memory region marked as direct-store causes a DSI exception.
1	0	Page-address translation access

A real addressing mode data access occurs when MSR[DR] = 0. If a 256-Mbyte boundary is crossed by any portion of the memory being accessed by an instruction (including load/store multiples), an alignment exception is taken.

4.5.6.2 Page Address Translation Access

A page address translation access occurs when MSR[DR] is set and there is a BAT miss. Note the following points:

- The following is true for all load and store instructions except multiples:
  - An alignment exception is taken if the operand spans a 4-Kbyte boundary.
  - Byte operands never cause an alignment exception.
  - Half-word operands can cause an alignment exception if the EA ends in 0xFFF0.
  - Word operands can cause an alignment exception if the EA ends in 0xFFD–FFF.
  - Double-word operands cause an alignment exception if the EA ends in 0xFF9–FFF.
- The **dcbz** instruction causes an alignment exception if the access is to a page or block with the W (write-through) or I (caching-inhibit) bit set.

A misaligned memory access that does not cause an alignment exception will not perform as well as an aligned access of the same type. The resulting performance degradation due to misaligned accesses depends on how well each individual access behaves with respect to the memory hierarchy. At a minimum, additional cache access cycles are required that can delay other processor resources from using the cache. More dramatically, for an access to a noncacheable page, each discrete access involves individual processor bus operations that reduce the effective bandwidth of that bus.

Finally, note that when the 602 is in page address translation mode, there is no special handling for accesses that fall into BAT regions.

4.5.6.3 Floating-Point Alignment Exceptions

The 602 implements the alignment exception as it is defined in the PowerPC architecture. For information on bit settings and how exception conditions are detected, refer to *The Programming Environments Manual*.

Note that the PowerPC architecture allows individual processors to determine whether an exception is required to handle various alignment conditions. The 602 initiates an alignment exception when it detects any of the following conditions:

- The operand of a floating-point load or store operation is not word-aligned.
- The operand of **lmw**, **stmw**, **lwarx**, or **stwcx.** instruction is not word-aligned. Note that unlike other alignment exceptions, which store the address as computed by the instruction in the DAR, alignment exceptions for load or store multiple instructions store that address value + 4 into the DAR.
- The operand of a **dcbz** instruction is in a page that is write-through or caching-inhibited for a virtual mode access.
- A little-endian access is misaligned
- A multiple access is attempted while the little-endian (MSR[LE]) bit is set

4.5.7 Program Exception (0x0700)

The 602 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.

When a program exception is taken, instruction execution for the handler begins at offset 0x0700. The exception conditions are as follows:

- Floating-point enabled exception—These exceptions correspond to IEEE-defined exception conditions, such as overflows, and divide by zeroes that may occur during the execution of a floating-point arithmetic instruction. As a group, these exceptions are enabled by the FE0 and FE1 bits in the in the MSR. Individual conditions are enabled by specific bits in the FPSCR. For general information about this exception, see *The Programming Environments Manual*. For more information about how these exceptions are implemented in the 602, see Section 4.5.7.1, "IEEE Floating-Point Exception Program Exceptions."

- 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 (including PowerPC instructions not implemented in the 602). These do not include any optional instructions 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 register user privilege bit, MSR[PR], is set. In the 602, this exception is generated for **mtspr** or **mfspr** with an invalid SPR field if SPR[0] = 1 and MSR[PR] = 1. This may not be true for all PowerPC processors.

- Trap—A trap type program exception is generated when any of the conditions specified in a trap instruction is met.

4.5.7.1 IEEE Floating-Point Exception Program Exceptions

In the 602, floating-point exceptions are signaled by condition bits set in the floating-point status and control register (FPSCR). They can cause the system floating-point enabled exception handler to be invoked. The 602 handles all floating-point exceptions precisely. The 602 implements the FPSCR as it is defined by the PowerPC architecture; for more information about the FPSCR, see *The Programming Environments Manual*.

Floating-point operations that change exception sticky bits in the FPSCR may suffer a performance penalty. When an exception is disabled in the FPSCR and MSR[FE] = 0, updates to the FPSCR exception sticky bits are serialized at the completion stage. This serialization may result in a one- or two-cycle execution delay. The penalty is incurred only when the exception bit is changed and not on subsequent operations with the same exception. See Chapter 6, "Instruction Timing," for a full description of completion serialization.

When an exception is enabled in the FPSCR, the instruction traps to the emulation trap exception vector without updating the FPSCR or the target FPR. The emulation trap exception handler is required to complete the instruction. The emulation trap exception handler is invoked regardless of the FE setting in the MSR.

The two IEEE floating-point imprecise modes, defined by the PowerPC architecture as when MSR[FE0] ≠ MSR[FE1], are treated as precise exceptions (that is, if MSR[FE0] = MSR[FE1] = 1). This is regardless of the setting of MSR[NI].

For the highest and most predictable floating-point performance, all exceptions should be disabled in the FPSCR and MSR. For more information about the program exception, see *The Programming Environments Manual*.

4.5.7.2 Illegal, Reserved, and Unimplemented Instructions Program Exceptions

In accordance with the PowerPC architecture, the 602 considers all instructions defined for 64-bit implementations and unimplemented optional instructions, such as **fsqrt**, **eciwx**, and **ecowx** as illegal and takes a program exception when one of these instructions is encountered. Likewise, if a supervisor-level instruction is encountered when the processor is in user-level mode, a privileged-instruction-type program exception is taken.

The 602 implements some instructions, such as double-precision floating-point and load/store string instructions in software. These instruction take the 602-specific emulation trap exception (0x1600) rather than a program exception.

4.5.8 Floating-Point Unavailable Exception (0x0800)

The floating-point unavailable exception is implemented in the 602 as it is 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, and 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 for the handler begins at offset 0x0800. Table 4-2 shows how the vector address can be determined.

4.5.9 Decrementer Interrupt (0x0900)

The 602 implements the decrementer interrupt exception as it is defined in the PowerPC architecture. A decrementer interrupt request is made when the decrementer counts down through zero. The request is held until there are no higher priority exceptions and MSR[EE] = 1. At this point the decrementer interrupt is taken. If multiple decrementer interrupt requests are received before the first can be reported, only one exception is reported. The occurrence of a decrementer interrupt cancels the request. Register settings for this exception are described in Chapter 6, "Exceptions," in *The Programming Environments Manual*.

When a decrementer interrupt is taken, instruction execution for the handler begins at offset 0x0900. Table 4-2 shows how the vector address is determined.

4.5.10 System Call Exception (0x0C00)

The 602 implements the system call exception as it is defined by the PowerPC architecture. A system call exception request is made when a system call (**sc**) instruction is completed. If no higher priority exception exists, the system call exception is taken, with SRR0 being set to the EA of the instruction following the **sc** instruction. 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 for the handler begins at offset 0x0C00. Table 4-2 shows how the vector address can be determined.

4.5.11 Trace Exception (0x0D00)

The trace exception is taken under one of the following conditions:

- When MSR[SE] is set, a single-step instruction trace exception is taken when no higher priority exception exists and any instruction (other than **rfi** or **isync**) is successfully completed. Note that other PowerPC processors will take the trace exception on **isync** instructions (when MSR[SE] is set); the 602 does not take the trace exception on **isync** instructions. Single-step instruction trace mode is described in Section 4.5.11.1, "Single-Step Instruction Trace Mode."

- When MSR[BE] is set, the branch trace exception is taken after each branch instruction is completed.

- The 602 deviates from the architecture by not taking trace exceptions on **isync** instructions. Single-step instruction trace mode is described in Section 4.5.11.2, "Branch Trace Mode."

Successful completion implies that the instruction caused no other exceptions. A trace exception is never taken for an **sc** instruction or for a trap instruction that takes a trap exception.

MSR[SE] and MSR[BE] are cleared when the trace exception is taken. In the normal use of this function, MSR[SE] and MSR[BE] are restored when the exception handler returns to the interrupted program using an **rfi** instruction.

Register settings for the trace mode are described in Table 4-16.

Table 4-16. Trace Exception—Register Settings

Register	Setting Description
SRR0	Set to the address of the instruction following the one for which the trace exception was generated.
SRR1	0–15 Cleared 16–31 Loaded from bits 16–31 of the MSR
MSR	AP 0 EE 0 FE0 0 IR 0 SA 0 PR 0 SE 0 DR 0 POW 0 FP 0 BE 0 RI 0 TGPR 0 ME — FE1 0 LE Set to value of ILE ILE —

Note that a trace or instruction address breakpoint exception condition generates a soft stop instead of an exception if soft stop has been enabled by the JTAG/COP logic. If trace and breakpoint conditions occur simultaneously, the breakpoint conditions receive higher priority.

When a trace exception is taken, instruction execution for the handler begins as offset 0x0D00. Table 4-2 shows how the vector address is determined.

4.5.11.1 Single-Step Instruction Trace Mode

The single-step instruction trace mode is enabled by setting MSR[SE]. Encountering the single-step breakpoint causes one of the following actions:

- Trap to address vector 0x0D00
- Soft stop (wait for quiescence)

The default single-step trace action is to trap after an instruction execution and completion. The soft stop option, in which the 602 stops in a restartable state after an instruction execution and completion, can be enabled only through the COP function. The ESP, which interfaces to the COP, can restart the 602 after a soft stop. For more information, see Section 7.2.10, "JTAG/Scan Interface Signals."

4.5.11.2 Branch Trace Mode

The branch trace mode is enabled by setting MSR[BE]. Encountering the branch trace breakpoint causes one of the following actions:

- Trap to exception vector 0x0D00
- Soft stop
- Hard stop

The default branch trace action is to trap after the completion of any branch instruction whenever MSR[BE] is set. However, if soft stop is enabled through the COP interface, the 602 stops in a restartable state. If hard stop is enabled through the COP interface, the 602 stops immediately without waiting to reach a restartable state. Therefore, the 602 is not

guaranteed to be restartable after a hard stop. For more information, see Section 7.2.10, "JTAG/Scan Interface Signals."

4.5.12 Instruction TLB Miss Exception (0x1000)

When the effective address for an instruction load, store, or cache operation cannot be translated by the ITLBs, an instruction TLB miss exception is generated. Register settings for the instruction and data TLB miss exceptions are described in Table 4-17.

Table 4-17. Instruction and Data TLB Miss Exceptions—Register Settings

Register	Setting Description
SRR0	Set to the address of the next instruction to be executed in the program for which the TLB miss exception was generated.

SRR1	0–3 Loaded from condition register CR0 field
	4–12 Cleared
	13 0 = data TLB miss
	           1 = instruction TLB miss
	14 0 = replace TLB associativity set 0
	           1 = replace TLB associativity set 1
	15 0 = data TLB miss on store (or C = 0)
	           1 = data TLB miss on load
	16–31 Loaded from bits 16–31 of the MSR

MSR	AP 0 EE 0 FE0 0 IR 0 SA 0 PR 0 SE 0 DR 0 POW 0 FP 0 BE 0 RI 0 TGPR 1 ME — FE1 0 LE Set to value of ILE ILE —

If the instruction TLB miss exception handler fails to find the desired PTE, then a page fault must be synthesized. The handler must restore the machine state and turn off the GPRs before invoking the ISI exception (0x0400).

Software table search operations are discussed in Chapter 5, "Memory Management."

When an instruction TLB miss exception is taken, instruction execution for the handler begins at offset 0x1000. Table 4-2 shows how the vector address is determined.

4.5.13 Data TLB Miss on Load Exception (0x1100)

When the effective address for a data load or cache operation cannot be translated by the DTLBs, a data TLB miss on load exception is generated. Register settings for the instruction and data TLB miss exceptions are described in Table 4-17.

If a data TLB miss exception handler fails to find the desired PTE, then a page fault must be synthesized. The handler must restore the machine state and turn off MSR[TGPR] before invoking the DSI exception (0x0300).

Software table search operations are discussed in Chapter 5, "Memory Management."

When a data TLB miss on load exception is taken, instruction execution for the handler begins at offset 0x1100. Table 4-2 shows how the vector address is determined.

4.5.14 Data TLB Miss on Store Exception (0x1200)

When the effective address for a data store or cache operation cannot be translated by the DTLBs, a data TLB miss on store exception is generated. The data TLB miss on store exception is also taken when the changed bit (C = 0) for a DTLB entry needs to be updated for a store operation. Register settings for the instruction and data TLB miss exceptions are described in Table 4-17.

If a data TLB miss exception handler fails to find the desired PTE, then a page fault must be synthesized. The handler must restore the machine state and turn off the TGPRs before invoking a DSI exception (0x0300).

Software table search operations are discussed in Chapter 5, "Memory Management."

When a data TLB miss on store exception is taken, instruction execution for the handler begins at offset 0x1200. Table 4-2 shows how the vector address is determined.

4.5.15 Instruction Address Breakpoint Exception (0x1300)

The instruction address breakpoint is controlled by the IABR special-purpose register. IABR[0–29] holds an effective address to which each instruction is compared. The exception is enabled by setting IABR[30]. Note that the 602 ignores the translation enable bit (IABR[31]). The exception is taken when an instruction breakpoint address matches on the next instruction to complete. The instruction tagged with the match is not completed before the instruction address breakpoint exception is taken.

The breakpoint action can be one of the following:

- Trap to exception vector at offset 0x1300 (default). Table 4-2 shows how the vector address is determined.
- Soft stop

The bit settings when an instruction address breakpoint exception is taken are shown in Table 4-18.

Table 4-18. Instruction Address Breakpoint Exception—Register Settings

Register	Setting Description
SRR0	Set to the address of the next instruction to be executed in the program for which the TLB miss exception was generated.
SRR1	0–15 Cleared 16–31 Loaded from bits 16–31 of the MSR
MSR	AP 0 EE 0 FE0 0 IR 0 SA 0 PR 0 SE 0 DR 0 POW 0 FP 0 BE 0 RI 0 TGPR 0 ME — FE1 0 LE Set to value of ILE ILE —

The default breakpoint action is to trap before the execution of the matching instruction.

The soft stop feature can be enabled only through the COP interface. With soft stop enabled, the 602 stops in a restartable state, while with hard stop enabled, the 602 stops immediately without attempting to reach a restartable state. Upon restarting from a soft stop, the matching instructions are executed and completed unless it generates an exception. For soft stops, the next ten instructions that could have passed the IABR check can be monitored only by single-stepping the processor. When soft stops are used, the address compare must be separated by at least 10 instructions.

If soft stop is enabled, only one soft stop is generated before completion of an instruction with an IABR match, regardless of whether a soft stop is generated before that instruction for any other reason, such as trace mode on for the preceding instruction or a COP soft stop request.

Table 4-19 shows the priority of actions taken when more than one mode is enabled for the same instruction.

Table 4-19. Breakpoint Action for Multiple Modes Enabled for the Same Address

IABR[IE}	MSR[BE]	MSR[SE]	First action	Next action	Comments
1	1	0	Instruction address	Trace (branch)	Enabling both modes is useful only if both trace and address breakpoint interrupts are needed.
1	0	1	Instruction address breakpoint	Trace (single-step)	Enabling both modes is useful only if different breakpoint actions are required.
0	1	1	Trace (branch)	None	The action for branch trace and single-step trace is the same. Enabling both trace modes is redundant except for hard stop on branches.
1	1	1	Instruction address breakpoint	Trace	Enabling all modes is redundant. This entry is for clarification only.

Note that a trace or instruction address breakpoint exception condition generates a soft stop instead of an exception if soft stop has been enabled by the JTAG/COP logic. If trace and breakpoint conditions occur simultaneously, the breakpoint conditions receive higher priority.

The 602 requires that an **mtspr** instruction that updates the IABR be followed by a context-synchronizing instruction. If the **mtspr** instruction enables the instruction address breakpoint exception, the context-synchronizing instruction cannot generate a breakpoint response. The 602 also cannot block a breakpoint response on the context-synchronizing instruction if the breakpoint was disabled by the **mtspr** instruction. See Section 2.1.4, "Synchronization Requirements for SPRs," for more information on this requirement.

4.5.16 System Management Interrupt (0x1400)

The system management interrupt behaves like an external interrupt except for the signal asserted and the vector taken. A system management interrupt is signaled to the 602 by the assertion of the $\overline{\text{SMI}}$ signal. The interrupt may not be recognized if a higher priority exception is detected simultaneously or if the MSR[EE] bit is cleared when $\overline{\text{SMI}}$ is asserted. Note that $\overline{\text{SMI}}$ takes priority over $\overline{\text{INT}}$ if they are recognized simultaneously.

After the $\overline{\text{SMI}}$ is detected (and provided that MSR[EE] is set), the 602 generates a recoverable halt to instruction completion. The 602 requires the next instruction in program order to complete (although it may cause an exception to be taken before doing so), block completion of any following instructions, and allow the completed store queue to drain. If any higher priority exceptions are encountered in this process, they are taken first and the system management interrupt is delayed until a recoverable halt is achieved. At this time the 602 saves state information and takes the system management interrupt.

The register settings for the external interrupt exception are shown in Table 4-20.

Table 4-20. System Management Interrupt—Register Settings

Register	Setting Description
SRR0	Set to the effective address of the instruction that the processor would have attempted to complete next if no interrupt conditions were present.
SRR1	0–15 Cleared 16–31 Loaded from bits 16–31 of the MSR
MSR	AP 0 EE 0 FE0 0 IR 0 SA 0 PR 0 SE 0 DR 0 POW 0 FP 0 BE 0 RI 0 TGPR 0 ME — FE1 0 LE Set to value of ILE ILE —

When a system management interrupt is taken, instruction execution for the handler begins at offset 0x1400. Table 4-2 shows how the vector address is determined.

The 602 recognizes the interrupt condition ($\overline{\text{SMI}}$ asserted) only if the MSR[EE] bit is set; and ignores the interrupt condition otherwise. To guarantee that the external interrupt is taken, the $\overline{\text{SMI}}$ signal must be held active until the 602 takes the interrupt. If the $\overline{\text{SMI}}$ signal is negated before the interrupt is taken, the 602 is not guaranteed to take a system management interrupt. The exception handler must send a command to the device that asserted $\overline{\text{SMI}}$, acknowledging the interrupt and instructing the device to negate $\overline{\text{SMI}}$.

4.5.17 Watchdog Timer Interrupt (0x1500)

The watchdog timer generates a periodic exception based on the carry-out of selected bits in the time base register. The watchdog timer is controlled by the timer control register (TCR), which is specific to the 602 and not defined by the PowerPC architecture. The TCR is shown in Figure 4-5.
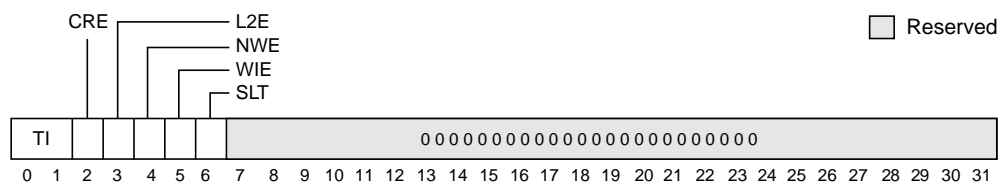
Figure 4-5. Timer Control Register (TCR)

The bits in the TCR are described in Table 4-21.

Table 4-21. Timer Control Register Bit Settings

Bit	Name	Description
0–1	TI	The timer interval bits indicate the number of clock cycles that should occur before the watchdog timer interrupt exception is taken. 00 2^{23} clock cycles (ca. 0.25 s) 01 2^{24} clock cycles (ca. 0.50 s) 10 2^{25} clock cycles (ca. 1.00 s) 11 2^{26} clock cycles (ca. 2.00 s) Approximate durations assume 33-MHz bus running in 2:1 mode. For example, if it is set as 0b00, as soon as bit 8 is set (that is, after 2e23 clock cycles) a carry-out occurs.
2	CRE	Timer core reset enable 0 Timer core reset disabled 1 Timer core reset enabled
3	L2E	Level 2 watchdog timer interrupt enable. Enables the watchdog timer level 2 interrupt after a carry-out occurs from the bit in the time base register specified by the user. 0 Timer level 2 interrupt disabled 1 Timer level 2 interrupt enabled
4	NWE	Next watchdog timer interrupt enable 0 Enable next interrupt 1 Disable next interrupt

Table 4-21. Timer Control Register Bit Settings (Continued)

Bit	Name	Description
5	WIE	Watchdog timer interrupt enable 0 Interrupt disabled 1 Interrupt enabled
6	SLT	Second-level exception taken. This bit is used by software to determine if the watchdog timer caused the soft reset. 0 Second-level soft reset not taken 1 Second-level soft reset taken
6–31	—	—

Software with supervisor-level access can select one of four time periods for the interrupts by setting TCR[TI], as shown in Table 4-21.

If the watchdog timer is enabled (TCR[WIE] is set), a level-2 interrupt condition is signalled after a carry-out occurs from the bit specified by the user. If the processor is operating properly, the exception handler must reset the watchdog timer by setting TCR[NWE]; otherwise, an internal reset of the processor core occurs after the next watchdog timer interval. This reset can be disabled by clearing TCR[CRE].

If the 602 is not operating correctly, the exception handler cannot set TCR[NWE]; therefore, when the second carry-out occurs, the watchdog timer asserts $\overline{\text{RESETO}}$ signal and generates a soft reset to the processor core. $\overline{\text{RESETO}}$ remains asserted until the third carry-out occurs, at which point it is deasserted and the process can be repeated. Note that $\overline{\text{RESETO}}$ also reflects the $\overline{\text{HRESET}}$ signal value.

The following details and assumptions should be noted with respect to using the watchdog timer interrupt:

- The initial value loaded in the time base register is assumed to be all zeros.
- The value, zero, is incremented
- For TCR[TI] = 0b00 as soon as bit 8 in the time base register is set. That is, a carry-out occurs after 2^{23} clock cycles).
- For sustained interrupt after 0.25 seconds, software must load the decrementer with all zeros after every 2^{37} carry-outs. Note that this does not preclude another value from being loaded into the time base register.

Table 4-22 shows the bit settings after a watchdog timer interrupt is taken.

Table 4-22. Watchdog Timer Interrupt—Register Settings

Register	Setting Description
SRR0	Set to the effective address of the instruction that the processor would have attempted to complete next if no interrupt conditions were present.
SRR1	0–15 Cleared 16–31 Loaded from bits 16–31 of the MSR
MSR	AP 0 EE 0 FE0 0 IR 0 SA 0 PR 0 SE 0 DR 0 POW 0 FP 0 BE 0 RI 0 TGPR 0 ME — FE1 0 LE Set to value of ILE ILE —

4.5.18 Emulation Trap Exception (0x1600)

The emulation trap exception is taken when a double-precision floating-point instruction, **fctiw**, or a load/store string instruction is encountered. An emulation trap exception is also generated if any of the operand's associated SP bits are not set for instructions requiring single-precision values as operands, or if the LT bits are not set for instructions requiring integer values as operands. Instructions that cause an emulation trap exception are listed in Section 6.8, "Instruction Latency Summary."

Table 4-23 shows the bit settings when an emulation trap exception is taken.

Table 4-23. Emulation Trap Exception—Register Settings

Register	Setting Description
SRR0	Set to the effective address of the instruction that the processor would have attempted to complete next if no interrupt conditions were present.
SRR1	0–15 Cleared 16–31 Loaded from bits 16–31 of the MSR
MSR	AP 0 EE 0 FE0 0 IR 0 SA 0 PR 0 SE 0 DR 0 POW 0 FP 0 BE 0 RI 0 TGPR 0 ME — FE1 0 LE Set to value of ILE ILE —

PowerPC 602 RISC Microprocessor User's Manual

Chapter 5
Memory Management

This chapter describes the PowerPC 602 microprocessor's implementation of the memory management unit (MMU) specifications provided by the PowerPC operating environment architecture (OEA) for PowerPC processors and also the 602-specific implementation features. For information about how the PowerPC architecture defines the memory management model, refer to Chapter 7, "Memory Management," in *The Programming Environments Manual.*

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 and I/O accesses (I/O accesses are assumed to be memory-mapped). In addition, the MMU provides access protection on a segment, block, or page basis.

The 602 implementation of the OEA-defined memory management model is similar to that of the PowerPC 603™ microprocessor with the following differences:

- The 602 implements an extra key bit in the SRR1 register that simplifies the table search software. This feature is implemented in the PowerPC 603e™ processor.

- The 602 does not support direct-store bus accesses; attempts to access a segment for which SR[T] = 1 causes a DSI or an ISI exception depending on the type of access. The same is true for the 603e.

- In addition to implementing the PowerPC exception model, the 602 can be made to operate in supervisor mode through the use of the 602-specific Enable Supervisor Access (**esa**) and Disable Supervisor Access (**dsa**) instructions. The ability to execute the **esa** instruction from a block or page is controlled by **esa** enable bits (SE) implemented in the IBATs, PTEG, and ITLB entries, which reside in the MMU.

- In addition to the translation/protection mechanisms defined by the PowerPC architecture, the 602 implements a protection-only mode. This mode is similar to the OEA-defined real addressing mode in that the effective address is used as the physical address. However, unlike real addressing mode, protection-only mode offers programmable memory protection. Additional general details are discussed separately in this overview.

- An additional NE bit is defined in IBATs and ITLB that control whether instructions can be executed from a specified page or block. The NE bit is similar to the OEA-defined SR[N] bit that controls instruction fetching privileges at the segment level.

- The 602-specific MSR[AP] bit provides an additional level of memory protection when the processor is in supervisor mode. This bit, which is valid only when the 602 is in supervisor mode, can be used to restrict supervisor-level software to accessing only memory space that is configured as user-level.

Protection-only mode is provided for special-purpose implementations that do not require the more complete paging functionality required for multipurpose personal computers, but need memory protection not offered by the OEA-defined real addressing mode. Protection-only mode is as follows:

- Each TLB can be configured to provide protection for 32, 4-Kbyte pages per TLB entry. A total of 4 Mbytes of memory can be protected in each TLB at one time. Protection consists of 1 bit per 4-Kbyte page to control instruction fetching (NE bit) in instruction pages and control write access (WE bit) in the data pages.

- Although the effective address is used as the physical address, the MMU's page translation mechanism is used to protect memory. In protection-only mode, only the 24-bit virtual segment ID (VSID) in segment register 0 (SR0) is used. This VSID also functions as a process ID in protection-only mode. Only the settings for the page from SR0 are used in this mode. Other entries can be written to, but are not used.

- The 602 provides programmable default cache control bits (WIMG) in the HID0 register to be used when the processor is running in real addressing mode or protection-only mode.

- The ESA enable base register (SEBR) and ESA enable register (SER) control the execution of the 602-specific **esa** instruction for each of the 32, 4-Kbyte pages of a 128-Kbyte block of memory at any one time.

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 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 602). In addition, two translation lookaside buffers (TLBs) are implemented on the 602 to keep recently-used page address translations on-chip. Although the PowerPC OEA describes one MMU (conceptually), the 602 hardware maintains separate TLBs and table search resources for instruction and data accesses that can be accessed independently (and simultaneously).

Therefore, the 602 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 602, 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 4.2, "Exception Processing," describes the MSR, which controls some of the critical functionality of the MMUs.

In protection-only mode, the 602 provides protection for up to 4 Mbytes of memory per TLB. In this case, effective addresses are not translated through the TLBs, but the TLB miss exceptions are still used to access the protection bits, NE and WE, stored in memory. Note also that ISI and DSI exceptions can still be caused by access protection violations.

This mode does not affect use of the BATs, which are available for protection and translation whenever translation is enabled (MSR[DR] or MSR[IR] = 1) and are maintained by the system software. As is the case with OEA-defined operations, an effective address match in the BATs takes priority over a hit in the TLBs in protection-only mode.

5.1 MMU Features

The 602 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 block address translation (BAT) mechanism for mapping large blocks of memory. Block sizes range from 128 Kbytes to 256 Mbytes and are software-programmable.

In addition, the 602 implements a protection-only mode in which each TLB protects up to 128 Kbytes per entry (4 Mbytes per TLB). Page address translation is not performed by the TLBs in protection-only mode; however, the BAT mechanism is not affected in protection-only mode and it still implements both protection and translation of the effective addresses as described earlier.

Table 5-1 summarizes all 602 MMU features including the architectural features of PowerPC MMUs (defined by the OEA) for 32-bit processors and the implementation-specific features provided by the 602.

Table 5-1. MMU Features Summary

Feature Category	Where Defined	Feature
Address ranges	OEA	2^{32} bytes of effective address
		2^{52} bytes of virtual address
		2^{32} bytes of physical address
Page size	OEA	4 Kbytes
Segment size	OEA	256 Mbytes
Block address translation	OEA	Range of 128 Kbyte–256 Mbyte sizes
		Implemented with IBAT and DBAT registers in BAT array
Memory protection	OEA	Segments selectable as no-execute through the use of the SR[N] bit
		Pages selectable as user/supervisor and read-only
		Blocks selectable as user/supervisor and read-only
	602	Additional no-execute protection bits (NE) that allow no-execute protection at the page and block level, in addition to the OEA-defined SR[N] bit that provides no-execute protection on a per-segment basis.
		SE bit that controls whether the 602's **esa** instruction can be executed from a particular 4-Kbyte page
		Protection-only mode. Provides memory protection without address translation. Unlike the OEA-defined real addressing mode, protection-only mode provides memory protection for the 602-specific no-execute bit (NE), SE bit (to enable or disable use of the **esa** instruction), and WE bit, which controls whether blocks or pages are write-enabled.
Page history	OEA	Referenced and changed bits defined and maintained
Page address translation	OEA	Translations stored as PTEs in hashed page tables in memory
		Page table size determined by mask in SDR1 register
	602	NE and SE bits defined in PTEs control no-execute and **esa** execution, respectively.
Cache attributes	602	The 602 implements programmable default cache control bits (WIMG) in HID0 used when address translation is disabled (MSR[DR] or MSR[IR] = 0) or when the processor is in protection-only mode (HID0[PO] = 1.)
TLBs	OEA	Instructions for maintaining optional TLBs (**tlbie** instruction in 602)
	602	32-entry, two-way set associative ITLB 32-entry, two-way set associative DTLB
		Alternate use for ITLB and DTLB entries in protection-only mode—Each ITLB entry holds 32 NE bits that indicate whether the corresponding 4-Kbyte page is configured as a no-execute page; each DTLB entry holds 32 WE bits that control whether the corresponding page is write-enabled.

Table 5-1. MMU Features Summary (Continued)

Feature Category	Where Defined	Feature
Segment descriptors	OEA	Stored as segment registers on-chip
Page table search support	602	Three MMU exceptions defined: ITLB miss exception, DTLB miss on load exception, and DTLB miss on store (or C = 0) exception; MMU-related bits set in SRR1 for these exceptions.
		IMISS and DMISS registers (missed effective address) HASH1 and HASH2 registers (PTEG address) ICMP and DCMP registers (for comparing PTEs) RPA register (for loading TLBs)
		tlbli rB instruction for loading ITLB entries **tlbld r**B instruction for loading DTLB entries
		Shadow registers for GPR0–GPR3 that can use r0–r3 in table search handler without corrupting r0–r3 in context that was previously executing. These registers are available as r0–r3 when MSR[TGPR] is set. MSR[TGPR] is a 602-specific bit that when set maps instruction accesses that would have been to GPR0–GPR3 to 602-defined shadow registers (TGPR0–TGPR3). The 602 automatically sets MSR[TGPR] whenever one of the three TLB miss exceptions occurs, allowing these exception handlers to have four registers that are used as scratchpad space without having to save or restore this part of the machine state that existed when the exception occurred. Note that MSR[TGPR] is restored to the value in SRR1 when the **rfi** instruction is executed.
Protection-only mode	602	The 602 provides an additional memory access mode for which there is no address translation (EA = physical address), but for which memory protection is provided for each 4-Kbyte page defined by the TLBs. This protection includes the NE bit, which provides no-execute protection, the SE bit, which controls the use of **esa** supervisor access, and the WE bit, which controls whether memory can be written. These additional bits are defined in the TLBs and the BATs and are propagated and managed through portions of the architecturally-defined page translation mechanism. To support the ESA supervisor access functionality in protection-only mode, the 602 defines an ESA enable register (SER) and an ESA enable base register (SEBR) that control whether the **esa** instruction can be executed in each specified 4-Kbyte page.
Support for **esa**/**dsa** supervisor access	602	The 602 defines resources to support a way for certain user-level programs to operate in supervisor mode without using the OEA-defined exception mechanism. When an enabled **esa** instruction is executed, the processor is given access to supervisor-level instructions, registers, and memory regions without requiring synchronization or changing the program flow. The 602 defines MSR[SA] which is used to indicate when processor has accessed supervisor mode through the use of the **esa** instruction. This bit is set automatically when the **esa** instruction is successfully executed. If this bit is not set, the **dsa** instruction cannot execute and a program exception occurs.
User-level memory access option for supervisor level	602	The 602 defines an additional bit, MSR[AP], that controls supervisor-level access to memory spaces defined as user-level access only. This bit is checked only when the processor is in supervisor mode (MSR[PR] = 0). If this bit is set, the processor can still access registers and instructions defined as supervisor-level only, but can access only those memory locations configured as user-level only. The processor can access memory locations configured as supervisor-level only if MSR[PR] = MSR[AP] = 0.

5.1.1 Overview of PowerPC 602 Processor-Specific Features

This section provides an overview of 602-specific features that involve the MMU.

5.1.1.1 Instruction-Related Protection Bits—NE and SE

The 602 provides resources that control instruction fetching and the ability to execute the **esa** instruction. This functionality is controlled by the NE and SE bits, which are described as follows:

- NE bit—No-execute bits are defined in the IBATs and ITLB entries (and consequently in the PTEs that define instruction space). The NE bit controls the ability to execute instructions from the block or page defined by the IBAT or PTE. If NE is set, instructions cannot be fetched from the corresponding block or page; attempting to do so causes an ISI instruction.

- SE bit—The SE bit controls whether the **esa** instruction can be executed from the corresponding block or page in memory. If the SE bit is set (and fetching is enabled for the same block or page), the **esa** instruction can execute, which puts the processor in supervisor mode. If the SE bit is cleared, the **esa** instruction can be fetched, but is not allowed to execute. Information that indicates whether the **esa** instruction can be executed follows the instruction through the 602 instruction pipeline and causes an illegal instruction program exception when an attempt is made to execute an **esa** instruction that is not enabled. Note that the SE bit is a don't care if the NE bit is set.

  For more information about the **esa** and **dsa** instructions, see Section 2.3.9, "Using the esa Instruction for Supervisor-Level Access."

Table 5-2 shows the access permissions for the SE and NE bits. Note that as with ITLB access permissions, SR0[T] or SR0[N] are not used to determine DTLB access privileges.

Table 5-2. Instruction Space Access Permissions

NE	SE	Meaning
0	0	The **esa** instruction cannot be executed. All other valid instructions can be executed.
0	1	Instructions can be fetched and **esa** instructions can be executed.
1	X	No access. If NE is set, SE is a don't care; no instructions can be fetched including the **esa** instruction; attempting to execute an instruction causes an ISI exception.

5.1.1.2 ESA Access and Memory Management

The 602 can be made to operate in supervisor mode either by taking an exception or by executing the 602-specific Enable Supervisor Access (**esa**) instruction. Executing the **esa** instruction allows the processor to access supervisor-level instructions, registers, and memory without encountering the latencies associated with the kind of exception handling required for processors used in multipurpose personal computers. Such latencies include synchronization to ensure precise operation, and the pipeline and memory access latencies associated with having to refetch from a new instruction path.

Note that after the **esa** instruction has been successfully executed, the program can fetch instructions from any page defined as instruction space for which fetching is enabled regardless of the setting of the corresponding SE bit. The SE bit controls only the execution of the **esa** instruction itself.

When the **esa** instruction is executed, MSR[SA, EE, PR, AP] bits are saved to the ESASRR and those bits are automatically set as follows (SA = 1, EE = 0, PR = 0, AP = 0). Clearing MSR[EE] disables external interrupts, clearing MSR[PR] puts the processor in supervisor mode, and clearing MSR[AP] gives the processor supervisor-level access to memory locations. MSR[SA] is a bit that indicates that the processor is operating in this **esa**-initiated supervisor mode. This bit is cleared when the Disable Supervisor Access instruction (**dsa**) is executed. If MSR[SA] is not set, attempting to execute **dsa** causes a program exception.

The processor remains in supervisor mode until the **dsa** instruction is executed. Note that the **dsa** instruction can be executed from any memory location for which instruction fetching is enabled—that is, the **dsa** instruction can be executed regardless of the setting of SE for the page in which it resides. When the **dsa** instruction is executed, MSR[SA, EE, PR, AP] are restored from the ESASRR and **esa** supervisor access ends.

For more information about the **esa** and **dsa** instructions, see Section 2.3.7, "PowerPC 602 Implementation-Specific Instructions."

Implementation of the ESA supervisor access feature affects the 602's MMU implementation in the following ways:

- The execution of the **esa** instruction is enabled on a page or block basis, so the MMU translation mechanism must be used to configure memory to allow or disallow this functionality. An additional SE bit is provided in the ITLB entries and IBATs to enable the **esa** instruction. Configuration of memory space defined by TLBs is handled by using the 602-defined TLB Load Instruction (**tlbli**) and TLB Load Data (**tlbld**) instructions. BATs are configured by using the **mtspr** instruction.

- This facility can be used regardless of whether the processor uses one of the architecturally defined translation mechanisms or the 602-specific protection-only mode. When the **esa** instruction is enabled in protection-only mode (for which the translation mechanism is not used to form the physical address, EA = PA), resources such as the RPA and TLBs that are otherwise defined for translation are redefined to support memory protection. In protection-only mode, the ESA enable register (SER) and ESA enable base register (SEBR) control **esa** execute privileges for each of the 32 pages of a 128-Kbyte block of memory.

- Note that instruction address translation must be enabled (MSR[IR] = 1) for **esa** to be executed; therefore, **esa** cannot be executed when the processor is in real addressing mode.

5.1.1.3 Protection-Only Mode Overview

The 602 provides an additional memory access mode for which there is no address translation (effective address = physical address), but for which some memory protection is provided. This protection includes the NE bit, which provides no-execute protection on a page level, the SE bit, which controls the use of **esa**/**dsa** supervisor access, and the WE bit, which controls whether memory can be written to on a page basis. In protection-only mode, additional bits are defined in the TLBs and are propagated and managed through portions of the architecturally-defined page translation mechanism.

In protection-only mode, the TLBs can be configured to provide protection for 32, 4-Kbyte pages per TLB entry. The 602 provides one 32-entry, two-way set-associative TLB for instructions and one for data. Therefore, a total of 4 Mbytes of memory can be protected in each TLB at one time—that is, 128 Kbytes per entry (32 x 4 Kbytes) and 4 Mbytes per TLB (2 x 16 x 128 Kbytes).

Protection consists of 1 bit per 4-Kbyte page to inhibit instruction fetching (NE bit) in the ITLB and to enable writes (WE bit) in the DTLB.

The TLB lookup procedure in protection-only mode is similar to that used in page address translation; however, only segment register entry 0 (SR0) is used. Other segment register entries can be written to, but are not used for address translation. The TLB lookup process is described in Section 5.6.1.1, "TLB Misses in Protection-Only Mode."

5.1.2 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a load, store, 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."

5.1.3 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.

Figure 5-2 and Figure 5-3 show the conceptual organization of the 602 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 and by cache instructions.

As shown in the figures, after an address is generated, the high-order bits of the effective address, EA0–EA19 (or a smaller set of address bits, EA0–EAn, in the cases of blocks), are translated into physical address bits PA0–PA19. The low-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 high-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 MSR[PR] and MSR[AP] 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 4.2, "Exception Processing," describes the MSR, which controls some of the critical functionality of the MMUs.

The figures show the way in which the A21–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 A20 and 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.
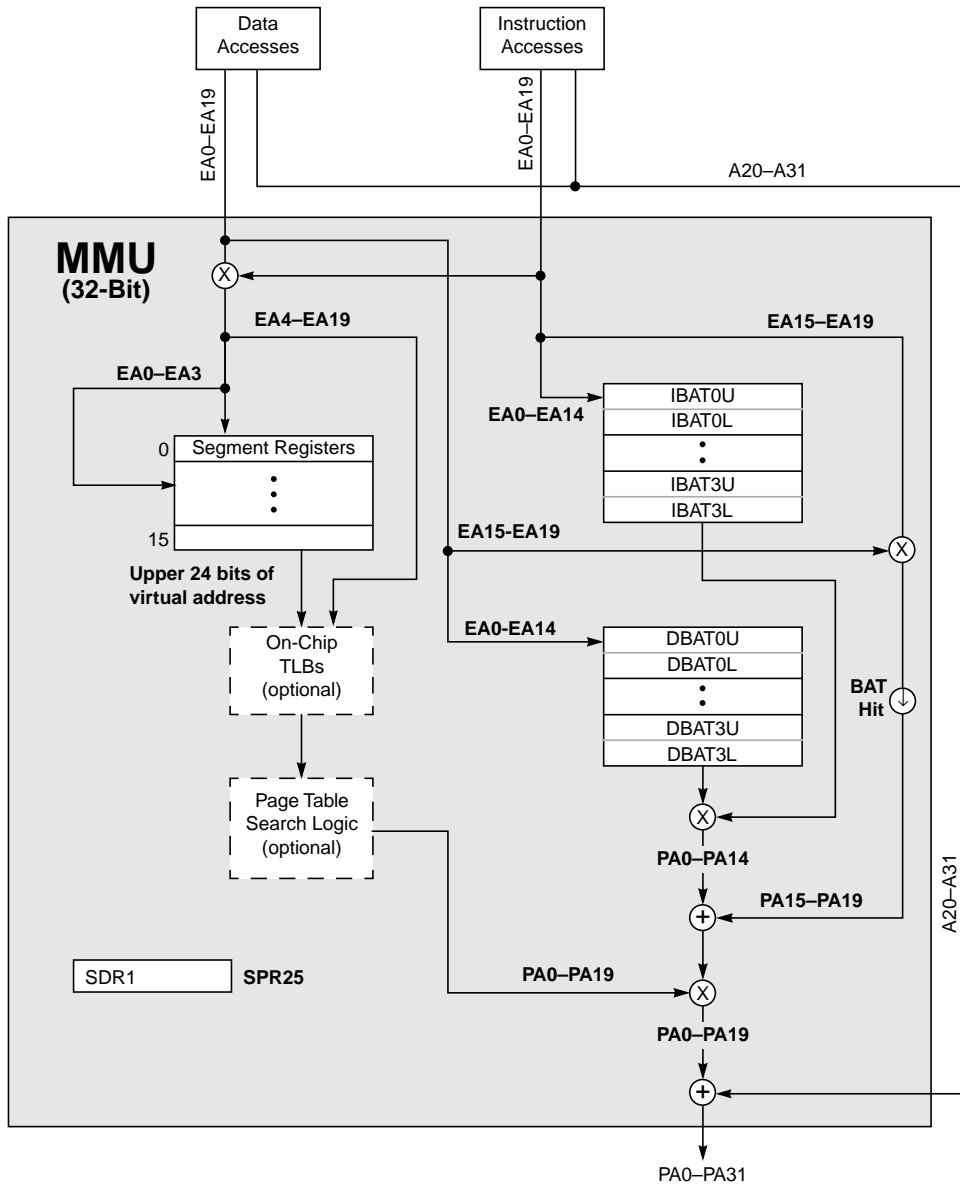
Figure 5-1. MMU Conceptual Block Diagram—32-Bit Implementations
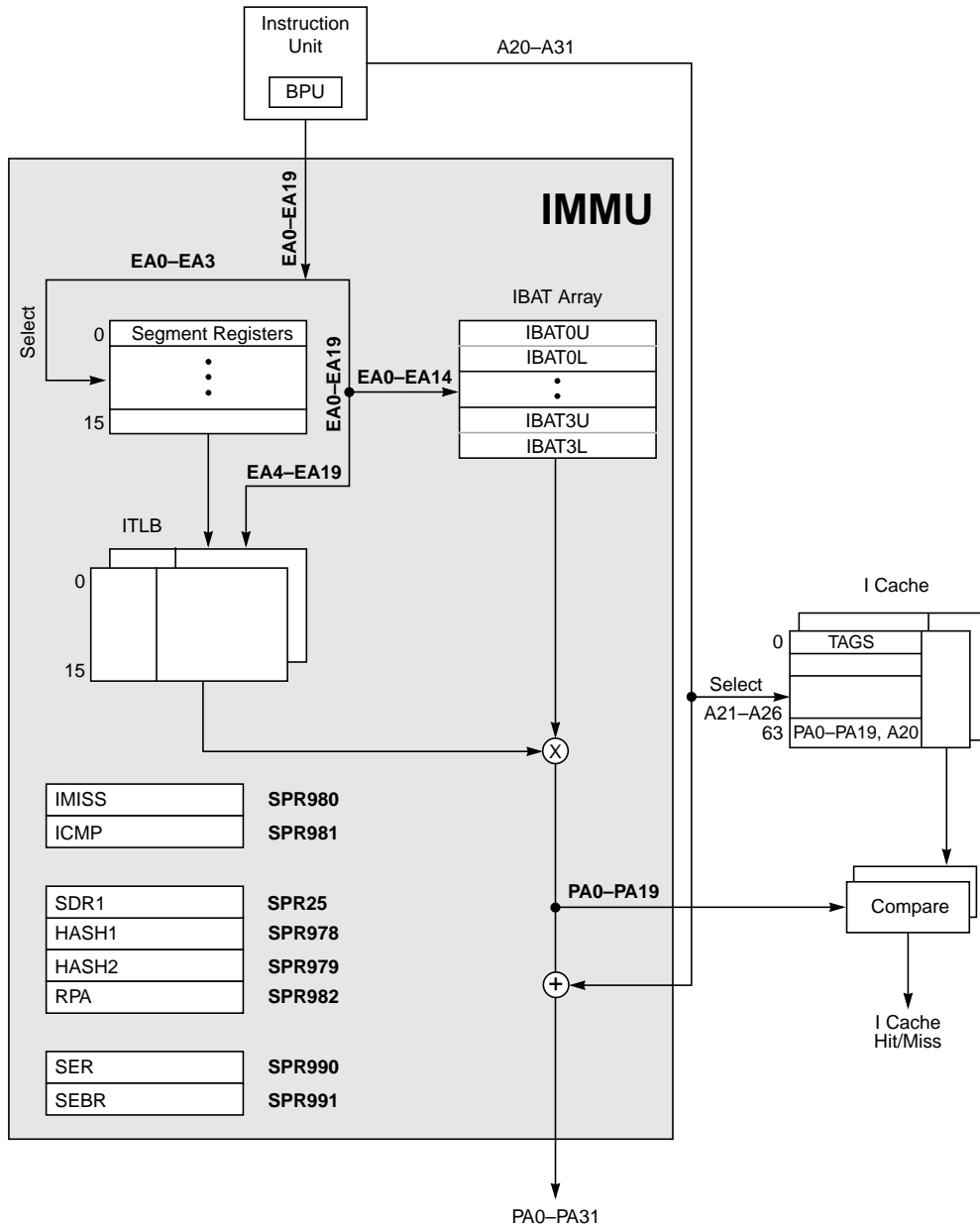
Figure 5-2. PowerPC 602 Microprocessor IMMU Block Diagram
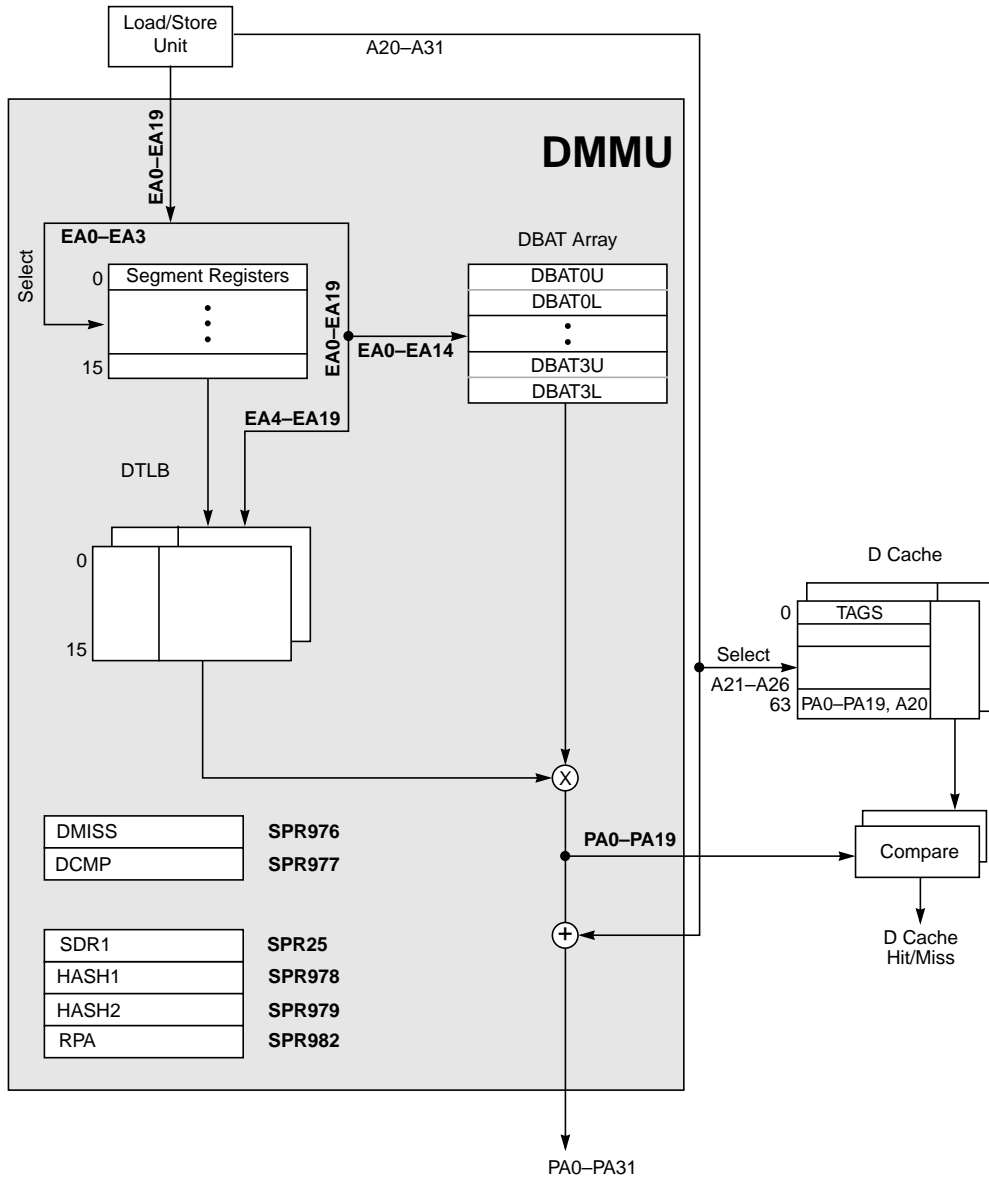
Figure 5-3. PowerPC 602 Microprocessor DMMU Block Diagram

PowerPC 602 RISC Microprocessor User's Manual

5.1.4 Address Translation Mechanisms

PowerPC processors support four types of address translation. In addition, the 602 supports an additional protection-only mode not defined by the PowerPC architecture. The memory management modes supported by the 602 are as follows:

- 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 implemented in the 602.

- Real addressing mode translation—when address translation is disabled, the physical address is identical to the effective address.

- Protection-only mode—An optional configuration of the TLBs that offers no-execute and write-enable protection for up to 4 Mbytes of memory per ITLB and DTLB, respectively. Although the effective address is used as the physical address, the MMU's translation mechanism is used to enforce protection. Protection-only mode is described in Section 5.6, "Protection-Only Mode."

Figure 5-4 shows the address translation mechanisms provided by the 602 MMUs. The segment descriptors shown in the figure control the page address translation mechanism. When an access uses page 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 (selected when the direct-store translation control bit (T bit) in the corresponding segment descriptor is set). 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 602 takes a DSI exception as described in Section 4.5.3, "DSI Exception (0x0300)."

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. When this occurs, the 602 vectors to exception handlers that search the page tables with software.

Block address translation occurs in parallel with page address translation and is similar to page address translation; however, fewer high-order effective address bits are translated into physical address bits (more low-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 (occurring in parallel) are ignored (even if the segment corresponds to the direct-store interface space).
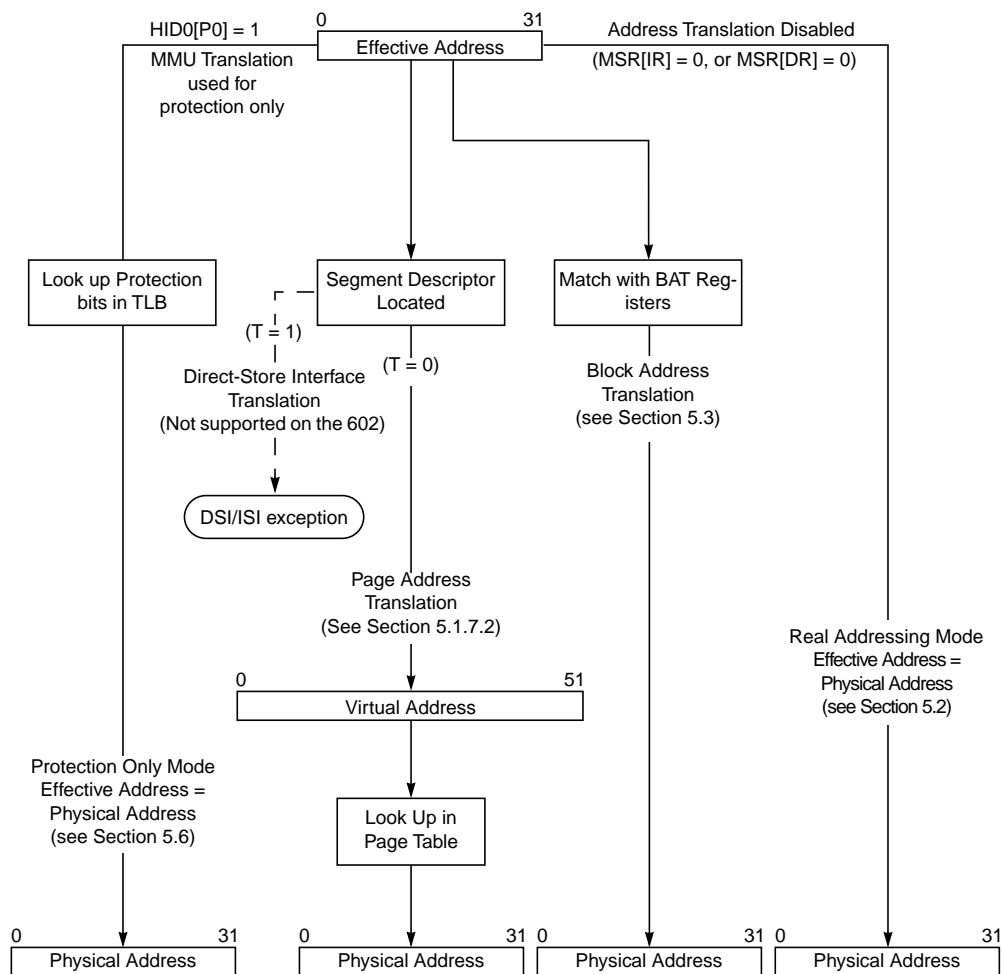
Figure 5-4. Address Translation Types

Real addressing mode translation occurs when address translation is disabled; 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.

Like real addressing mode, the 602-specific protection-only mode does not use the address translation mechanism to generate a physical address. However, unlike the real addressing mode, the protection-only mode provides memory protection features that require the use of the address translation mechanism. See Section 5.6, "Protection-Only Mode," for more information about protection-only mode.

Table 5-3 shows which 602 functions can be used in the four translation/protection modes supported by the 602.

Table 5-3. PowerPC 602 Microprocessor Feature Mapping

MMU Mode	esa Supervisor Access (SE bit)	Support for No-Execute (NE Bit)	Support for Write-Enable (WE Bit)	Use of MSR[AP]	Use of HID0[WIMG]
Page address translation	Yes	Yes	No	Yes	No
Block address translation	Yes	Yes	No	Yes	No
Real addressing mode	No	No	No	Yes	Yes
Protection-only mode	Yes	Yes	Yes	Yes	Yes

5.1.5 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 5-4 shows the eight protection options supported by the MMUs for pages along with page address translation.

Table 5-4. 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	—	√	—	—	√	√

Table 5-4. Access Protection Options for Pages (Continued)

Option	User Read		User Write	Supervisor Read		Supervisor Write
	I-Fetch	Data		I-Fetch	Data	
Both user/supervisor	√	√	√	√	√	√
Both user/supervisor-no-execute	—	√	√	—	√	√
Both read-only	√	√	—	√	√	—
Both read-only-no-execute	—	√	—	—	√	—

√ access permitted
— protection violation

The operating system programs whether instructions can be fetched from an area of memory by appropriately using the no-execute option provided in the segment descriptor or the 602-defined NE bit in the PTE. 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.

The 602 also defines an additional bit, SE, that controls whether the 602-specific **esa** instruction can be executed, thus allowing the processor to operate in supervisor mode without using the OEA-defined exception mechanism. The relationship between the **esa** instruction and the MMU is described in Section 5.6.2, "ESA Enable Protection (Instruction Space Only)."

The 602 also defines the MSR[AP] bit which controls whether the processor running in supervisor mode has user- or supervisor-level memory access. MSR[AP] is examined only when the process is in supervisor mode (MSR[PR] = 0). If this bit is set, the processor has user-level memory access.

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.

In protection-only mode (HID0[PO] = 1), pages defined as instruction space are protected by the NE bits in the ITLB and SE bits from the SER. DTLB pages are protected by the WE bit only. For instruction fetches, the NE bit controls fetching and the SE bit controls the ability to execute the **esa** instruction on a per-page basis.

Also in protection-only mode, for store instructions, the WE bit controls write access to a page; read access is permitted for all pages. Thus, all loads from the data cache or memory are permitted for pages mapped in the DTLB, but stores are disallowed unless the appropriate WE bit in the DTLB is set.

Note that the protection-only mode does not use SR[N] to determine execution/protection violations. In addition, there is an interaction with the key bits in segment register 0 (SR0). This is described in Section 5.6, "Protection-Only Mode."

For more information on memory protection, see "Memory Protection Facilities," in Chapter 7, "Memory Management," in the *The Programming Environments Manual*.

5.1.6 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. Although 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. The software table search routines used by the 602 set the R bit when a PTE is accessed; the 602 causes an exception (to vector to the software table search routines) when the C bit in the corresponding TLB entry requires updating. Note that the R and C bits are not maintained in protection-only mode and as a result do not cause exceptions for this case.

5.1.7 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.

5.1.7.1 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 translation is used (physical address equals effective address) and the access continues to the memory subsystem as described in Section 5.2, "Real Addressing Mode." Note also that the effective address also equals the physical address in protection-only mode; however, portions of the MMU which are disabled in real addressing mode, are used in protection-only mode in order to enforce memory protection.

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 page address translation.
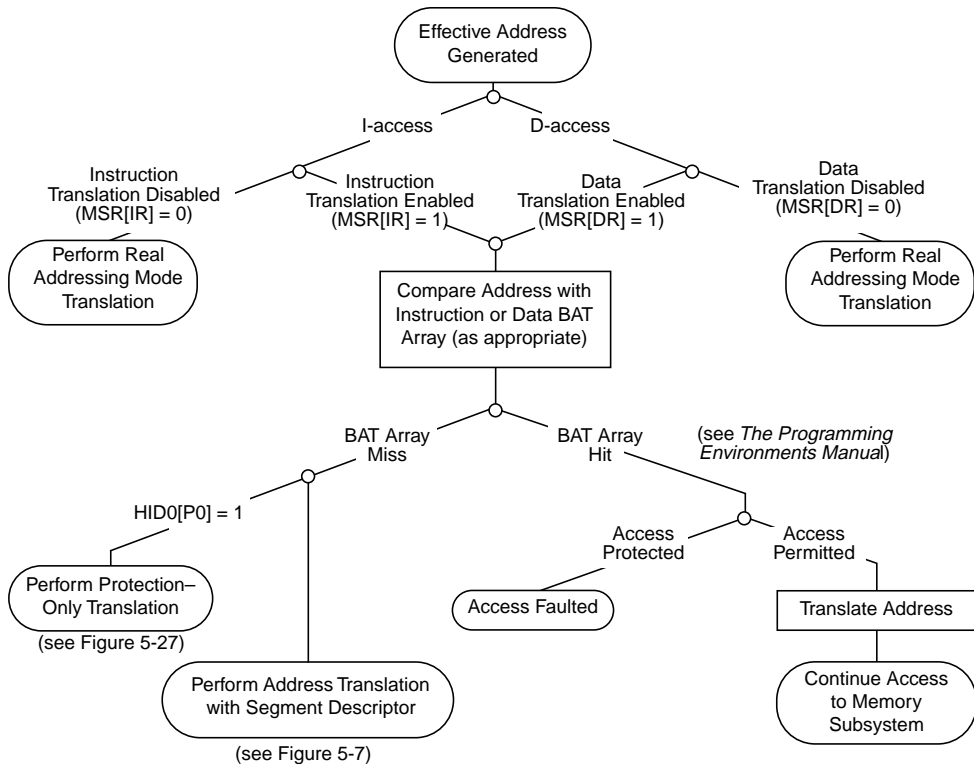
Figure 5-5. General Flow of Address Translation (Real Addressing Mode and Block)

Note that if the BAT array search results in a hit, the access is qualified with the appropriate protection bits. The 602 defines additional bits that are maintained in this process—the SE bit controls whether the **esa** instruction can be fetched and the NE bit specifies whether instructions can be executed from the referenced memory location. These bits are described in Section 2.1.1.4, "BAT Registers," If the access violates the protection mechanism, an exception (ISI or DSI exception) is generated.

5.1.7.2 Page Address Translation Selection

If address translation is enabled (real addressing mode not selected) and the effective address information does not match with a BAT array entry, MSR[PO] is checked to see if protection-only mode is selected. If the PO bit is set, protection-only mode, described in Section 5.6, "Protection-Only Mode," is used. If MSR[PO] is cleared, the segment descriptor must be located. Note that the 602 does not implement the direct-store interface and accesses to segments for which SR[T] is set cause a DSI or an ISI exception, depending on the type of access. Figure 5-6 also shows the way in which the no-execute protection is enforced at the SR level; 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 602 implements TLBs, these branches are valid, and described in more detail throughout this chapter.
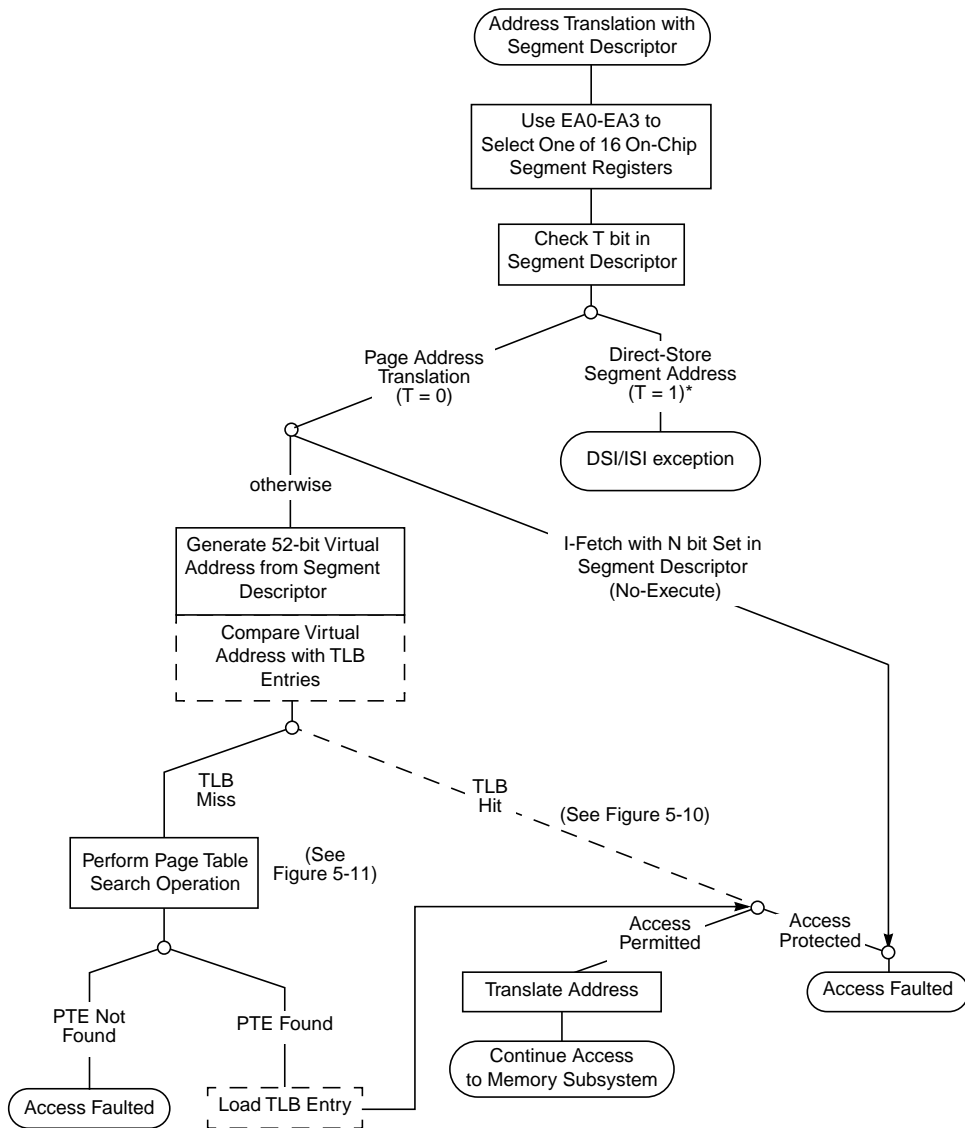
Figure 5-6. Address Translation with Segment Descriptor

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 602 has two 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 602 traps to one of three exception handlers for the system software to perform the page table search. If the PTE is successfully matched, 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), an exception (instruction access or data access) is generated.

If the PTE is not found by the table search operation, a page fault condition exists, and the TLB miss exception handlers synthesize either an ISI or DSI exception to handle the page fault.

5.1.8 MMU Exceptions Summary

To complete any memory access, the effective address must be translated to a physical address. In the 602, an MMU exception condition occurs if this translation fails for one of the following reasons:

- Page fault—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.

Additionally, because the 602 relies on software to perform table search operations, the processor also takes an exception when either of the following occurs:

- There is a miss in the corresponding (instruction or data) TLB (including protection-only mode).
- The page table requires an update to the changed (C) bit.

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.

Because a page fault condition (PTE not found in the page tables in memory) is detected by the software that performs the table search operation (and not the 602 hardware), it does not cause 602 exception in the strictest sense in that exception processing as described in Chapter 4, "Exceptions" does not occur. However, in order to maintain architectural compatibility with software written for other PowerPC devices, the software that detects this condition should synthesize an exception by setting the appropriate bits in the DSISR

or SRR1 and branching to the ISI or DSI exception handler. Refer to Section 5.5.2, "Table Search Operation with the PowerPC 602 Microprocessor," for more information and examples of this exception software. The remainder of this chapter assumes that the table search software emulates this exception and refers to this condition as an exception.

The translation exception conditions defined by the OEA for 32-bit implementations cause either the ISI or the DSI exception to be taken as shown in Table 5-5.

Table 5-5. Translation Exception Conditions

Condition	Description	Exception
Page fault (no PTE found)	No matching PTE found in page tables (and no matching BAT array entry) Note that the 602 hardware does not vector to these exceptions automatically. It is assumed that the software that performs the table search operations vectors to these exceptions and sets the appropriate bits when a page fault condition occurs	Instruction access: ISI exception SRR1[1] = 1
		Data 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*. Note that the table search software can also vector to these exception conditions	Instruction access: ISI exception SRR1[4] = 1
		Data access: DSI exception DSISR[4] =1
Page protection violation	Conditions described for page in "Memory Segment Model" in Chapter 7, "Memory Management," in *The Programming Environments Manual.*	Instruction access: ISI exception SRR1[4] = 1
		Data access: DSI exception DSISR[4] =1
No-execute protection violation	Attempt to fetch instruction when SR[N], IBAT[NE], or PTE[NE] = 1	ISI exception SRR1[3] = 1
Instruction fetch from segment where SR[T] = 1	Attempt to fetch instruction when SR[T] = 1	ISI exception SRR1[3] =1
Data access to segment where SR[T] = 1 (including floating-point access) (602-specific condition)	Attempt to perform load or store (including floating-point load or store) when SR[T] = 1	DSI exception DSISR[5] =1
Instruction fetch from guarded memory with MSR[IR] = 1	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 in the 602. These exception conditions map to the processor exception as shown in Table 5-6. For example, the 602 also defines three exception conditions to support software table searching. The only 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 (0x0600)."

Note that some exception conditions depend upon whether the memory area is set up as write-though (W = 1) or caching-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," of this book 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 5-6. Other MMU Exception Conditions for the PowerPC 602 Processor

Condition	Description	Exception
TLB miss for an instruction fetch	No matching entry found in ITLB	ITLB miss exception SRR1[13] = 1 MSR[14] = 1
TLB miss for a data access	No matching entry found in DTLB for data access	Load: DTLB miss on load exception MSR[14] = 1
		Store: DTLB miss on store exception SRR1[15] =1 MSR[14] = 1
Store operation and C = 0	Matching DLTB entry has C = 0 and access is a store	DTLB miss on store exception SRR1[15] =1 MSR[14] = 1
dcbz with W = 1 or I = 1	**dcbz** instruction to write-through or caching-inhibited segment or block	Alignment exception (not required by architecture for this condition)
lwarx or **stwcx.** 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	Floating-point memory access when SR[T] = 1	See data access to direct-store segment in Table 5-5.
Load or store would cause a direct-store error	Does not occur in 602	Does not apply
eciwx or **ecowx** attempted	**eciwx** and **ecowx** are not supported on the 602	DSI exception DSISR[11] = 1
lmw or **stmw** instruction attempted in little-endian mode	**lmw** or **stmw** 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)
Attempt to execute **esa** instruction from page or block for which SE = 0.	The **esa** instruction was fetched from a page or block for which it is not enabled. This could occur either when either the SE bit or the key equals zero.	Illegal instruction program exception

5.1.9 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 602 implements all TLB-related instructions except **tlbia**, which is treated as an illegal instruction. The 602 also uses some implementation-specific instructions to load two on-chip TLBs.

Because the MMU specification for PowerPC processors is so flexible, it is recommended that the software that uses these instructions and registers be "encapsulated" into subroutines to minimize the impact of migrating across the family of implementations.

Table 5-7 summarizes 602 instructions that specifically control the MMU. For more detailed information about the instructions, refer to Chapter 2, "PowerPC 602 Microprocessor Programming Model," in this book and Chapter 8, "Instruction Set," in *The Programming Environments Manual*.

Table 5-7. PowerPC 602 Microprocessor Instruction Summary—Control MMUs

Instruction	Description
mtsr SR,rS	Move to Segment Register SR[SR#]← **r**S
mtsrin rS,rB	Move to Segment Register Indirect SR[**r**B[0–3]]←rS
mfsr rD,SR	Move from Segment Register **r**D←SR[SR#]
mfsrin rD,rB	Move from Segment Register Indirect **r**D←SR[**r**B[0–3]]
tlbie rB*	TLB Invalidate Entry For effective address specified by **r**B, TLB[V]←0 Invalidates both TLB entries indexed by the EA and operates on both the ITLBs and DTLBs simultaneously invalidating four TLB entries. The index corresponds to EA[16–19].
tlbsync*	TLB Synchronize Implemented as a no-op on the 602
tlbli (602-specific)	TLB Load Instruction Loads data provided in the ICMP, IMISS, and RPA registers into the ITLB. Note that the format for RPA differs if the 602 is running in protection-only mode.

Table 5-7. PowerPC 602 Microprocessor Instruction Summary—Control MMUs

Instruction	Description
tlbld (602-specific)	TLB Load Data Loads data provided in the DCMP, DMISS, and RPA registers into the DTLB. Note that the format for RPA differs if the 602 is running in protection-only mode.

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

Table 5-8 summarizes the registers that the operating system uses to program the 602 MMUs. These registers are accessible to supervisor-level software only. These registers are described in Chapter 2, "Register Set," in *The Programming Environments Manual.* The 602-specific registers are described in Chapter 2, "PowerPC 602 Microprocessor Programming Model," of this book.

Table 5-8. PowerPC 602 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 segment registers are accessed by the **mtsr**, **mtsrin**, **mfsr**, and **mfsrin** instructions. In protection-only mode, the settings in SR0 are used for the entire memory space.
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. Two additional bits are specified in the 602—the NE bit provides no-execute protection, and the SE bit controls whether the **esa** instruction can be executed from the specified block.
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 SPR is accessed by the **mtspr** and **mfspr** instructions.
Instruction TLB miss address and data TLB miss address registers (IMISS and DMISS)	When a TLB miss exception occurs, the IMISS or DMISS register contains the 32-bit effective address of the instruction or data access, respectively, that caused the miss. Note that the 602 always loads a big-endian address into the DMISS register. These registers are 602-specific.
Primary and secondary hash address registers (HASH1 and HASH2)	HASH1 and HASH2 hold the primary and secondary PTEG addresses that correspond to the address causing a TLB miss. These PTEG addresses are automatically derived by the 602 by performing the primary and secondary hashing function on the contents of IMISS or DMISS, for an ITLB or DTLB miss exception, respectively. These registers are 602-specific.
Instruction and data PTE compare registers (ICMP and DCMP)	The ICMP and DCMP registers contain the word to be compared with the first word of a PTE in the table search software routine to determine if a PTE contains the address translation for the instruction or data access. The contents of ICMP and DCMP are automatically derived by the 602 when a TLB miss exception occurs. These registers are 602-specific.
Required physical address register (RPA)	The system software loads a TLB entry by loading the second word of the matching PTE entry into the RPA register and then executing the **tlbli** or **tlbld** instruction (for loading the ITLB or DTLB, respectively). The organization of the RPA contents is different when the processor is running in protection-only mode. This register is 602-specific.

Table 5-8. PowerPC 602 Microprocessor MMU Registers (Continued)

Register	Description
ESA Enable Base Register (SEBR) (protection-only mode)	SEBR[0–14] are compared with EA[0–14] to determine whether the address associated with the EA is in a 4-Kbyte page in which the **esa** instruction can be executed. If the bits match, EA[15-19] identify the bit in the SER that corresponds to the page of the EA.
ESA Enable Register (protection-only mode)	The SER register is composed of 32 SE bits which, if they are set, enable the execution of the **esa** instruction for the corresponding 4-Kbyte page.

Note that the 602 contains other features that don't specifically control the 602 MMU but that are implemented to increase performance and flexibility. These are as follows:

- Complete set of shadow segment registers for the instruction MMU. These registers are invisible to the programming model, as described in Section 5.4.4, "TLB Description."

- Temporary GPR0–GPR3. These registers are available as **r0**–**r3** when MSR[TGPR] is set. The 602 automatically sets MSR[TGPR] whenever one of the three TLB miss exceptions occurs, allowing these exception handlers to have four registers that are used as scratchpad space, without having to save or restore this part of the machine state that existed when the exception occurred. Note that MSR[TGPR] is restored to the value in SRR1 when the **rfi** instruction is executed. Refer to Section 5.5.2, "Table Search Operation with the PowerPC 602 Microprocessor," for code examples that take advantage of these registers.

In addition, the 602 also automatically saves the values of CR[CR0] of the executing context to SRR1[0–3] whenever one of the three TLB miss exceptions occurs. Thus, the exception handler can set CR[CR0] bits and branch accordingly in the exception handler routine, without having to save the existing CR[CR0] bits. However, the exception handler must restore these bits to CR[CR0] before executing the **rfi** instruction. There are also four other bits saved in SRR1 whenever a TLB miss exception occurs that give information about whether the access was an instruction or data access, and if it was a data access, whether it was for a load or a store instruction. Also these bits give some information related to the protection attributes for the access, and which set in the TLB will be replaced when the next TLB entry is loaded. Refer to Section 5.5.2.1, "Resources for Table Search Operations," for more information on these bits and their use.

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 "Synchronization Requirements for Special Registers and for Lookaside Buffers" in Chapter 2, "PowerPC Register Set," in *The Programming Environments Manual.*

Note that in the 602, the HID0[WIMG] bits provide programmable cache control attributes when real addressing is used. The real addressing mode also supports the 602-specific MSR[AP] bit, which can be used to restrict memory accesses by supervisor-level programs to only user-level memory locations. This bit is valid only for supervisor mode.

Note that the **esa** instruction and the protection provided by the SE, WE, and NE bits are not supported in real addressing mode. For implementations that require such protection without using address translation, the 602 provides a protection-only mode, described in Section 5.6, "Protection-Only Mode."

Implementation Note—When the processor is in either real addressing mode or protection-only mode, care should be taken when clearing HID0[G]. The 602 allows out-of-order loads to access the processor bus. If an out-of-order load follows an instruction that causes an exception, the load/store unit may pass the out-of-order load operation onto the system bus. Because this load cannot be cancelled, depending on the temporal position of the faulting instruction, translation may be enabled when the instruction passes through the instruction stream but may be disabled when the cache control information and address reach the bus. Setting HID0[G] prevents such a load operation from accessing the bus.

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.

The software model for block address translation in the 602 is described in Chapter 7, "Memory Management," in *The Programming Environments Manual* for 32-bit implementations.

In addition to the functionality defined by the OEA, the 602 supports two additional memory-protection features at the block level that are supported by the implementation of two 602-specific bits in the IBAT registers that are reserved in the OEA definition. These are as follows:

- The no-execute bit (NE), IBATL[21], indicates whether instructions can be fetched from the current block of memory.

- The ESA enable bit (SE), IBATL[22], indicates whether the execution of the **esa** instruction is enabled in the current block of memory.

Figure 5-7 shows the flow when an address hits in a BAT. Note that the flow differs from that defined for the PowerPC architecture except that IBAT[NE] is checked before the PP bits are checked.
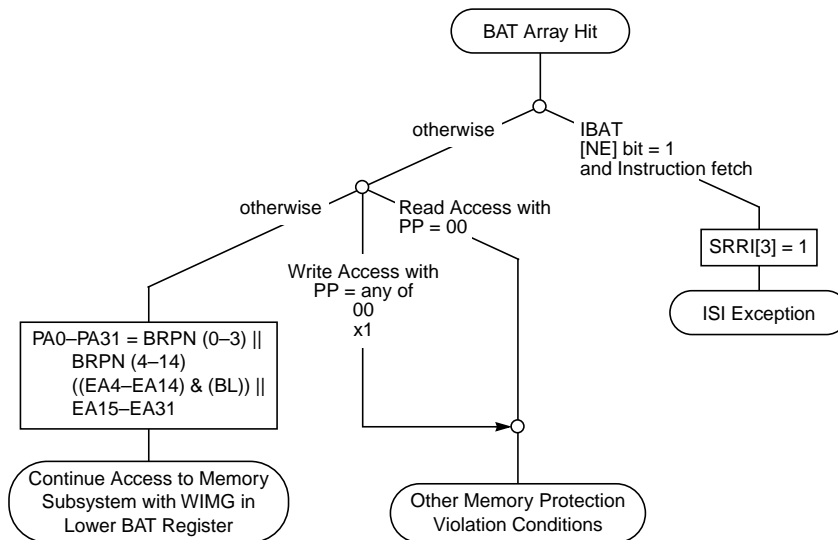
(See *The Programming Environments Manual*)

Figure 5-7. Flow for a BAT Array Hit

If the NE bit is set, the SE bit is a don't care because fetching is disabled for this block. Although the NE bit controls the ability to fetch instructions (including the **esa** instruction) from memory space specified, the SE bit does not prevent the **esa** instruction from being fetched. If the SE bit disables the **esa** instruction, it is not detected until after the processor attempts to execute the instruction, at which point an illegal instruction program exception is taken.

Implementation Note—The 602 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.

5.4 Memory Segment Model

The 602 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 HID0[PO] bit is set, the processor is in protection-only mode described in Section 5.6, "Protection-Only Mode." If HID0[PO] is not set, the translation proceeds with 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 602.

5.4.1 PTE Format in the PowerPC 602 Microprocessor

Figure 5-8 shows the format of the two words that comprise a PTE for 32-bit implementations. Note that the NE and SE bits, which are implemented in the IBATs as described in Section 5.3, "Block Address Translation," are also implemented at the page level. These bits control instruction fetching and the ability to use the **esa** instruction to enter supervisor mode.
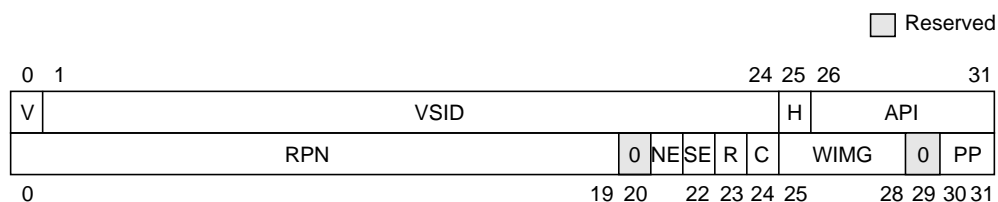
Figure 5-8. Page Table Entry Format—PowerPC 602 Processor

Table 5-9 lists the corresponding bit definitions for each word in a PTE as defined above.

Table 5-9. PTE Bit Definitions—PowerPC 602 Processor

Word	Bit	Name	Description
0	0	V	Entry valid (V = 1) or invalid (V = 0)
	1–24	VSID	Virtual segment ID
	25	H	Hash function identifier
	26–31	API	Abbreviated page index
1	0–19	RPN	Physical page number
	20	—	Reserved
	21	NE	No execute. The NE bit controls execute privileges for the page. If this bit is set, instructions cannot be fetched from this page. Note that setting SR[N] also inhibits execute privileges on a 256-Mbyte basis and overrides a setting of zero for the NE bit. The NE bit is valid only in instruction space. This bit is 602-specific.
	22	SE	Special execute. The SE bit controls whether the **esa** instruction, which puts the processor in supervisor mode, can execute from this page. The SE bit is valid only in instruction space. This bit is 602-specific.
	23	R	Referenced bit
	24	C	Changed bit
	25–28	WIMG	Memory/cache control bits
	29	—	Reserved
	30–31	PP	Page protection bits

5.4.2 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 602 hardware and the table search 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 protection-only mode. 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 602, the referenced and changed bits are updated as follows:

- For TLB hits, the C bit is updated according to Table 5-10.
- 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 5-10. 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: Table search operation required to update C bit. Causes a DTLB miss on store exception
11	No special action for read or write

Table 5-10 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 602 causes the R bit to be set for the execution of the **dcbt** or **dcbtst** instruction to that page (by causing a TLB miss exception to load the TLB entry in the case of a TLB miss). However, neither of these instructions cause the C bit to be set.

The update of the referenced and changed bits is performed by PowerPC processors as if address translation were disabled (real addressing mode translation). Additionally, these updates should be performed with single-beat read and byte write transactions on the bus.

5.4.2.1 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 R bit is then set 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 602 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 **stwcx.** instruction when no store is performed because a reservation does not exist
- Accesses that cause exceptions and are not completed

5.4.2.2 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 602). 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, it is set and a table search operation is performed to also set the C bit in the corresponding PTE in the page table. The 602 causes a data TLB miss on store exception for this case so that the software can perform the table search operation for setting the C bit.

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 all conditional branches occurring earlier in the program have been resolved (such that the store is guaranteed to be in the execution path). 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 because no reservation exists.

- 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.

5.4.2.3 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.

In implementations that do not maintain the R and C bits in hardware (such as the 602), software assistance is required. For these processors, the information in this section still applies, except that the software performing the updates is constrained to the rules described (that is, must set bits shown as guaranteed to be set and must not set bits shown as guaranteed to not be set).

Table 5-11 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 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 and by the cache management instructions that are treated as a store with respect to address

translation. In the columns for the 602, the combination of the 602 itself and the software used to search the page tables (described in Section 5.5.2, "Table Search Operation with the PowerPC 602 Microprocessor") is assumed.

Table 5-11. Model for Guaranteed R and C Bit Settings

Priority	Scenario	R Bit Set		C Bit Set	
		OEA	602	OEA	602
1	No-execute protection violation	Maybe	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[1]	No	Maybe[1]	No
6	Store conditional (**stwcx**.) with no reservation	Maybe[1]	Yes	Maybe[1]	Yes
7	In-order instruction fetch	Yes[2]	Yes	No	No
8	Load instruction	Yes	Yes	No	No
9	Store or **dcbz** instruction	Yes	Yes	Yes	Yes
10	**icbi**, **dcbt**, **dcbtst**, **dcbst**, or **dcbf** instruction	Maybe	Yes	No	No
11	**dcbi** instruction	Maybe[1]	Yes	Maybe[1]	Yes

Notes:
1. If C is set, R is guaranteed to also be set.
2. Includes the case in which the instruction was fetched out of order and R was not set (does not apply for 602).

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

5.4.3 Page Memory Protection

The 602 implements page memory protection as it is defined in Chapter 7, "Memory Management," in *The Programming Environments Manual*.

In addition to the functionality defined by the OEA, the 602 supports two additional memory-protection features at the page level that are supported by the implementation of two 602-specific bits in the PTEs and TLBs. These are as follows:

- The no-execute bit (NE), PTE[21], indicates whether instructions can execute from the page. Figure 5-10 shows how the NE bit is checked as part of the translation process.

- The ESA enable bit (SE), PTE[22], indicates whether the use of the **esa** instruction is enabled for the page.

Note that the NE bit functions like SR[N] by controlling the ability to fetch instructions from the corresponding memory space. If the NE bit is set, no instructions can be fetched, including the **esa** instruction, so in this case the SE bit is a don't care. If the NE bit is cleared, instructions can be fetched, including the **esa** instruction, regardless whether the SE bit is set. If the SE bit is cleared the **esa** supervisor access is disabled. The **esa** instruction can be fetched, but causes an exception when the processor attempts to execute it. For a detailed flow diagram, see Figure 5-10.

5.4.4 TLB Description

This section describes the hardware resources provided in the 602 to facilitate the page address translation process. Note that the hardware implementation of the MMU is not specified by the architecture, and while this description applies to the 602, it does not necessarily apply to other PowerPC processors.

Note that the TLBs are redefined when the processor is operating in protection-only mode. The low-order 32 bits of ITLB entries hold 32 NE bits, each of which indicates whether the corresponding 4-Kbyte page is configured as no-execute. The low-order 32 bits of the DTLB entries hold 32 WE bits that indicate whether the corresponding 4-Kbyte page is configured as write-enabled. For more information about protection-only mode, see Section 5.6, "Protection-Only Mode." The remainder of this section describes TLB operation when the processor is not running in protection-only mode.

5.4.4.1 TLB Organization

Because the 602 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. Figure 5-9 shows the relationships between these resources within both the IMMU and DMMU, and how the various portions of the effective address are used in the address translation process.
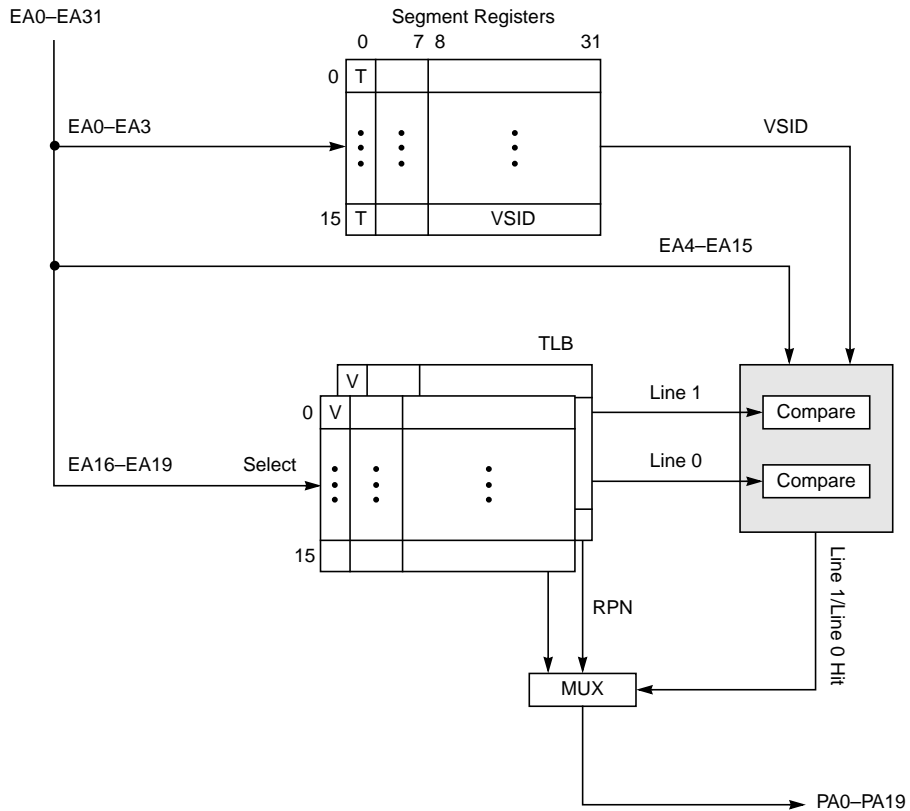
Figure 5-9. Segment Register and TLB Organization

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. ITLB miss exceptions are reported when there are no more instructions to be dispatched or retired (the pipeline is empty), and DTLB miss conditions are reported when the load or store instruction is ready to be retired. Refer to Chapter 6, "Instruction Timing," for more detailed information about the internal pipelines and the reporting of exceptions.

As TLB entries are on-chip copies of PTEs in the page tables in memory, they are similar in structure. TLB entries consist of two words—the high-order word contains the VSID and API fields of the high-order word of the PTE and the low-order word contains the RPN, the C bit, the WIMG bits, and the PP bits (as in the low-order word of the PTE). In order to uniquely identify a TLB entry as the required PTE, the PTE also contains six more bits of the page index, EA10–EA15 (in addition to the API bits of the PTE).

When an instruction or data access occurs, the effective address is routed to the appropriate MMU. EA0–EA3 select one of the 16 segment registers and the remaining effective address bits and the virtual address from the segment register is passed to the TLB. EA16–EA19 then select two entries in the TLB; the valid bit is checked and EA10–EA15, the VSID, and API fields for the access are then compared with the corresponding values in the TLB entries. If one of the entries hits, the NE bit is checked for instruction accesses, the PP bits are checked for a protection violation, and the C bit is checked. If these bits don't cause an exception, the RPN value is passed to the memory subsystem and the WIMG and SE are then used as attributes for the access.

Although address translation is disabled on any reset condition, the valid bits of the BAT array and TLB entries are not automatically cleared. Thus TLB entries must be explicitly cleared by the system software (with the **tlbie** instruction) before the valid entries are loaded and address translation is enabled.

5.4.4.2 TLB Entry Invalidation

For the PowerPC processors, such as the 602, that implement TLB structures to maintain on-chip copies of the PTEs that are resident in physical memory, the optional **tlbie** instruction provides a way to invalidate the TLB entries. Note that the execution of the **tlbie** instruction in the 602 invalidates four entries—both the ITLB entries indexed by EA16–EA19 and both the indexed entries of the DTLB.

The architecture allows **tlbie** to optionally enable a TLB invalidate signaling mechanism in hardware so that other processors also invalidate their resident copies of the matching PTE. The 602 does not signal the TLB invalidation to other processors nor does it perform any action when a TLB invalidation is performed by another processor.

The **tlbsync** instruction is treated as a no-op on the 602.

The **tlbia** instruction is not implemented on the 602 and when its opcode is encountered, an illegal instruction program exception is generated. To invalidate all entries of both TLBs, 32 **tlbie** instructions must be executed, incrementing EA16–EA19 by one each time. See Chapter 8, "Instruction Set," in *The Programming Environments Manual* for detailed information about the **tlbie** instruction.

5.4.5 Page Address Translation Summary

Figure 5-10 provides the detailed flow for the page address translation mechanism. The figure includes the checking of the N bit in the segment register and then expands on the "TLB Hit" branch of Figure 5-6, including the checking of ITLB[NE]. The detailed flow for the "TLB Miss" branch of Figure 5-6 is described in Section 5.5.1, "Page Table Search Operation—Conceptual Flow." 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 caching-inhibited (W = 1 or I = 1), the alignment exception is generated. The remaining checking of memory protection violation conditions for page address translation is described in

Chapter 7, "Memory Management," in *The Programming Environments Manual* for 32-bit implementations.
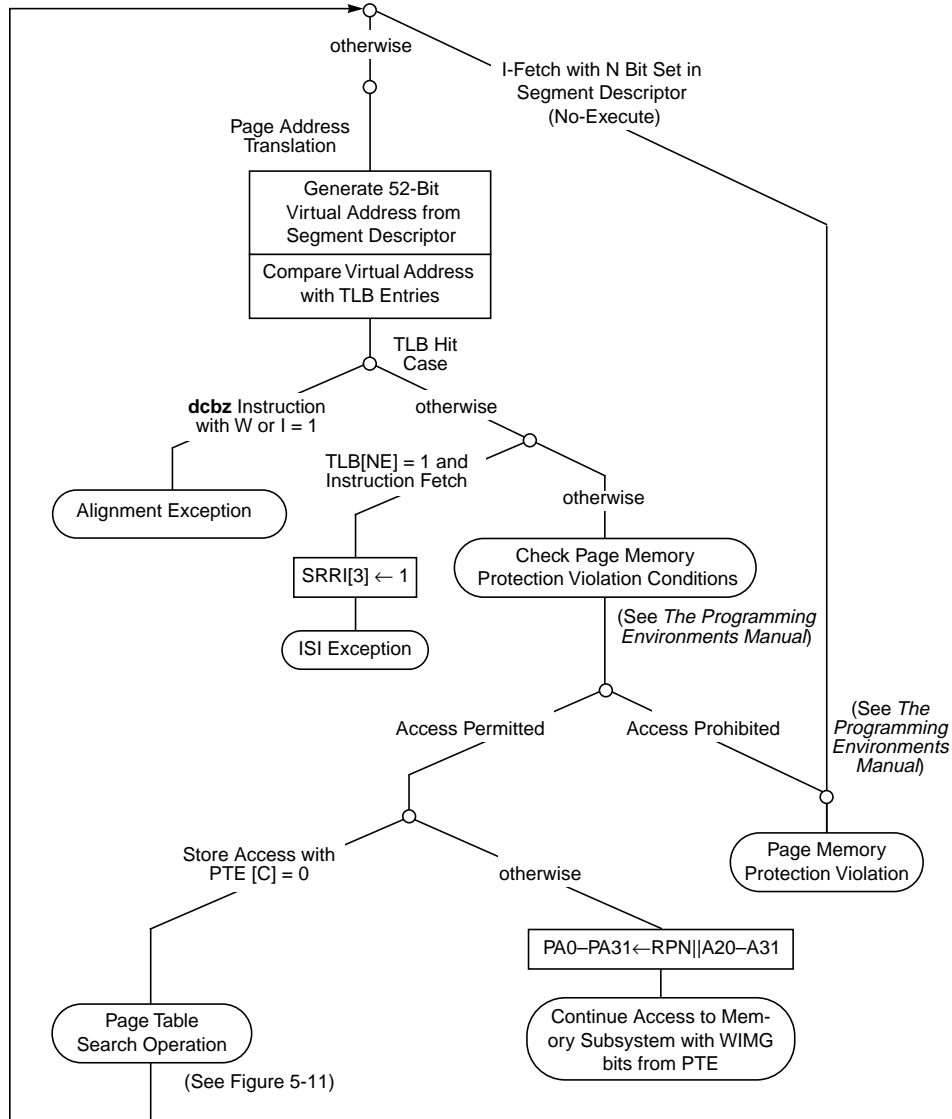
Figure 5-10. Page Address Translation Flow for PowerPC 602 Processor—TLB Hit

5.5 Page Table Search Operation

As stated earlier, the operating system must synthesize the table search algorithm for setting up the tables. In the case of the 602, the TLB miss exception handlers also use this algorithm (with the assistance of some hardware-generated values) to load TLB entries when TLB misses occur as described in Section 5.5.2, "Table Search Operation with the PowerPC 602 Microprocessor."

5.5.1 Page Table Search Operation—Conceptual Flow

The table search process for a PowerPC processor varies slightly for 64- and 32-bit implementations. The main differences are the address ranges and PTE formats specified. An outline of the page table search process performed by a 32-bit implementation (such as the 602) is as follows:

1. The 32-bit physical address of the primary PTEG is generated as described in Chapter 7, "Memory Management," in *The Programming Environments Manual* for 32-bit implementations.

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 burst in 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 eight 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 burst 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.

8. If a match is found, the PTE is written into the on-chip TLB (if implemented, as in the 602) 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 and the table search is complete.

9. If a match is not found within the eight PTEs of the secondary PTEG, the search fails, and a page fault exception condition occurs (either an ISI exception or a DSI exception). Note that the software routines that implement this algorithm for the 602 must synthesize this condition by appropriately setting the bits in SRR1 (or DSISR) and branching to the ISI or DSI handler routine.

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 5-11 and Figure 5-12 provide conceptual flow diagrams of primary and secondary page table search operations, respectively as described in the OEA for 32-bit processors. Recall that the architecture allows for implementations to perform the page table search operations automatically (in hardware) or software assist may be required, as is the case with the 602. Also, the elements in the figure that apply to TLBs are shown as optional because TLBs are not required by the architecture.

Figure 5-11 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 in the case of a memory protection violation.
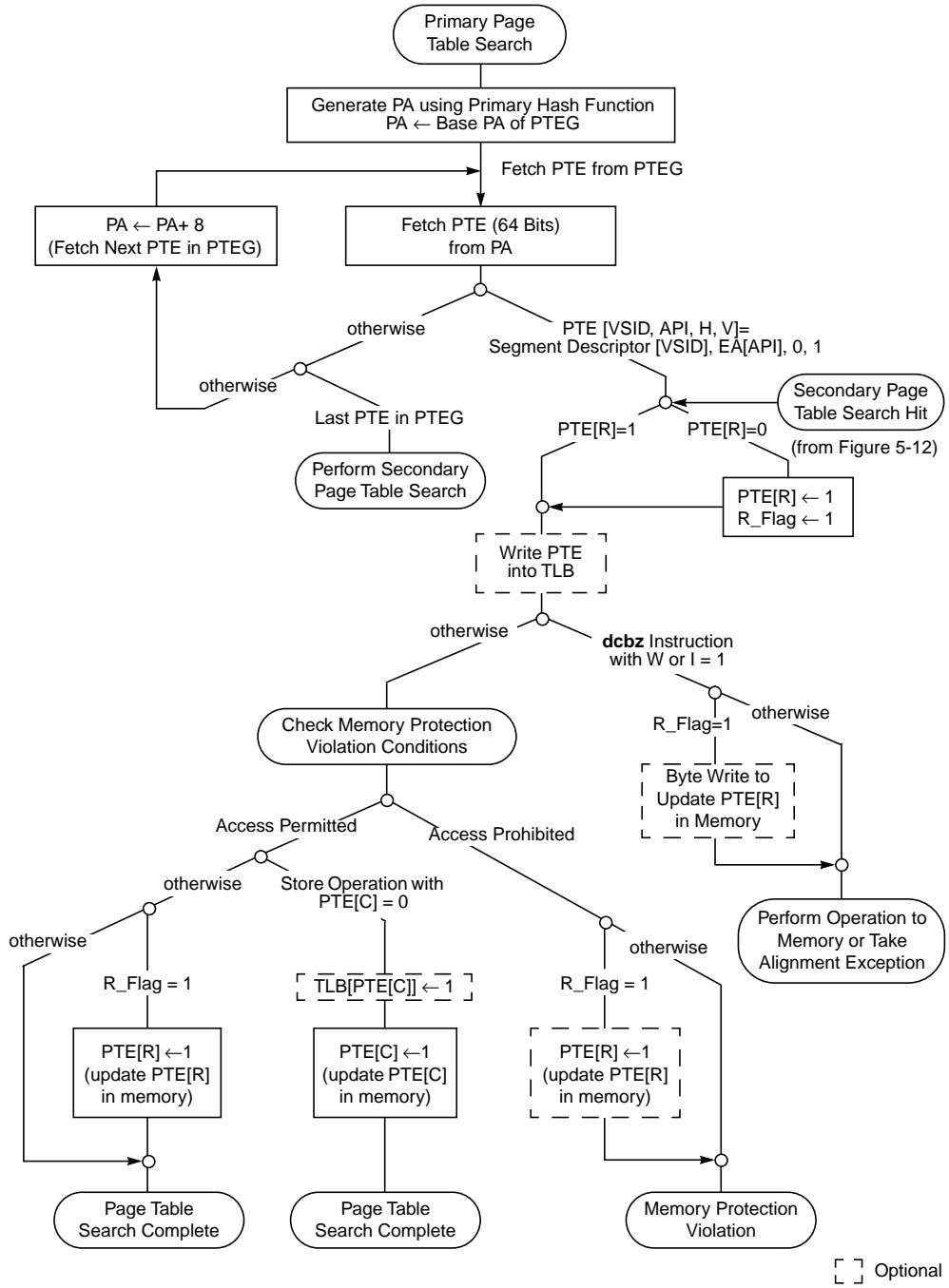
Figure 5-11. Primary Page Table Search—Conceptual Flow
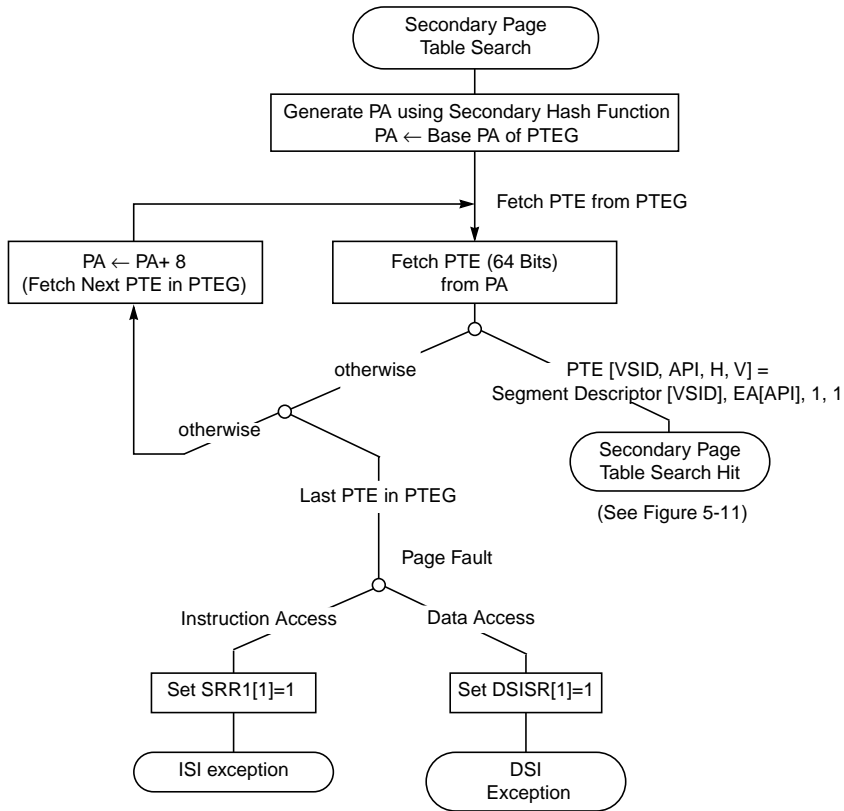
Figure 5-12. Secondary Page Table Search Flow—Conceptual Flow

5.5.2 Table Search Operation with the PowerPC 602 Microprocessor

The 602 has a set of implementation-specific registers, exceptions, and instructions that facilitate very efficient software searching of the page tables in memory. This section describes those resources that can be used in a 602 system for an efficient search of the translation tables in software. These three code sequences can be used as handlers for the three exceptions requiring access to the PTEs in the page tables in memory—instruction TLB miss, data TLB miss on load, and data TLB miss on store exceptions.

5.5.2.1 Resources for Table Search Operations

In addition to setting up the translation page tables in memory, the system software must assist the processor in loading PTEs into the on-chip TLBs. When a required TLB entry is not found in the appropriate TLB, the processor vectors to one of the three TLB miss exception handlers so that the software can perform a table search operation and load the TLB. When this occurs, the processor automatically saves information about the access and the executing context. Table 5-12 provides a summary of the implementation-specific

exceptions, registers, and instructions, that can be used by the TLB miss exception handler software in 602 systems. Refer to Chapter 4, "Exceptions," for more information about exception processing.

Table 5-12. Implementation-Specific Resources for Table Search Operations

Resource	Name	Description
Exceptions	Instruction TLB miss exception (vector offset 0x1000)	No matching entry found in ITLB
	Data TLB miss on load exception (vector offset 0x1100)	No matching entry found in DTLB for a load data access
	Data TLB miss on store exception—also caused when changed bit must be updated (vector offset 0x1200)	No matching entry found in DTLB for a store data access or matching DLTB entry has C = 0 and access is a store.
Registers	IMISS and DMISS	When a TLB miss exception occurs, the IMISS or DMISS register contains the 32-bit effective address of the instruction or data access that caused the exception.
	ICMP and DCMP	The ICMP and DCMP registers contain the word to be compared with the first word of a PTE in the table search software routine to determine if a PTE contains the address translation for the instruction or data access. The contents of ICMP and DCMP are automatically derived by the 602 when a TLB miss exception occurs.
	HASH1 and HASH2	The HASH1 and HASH2 registers contain the primary and secondary PTEG addresses that correspond to the address causing a TLB miss. These PTEG addresses are automatically derived by the 602 by performing the primary and secondary hashing function on the contents of IMISS or DMISS, for an ITLB or DTLB miss exception, respectively.
	RPA	The system software loads a TLB entry by loading the second word of the matching PTE entry into the RPA register and then executing the **tlbli** or **tlbld** instruction (for loading the ITLB or DTLB, respectively). Note that the format for the RPA register is different for protection-only mode.
Instructions	**tlbli r**B	Loads data from the IMISS, ICMP, and RPA registers into the ITLB entry selected by <ea> and SRR1[WAY]. See Section 2.3.7, "PowerPC 602 Implementation-Specific Instructions."
	tlbld rB	Loads data from the DMISS, DCMP, and RPA registers into the DTLB entry selected by <ea> and SRR1[WAY]. See Section 2.3.7, "PowerPC 602 Implementation-Specific Instructions."

In addition, the 602 contains the following other features that don't specifically control the 602 MMU but that are implemented to increase performance and flexibility in the software table search routines whenever one of the three TLB miss exceptions occurs:

- Temporary GPR0–GPR3. These registers are available as **r0**–**r3** when MSR[TGPR] is set. The 602 automatically sets MSR[TGPR] for these cases, allowing these exception handlers to have four registers that are used as scratchpad space, without having to save or restore this part of the machine state that existed when the exception occurred. Note that MSR[TGPR] is cleared when the **rfi** instruction is executed because the old MSR value (with MSR[TGPR] = 0) saved in SRR1 is restored.

- The 602 also automatically saves the values of CR[CR0] of the executing context to SRR1[0–3]. Thus, the exception handler can set CR[CR0] bits and branch accordingly in the exception handler routine, without having to save the existing CR[CR0] bits. However, the exception handler must restore these bits to CR[CR0] before executing the **rfi** instruction.

- Also saved in SRR1 are two bits identifying the type of miss (SRR1[D/I] identifies instruction or data, and SRR1[L/S] identifies a load or store). Additionally, SRR1[WAY] identifies the associativity class of the TLB entry selected for replacement by the LRU algorithm. The software can change this value, effectively over-writing the replacement algorithm. Finally, the SRR1 [KEY] bit is used by the table search software to determine if there is a protection violation associated with the access (useful on data write misses for determining if the C bit should be updated in the table). Table 5-13 summarizes the SRR1 bits updated whenever one of the three TLB miss exceptions occurs.

Table 5-13. SRR1 Bits Specific to the PowerPC 602 Microprocessor

Bit Number	Name	Function
0–3	CRF0	Condition register field 0 bits
12	KEY	Key for TLB miss (either Ks or Kp from segment register, depending on whether the access is a user or supervisor access)
13	D/I	Set if instruction TLB miss
14	WAY	Next TLB set to be replaced (set per LRU)
15	S/L	Set if data TLB miss was for a load instruction

The KEY bit saved in SRR1 is derived as shown in Figure 5-13.

```
Select KEY from segment register:
     If MSR[PR] = 0, KEY= Ks
     If MSR[PR] = 1, KEY= Kp
```

Figure 5-13. Derivation of KEY bit for SRR1

The remainder of this section describes the format of the implementation-specific SPRs that are not defined by the PowerPC architecture, but are used by the TLB miss exception handlers. These registers can be accessed by supervisor-level instructions only. Any attempt to access these SPRs with user-level instructions results in a privileged instruction program exception. As DMISS, IMISS, DCMP, ICMP, HASH1, HASH2, and RPA are used to access the translation tables for software table search operations, they should only be accessed when address translation is disabled (that is, MSR[IR] = 0 and MSR[DR] = 0). Note that MSR[IR] and MSR[DR] are cleared by the processor whenever an exception occurs.

5.5.2.1.1 Data and Instruction TLB Miss Address Registers (DMISS and IMISS)

The DMISS and IMISS registers have the same format as shown in Figure 5-14. They are loaded automatically upon a data or instruction TLB miss. The DMISS and IMISS registers contain the effective page address of the access which caused the TLB miss exception. The contents are used by the processor when calculating the values of HASH1 and HASH2, and by the **tlbld** and **tlbli** instructions when loading a new TLB entry. Note that the 602 always loads a big-endian address into the DMISS register. These registers are read-only to the software.

Effective Page Address
0   31
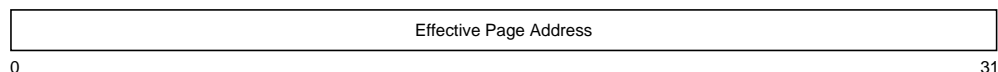
Figure 5-14. DMISS and IMISS Registers

5.5.2.1.2 Data and Instruction PTE Compare Registers (DCMP and ICMP)

The DCMP and ICMP registers are shown in Figure 5-15. These registers contain the first word in the required PTE. The contents are constructed automatically from the contents of the segment registers and the effective address (DMISS or IMISS) when a TLB miss exception occurs. Each PTE read from the tables in memory during the table search process should be compared with this value to determine whether or not the PTE is a match. Upon execution of a **tlbld** or **tlbli** instruction, the contents of the DCMP or ICMP register is loaded into the first word of the selected TLB entry.

V	VSID	H	API
0 1	24	25 26	31
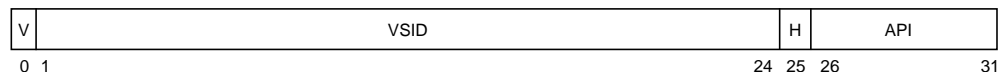
Figure 5-15. DCMP and ICMP Registers

Table 5-14 describes the bit settings for the DCMP and ICMP registers.

Table 5-14. DCMP and ICMP Bit Settings

Bits	Name	Description
0	V	Valid bit. Set by the processor on a TLB miss exception.
1–24	VSID	Virtual segment ID. Copied from VSID field of corresponding segment register.
25	H	Hash function identifier. Cleared by the processor on a TLB miss exception
26–31	API	Abbreviated page index. Copied from API of effective address.

5.5.2.1.3 Primary and Secondary Hash Address Registers (HASH1 and HASH2)

The HASH1 and HASH2 registers contain the physical addresses of the primary and secondary PTEGs for the access that caused the TLB miss exception. Only bits 7–25 differ between them. For convenience, the processor automatically constructs the full physical address by routing bits 0–6 of SDR1 into HASH1 and HASH2 and clearing the low-order six bits. These registers are read-only and are constructed from the contents of the DMISS or IMISS register. The format for the HASH1 and HASH2 registers is shown in Figure 5-16.

☐ Reserved

HTABORG	Hashed Page Address	0 0 0 0 0 0
0 6	7 25	26 31
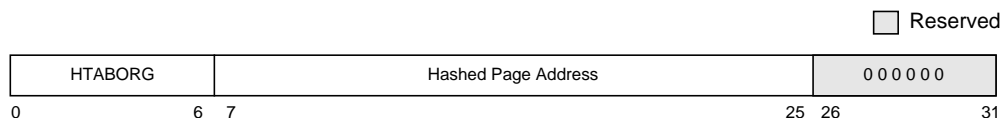
Figure 5-16. HASH1 and HASH2 Registers

Table 5-15 describes the bit settings of the HASH1 and HASH2 registers.

Table 5-15. HASH1 and HASH2 Bit Settings

Bits	Name	Description
0–6	HTABORG[0–6]	Copy of the high-order 7 bits of the HTABORG field from SDR1
7–25	Hashed page address	Address bits 7–25 of the PTEG to be searched.
26–31	—	Reserved

5.5.2.1.4 Required Physical Address (RPA) Register

The RPA is shown in Figure 5-17. During a page table search operation, the software must load the RPA with the second word of the correct PTE. When a **tlbli** or **tlbld** instruction is executed, data from the IMISS and ICMP (or DMISS and DCMP) and the RPA registers is loaded into the selected TLB entry. The TLB entry is selected by the effective address of the access (loaded by the table search software from the DMISS or IMISS register) and the SRR1[WAY] bit.

Note that the organization and operation of the RPA is different when the 602 is operating in protection-only mode, corresponding with the content of the TLB entries and PTEs.

When the **tlbld** or **tlbli** instruction is executed, bits from the specified EA (written from the IMISS or DMISS registers) and the contents of the RPA register are merged with the contents of either the DCMP or ICMP register and are loaded into the selected TLB entry.

☐ Reserved

RPN	0	NE	SE	R	C	WIMG	0	PP
0	19	20	21 22	23	24	25	28 29	30 31
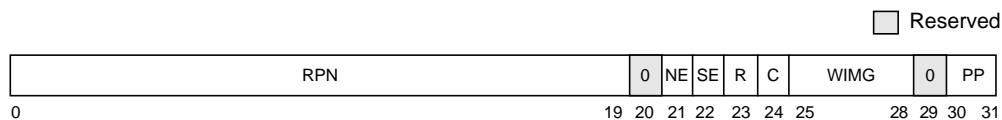
Figure 5-17. Required Physical Address (RPA) Register—Default Configuration

Table 5-16 describes the bit settings of the RPA register when HID0[PO] = 0.

Table 5-16. RPA Bit Settings—Default Configuration

Bits	Name	Description
0–19	RPN	Physical page number from PTE
20	—	Reserved
21	NE	No execute. The NE bit controls execution privileges for that page. If NE = 1, instructions cannot be fetched from that 4-Kbyte page. This bit is valid only for ITLB entries.
22	SE	Controls ability to execute **esa** instruction from this page. Valid only for ITLB entries.
23	R	Referenced bit from PTE
24	C	Changed bit from PTE
25–28	WIMG	Memory/cache access attribute bits
29	—	Reserved
30–31	PP	Page protection bits from PTE

5.5.2.2 Software Table Search Operation

When a TLB miss occurs, the instruction or data MMU loads the IMISS or DMISS register, respectively, with the effective address of the access. The processor completes all instructions dispatched prior to the exception, status information is saved in SRR1, and one of the three TLB miss exceptions is taken. In addition, the processor loads the ICMP or DCMP register with the value to be compared with the first word of PTEs in the tables in memory.

The software should then access the first PTE at the address pointed to by HASH1. The first word of the PTE should be loaded and compared to the contents of DCMP or ICMP. If there is a match, the required PTE has been found and the second word of the PTE is loaded from memory into the RPA register. Then the **tlbli** or **tlbld** instruction is executed, which loads data from the IMISS and ICMP (or DMISS and DCMP) and RPA registers into the selected TLB entry. The TLB entry is selected by the effective address of the access and the SRR1[WAY] bit.

If the compare did not result in a match, however, the PTEG address is incremented to point to the next PTE in the table and the above sequence is repeated. If none of the eight PTEs in the primary PTEG matches, the sequence is then repeated using the secondary PTEG (at the address contained in HASH2).

If the PTE is also not found in the eight entries of the secondary page table, a page fault condition exists, and a page fault exception must be synthesized. Thus the appropriate bits must be set in SRR1 (or DSISR) and the TLB miss handler must branch to either the ISI or DSI exception handler, which handles the page fault condition.

This section provides a flow diagram outlining some example software that can be used to handle the three TLB miss exceptions, as well as an assembly language example that implements that flow.

5.5.2.2.1 Flow for Example Exception Handlers

Figure 5-18 shows the flow for the example TLB miss exception handlers. The flow shown is common for the three exception handlers, except that the IMISS and ICMP registers are used for the instruction TLB miss exception while the DMISS and DCMP registers are used for the two data TLB miss exceptions. Also, for the cases of store instructions that cause either a TLB miss or require a table search operation to update the C bit, the flow shows that the C bit is set in both the TLB entry and the PTE in memory. Finally, in the case of a page fault (no PTE found in the table search operation), the setup for the ISI or DSI exception is slightly different for these two cases.

Figure 5-19 shows the flow for checking and setting the R and C bits and Figure 5-20 shows the flow for synthesizing a page fault exception when no PTE is found. Figure 5-21 shows the flow for managing the cases of a TLB miss on an instruction access to guarded memory, and a TLB miss when $C = 0$ and a protection violation exists. The set up for these protection violation exceptions is very similar to that of page fault conditions (as shown in Figure 5-20) except that different bits in SRR1 (and DSISR) are set.
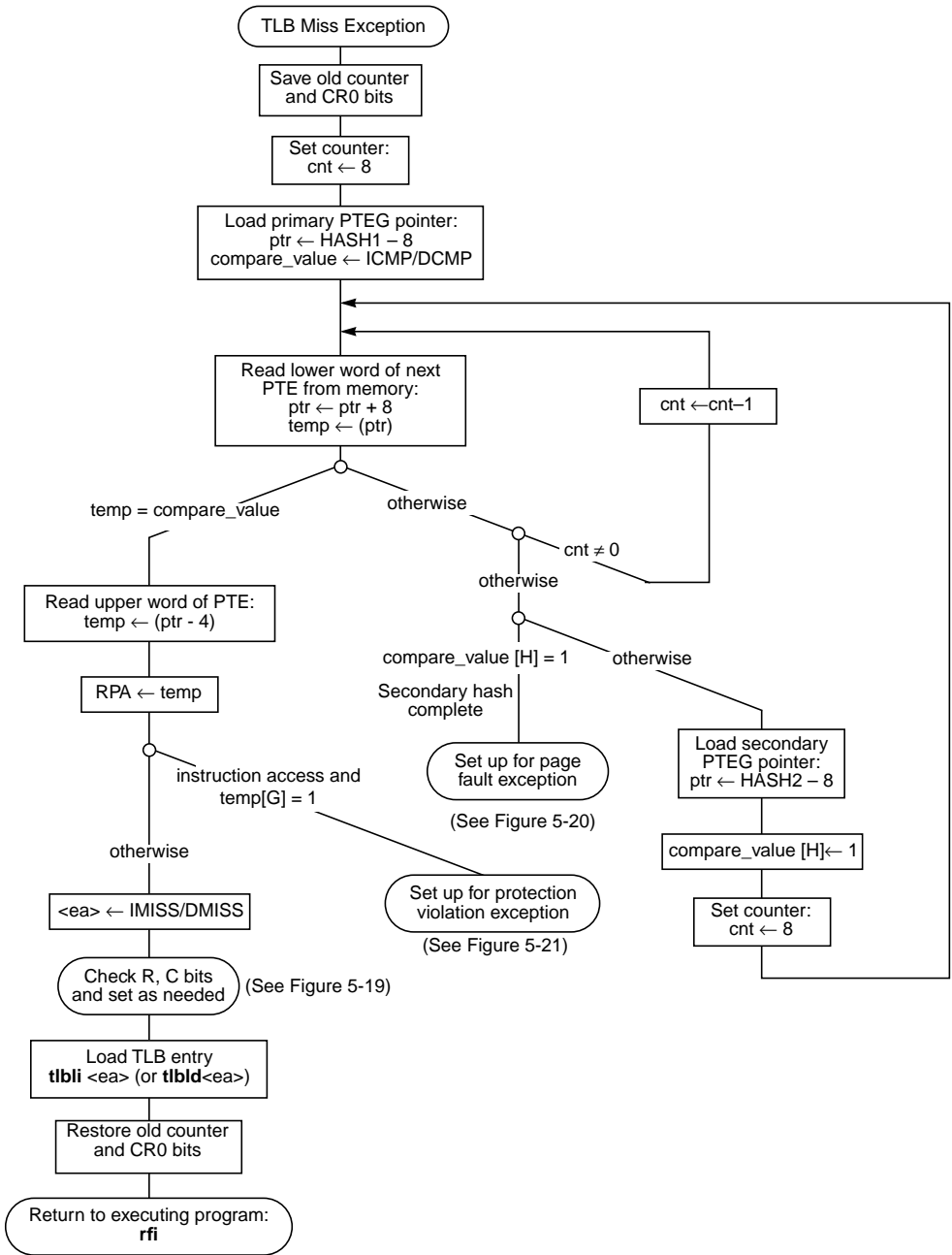
Figure 5-18. Flow for Example Software Table Search Operation
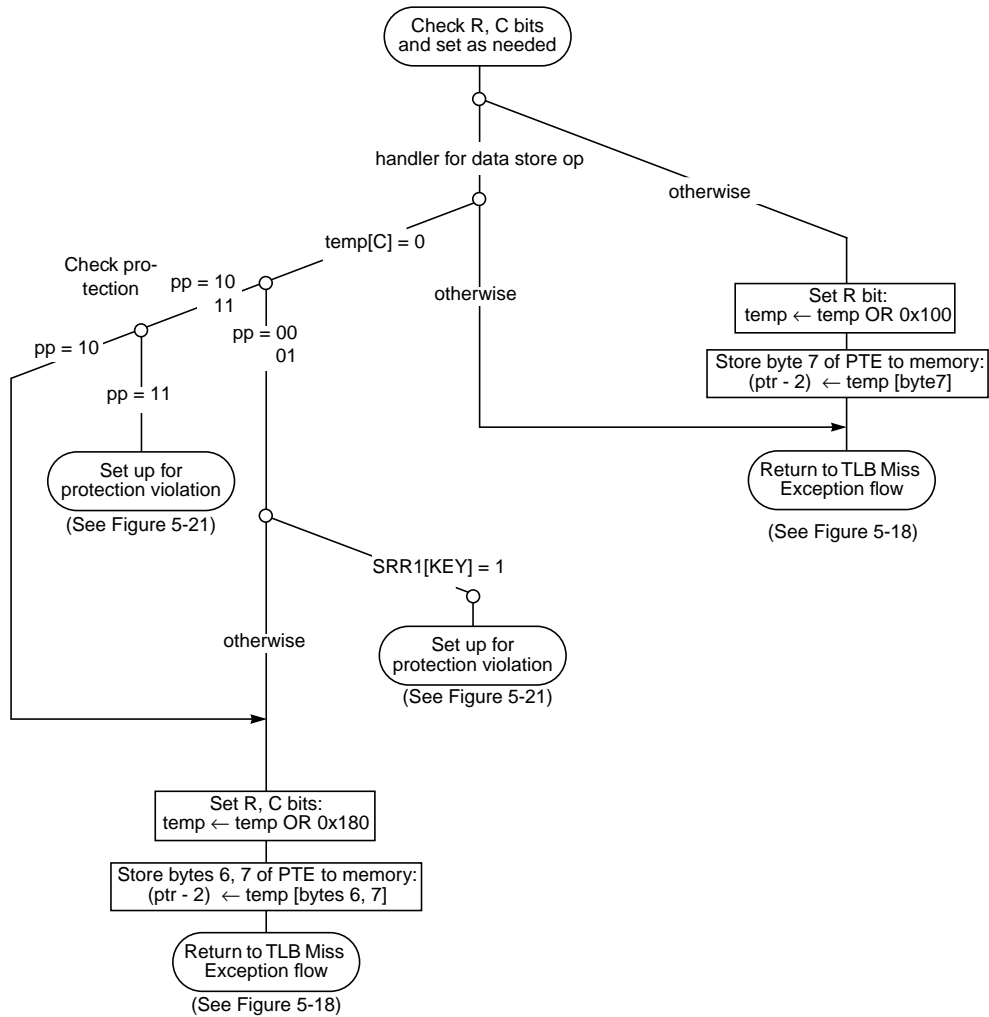
Figure 5-19. Check and Set R, C Bit Flow
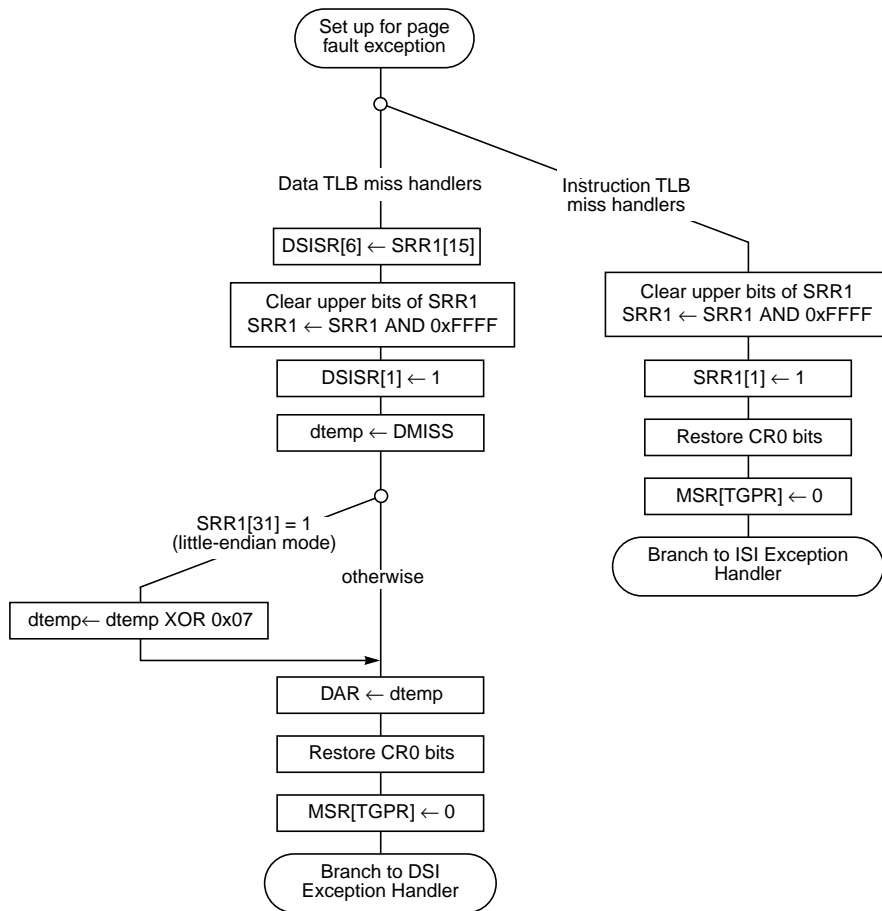
Figure 5-20. Page Fault Setup Flow
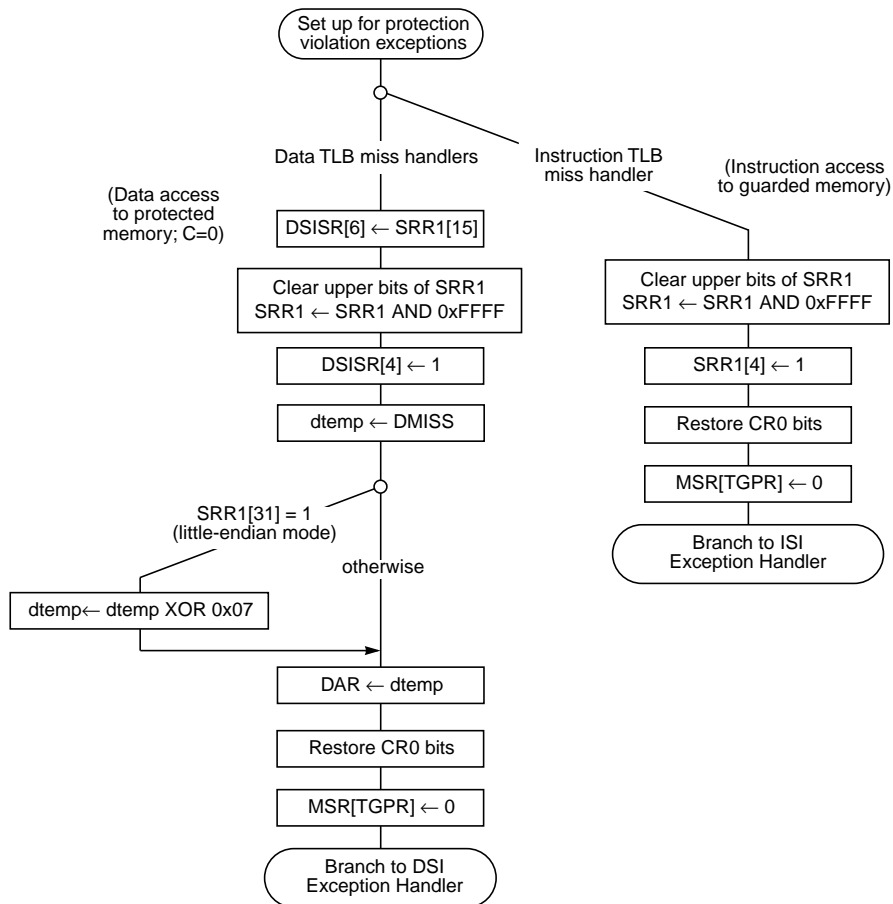
Figure 5-21. Setup for Protection Violation Exceptions

5.5.2.2.2 Code for Example Exception Handlers

This section provides some assembly language examples that implement the flow diagrams described above. Note that although these routines fit into a few cache lines, they are supplied only as a functional example; they could be further optimized for faster performance.

Note that a copy of this code can be downloaded by accessing one of the online facilities listed in "Motorola Electronic Support," and "IBM Electronic Support," in the preface of this document.

```
# TLB software reload
#
# New Instructions:
#         dtba      - write the dtb with the pte in rpa reg
```

```
#         itba          - write the itb with the pte in rpa reg
# New SPRs
#         dmiss         - address of dstream miss
#         imiss         - address of istream miss
#         hash1         - address primary hash PTEG address
#         hash2         - returns secondary hash PTEG address
#         iCmp          - returns the primary istream compare value
#         dCmp          - returns the primary dstream compare value
#         rpa           - the second word of pte used bye xtba
#
# gpr r0..r4 are shadowed
#
# there are three flows.
#         tlbDataMiss- tb miss on data load
#         tlbchange   - tb store with change bit == 0
#         tlbInstrMiss- tb miss on instruction Fetch
#
#+
# place labels for rel branches
#-
#.machine PPC_603
#.set     r0, 0
#.set     r1, 1
#.set     r2, 2
#.set     r3, 3
#.set     dMiss, 1010
#.set     dCmp,  1011
#.set     hash1, 1012
#.set     hash2, 1013
#.set     iMiss, 1014
#.set     iCmp,  1015
#.set     rpa, 1010
#.set     c0, 0
#.set     dar, 19
#.set     dsisr, 18
#.set     srr0, 26
#.set     srr1, 27
#.set     tlbldR0, 0x7E0007A4
#.set     tlbliR0, 0x7E0007E4

#.csect tlbmiss[PR]
#vec0:
#.globl vec0
#

.orig     0x300
vec300:
.orig     0x400
vec400:

#+
```

```
# Instruction TB miss flow
# Entry:
#          Vec = 1000
#          srr0          -> address of instruction that missed
#          srr1          -> 0:3=cr0 4=lru way bit 16:31 = saved MSR
#          msr<tgpr> -> 1
#          iMiss         -> ea that missed
#          iCmp          -> the compare value for the va that missed
#          hash1         -> pointer to first hash pteg
#          hash2         -> pointer to second hash pteg
#
# Register usage:
#          r0 is saved counter
#          r1 is junk
#          r2 is pointer to pteg
#          r3 is current compare value
#-


        .orig       0x1000

tlbInstrMiss:
            mfspr      r2,hash1          # get first pointer
            addi       r1,r0,8           # load 8 for counter
            mfspr      r0,ctr            # save counter
            mfspr      r3,iCmp           # get first compare value
            addi       r2,r2,-8          # predecrement pointer
im0:        mtspr      ctr,r1            # load counter
im1:        lwzu       r1,8(r2)          # get next pte
            cmp        0, r1,r3          # see if found pte
            bdnzf      eq,im1
            bne        instrSecHash
            lwz        r1,4(r2)          # load tlb entry PTE lower-word
            andi.      r3,r1,8           # check G-bit
            bne        doISIg
            mtspr      ctr,r0            # restore counter
            mfspr      r0,iMiss          # get the miss address for the dtba
            mfspr      r3,srr1           # get the saved cr0 bits
            mtcrf      0x80,r3           # restore CR0
            mtspr      rpa,r1            # set the PTE
            ori        r1,r1,0x100       # set reference bit
            rlwinm     r1,r1,24,8,31     #get byte 7 of PTE
            tlbli      r0                # load the tb
            stb        r1,6(r2)          # update page table
            rfi                          # and back we go
#+
# Register usage:
#          r0 is saved counter
#          r1 is junk
#          r2 is pointer to PTEG
#          r3 is current compare value
#-
```

```
instrSecHash:
          andi.     r1,r3,0x0040    # see if we have done second hash
          bne       doISI
          mfspr     r2,hash2        # get the second pointer
          ori       r3,r3,0x0040    # change the compare value
          addi      r1,r0,8         # load 8 for counter
          addi      r2,r2,-8        # pre dec for update on load
          b         im0             # try second hash
#+
# entry Not Found: cause an isi exception
# guarded storage protection violation: cause an isi exception
# Entry:
#         r0 is saved counter
#         r1 is junk
#         r2 is pointer to PTEG
#         r3 is current compare value
#-

doISIg:   mfspr     r3,srr1         # get srr1
          andi.     r2,r3,0xffff    # clean upper srr1
          addis     r2,r2,0x1000    # or in srr<3> = 1 to flag guarded violation
          b         isi1:

doISIp:
          mfspr     r3,srr1         # get srr1
          andi.     r2,r3,0xffff    # clean upper srr1
          addis     r2,r2,0x0800    # or in srr<4> = 1 to flag prot violation
          b         isi1:
doISI:
          mfspr     r3,srr1         # get srr1
          andi.     r2,r3,0xffff    # clean upper srr1
          addis     r2,r2,0x4000    # or in srr<1> = 1 to flag PTE not found

isi1:     mtspr     ctr,r0          # restore counter
          mtspr     srr1,r2         # set srr1
          mfmsr     r0              # get msr
          xoris     r0,r0,0x0002    # flip the msr<tgpr> bit
          mtcrf     0x80,r3         # restore CR0
          mtmsr     r0              # flip back to the native gprs
          isync                     # sync the mtmsr
          b         vec400          # go to isi exception
#+
# Data TB miss flow
# Entry:
#         Vec = 1100
#         srr0      -> address of instruction that caused data tb miss
#         srr1      -> 0:3=cr0 4=lru way bit 5=1 if store 16:31 = saved MSR
#         msr<tgpr> -> 1
#         dMiss     -> ea that missed
#         dCmp      -> the compare value for the va that missed
#         hash1     -> pointer to first hash PTEG
```

```
#          hash2      -> pointer to second hash PTEG
#
# Register usage:
#          r0 is saved counter
#          r1 is junk
#          r2 is pointer to PTEG
#          r3 is current compare value
#-

#.csect    tlbmiss[PR]
.orig      0x1100

tlbDataMiss:
           mfspr      r2,hash1        # get first pointer
           addi       r1,r0,8         # load 8 for counter
           mfspr      r0,ctr          # save counter
           mfspr      r3,dCmp         # get first compare value
           addi       r2,r2,-8        # pre dec the pointer
dm0:       mtspr      ctr,r1          # load counter
dm1:       lwzu       r1,8(r2)        # get next PTE
           cmp        0,r1,r3         # see if found PTE
           bdnzf      eq,dm1
           bne        dataSecHash
           lwz        r1,4(r2)        # load tlb entry PTE lower-word
           mtspr      ctr,r0          # restore counter
           mfspr      r0,dMiss        # get the miss address for the dtba
           mfspr      r3,srr1         # get the saved cr0 bits
           mtcrf      0x80,r3         # restore CR0
           mtspr      rpa,r1          # set the PTE
           ori        r1,r1,0x100     # set reference bit
           rlwinm     r1,r1,24,8,31   # get byte 7 of PTE
           tlbld      r0              # load the tlb
           stb        r1,6(r2)        # update page table
           rfi                        # and back we go
#+
# Register usage:
#          r0 is saved counter
#          r1 is junk
#          r2 is pointer to PTEG
#          r3 is current compare value
#-
dataSecHash:
           andi.      r1,r3,0x0040    # see if we have done second hash
           bc         !CR,EQ,doDSI    # yes, take dsi exception
           mfspr      r2,hash2        # get the second pointer
           ori        r3,r3,0x0040    # change the compare value
           addi       r1,r0,8         # load 8 for counter
           addi       r2,r2,-8        # predecrement for update on load
           b          dm0             # try second hash

#+
```

```
# entry not found: cause a dsi exception
# protection violation: cause a dsi exception
# Entry:
#           r0 is saved counter
#           r1 is junk
#           r2 is pointer to PTEG
#           r3 is current compare value
#-
doDSI:
          mfspr     r3,srr1           # get srr1
          rlwinm    r1,r3,9,6,6       # get srr1<flag> to bit 6 for load/store, zero rest
          addis     r1,r1,0x4000      # or in dsisr<1> = 1 to flag PTE not found
          b         dsi1:
doDSIp:
          mfspr     r3,srr1           # get srr1
          rlwinm    r1,r3,9,6,6       # get srr1<flag> to bit 6 for load/store, zero rest
          addis     r1,r1,0x0800      # or in dsisr<4> = 1 to flag prot violation
dsi1:     mtspr     ctr,r0            # restore counter
          andi.     r2,r3,0xffff      # clear upper bits of srr1
          mtspr     srr1,r2           # set srr1
          mtspr     dsisr,r1          # load the dsisr
          mfspr     r1,dMiss          # get miss address
          rlwinm.   r2,r2,0,31,31     # test LE bit
          bne       dsi2
          xori      r1,r1,0x07        # demunge the data address
dsi2:     mtspr     dar,r1            # put in dar
          mfmsr     r0                # get msr
          xoris     r0,r0,0x0002      # clear the msr<tgpr> bit
          mtcrf     0x80,r3           # restore CR0
          mtmsr     r0                # flip back to the native gprs
          isync                       # sync the mtmsr
          b         vec300            # go to dsi exception


#+
# C=0 TB flow
# Entry:
#           Vec = 1200
#           srr0       -> address of store that caused the trap
#           srr1       -> 0:3=cr0 4=lru way bit 5=1 16:31 = saved MSR
#           msr<tgpr>  -> 1
#           dMiss      -> ea that missed
#           dCmp       -> the compare value for the va that missed
#           hash1      -> pointer to first hash PTEG
#           hash2      -> pointer to second hash PTEG
#
# Register usage:
#           r0 is saved counter
#           r1 is junk
#           r2 is pointer to PTEG
#           r3 is current compare value
#-
```

```
#.csect     tlbmiss[PR]
.orig       0x1200

tlbCeq0:
            mfspr     r2,hash1        # get first pointer
            addi      r1,r0,8         # load 8 for counter
            mfspr     r0,ctr          # save counter
            mfspr     r3,dCmp         # get first compare value
            addi      r2,r2,-8        # pre dec the pointer
ceq0:       mtspr     ctr,r1          # load counter
ceq1:       lwzu      r1,8(r2)        # get next PTE
            cmp       0, r1,r3        # see if found PTE
            bdnzf     eq,ceq1
            bne       cEq0SecHash
            lwz       r1,4(r2)        # load tlb entry PTE lower-word
            andi.     r3,r1,0x80      # check the C-bit
            beq       cEq0ChkProt
            mtspr     ctr,r0          # restore counter
            mfspr     r0,dMiss        # get the miss address for the dtba
            mfspr     r3,srr1         # get the saved cr0 bits
            mtcrf     0x80,r3         # restore CR0
            mtspr     rpa,r1          # set the PTE
            tlbld     r0              # load the tb
            rfi                       # and back we go
#+
# Register usage:
#           r0 is saved counter
#           r1 is junk
#           r2 is pointer to PTEG
#           r3 is current compare value
#-
cEq0SecHash:
            andi.     r1,r3,0x0040    # see if we have done second hash
            bne       doDSI
            mfspr     r2,hash2        # get the second pointer
            ori       r3,r3,0x0040    # change the compare value
            addi      r1,r0,8         # load 8 for counter
            addi      r2,r2,-8        # pre dec for update on load
            b         ceq0            # try second hash
#+
# entry found and PTE(c-bit==0):
# (check protection before setting PTE(c-bit)
# Register usage:
#           r0 is saved counter
#           r1 is PTE entry
#           r2 is pointer to PTEG
#           r3 is trashed
#-

# Note: The following code is specific to 603e. It uses
```

```
# the 'KEY' bit of SRR0 to speedup protection checking.
#
cEq0ChkProt:
           rlwinm.     r3,r1,30,0,1      # test PP
           bge-        chk0
           andi.       r3,r1,1           # test PP[0]
           beq+        chk2
           b           doDSIp            # else DSIp
chk0:      mfspr       r3,srr1           # get SRR0
           andis.      r3,r3,0x0008      # test the KEY bit SRR0[12]
           beq+        chk2
           b           doDSIp            # else DISp
chk2:      mtspr       ctr,r0            # restore counter
           mfspr       r0,dMiss          # get the miss address for the dtba
           mfspr       r3,srr1           # get the saved cr0 bits
           mtcrf       0x80,r3           # restore CR0
           ori         r1,r1,0x180       # set reference and change bit
           mtspr       rpa,r1            # set the PTE
           tlbld       R0                # load the tb
           sth         r1,6(r2)          # update page table
           rfi                           # and back we go


# Note: The following code doesn't rely upon the 'KEY' bit
# and works on 603, and 603e. (although slightly slower on 603e)
#
# cEq0ChkProt:
#          rlwinm.     r3,r1,30,0,1      # test PP
#          bge-        chk0              # if (PP==00 or PP==01) goto chk0:
#          andi.       r3,r1,1           # test PP[0]
#          beq+        chk2              # return if PP[0]==0
#          b           doDSIp            # else DSIp
# chk0:    mfspr       r3,srr1           # get old msr
#          andi.       r3,r3,0x4000      # get PR bit
#          beq+        chk1              # if (PR==0) goto chk1:
#          mfspr       r3,dmiss          # get miss address
#          mfsrin      r3,r3             # get associated segment register
#          andi.       r3,r3,4           # test Kp bit
#          beq+        chk2              # if (Kp==0) goto chk2
#          b           doDSIp            # else DSIp
# chk1:    mfspr       r3,dmiss          # get miss address
#          mfsrin      r3,r3             # get associated segment register
#          andi.       r3,r3,2           # test Ks bit
#          beq+        chk2              # if (Ks==0) goto chk2:
#          b           doDSIp            # else DSIp
# chk2:    mtspr       ctr,r0            # restore counter
#          mfspr       r0,dMiss          # get the miss address for the dtba
#          mfspr       r3,srr1           # get the saved cr0 bits
#          mtcrf       0x80,r3           # restore CR0
#          ori         r1,r1,0x180       # set reference and change bit
#          mtspr       rpa,r1            # set the PTE
```

```
#        tlbld       R0             # load the tb
#        sth         r1,6(r2)       # update page table
#        rfi                        # and back we go
```

5.5.3 Page Table Updates

When TLBs are implemented (as in the 602) they are defined as noncoherent caches of the page tables. TLB entries must be flushed explicitly with the TLB invalidate entry instruction (**tlbie**) whenever the corresponding PTE is modified. As the 602 is intended primarily for uniprocessor environments, it does not provide coherency of TLBs between multiple processors. If the 602 is used in a multiprocessor environment where TLB coherency is required, all synchronization must be implemented in software.

Processors may write referenced and changed bits with unsynchronized, atomic byte-store operations. Note that the V, R, and C bits each resides in a distinct byte of a PTE. Therefore, extreme care must be taken to use byte-writes when updating only one of these bits.

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.

5.5.4 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*.

5.6 Protection-Only Mode

The 602 implements an additional memory management resource not defined by the PowerPC architecture called protection-only mode, which is activated after a BAT miss when HID0[PO] = 1. In this mode, the effective address is used as the physical address. However, unlike the OEA-defined real addressing mode, some protection is provided. This is accomplished by using, and in some cases redefining, resources used in page address translation. For example, in protection-only mode the bits in the DTLB entries are redefined to hold 32 WE (write enable) bits, which indicate whether the corresponding 4-Kbyte page of data space is write-protected. Likewise, the ITLB entries are redefined to hold 32 NE (no-execute) bits to indicate whether the corresponding page allows instruction execution.

In protection-only mode (as in real addressing mode), the default cache attributes are controlled by the settings of HID0[WIMG] (bits 28–31).

Implementation Note—When the processor is in either real addressing mode or protection-only mode, care should be taken when clearing HID0[G]. The 602 allows out-of-order loads to access the processor bus. If an out-of-order load follows an instruction that causes an exception, the load/store unit may pass the out-of-order load operation onto the system bus. Because this load cannot be cancelled, depending on the temporal position of the faulting instruction, translation may be enabled when the instruction passes through the instruction stream but may be disabled when the cache control information and address reach the bus. Setting HID0[G] prevents such a load operation from accessing the bus.

5.6.1 Use of Translation Resources in Protection-Only Mode

The instruction and data MMUs each use a dedicated 32-entry, two-way set-associative TLB. In the OEA-defined page addressing mode, described in Section 5.4, "Memory Segment Model," each TLB entry provides translation and protection for a 4-Kbyte page in memory. In protection-only mode, the TLBs are not used for storing page address translations, so the bits are redefined such that each TLB entry maps a 128-Kbyte region with bits within the TLB entry that provide protection information for the 32, 4-Kbyte pages within that region.

If HID0[PO] = 1 and a hit occurs in the TLB, the NE and WE bits are used for checking access permissions for instruction and data accesses, respectively. The cache control bits (WIMG) are set as programmed in the HID0 register. The effective address is used as the physical address.

To locate the NE or WE bit for this page, the effective address and 24-bit VSID from SR0 are used to create a 56-bit virtual address. This address is used to index into and compare against entries in the TLB and can be used as a process ID for operations in protection-only mode for locating these bits.

Figure 5-22 shows the TLB lookup operation when the processor is in protection-only mode. The VSID is taken only from SR0 (when the processor is in page addressing mode, all 16 segment registers can be used). Note that in protection-only mode, SR0[N], which ordinarily controls instruction fetching at the segment level, is ignored and no-execute privileges are configured on a 4-Kbyte page basis using the NE bits. EA0–EA14 and the VSID from SR0 are used to select the correct ITLB entry. From this entry, EA15–EA19 select the correct NE bit for the page of the access.
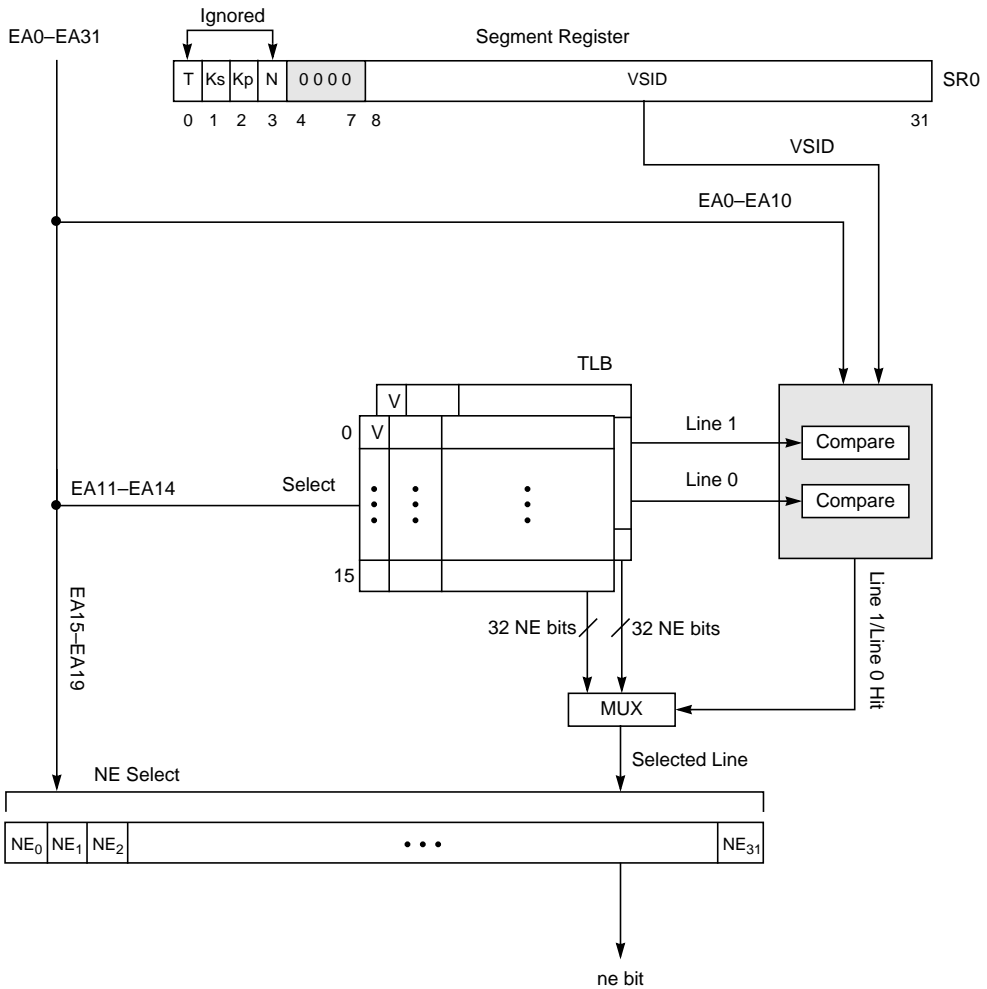
Figure 5-22. TLB Lookup Operation in Protection-Only Mode

5.6.1.1 TLB Misses in Protection-Only Mode

Regardless of whether the processor is in protection-only mode, when a TLB miss occurs, one of three exceptions is taken—an ITLB miss exception, a DTLB miss on load exception, or a DTLB miss on store (or C = 0) exception. Likewise, when the TLB miss occurs in protection-only mode, the same resources used to perform the software tablewalk in page accessing mode are available in protection-only mode, as follows:

- The IMISS and DMISS registers hold the missed effective address.
- The HASH1 and HASH2 registers hold the PTEG address.

- The ICMP and DCMP registers are used for comparing PTEs.
- The RPA register is used for loading TLBs; however, in protection-only mode, the RPA holds the 32 WE or NE bits that are loaded into the TLB entries.

Note that the TLB miss mechanism can be used to maintain translation and configuration information for pages that extend beyond the 32 TLB entries that can be defined for each MMU in protection-only mode. These exceptions are described in Chapter 4, "Exceptions."

Descriptions of exception handlers are described in Section 5.5.2.2.1, "Flow for Example Exception Handlers." Note that these examples are conceptual and provide a general notion of what is possible for an exception handler in protection-only mode.

5.6.1.2 Access Protection in Protection-Only Mode

In protection-only mode, pages mapped by the ITLB are protected by the NE bit in the TLB and SE bits from the SER. DTLB pages are protected by only the WE bit in the PTE. For instruction fetches, the NE bit controls execute privileges in general and the SE bit controls whether the **esa** instruction can be executed from the corresponding page. Note that if fetching is disabled (NE = 1), no instructions can be fetched (including the **esa** instruction) and SE is a don't care.

For store instructions, the WE bit controls write access to a page; read access is permitted for all pages. All loads from the data cache or memory are permitted for pages mapped in the DTLB, but data cannot be stored unless the appropriate WE bit in the DTLB is set. Note that as with ITLB access permissions, the SR0[T] or SR0[N] bits are not used to determine DTLB access privileges in protection-only mode.

5.6.1.3 Required Physical Address Register in Protection-Only Mode

During a page table search operation, the RPA register is loaded with the second word of the appropriate PTE. In protection-only mode, the PTE contains the 32 WE or NE bits loaded into the TLBs via the RPA register. Also, as shown in Figure 5-23 and Figure 5-24, the contents of the RPA register are different for instruction and data TLB loads.

Before the TLB Load Instruction (**tlbli**) is executed in protection-only mode, the RPA register should be loaded with 32 NE bits. Figure 5-23 shows the proper contents of the RPA for an ITLB load.

NE_0	NE_1	NE_2	NE_3	NE_4	NE_5	NE_6	NE_7	NE_8	NE_9	NE_{10}	NE_{11}	NE_{12}	NE_{13}	NE_{14}	NE_{15}	NE_{16}	NE_{17}	NE_{18}	NE_{19}	NE_{20}	NE_{21}	NE_{22}	NE_{23}	NE_{24}	NE_{25}	NE_{26}	NE_{27}	NE_{28}	NE_{29}	NE_{30}	NE_{31}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
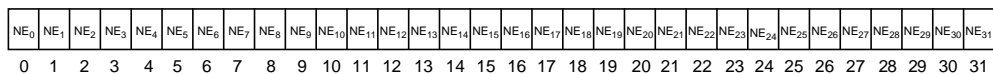
Figure 5-23. RPA for ITLB Load Operations in Protection-Only Mode

As shown in Figure 5-24, before a TLB Load Data (**tlbld**) instruction is executed, the RPA register should be loaded with 32 WE bits.

WE$_0$	WE$_1$	WE$_2$	WE$_3$	WE$_4$	WE$_5$	WE$_6$	WE$_7$	WE$_8$	WE$_9$	WE$_{10}$	WE$_{11}$	WE$_{12}$	WE$_{13}$	WE$_{14}$	WE$_{15}$	WE$_{16}$	WE$_{17}$	WE$_{18}$	WE$_{19}$	WE$_{20}$	WE$_{21}$	WE$_{22}$	WE$_{23}$	WE$_{24}$	WE$_{25}$	WE$_{26}$	WE$_{27}$	WE$_{28}$	WE$_{29}$	WE$_{30}$	WE$_{31}$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
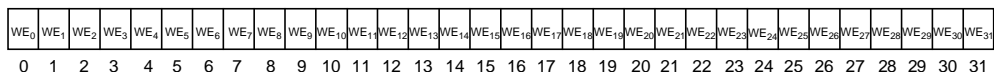
Figure 5-24. RPA for DTLB Load Operations in Protection-Only Mode

5.6.2 ESA Enable Protection (Instruction Space Only)

The 602 defines a user-level instruction, Enable Supervisor Access (**esa**) that, when successfully executed, allows the processor to function in supervisor mode without taking an exception. To control the execution of this instruction, it must be enabled for the block or page on which it resides, and for this purpose an extra bit (SE) is implemented in the BAT registers for block address translations and in the TLB/PTEs for page address translations. The use of this instruction is not supported for real addressing mode (MSR[IR] or MSR[DR] = 0).

The 602 defines two additional registers, the ESA enable base register (SEBR) and the ESA enable register (SER), that allow the use of the **esa** instruction in protection-only mode. These two registers work together to control **esa** privileges for each 4-Kbyte block in much the same way as the TLBs provide no-execute and write-enable protection for instruction and data space.

As described in the preceding section, in protection-only mode, the TLBs store only one protection bit for each 4-Kbyte page—an NE bit for each page of instruction space and a WE bit for each page of data space. Similarly, the SE protection registers provide an additional protection bit for 32 contiguous 4-Kbyte pages (128-Kbyte region). This additional protection bit controls whether an **esa** instruction from the region specified in the SEBR can be executed.

The SEBR, shown in Figure 5-25, contains the base address of the 128-Kbyte region that is protected by the SE bits in the SER. The 15-bit base address in this register corresponds to bits EA0–EA14. For the correct SE bit to be located, the effective address of an instruction fetch must match the base address in SEBR. If the effective address does not match the base address, a value of SE = 0 is assumed, and the execution of **esa** instructions in that page is disabled.

☐ Reserved

Base Address	0000000000000000
0 14	15 31
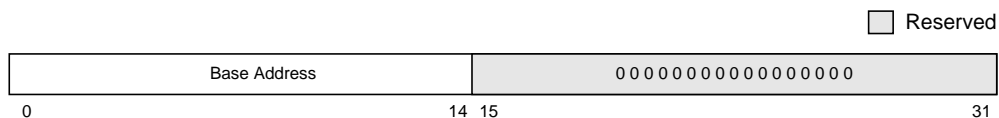
Figure 5-25. ESA Enable Base Register (SEBR)

If a match occurs, bits EA14–EA19 identify the SER bit that corresponds to the 4-Kbyte page associated with the **esa** instruction. The SER, shown in Figure 5-26, contains 32 SE bits.

SE$_0$	SE$_1$	SE$_2$	SE$_3$	SE$_4$	SE$_5$	SE$_6$	SE$_7$	SE$_8$	SE$_9$	SE$_{10}$	SE$_{11}$	SE$_{12}$	SE$_{13}$	SE$_{14}$	SE$_{15}$	SE$_{16}$	SE$_{17}$	SE$_{18}$	SE$_{19}$	SE$_{20}$	SE$_{21}$	SE$_{22}$	SE$_{23}$	SE$_{24}$	SE$_{25}$	SE$_{26}$	SE$_{27}$	SE$_{28}$	SE$_{29}$	SE$_{30}$	SE$_{31}$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Figure 5-26. ESA Enable Register (SER)

The SER and SEBR registers do not affect protection checking unless the processor is in protection-only mode. These registers can be read and written to by using the **mfspr** and **mtspr** instructions but are not used by the MMU unless the processor is in protection-only mode.

5.6.3 Translation Flow in Protection-Only Mode

Figure 5-27 shows a detailed flow diagram of how the translation mechanism is used to locate the NE and WE bits when the processor is in protection-only mode. It assumes that a BAT miss has occurred and that the HID0[PO] bit is set.

The flow diagram shows first how the appropriate key bit from SR0 is checked. If the key is 0, the NE and WE bits are predetermined, as shown in Figure 5-28.

If the key bit is 1, the WE bit is selected for data accesses and the NE bit is selected for instruction accesses. Note that in case of a TLB miss, one of the three TLB miss exceptions is taken—an ITLB miss exception, a DTLB miss on load exception, or a DTLB miss on store (or C = 0) exception, as is the case when the processor is not operating in protection-only mode.

For instruction accesses, if the NE bit allows instructions to be fetched, the SE bit is accessed and if an **esa** instruction is fetched from this page, the SE value is used to determine whether it can be executed.

Protection-Only
Translation Operation

Get Ks, Kp from SR0
Ks ← SR0[1]
Kp ← SR0[2]

Select Key:
If MSR[PR] = 0, Key = Ks
If MSR[PR] = 1, Key = Kp

key = 0

See Figure 5-28   key = 1

Use Upper 44 Bits of VA to Check for
TLB Hit (to Get NE/WE Bit)

DTLB hit
(Data Access)   TLB Miss   ITLB hit
(Instruction Access)

we ← Selected WE Bit   Exception
Table Search S/W
(See Section 5.6.1.1)   ne ← Selected NE Bit

we = 1   we = 0   ne = 1   ne = 0

Load   Store   SEBR[0–14] =
EA[0–14]   otherwise

DSISR[4] ← 1   SRRI[3] ← 1   index ← EA[15-19]   se ← 0

DSI Exception   ISI Exception   se ← SER[index]

End of Key = 0 Flow

(from Figure 5-28)

PA ← EA
Append se bit
to address

Perform Access to/from Cache
Instruction Accesses Allowed as Shown in Table 5-2

Figure 5-27. Translation Flow in Protection-Only Mode

PowerPC 602 RISC Microprocessor User's Manual

Figure 5-28 shows how the NE, SE, and WE bits are determined when the key = 0. Note that the NE and WE bits are predetermined such that write access is enabled for data pages (WE = 1) and execution is enabled for instruction pages (NE = 0). The value of SE is determined by comparing SEBR[0–14] with the low-order 15 bits of the effective address. If a match occurs, the **esa** instruction is enabled; otherwise, it is disabled.
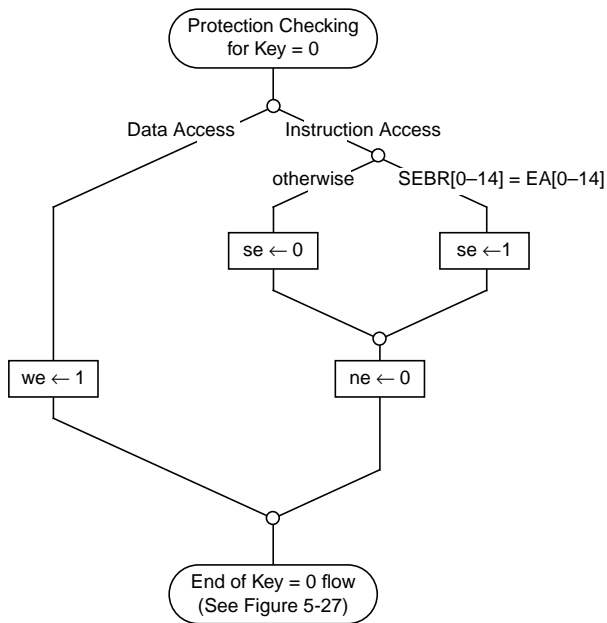
Figure 5-28. Protection Checking with Key = 0 in Protection-Only Mode

PowerPC 602 RISC Microprocessor User's Manual

Chapter 6
Instruction Timing

This chapter describes instruction prefetch and execution through all of the execution units of the PowerPC 602 microprocessor. It also provides examples of instruction sequences showing concurrent execution and various register dependencies to illustrate timing interactions. Bus signals described in this chapter are only accurate to within half-clock cycle increments. See Chapter 8, "System Interface Operation," for more specific information regarding bus operation timing. Instruction mnemonics for PowerPC instructions used in this chapter can be identified by referring to *The Programming Environments Manual*; 602-specific instructions are discussed in Section 2.3, "Instruction Set Summary."

6.1 Instruction Timing Overview

As shown in Figure 6-1, the 602 handles instructions in such a way that (with the exception of branch instructions) they are fetched, dispatched, and completed in program order. However, the 602 implements three independent execution units, which allow multiple instructions to be executed simultaneously. After instructions are executed, they are completed and their results are written back to the architected registers (for example, the FPRs or GPRs) in program order.

As a PowerPC processor, the 602 has been designed to minimize average instruction execution latency. Latency is defined as the number of clock cycles necessary to execute an instruction and make ready the results of that execution for a subsequent instruction. For many of the instructions in the 602, this can be simplified to include only the execute phase for a particular instruction. However, data access instructions require additional clock cycles between the execute and the completion/writeback stage due to memory latencies. Most integer and logical 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 execute.
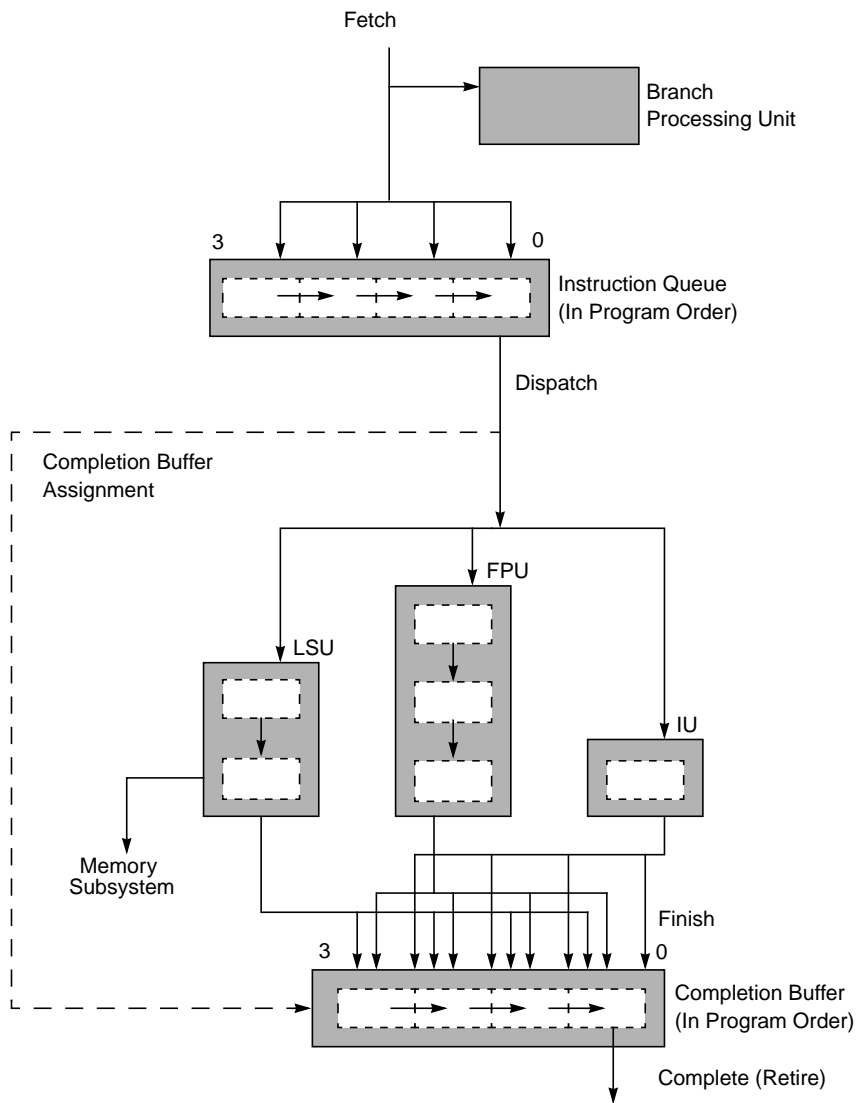
Figure 6-1. Instruction Flow Diagram

Effective throughput approaching one instruction per clock cycle can be realized by the many performance features in the 602 including pipelining, branch folding, rename registers, and multiple execution units that operate independently and in parallel.

The load/store and floating-point units on the 602 are pipelined, which means that the execution units are broken into stages. Each stage performs a specific step, which contributes to the overall execution of an instruction. The pipelined design is analogous to

an assembly line where workers perform a specific task and pass the partially complete product to the next worker.

Figure 6-2 shows a graphical representation of a typical pipelined execution unit.

CLOCK 0	(STAGE 1) A	(STAGE 2)	(STAGE 3)
CLOCK 1	(STAGE 1) B	(STAGE 2) A	(STAGE 3)
CLOCK 2	(STAGE 1) C	(STAGE 2) B	(STAGE 3) A
CLOCK 3	(STAGE 1) D	(STAGE 2) C	(STAGE 3) B

Figure 6-2. Pipelined Execution Unit

When an instruction is issued to a pipelined execution unit, the first stage in the pipeline begins its designated work on that instruction. As an instruction is passed from one stage in the pipeline to the next, evacuated stages may accept new instructions. This design allows a single execution unit to be working on several different instructions simultaneously. While it may take several cycles for a given instruction to propagate through the execution pipeline, once the pipeline has been filled with instructions the execution unit can complete one instruction per clock.

If the number of stages in each pipeline equals the total latency in clock cycles of its respective execution unit, the processor can continuously issue instructions to the same execution unit without stalling. Thus, when enough instructions have been issued to a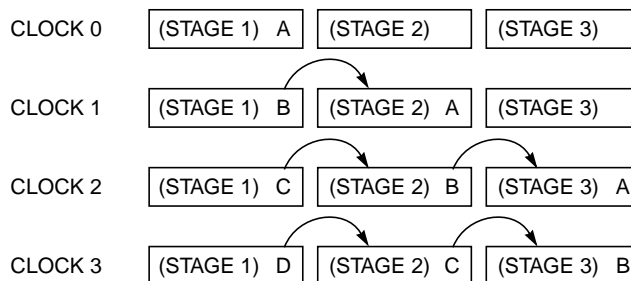n execution unit to fill its pipeline, the first instruction completes execution and exits the pipeline, allowing subsequent instructions to be issued into the tail of the pipeline without interruption. This is illustrated for the 602's three execution units in Figure 6-3 in the following section.

The 602's completion buffer can retire one instruction on every clock cycle. In general, instruction processing is accomplished in four stages—the fetch stage, the dispatch stage, the execute stage, and the completion/writeback 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. Operations specified by the instruction are performed during the execute stage. In the completion/writeback stage, the results of the execute stage are used to update the architected registers. The completion buffer ensures that instructions write back and are retired in program order and also ensures the 602's precise exception model.

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 PowerPC 602 Microprocessor Pipeline Organization

The instruction pipeline of the 602 has four major stages—fetch, dispatch, execute, and complete/writeback. Each instruction executed by the machine flows through some or all of these stages as shown in Figure 6-3. Some instructions spend multiple cycles in a stage.

Integer Unit Instruction Pipeline

Load Store Unit Instruction Pipeline

LSU1—EA Calculation
LSU2—Cache Access

Floating-Point Unit Instruction Pipeline
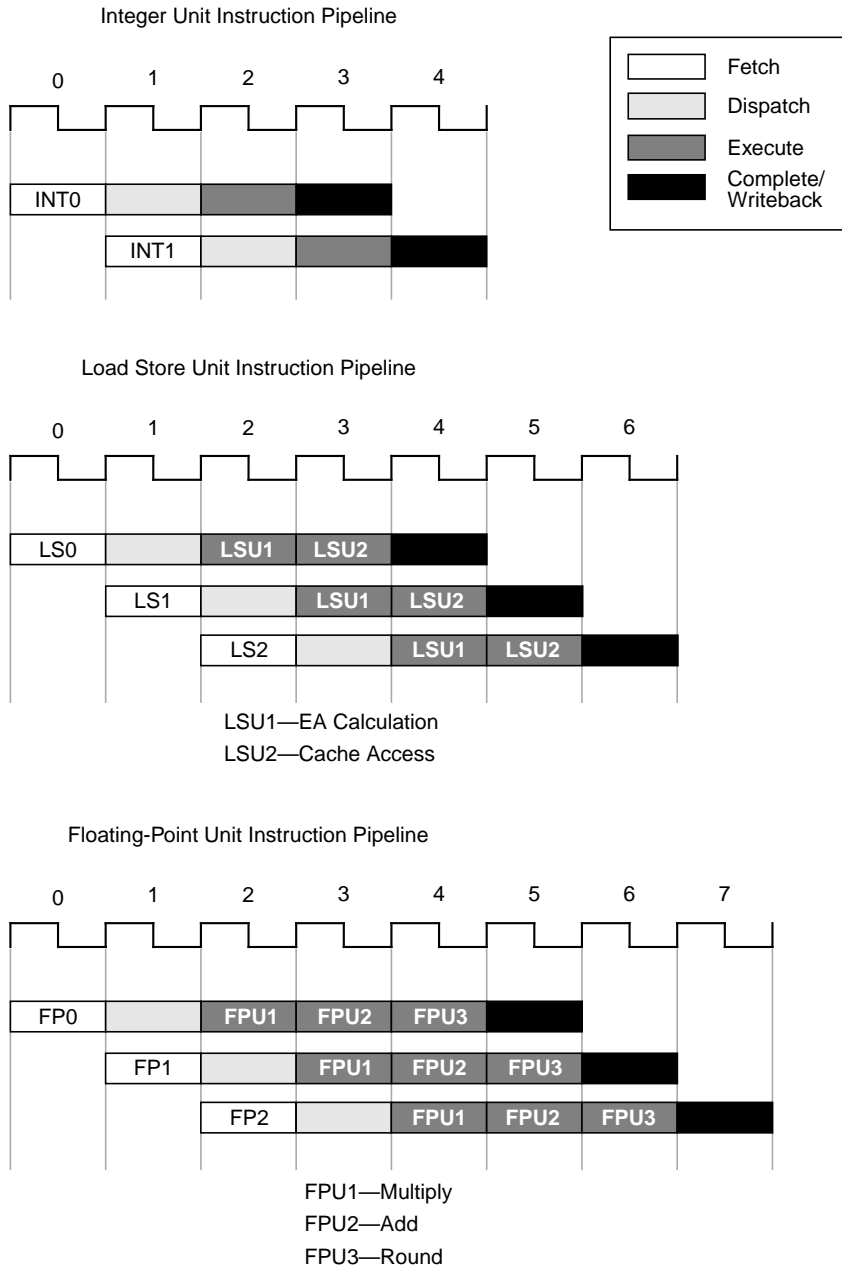
FPU1—Multiply
FPU2—Add
FPU3—Round

Figure 6-3. Pipeline Diagrams for the PowerPC 602 Processor Execution Units

Note that the timing examples in Figure 6-3 are for typical instructions and do not apply to all instructions that execute in a particular unit.

The stages in the 602 are described as follows:

- The fetch stage primarily involves retrieving instructions from the memory system and determining where the next instruction fetch should occur. The instructions retrieved from the memory system are either latched into an instruction buffer or the dispatch buffer for subsequent consideration by the dispatch stage. The BPU also decodes branches during the fetch stage and attempts to fold out branch instructions.

- The dispatch stage decodes the instructions supplied by the instruction fetch stage, and determines which instructions can be dispatched in the current cycle. In addition, the source operands are read from the appropriate register file and dispatched with the instruction to the execute stage. At the end of the dispatch stage, dispatched instructions and their operands are latched into execution unit input latches.

- During the execute stage, each execution unit that has an executable instruction execute the selected instruction (perhaps over multiple cycles), write the instruction's result into the appropriate rename buffer, and notify the completion stage that the instruction has finished execution. In the cases of an internal exception, the execution unit reports the exception to completion/writeback stage and discontinues instruction execution until the exception is handled. The exception is not addressed until that instruction is the next to be completed.

  Execution of instructions is handled by one of three execution units, which operate in parallel:

  — As shown in Figure 6-3, most single-precision floating-point instructions are pipelined within the FPU. The stages for the FPU are multiply, add, and normalize-round. A different instruction can occupy each stage, allowing up to three instructions to be executing in the FPU concurrently.

  — Execution of most load or store instructions is pipelined. As indicated in Figure 6-3, the LSU has two stages—effective address calculation and MMU translation is performed in the first stage and data is accessed in the cache in the second stage.

  — The integer unit consists of a single stage, and most instructions that execute in the integer stage take only one clock cycle in that stage. Some instructions, such as integer divides and multiplies, take multiple cycles in that integer execute stage and subsequent instructions that execute in the integer unit wait for those instructions to execute.

  Note that because instruction results are written to the rename buffers at the end of the last execution cycle, those results are available for use by any instructions that need those results as source operands. However, those results are not written to the architected registers until the end of the completion/writeback stage.

- The complete/writeback stage maintains the correct architectural machine state and commits it to the machine architectural registers at the proper time. If the completion logic detects an instruction containing an exception status, all following instructions are cancelled, their execution results in rename buffers are discarded, and the correct instruction stream is fetched.

6.3 Timing Considerations

Although the 602 appears to the programmer to execute instructions in sequential order, the 602 provides increased performance by executing multiple instructions at a time, and using hardware to manage dependencies. All instructions (except for those resolved branch instructions that can be folded out of the instruction stream) complete and write back their results to architected registers in program order. However, the use of rename registers allows the results of an instruction that has been executed but not yet allowed to complete to be made available to a subsequent instruction that needs those results as source data.

When an instruction is issued, the register file or its associated rename registers place the appropriate source data on the appropriate bus. The corresponding execution unit then reads the data from the bus.

The 602 contains the following execution units that operate independently and in parallel:

- Branch processing unit (BPU)
- 32-bit integer unit (IU)
- 32-bit floating-point unit (FPU) for single-precision operations
- Load/store unit (LSU)

The 602's branch processing unit decodes and executes branches immediately after they are fetched. The resources of the branch unit include—a count register (CTR) rename register for writing to the CTR with the **mtspr** instruction, a link register (LR) rename register for writing to the LR with the **mtspr** instruction, a link register rename register for branches specifying an update of the link register, and a branch reservation station for conditional branches that cannot be resolved due to a CR-data dependency.

When a conditional branch cannot be resolved due to a CR-data dependency, the branch is predicted and instructions are executed out-of-order down the predicted path. Note that instructions cannot complete and write back their results to architected registers until the branch is resolved; however, they can make their results available to subsequent instructions. If the branch resolves as incorrectly guessed then the following occurs:

1. Instructions that preceded the branch are allowed to complete.
2. The instruction buffer is purged.
3. Fetching of the correct path begins.

When the IU or FPU finishes executing an instruction, it places any results into a rename buffer for general-purpose register (GPR), floating-point register (FPR), link register, counter register, or conditional register. Results are not stored into the associated architected registers until the write-back stage, which helps ensure a precise exception model.

6.3.1 Instruction Fetch Timing

Instruction fetch latency depends on whether the instruction is in the cache (cache hit). If the instruction is not in the cache, additional latency is required to access the instruction from an off-chip memory resource. In turn, this latency is affected by the bus frequency. These issues are discussed further in the following sections.

6.3.1.1 Cache Arbitration

When the instruction fetcher tries to fetch instructions from the on-chip cache, one of two things may occur:

- If the instruction cache is idle and the instructions are in the cache, the cache supplies the requested instructions on the next clock cycle.

- Because the 602's caches are nonblocking during line-fill operations, if the instruction cache is performing a cache-line-reload, the requested instruction is forwarded to the BPU at the same time that is made available in the cache.

  Additionally, if a branch instruction is fetched from the cache block that is being filled, it may point either to another address within same block or to another location.

  — If the target instruction is elsewhere in the same cache block, that instruction can be fetched as soon as it becomes available in the instruction cache.

  — If the target instruction is elsewhere in the cache, the instruction can be fetched without having to wait for the entire cache block to be updated.

  Note, however, that in both of these cases, instructions can be fetched from the cache only while it is not being written to as part of the cache line refill. This is shown in Section 6.3.1.3, "Cache Miss."

6.3.1.2 Cache Hit

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 (in other words, a cache hit has occurred), there is only one clock cycle between the time that the instruction fetcher requests an instruction and the time that the instruction enters the IQ.

Figure 6-4 shows a brief example of instruction fetching that hits in the on-chip cache.
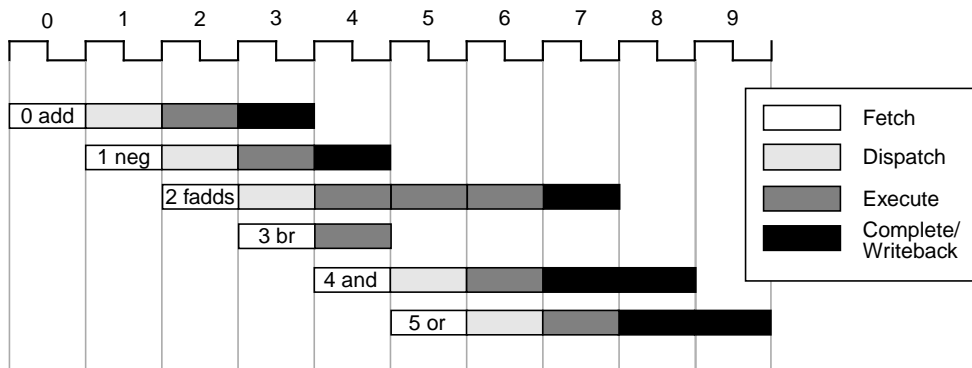
Figure 6-4. Instruction Timing—Cache Hit

1. During clock cycle 0, instruction 0 (an **add** instruction) is fetched.

2. In clock cycle 1, a **neg** instruction that uses the results of instruction 0 as a source operand is fetched while instruction 0 is in the dispatch stage.

3. During clock cycle 2, instruction 2 (**fadds**) is fetched, instruction 1 is in dispatch, and instruction 0 is executed in the IU and its results are placed in a rename buffer, making those results available as a source operand for the **neg** instruction.

   During clock cycle 3, an unconditional branch instruction is fetched into the branch unit. The BPU immediately determines that the branch changed the program flow and sends a request to the on-chip cache for the new instruction stream.

   Previous instructions continue to proceed down the pipeline. Note that even though the results of instruction 0 are not written back to the architected GPR until the end of this cycle, instruction 1 accesses those results from the GPR rename buffer as a source operand and executes without delay.

4. In clock cycle 4, an **and** instruction is fetched from the new path, the branch instruction is folded, and the **fadds** instruction enters the second execute stage of the FPU pipeline, and instruction 1 completes and writes back.

5. In clock cycle 5, instruction 5 (an **or**) is fetched, while previous instructions proceed down the pipeline without encountering stalls.

6. In clock cycle 6, instruction 5 is in the dispatch stage, instruction 4 is in the IU execute stage, and instruction 2 is in the third of the FPU execute stages.

7. In clock cycle 7, both instruction 2 and instruction 4 enter the completion/write back stage. However, because only one instruction can complete and write back per clock cycle, instruction 4 must wait for an additional cycle in order to complete. Note that even though instruction 4, an **and** instruction, cannot complete, it does not prevent the subsequent **or** instruction from going into the IU execute stage.

8. In clock cycle 8, instruction 4 is able to complete and write back, but again because only one instruction can complete per clock cycle, instruction 5 must wait to complete on the subsequent cycle.

6.3.1.3 Cache Miss

Figure 6-5 shows a brief example of an instruction fetch that misses in the on-chip cache and how the bus timing affects the instruction issue. This example shows the simplest bus timing—the processor/bus clock ratio is 2:1, the data bus is in 64-bit mode, the address phase is the shortest possible (a single cycle of the slower bus clock), and there are no wait states between each beat of data.
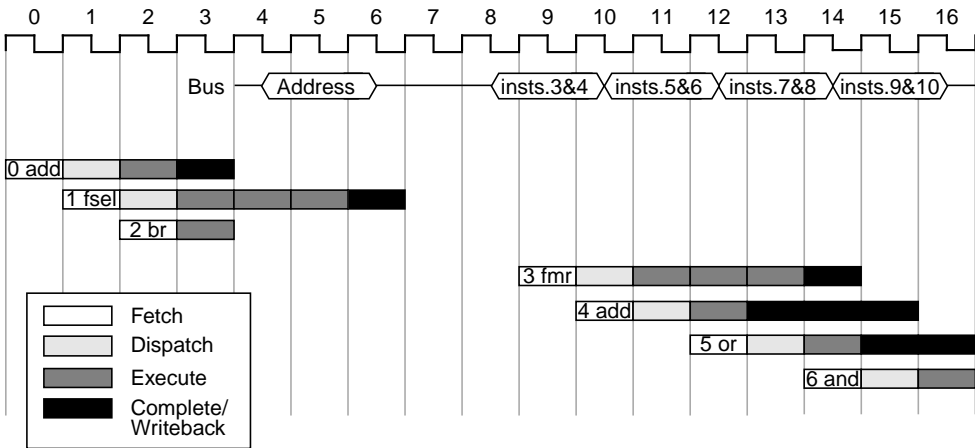
Figure 6-5. Instruction Timing—Cache Miss

1. In clock cycle 0, an integer add instruction (instruction 0) is fetched.

2. In clock cycle 1, an **fsel** instruction (instruction 1) is fetched and instruction 0 is in the dispatch stage.

3. In clock cycle 2, an unconditional branch instruction is fetched, and immediately resolved. The target instruction is not in the instruction cache, so a line-fill operation is required.

4. Clock cycles 4–8 show the bus latency for a burst-read operation in 64-bit mode with a single-cycle address phase.

5. In clock cycle 9, the target instruction (an **fmr** instruction) arrives in beat one and is made available to both the instruction cache and to instruction unit.

6. In clock cycle 10, the second instruction that arrived in the first beat (instruction 4) is fetched from the cache. Note that this is possible because in 2:1 mode, the cache is being updated on alternate clock cycles, leaving it available for instruction fetching on every other clock cycle.

7. In clock cycle 11, the cache is being written to, so no instructions can be fetched. Instructions 3 and 4 continue through the pipeline.

8. Instruction 5 (from data beat 2) is fetched in clock cycle 12.

9. The first instruction from the second data beat (instruction 5) is fetched. This illustrates the fact that the 602's instruction cache is nonblocking, so the fetch logic does not have to wait for the entire cache block to be loaded to fetch additional instructions.

10. Additional instructions are fetched from the cache block without interruption on every other clock cycle. By clock cycle 15, the line-fill operation is complete and instructions can be fetched one instruction per processor clock cycle.

6.3.2 Instruction Dispatch and Completion Considerations

Several factors may affect the 602's ability to dispatch instructions, including availability of the execution units, rename registers, and the completion buffer, and the handling of dispatch-serialized instructions.

To avoid dispatch unit stalls due to instruction data dependencies, the 602 provides a reservation station for each execution unit. If a data dependency prevents an instruction from beginning execution, that instruction is dispatched to the reservation station associated with its execution unit, thereby 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 same clock cycle that the register file is being updated. If the second instruction in the dispatch unit requires the same execution unit, that instruction cannot be dispatched until the first instruction executes.

The completion unit records the program order, and even though instructions may execute out of order, it ensures that the results are written back and the instructions are retired in program order. Completing an instruction implies the commitment of the results of instruction execution to the architected registers and ensures a precise exception model when the 602 must recover from a mispredicted branch or an exception.

Instruction state and all information required for completion is kept in a first-in-first-out (FIFO) queue of four completion buffers. A single completion buffer is allocated for each instruction as it is dispatched by the dispatch unit. If no space is available in the completion buffer available, the dispatch unit stalls. While one instruction per clock cycle can be completed and retired in program order from the completion unit, instruction completion can be stalled by the instruction reaching the last position in the completion queue while the instruction is still being executed.

Because the 602 can execute instructions out of order, the in-order completion 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 before the exception is taken.

6.3.3 Rename Register Operation

To avoid contention for a register file location, the 602 provides rename registers for storing instruction results before the completion unit commits them to the architected register.

The 602 has the following rename register resources:

- Four GPR rename buffers
- Four FPR rename buffers
- One rename buffer each for the CR, LR, and CTR

When the dispatch unit dispatches an instruction to its execution unit, it allocates a rename register for the results of that instruction. If an instruction is dispatched to a reservation station associated with an execution unit due to a data dependency, the dispatcher also provides a tag to the execution unit identifying which rename register forwards the required data upon instruction completion. When the data is available in the rename register, the pending execution may begin.

The completion unit does not transfer instruction results from the rename registers to the architected registers until the instruction can be retired from the completion queue without exceptions and until any unresolved branch conditions preceding it in the completion queue have been resolved. If a branch is mispredicted, instructions associated with the mispredicted branch are flushed from the completion queue and the results of those instructions are flushed from the rename registers.

6.4 Execution Unit Timings

The following sections describe instruction timing considerations within each of the respective execution units in the 602. Refer to Table 6-1 for branch instruction execution timing.

6.4.1 Branch Processing Unit Execution Timing

Flow control operations (conditional branches, unconditional branches, and traps) are typically time-consuming to execute in most machines because they disrupt normal instruction flow. When the program flow changes, the IQ must be reloaded with the target instruction stream, during which time the execution units can only execute instructions dispatched before the change in the program flow. Waiting for instructions to be fetched from the new address can cause inactivity in the execution units.

Performance features such as branch folding and static branch prediction help minimize the penalties associated with flow-control operations.

The timing for branch instruction execution is determined by many factors including the following:

- Whether the branch is taken
- Whether the target instruction stream is in the on-chip cache
- Whether the branch is predicted
- Whether the prediction is correct

6.4.1.1 Branch Folding

When a branch instruction is encountered by the fetcher, the BPU immediately tries to pull that instruction out of the instruction stream and resolve it. When the BPU pulls the branch instruction out of the instruction stream, the instruction above the branch is shifted down to take the place of the removed branch. The technique of removing the branch instruction from the instruction sequence seen by the other execution units is known as branch folding.

Often, if the prediction is correct or if the branch is unconditional, branch folding can reduce the penalties of flow control instructions to zero since instruction execution may proceed as though the branch was never there. A branch can be predicted to be either taken or not taken:

- Branch taken—If the folded branch instruction changes program flow, the BPU immediately requests the instructions at the new target from the on-chip cache. In most cases, the new instructions arrive in the IQ before any bubbles are introduced into the execution units.
- Branch not taken—If the folded branch instruction does not change program flow, the branch instruction is already removed from the instruction stream and execution continues as if no branch instruction were in the original sequence.

When a conditional branch cannot be resolved due to a CR data dependency, the branch is predicted by means of static branch prediction, and instruction fetching proceeds down the predicted path. If the branch prediction was incorrect when the branch is resolved, the instruction queue and all instructions associated with the mispredicted branch are purged. The results of any instructions from the mispredicted branch that may have executed are also purged and are not written to architected registers. Instruction fetching resumes down the correct path.

There are several situations where instruction sequences create dependencies that prevent a branch instruction from being resolved immediately, thereby causing the instructions that are the target of the mispredicted branch to be executed out of order.

The instruction sequences and the resulting action of the branch instruction are described as follows:

- An **mtspr**(LR) followed by a **bclr**—Fetching stops, and the branch waits for the **mtspr** to execute.

- An **mtspr**(CTR) followed by a **bcctr**—Fetching stops, and the branch waits for the **mtspr** to execute.

- An **mtspr**(CTR) followed by a **bc**(CTR)—Fetching stops, and the branch waits for the **mtspr** to execute.

- A **bc**(CTR) followed by another **bc**(CTR)—Fetching stops and the second branch waits for the first branch to be completed.

- A **bc**(CTR) followed by a **bcctr**—Fetching stops, and the **bcctr** waits for the first branch to be completed.

- A branch(LK = 1) followed by a branch(LK = 1)—Fetching stops, and the second branch waits for the first branch to be completed. (**Note**: a **bl** instruction does not have to wait for a branch(LK = 1) to complete.)

- A **bc**(based-on-CR) waiting for resolution due to a CR-dependency followed by a **bc**(based-on-CR)—Fetching stops, and the second branch waits for the first CR-dependency to be resolved. (**Note**: branch conditions can be a function of the CTR and the CR; if the CTR condition is sufficient to resolve the branch, then a CR-dependency is ignored.)

6.4.1.2 Static Branch Prediction

Static branch prediction is a mechanism by which software (for example, compilers) can give a hint to the machine hardware about the direction the branch is likely to take. When a branch instruction encounters a data dependency, the BPU waits for the required condition code to become available. Rather than stalling instruction issue until the source operand is ready, the 602 predicts the path a branch instruction is likely to take, and instructions are fetched and executed along that path. When the branch operand becomes available, the branch is evaluated. If the predicted path was correct, program flow continues along that path uninterrupted; otherwise, the processor backs up, and program flow resumes along the correct path.

There is a scenario where a flow-control instruction is not predicted on the 602. If the target address of the branch (link or count register) is modified by an instruction that appears before the branch instruction, the BPU must wait until the target address is available.

The 602 executes through one level of prediction. The microprocessor may not predict a branch if a prior branch instruction is unresolved.

The number of instructions that can be executed after the issue of a predicted branch instruction is limited by the fact that like other out-of-order instructions, instructions in unresolved branch cannot update the register files or memory. That is, instructions can execute and make their results available for execution by subsequent instructions, they

cannot reach the complete/write-back stage in the completion unit and they instead stall in the completion unit. As a result, the completion queue may become full, at which point subsequent instructions can no longer be dispatched until the branch is resolved.

In the case of a misprediction, the 602 can redirect its machine state rather painlessly because the programing model has not been updated. When a branch is found to be mispredicted, instructions associated with the mispredicted branch are flushed and their results flushed from the rename registers. The architected register state is not affected by out-of-order execution.

6.4.1.2.1 Predicted Branch Timing Examples

Figure 6-6 depicts the cases where branch instructions are predicted, and shows both "taken" and "not taken" branch outcomes.
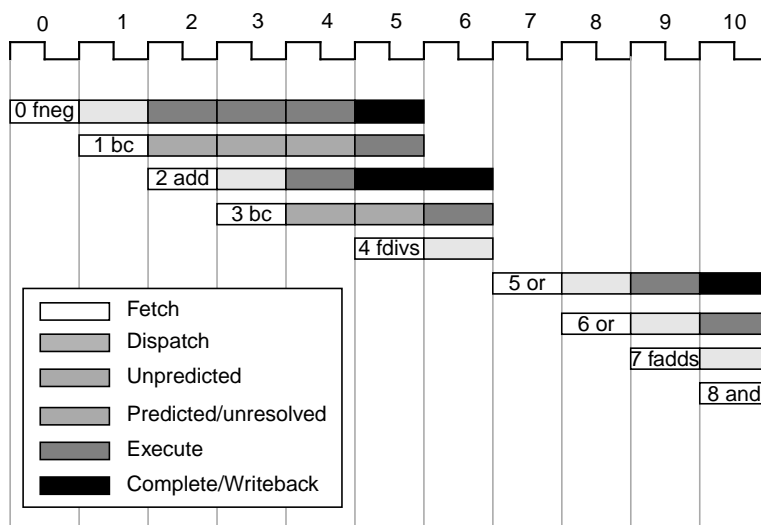
Figure 6-6. Branch Instruction Timing

1. During clock cycle 0, instruction 0 (an **fneg** instruction) is fetched.

2. In clock cycle 1, instruction 1, a Branch Conditional (**bc**) instruction whose resolution depends on how instruction 0 affects the CR is fetched as instruction 0 is dispatched to the FPU execution unit. Notice that the BPU has a combined decode/execute stage, thus the branch (instruction 1) is predicted as "not taken" during clock cycle 2 because its source register (condition register) will not become available until clock cycle 5.

3. In clock cycle 2, the **bc** instruction remains predicted but unable to resolve waiting for the results from instruction 0. An **add** instruction (instruction 2) is fetched— sequentially according to the prediction of instruction 1.

4. During clock cycle 3, instruction 0 is in FPU execute stage 2, and instruction 1 remains unresolved, waiting for the results of instruction 0. Also, the **add** instruction is dispatched, and a second **bc** instruction (instruction 3) is fetched.

5. During clock cycle 4, instruction 0 enters its final execute stage, instruction 1 remains unresolved, instruction 2 is in the IU execute stage, and the second **bc** instruction remains unable to predict waiting the resolution of the first **bc** instruction. Therefore, no instruction is fetched.

6. In clock cycle 5, instruction 0 enters the complete stage, which resolves the data dependency for the first branch (instruction 1); in this case the prediction was correct. When instruction 1 is resolved, the BPU is free to predict **bc** (instruction 3), and it too is predicted as "not taken". This branch depends on how instruction 2 affects the CR and therefore must wait for it to complete and write back its results. Although instruction 2 had executed, it cannot exit the complete stage because only one instruction can complete per clock cycle. Also in this clock cycle, an **fdivs** instruction (instruction 4) is fetched.

7. During clock 6, instruction 2 writes back its results so the second branch instruction is resolved, and the prediction was incorrect. As a result, the **fdivs** instruction, which is now in dispatch stage must be flushed from the pipeline and fetching must begin at the target address.

8. In clock cycle 7, the instruction fetcher begins fetching instructions from the instruction cache. In this case there is a cache hit, so the target **or** instruction is fetched, and the instruction flow continues.

6.4.2 Integer Unit Execution Timing

The integer unit executes all integer and logical instructions. Many of these instructions execute in a single clock cycle. The integer unit has one execute phase in its pipeline, thus when a multicycle integer instruction is being executed, no other integer instructions may begin an execute phase. Refer to Table 6-2 for integer instruction execution timing.

6.4.3 Floating-Point Unit

The 602's FPU performs single-precision floating-point operations compliant with the IEEE-754 floating-point standard, and can produce non-IEEE results for time-critical operations. These modes of operation are described in Section 2.3.4.2.2, "IEEE Mode (FPSCR[NI] = 0)," and Section 2.3.4.2.3, "Non-IEEE Mode (FPSCR[NI] = 1)."

Single-precision multiplies, multiply-adds, adds, and subtracts execute in a three-stage pipeline with three-cycle completion latency allowing throughput of one single-precision instruction per cycle. Single-precision divide operations require multiple cycles to complete.

All operations involving double-precision operands and operations that produce denormalized numbers require emulation routines, and therefore have longer execution latency.

There are two 32-bit, special-purpose registers—SP and LT. Each bit of each register corresponds to a single 32-bit FPR.

- If either the SP bit or the LT bit is set, the associated register contains valid data— SP designating single-precision floating-point data, and LT designating integer data.

- If neither bit is set, the data resides in memory in the associated double-precision emulated FPR. The operation of these registers is described in greater detail in Section 2.1.2.4.1, "Floating-Point Tag Registers (SP and LT)."

The block diagram for the floating-point execution unit is shown in Figure 6-7.
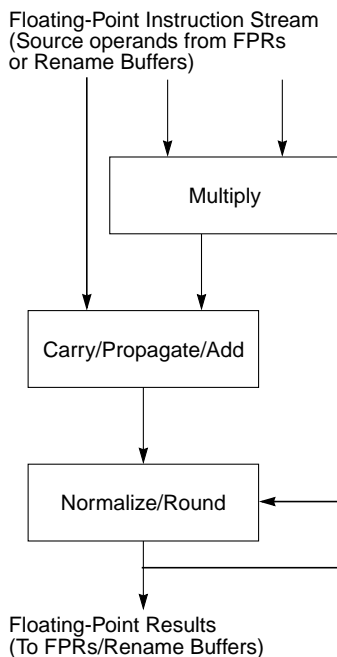
Figure 6-7. FPU Block Diagram

6.4.4 Floating-Point Unit Execution Timing

The FPU on the 602 executes all hardware-supported 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 (one at each of the three stages in the FPU pipeline). While most floating-point instructions execute with three- or four-cycle latency, and one-cycle throughput, two instructions (**fdivs** and **fres**) execute with latencies of 18 cycles. The **fdivs**, **fres**, **mtfsb0**, **mtfsb1**, **mtfsfi**, **mffs**, and **mtfsf** instructions block the FPU pipeline until they complete execution, and thereby inhibit the dispatch 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 updated. Refer to Table 6-5 for floating-point instruction execution timing.

6.4.5 Load/Store Unit Execution Timing

The LSU has two pipeline stages—the first is for effective address calculation and MMU translation, and the second is for accessing the data in the cache. The execution of most load and store instructions is pipelined, as follows:

- Load instructions have a two-cycle latency and one-cycle throughput. Load instructions block subsequent access to the cache until the critical word is forwarded.

- Store instructions have a two-cycle latency and two-cycle throughput.

Refer to Table 6-6 for load and store instruction execution timing.

6.5 Memory Performance Considerations

Due to the 602's instruction execution throughput of three instructions per clock cycle, lack of data bandwidth can become a performance bottleneck. For the 602 to approach its potential performance levels, it must be able to read and write data quickly and efficiently. In a multiprocessor 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 avoid such contention, the PowerPC architecture defines three memory update modes—copy-back, write-through, and caching-inhibited. Each page of memory is specified to be in one of the following modes:

- If a page or block is in copy-back mode, data being stored to that page is written only to the on-chip cache.

- If a page or block is in write-through mode, write operations to that page update the on-chip cache on hits and always update main memory.

- If a page or block is caching-inhibited, data in that page is never stored in the on-chip cache.

This section describes how performance is affected by each memory update mode. For details about the operation of the on-chip cache and the memory update modes, see Chapter 3, "Instruction and Data Cache Operation."

6.5.1 Copy-Back Mode

When storing data while in copy-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 cache block 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, copy-back mode may be preferred when external bus bandwidth is a potential bottleneck—for example, in a multiprocessor environment. Copy-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 or block that is in copy-back mode, snooping must be enabled to allow copy-back operations and cache invalidations of modified data. The 602 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 602'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 602 invalidates the cache block. The other device is then free to attempt an access to the updated memory address. See Chapter 3, "Instruction and Data Cache Operation," for complete information about bus snooping. For an understanding of how bus latency is affected by snooping, as well as by the operations that occur as a result of a snoop hit, see Section 8.4, "Memory Coherency and Bus Protocol."

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

6.5.2 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 line on a cache miss is undesirable. Automatic copy-back of cached data is not performed 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 is busy for the extra clock cycles required to perform the memory update; therefore, load operations that miss the on-chip cache must wait while the external store operation completes.

6.5.3 Caching-Inhibited Accesses

If a memory page is specified to be caching-inhibited, data from this page is not stored in the on-chip cache.

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

In summary, the copy-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 caching-inhibited mode causes memory access for both loads and stores.

6.6 Instruction Scheduling Guidelines

Instruction scheduling on the 602 can be improved by observing the following guidelines:

- Implement good static branch prediction (setting of y bit in BO field)
- When branch prediction is either uncertain or an even probability, predict that the branch is not taken.
- To reduce mispredictions, separate the instruction that sets CR bits from the branch instruction that evaluates them; separation by more than nine instructions ensures that the CR bits are immediately available for evaluation.
- When branching conditionally to a location specified by the count or link register, or when branching conditionally based on the value in the count register, separate the **mtspr** instruction that initializes the CTR or LR from the branch instruction performing the evaluation. Separation of the branch instruction and the **mtspr** instruction by more than nine instructions ensures the register values are immediately available for use by the branch instruction.
- Schedule instructions to minimize execution-unit-busy stalls.
- Avoid using serializing instructions.
- Schedule instructions to avoid dispatch stalls due to renamed resource limitations by observing the following guidelines:
  — Only four instructions can be in the complete/writeback stage at any one time
  — Schedule code to avoid a need for more than four GPR or FPR rename buffers at a time, so even though space may be available in the first FPU execute stage, a floating-point instruction cannot enter the execute stage if all four FPR rename registers are occupied.

6.6.1 Branch, Dispatch, and Completion Unit Resource Requirements

This section describes the specific resources required to avoid stalls during branch resolution, instruction dispatching, and instruction completion.

6.6.1.1 Branch Resolution Resource Requirements

The following is a list of branch instructions and the resources required to avoid stalling the fetch unit in the course of branch resolution:

- The **bclr** instruction requires LR availability.
- The **bcctr** instruction requires CTR availability.
- "Branch and link" instructions require shadow LR availability.
- The "branch conditional on counter decrement and CR condition" requires CTR availability or the CR condition must be false, and 602 cannot be executing instructions out-of order when the branch instruction is encountered by the BPU.
- The "branch conditional on CR condition" cannot be executed out of order.

6.6.1.2 Dispatch Unit Resource Requirements

The following is a list of resources required to avoid stalls in the dispatch unit:

- Needed execution unit is available.
- Needed GPR rename register(s) are available.
- Needed FPR rename registers are available.
- Completion buffer is not full.
- Instruction is dispatch-serialized and completion buffer is empty.
- A dispatch-serialized instruction is not currently being executed.

6.6.1.3 Completion Unit Resource Requirements

The following is a list of resources required to avoid stalls in the completion unit—CQ[0] is the completion buffer located at the end of the completion queue:

- Instruction in CQ[0] must be finished
- Instruction in CQ[0] must not cause an exception

6.7 Instruction Serialization Modes

This section describes the operation of the 602's serialization modes, which are as follows:

- Completion serialization
- Dispatch serialization
- Refetch serialization
- FPU serialization

6.7.1 Completion Serialization

A completion-serialized instruction is held for execution in its functional unit until all prior instructions have completed. The instruction executes when it is next to complete in program order.

Completion serialization is used for instructions that access or modify nonrenamed resources or directly access renamed resources. Results from these instructions are not available or forwarded for subsequent instructions until the instruction completes. Consecutive completion serialized instructions in a program limit completion to one instruction per cycle. The following instructions are completion serialized:

- Instructions that modify the CR register, excluding record operations and compare instructions
- Instructions that access or modify system control or status registers
- Instructions that manage caches, TLBs, or BAT registers
- Instructions that access nonrenamed GPRs such as load multiple instructions
- Instructions defined by the architecture as synchronizing—**sync**, **isync**, **rfi**, and **sc**
- All store instructions

6.7.2 Dispatch Serialization

Some completion-serialized instructions are also dispatch serialized. These instructions inhibit dispatching of subsequent instructions until the instruction completes. Dispatch serialization is used for instructions that access resources that can be renamed used by the dispatcher.

The following instructions are dispatch-serialized:

- **dcbf**, **dcbi**, **dcbst**, **dcbz**, and **icbi**
- **tlbld**, **tlbli**, and **tlbie**
- **lmw** and **stmw**
- **mcrfs**, **mcrxr**
- **mfspr** with SPRs—IBATn, DBATn, LT, and SP
- **mffs**, **mfsr**, and **mfsrin**
- **mtspr** with SPRs—SER, SEBR, ESASRR, XER, IBATn, and DBATn
- **mtfsb1**, **mtfsf**, **mtfsfi**, **mtmsr** (if a store operation is pending), **mtssb0**, **mtsr**, and **mtsrin**,
- **fctiwz**
- **sync** and **isync**

6.7.3 Refetch Serialization

There is one instruction, **isync**, that is refetch serializing. After an **isync** instruction completes, it forces subsequent instructions to be refetched.

6.7.4 FPU Serialization

The FPU serializes divide instructions that are not pipelined. No new instructions are accepted from the dispatch unit until FPU-serialized divide instruction finishes.

6.8 Instruction Latency Summary

Instruction latency is shown in Table 6-1 through Table 6-6.

- Pipelined floating-point instructions are shown with number of clocks in each stage separated by dashes.
- Instructions that must be emulated by software are identified by the exception that they take.

The "^" indicates certain instructions (integer instructions, floating-point compare instructions, and floating-point instructions with Rc = 1) that immediately forward their CR results to the BPU for fast branch resolution. All other instructions forward their CR results when they reach the completion stage.

Note that the BPU folds (removes) certain branch instructions from the instruction stream before the stream reaches the dispatcher. In certain cases, the BPU also predicts branches and supplies an instruction stream to the dispatcher at the target address indicated by the predicted branch. Therefore, although this section accurately indicates the number of cycles an instruction executes in the appropriate unit, determining the elapsed time or cycles to execute a sequence of instructions is difficult and beyond the scope of this document.

6.8.1 BPU Instruction Timings

Table 6-1 lists the timings for executing branch instructions. Note that these timings do not identify the latency of the actual branching, which can be affected by factors such as the accuracy of the branch prediction and memory latency.

Table 6-1. BPU Operations

Mnemonic	Primary	Extended	Cycles
bcctr[l]	19	528	1*
bclr[l]	19	016	1*
bc[l][a]	16	—	1*
b[l][a]	18	—	1*

*These operations may be folded for an effective cycle time of 0.

6.8.2 Integer Unit Instruction Timings

The IU in the 602 is responsible for executing integer computational and logical instructions. Timings for the IU instructions are listed in Table 6-2.

Table 6-2. Integer Unit Operations

Mnemonic	Primary	Extended	Cycles
addc[o][.]	31	010	1
adde[o][.]	31	138	1
adde[o][.]	31	138	1
addi	14	—	1
addic	12	—	1
addic.	13	—	1
addis	15	—	1
addme[o][.]	31	234	1
addze[o][.]	31	202	1
add[o][.]	31	266	1
andc[.]	31	060	1
andi.	28	—	1

Table 6-2. Integer Unit Operations (Continued)

Mnemonic	Primary	Extended	Cycles
andis.	29	—	1
and[.]	31	028	1
cmp	31	000	1^
cmpi	11	—	1^
cmpl	31	032	1^
cmpli	10	—	1^
cntlzw[.]	31	026	1
divwu[o][.]	31	459	37
divw[o][.]	31	491	37
dsa	31	628	1
eieio	31	854	no op
eqv[.]	31	284	1
esa	31	596	1
extsb[.]	31	954	1
extsh[.]	31	922	1
mfmsr	31	083	1
mfrom	31	265	1
mfspr (DBATs)	31	339	3&
mfspr (IBATs)	31	339	3&
mfspr (not I/DBATs)	31	339	1
mfsr	31	595	3&
mfsrin	31	659	3&
mftb	31	371	1
mtmsr	31	146	2
mtspr (IBATs)	31	467	2&
mtspr (not IBATs)	31	467	2
mtsr	31	210	3&
mtsrin	31	242	3&
mttb	31	467	1
mulhwu[.]	31	011	1-1,2-1,3-1,4-1
mulhw[.]	31	075	1-1,2-1,3-1
mulli	07	—	1,1-1

Table 6-2. Integer Unit Operations (Continued)

Mnemonic	Primary	Extended	Cycles
mullw[o][.]	31	235	1-1.2-1,3-1
nand[.]	31	476	1
neg[o][.]	31	104	1
nor[.]	31	124	1
orc[.]	31	412	1
ori	24	—	1
oris	25	—	1
or[.]	31	444	1
rlwimi[.]	20	—	1
rlwinm[.]	21	—	1
rlwnm[.]	23	—	1
slw[.]	31	024	1
srawi[.]	31	824	1
sraw[.]	31	792	1
srw[.]	31	536	1
subfc[o][.]	31	008	1
subfe[o][.]	31	136	1
subfic	08	—	1
subfme[o][.]	31	232	1
subfze[o][.]	31	200	1
subf[o][.]	31	040	1
sync	31	598	1&
tw	31	004	2
twi	03	—	2
xori	26	—	1
xoris	27	—	1
xor[.]	31	316	1

Condition register logical instructions are executed in the IU. Timings for these instructions are shown in Table 6-3.

Table 6-3. Condition Register Logical Operations

Mnemonic	Primary	Extended	Cycles
crand	19	257	1
crandc	19	129	1
creqv	19	289	1
crnand	19	225	1
crnor	19	033	1
cror	19	449	1
crorc	19	417	1
crxor	19	193	1
mcrf	19	000	1
mcrxr	31	512	1&
mfcr	31	019	1
mtcrf	31	144	1

6.8.3 Synchronization Instructions

Several instructions are not dispatched and executed in an execution unit, but rather are sent directly to the completion buffer where they are allowed to perform the appropriate operation in the correct order.

Table 6-4. Synchronization Instructions

Mnemonic	Primary	Extended	Cycles
isync	19	150	1&
rfi	19	050	3
sc	17	- -1	3

6.8.4 FPU Instruction Timings

Timings for floating-point instructions are shown in Table 6-5. Instructions with a single entry in the cycles column are not pipelined; for these instructions the FPU is not available for additional instruction execution.

Table 6-5. FPU Operations

Mnemonic	Primary	Extended	Cycles
fabs[.]	63	264	1-1-1^
fadds[.]	59	021	1-1-1^
fadd[.]	63	021	Emulation trap
fcmpo	63	032	1-1-1^
fcmpu	63	000	1-1-1^
fctiwz[.]	63	015	1-1-1^
fctiw[.]	63	014	Emulation trap
fdivs[.]	59	018	18^
fdiv[.]	63	018	Emulation trap
fmadds[.]	59	029	1-1-1^
fmadd[.]	63	029	Emulation trap
fmr[.]	63	072	1-1-1^
fmsubs[.]	59	028	1-1-1^
fmsub[.]	63	028	Emulation trap
fmuls[.]	59	025	1-1-1^
fmul[.]	63	025	Emulation trap
fnabs[.]	63	136	1-1-1^
fneg[.]	63	040	1-1-1^
fnmadds[.]	59	031	1-1-1^
fnmadd[.]	63	031	Emulation trap
fnmsubs[.]	59	030	1-1-1^
fnmsub[.]	63	030	Emulation trap
fres[.]	59	024	18^
frsp[.]	63	012	1-1-1^
frsqrte[.]	63	026	1-1-1^
fsel[.]	63	023	1-1-1^
fsubs[.]	59	020	1-1-1^
fsub[.]	63	020	Emulation trap
mcrfs	63	064	1-1-1&

Table 6-5. FPU Operations (Continued)

Mnemonic	Primary	Extended	Cycles
mffs[.]	63	583	1-1-1&^
mtfsb0[.]	63	070	1-1-1&^
mtfsb1[.]	63	038	1-1-1&^
mtfsfi[.]	63	134	1-1-1&^
mtfsf[.]	63	711	1-1-1&^

Note that all single-precision instructions take an emulation trap exception if any of the operands' SP bits are cleared (indicating that the operand is an integer or a double-precision floating-point number). The **mtfsf** instruction takes an emulation trap exception if the LT bit is cleared.

6.8.5 Load/Store Unit Instruction Timings

Pipelined load/store instructions are shown in Table 6-6 with cycles of total latency and throughput cycles separated by a colon.

Table 6-6. Load/Store Unit Instruction Timings

Mnemonic	Primary	Extended	Cycles
dcbf	31	086	2/5&
dcbi	31	470	2&
dcbst	31	054	2/5&
dcbt	31	278	1
dcbtst	31	246	1
dcbz	31	1014	5&
eciwx	31	310	Program exception
ecowx	31	438	Program exception
icbi	31	982	3&
lbz	34	—	2:1
lbzu	35	—	2:1
lbzux	31	119	2:1
lbzx	31	087	2:1
lfd	50	—	3:2
lfdu	51	—	3:2
lfdux	31	631	3:2
lfdx	31	599	3:2

Table 6-6. Load/Store Unit Instruction Timings (Continued)

Mnemonic	Primary	Extended	Cycles
lfs	48	—	2:1
lfsu	49	—	2:1
lfsux	31	567	2:1
lfsx	31	535	2:1
lha	42	—	2:1
lhau	43	—	2:1
lhaux	31	375	2:1
lhax	31	343	2:1
lhbrx	31	790	2:1
lhz	40	—	2:1
lhzu	41	—	2:1
lhzux	31	311	2:1
lhzx	31	279	2:1
lmw	46	—	1+ n&
lswi	31	597	Emulation trap
lswx	31	533	Emulation trap
lwarx	31	020	2:1
lwbrx	31	534	2:1
lwz	32	—	2:1
lwzu	33	—	2:1
lwzux	31	055	2:1
lwzx	31	023	2:1
stb	38	—	2:1
stbu	39	—	2:1
stbux	31	247	2:1
stbx	31	215	2:1
stfd	54	—	3:2
stfdu	55	—	3:2
stfdux	31	759	3:2
stfdx	31	727	3:2
stfiwx	31	983	2:1
stfs	52	—	2:1

Table 6-6. Load/Store Unit Instruction Timings (Continued)

Mnemonic	Primary	Extended	Cycles
stfsu	53	—	2:1
stfsux	31	695	2:1
stfsx	31	663	2:1
sth	44	—	2:1
sthbrx	31	918	2:1
sthu	45	—	2:1
sthux	31	439	2:1
sthx	31	407	2:1
stmw	47	—	n&
stswi	31	725	Emulation trap
stswx	31	661	Emulation trap
stw	36	—	2:1
stwbrx	31	662	2:1
stwcx.	31	150	2:1
stwu	37	—	2:1
stwux	31	183	2:1
stwx	31	151	2:1
tlbie	31	306	3&
tlbld	31	978	3&
tlbli	31	1010	3&

Note that the **lfd**, **lfdu**, **lfdx**, and **lfdux** instructions trap to the emulation trap exception (0x01600) if the operand is not within the single-precision range (regarding both the exponent and fraction) or if the operand is a NaN, an infinity, or a denormalized number. The **stfd**, **stfdu**, **stfdx**, and **stfdux** instructions trap to the emulation trap exception if the SP bit associated with the operand FPR is "OFF" or if the FPR contains a NaN, an infinity, or a denormalized number.

6.8.6 Effect of Operand Placement on Performance

The location and alignment of operands in memory affect relative performance of memory accesses, and in some cases affect it significantly. Data and instructions should be organized in memory to minimize the number of effective address calculations and minimize the number of alignment exceptions.

The following list characterizes the efficiency of memory accesses relating to operand placement:

- One effective address generated—As long as the entire memory access falls within a double word, only one effective address is generated. For optimal performance, operands should not cross double-word alignment boundaries.

- Multiple effective addresses generated—When an operand crosses a double-word boundary (for example, when a double-word operand is not double-word-aligned), an additional effective address must be generated for each double word in memory in which the operand resides. If the double-word boundary is also at a cache block boundary, a cache miss may occur.

- Alignment exception is generated—An alignment exception or other exception is generated by the memory operation. Note that any little-endian access that does not fall on its natural alignment boundary causes an alignment exception. If an access crosses a page boundary, a page miss may occur.

6.8.7 Effect of Floating-Point Exceptions on Performance

Floating-point operations that affect the exception sticky bits in the FPSCR may incur performance penalties.

When an exception is disabled in the FPSCR and MSR[FE] = 0, updates to the FPSCR exception sticky bits are completion-serialized, as described in Section 6.7.1, "Completion Serialization." This serialization may incur a one- or two-cycle execution delay. The penalty is incurred only on transitions to the exception bit and not on subsequent operations with the same exception.

When an exception is enabled in the FPSCR, the instruction causes an emulation trap exception without updating the FPSCR or the target FPR. The emulation trap exception handler is required to complete the instruction and is invoked regardless of the setting of MSR[FE].

For the fastest and most predictable floating-point performance, all exceptions should be disabled in the FPSCR and MSR.

Chapter 7
Signal Descriptions

The PowerPC 602 microprocessor bus interface has a single 64-bit bus that is time-multiplexed for use as address and data bus. This technique provides the most efficient use of signals on the processor while having most of the signals of other current 32-bit PowerPC processors. If a nonmultiplexed bus is desired, the 602 bus can also be demultiplexed at the processor with a set of address latches. The 602 bus supports cache-coherent DMA operations.

A summary of the bus features are listed as follows:

- 32-bit address and address attributes
- 64-bit bus for data transfer that can be dynamically resized for 32-bit data operations
- Snooping to support cache-coherent DMA operations
- Injected snoops allowed during a burst read transaction
- Provides line-fill read address on the address phase of the write-back transaction

This chapter describes the 602'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 TT0–TT4 (transfer type signals), are referred to as asserted when they are high and negated when they are low.

The 602 signals are grouped as follows:

- Arbitration signals—The 602 uses these signals to arbitrate for bus mastership.
- Transfer start signal—The transfer start signal indicates that a bus master has begun a transaction.
- Address transfer signals—These signals, which correspond to the data signals D0–D31 during the data phase, transfer the address during the address phase.
- Transfer attribute signals—These signals, which share the same physical connections as D31–D63, provide information about the type of transfer, such as the transfer size and whether the transaction is bursted, write-through, or caching-inhibited.
- Address transfer termination signals—These signals are used to acknowledge the end of the address phase. They also indicate whether a condition exists that requires the address phase to be repeated.
- Data transfer signals—These 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 nonburst transaction, the data termination signals also indicate the end of the transaction, while in burst accesses, the data termination signals apply to individual beats and indicate the end of the data phase only after the final data beat. They also indicate whether a condition exists that requires entire transaction to be repeated.
- System status signals—These signals include the external interrupt signal, checkstop signals, and both soft- and hard-reset signals. These signals are used to interrupt and to reset the processor.
- JTAG/COP interface signals—The JTAG interface and common on-chip processor (COP) unit provides a serial interface to the system for performing monitoring and boundary tests.
- Clock signals—These signals provide for system clock input and frequency control.

7.1 Signal Configuration

Figure 7-1 illustrates the 602's signal configuration, showing how the signals are grouped.

NOTE

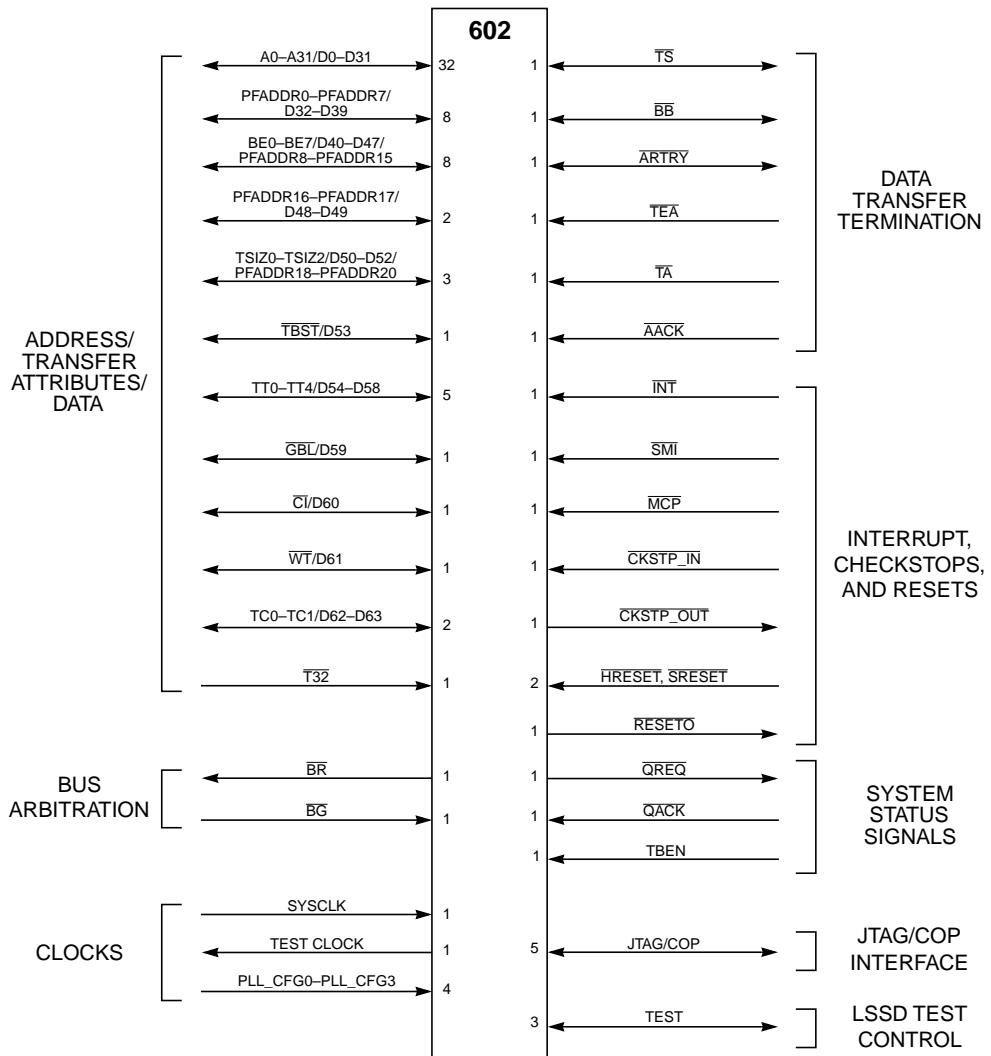A pinout showing actual pin numbers is included in the 602 hardware specifications.

Figure 7-1. PowerPC 602 Microprocessor Signal Groups

7.1.1 Time-Multiplexed System Bus

To conserve space and power, the 602 implements a time-multiplexed bus. That is, the same signals from the processor serve different functions depending on whether it is being used to transfer an address or data. Note that A0–A31 are equivalent to D0–D31. The remaining 32 bits are used for transfer attribute signals, such as global ($\overline{GBL}$) and transfer burst ($\overline{TBST}$). Signal assignments for data and address cycles are shown in Table 7-1.

Table 7-1. Time-Multiplexed Signal Assignments

Data Phase	Address Phase	
	Nonburst Transactions	**Burst Transactions**
D0–D31	Address signals (A0–A31)	Address signals (A0–A31)
D32–D39	—	Prefetch line-fill address signals (PFADDR0–PFADDR20). (Cache copy-back line-fill only)
D40–D47	Byte enable (BE0–BE7)	
D48–D49	—	
D50–D52	Transfer size (TSIZ0–TSIZ2)	
D53	Transfer burst ($\overline{TBST}$)	Transfer burst ($\overline{TBST}$)
D54–D58	Transfer type (TT0–TT4)	Transfer type (TT0–TT4)
D59	Global ($\overline{GBL}$)	Global ($\overline{GBL}$)
D60	Cache inhibit ($\overline{CI}$)	Cache inhibit ($\overline{CI}$)
D61	Write through ($\overline{WT}$)	Write through ($\overline{WT}$)
D62–D63	Transfer code (TC0–TC1)	Transfer code (TC0–TC1)

When the bus is in the data phase, it can be made to function as a 32- or 64-bit data bus. The bus width is determined by the target data bus width ($\overline{T32}$) signal, described in Section 7.2.7.2, "Target Data Bus 32 (T32)—Input."

7.2 Signal Descriptions

This section describes individual 602 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 Bus Arbitration Signals

The arbitration signals are a collection of input and output signals the 602 uses to request the bus and the system uses to control bus mastership. For a detailed description of how these signals interact, see Section 8.3.1, "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 602. Following are the state meaning and timing comments for the $\overline{\text{BR}}$ signal.

State Meaning Asserted—Indicates that the 602 is requesting mastership of the bus and is waiting for a qualified $\overline{\text{BG}}$ to begin the bus transaction. See Section 8.3.1, "Bus Arbitration."

Negated—Indicates that the 602 is not requesting the bus. The 602 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 the 602 is not parked and a bus transaction is needed.

Negation—Occurs for at least one bus clock cycle after an accepted, qualified bus grant (see $\overline{\text{BG}}$, $\overline{\text{TS}}$, $\overline{\text{ARTRY}}$, and $\overline{\text{BB}}$), 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, unless this 602 asserted $\overline{\text{ARTRY}}$ and needs to perform a snoop write back. $\overline{\text{BR}}$ may also be negated if the 602 cancels the internal bus request (need_bus condition) before receiving a qualified $\overline{\text{BG}}$.

High-impedance—Occurs during a hard reset or checkstop condition.

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

The bus grant ($\overline{\text{BG}}$) signal is an input signal on the 602. Following are the state meaning and timing comments for the $\overline{\text{BG}}$ signal.

State Meaning Asserted—Indicates that the 602 may, with the proper qualification, assume mastership of the bus. A qualified bus grant occurs when $\overline{\text{BG}}$ is asserted and $\overline{\text{BB}}$ (bus busy), $\overline{\text{TS}}$, and $\overline{\text{ARTRY}}$ are negated (indicating that no other devices currently are using the bus). Note that the $\overline{\text{BR}}$ is not part of the qualified bus grant; if the processor is parked, it does not assert the $\overline{\text{BR}}$ signal when it needs the bus. See Section 8.3.1, "Bus Arbitration."

Negated— Indicates that the 602 has not been designated to be the next potential bus master.

Timing Comments Assertion—May occur at any time to indicate the 602 is free to use the bus. If the 602 has a second transaction pending, (except for cache copy-back line-fill operations) it does not look for $\overline{\text{BG}}$ to be asserted until the clock cycle after $\overline{\text{BB}}$ is negated. If the current transaction is a copy-back line-fill, the 602 samples the $\overline{\text{BG}}$ signal during the last data beat.

Negation—May occur at any time to indicate to the 602 that it cannot use the bus. The 602 can 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 602 that it was free to take mastership (if qualified).

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

The transfer start signal, $\overline{\text{TS}}$, indicates that an address phase of a transfer has begun. For detailed information about how the $\overline{\text{TS}}$ signal interacts with other signals, refer to Section 8.3.2, "Address Transfer Subphase."

The $\overline{\text{TS}}$ signal is both an input and an output signal on the 602.

7.2.2.1 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 602 is the current bus owner and is in the address phrase has begun a bus transaction and that the address and transfer attribute signals are valid.
	The 602 may not assume bus mastership if the bus request is cancelled internally by the cycle a qualified bus grant would have been recognized.
	Negated—Indicates that the 602 is not in the address phase
Timing Comments	Assertion— Occurs in the cycle after a qualified bus grant is accepted by the 602 and remains asserted for the duration of the address phase.
	Negation/High Impedance (turn-off sequencing)—Negates for one bus clock, then goes to the high impedance state.

7.2.2.2 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 and transfer attribute signals are valid for snooping (see $\overline{\text{GBL}}$).
	Negated—Indicates that no bus transaction is occurring and that the bus may be available for use by the 602.
Timing Comments	Assertion—May occur whenever the 602 must be prevented from using the bus.
	Negation—May occur whenever the 602 is permitted access to the bus. Must occur one bus clock cycle after $\overline{\text{TS}}$ is asserted.

7.2.3 Address Transfer Signals

The address transfer signals are used to transmit the address for the address transfer. For a detailed description of how these signals interact, refer to Section 8.3.2, "Address Transfer Subphase."

7.2.3.1 Address Signals (A0–A31)

The address signals (A0–A31) are both input and output signals. Because the bus is time-multiplexed, the physical connections serve dual purposes. The additional function these signals serve are identified here and described elsewhere in this chapter.

7.2.3.1.1 Address Signals (A0–A31)—Output

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 read operations, the bus presents the double-word-aligned address containing the critical double word of the cache block that missed the cache.
	On burst write operations, the bus presents the first double word of the cache block.
	Note that the address output during burst operations is not incremented. See Section 8.3.2, "Address Transfer Subphase."
Timing Comments	Assertion/Negation—Driven valid on the same cycle that $\overline{\text{TS}}$ is asserted and remains valid for the duration of the address phase.
	High Impedance—Occurs the cycle following the assertion of $\overline{\text{AACK}}$. No precharge action is performed on release.
Alternate Use	Data output signals (D0–D31). See Section 7.2.7.1.1, "Data Signals (D0–D63)—Output."

7.2.3.1.2 Address Signals (A0–A31)—Input

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 $\overline{\text{TS}}$; the 602 samples the address signals only on this cycle.
Alternate Use	Data signals (D0–D31). See Section 7.2.7.1.2, "Data Signals (D0–D63)—Input."

7.2.3.1.3 Prefetch Line-Fill Address (PFADDR0–PFADDR20)—Output

The prefetch line-fill address signals, PFADDR0–PFADDR20, help expedite a cache-line-fill when room in the cache must be made when a read misses in the cache and modified data in the least-recently used cache block is cast-out and written back to external memory. The prefetch line-fill address signals specify the cache-line-fill read address.

When the 602 misses in its cache and must write modified data to external memory, it outputs the upper 21 bits (A0–A20) of the missing read address on the address phase of the write-back transaction. The low-order 6 address bits (A21–A26) of the read address match the low-order 6 bits of the write-back address because the index into the internal cache is the same for both the line-fill read address and the write-back address. This read address ensures that a system's memory controller can prefetch the read data, making it available immediately when the 602 performs a line-fill operation.

Following are the state meaning and timing comments for the PFADDR0–PFADDR20 output signals.

State Meaning   Asserted/Negated—This address is valid only when TC0 is negated, TC1 is asserted, and $\overline{\text{TBST}}$ is asserted.

Timing Comments   Assertion/Negation—Driven valid on the same cycle that $\overline{\text{TS}}$ is asserted and remains driven/valid for the duration of the address phase.

Alternate Use   The alternate uses for PFADDR0–PFADDR20 are shown in Table 7-2.

Table 7-2. Alternate Uses for PFADDR0–PFADDR20

PFADDR Signal	Alternate Use
PFADDR0–PFADDR7	Data output (D32–D39). See Section 7.2.7.1.1, "Data Signals (D0–D63)—Output."
PFADDR8–PFADDR15	Byte enable (BE0–BE7). See Section 7.2.4.3, "Byte Enable (BE0–BE7)." Data output (D40–D47). See Section 7.2.7.1.1, "Data Signals (D0–D63)—Output."
PFADDR16–PFADDR17	Data output (D48–D49). See Section 7.2.7.1.1, "Data Signals (D0–D63)—Output."
PFADDR18–PFADDR20	Transfer size 0–2 (TSIZ0–TSIZ2). See Section 7.2.4.2, "Transfer Size (TSIZ0–TSIZ2)—Output," Data output (D50–D52). See Section 7.2.7.1.1, "Data Signals (D0–D63)—Output."

7.2.4 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 nonburst transfer. For a detailed description of how these signals interact, see Section 8.3.2, "Address Transfer Subphase."

7.2.4.1 Transfer Type (TT0–TT4)

The transfer type (TT0–TT4) signals consist of five input/output signals on the 602. For a complete description of TT0–TT4 signals and for transfer type encodings, see Table 7-3.

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 602.

State Meaning   Asserted/Negated—Indicates the type of transfer in progress; see Table 7-3.

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

Alternate Use   The $\overline{TT0}$–$\overline{TT4}$ signals are also used as D54–D58. See Section 7.2.7.1, "Data Signals (D0–D63)."

7.2.4.1.2 Transfer Type (TT0–TT4)—Input

Following are the state meaning and timing comments for the TT0–TT3 input signals on the 602.

State Meaning   Asserted/Negated—Indicates the type of transfer in progress; see Table 7-3.

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

Alternate Use   The $\overline{TT0}$–$\overline{TT4}$ signals are also used as D54–D58; see Section 7.2.7.1, "Data Signals (D0–D63)."

Table 7-3 describes the transfer-type encodings.

Table 7-3. TT0–TT4 Encodings

TT0–TT4	60x Bus Specification		602 as Master		602 as Snooper
	Command	Transaction	Bus Transaction	Source of Transaction	Action on Hit
00000	Clean block	Address only	n/a	n/a	Clean
00100	Flush block	Address only	n/a	n/a	Flush
01000	SYNC	Address only	n/a	n/a	n/a
01100	Kill block	Address only	Address only	**dcbz**	Kill
10000	EIEIO	Address only	n/a	n/a	n/a
10100	Graphics write (**ecowx**)	Nonburst write*	n/a	n/a	n/a
11000	TLB invalidate	Address only	n/a	n/a	n/a
11100	Graphics read (**eciwx**)	Nonburst read*	n/a	n/a	n/a
00001	**lwarx** reservation set	Address only	n/a	n/a	n/a
00101	**stwcx.** reservation clear	Address only	n/a	n/a	n/a
01001	TLBSYNC	Address only	n/a	n/a	n/a
01101	ICBI	Address only	n/a	n/a	n/a
1XX01	Reserved	—	n/a	n/a	n/a

Table 7-3. TT0–TT4 Encodings (Continued)

TT0–TT4	60x Bus Specification		602 as Master		602 as Snooper
	Command	Transaction	Bus Transaction	Source of Transaction	Action on Hit
00010	Write-with-flush	Nonburst write* or burst	Nonburst write*	Caching-inhibited or write-through store	Flush
00110	Write-with-kill	Burst	Burst (not global)	Castout or snoop copyback	Kill
01010	Read	Nonburst read* or burst	Nonburst read*	Caching-inhibited load	Clean or flush
01110	Read-with-intent-to-modify	Burst	Burst	Load/store miss	Flush
10010	Write-with-flush-atomic	Nonburst write*	Nonburst write*	**stwcx**	Flush
10110	(Reserved)	n/a	n/a	n/a	n/a
11010	Read-atomic	Nonburst read* or burst	Nonburst read*	**lwarx** (caching-inhibited load)	Clean or flush
11110	Read-with-intent-to-modify-atomic	Burst	Burst	**lwarx** (load miss)	Flush
00X11	(Reserved)	—	n/a	n/a	n/a
01011	Read-with-no-intent-to-modify	Nonburst read* or burst	n/a	n/a	Clean
01111	(Reserved)	—	n/a	n/a	n/a
1XX11	(Reserved)	—	n/a	n/a	n/a

* Note that these transactions take two beats if the bus is operating in 32-bit data mode.

7.2.4.2 Transfer Size (TSIZ0–TSIZ2)—Output

The transfer size (TSIZ0–TSIZ2) signals consist of three output signals on the 602. Following are the state meaning and timing comments for the TSIZ0–TSIZ2 output signals on the 602.

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 7-4.

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

Table 7-4. Data Transfer Size

$\overline{\text{TBST}}$	TSIZ0–TSIZ2	Transfer Size	Comments
Negated	001	1 byte	Byte
Negated	010	2 bytes	Half word
Negated	011	3 bytes	—
Negated	100	4 bytes	Word
Negated	101	5 bytes	—
Negated	110	6 bytes	—
Negated	111	7 bytes	—
Negated	000	8 bytes	Double word (bus width in 64-bit mode)
Asserted	Invalid	32 bytes	Four double word (four data beats in 64-bit mode)

Alternate Use   The TSIZ0–TSIZ2 signals are also used as D50–D52 and PFADDR18–PFADDR20. For more information, see Section 7.2.7.1, "Data Signals (D0–D63)," and Section 7.2.3.1.3, "Prefetch Line-Fill Address (PFADDR0–PFADDR20)—Output."

7.2.4.3 Byte Enable (BE0–BE7)

Following are the state meaning and timing comments for the eight input/output byte enable (BE0–BE7) signals.

State Meaning   Asserted/Negated— Indicates which data bytes are valid during a nonburst operation. For burst operation, all the byte enables are invalid. If $\overline{\text{BE}}$ is used, the A29–A31 and TSIZ0–TSIZ2 signals can be ignored for nonburst operations.

Figure 7-2 shows how the byte lanes correspond to the individual byte enable signals. The use of the byte enable signals can simplify the system design because the transfer size and A29–A31 signals are not needed.

0	7 8	15 16	23 24	31 32	39 40	47 48	55 56	63
Byte lane 0	Byte lane 1	Byte lane 2	Byte lane 3	Byte lane 4	Byte lane 5	Byte lane 6	Byte lane 7
BE0	BE1	BE2	BE3	BE4	BE5	BE6	BE7
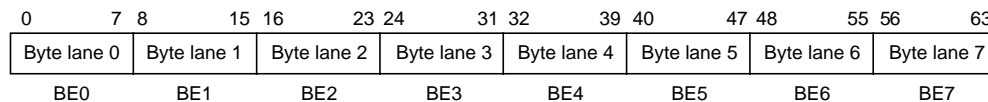
Figure 7-2. Address Format/Data Format Using Byte Enable Signals

For more information, see Section 8.3.2.2.3, "Alignment."

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

Alternate Use   The BE0–BE7 signals are also used as PFADDR8–PFADDR15 and for the data signals D40–D47. See Section 7.2.3.1.3, "Prefetch Line-Fill Address (PFADDR0–PFADDR20)—Output," and Section 7.2.7.1.1, "Data Signals (D0–D63)—Output."

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

The transfer burst ($\overline{\text{TBST}}$) signal is an input/output signal on the 602.

7.2.4.4.1 Transfer Burst ($\overline{\text{TBST}}$)—Output

Following are the state meaning and timing comments for the $\overline{\text{TBST}}$ output signal. Table 7-4 shows how the $\overline{\text{TBST}}$ signal is used with the TSIZ signals to determine the transfer size.

State Meaning   Asserted—Indicates that a burst transfer is in progress. For more information, see Section 7.2.4.2, "Transfer Size (TSIZ0–TSIZ2)—Output." See Table 7-4.

                      Negated—Indicates that a burst transfer is not in progress.

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

Alternate Use   The $\overline{\text{TBST}}$ signal is also used as D53. See Section 7.2.7.1, "Data Signals (D0–D63)."

7.2.4.4.2 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—Used when snooping for nonburst reads (read with no intent to cache). See Table 7-4.

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

Alternate Use   The $\overline{\text{TBST}}$ signal is also used as D53. See Section 7.2.7.1, "Data Signals (D0–D63)."

7.2.4.5 Transfer Code (TC0–TC1)—Output

The transfer code (TC0–TC1) consists of two output signals on the 602. Following are the state meaning and timing comments for the TC0–TC1 signals.

State Meaning   Asserted/Negated—Represents a special encoding for the transfer in progress (see Table 7-5).

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

Alternate Use   The TC0–TC1 signals are also used as D62–D63. See Section 7.2.7.1, "Data Signals (D0–D63)."

Table 7-5. Encodings for TC0–TC1 Signals

TC0–TC1	Read	Write
00	Data transaction	Normal write
01	N/A	Copy-back line-fill
10	Instruction fetch	N/A
11	Reserved	Reserved

7.2.4.6 Cache Inhibit ($\overline{\text{CI}}$)—Output

The cache inhibit ($\overline{\text{CI}}$) signal is an output signal on the 602. Following are the state meaning and timing comments for the $\overline{\text{CI}}$ signal.

State Meaning   Asserted—Indicates that a nonburst 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 602 data cache.

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

Alternate Use   The $\overline{\text{CI}}$ signal is also used as D60. See Section 7.2.7.1, "Data Signals (D0–D63)."

7.2.4.7 Write-Through ($\overline{\text{WT}}$)—Output

The write-through ($\overline{\text{WT}}$) signal is an output signal on the 602. Following are the state meaning and timing comments for the $\overline{\text{WT}}$ signal.

State Meaning   Asserted—Indicates that a nonburst 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/High Impedance—The same as A0–A31.

Alternate Use   The $\overline{\text{WT}}$ signal is also used as D61. See Section 7.2.7.1, "Data Signals (D0–D63)."

7.2.4.8 Global ($\overline{\text{GBL}}$)

The global ($\overline{\text{GBL}}$) signal is an input/output signal on the 602.

7.2.4.8.1 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/High Impedance—The same as A0–A31.

Alternate Use The $\overline{\text{GBL}}$ signal is also used as D59. See Section 7.2.7.1, "Data Signals (D0–D63)."

7.2.4.8.2 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 must be snooped by the 602.

Negated—Indicates that a transaction is not snooped by the 602

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

Alternate Use The $\overline{\text{GBL}}$ signal is also used as D59. See Section 7.2.7.1, "Data Signals (D0–D63)."

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 8.3.2.3, "Address Phase 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 602. A slave device uses this signal to indicate when the address phase has completed. The 602 allows the $\overline{\text{AACK}}$ signal to be asserted in the first clock cycle as the assertion of $\overline{\text{TS}}$ (the same clock cycle that the address is made available on the bus). This is referred to as a single-cycle address phase, and is discussed in Section 8.3, "Address Bus Phase."

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

State Meaning Asserted—Indicates that the address phase is complete.

Negated—When $\overline{\text{TS}}$ is asserted, negating $\overline{\text{AACK}}$ indicates that the address phase cannot terminate—the 602 continues to drive the address and transfer attribute signals.

Timing Comments Assertion—May occur as early as the assertion of $\overline{\text{TS}}$ (single-cycle address phase). The target device can delay asserting $\overline{\text{AACK}}$ to allow adequate address access time; for example, to support slow snooping devices. $\overline{\text{AACK}}$ is asserted for only one clock cycle.

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 602.

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 602 detects a condition in which a transaction must be retried. If the 602 needs to update memory as a result of the snoop that caused the retry, it asserts $\overline{\text{BR}}$ the cycle after the $\overline{\text{ARTRY}}$ is asserted.

High Impedance—Indicates that the 602 does not need to retry the transaction.

Timing Comments Assertion—(Single-cycle address phase). The $\overline{\text{ARTRY}}$ signal is asserted the second clock cycle after $\overline{\text{TS}}$ is asserted if a retry is required. This relationship is shown in Figure 7-3.

(Multicycle address phase). When the address phase lasts longer than one cycle, $\overline{\text{ARTRY}}$ can be asserted on the clock cycle after $\overline{\text{TS}}$ is asserted as shown in Figure 7-4.

Negation—(Single-cycle address phase). As shown in Figure 7-3, occurs the second bus cycle after the assertion of $\overline{\text{AACK}}$. Since this signal may be simultaneously driven by multiple devices, it negates in a unique fashion.

(Multicycle address phase). As shown in Figure 7-4, occurs the second bus cycle after the assertion of $\overline{\text{AACK}}$. Since this signal may be simultaneously driven by multiple devices, it negates in a unique fashion.
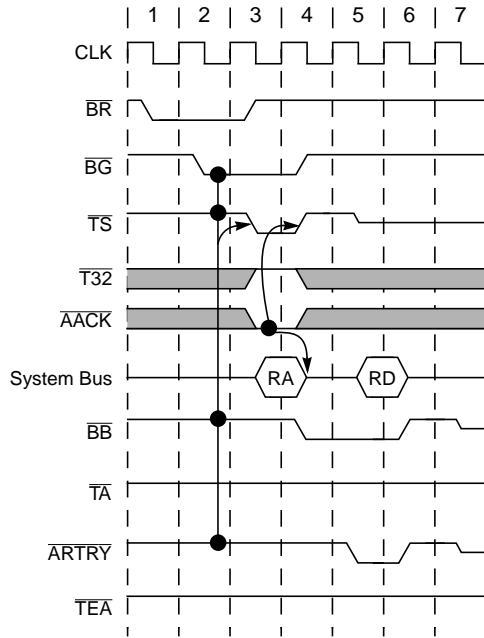
Figure 7-3. $\overline{\text{ARTRY}}$ During Other Master Read—Single-Cycle Address Phase
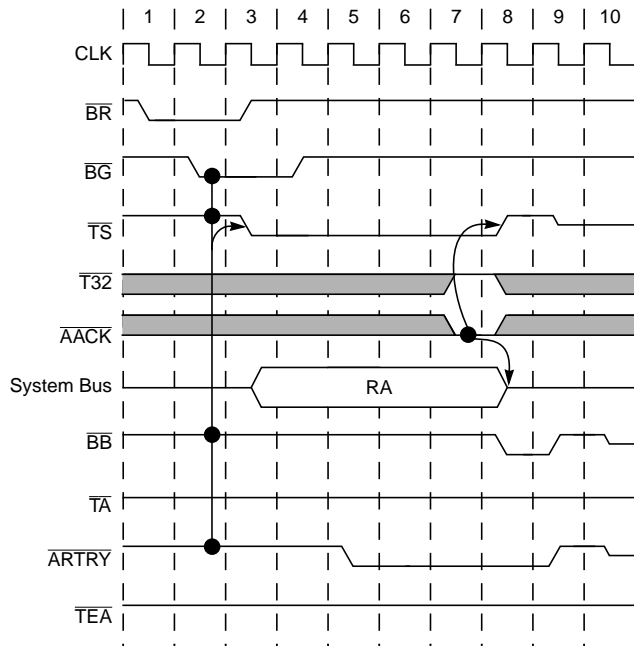
Figure 7-4. $\overline{\text{ARTRY}}$ During Other Master Read Transaction—Multicycle Address Phase

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 602 is the bus master, $\overline{\text{ARTRY}}$ indicates that the 602 must retry the current transaction and immediately negate $\overline{\text{BR}}$ (if asserted). For single-cycle address phases, $\overline{\text{ARTRY}}$ is sampled on the second clock cycle after $\overline{\text{TS}}$ is negated, as shown in Figure 7-5.

For multicycle address phases, the $\overline{\text{ARTRY}}$ signal can be asserted on the clock cycle after the assertion of $\overline{\text{TS}}$, as shown in Figure 7-6.

Negated/High Impedance—Indicates that the 602 does not need to retry the transaction.
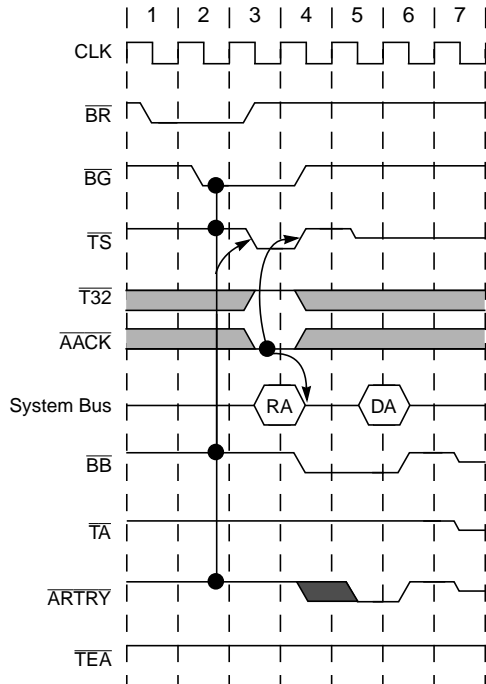
Figure 7-5. $\overline{\text{ARTRY}}$ During Read Transaction—Single-Cycle Address Phase

Timing Comments   Assertion—May occur as early as the clock cycle following the assertion of $\overline{\text{TS}}$. As shown in Figure 7-5, for single-cycle address phases, assertion must occur by the second bus clock cycle after the assertion of $\overline{\text{TS}}$ if the transaction must be retried. As shown in Figure 7-6, for multicycle address phases, assertion must occur by the bus clock cycle immediately after $\overline{\text{TS}}$ is negated.

Negation—In both single-cycle and multicycle address phases, $\overline{\text{ARTRY}}$ is negated in the clock cycle after it is sampled.
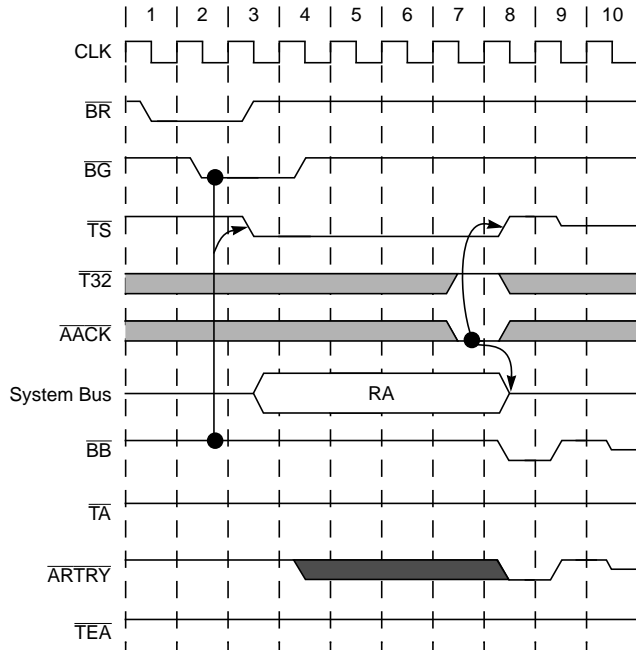
Figure 7-6. $\overline{\text{ARTRY}}$ During PowerPC 602 Processor Read Transaction—Multicycle Address Phase

7.2.6 Data Phase Signal

The $\overline{\text{BB}}$ (bus busy) signal indicates that the 602 is in the data phase.

7.2.6.1 Bus Busy ($\overline{\text{BB}}$)

The bus busy ($\overline{\text{BB}}$) signal is both an input and output signal on the 602.

7.2.6.1.1 Bus Busy ($\overline{\text{BB}}$)—Output

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

State Meaning   Asserted—Indicates that the 602 is the bus master and that the transaction is in the data phase.

    Negated—If the 602 is the bus master, indicates that it is not in the data phase; negated when the 602 is not bus master.

Timing Comments   Assertion—Occurs during the bus clock cycle following a qualified $\overline{\text{AACK}}$ and remains asserted for duration of the data phase.

    Negation—Is negated synchronously with the last beat of a data transaction.

    High Impedance—Occurs after $\overline{\text{BB}}$ is negated.

7.2.6.1.2 Bus Busy ($\overline{\text{BB}}$)—Input

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

State Meaning   Asserted—Indicates that another device is bus master (used to determine a qualified bus grant).

                                          Negated—Indicates that the bus is free for use by the 602. Must be sampled as negated to achieve a qualified bus grant.

Timing Comments   Assertion—Required when the 602 must be prevented from using the bus.

                                          Negation—May occur whenever the bus is available.

7.2.7 Data Transfer Signals

Like the address transfer signals, the data transfer signals are used to transmit data between the processor and system memory. For a detailed description of how the data transfer signals interact, see Section 8.3.3.1, "Data Transfer."

7.2.7.1 Data Signals (D0–D63)

The data signals (D0–D63) are both input and output signals on the 602. Following are the state meaning and timing comments for the data signals.

State Meaning   See Table 7-6 for the data byte lane assignments. Note that when the bus is operating in 32-bit mode, lanes 0–3 are used.

Timing Comments   When the bus is in 64-bit mode, the bus is driven once for noncached transactions and four times for cache transactions (bursts). When the bus is in 32-bit mode, 32-bit transfers require a single beat, transfers of 33–64 bits take two beats, and cache block transfers take eight beats. For more information, see Section 7.2.7.2, "Target Data Bus 32 (T32)—Input."

Table 7-6. Data Lane Assignments

Data Signals	Byte Lane
D0–D7	0
D8–D15	1
D16–D23	2
D24–D31	3
D32–D39	4
D40–D47	5
D48–D55	6
D56–D63	7

Alternate Use   Table 7-7 lists the alternate assignments for D0–D63.

Table 7-7. Alternate Uses of the Data Signals (D0–D63)

Data Signals	Alternate Use
D0–D31	Address signals (A0–A31); see Section 7.2.3.1, "Address Signals (A0–A31)."
D32–D39	Prefetch line-fill address signals (PFADDR0–PFADDR7); see Section 7.2.3.1.3, "Prefetch Line-Fill Address (PFADDR0–PFADDR20)—Output."
D40–D47	Prefetch line-fill address signals (PFADDR8–PFADDR15); see Section 7.2.3.1.3, "Prefetch Line-Fill Address (PFADDR0–PFADDR20)—Output." Byte enable signals (BE0–BE7); see Section 7.2.4.3, "Byte Enable (BE0–BE7)."
D48–D49	Prefetch line-fill address signals (PFADDR16–PFADDR17); see Section 7.2.3.1.3, "Prefetch Line-Fill Address (PFADDR0–PFADDR20)—Output."
D50–D52	Prefetch line-fill address signals (PFADDR18–PFADDR20); see Section 7.2.3.1.3, "Prefetch Line-Fill Address (PFADDR0–PFADDR20)—Output." Transfer size (TSIZ0–TSIZ2); see Section 7.2.4.2, "Transfer Size (TSIZ0–TSIZ2)—Output."
D53	Transfer burst ($\overline{\text{TBST}}$); see Section 7.2.4.4, "Transfer Burst (TBST)."
D54–D58	Transfer type signals (TT0–TT4); see Section 7.2.4.1, "Transfer Type (TT0–TT4)."
D59	Global ($\overline{\text{GBL}}$); see Section 7.2.4.8, "Global (GBL)."
D60	Cache inhibit ($\overline{\text{CI}}$); see Section 7.2.4.6, "Cache Inhibit (CI)—Output."
D61	Write through ($\overline{\text{WT}}$); see Section 7.2.4.7, "Write-Through (WT)—Output."
D62–D63	Transfer code (TC0–TC1); see Section 7.2.4.5, "Transfer Code (TC0–TC1)—Output."

7.2.7.1.1 Data Signals (D0–D63)—Output

Following are the state meaning and timing comments for the data output signals.

State Meaning   Asserted/Negated—Represents the state of data during a data write. Unused byte lanes do not supply valid data.

Timing Comments   Assertion/Negation—Initial beat coincides with $\overline{\text{BB}}$ 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 $\overline{\text{TA}}$ or following the assertion of $\overline{\text{TEA}}$ or certain $\overline{\text{ARTRY}}$ cases. Address retry operations are described in Section 8.3.3.2, "Data Phase Termination."

7.2.7.1.2 Data Signals (D0–D63)—Input

Following are the state meaning and timing comments for the data input signals.

State Meaning   Asserted/Negated—Represents the state of data during a data read transaction. For nonburst read operations in 64-bit mode (cache inhibited or write through operations), unselected byte lanes are not used.

Timing Comments   Assertion/Negation—Must be valid on the same bus clock cycle that $\overline{\text{TA}}$ is asserted.

7.2.7.2 Target Data Bus 32 ($\overline{\text{T32}}$)—Input

The $\overline{\text{T32}}$ signal is used to dynamically indicate the width of the bus.

State Meaning   Asserted—Indicates that only 32 bits, D0–D31, are used to transfer data. In 32-bit mode, the processor can generate single-beat (up to 4 bytes) and double-beat (from 5 to 8 bytes) nonburst transactions. In a double-beat transfer, the high-order word is transferred on the first beat and the low-order word is transferred on the second. Likewise, for each of the four double words of a burst transaction, the high-order word of a double word is always transferred before the low-order word.

Negated—Indicates that all 64 bits, D0–D63, are used to transfer data. For any transaction less or equal to 64 bits, the 602 performs a nonburst transaction.

Timing Comments   Assertion/Negation—Simultaneous with the assertion of $\overline{\text{AACK}}$. If dynamic bus sizing is not required, this signal can be tied low for static 32-bit systems and high for static 64-bit systems.

7.2.8 Data Transfer Termination Signals

Data termination signals are required after each data beat in a data transfer. Note that for nonburst transactions, the data termination signals also indicate the end of the data phase, while for burst accesses, the data termination signals apply to individual beats and indicate the end of the data phase only after the final data beat.

For a detailed description of how these signals interact, see Section 8.3.3.2, "Data Phase 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 602. Following are the state meaning and timing comments for the $\overline{\text{TA}}$ signal.

State Meaning   Asserted— Indicates that valid data on the bus has been provided or accepted by the system.

Negated—For read operations, indicates that the valid read data is not on the bus. For write operations indicates that the data beat must be extended.

Timing Comments   Assertion—May occur on any cycle during the 602's normal data transfer but not on the cycle before $\overline{\text{ARTRY}}$ is asserted if $\overline{\text{ARTRY}}$ cancellation is to be used.

Negation—For multiple-beat operations, $\overline{\text{TA}}$ must be negated the cycle after it is asserted unless conditions require $\overline{\text{TA}}$ to be asserted for the next data beat.

7.2.8.2 Transfer Error Acknowledge (TEA)—Input

The transfer error acknowledge ($\overline{\text{TEA}}$) signal is input only on the 602. Following are the state meaning and timing comments for the $\overline{\text{TEA}}$ signal.

State Meaning   Asserted—Indicates that a bus error occurred and that on the following cycle the 602 must terminate the transaction internally. The 602 may also take a machine check exception or may enter checkstop state if the machine check enable bit (MSR[ME]) is cleared. For more information, see Section 4.5.2.2, "Checkstop State (MSR[ME] = 0)." Assertion terminates the current transaction; that is, assertion of $\overline{\text{TA}}$ is ignored. The assertion of $\overline{\text{TEA}}$ causes the negation/high impedance of $\overline{\text{BB}}$ in the next clock cycle. However, data entering the GPR or the cache is not invalidated.

Negated—Indicates that no bus error was detected.

Timing Comments   Assertion/Negation—Assertion may occur on any cycle during the 602's normal operation (while $\overline{\text{BB}}$ is asserted and on the cycle after $\overline{\text{TA}}$ during reads). Assertion should occur for one cycle only.

Note: The system must ensure that $\overline{\text{TEA}}$ is negated by the start of the next transaction.

7.2.9 System Status Signals

Most system status signals are input signals that indicate when exceptions are received, when checkstop conditions have occurred, and when the 602 must be reset. The 602 generates the output signal, $\overline{\text{CKSTP\_OUT}}$, when it detects a checkstop condition.

7.2.9.1 Interrupt (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 602 initiates an interrupt if MSR[EE] is set; otherwise, the 602 ignores the interrupt. To guarantee that the 602 will take the external interrupt, the $\overline{\text{INT}}$ signal must be held active until the 602 takes the interrupt; otherwise, whether the 602 takes an external interrupt, depends on whether the MSR[EE] bit was set while the $\overline{\text{INT}}$ signal was held active.

Negated—Indicates that normal operation should proceed.

Timing Comments   Assertion—May occur at any time and may be asserted asynchronously to SYSCLK. The $\overline{\text{INT}}$ input is level-sensitive. Negation—Should not occur until the exception is taken.

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 602 initiates a system management interrupt operation if the MSR[EE] is set; otherwise, the 602 ignores the interrupt condition. The 602 must hold the $\overline{\text{SMI}}$ signal active until the interrupt is taken.

Negated—Indicates that normal operation should proceed.

Timing Comments   Assertion—May occur at any time and may be asserted asynchronously to SYSCLK. The $\overline{\text{SMI}}$ input is level-sensitive.

Negation—Should not occur until interrupt is taken.

7.2.9.3 Machine Check Interrupt ($\overline{\text{MCP}}$)—Input

The machine check interrupt ($\overline{\text{MCP}}$) signal is input only on the 602. Following are the state meaning and timing comments for the $\overline{\text{MCP}}$ signal.

State Meaning   Asserted—The 602 initiates a machine check interrupt operation if MSR[ME] and HID0[EMCP] are set; if MSR[ME] is cleared and HID0[EMCP] is set, the 602 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 602 ignores the interrupt condition. The $\overline{\text{MCP}}$ pin must be held asserted for two bus clock cycles.

Negated—Indicates that no machine check exception is being requested; normal operation should continue.

Timing Comments   Assertion—May occur at any time and may be asserted asynchronously to the SYSCLK. The $\overline{\text{MCP}}$ input is negative edge-sensitive.
Negation—May occur any time after the minimum $\overline{\text{MCP}}$ pulse width has been met

7.2.9.4 Checkstop Input ($\overline{\text{CKSTP\_IN}}$)—Input

The checkstop input ($\overline{\text{CKSTP\_IN}}$) signal is input only on the 602. Following are the state meaning and timing comments for the $\overline{\text{CKSTP\_IN}}$ signal.

State Meaning   Asserted—Indicates that the 602 must terminate operation and enter checkstop state 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. $\overline{\text{CKSTP\_IN}}$ is not maskable.

Negated—Indicates that normal operation should proceed.

Timing Comments   Assertion—May occur at any time and may be asserted asynchronously to the SYSCLK.

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 output ($\overline{\text{CKSTP\_OUT}}$) signal is output only on the 602. Note that the ($\overline{\text{CKSTP\_OUT}}$) signal is an open-drain type output and is either asserted or in high-impedance state. It requires an external pull-up resistor (for example, 10 kΩ to V_{DD}) to assure proper de-assertion 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 602 has detected a checkstop condition and has ceased operation.

                         Negated—Indicates that the 602 is operating normally.

Timing Comments   Assertion—May occur at any time and may be asserted asynchronously to SYSCLK.

                         Negation—Is negated upon assertion of $\overline{\text{HRESET}}$.

7.2.9.6 Reset Signals

There are two reset signals on the 602—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 system reset exception as described in Section 4.5.1.1, "Hard Reset and Power-On Reset." Output drivers are released to high impedance during the assertion of $\overline{\text{HRESET}}$.

                         Negated—Indicates that normal operation should proceed.

Timing Comments   Assertion—May occur at any time and may be asserted asynchronously to SYSCLK.

                         Negation—May occur any time after the minimum hard reset pulse width has been met.

This input 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 system reset exception as described in Section 4.5.1.2, "Soft Reset."

Negated—Indicates that a soft reset is not being requested; normal operation should proceed.

Timing Comments   Assertion—May occur at any time and may be asserted asynchronously to SYSCLK. The $\overline{\text{SRESET}}$ input is negative edge-sensitive.

Negation—May occur any time after the minimum $\overline{\text{SRESET}}$ pulse width has been met.

This input has additional functionality in certain test modes.

7.2.9.6.3 Reset Out ($\overline{\text{RESETO}}$)—Output

The $\overline{\text{RESETO}}$ signal is asserted whenever it is signalled by the watchdog timer or when the $\overline{\text{HRESET}}$ signal is asserted.

Note that when the $\overline{\text{HRESET}}$ signal is asserted, the 602 puts all output signals in high-impedance state. In order for $\overline{\text{RESETO}}$ to continue to be asserted, it should be pulled low.

State Meaning   Asserted—Indicates that $\overline{\text{HRESET}}$ is asserted or the watchdog timer reset signal is active.
Negated—Indicates that normal operation should proceed.

Timing Comments   Assertion—May occur any time asynchronously to SYSCLK.

Negation—May occur any time after the minimum hard reset pulse width has been met.

7.2.9.7 Quiescent Request ($\overline{\text{QREQ}}$)—Output

The quiescent request ($\overline{\text{QREQ}}$) signal is output only. Following are the state meaning and timing comments for the $\overline{\text{QREQ}}$ signal.

State Meaning   Asserted—Indicates that the 602 is requesting all bus activity normally required to be snooped to terminate or to pause so the 602 may enter a quiescent (low power) state. Once the 602 has entered a quiescent state, it no longer snoops bus activity.

Negated—Indicates that the 602 is not making a request to enter the quiescent state.

Timing Comments   Assertion/Negation—May occur on any cycle. The $\overline{\text{QREQ}}$ signal remains asserted for the duration of the quiescent state.

7.2.9.8 Quiescent Acknowledge ($\overline{\text{QACK}}$)—Input

The quiescent acknowledge ($\overline{\text{QACK}}$) signal is input only. Following are the state meaning and timing comments for the $\overline{\text{QACK}}$ signal.

State Meaning   Asserted—Indicates that all bus activity that requires snooping has terminated or paused, and that the 602 may enter the quiescent (or low power) state.

Negated—Indicates that the 602 may not enter a quiescent state and must continue snooping the bus.

Timing Comments Assertion/Negation—May occur on any cycle following the assertion of $\overline{QREQ}$, and must be held asserted for a minimum of one bus clock cycle.

Note that at start-up, $\overline{QACK}$ is sampled at the negation of $\overline{HRESET}$ to select reduced-pinout mode; if $\overline{QACK}$ is asserted at start-up, reduced-pinout mode is disabled.

7.2.9.9 Time Base Enable (TBEN)—Input

The time base enable (TBEN) signal is input only on the 602. Following are the state meaning and timing comments for the TBEN signal. This input is essentially a "count enable" control for the time base registers.

State Meaning Asserted—Indicates that the time base facility should continue clocking.

Negated—Indicates the time base facility should stop clocking.

Timing Comments Assertion/Negation—May occur on any cycle.

7.2.10 JTAG/Scan Interface Signals

The 602 has extensive on-chip test capability including the following:

- Built-in instruction and data cache self test (BIST)
- Debug control/observation (COP)
- Boundary scan that supports most functions defined by JTAG IEEE 1149.1

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.

A detailed discussion of the 602 boundary-scan test functions is provided in Appendix C, "Boundary-Scan Testing Support.".

The COP/boundary scan interface is shown in Figure 7-7.

TDI (Test Data Input)

TMS (Test Mode Select)

TCK (Test Clock input)

TDO (Test Data Output)

TRST (Test Reset)
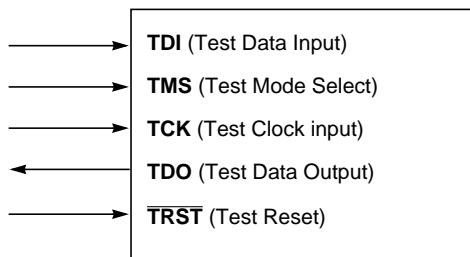
Figure 7-7. Boundary-Scan Interface

The following sections describe the test access port (TAP) signals used for boundary-scan testing.

7.2.10.1 Test Data Output (TDO)—Output

Following is the state meaning for the TDO output signal.

State Meaning   Asserted/Negated—The contents of the selected internal instruction or data register are shifted out onto this signal on the falling edge of TCK. The TDO signal will remain in a high-impedance state except when scanning of data is in progress.

7.2.10.2 Test Data Input (TDI)—Input

Following is the state meaning for the TDI input signal.

State Meaning   Asserted/Negated—The value presented on this signal on the rising edge of TCK is clocked into the selected test instruction or data register.

7.2.10.3 Test Clock (TCK)—Input

The test clock (TCK) signal is an input on the 602. Following is the state meaning for the TCK input signal.

State Meaning   Asserted/Negated—This input should be driven by a free-running clock signal with a 50% duty cycle. Input signals to the test access port (TAP) are clocked in on the rising edge of TCK. Changes to the TAP output signals occur on the falling edge of TCK. The test logic allows TCK to be stopped.

Note that this input contains an internal pull-up resistor to ensure that an unterminated input appears as a high signal level to the test logic.

7.2.10.4 Test Mode Select (TMS)—Input

The test mode select (TMS) signal is an input on the 602. Following is the state meaning for the TMS input signal.

State Meaning   Asserted/Negated—This signal is decoded by the internal TAP controller to distinguish the primary operation of the test support circuitry.

Note that this input contains an internal pull-up resistor to ensure that an unterminated input appears as a high signal level to the test logic.

7.2.10.5 Test Reset ($\overline{\text{TRST}}$)—Input

The test reset ($\overline{\text{TRST}}$) signal is an input on the 602. Following is the state meaning for the $\overline{\text{TRST}}$ input signal.

State Meaning   Asserted—This input causes asynchronous initialization of the internal test access port controller. During power-on reset, the system should assert $\overline{\text{TRST}}$ to reset the boundary-scan control logic.

Negated—Indicates normal operation.

Note that this input contains an internal pull-up resistor to ensure that an unterminated input appears as a high signal level to the test logic. Note that if boundary-scan is not used for a design, $\overline{TRST}$ can be connected to $\overline{HRESET}$.

7.2.11 Clock Signals

The clock signal inputs of the 602 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.

Refer to the 602 hardware specifications for exact timing relationships of the clock signals.

7.2.11.1 System Clock (SYSCLK)—Input

The 602 requires a single system clock (SYSCLK) input. This input sets the frequency of operation for the bus interface. Internally, the 602 uses a phase-locked 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 an integer multiple (x1, x2, x3, or x4) of the SYSCLK frequency allowing the CPU core to operate at an equal or greater frequency than the bus interface. The state meanings are as follows:

State Meaning   Asserted/Negated—The SYSCLK input is the primary clock input for the 602 and represents the bus clock frequency for 602 bus operation. Internally, the 602 may be operating at an integer multiple of the bus clock frequency.

Timing Comments   Duty cycle—Refer to the 602 hardware specifications for timing comments. A loose duty cycle is allowed.
Note: SYSCLK is the frequency reference for the internal PLL clock generator. To ensure proper PLL operation, SYSCLK must not be suspended or varied during normal operation.

7.2.11.2 Test Clock (CLK_OUT)—Output

The Test Clock (CLK_OUT) signal is an output signal (output-only) on the 602. 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. The test clock frequency is chosen through the HID0 register. If HID0[SBCLK] (bit 4) is set, the test clock uses the bus clock frequency; otherwise, the test clock uses the processor clock frequency. The default state of the CLK_OUT signal is high-impedance. The CLK_OUT signal is provided for testing only.

Timing Comments   Assertion/Negation—Refer to the 602 hardware specifications for timing comments.

See Section 2.1.2.1.1, "Hardware Implementation Register 0 (HID0)," for information on configuring CLK_OUT through software.

7.2.11.3 PLL Configuration (PLL_CFG0–PLL_CFG3)—Input

The PLL (phase-lock loop) is configured by the PLL_CFG0–PLL_CFG3 signals. For a given SYSCLK (bus) frequency, the PLL configuration signals 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. PLL_CFG0–PLL_CFG1 determine the processor/SYSCLK frequency ratio. PLL_CFG2–PLL_CFG3 determine the processor/PLL frequency ratio. Table 7-8 shows the permissible settings for the PLL_CFG signals and the resultant frequency relationships; see the 602 hardware specifications for exact settings.

Timing Comments   Assertion/Negation—Must remain stable during operation; should only be changed during the assertion of $\overline{\text{HRESET}}$, or during sleep mode.

Table 7-8. PLL Configuration

PLL_CFG0–PLL_CFG1	CPU (Core) Frequency	PLL_CFG2–PLL_CFG3	PLL (VCO) Frequency	Example SYSCLK/CPU and Resulting PLL Frequencies (MHz)			
				PLL = 133.3	**PLL = 150**	**PLL = 160**	**PLL = 200**
01	SYSCLK x 2	01	CPU x 2	33.33/66.66	37.5/75	40/80	—
			CPU x 4	16.66/33.33	18.75/37.5	20/40	25/50
10	SYSCLK x 3	00	CPU x 2	22.22/66.66	25/75	26.66/80	—
00	—	10	—	PLL bypass— the SYSCLK input signal clocks the internal processor directly and the bus is set for 1:1 mode operation.			
11	—	11	—	Clock off—no clocking occurs regardless of the SYSCLK input.			

Notes:

The resulting CPU / SYSCLK frequencies shown are for reference. Some PLL configurations may select bus, CPU, or PLL frequencies that are not useful, not supported, or not tested.

Although 1:1 mode is not an operational mode, it may be used for testing.

7.2.12 Power and Ground Signals

The 602 provides the following additional connections for power and ground:

- V_{DD} and OV_{DD}—V_{DD} and OV_{DD} provide the connection for the supply voltage. On the 602, there is no electrical distinction between the V_{DD} and the OV_{DD} signals. These signals are internally shorted together.

- AV_{DD}—The AV_{DD} power signal provides power to the clock generation phase-lock loop. See the 602 hardware specifications for information on how to use this signal.

- GND and OGND—The GND and OGND signals provide the connection for grounding the 602. On the 602, there is no electrical distinction between the GND and OGND signals. These signals are internally shorted together.

Chapter 8
System Interface Operation

This chapter describes the PowerPC 602 microprocessor bus interface and its operation. It shows how the 602 signals, defined in Chapter 7, "Signal Descriptions," interact to perform address and data transfers. This chapter includes timing diagrams that illustrate the operation of the 602's time-multiplexed bus and its ability to dynamically function as a 32- or 64-bit data bus.

8.1 PowerPC 602 Microprocessor System Interface Overview

The 602 bus interface is a time-multiplexed interface. That is, the 64 physical connections that are used to transfer data during the data phase are used to transfer the 32-bit address, as well as other information, during the address phase. This double use of physical connections greatly reduces the number of physical connections to the processor as well as the power requirements.

During the address phase, the high-order 32 connections are used to transfer the 32-bit address, and the low-order connections are used to transfer information about attributes of the subsequent data transfer, such as the size of the data and whether the transfer is a burst or nonburst operation—signals that have dedicated pins on other PowerPC processors, such as the PowerPC 603 and the PowerPC 604 microprocessors.

During the data phase, the 64 physical connections are used exclusively to transfer data. However, because the 602 supports dynamic bus sizing, the bus can function as either a 64- or 32-bit data bus depending on the device with which the 602 is communicating.

The 602 on-chip caches can be configured as either write-through or write-back. In write-back mode, most transactions are burst-read memory operations that update an entire cache line (referred to here as a cache block), followed by burst-write operations, and noncacheable (write-through) operations. Additionally, there can be address-only operations (for example, global memory operations that are snooped), atomic memory operations, and address retry activity (for example, when a snooped read access hits a modified block in the cache).

When the data bus is in 64-bit mode, all burst operations consist of four data beats and transfer 32 bytes (an eight-word cache block) per transaction; all nonburst operations are single-beat transactions that transfer up to 8 bytes (1 double word).

When the data bus is in 32-bit mode, burst transactions also transfer a full eight-word cache block of data, but because it can do so only one word at a time, it takes eight beats. There are two types of nonburst operations in 32-bit mode—single-beat operations that transfer up to 32 bits, and double-beat operations that transfer up to 64 bits.

Access to the system interface is granted through an external arbitration mechanism that allows devices to compete for bus mastership. This arbitration mechanism allows the 602 to be integrated into systems that implement various fairness and bus parking procedures to avoid arbitration overhead.

8.1.1 Operation of the Instruction and Data Caches

The 602 provides independent instruction and data caches. Each cache is a physically-addressed, 4-Kbyte cache with two-way set-associativity. Both caches consist of 64 sets of two cache blocks, with eight words in each cache block.

The data cache tags are single-ported, so snoop accesses cannot occur simultaneously with load or store operations. Snoop accesses have the highest priority and are given first access to the tags, unless the snoop access occurs when a tag is being accessed (for example, by the load/store unit), in which case the snoop is retried and must rearbitrate for cache access.

On a snoop hit, the snooping device asserts the address retry ($\overline{\text{ARTRY}}$) signal, causing the operation whose address caused the snoop hit to be delayed so the snooping device can perform the necessary bus operation to ensure cache coherency. After the snooping device completes its operation, the original operation can be retried. On a snoop miss, the load or store operation deferred due to a snoop access is performed on the clock cycle following the snoop. Bus timing for snoop operations can be found in Section 8.5.4, "Snooping."

The 602 supports an additional snooping mechanism, known as injected snooping. While the 602 as a bus master performs a burst-read transaction, the read target device can inject the snoop address onto the bus between data beats. Injected snooping is described in Section 8.4.2, "Qualified Snoop Conditions," and is illustrated in Section 8.5.4.7, "Injected Snoop Timings."

The 602 supports a three-state coherency protocol that supports the modified, exclusive, and invalid (MEI) cache states. The protocol is a subset of the MESI (modified/exclusive/shared/invalid) four-state protocol and operates coherently in systems that contain four-state caches. Except for the **dcbz** instruction, the 602 does not broadcast cache control instructions. The cache control instructions are intended for the management of the local cache but not for other caches in the system.

Cache blocks in the 602 are loaded in four beats of 64 bits each (or eight beats of 32 bits each when the bus is operating in 32-bit bus mode). The burst load is performed as "critical

double word first." The critical double word is simultaneously written to the cache and forwarded to the requesting unit, thus minimizing stalls due to load delays. As additional data or instructions arrive, they can be accessed by the requesting unit.

For further details regarding byte ordering, see Section 8.3.2.2, "Transfer Attributes."

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.
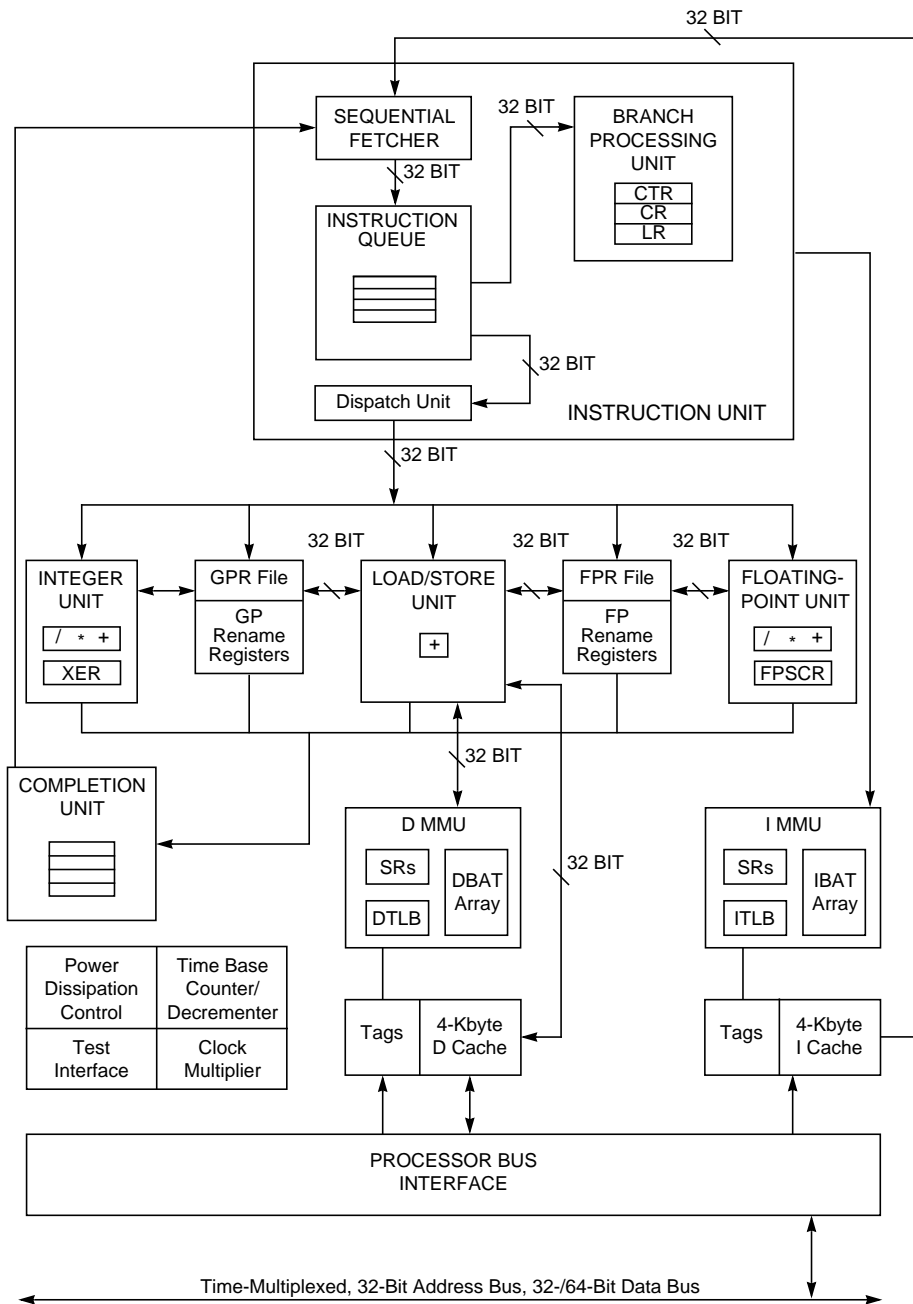
Figure 8-1. PowerPC 602 Microprocessor Block Diagram

Cache blocks are selected for replacement based on an LRU (least-recently used) algorithm. Each time a cache block is accessed, it is tagged as the most-recently used cache block of the set. When a miss occurs, if both cache blocks in the set are marked as valid, the least recently used cache block is replaced with the new data. Coherency is maintained in the data caches, and if the data in the least-recently used cache block in a data cache is modified it is written back to memory before the burst read operation.

8.1.2 32-Bit Data Bus Mode

The 602 supports an optional 32-bit data bus mode. The 32-bit data bus mode operates the same as the 64-bit data bus mode with the exception of the byte lanes involved in the transfer and the number of data beats that are performed. The number of data beats required for a data phase in the 32-bit data bus mode is one, two, or eight beats depending on the amount of the data being transferred and the cache attributes for the address (determined by the W, I, and M bits). For additional information about 32-bit data bus mode, see the examples in Section 8.5, "Bus Timing Examples."

8.1.3 Clocks

The 602 requires a single system clock input (SYSCLK). This input sets the frequency of operation for the bus interface. Internally, the 602 uses a phase-locked loop (PLL) circuit to generate a master clock for all of the processor circuity (including the bus interface circuitry) which is phase-locked to the SYSCLK input. The master clock may be set to an integer multiple—either double or triple the frequency of SYSCLK, allowing the processor core to run at optimum speed independently from the bus speed.

The PLL is configured by the PLL_CFG0–PLL_CFG3 signals. For a given bus frequency (SYSCLK), these pins set the processor frequency and PLL (VCO) frequency. The encoding for the PLL configuration pins are shown in Section 7.2.11.3, "PLL Configuration (PLL_CFG0–PLL_CFG3)—Input."

All signals for the 602 bus interface are specified with respect to the rising-edge of the external system clock input (SYSCLK) and are guaranteed to be sampled as inputs or changed as outputs with respect to that edge. Since the same clock edge is referenced for driving or sampling the bus signals, the possibility of clock skew could exist between various modules in a system due to routing or the use of multiple clocks. The system must handle any such clock skew problems.

8.1.4 Operation of the System Interface

The following sections describe how the 602 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.

	Bar over signal name indicates active low
$\overline{bg}$	602 input (while 602 is a bus master)
$\overline{BR}$	602 output (while 602 is a bus master)
ADDR+	602 output (grouped: here, address plus attributes)
$\overline{qual\ BG}$	602 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
	602 three-state output or input
	602 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

The 602 interface is synchronous—all 602 input signals are sampled and output signals are driven on the rising edge of the bus clock cycle (see the 602 hardware specifications for exact timing information).

8.2 Memory Access Protocol

Memory accesses are divided into address and data phases, each of which consists of subphases, as shown in Figure 8-3.

Figure 8-3 shows the address and data phases with their respective subphases—arbitration, transfer, and termination. It shows a data transfer that consists of a nonburst transfer. Burst transfers of 32-byte cache blocks require data transfer termination signals for each beat of data.

```
          ADDRESS PHASE                                    DATA PHASE

┌──────────────┬─────────────────┬──────────────────┬────────────────┬──────────────────┐
│  Arbitration │ Address Transfer │ Address Termina- │ Data Transfer  │ Data Termination │
└──────────────┴─────────────────┴──────────────────┴────────────────┴──────────────────┘
```
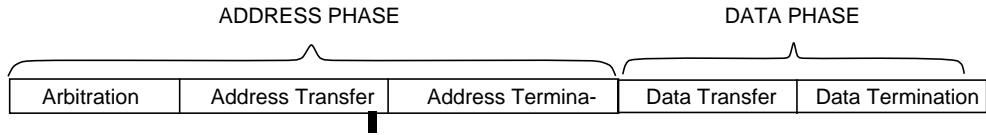
Figure 8-3. Address and Data Phases of a Memory Transaction

The basic functions of the address and data phase are as follows:

- **Address phase**
  - Arbitration: During arbitration, bus arbitration signals are used to gain bus mastership.
  - Transfer: When the 602 becomes bus master it transfers the address and the transfer attributes.
  - Termination: After the address transfer, the system signals that the address phase is complete or that it must be repeated.

- **Data Phase**
  - Transfer: After the address phase, the 602 samples the data bus for read operations or drives the data bus for write operations.
  - Termination: Data termination signals are required for each data beat in a data transfer. Note that in a single-beat transaction, the data termination signals also indicate the end of the phase, while in burst accesses, the data termination signals apply to individual beats and indicate the end of the phase after the final data beat.

The 602 generates address-only bus transfers as the result of execution of the **dcbz** instructions, which use only the address bus with no data transfer involved. Additionally, the 602's retry capability provides an efficient snooping protocol for systems with multiple memory systems (including caches) that must remain coherent. Timing examples for address-only transactions are shown in Section 8.5.5.1, "Single-Cycle Address-Only Transaction," and Section 8.5.5.2, "Multicycle Address-Only Transaction."

8.3 Address Bus Phase

This section describes the three subphases of the address phase—bus arbitration, address transfer, and address termination.

8.3.1 Bus Arbitration

Before the 602 can access the system bus, it must first be granted mastership of the bus.

When the instruction fetcher or the processor core generates the need to access memory, it generates an internal signal (identified here conceptually as the $\overline{\text{need\_bus}}$ signal). For the

602 to access the bus, it must be granted bus mastership, which occurs after other devices have completed their transactions and the 602 is given a qualified bus grant. The conditions necessary for a qualified bus grant are described in Table 8-1.

Table 8-1. Input Conditions for a Qualified Bus Grant

Signal	State (Input)	Description
Bus Grant $\overline{\text{BG}}$	Asserted	Assertion indicates that the 602 may, with the proper qualification, assume mastership of the bus. If $\overline{\text{BG}}$ is asserted for the 602 before it is needed (that is, 602 is parked), the 602 does not assert $\overline{\text{BR}}$.
Bus Busy $\overline{\text{BB}}$	Negated	The negation of $\overline{\text{TS}}$ and $\overline{\text{BB}}$ indicate that no other master is currently using the bus. If the $\overline{\text{BB}}$ input is asserted, another bus device is in its data phase and the current 602 cannot have bus access until the operation completes successfully and negates $\overline{\text{BB}}$.
Transfer Start $\overline{\text{TS}}$	Negated	The negation of $\overline{\text{TS}}$ and $\overline{\text{BB}}$ indicate that no other master is currently using the bus. If the $\overline{\text{TS}}$ input is asserted, the address and transfer attribute signals are valid for another device and the bus is not available.
Address Retry $\overline{\text{ARTRY}}$	Negated for at least one bus clock cycle	If the $\overline{\text{ARTRY}}$ input is asserted, a transfer is being retried on the bus and the bus is not available. Negating $\overline{\text{ARTRY}}$ indicates that the address retry window for any just-completed address phase has passed. Note that the sampling of $\overline{\text{ARTRY}}$ requires additional qualification since $\overline{\text{ARTRY}}$ may be set to the high impedance state the second cycle following the assertion of $\overline{\text{AACK}}$ and cannot be sampled reliably on this clock.

Note that the bus request ($\overline{\text{BR}}$) signal is not a part of a qualified bus grant. Although asserting $\overline{\text{BR}}$ indicates that the 602 is requesting mastership of the bus, the 602 does not assert $\overline{\text{BR}}$ if it is parked on the bus (that is, if $\overline{\text{BG}}$ is already asserted when the $\overline{\text{need\_bus}}$ internal signal is generated).

For systems that share the system bus with other processors or devices, the logic required for arbitration can be complex, whereas, for designs in which the 602 is the only device that accesses the system bus, arbitration can be very simplified. For example, the $\overline{\text{BG}}$ signal can be connected low, which eliminates the need to assert $\overline{\text{BR}}$ when a "need_bus" condition occurs. This section described bus arbitration under two conditions—when the 602 is not the current bus master and must compete with other resources for it and when the processor is parked on the bus.

Arbiter implementations may require additional signals to coordinate bus master/slave/snooping activities. Note that bus busy ($\overline{\text{BB}}$) is a bidirectional signal. These signals are inputs unless the 602 has mastership of the bus; they must be connected high through pull-up resistors so that they remain negated when no devices have control of the buses.

Upon recognizing a qualified bus grant, the 602 takes bus mastership by asserting $\overline{\text{TS}}$ (and by negating $\overline{\text{BR}}$ if the 602 was not parked. At the same time, the 602 drives the address and transfer attributes for the requested access.

The timing for the nonparked case are described in Section 8.3.1.1, "Bus Arbitration—Nonparked Case."

8.3.1.1 Bus Arbitration—Nonparked Case

When the 602 needs to access the external bus and does not have a qualified bus grant, it asserts bus request ($\overline{BR}$) until it is granted bus mastership and the bus is available (see Figure 8-4). Note that the 602 can cancel the bus request before the bus has been granted. The external arbiter must grant master-elect status to the potential master by asserting the bus grant ($\overline{BG}$) signal. The 602 requesting the bus determines that the bus is available when the $\overline{BB}$ input is negated. When the bus is not busy ($\overline{BB}$ and $\overline{TS}$ inputs are negated), $\overline{BG}$ is asserted and the address retry ($\overline{ARTRY}$) input is negated, and was negated the previous cycle; the 602 has what is referred to as a qualified bus grant. The 602 assumes bus mastership by asserting $\overline{TS}$ when it receives a qualified bus grant. The $\overline{TS}$ signal indicates that the address and transfer attribute signals are valid and that the memory operation can begin.
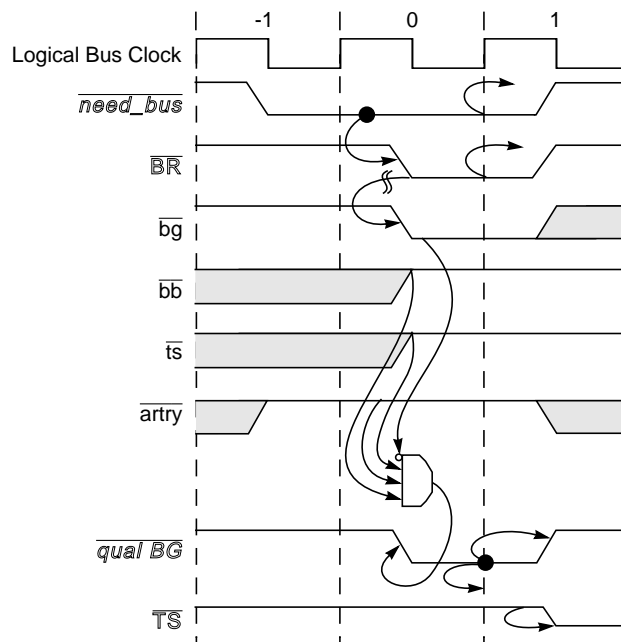
Figure 8-4. Bus Arbitration—Nonparked Case

External arbiters must allow only one device at a time to be the bus master. In implementations in which no other device can be a master, $\overline{BG}$ can be grounded (always asserted) to continually grant mastership of the bus to the 602.

If the 602 asserts $\overline{BR}$ before the external arbiter asserts $\overline{BG}$, the 602 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 one bus clock cycle required for arbitration are eliminated if the 602 is parked, reducing overall memory latency for a transaction. The 602 always negates $\overline{BB}$ for at least one bus clock cycle after $\overline{AACK}$ is asserted, even if it is parked and has another transaction pending.

8.3.1.2 Bus Arbitration—Parked Case

A device is said to be parked when the $\overline{BG}$ input signal is asserted before the device has generated a reason to access the bus. If $\overline{BG}$ is asserted, an internal "need_bus" condition does not assert the $\overline{BR}$ signal, reducing by at least one cycle the time required to access the bus.

At its simplest, in a single-processor system the $\overline{BG}$ signal can be connected asserted (low) so there is never a need to assert the $\overline{BR}$ signal. Bus parking may also be used in a multiprocessor system when an external arbiter uses some scheme to leave the bus granted to the device most likely to use it. 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.

In the nonparked case (described in Section 8.3.1.2, "Bus Arbitration—Parked Case"), the 602 must first assert $\overline{BR}$ to the arbiter to request the bus, and then may need wait to receive a bus grant from the arbiter.

When the 602 is parked on the bus and it determines a need to perform a bus transaction internally ("need_bus"), it does not assert $\overline{BR}$ but immediately assumes bus ownership on the next cycle. Eliminating the need to assert $\overline{BR}$ reduces the overall access latency seen by the 602 by one cycle.

Bus timing for the parked case is shown in Figure 8-5.
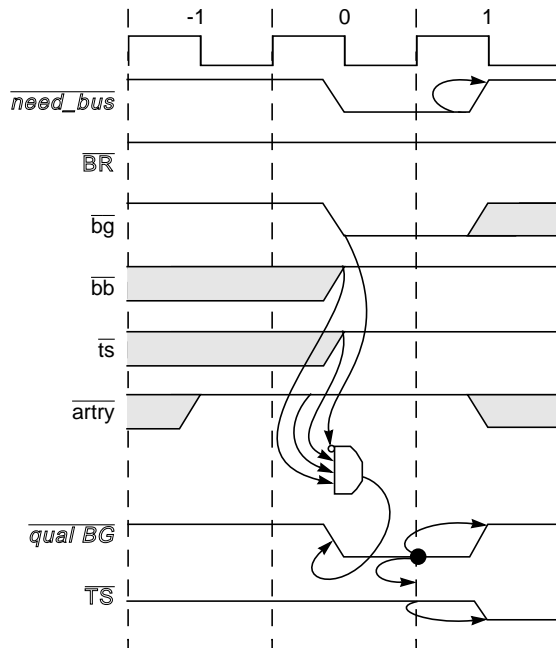
Figure 8-5. Bus Arbitration Showing Bus Parking

When the 602 receives a qualified bus grant, it assumes bus mastership by asserting $\overline{BB}$ and negating the $\overline{BR}$ output signal in the nonparked case. Meanwhile, the 602 drives the address for the requested access onto the bus and asserts $\overline{TS}$ to indicate the start of a new transaction.

When designing external bus arbitration logic, note that the 602 may assert $\overline{BR}$ without using the bus after it receives the qualified bus grant. For example, in a system using bus snooping, the 602 may assert $\overline{BR}$ to perform a read-with-intent-to-modify-atomic (RWITMA) operation but may cancel that operation if it snoops an access that cancels the reservation associated with the RWITMA. Once the 602 is granted the bus, it no longer needs to perform the RWITMA; therefore, the 602 does not assert $\overline{TS}$ and does not use the bus for the read operation. Note that the 602 asserts $\overline{BR}$ for at least one clock cycle in these instances.

8.3.2 Address Transfer Subphase

During the address subphase, the bus master transmits the physical address and transfer attributes to any slave devices. To ensure cache coherency, snooping logic may monitor the transfer. The signals used in this phase are transfer start ($\overline{TS}$) and the address and attributes signals described in Section 7.2.4, "Transfer Attribute Signals." $\overline{TS}$ indicates that the 602 has begun a bus transaction and that the address and transfer attributes are valid. The 602 always asserts $\overline{TS}$ to begin a transaction and requires other masters to do the same.

The timing for the transfer start and address attribute signals is shown in Figure 8-6, as well as in the many examples in Section 8.5, "Bus Timing Examples."

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 8.3.2.3, "Address Phase Termination."

The signals used in the address transfer include the following signals:

- Address transfer start signal—transfer start ($\overline{\text{TS}}$)
- Address transfer signals—Address bus (A0–A31)
- Address transfer attribute signals—Transfer type (TT0–TT4), transfer code (TC0–TC1), transfer size (TSIZ0–TSIZ2), transfer burst ($\overline{\text{TBST}}$), cache inhibit ($\overline{\text{CI}}$), write-through ($\overline{\text{WT}}$), and global ($\overline{\text{GBL}}$). The 602 also has byte enable signals (BE0–BE7) that can be used instead of the transfer size signals.

Figure 8-6 shows that the timing for all of these signals, except $\overline{\text{TS}}$ 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 602 has begun an address transfer and that the address and transfer attributes are valid (within the context of a synchronous bus). The $\overline{\text{TS}}$ signal remains asserted for the entire address transfer and indicates to other processors that the bus is in use. Keeping the $\overline{\text{TS}}$ signal asserted prevents another device from achieving a qualified bus grant.
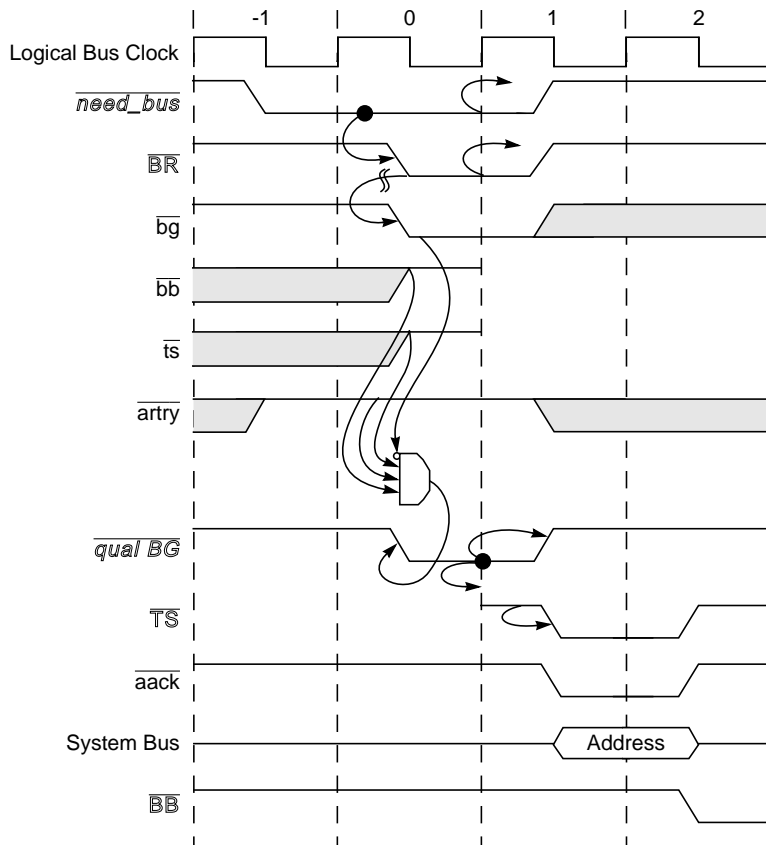
Figure 8-6. Address Bus Transfer

Figure 8-6 shows the fastest possible address cycle; that is, the address cycle lasts only one bus clock cycle and is simultaneous with the assertion of $\overline{\text{TS}}$ and $\overline{\text{AACK}}$.

8.3.2.1 Address Phase Signal Configurations

Because the 602 bus is multiplexed, signal connections have multiple signal assignments depending not only on whether the processor is in address or data phase, but whether the data transfer is a burst or nonburst transfer. Table 8-2 summarizes these signal assignments.

Table 8-2. Time-Multiplexed Signal Assignments

Data Phase Signals	Address Phase Address Signals/Transfer Attribute Signal Assignments	
	Nonburst Transactions	Burst Transactions
D0–D31	Address bus (A0–A31)	Address bus (A0–A31)
D32–D39	Reserved—Don't care	Prefetch read address bus (PFADDR0 to PFADDR20)
D40–D47	Byte enable (BE0–BE7)	
D48–D49	Reserved —Don't care	
D50–D52	Transfer size (TSIZ0–TSIZ2)	
D53	Transfer burst ($\overline{\text{TBST}}$) (negated)	Transfer burst ($\overline{\text{TBST}}$) (asserted)
D54–D58	Transfer type (TT0–TT4)	Transfer type (TT0–TT4)
D59	Global ($\overline{\text{GBL}}$)	Global ($\overline{\text{GBL}}$)
D60	Cache inhibit ($\overline{\text{CI}}$)	Cache inhibit ($\overline{\text{CI}}$)
D61	Write through ($\overline{\text{WT}}$)	Write through ($\overline{\text{WT}}$)
D62–D63	Transfer code (TC0–TC1)	Transfer code (TC0–TC1)

Note that if the byte enable signals (BE0–BE7) are used, the TSIZ0–TSIZ2 and A29–A31 signals can be ignored. The TSIZ signals are provided for compatibility with other PowerPC processors; however, the byte enable signals provide a simpler way of identifying both the size and the starting address of the data to be transferred by identifying the byte lanes that are to be used in the data phase. This correlation between the byte enable signals and the byte lanes is shown in Figure 8-7.
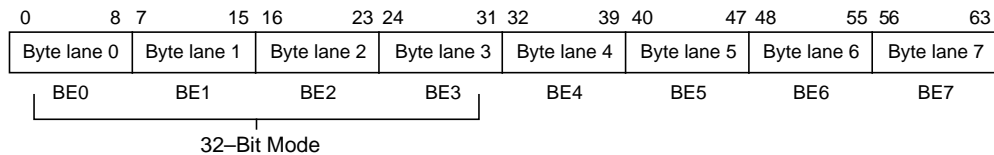
Figure 8-7. Data Format Using Byte Enable Signals

Note that in 32-bit mode, only byte lanes 0–3 are used. For more information about the byte enable signals, see Section 7.2.4.3, "Byte Enable (BE0–BE7)." The two methods of specifying data size and starting address are discussed in Section 8.3.2.2, "Transfer Attributes," and are summarized in Table 8-3.

8.3.2.2 Transfer Attributes

During the address phase, the 602 not only transfers the address for the memory access, but also information about the overall transaction as well, such as whether there is a data phase, and, if so, whether the data phase consists of a burst or nonburst transaction, the size,

alignment, and the order of the data to be transferred. Those attributes are signaled by the high-order pins (32–63) on the multiplexed bus and are described in the following sections.

8.3.2.2.1 Transfer Type Encodings

The transfer type signals (TT0–TT4) indicate the type of transaction in progress. They also provide information on how the 602's caches handle the transaction and instruct other caches in the system how to treat the transaction for cache coherency purposes. The transfer type encodings are shown in Table 8-3.

The transfer type signals are also snooped by 602 and instruct 602 how to handle a cache block on a snoop hit. The transfer type signals may specify clean, flush, or kill operations (these operations are described in Table 3-5).

Table 8-3. Transfer Type Encoding

| TT0–TT4 | Command | 602 Master | | 602 Snooper |
		Bus Transaction	Source of Transaction	Hit Response
00000	Clean block	n/a	n/a	Clean
00100	Flush block	n/a	n/a	Flush
01000	**sync**	n/a	n/a	n/a
01100	Kill block	Address-only	**dcbz**	Kill
10000	**eieio**	n/a	n/a	n/a
10100	Graphics write	n/a	n/a	n/a
11000	TLB invalidate	n/a	n/a	n/a
11100	graphics read	n/a	n/a	n/a
00001	**lwarx** reservation set	n/a	n/a	n/a
00101	**stwcx.** reservation clear	n/a	n/a	n/a
01001	**tlbsync**	n/a	n/a	n/a
01101	**icbi**	n/a	n/a	n/a
1XX01	Reserved	n/a	n/a	n/a
00010	Write-with-flush	Nonburst write	CI or WT store	Flush
00110	Write-with-kill	Burst (not global)	Castout or snoop copy-back	Kill
01010	Read	Nonburst read	CI Load	Clean or flush
01110	Read-with-intent-to-modify	Burst	Load miss or store miss	Flush
10010	Write-with-flush-atomic	Nonburst write	**stwcx**	Flush
10110	(Reserved)	n/a	n/a	n/a
11010	Read-atomic	Nonburst read	**lwarx** (CI load)	Clean or flush
11110	Read-with-intent-to-modify-atomic	Burst	**lwarx** (load miss)	Flush
00X11	(Reserved)	n/a	n/a	n/a

Table 8-3. Transfer Type Encoding (Continued)

TT0–TT4	Command	602 Master		602 Snooper
		Bus Transaction	Source of Transaction	Hit Response
01011	Read-with-no-intent-to-cache	n/a	n/a	Clean
01111	Reserved	n/a	n/a	n/a
1XX11	Reserved	n/a	n/a	n/a

Notes:

1. Snoop hits cancel the bit set by an **lwarx** instruction.

2. For read operations, whether the 602 cleans or flushes the cache block during a snoop is determined by the $\overline{\text{TBST}}$ input signal. Nonburst read operations ($\overline{\text{TBST}}$ negated) clean to emulate read-with-no-intent-to-cache.

3. Cast-out and snoop copy-back operations are generally marked as nonglobal and are not snooped (except for reservation monitoring). Other masters, however, may perform DMA write operations with the same transfer type encoding and marked global.

4. A write operation (whether global or nonglobal) cancels an active reservation during a snoop hit in the reservation register (independent of snoop hit in cache).

5. The TT1 signal may be generally unincorporated as a "read verses write" indicator for the bus.

8.3.2.2.2 Transfer Size and Burst Ordering

The transfer size (TSIZ0–TSIZ2) signals indicate the size of the requested data transfer. The transfer size signals may be used with $\overline{\text{TBST}}$ and A27–A31 to determine which byte lanes of the data bus are used for the transfer. For nonburst transfers, the transfer size signals specify the number of bytes starting from the byte location addressed by A27–A31. For burst transfers, double words are always assumed for each data beat of the burst. The 602 always attempts to transfer 4 double words during a burst transaction. Burst read transfers are performed critical-double-word-first and wraparound the end of the cache block. Burst write transfers are always performed zero-word-first.

Table 8-4. Data Transfer Size

$\overline{\text{TBST}}$	TSIZ0–TSIZ2	Transfer Size	Comments
Negated	001	1 byte	Byte
Negated	010	2 bytes	Half word
Negated	011	3 bytes	—
Negated	100	4 bytes	Word
Negated	101	5 bytes	—
Negated	110	6 bytes	—
Negated	111	7 bytes	—
Negated	000	8 bytes	Double word (bus width in 64-bit mode)
Asserted	Invalid	32 bytes	Four double words (four data beats in 64-bit mode)

The basic coherency size (cache block size) of the bus is 32 bytes for the 602. For proper snooping, data transfers that cross an aligned 32-byte boundary must present a new address onto the bus at that boundary or must operate as not coherent with respect to the 602.

The 602 never generates a bus transaction with a transfer size of 5, 6, or 7 bytes. Other PowerPC processors may specify these transfer sizes.

Table 8-5 shows the order in which double words are transferred during burst operations in 64-bit mode.

Table 8-5. Burst Ordering–64-Bit Mode

Data Beat	For Starting Address:			
	A27–A28 = 00	A27–A28 = 01	A27–A28 = 10	A27–A28 = 11
Data beat 1	Double word 0	Double word 1	Double word 2	Double word 3
Data beat 2	Double word 1	Double word 2	Double word 3	Double word 0
Data beat 3	Double word 2	Double word 3	Double word 0	Double word 1
Data beat 4	Double word 3	Double word 0	Double word 1	Double word 2

The A27–A28 signals specify the first double word of the 32-byte block being transferred; the remaining double words to transfer must wrap around the block. A29–A31 are always "don't cares" for burst transfers by the 602.

Table 8-6 shows the order in which words are transferred during burst operations in 32-bit mode.

Table 8-6. Burst Ordering—32-Bit Mode

Data Beat	For Starting Address:			
	A27–A28 = 00	A27–A28 = 01	A27–A28 = 10	A27–A28 = 11
Data beat 1	High word/DW0	High word/DW1	High word/DW2	High word/DW3
Data beat 2	Low word/DW0	Low word/DW1	Low word/DW2	Low word/DW3
Data beat 3	High word/DW1	High word/DW2	High word/DW3	High word/DW0
Data beat 4	Low word/DW1	Low word/DW2	Low word/DW3	Low word/DW0
Data beat 5	High word/DW2	High word/DW3	High word/DW0	High word/DW1
Data beat 6	Low word/DW2	Low word/DW3	Low word/DW0	Low word/DW1
Data beat 7	High word/DW3	High word/DW0	High word/DW1	High word/DW2
Data beat 8	Low word/DW3	Low word/DW0	Low word/DW1	Low word/DW2

8.3.2.2.3 Alignment

Aligned transfers are byte, half-word, word, and double-word transfers that lie on their respective natural address boundaries (bytes on byte address boundaries, words on word address boundaries, etc.). The supported aligned transfers for 64-bit mode are shown in Table 8-7. Note that if the byte enable signals are used to specify the byte lanes to be used, the TSIZ0–TSIZ2 and A29–A31 signals can be ignored, and vice versa.

Although the 602 supports only single-precision floating-point arithmetic in hardware and provides only 32-bit FPRs in hardware, it can perform single-precision operations on double-precision operands. The 602 does support load and store double-precision instructions. If the operand can be represented as a 32-bit single-precision operand, it is converted. Otherwise, an emulation trap exception (0x1600) is taken.

Misaligned transfers are supported in the 602.

Table 8-7. Data Transfers—64-Bit Mode

Program Transfer Size	Bus BE0–BE7	Bus TSIZ0–TSIZ2	Bus A29–A31	Byte Lanes							
				D0...				...D63			
				0	1	2	3	4	5	6	7
Byte	10000000	001	000	√	—	—	—	—	—	—	—
	01000000	001	001	—	√	—	—	—	—	—	—
	00100000	001	010	—	—	√	—	—	—	—	—
	00010000	001	011	—	—	—	√	—	—	—	—
	00001000	001	100	—	—	—	—	√	—	—	—
	00000100	001	101	—	—	—	—	—	√	—	—
	00000010	001	110	—	—	—	—	—	—	√	—
	00000001	001	111	—	—	—	—	—	—	—	√
Half Word	11000000	010	000	√	√	—	—	—	—	—	—
	01100000	010	001	—	√	√	—	—	—	—	—
	00110000	010	010	—	—	√	√	—	—	—	—
	00011000	010	011	—	—	—	√	√	—	—	—
	00001100	010	100	—	—	—	—	√	√	—	—
	00000110	010	101	—	—	—	—	—	√	√	—
	00000011	010	110	—	—	—	—	—	—	√	√
Triple byte	11100000	011	000	√	√	√	—	—	—	—	—
	00000111	011	101	—	—	—	—	—	—	—	—

Table 8-7. Data Transfers—64-Bit Mode (Continued)

Program Transfer Size	Bus BE0–BE7	Bus TSIZ0–TSIZ2	Bus A29–A31	Byte Lanes							
				D0...						...D63	
				0	1	2	3	4	5	6	7
Word	11110000	100	000	√	√	√	√	—	—	—	—
	01111000	100	001	—	√	√	√	√	—	—	—
	00111100	100	010	—	—	√	√	√	√	—	—
	00011110	100	011	—	—	—	√	√	√	√	—
	00001111	100	100	—	—	—	—	√	√	√	√
Double Word	11111111	000	000	√	√	√	√	√	√	√	√

√ Lanes that are read or written during that bus transaction
— Lanes that are ignored during read transactions and driven with undefined data during write transactions

The supported aligned transfers for 32-bit mode are shown in Table 8-8.

Table 8-8. Data Transfers—32-Bit Mode

Program Transfer Size	Bus BE0–BE7	Bus TSIZ0–TSIZ2	Bus A29–A31	Byte Lanes				Number of Beats
				DH0...			...DH31	
				0	1	2	3	
Byte	10000000	001	000	√	—	—	—	Single beat
	01000000	001	001	—	√	—	—	Single beat
	00100000	001	010	—	—	√	—	Single beat
	00010000	001	011	—	—	—	√	Single beat
	00001000	001	100	√	—	—	—	Single beat
	00000100	001	101	—	√	—	—	Single beat
	00000010	001	110	—	—	√	—	Single beat
	00000001	001	111	—	—	—	√	Single beat
Half Word	11000000	010	000	√	√	—	—	Single beat
	01100000	010	001	—	√	√	—	Single beat
	00110000	010	010	—	—	√	√	Single beat
	00011000	010	011	— / √	—	—	√ / —	Beat 1 / Beat 2
	00001100	010	100	√	√	—	—	Single beat
	00000110	010	101	—	√	√	—	Single beat
	00000011	010	110	—	—	√	√	Single beat

Table 8-8. Data Transfers—32-Bit Mode (Continued)

Program Transfer Size	Bus BE0–BE7	Bus TSIZ0–TSIZ2	Bus A29–A31	Byte Lanes DH0...	...DH31			Number of Beats
				0	1	2	3	
Triple byte	11100000	011	000	√	√	√	—	Single beat
	00000111	011	101	—	√	√	√	Single beat
Word	11110000	100	000	√	√	√	√	Single beat
	01111000	100	001	— √	√ —	√ —	√ —	Beat 1 Beat 2
	00111100	100	010	— √	— √	√ —	√ —	Beat 1 Beat 2
	00011110	100	011	— √	— √	— √	√ —	Beat 1 Beat 2
	00001111	100	100	√	√	√	√	Single beat
Double Word	11111111	000	000	√ √	√ √	√ √	√ √	Beat 1 Beat 2

√ Lanes that are read or written during that bus transaction
— Lanes that are ignored during read transactions and driven with undefined data during write transactions

Because the 602 has an on-chip, copy-back primary cache, most bus transactions issued by the 602 are double-word-aligned burst-read or burst-write operations. Only those nonburst transactions that bypass or miss in the cache (caching-inhibited and write-through transactions) generate alignment considerations on the bus.

Note that when a program generates a misaligned request that crosses a word boundary, two bus transactions may be required to serve the request, which may in turn encounter additional latency due to such factors as cache misses, page faults, or cacheability attributes,

8.3.2.2.4 Transfer Code
These attributes provide further descriptive information about the transaction. The transfer code signals (TC0–TC1) are encoded differently for read and write operations. Their encodings are shown in Table 8-9.

Table 8-9. Transfer Code Signal Encoding

TC0–TC1	Read	Write
00	Data transaction	Normal write
01	N/A	Copy-back line-fill
10	Instruction fetch	N/A
11	Reserved	Reserved

8.3.2.2.5 Address/Transfer Attribute Summary

Table 8-10 summarizes the address and transfer attribute information presented on the bus by the 602 for various processor or snoop-related transactions.

Table 8-10. Address/Transfer Attribute Summary

Bus Transaction	A0–A31	TT0–TT4	$\overline{\text{TBST}}$	TSIZ0–TSIZ2	$\overline{\text{WT}}$ $\overline{\text{CI}}$ $\overline{\text{GBL}}$	TC0–TC1
Instruction Fetch						
Cacheable	A0–A28‖xxx	0 1 1 1 0	0	Invalid	1 1 1	1 0
Caching-Inhibited	A0–A28‖000	0 1 0 1 0	1	0 0 0	1 0 1	1 0
Caching-Inhibited	A0–A28‖100	0 1 0 1 0	1	0 0 0	1 0 1	1 0
Cache Operations						
Line-fill (cache miss)	A0–A28‖xxx	0 1 1 1 0	0	0 1 0	¬W, 1 ¬M [1]	0 0
Castout	A0–A26‖00xxx	0 0 1 1 0	0	0 1 0	1 1 1	0 0
Snoop copy-back	A0–A26‖00xxx	0 0 1 1 0	0	0 1 0	1 1 1	0 0
Cache Bypass Operations						
Single-beat read (CI)	A0–A31	0 1 0 1 0	1	Size (see Table 7-4)	¬W, 0 ¬M	0 0
Single-beat write (CI)	A0–A31	0 0 0 1 0	1	Size (see Table 7-4)	¬W, 0 ¬M	0 0
Single-beat write (WT)	A0–A31	0 0 0 1 0	1	Size (see Table 7-4)	0 ¬I, ¬M	0 0
Special Instructions						
dcbz (address-only)	A0–A26‖00000	0 1 1 0 0	0	0 1 0	¬W, 1 0	0 0

[1] W,I,M = WIM bits from PTEs, BATs, or HID0; ¬ = Complement

Note that in Table 8-10, the $\overline{\text{WT}}$, $\overline{\text{CI}}$, and $\overline{\text{GBL}}$ signals correspond to the WIM bits, which are defined in the BATs (for block address translations), the PTEs (for page address translation), or in HID0 (for real addressing mode and protection-only mode).

8.3.2.3 Address Phase Termination

Two signals are used to terminate the address phase—$\overline{\text{AACK}}$ and $\overline{\text{ARTRY}}$.

The 602 does not terminate the address phase until $\overline{\text{AACK}}$ is asserted. As shown in the previous examples, in the fastest address cycle, the $\overline{\text{AACK}}$ signal can be asserted simultaneously with the $\overline{\text{TS}}$ cycle to result in a single-cycle address phase. In some systems, $\overline{\text{AACK}}$ can be connected low to minimize each address phase to one bus cycle. However, the system can use $\overline{\text{AACK}}$ to extend or pace the address phase.

After the address phase, $\overline{\text{TS}}$ always is driven high for one bus cycle. The $\overline{\text{AACK}}$ signal must be asserted for one bus cycle only.

The address phase can be terminated with requirement to rerun, or retry, if $\overline{\text{ARTRY}}$ is asserted during the address phase and through the cycle following $\overline{\text{AACK}}$ (see Figure 8-8). This causes the entire transaction—address and data phase—to be rerun. As a snooper, the 602 asserts $\overline{\text{ARTRY}}$ for a snooped transaction that hits modified data in the data cache and must be written to memory, or if the snooped transaction could not be serviced. As a bus master, the 602 responds to an assertion of $\overline{\text{ARTRY}}$ by aborting the bus transaction and re-requesting the bus, if $\overline{\text{BG}}$ is deasserted. Internally, the address queue that was retried is continually rearbitrated with the other internal queues until the next qualified bus grant is recognized.

If an address retry is required, the $\overline{\text{ARTRY}}$ response may be asserted by a snooper as early as the second cycle after the assertion of $\overline{\text{TS}}$. (the 602, however, may not assert $\overline{\text{ARTRY}}$ until the third cycle after $\overline{\text{TS}}$). Once asserted, $\overline{\text{ARTRY}}$ must remain asserted through the cycle after the assertion cycle 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 phase is referred to as an early $\overline{\text{ARTRY}}$. If $\overline{\text{AACK}}$ is connected low, $\overline{\text{ARTRY}}$ must be asserted on the second cycle (clock cycle 3) after the assertion of $\overline{\text{TS}}$.
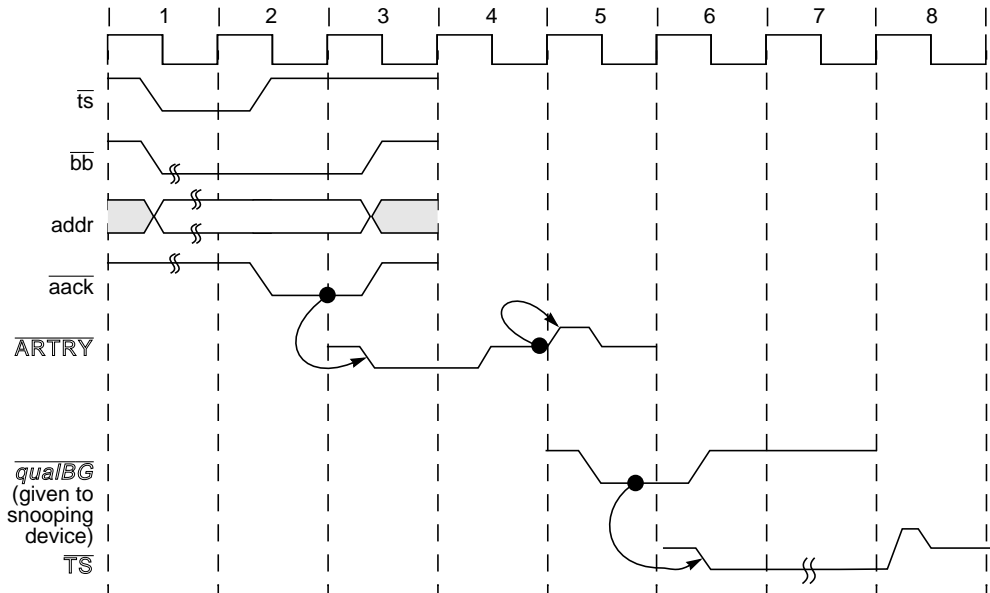
Figure 8-8. Snooped Address Cycle with $\overline{\text{ARTRY}}$

As a bus master, the 602 recognizes either an early or a qualified $\overline{\text{ARTRY}}$ and prevents the data phase associated with the retried address phase from beginning. During a qualified $\overline{\text{ARTRY}}$, the 602 also determines whether it should negate $\overline{\text{BR}}$ and ignore $\overline{\text{BG}}$ on the following cycle. The following cycle is the snooping window, during which only the device that asserted $\overline{\text{ARTRY}}$ can assert $\overline{\text{BR}}$. This guarantees the snooping device an opportunity to request and be granted the bus before the just-retried master can restart its transaction. During this window, $\overline{\text{BG}}$ is also blocked so a pipelined arbiter (one that clocks requests in and clocks grants out) has a chance to negate $\overline{\text{BG}}$ to an already granted potential bus master to perform a new arbitration.

8.3.3 Data Phase

After the address phase, the 602 asserts $\overline{\text{BB}}$, begins driving or sampling the data bus, and sampling the transfer acknowledge signal. The data phase consists of the data transfer itself and data termination.

8.3.3.1 Data Transfer

The data transfer signals are D0–D63. These signals form a 64-bit data path for read and write operations when the processor is in 64-bit mode ($\overline{\text{T32}}$ is negated); D0–D31 form a 32-bit data path when the processor is in 32-bit mode ($\overline{\text{T32}}$ is asserted). D32–D63 are ignored in 32-bit mode.

In 64-bit mode, the 602 transfers data in either single-beat (nonburst) transfers or four-beat burst transfers. Nonburst operations can transfer from one to eight bytes at a time and can

be misaligned. Burst transfers are used by the 602 to transfer cache blocks into or out of its internal cache. Nonburst transfers are either caching-inhibited or write-through write operations. For more information see Section 8.3.2.2, "Transfer Attributes."

8.3.3.2 Data Phase Termination

The following three signals are used to terminate the individual data beats of the data phase and the bus phase—$\overline{TA}$, $\overline{TEA}$, and $\overline{ARTRY}$:

- Asserting $\overline{TA}$ signals normal termination of a data beat or the transaction (last data beat of burst). 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.

- Asserting $\overline{TEA}$ signals a nonrecoverable error during the data transaction. It may be asserted on any cycle while $\overline{BB}$ is asserted. Asserting $\overline{TEA}$ terminates the bus phase immediately even if it is in the middle of a burst, however, it does not prevent incorrect data that has just been acknowledged with $\overline{TA}$ from being written into the 602's cache or register files. Asserting $\overline{TEA}$ causes either a machine check exception or a checkstop condition, depending on the setting of MSR[ME].

- Asserting $\overline{ARTRY}$ terminates the bus phase immediately. $\overline{ARTRY}$ is typically asserted in response to a device snooping and hitting an address on the bus. In such cases, asserting $\overline{ARTRY}$ delays the interrupted transaction so the snooping device can perform an operation (such as a write-back or castout operation) to ensure cache coherency.

Upon receiving a final or only termination condition, the 602 negates $\overline{BB}$ for at least one cycle.

8.3.3.3 Normal Single-Beat Termination

Normal termination of a single-beat data read operation occurs when $\overline{TA}$ is asserted by a responding slave; see Figure 8-9.
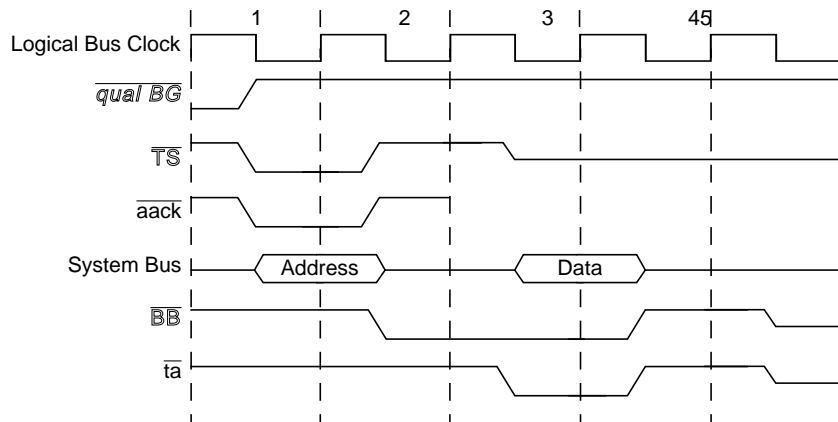
Figure 8-9. Normal Single-Beat Read Termination

Normal termination of a burst transfer occurs when $\overline{\text{TA}}$ is asserted for four bus clock cycles (in 64-bit mode), as shown in Figure 8-10. 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 or write bursts to terminate successfully, $\overline{\text{TEA}}$ must remain negated during the transfer. The only difference for a 32-bit mode is the number of data beats.
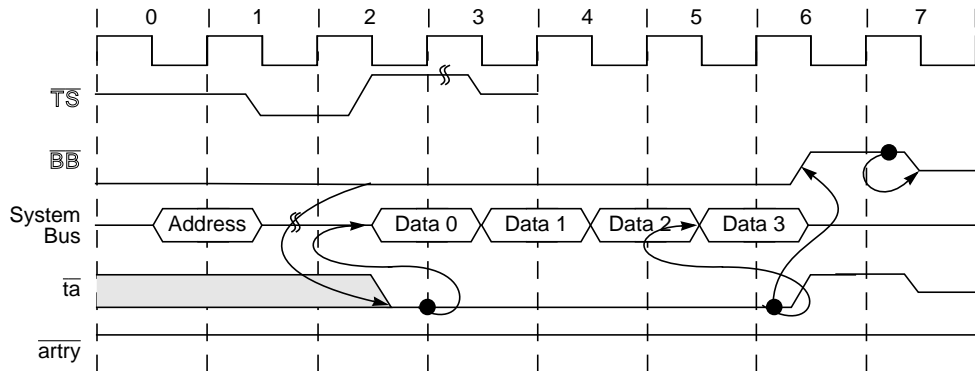
Figure 8-10. Normal Burst Transaction

8.4 Memory Coherency and Bus Protocol

The 602 has a copy-back cache which relies on bus snooping to maintain cache coherency in a uniprocessor system with coherent caches. The 602 implements a three-state MEI cache coherency protocol. The three-state cache coherency protocol is a subset of the four-state MESI protocol with the shared state not supported. For more details, see Chapter 7, "Signal Descriptions."

8.4.1 Effect on Read Operations

The three-state (MEI) cache coherency protocol of the 602 affects read operations on the bus in the following ways:

- 602 as bus master—All read operations (except for those that are caching-inhibited) are signalled on the bus as RWITM to force flushing of the cache block from other caches.

- 602 as snooper—All read operations snooped from the bus (except for those that are caching-inhibited) are interpreted as RWITM to cause flushing from the 602's cache. A caching-inhibited read is inferred by the 602 when the transaction is a nonburst read ($\overline{\text{TBST}}$ not asserted).

These actions for read operations allow the 602 to operate successfully on the bus with bus masters that support either MEI or MESI protocol. Table 8-3 summarizes the 602 signals used to maintain coherency.

8.4.2 Qualified Snoop Conditions

During the cycle that $\overline{\text{TS}}$ is asserted by another bus master, other 602s snoop bus transactions for any of the following conditions:

- Basic transfer protocol (signaled by $\overline{\text{TS}}$)

    — The global signal ($\overline{\text{GBL}}$) is asserted indicating that coherency enforcement is required.

    — A reservation is currently active in the 602 as the result of an **lwarx** instruction, and the snooped address indicates a write or kill operation in the transfer type attributes (TT0–TT4). These transactions are snooped regardless of whether $\overline{\text{GBL}}$ is asserted to support reservations in the three-state cache protocol.

- Injected snoop during read transaction (signaled by $\overline{\text{TS}}$ and $\overline{\text{TA}}$)

    While the 602 as a bus master performs a burst read transaction, the read target device can inject snoops by asserting $\overline{\text{TS}}$, negating $\overline{\text{TA}}$, and driving the snooped address on the bus. The window of the injected snoop is from the third cycle following the assertion of $\overline{\text{BB}}$ to the cycle before the last read data beat is transferred into the 602. If the injected snoop generates a hit, it asserts $\overline{\text{ARTRY}}$, but does not perform the snoop push operation. (The 602 does not perform clean or flush operations to the internal cache). For the 602 to perform a normal snoop, the read target device must regenerate the snooped address as a normal transaction (that is, read, write, or address-only transaction) after the current read transaction is completed. If the snoop hits for kill, the 602 invalidates the matched cache block.

    Timing for injected snoop operations is shown in Section 8.5.4.7, "Injected Snoop Timings."

8.4.3 Internal Snoop Sources

When a qualified snoop condition is detected on the bus, the snooped address associated with $\overline{\text{TS}}$ is compared against the data cache tags, memory queues, and any other appropriate memory elements as appropriate for the following conditions detected during the assertion of $\overline{\text{TS}}$:

- Data cache tags—The data cache tags are snooped for standard data cache coherency support. Instruction caches are not snooped.
- Reservation—This is for the **lwarx**/**stwcx.** instructions.

8.4.4 Reaction on Qualified Snoops

The 602 asserts $\overline{\text{ARTRY}}$ to the current bus master when a snooped transaction presents a cache or memory queue coherency problem to the 602 as described above. The assertion of $\overline{\text{ARTRY}}$ should signal the other master to abort its transaction and retry it later after the 602 can first perform a write operation back to memory. The 602 may also retry a bus transaction if the cache tags are being accessed (for example, when the LSU is accessing or updating the tags) prevent it from snooping the transaction on that cycle. Bus timing operations are shown in Section 8.5.4, "Snooping."

8.4.5 Special Instructions

The cache control, TLB management, and synchronization instructions supported by the 602 may affect or be affected by the operation of the bus. Only **dcbz** is actively broadcast through address-only transactions on the bus, and the 602 snoops only KILL operations broadcast by other masters. However, these instructions may indirectly initiate bus transactions, or their completion may be linked to the bus. Table 8-11 summarizes how these instructions may operate with respect to the bus.

Table 8-11. Bus Impact for Special Instructions

Instruction	Possible Bus Operation	Comment
sync	None	Allows queued bus operations (except instruction and touch load operations) to complete.
tlbie	None	—
tlbsync	None	—
eieio	None	No-op. The **eieio** instruction is not needed on the 602 because the caching-inhibited and write-through operations are performed in order.
icbi	None	—
dcbi	None	—
dcbf	Write-with-kill	Occurs only if cache block is modified
dcbst	Write-with-kill	Occurs only if cache block is modified
dcbz	Kill block (address-only)	Serves as broadcast to other masters for cache coherency; occurs only if the cache block is marked as global even if cache block matches and is modified.
	Write-with-kill	May occur as a result of normal cache replacement in case of a cache miss.
dcbt, **dcbtst**	Read-with-intent-to-modify	Fetched cache block is stored in the touch load queue; see Section 2.5.1.

Note: This table does not address the impact of WIM settings, nor does it completely describe the mechanisms for the operations described. It is intended only to show the possible bus relationships that may exist.

8.5 Bus Timing Examples

The timings in this section take into account the following variables.

- Length of the address phase—There are two types of address phases:
  - Single-cycle address phase—For these transactions, the address phase lasts only one bus clock cycle because $\overline{\text{AACK}}$ is asserted simultaneously with $\overline{\text{TS}}$. The $\overline{\text{AACK}}$ signal can be connected low to ensure a single-cycle address phase.
  - Multicycle address phase—This describes any address phase for which $\overline{\text{AACK}}$ is not asserted simultaneously with $\overline{\text{TS}}$.

- Burst or nonburst transfers—The different types of transactions supported depend on the amount of data to be transferred. This is often affected by the settings of the W, I, and M bits.
  - Burst transactions transfer an eight-word cache block between the on-chip cache and system memory. This can happen only when memory is addressed in memory space that is designated as cacheable.
  - Nonburst transactions transfer up to 64 bits of data between the processor and system memory. It should be pointed out that a single-beat transaction (transferring up to 64 bits) is the only nonburst transaction supported in 64-bit mode. There are two types of nonburst transactions in 32-bit mode—single-beat transactions that transfer up to 32 bits and double-beat transactions that transfer up to 64 bits.

- 64- or 32-bit data bus mode—Whether the bus operates as a 32- or 64-bit bus is determined by the setting of $\overline{\text{T32}}$. Bus width can be changed dynamically or the signal can be connected asserted for systems that use a static 32-bit bus or negated for systems that use a static 64-bit bus.
  - 64-bit—If $\overline{\text{T32}}$ is negated, the data bus behaves as a 64-bit bus. Burst transactions take four beats to transfer an eight-word cache block. Nonburst transactions are nonburst operations that can transfer up to 64 bits of data. Timings for basic 32-bit mode operations are described in Section 8.5.1, "64-Bit Data Bus Mode Basic Transactions."
  - 32-bit—If $\overline{\text{T32}}$ is asserted, the bus operates as a 32-bit bus. Burst transactions take eight beats of 32 bits each to transfer an eight-word cache block. Nonburst transactions consist of single- or double-beat operations that can transfer up to 64 bits of data. Timings for basic 32-bit mode operations are described in Section 8.5.2, "32-Bit Bus Mode Basic Transactions."

- Whether wait states are inserted between bursts—Each beat of data must be acknowledged by the assertion of the $\overline{\text{TA}}$ signal. In a multiple-beat transaction (a four-beat burst or a double-beat nonburst operation), the next beat is delayed if the $\overline{\text{TA}}$ is negated between beats. An example of a data transaction that has wait states can be seen in Section 8.5.1.2, "Burst Read Transaction with a Single-Cycle Address Phase—64-Bit Mode."

The $\overline{\text{TA}}$ signal can remain asserted during the transaction to indicate that there is no need to introduce wait states. In the following examples, this is referred to as "fastest data transaction." An example of a data transaction with no wait states is shown in Section 8.5.1.6, "Burst Write Transaction—64-Bit Mode."

- Whether the transaction is a read, write, or address-only transaction

8.5.1 64-Bit Data Bus Mode Basic Transactions

This section presents basic bus transactions when the processor is in 64-bit mode. They include the following:

- Nonburst read transaction with single-cycle address phase (Section 8.5.1.1)
- Burst read transaction with a single-cycle address phase (Section 8.5.1.2)
- Burst read transaction with a multicycle address phase (Section 8.5.1.4)
- Nonburst write transaction (Section 8.5.1.5)
- Burst write transaction with a single-cycle address phase (Section 8.5.1.6)
- Slower burst write transaction (Section 8.5.1.7)

These examples show many characteristics that are common to 32-bit mode transactions as well, such as the timing for single- and multicycle address phases and how wait states can be inserted between beats of a burst operation by asserting and negating $\overline{\text{TA}}$.

8.5.1.1 Nonburst Read Transaction—64-Bit Mode

Figure 8-11 shows a nonburst read operation with the single-cycle address phase. Note that this operation is the same regardless of whether the bus is in 32- or 64-bit mode. Note that to transfer 64 bits of data in 32-bit mode, a double-beat transaction is necessary. This is shown in Figure 8-19.
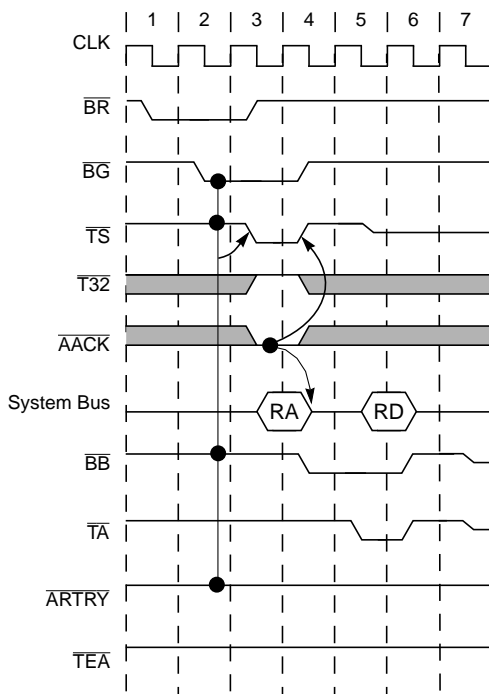
Figure 8-11. Nonburst Read Transaction, Single-Cycle Address Phase—64-Bit Mode

The signal interactions are as follows:

1. In clock cycle 1, the bus request is issued ($\overline{BR}$ is asserted).

2. In clock cycle 2, the bus is granted ($\overline{BG}$ is asserted). In this example, there is no other activity on the bus; $\overline{BB}$, $\overline{TS}$, and $\overline{ARTRY}$ signals are sampled as negated, so the 602 receives a qualified bus grant.

3. In clock cycle 3, transfer start ($\overline{TS}$) is asserted and the address is made available on the bus. The $\overline{AACK}$ signal, which is connected asserted, is sampled. Because it is always asserted, the address phase is guaranteed to be the single-cycle address phase (one bus clock cycle).

4. In clock cycle 4, $\overline{BB}$ is asserted indicating the start of the data phase.

5. In clock cycle 5, the slave device makes the data available on the bus and the transfer is acknowledged ($\overline{TA}$ is asserted).

6. In clock cycle 6, the data transfer completes, and the $\overline{TA}$ and $\overline{BB}$ are negated.

8.5.1.2 Burst Read Transaction with a Single-Cycle Address Phase— 64-Bit Mode

Figure 8-12 shows a burst read operation with the single-cycle address with the bus operating as a 64-bit bus. This example is identical to the nonburst read example shown in Section 8.5.1.1, "Nonburst Read Transaction—64-Bit Mode." However, here a four-beat burst operation occurs that updates a cache block.
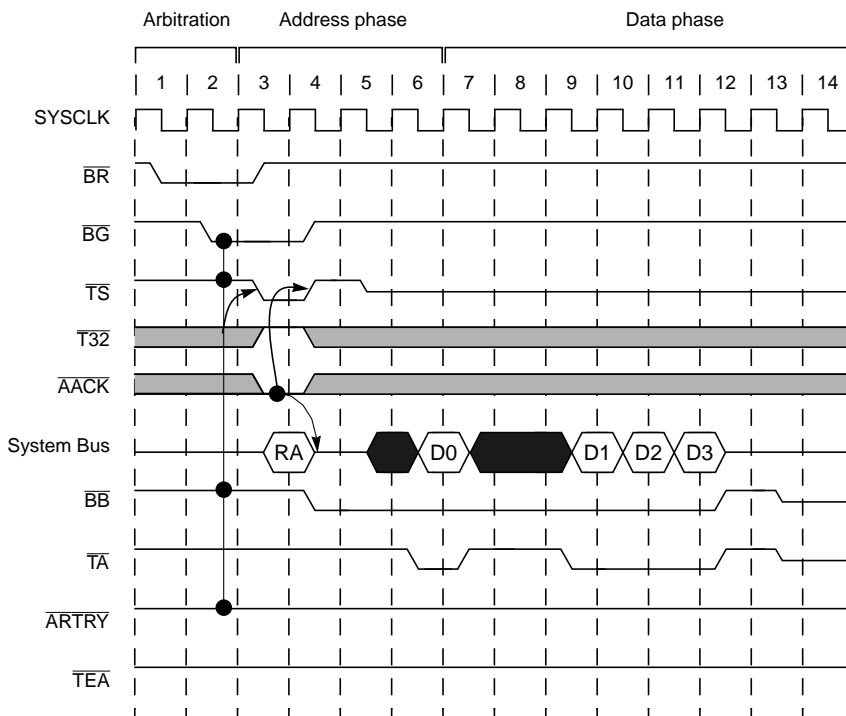
Figure 8-12. Burst Read Transaction with a Single-Cycle Address Phase—64-Bit Mode

The signal interactions are as follows:

1. Clock cycles 1–4 show the single-cycle address phase and are identical to the nonburst case described in Section 8.5.1.1, "Nonburst Read Transaction—64-Bit Mode."

2. In clock cycle 5, $\overline{\text{TS}}$ is three-stated and the slave device does not yet drive the data bus.

3. In clock cycle 6, the slave device drives data beat 0 on the memory bus; meanwhile, the $\overline{\text{TA}}$ signal is asserted.

4. In clock cycle 7, the slave device is not yet driving the next data beat and the $\overline{\text{TA}}$ signal is negated and remains so through clock cycle 8.

5. In clock cycle 9, the slave device drives data beat 2 on the memory bus while $\overline{\text{TA}}$ is reasserted.

6. $\overline{\text{TA}}$ remains asserted through clock cycles 9–11, while data beats 1–3 are transferred.

7. In clock cycle 12, the final data beat arrives and $\overline{\text{TA}}$ is negated.

8.5.1.3 Burst Read Transaction with a Single-Cycle Address Phase/Shortest Data Phase—64-Bit Mode

Table 8-7 shows a burst read transaction in which the four data beats occur without interruption. In this example $\overline{\text{TA}}$ remains asserted throughout the data transfer.
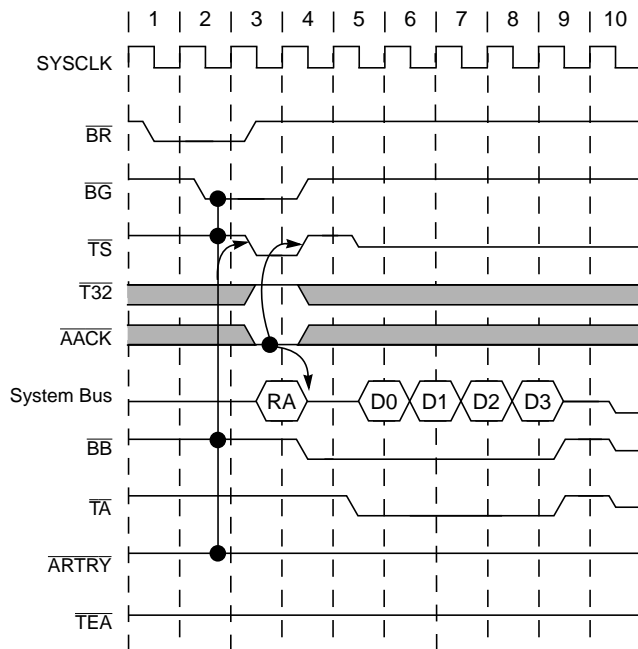
Figure 8-13. Burst Read Transaction with a Single-Cycle Address Phase/Shortest Data Phase—64-Bit Mode

8.5.1.4 Burst Read Transaction with a Multicycle Address Phase—64-Bit Mode

Figure 8-14 shows a burst-read transaction with a multicycle address phase.
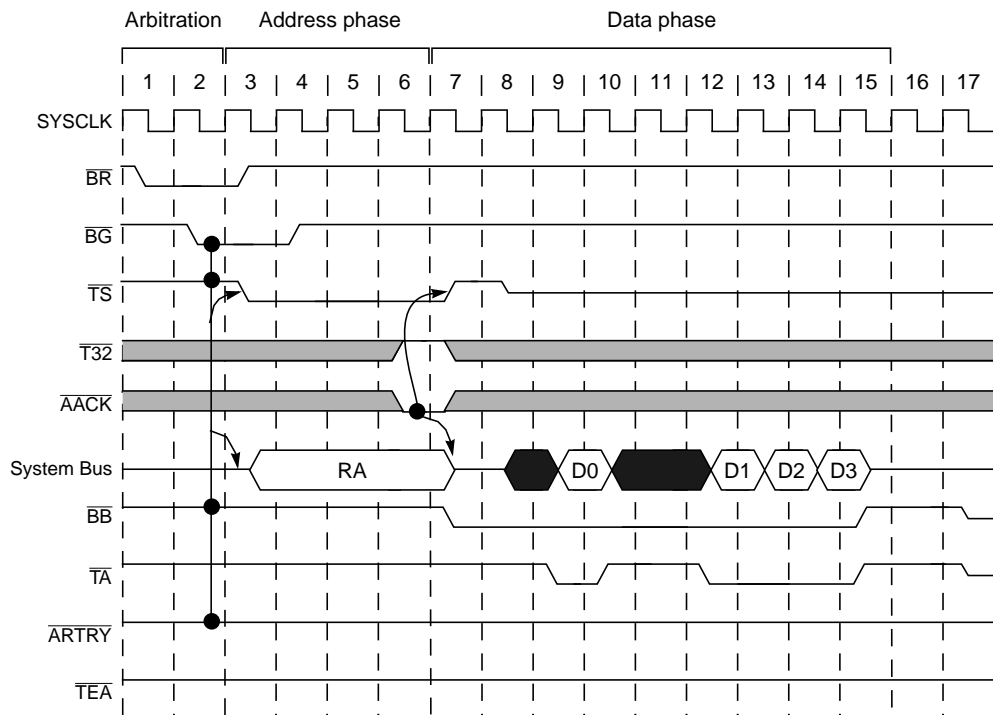
Figure 8-14. Burst Read Transaction with a Multicycle Address Phase—64-Bit Mode

In this example, it takes four clock cycles for the address to be transferred on the bus. In this example, the four data beats are paced by the $\overline{TA}$ signal, but this is independent of the length of the data phase.

8.5.1.5 Nonburst Write Transaction—64-Bit Mode

Figure 8-15 illustrates a nonburst write transaction with the single-cycle address phase in 64-bit mode. Note the similarities with Section 8.5.1.1, "Nonburst Read Transaction—64-Bit Mode." The essential difference between these transactions is that the memory bus is not put into high-impedance for the clock cycle after the address is transferred (clock cycle 4).
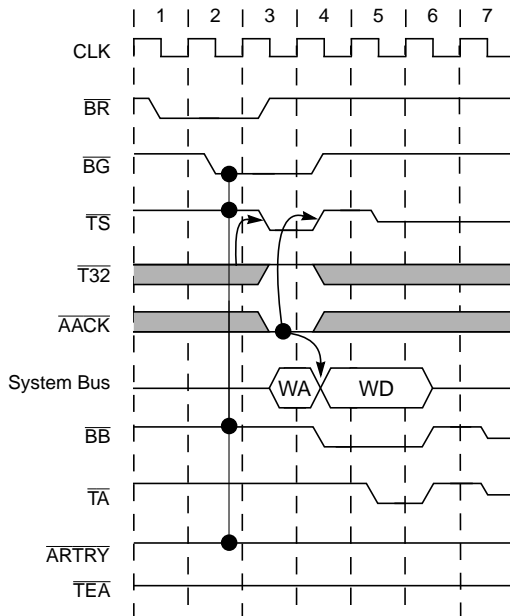
Figure 8-15. Fastest Nonburst Write Transaction—64-Bit Mode

The signal interactions are as follows:

1. Clock cycles 1–3 are identical to the nonburst read example in Figure 8-11, except the address transfer carries a write address (WA) rather than a read address.

2. The essential difference between read and write operations is seen in clock cycle 4. The memory bus does not need to put in high-impedance, and the write data becomes available immediately after the address is transferred. Likewise the $\overline{\text{BB}}$ signal is asserted after the address is transferred.

3. As in the nonburst read example, in clock cycle 5, $\overline{\text{TA}}$ is asserted to end the transfer.

4. Clock cycles 6 and 7 are the same as the burst read example.

8.5.1.6 Burst Write Transaction—64-Bit Mode

Figure 8-16 shows a simple burst write transaction with single-cycle address phase and with no wait states between data beats.
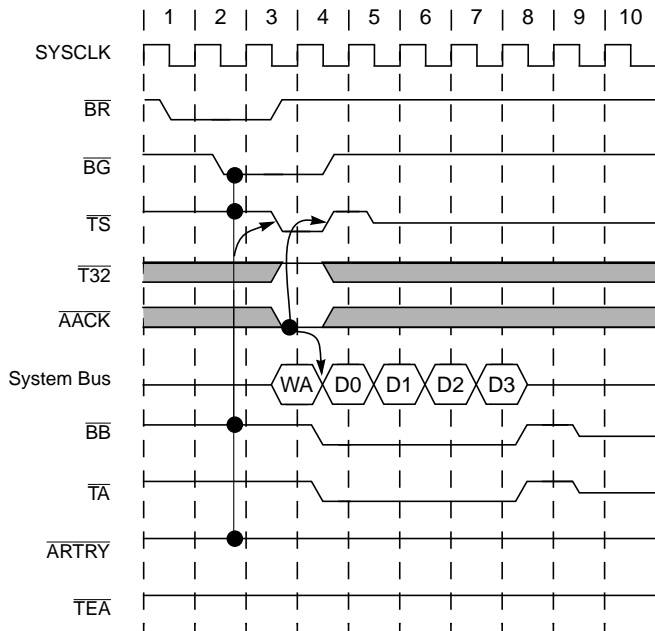
Note: This transaction should be used only when $\overline{\text{ARTRY}}$ is not asserted.

Figure 8-16. Fastest Burst Write Transaction with Negated $\overline{\text{GBL}}$ Signal (Single-Cycle Address Phase)—64-Bit Mode

The signal interactions are as follows:

1. Clock cycles 1–3 are identical to those in Section 8.5.1.6, "Burst Write Transaction—64-Bit Mode."

2. Because $\overline{\text{GBL}}$ is negated, the transaction is not snooped by other devices and $\overline{\text{TA}}$ can be asserted simultaneously with $\overline{\text{BB}}$ in clock cycle 4.

3. The $\overline{\text{BB}}$ and $\overline{\text{TA}}$ signals remain asserted while the four data beats are transferred in clock cycles 4–8.

4. After the last data beat in clock cycle 8, $\overline{\text{BB}}$ and $\overline{\text{TA}}$ are negated, after which the memory bus is put in high-impedance state and the timing behavior continues as in the previous examples.

In this example, snooping is not required, therefore it can be assumed that the $\overline{\text{GBL}}$ signal is not asserted. This allows the $\overline{\text{TA}}$ signal to be asserted in the same clock as $\overline{\text{BB}}$, immediately after the address is transferred on the memory bus. For an example showing how this transaction is performed when the $\overline{\text{GBL}}$ signal is asserted, see Section 8.5.4, "Snooping."

8.5.1.7 Slower Burst Write Transaction—64-Bit Mode

In previous examples showing burst transactions, the $\overline{\text{TA}}$ signal has remained asserted throughout the data beats, eliminating potential wait states. Figure 8-17 shows a burst write transaction in which the $\overline{\text{TA}}$ signal is not held asserted throughout the four-beat data transfer.
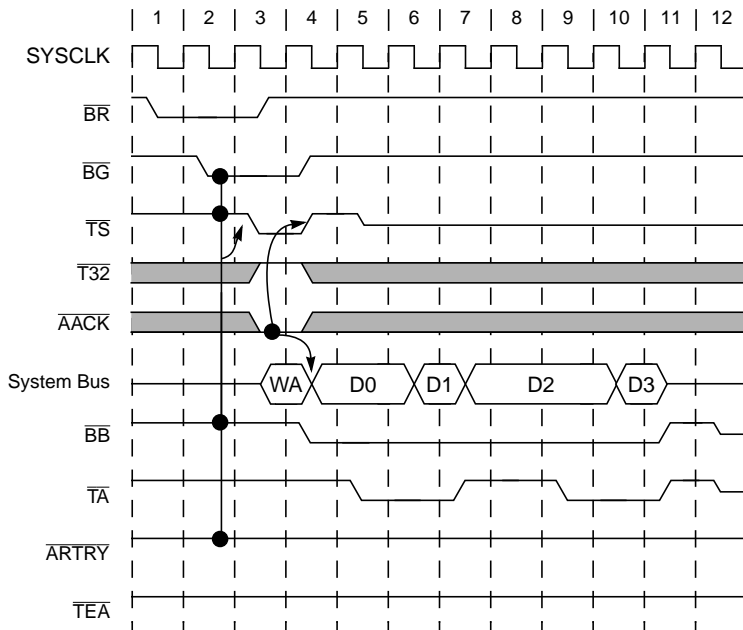
Figure 8-17. Slow Burst Write Transaction

The signal interactions are as follows:

1. Clock cycles 1–3 show the standard timing for a single-cycle address phase transaction.

2. In clock cycle 4, data is available on the memory bus (D0), but cannot complete until clock cycle 6 because $\overline{\text{TA}}$ is not asserted simultaneously with $\overline{\text{BB}}$.

3. The $\overline{\text{TA}}$ signal remains asserted through clock cycle 6 (allowing data beat 1 to complete), but is negated in clock cycle 7, which prolongs data beat 2.

4. The $\overline{\text{TA}}$ signal is asserted in clock cycle 9, allowing the D2 to complete on the following clock cycle (10).

5. The $\overline{\text{TA}}$ signal remains asserted allowing D3 to complete in clock cycle 11 with no additional wait states.

6. After D3 is transferred, the transaction concludes as normal.

8.5.2 32-Bit Bus Mode Basic Transactions

This section describes basic read and write operations when the 602 is operating in 32-bit mode. Many details that are common between 32-bit and 64-bit mode transactions, such as the timing for the single- and multicycle address phases, are illustrated in the previous examples and are not repeated here.

The examples illustrate the following:

- Single-beat read transactions (Section 8.5.2.1)
- Double-beat read transactions (Section 8.5.2.2)
- Burst read operations (Section 8.5.2.3)
- Burst read transaction with a multicycle address phase (Section 8.5.2.4)
- Write transactions in 32-bit mode (Section 8.5.2.5)

8.5.2.1 Single-Beat Read Transactions—32-Bit Only

A single-beat read transaction in 3-bit mode is shown in Figure 8-18. In this example, up to 32 bits are transferred in the data beat in clock cycle 5. The timing differs from the 64-bit nonburst operation (shown in Figure 8-11) only in that the $\overline{T32}$ pin is asserted (low).
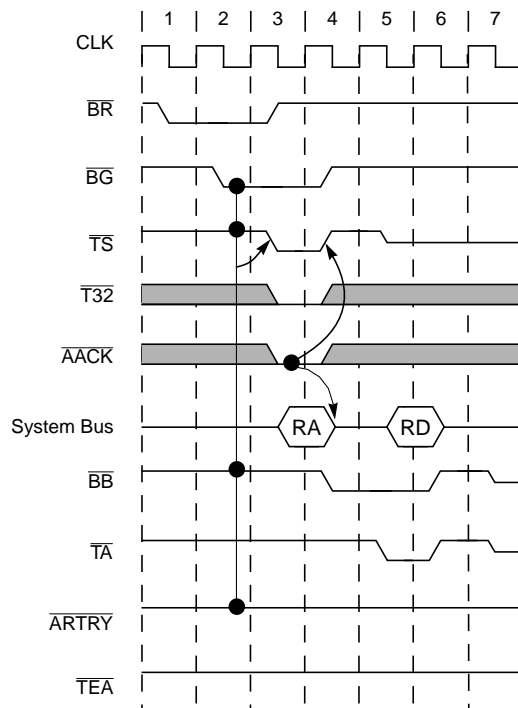
Figure 8-18. Single-Beat Read Transactions—32-Bit Only

8.5.2.2 Double-Beat Read Transactions—32-Bit Only

When the processor data bus is operating in 32-bit mode, it takes more than one beat to transfer double-word data types (for example, double-precision floating-point operands). For these situations, the 602 generates a two-beat memory access. Note that this does not require an additional address phase.
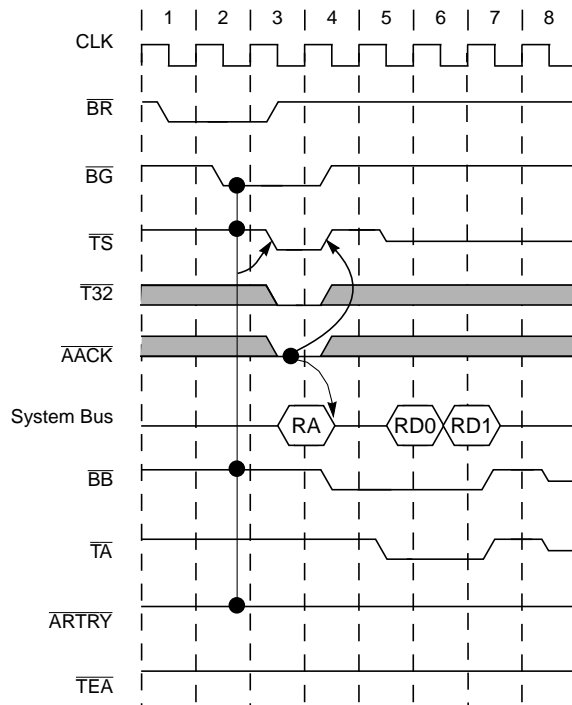
Figure 8-19. Double-Beat Read Transactions—32-Bit Only

The signal interactions are as follows:

1. Clock cycles 1 and 2 are like those in Section 8.5.1.1, "Nonburst Read Transaction—64-Bit Mode."

2. In clock cycle 3, the 602 asserts $\overline{\text{TS}}$ and drives the address on the bus. During this clock cycle, the 602 samples the $\overline{\text{T32}}$ and $\overline{\text{AACK}}$ signals. $\overline{\text{AACK}}$ is connected asserted guaranteeing a single-cycle address phase (one bus clock cycle). $\overline{\text{T32}}$ is asserted and is latched when $\overline{\text{AACK}}$ is sampled. Asserting $\overline{\text{T32}}$ indicates that the slave is a 32-bit device; and ensures that the bus will function as a 32-bit data bus in the data phase.

3. As in the 64-bit example, in clock cycle 4, the bus busy signal, $\overline{\text{BB}}$, is asserted.

4. In clock cycle 5, the first 32-bit half of the data (RD0) is made available on the bus and the transfer is acknowledged ($\overline{\text{TA}}$ is asserted). Note that $\overline{\text{TA}}$ cannot be asserted while the address/data bus are in high-impedance state.

5. In clock cycle 6, the second word (RD0) of data is transferred and the $\overline{\text{TA}}$ and $\overline{\text{BB}}$ signals remain asserted. Note that as with other burst operations, here the $\overline{\text{TA}}$ signal can be held asserted through the duration of the transfer or it can be alternately asserted and negated to pace the data beats.

6. In clock cycle 7, the second beat of data is transferred and the $\overline{\text{TA}}$ and $\overline{\text{BB}}$ signals are negated.

8.5.2.3 Burst Read Operations—32-Bit

Figure 8-20 shows a burst read operation with the single-cycle address with the bus operating as a 64-bit bus. This example is identical to the nonburst read example shown in Section 8.5.1.2, "Burst Read Transaction with a Single-Cycle Address Phase—64-Bit Mode." However, since the data bus is half as wide when it operates in 32-bit mode, an eight-beat burst operation (32-bits per beat) is required to update the cache block.
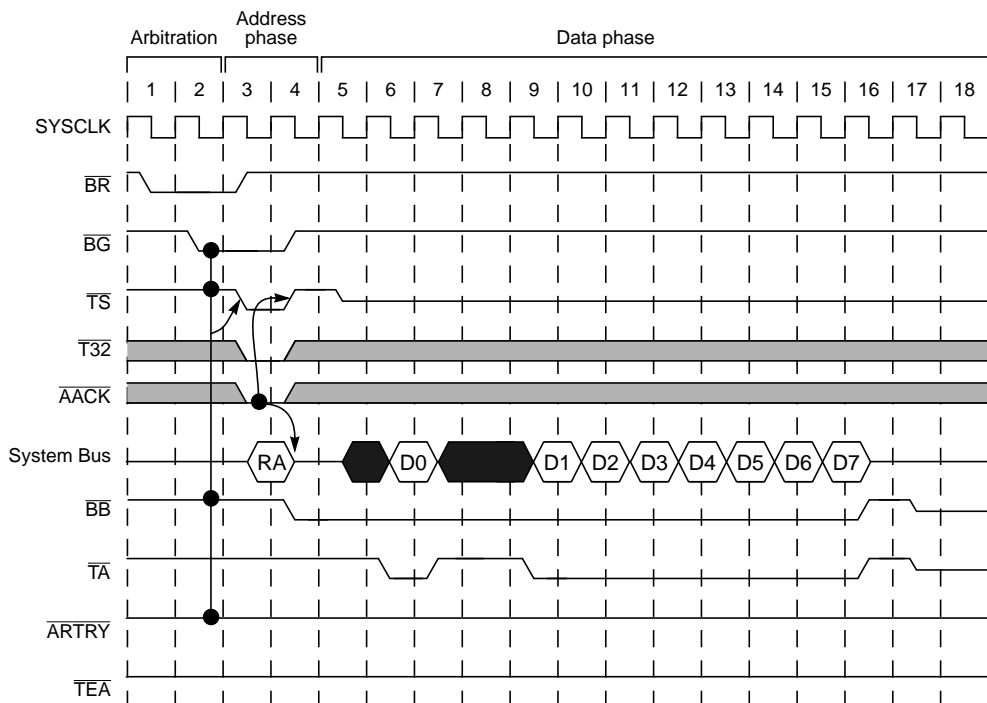
Figure 8-20. Burst Read Transaction with a Single-Cycle Address Phase—32-Bit

The 32-bit burst read transaction differs from the 64-bit burst read transaction primarily in that it requires eight data beats instead of four. $\overline{\text{TA}}$ must be asserted for each of these beats.

8.5.2.4 Burst Read Transaction with a Multicycle Address Phase— 32-Bit Mode

Figure 8-21 shows a burst-read transaction with a multicycle address phase and a data phase in which wait states are inserted. The timing for this example is identical to the 64-bit example in Figure 8-13, except for the fact that it requires four additional beats.
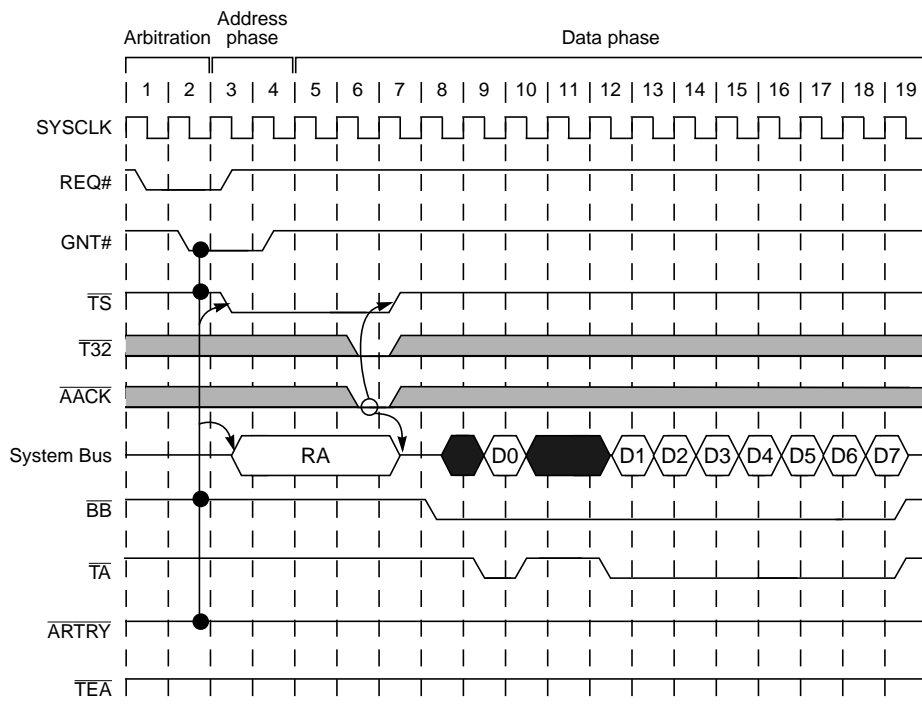
Figure 8-21. Burst Read Transaction with a Multicycle Address Phase—32-Bit Mode

8.5.2.5 Write Transactions in 32-Bit Mode

Tis section provides examples of write transactions in 32-bit mode, including the following:

- Fastest single-beat write transaction (Section 8.5.2.5.1)
- Fastest double-beat write transaction (32-bit mode only) (Section 8.5.2.5.2)
- Fastest burst write transaction (Section 8.5.2.5.3)

All three of these examples use a single-cycle address phase, and the two multiple-beat transactions have no wait states.

8.5.2.5.1 Fastest Single-Beat Write Transaction—32-Bit Mode

In Figure 8-22, the 602 performs the fastest single-beat write operation.
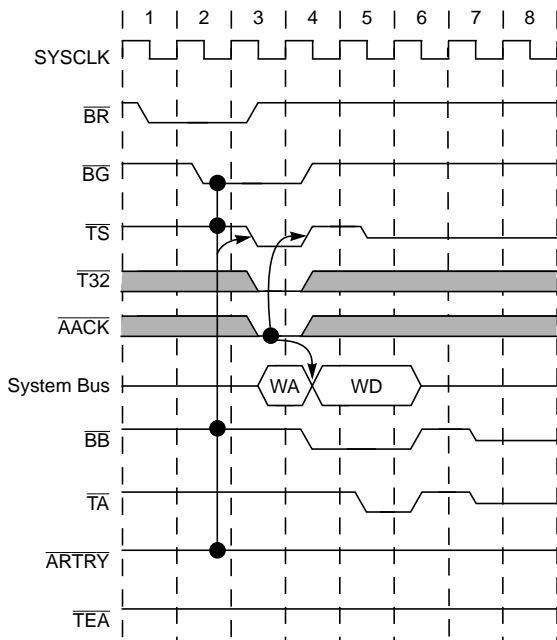
Figure 8-22. Fastest Single-Beat Write Transaction—32-Bit Mode

Two clock cycles are required to transmit the word of data because, as with single-beat transactions in 64-bit mode described in Section 8.5.1.5, "Nonburst Write Transaction—64-Bit Mode," $\overline{TA}$ cannot be asserted until the second clock cycle after the write address is transmitted.

8.5.2.5.2 Fastest Double-Beat Write Transaction—32-Bit Mode Only

Figure 8-23 shows a double-beat write transaction with no wait states between the two data beats. This transaction is supported only in 32-bit mode.
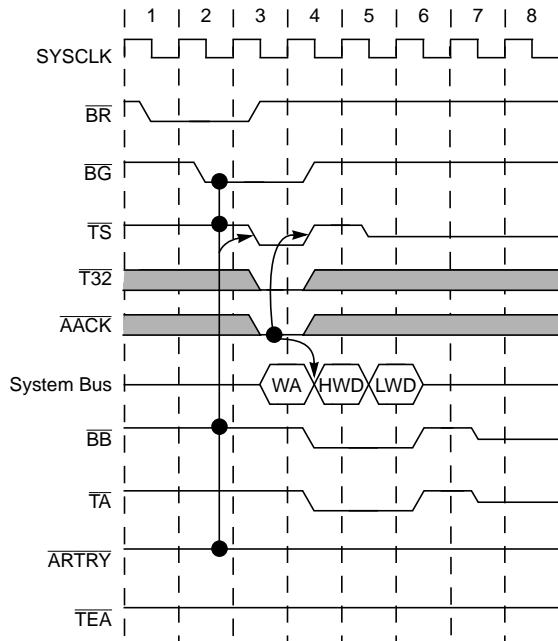
Figure 8-23. Fastest Double-Beat Write Transaction—32-Bit Mode

This example differs from Figure 8-24 in that $\overline{\text{TA}}$ can be asserted a clock cycle earlier than in the single-beat write case (as is also the case with multiple-beat (burst) operations in 64-bit mode).

8.5.2.5.3 Fastest Burst Write Transaction—32-Bit Mode

Figure 8-24 shows the fastest possible burst write transaction in 32-bit mode.
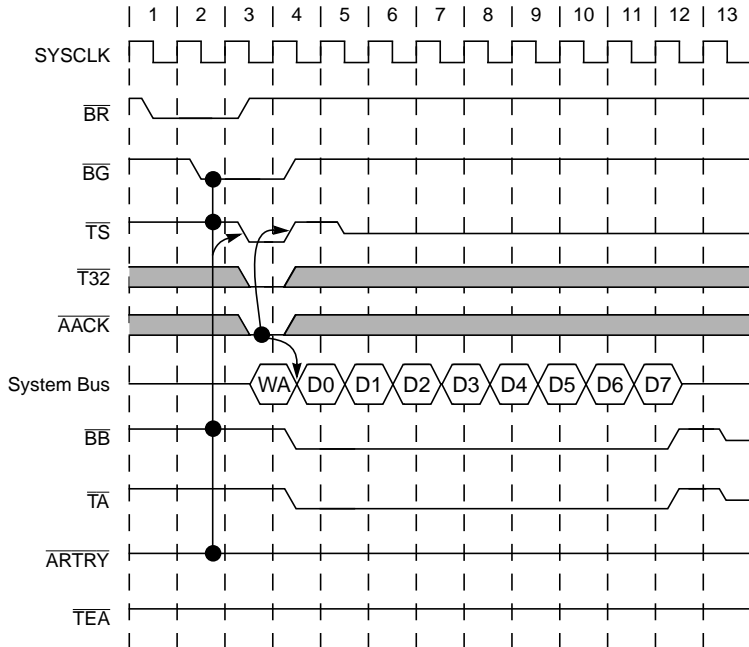
Figure 8-24. Fastest Burst Write Transaction—32-Bit Mode

This example shows the single-cycle address phase, and since this is a write transaction, the first data beat can be made available on the next beat. In the fastest burst write transaction, there are no wait states inserted between data beats. In this case the $\overline{\text{TA}}$ signal remains asserted throughout while all eight data beats are transferred.

8.5.3 Consecutive Operations

Previous examples have shown the timings for basic read and write operations in 32- and 64-bit mode. The examples in this section show the latency that can be encountered between transactions

8.5.3.1 Consecutive Nonburst Write-Read Transaction

Figure 8-25 shows the bus timing for a nonburst write followed by a nonburst read.
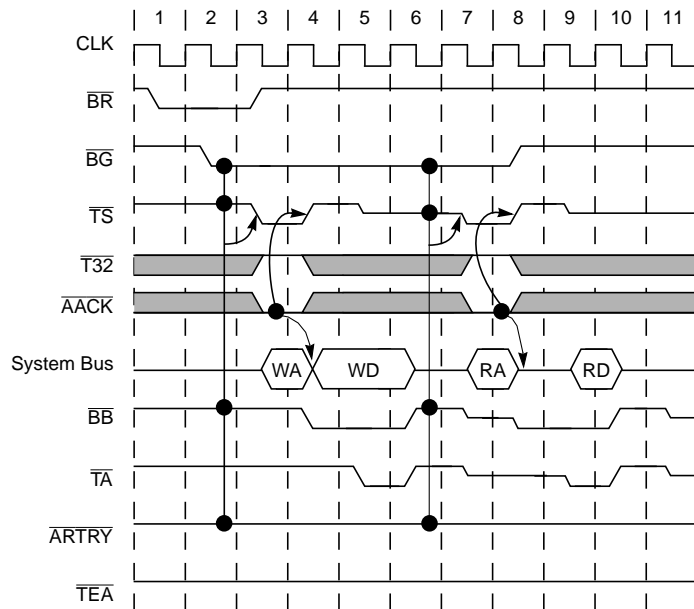
Figure 8-25. Consecutive Nonburst Write-Read Transaction

The signal interactions are as follows:

1. Clock cycles 1–3 show the timing for a single-cycle address phase operation.

2. Clock cycles 1–6 are identical to the nonburst write operation shown in Section 8.5.1.5, "Nonburst Write Transaction—64-Bit Mode" except that the $\overline{BG}$ signal remains asserted so the bus remains granted after the write operation. An additional bus request is not required because the bus is parked.

3. In clock cycle 6, $\overline{BB}$ is sampled. Because it is not asserted, the subsequent read operation can process without the 602 having to rearbitrate for the bus.

4. In clock cycle 7, the read address is supplied to the memory bus and a nonburst read operation identical to that described in Section 8.5.1.1, "Nonburst Read Transaction—64-Bit Mode."

8.5.3.2 Consecutive Nonburst Read-Write Transaction

Figure 8-26 shows a nonburst read transaction followed by a nonburst write transaction.
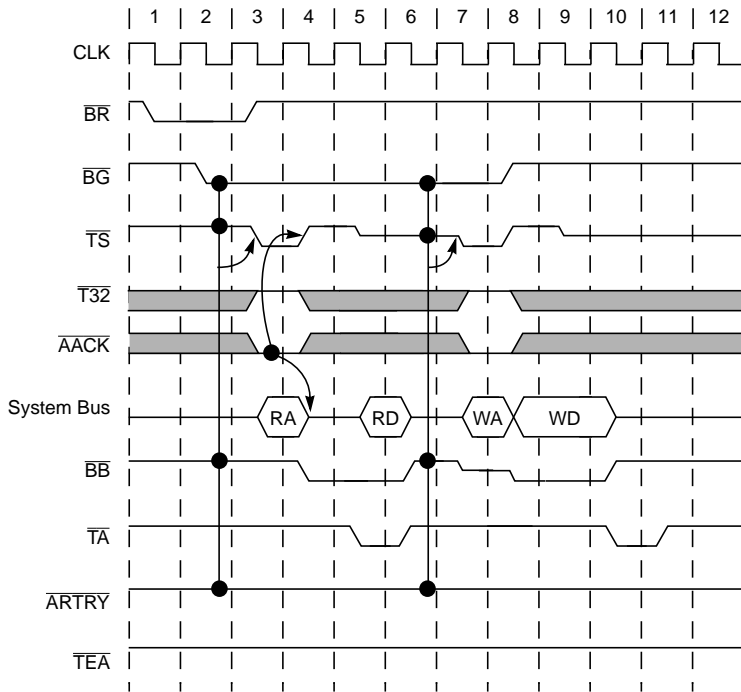
Figure 8-26. Consecutive Nonburst Read-Write Transaction

The signal interactions are as follows:

1. Clock cycles 1–6 are identical to the nonburst read example shown in Section 8.5.1.1, "Nonburst Read Transaction—64-Bit Mode," again except for the fact that the bus remains parked ($\overline{BG}$ remains asserted).

2. In clock cycle 6, the $\overline{BB}$ signal is sampled because it is not asserted; the 602 is free to begin the subsequent write transaction without having to rearbitrate for the bus.

3. In clock cycle 7, a nonburst write transaction like the one described in Section 8.5.1.5, "Nonburst Write Transaction—64-Bit Mode," begins.

4. The nonburst write operation proceeds as described in Section 8.5.1.5, "Nonburst Write Transaction—64-Bit Mode."

8.5.3.3 Consecutive Burst Write-Read Transaction

Similar to the previous examples showing consecutive nonburst operations, the following example shows a burst write operation followed by a burst read operation. Note that once again that the $\overline{BG}$ input remains asserted so the 602 does not need to assert $\overline{BR}$.
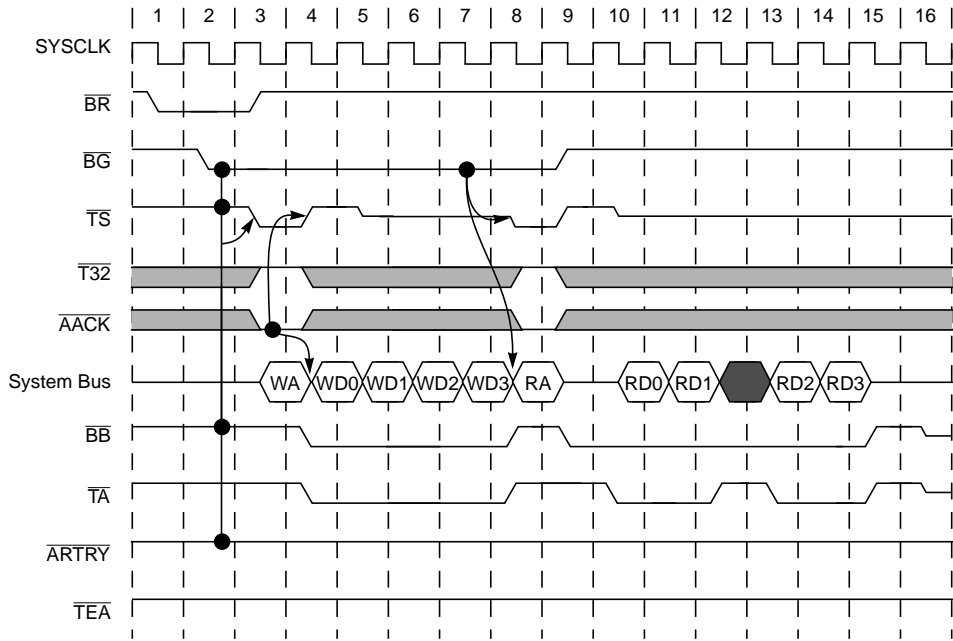
Figure 8-27. Consecutive Burst Write-Read Transaction

The signal interactions are as follows:

1. Clock cycles 1–7 are identical to the burst write transaction described in Section 8.5.1.6, "Burst Write Transaction—64-Bit Mode," except that the $\overline{BG}$ signal remains asserted so the 602 does not need to rearbitrate for the burst read operation that follows.

2. The read address is made available in clock cycle 7, and like the example shown in Section 8.5.1.2, "Burst Read Transaction with a Single-Cycle Address Phase—64-Bit Mode," there is a single-cycle pause before the first data beat is available on the memory bus.

3. The $\overline{TA}$ signal remains asserted for two clock cycles, so the first two data beats are transferred without interruption.

4. The $\overline{TA}$ signal is negated in clock cycle 12, which causes a wait state to be inserted in the data transfer.

5. The $\overline{TA}$ signal is asserted again in clock cycle 13, which allows the two remaining data beats to complete. The transaction completes as normal.

8.5.3.4 Consecutive Burst Read-Write Transaction

In this example, a burst write operation follows a burst read. Note again in this case the bus arbiter allows the $\overline{BG}$ signals to remain asserted throughout the first transactions so additional arbitration is not required for the write transaction.
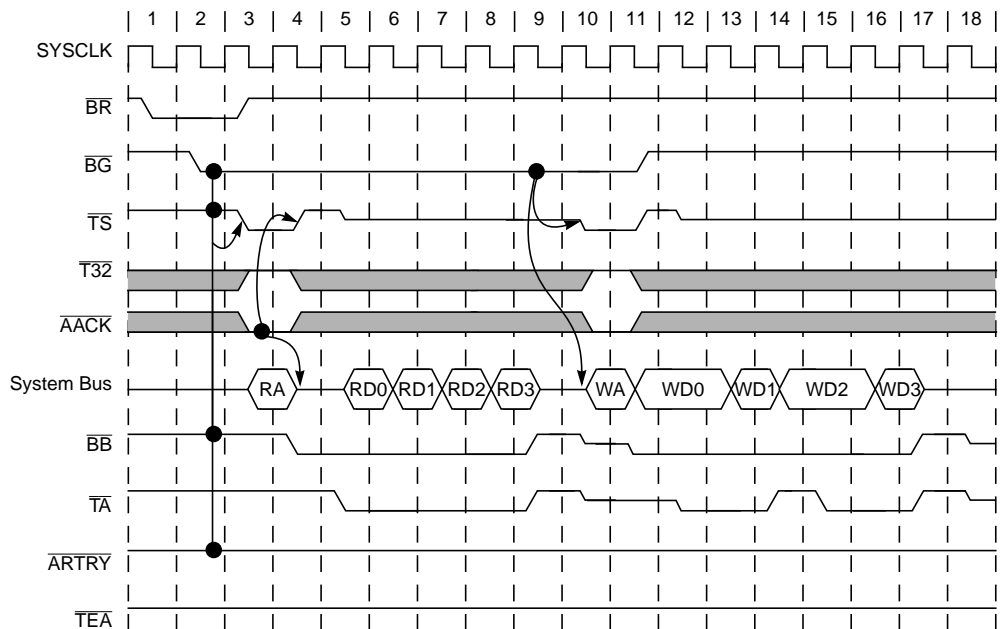
Figure 8-28. Consecutive Burst Read-Write Transaction

In Figure 8-28 the burst read transaction is the fastest possible burst read operation—that is, the $\overline{\text{TA}}$ signal remains asserted so there are no wait states between data beats. In the burst write transaction shown, data beats, WD0 and WD2, are prolonged by the negation of the $\overline{\text{TA}}$ signal.

8.5.4 Snooping

This section describes bus timing for systems that use multiple caching devices. Such systems must ensure cache coherency by snooping read and write addresses transferred on the bus, checking those addresses against regions of memory that are configured as cacheable, and responding when one of these addresses matches (a snoop hit occurs). Ensuring cache coherency affects the timing illustrated in the previous examples in the following ways:

- For an address to be snooped, the $\overline{\text{GBL}}$ signal must be asserted, and this causes additional bus activity, as shown in Section 8.5.4.1, "Fastest Burst Write Transaction with Asserted GBL Signal."

- When a snoop hit occurs, the operation that corresponds with the address that was broadcast must not be allowed to complete until the snooping device performs the bus operations necessary for it to maintain cache coherency. The snooped bus operation is interrupted by the assertion of the $\overline{\text{ARTRY}}$ signal, as described in Section 8.5.4.2, "Address Retry During 602 Read Transaction—Single-Cycle

Address Phase," Section 8.5.4.4, "ARTRY During Other Master Read Transaction—Single-Cycle Address Phase," Section 8.5.4.3, "Address Retry During 602 Read Transaction—Multicycle Address Phase," and Section 8.5.4.5, "ARTRY During Other Master Read Transaction—Multicycle Address Phase."

- Typically, the response to a snoop hit is to write back modified data to system memory. An example of this is shown in Section 8.5.4.6, "Snoop Hit—Write-Back Transaction."

- In addition to the address snooping mechanism described above, the 602 supports injected snoop operations that can occur between data beats of a burst operation in either 32- or 64-bit mode. Several examples of this are shown in Section 8.5.4.7, "Injected Snoop Timings."

Snooping requires additional clock cycles; a performance improvement can be gained by asserting $\overline{GBL}$ only unless snooping is required. This is shown in the following two examples.

8.5.4.1 Fastest Burst Write Transaction with Asserted $\overline{GBL}$ Signal

Figure 8-29 shows the same transaction as shown in Section 8.5.1.6, "Burst Write Transaction—64-Bit Mode."
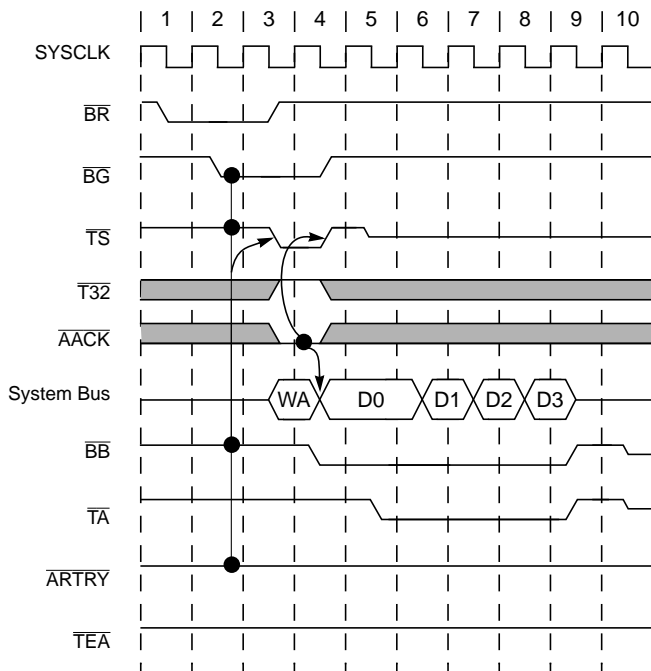
Figure 8-29. Fastest Burst Write Transaction with Asserted $\overline{GBL}$ Signal

The primary difference between these two examples can be seen in clock cycle 4. Because $\overline{\text{GBL}}$ is asserted, snooping must occur, and although data becomes available in the second half of clock cycle 4 immediately after the address is transferred, the $\overline{\text{TA}}$ signal cannot be asserted until one clock cycle later. This forces the first beat of data (D0) to take two cycles to transfer instead of one cycle when $\overline{\text{GBL}}$ is negated.

While this example shows the effect that snooping has on bus timing, it does not show the timing when there is a snoop hit of the address. These timings are shown in the following examples.

8.5.4.2 Address Retry During 602 Read Transaction—Single-Cycle Address Phase

The $\overline{\text{ARTRY}}$ signal is not sampled until the second ck after the read address is transferred. When a read operation uses the single-cycle address phase, the $\overline{\text{ARTRY}}$ signal is not sampled until the second clock cycle after the address is transferred.

Figure 8-30 shows the $\overline{\text{ARTRY}}$ signal asserted after snoop hit on a read operation with the single-cycle address phase. This example illustrates the fact that the $\overline{\text{ARTRY}}$ signal is not sampled until the clock cycle after the address signals are no longer in high impedance— that is, the second clock cycle after the read address is transferred.
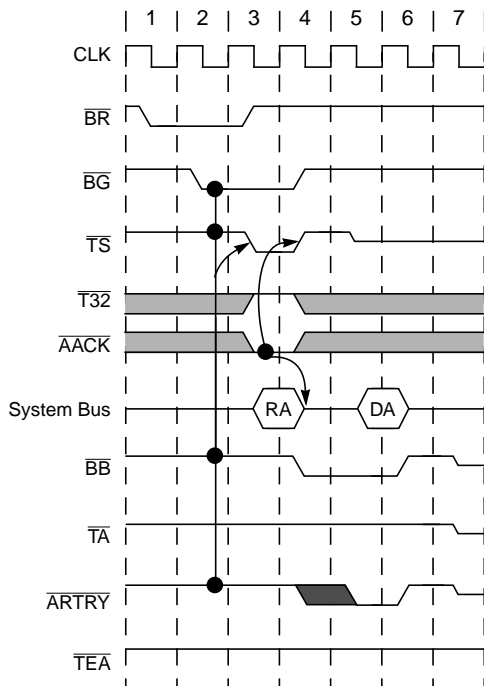
Figure 8-30. $\overline{\text{ARTRY}}$ During Read Transaction—Single-Cycle Address Phase

Note that the slave device may assert $\overline{\text{ARTRY}}$ in clock cycle 4; the 602 does not sample $\overline{\text{ARTRY}}$ until the next clock cycle simultaneously with the read data being put on the bus. Because $\overline{\text{ARTRY}}$ is asserted in this case, the data transfer is not allowed to complete; $\overline{\text{TA}}$ is not asserted.

8.5.4.3 Address Retry During 602 Read Transaction—Multicycle Address Phase

In Figure 8-31 the multicycle address phase is used, and the slave device may assert $\overline{\text{ARTRY}}$ as early as clock cycle 4; it is not sampled until after the address is transferred.
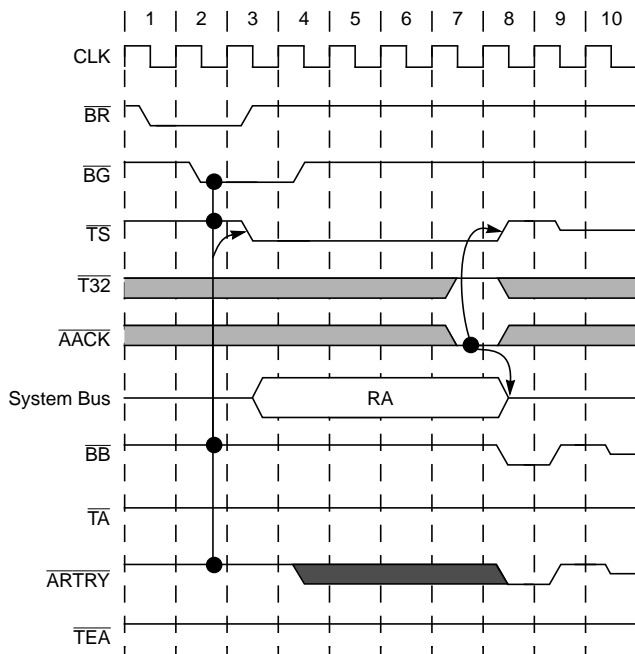
Figure 8-31. $\overline{\text{ARTRY}}$ During 602 Read Transaction—Multicycle Address Phase

8.5.4.4 $\overline{\text{ARTRY}}$ During Other Master Read Transaction—Single-Cycle Address Phase

Figure 8-32 shows the 602 snooping a bus operation by another bus master.
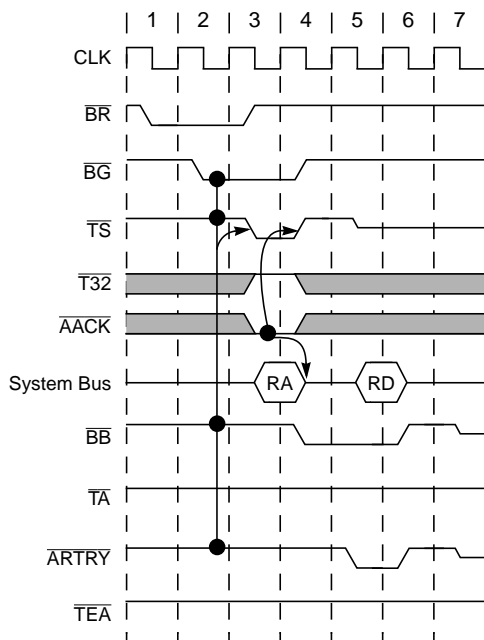
Figure 8-32. $\overline{\text{ARTRY}}$ During Other Master Read—Single-Cycle Address Phase

Note that the $\overline{\text{ARTRY}}$ signal cannot be asserted on the cycle immediately following the address transfer as it can when the transaction has a multicycle address phase (shown in Figure 8-33). As described in Section 7.2.5.2, "Address Retry (ARTRY)," the $\overline{\text{ARTRY}}$ signal is asserted on the second bus cycle after the assertion of the $\overline{\text{TS}}$ signal and is negated on the second cycle following the negation of the $\overline{\text{AACK}}$ signal.

8.5.4.5 $\overline{\text{ARTRY}}$ During Other Master Read Transaction—Multicycle Address Phase

Figure 8-33 shows the 602 asserting $\overline{\text{ARTRY}}$ after a snoop hit on an address broadcast by another device that shares the memory bus.
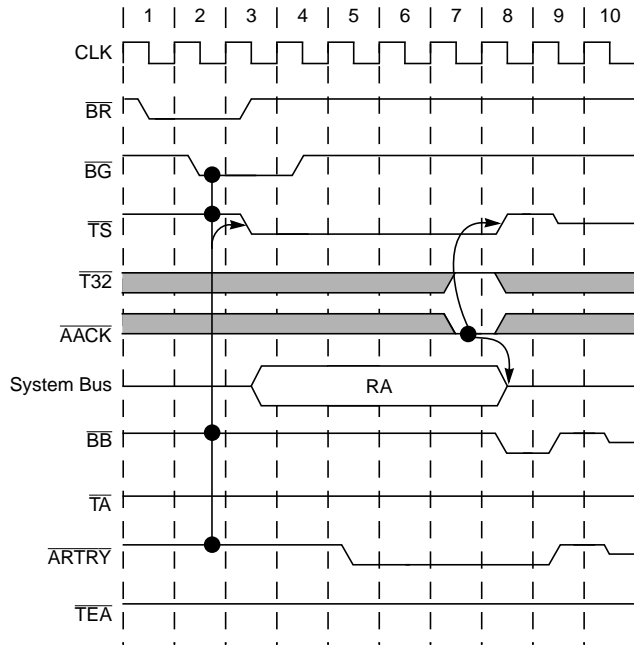
Figure 8-33. $\overline{\text{ARTRY}}$ During Other Master Read Transaction—Multicycle Address Phase

As in the previous example, the $\overline{\text{ARTRY}}$ signal is asserted in the second cycle following the assertion of $\overline{\text{TS}}$, as described in Section 7.2.5.2, "Address Retry (ARTRY)." In the case of the multicycle address phase, the assertion of the $\overline{\text{AACK}}$ signal must be delayed for the address to be decoded. Because the negation of the $\overline{\text{ARTRY}}$ signal must occur no sooner than two clock cycles after the assertion of the $\overline{\text{AACK}}$ signal, $\overline{\text{ARTRY}}$ is held asserted from clock cycles 5–9.

8.5.4.6 Snoop Hit—Write-Back Transaction

Figure 8-34 shows the bus timing when a snoop hit occurs on a read address. In this example, a DMA device has requested and has been granted the bus for a burst read operation, and the read address is snooped by the 602.
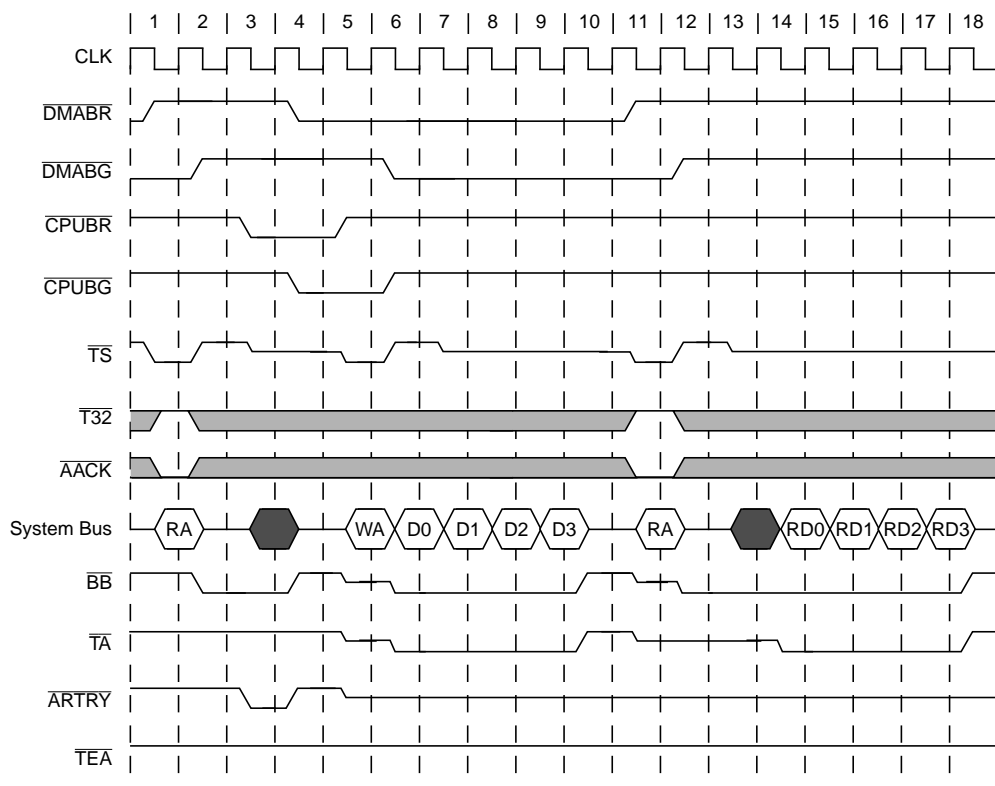
Figure 8-34. Snoop Hit—Write-Back Transaction

The signal interactions are as follows:

1. In clock cycles 1–3, the DMA device has been granted mastership of the memory bus ($\overline{\text{DMABG}}$ asserted). It broadcasts the read address in clock cycle 1.

2. The 602 snoops and hits the read address, and asserts the $\overline{\text{ARTRY}}$ and $\overline{\text{CPUBR}}$ signals in clock cycle 3.

3. In clock cycle 4, the bus busy signal, $\overline{\text{BB}}$, is negated for the DMA operation simultaneously with the assertion of the bus grant signal ($\overline{\text{CPUBG}}$) to the 602 and the reassertion of the DMA device's bus request signal ($\overline{\text{DMABR}}$).

4. In clock cycle 5, the $\overline{\text{TS}}$ signal is asserted and the 602 puts the write address onto the memory bus.

5. In clock cycle 6, a four-beat write operation begins and the memory bus grant is given to the DMA device so it can retry its read transaction when the write operation completes.

6. In clock cycle 11, the DMA device starts its read transaction again, and it completes as normal.

8.5.4.7 Injected Snoop Timings

As described in Section 8.4.2, "Qualified Snoop Conditions," injected snoops can occur during read transactions (signalled by $\overline{\text{TS}}$ and $\overline{\text{TA}}$):

When a 602 bus master performs a burst read transaction, the read target device can inject snoops by asserting $\overline{\text{TS}}$, negating $\overline{\text{TA}}$, and driving the snooped address on the address bus. As shown in Section 8.5.4.7.1, "First Injected Snoop in the Injected Snoop Window," and Section 8.5.4.7.2, "Last Injected Snoop in the Injected Snoop Window," the window of the injected snoop is from the third cycle following the assertion of $\overline{\text{BB}}$ to the cycle before the last read data beat is transferred.

8.5.4.7.1 First Injected Snoop in the Injected Snoop Window

The example shown in Figure 8-35 shows a snoop injected at the earliest possible moment in a read operation.
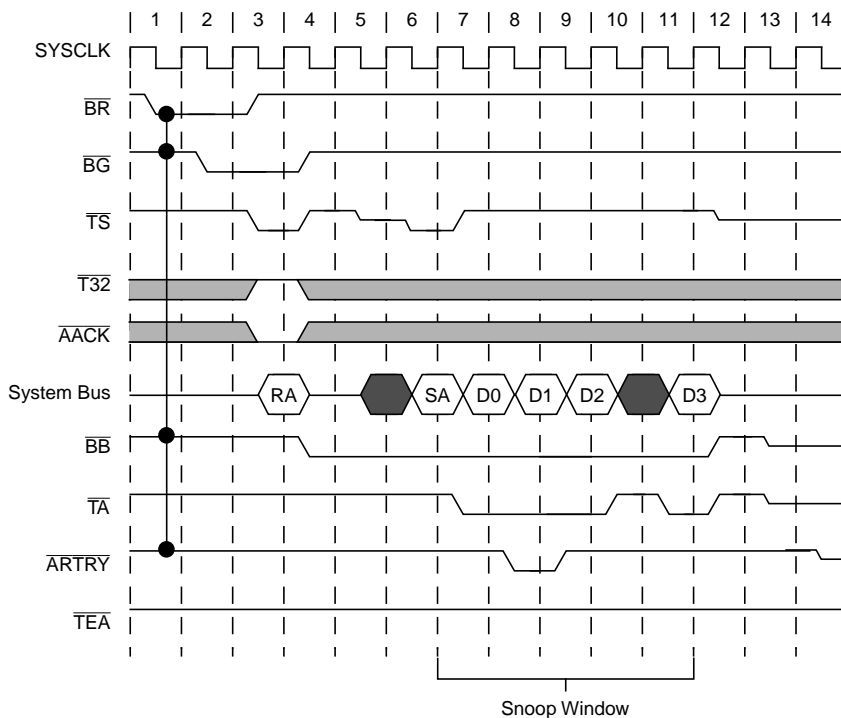
Note: $\overline{\text{TA}}$ must be driven high (negated) on the snoop cycle (clock #6).

Figure 8-35. First Injected Snoop in the Injected Snoop Window

The signal interactions are as follows:

1. The burst read transaction begins as normal, as shown in Section 8.5.1.1, "Nonburst Read Transaction—64-Bit Mode."

2. The first difference occurs in clock cycle 6 when the other bus device signals its intentions to snoop by asserting $\overline{TS}$, negates $\overline{TA}$, and puts its snoop address out on the memory bus, in place of the first data beat.

3. In clock cycle 7, the snoop address transaction completes, $\overline{TS}$ is negated, $\overline{TA}$ is asserted, and the first data beat is transferred.

4. In this example, there is a snoop hit, so in clock cycle 8, the 602 asserts $\overline{ARTRY}$ to force the other bus device to postpone its operation.

5. In the rest of the diagram, the 602's read operation completes as normal.

8.5.4.7.2 Last Injected Snoop in the Injected Snoop Window

Figure 8-36 shows an operation similar to that in Figure 8-35. However, in this example, the snoop address is broadcast at the last possible moment—before the last data beat.
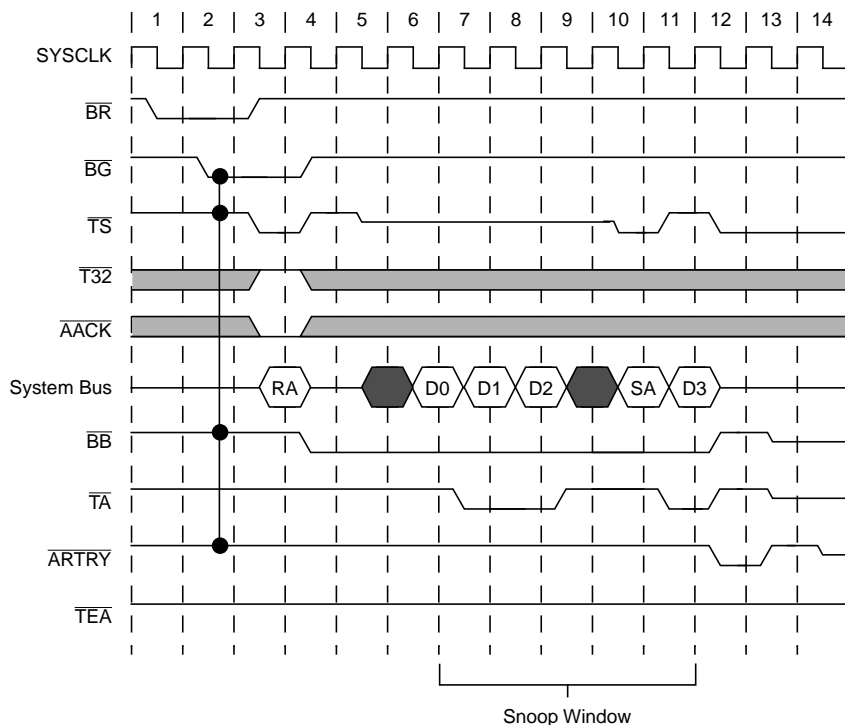
Note: $\overline{TA}$ must be driven high (negated) on the snoop cycle (clock #10).

Figure 8-36. Last Injected Snoop in the Injected Snoop Window

As in the previous example, the other bus master takes advantage of its ability to inject a snoop during a burst read operation. In this case, the other device asserts $\overline{\text{TS}}$ and broadcasts the snoop address in clock cycle 10. In the next clock cycle, the $\overline{\text{TA}}$ signal is reasserted and the final data beat is transferred. Once again, a snoop hit occurs, and the 602 asserts the $\overline{\text{ARTRY}}$ signal (clock cycle 12).

8.5.5 Address-Only Transactions

The 602 generates an address-only bus transaction only when a **dcbz** instruction is executed. Examples of address-only transactions are given in Section 8.5.5.1, "Single-Cycle Address-Only Transaction," and Section 8.5.5.2, "Multicycle Address-Only Transaction."

8.5.5.1 Single-Cycle Address-Only Transaction

Because many address-only operations cause other devices to perform an action (for example by either flushing, killing, or clearing the contents of a cache block), it is typical for another device to snoop the address, and assert the $\overline{\text{ARTRY}}$ signal to perform the required operation. The timing for the $\overline{\text{ARTRY}}$ signal is included in Figure 8-37.
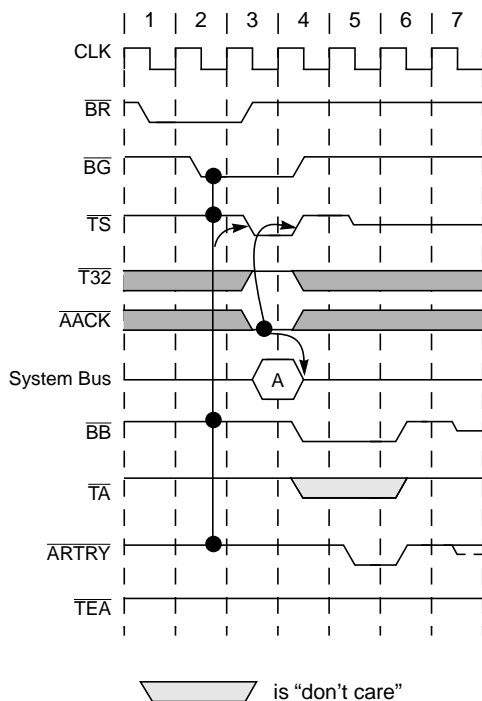
Figure 8-37. Single-Cycle Address-Only Transaction

Here the transfer acknowledge signal is treated as a "don't care" because there is no data phase. Similar to the example shown in Section 8.5.4.4, "ARTRY During Other Master Read Transaction—Single-Cycle Address Phase," the $\overline{\text{ARTRY}}$ signal is asserted by a snooping device in the second clock cycle after the address is transmitted.

8.5.5.2 Multicycle Address-Only Transaction

Figure 8-38 shows a multicycle address-only transaction, again showing the timing for the assertion of $\overline{\text{ARTRY}}$ by a snooping device that hits the address. As shown in Section 8.5.4.5, "ARTRY During Other Master Read Transaction—Multicycle Address Phase," the $\overline{\text{ARTRY}}$ signal is asserted in the clock cycle after the address has been transferred. Again, because there is no data to be transferred, the $\overline{\text{TA}}$ signal is a don't care.
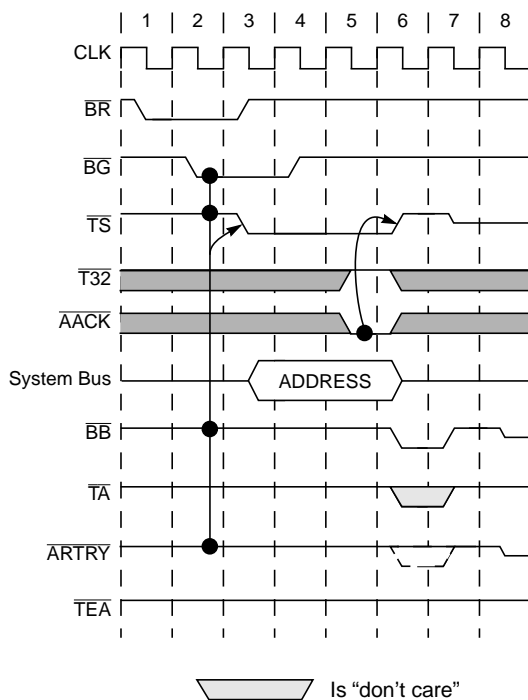
Figure 8-38. Multicycle Address-Only Transaction

PowerPC 602 RISC Microprocessor User's Manual

Chapter 9
Power Management

The PowerPC 602 microprocessor is specifically designed for low-power operation. The 602 provides both automatic and program-controllable power reduction modes for progressive reduction of power consumption. This chapter describes the hardware support provided by the 602 for power management.

9.1 Dynamic Power Management

Dynamic power management automatically powers up and down individual execution units of the 602 on a demand basis. For example, if no floating-point instructions are being executed, the floating-point unit is automatically powered down. Power is not actually removed from the execution unit; instead, each execution unit has an independent clock input, which is automatically controlled on a clock-by-clock basis. Since CMOS circuits consume negligible power when they are not switching, stopping the clock to an execution unit effectively eliminates its power consumption. Dynamic power management is transparent to software and external hardware and is enabled by setting HID0[DPM] (bit 11) on power-up, or following the assertion of $\overline{\text{HRESET}}$.

9.2 Programmable Power Modes

The 602 provides four programmable power states—full power, doze, nap, and sleep. Software selects these modes by setting one (and only one) of three power-saving mode bits—HID0[DOZE], HID0[NAP], and HID0[SLEEP] (bits 8, 9, and 10, respectively).

Hardware can enable a power management state through external asynchronous interrupts. The hardware interrupt causes the transfer of program flow to the interrupt handler, which sets the appropriate mode. The 602 provides a separate exception and exception vector for power management—the system management interrupt (SMI). The 602 also contains a decrement timer, which allows it to enter the nap or doze mode for a predetermined period and then return to full power operation through the decrementer interrupt.

The 602 cannot switch from one power management mode to another without returning to full power mode. Nap and sleep modes disable bus snooping; therefore, a hardware handshake is provided to ensure coherency before the 602 enters these power management modes. Table 9-1 summarizes the power states.

Table 9-1. PowerPC 602 Microprocessor Programmable Power Modes

PM Mode	Functioning Units	Activation Method	Full-Power Wake Up Method
Full power	All units active	—	—
Full power (with dynamic power management)	Requested logic by demand	Instruction dispatch	—
Doze	• Bus snooping • Data cache as needed • Decrementer timer	Software	External asynchronous exceptions Decrementer interrupt Reset
Nap	Decrementer timer	Hardware/software	External asynchronous exceptions Decrementer interrupt Reset
Sleep	None	Hardware/software	External asynchronous exceptions Reset

9.2.1 Power Management Modes

The following sections describe the characteristics of the 602's power management modes, the requirements for entering and exiting the various modes, and the system capabilities provided by the 602 while the power management modes are active.

9.2.1.1 Full-Power Mode with Dynamic Power Management Disabled

Full-power mode with dynamic power management disabled power mode is selected by clearing HID0[DPM].

- Default state following power-up and $\overline{\text{HRESET}}$
- All functional units are operating at full processor speed at all times.

9.2.1.2 Full-Power Mode with Dynamic Power Management Enabled

Full-power mode with dynamic power management enabled (HID0[DPM] = 1) provides on-chip power management without affecting the functionality or performance of the 602.

- Required functional units are operating at full processor speed
- Functional units are clocked only when needed
- No software or hardware intervention required after mode is set
- Software/hardware and performance transparent

9.2.1.3 Doze Mode

Doze mode disables most functional units but maintains cache coherency by enabling the bus interface unit and snooping. A snoop hit causes the 602 to enable the data cache, copy the data back to memory, disable the cache, and fully return to the doze state.

- Most functional units disabled
- Bus snooping and time base/decrementer still enabled
- Doze mode sequence
  - Set doze bit (HID0[8] = 1)
  - 602 enters doze mode after several processor clocks
- Several methods of returning to full-power mode
  - Assert $\overline{\text{INT}}$, $\overline{\text{SMI}}$, $\overline{\text{MCP}}$ or decrementer exceptions
  - Assert $\overline{\text{HRESET}}$ or $\overline{\text{SRESET}}$
- Transition to full power state takes no more than a few processor cycles
- Phase-locked loop (PLL) running and locked to SYSCLK

9.2.1.4 Nap Mode

The nap mode disables the 602 but maintains the PLL and the time base/decrementer. The time base can be used to restore the 602 to full power state after a programmed amount of time. Because bus snooping is disabled for nap and sleep modes, a hardware handshake using the quiesce request ($\overline{\text{QREQ}}$) and quiesce acknowledge ($\overline{\text{QACK}}$) signals are required to maintain data coherency. The 602 will assert the $\overline{\text{QREQ}}$ signal to indicate that it is ready to disable bus snooping. When the system ensures that snooping is no longer necessary, it asserts $\overline{\text{QACK}}$ and the 602 enters the sleep or nap mode.

- Time base/decrementer still enabled
- Most functional units disabled (including bus snooping)
- All nonessential input receivers disabled
- Nap mode sequence
  - Set nap bit (HID0[9] = 1)
  - 602 asserts quiesce request ($\overline{\text{QREQ}}$)
  - System asserts quiesce acknowledge ($\overline{\text{QACK}}$)
  - 602 enters sleep mode after several processor clocks
- Several methods of returning to full-power mode
  - Assertion of the $\overline{\text{INT}}$, $\overline{\text{SMI}}$, and $\overline{\text{MCP}}$ signals or occurrence of a decrementer interrupt
  - Assert hard or soft reset
- Transition to full power takes no more than a few processor cycles
- PLL running and locked to SYSCLK

9.2.1.5 Sleep Mode

Sleep mode consumes the least amount of power of the four modes because all functional units are disabled. To conserve the maximum amount of power, the PLL may be disabled and the SYSCLK may be removed. Due to the fully static design of the 602, the internal processor state is preserved when no internal clock is present. Because the time base and decrementer are disabled while the 602 is in sleep mode, the 602 microprocessor's time base contents must be updated from an external time base following sleep mode if accurate time-of-day maintenance is required. Before the 602 enters the sleep mode, the 602 will assert the $\overline{QREQ}$ signal to indicate that it is ready to disable bus snooping. When the system has ensured that snooping is no longer necessary, it asserts $\overline{QACK}$ and the 602 enters sleep mode.

- All functional units disabled (including bus snooping and time base)
- All nonessential input receivers disabled
  - Internal clock regenerators disabled
  - PLL still running (see below)
- Sleep mode sequence
  - Set sleep bit (HID0[10] = 1)
  - 602 asserts quiesce request ($\overline{QREQ}$)
  - System asserts quiesce acknowledge ($\overline{QACK}$)
  - 602 enters sleep mode after several processor clocks
- Several methods of returning to full-power mode
  - Assert $\overline{INT}$, $\overline{SMI}$, or $\overline{MCP}$ interrupts
  - Assert hard or soft reset
- PLL may be disabled and SYSCLK may be removed in sleep mode
- Return to full-power mode after PLL and SYSCLK disabled in sleep mode
  - Enable SYSCLK
  - Reconfigure PLL into desired processor clock mode
  - System logic waits for PLL start-up and relock time (100 μsec)
  - System logic asserts one of the sleep recovery signals (for example, INT or SMI)

9.2.2 Power Management Software Considerations

All outstanding bus operations must complete before nap or sleep modes are entered. Normally, a power management mode is selected by setting the appropriate HID0 mode bit during system configuration. Later, the power management mode is selected by setting the MSR[POW] bit. To provide a clean transition into and out of the power management mode, the **mtmsr**[POW] should be preceded by a **sync** instruction and followed by an **isync** instruction.

Appendix A
PowerPC Instruction Set Listings

This appendix lists the PowerPC 602 microprocessor's instruction set as well as the additional PowerPC instructions not implemented in the 602. 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 PowerPC architecture in alphabetical order by mnemonic.

Table A-1. Complete Instruction List Sorted by Mnemonic

Key:

Reserved bits	Instruction not implemented in the 602

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
addx	31	D	A	B	OE	266	Rc
addcx	31	D	A	B	OE	10	Rc
addex	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			
addmex	31	D	A	0 0 0 0 0	OE	234	Rc
addzex	31	D	A	0 0 0 0 0	OE	202	Rc
andx	31	S	A	B		28	Rc
andcx	31	S	A	B		60	Rc

Name	0	6–10	11–15	16–20	21–30	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
dsa [1,6]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	628	0
eciwx	31	D	A	B	310	0
ecowx	31	S	A	B	438	0

Name	0	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
eieio	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	854		0
eqvx	31	S	A	B	284		Rc
esa [1,6]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	596		0
extsbx	31	S	A	0 0 0 0 0	954		Rc
extshx	31	S	A	0 0 0 0 0	922		Rc
extswx [4]	31	S	A	0 0 0 0 0	986		Rc
fabsx	63	D	0 0 0 0 0	B	264		Rc
faddx	63	D	A	B	0 0 0 0 0	21	Rc
faddsx	59	D	A	B	0 0 0 0 0	21	Rc
fcfidx [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
fctidx [4]	63	D	0 0 0 0 0	B	814		Rc
fctidzx [4]	63	D	0 0 0 0 0	B	815		Rc
fctiwx	63	D	0 0 0 0 0	B	14		Rc
fctiwzx	63	D	0 0 0 0 0	B	15		Rc
fdivx	63	D	A	B	0 0 0 0 0	18	Rc
fdivsx	59	D	A	B	0 0 0 0 0	18	Rc
fmaddx	63	D	A	B	C	29	Rc
fmaddsx	59	D	A	B	C	29	Rc
fmrx	63	D	0 0 0 0 0	B	72		Rc
fmsubx	63	D	A	B	C	28	Rc
fmsubsx	59	D	A	B	C	28	Rc
fmulx	63	D	A	0 0 0 0 0	C	25	Rc
fmulsx	59	D	A	0 0 0 0 0	C	25	Rc
fnabsx	63	D	0 0 0 0 0	B	136		Rc
fnegx	63	D	0 0 0 0 0	B	40		Rc
fnmaddx	63	D	A	B	C	31	Rc
fnmaddsx	59	D	A	B	C	31	Rc
fnmsubx	63	D	A	B	C	30	Rc
fnmsubsx	59	D	A	B	C	30	Rc
fresx [5]	59	D	0 0 0 0 0	B	0 0 0 0 0	24	Rc
frspx	63	D	0 0 0 0 0	B	12		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
frsqrtex [5]	63	D	0 0 0 0 0	B	0 0 0 0 0	26	Rc
fselx [5]	63	D	A	B	C	23	Rc
fsqrtx [5]	63	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsqrtsx [5]	59	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsubx	63	D	A	B	0 0 0 0 0	20	Rc
fsubsx	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

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

Name	0–5	6–10	11–15	16–20	21–30	31
lhzx	31	D	A	B	279	0
lmw [3]	46	D	A	d		
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
mfrom [1,6]	31	D	A	0 0 0 0 0	265	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
mtfsb0x	63	crbD	0 0 0 0 0	0 0 0 0 0	70	Rc
mtfsb1x	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

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
mulhdx [4]	31	D	A	B	0	73	Rc
mulhdux [4]	31	D	A	B	0	9	Rc
mulhwx	31	D	A	B	0	75	Rc
mulhwux	31	D	A	B	0	11	Rc
mulldx [4]	31	D	A	B	OE	233	Rc
mulli	7	D	A	SIMM			
mullwx	31	D	A	B	OE	235	Rc
nandx	31	S	A	B		476	Rc
negx	31	D	A	0 0 0 0 0	OE	104	Rc
norx	31	S	A	B		124	Rc
orx	31	S	A	B		444	Rc
orcx	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
rldclx [4]	30	S	A	B	mb	8	Rc
rldcrx [4]	30	S	A	B	me	9	Rc
rldicx [4]	30	S	A	sh	mb	2	sh Rc
rldiclx [4]	30	S	A	sh	mb	0	sh Rc
rldicrx [4]	30	S	A	sh	me	1	sh Rc
rldimix [4]	30	S	A	sh	mb	3	sh Rc
rlwimix	20	S	A	SH	MB	ME	Rc
rlwinmx	21	S	A	SH	MB	ME	Rc
rlwnmx	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
sldx [4]	31	S	A	B		27	Rc
slwx	31	S	A	B		24	Rc
sradx [4]	31	S	A	B		794	Rc
sradix [4]	31	S	A	sh		413	sh Rc
srawx	31	S	A	B		792	Rc
srawix	31	S	A	SH		824	Rc

Name	0	S	A	B		Rc
srd$x$ [4]	31	S	A	B	539	Rc
srw$x$	31	S	A	B	536	Rc
stb	38	S	A	d		
stbu	39	S	A	d		
stbux	31	S	A	B	247	0
stbx	31	S	A	B	215	0
std [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

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
stwx	31	S	A	B		151	0
subf*x*	31	D	A	B	OE	40	Rc
subfc*x*	31	D	A	B	OE	8	Rc
subfe*x*	31	D	A	B	OE	136	Rc
subfic	08	D	A	SIMM			
subfme*x*	31	D	A	0 0 0 0 0	OE	232	Rc
subfze*x*	31	D	A	0 0 0 0 0	OE	200	Rc
sync	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0		598	0
td [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
tlbld [1,6]	31	0 0 0 0 0	0 0 0 0 0	B		978	0
tlbli [1,6]	31	0 0 0 0 0	0 0 0 0 0	B		1010	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 in the PowerPC architecture
[6] 602-implementation specific instruction

PowerPC 602 RISC Microprocessor User's Manual

A.2 Instructions Sorted by Opcode

Table A-2 lists the instructions defined in the PowerPC architecture in numeric order by opcode.

Key:

☐ Reserved bits   ▨ Instruction not implemented in the 602

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 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
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

Name	0 ... 5	6 ... 10	11 ... 15	16 ... 20	21 ... 25	26 ... 30	31
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
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

Name	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30	31
cntlzdx [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
andcx	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
mulhdx [4]	0 1 1 1 1 1	D	A	B	0	0 0 0 1 0 0 1 0 0 1	Rc
mulhwx	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
negx	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
norx	0 1 1 1 1 1	S	A	B		0 0 0 1 1 1 1 1 0 0	Rc
subfex	0 1 1 1 1 1	D	A	B	OE	0 0 1 0 0 0 1 0 0 0	Rc
addex	0 1 1 1 1 1	D	A	B	OE	0 0 1 0 0 0 1 0 1 0	Rc
mtcrf	0 1 1 1 1 1	S	0	CRM	0	0 0 1 0 0 1 0 0 0 0	0
mtmsr	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
subfzex	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
addzex	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
subfmex	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
addmex	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
mullwx	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

Name	0–5	6–10	11–15	16–20	21	22–30	31
mfrom [1,6]	0 1 1 1 1 1	D	A	0 0 0 0 0		0 1 0 0 0 0 1 0 0 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
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

Name	0 – 5	6 – 10	11 – 15	16 – 20	21 – 30	31
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
esa [1,6]	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 0 0	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
dsa [1,6]	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 1 1 0 1 0 0	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
tlbld [1,6]	0 1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	B	1 1 1 1 0 1 0 0 1 0	0
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
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

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
tlbli [1,6]	0 1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	B	1 1 1 1 1 1 0 0 1 0	0
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
fdivsx	1 1 1 0 1 1	D	A	B	0 0 0 0 0 1 0 0 1 0	Rc
fsubsx	1 1 1 0 1 1	D	A	B	0 0 0 0 0 1 0 1 0 0	Rc
faddsx	1 1 1 0 1 1	D	A	B	0 0 0 0 0 1 0 1 0 1	Rc
fsqrtsx [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

Name	0-5	6-10	11-15	16-20	21-25	26-30	31
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
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

Name	0 5	6	7 8 9 10 11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
mtfsf*x*	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 in the PowerPC architecture
[6] 602-implementation specific instruction

A.3 Instructions Grouped by Functional Categories

Table A-3 through Table A-30 list the PowerPC instructions grouped by function.

Key:

☐ Reserved bits	■ Instruction not implemented in the 602

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			
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	12-15	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	11-15	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	11-15	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

Name	0	6	11	16	21	26	30	31
rldicrx [4]	30	S	A	sh	me	1	sh	Rc
rldimix [4]	30	S	A	sh	mb	3	sh	Rc
rlwimix	22	S	A	SH	MB	ME		Rc
rlwinmx	20	S	A	SH	MB	ME		Rc
rlwnmx	21	S	A	SH	MB	ME		Rc

Table A-7. Integer Shift Instructions

Name	0	6	11	16	21	30	31
sldx [4]	31	S	A	B	27		Rc
slwx	31	S	A	B	24		Rc
sradx [4]	31	S	A	B	794		Rc
sradix [4]	31	S	A	sh	413	sh	Rc
srawx	31	S	A	B	792		Rc
srawix	31	S	A	SH	824		Rc
srdx [4]	31	S	A	B	539		Rc
srwx	31	S	A	B	536		Rc

Table A-8. Floating-Point Arithmetic Instructions

Name	0	6	11	16	21	26	31
faddx	63	D	A	B	0 0 0 0 0	21	Rc
faddsx	59	D	A	B	0 0 0 0 0	21	Rc
fdivx	63	D	A	B	0 0 0 0 0	18	Rc
fdivsx	59	D	A	B	0 0 0 0 0	18	Rc
fmulx	63	D	A	0 0 0 0 0	C	25	Rc
fmulsx	59	D	A	0 0 0 0 0	C	25	Rc
fresx [5]	59	D	0 0 0 0 0	B	0 0 0 0 0	24	Rc
frsqrtex [5]	63	D	0 0 0 0 0	B	0 0 0 0 0	26	Rc
fsubx	63	D	A	B	0 0 0 0 0	20	Rc
fsubsx	59	D	A	B	0 0 0 0 0	20	Rc
fselx [5]	63	D	A	B	C	23	Rc
fsqrtx [5]	63	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsqrtsx [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	6–10	11–15	16–20	21–25	26–30	31
fmaddx	63	D	A	B	C	29	Rc
fmaddsx	59	D	A	B	C	29	Rc
fmsubx	63	D	A	B	C	28	Rc
fmsubsx	59	D	A	B	C	28	Rc
fnmaddx	63	D	A	B	C	31	Rc
fnmaddsx	59	D	A	B	C	31	Rc
fnmsubx	63	D	A	B	C	30	Rc
fnmsubsx	59	D	A	B	C	30	Rc

Table A-10. Floating-Point Rounding and Conversion Instructions

Name	0–5	6–10	11–15	16–20	21–30	31
fcfidx [4]	63	D	0 0 0 0 0	B	846	Rc
fctidx [4]	63	D	0 0 0 0 0	B	814	Rc
fctidzx [4]	63	D	0 0 0 0 0	B	815	Rc
fctiwx	63	D	0 0 0 0 0	B	14	Rc
fctiwzx	63	D	0 0 0 0 0	B	15	Rc
frspx	63	D	0 0 0 0 0	B	12	Rc

Table A-11. Floating-Point Compare Instructions

Name	0–5	6–8	9–10	11–15	16–20	21–30	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	6–8	9–10	11–13	14–15	16–20	21–30	31
mcrfs	63	crfD	0 0	crfS	0 0	0 0 0 0 0	64	0
mffsx	63	D		0 0 0 0 0		0 0 0 0 0	583	Rc
mtfsb0x	63	crbD		0 0 0 0 0		0 0 0 0 0	70	Rc
mtfsb1x	63	crbD		0 0 0 0 0		0 0 0 0 0	38	Rc
mtfsfx	31	0	FM		0	B	711	Rc
mtfsfix	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 — 10	11 — 15	16 — 20	21 — 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

Name						
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 Instructions

Name	0　　5	6　7　8　9　10	11　12　13　14　15	16　17　18　19　20	21　22　23　24　25　26　27　28　29　30	31
eieio	31	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	00000	00000	00000	598	0

Table A-19. Floating-Point Load Instructions

Name	0	5 6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
lfd	50	D	A	d		
lfdu	51	D	A	d		
lfdux	31	D	A	B	631	0
lfdx	31	D	A	B	599	0
lfs	48	D	A	d		
lfsu	49	D	A	d		
lfsux	31	D	A	B	567	0
lfsx	31	D	A	B	535	0

Table A-20. Floating-Point Store Instructions

Name	0	5 6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
stfd	54	S	A	d		
stfdu	55	S	A	d		
stfdux	31	S	A	B	759	0
stfdx	31	S	A	B	727	0
stfiwx [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 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
fabsx	63	D	00000	B	264	Rc
fmrx	63	D	00000	B	72	Rc
fnabsx	63	D	00000	B	136	Rc
fnegx	63	D	00000	B	40	Rc

Table A-22. Branch Instructions

Name	0 · · · · · 5	6 · · · · 10	11 · · · 15	16 · · · 20	21 · · · · · · 30	30	31
bx	18	LI				AA	LK
bcx	16	BO	BI	BD		AA	LK
bcctrx	19	BO	BI	0 0 0 0 0	528		LK
bclrx	19	BO	BI	0 0 0 0 0	16		LK

Table A-23. Condition Register Logical Instructions

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
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 · · · · · 30	31
dsa [6]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	628	0
esa [6]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	596	0
mfrom [1,6]	31	D	A	0 0 0 0 0	265	0
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 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
td [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 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
mcrxr	31	crfS 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 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
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 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
mfsr [1]	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 7 8 9 10 11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
slbia [1,4,5]	31	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
tlbld [1,6]	31	0 0 0 0 0	0 0 0 0 0	B	978	0
tlbli [1,6]	31	0 0 0 0 0	0 0 0 0 0	B	1010	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 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
eciwx	31	D	A	B	310	0
ecowx	31	S	A	B	438	0

[1] Supervisor-level instruction
[2] Supervisor- and user-level instruction
[3] Load and store string or multiple instruction
[4] 64-bit instruction
[5] Optional in the PowerPC architecture
[6] 602-implementation specific instruction

A.4 Instructions Sorted by Form

Table A-31 through Table A-45 list the PowerPC instructions grouped by form.

Key:

☐ Reserved bits	☐ Instruction not implemented in the 602

Table A-31. I-Form

OPCD	LI	AA	LK

Specific Instruction

Name	0					5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

bx	18	LI	AA	LK

Table A-32. B-Form

OPCD	BO	BI	BD	AA	LK

Specific Instruction

Name	0					5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

bcx	16	BO	BI	BD	AA	LK

Table A-33. SC-Form

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 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 0	1	0

Table A-34. D-Form

OPCD	D			A	d
OPCD	D			A	SIMM
OPCD	S			A	d
OPCD	S			A	UIMM
OPCD	crfD	0	L	A	SIMM
OPCD	crfD	0	L	A	UIMM
OPCD	TO			A	SIMM

Specific Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
addi	14		D			A			SIMM																			
addic	12		D			A			SIMM																			
addic.	13		D			A			SIMM																			
addis	15		D			A			SIMM																			
andi.	28		S			A			UIMM																			
andis.	29		S			A			UIMM																			
cmpi	11		crfD		0	L		A			SIMM																	
cmpli	10		crfD		0	L		A			UIMM																	
lbz	34		D			A			d																			
lbzu	35		D			A			d																			
lfd	50		D			A			d																			
lfdu	51		D			A			d																			
lfs	48		D			A			d																			
lfsu	49		D			A			d																			
lha	42		D			A			d																			
lhau	43		D			A			d																			
lhz	40		D			A			d																			
lhzu	41		D			A			d																			
lmw [3]	46		D			A			d																			
lwz	32		D			A			d																			
lwzu	33		D			A			d																			
mulli	7		D			A			SIMM																			
ori	24		S			A			UIMM																			
oris	25		S			A			UIMM																			
stb	38		S			A			d																			
stbu	39		S			A			d																			
stfd	54		S			A			d																			
stfdu	55		S			A			d																			
stfs	52		S			A			d																			
stfsu	53		S			A			d																			
sth	44		S			A			d																			
sthu	45		S			A			d																			
stmw [3]	47		S			A			d																			

stw	36	S	A	d
stwu	37	S	A	d
subfic	08	D	A	SIMM
tdi [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	OPCD	f1	f2	f3	XO	Rc
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
dsa [6]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	628	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
esa [6]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	596	0
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
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		00000	B	814	Rc
fctidz*x* [4]	63	D		00000	B	815	Rc
fctiw*x*	63	D		00000	B	14	Rc
fctiwz*x*	63	D		00000	B	15	Rc
fmr*x*	63	D		00000	B	72	Rc
fnabs*x*	63	D		00000	B	136	Rc
fneg*x*	63	D		00000	B	40	Rc
frsp*x*	63	D		00000	B	12	Rc
icbi	31	00000		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
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	00000	64	0
mcrxr	31	crfD	0 0	00000	00000	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
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
tlbld [1,6]	31	0 0 0 0 0	0 0 0 0 0	B	978	0
tlbli [1,6]	31	0 0 0 0 0	0 0 0 0 0	B	1010	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
xorx	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
bcctrx	19	BO	BI	0 0 0 0 0	528	LK
bclrx	19	BO	BI	0 0 0 0 0	16	LK
crand	19	crbD	crbA	crbB	257	0
crandc	19	crbD	crbA	crbB	129	0
creqv	19	crbD	crbA	crbB	289	0
crnand	19	crbD	crbA	crbB	225	0
crnor	19	crbD	crbA	crbB	33	0
cror	19	crbD	crbA	crbB	449	0
crorc	19	crbD	crbA	crbB	417	0
crxor	19	crbD	crbA	crbB	193	0

isync	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	150	0
mcrf	19	crfD 0 0	crfS 0 0	0 0 0 0 0	0	0
rfi [1]	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	50	0

Table A-38. XFX-Form

OPCD	D	spr	XO	0
OPCD	D	0 CRM 0	XO	0
OPCD	S	spr	XO	0
OPCD	D	tbr	XO	0

Specific Instructions

Name 0   5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

mfspr [2]	31	D	spr	339	0
mftb	31	D	tbr	371	0
mtcrf	31	S	0 CRM 0	144	0
mtspr [2]	31	D	spr	467	0

Table A-39. XFL-Form

OPCD	0	FM	0	B	XO	Rc

Specific Instructions

Name 0   5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

mtfsf x	63	0	FM	0	B	711	Rc

Table A-40. XS-Form

OPCD	S	A	sh	XO	sh	Rc

Specific Instructions

Name 0   5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

sradi x [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

PowerPC 602 RISC Microprocessor User's Manual

Name		D	A	B	OE		Rc
addex	31	D	A	B	OE	138	Rc
addmex	31	D	A	0 0 0 0 0	OE	234	Rc
addzex	31	D	A	0 0 0 0 0	OE	202	Rc
divdx [4]	31	D	A	B	OE	489	Rc
divdux [4]	31	D	A	B	OE	457	Rc
divwx	31	D	A	B	OE	491	Rc
divwux	31	D	A	B	OE	459	Rc
mulhdx [4]	31	D	A	B	0	73	Rc
mulhdux [4]	31	D	A	B	0	9	Rc
mulhwx	31	D	A	B	0	75	Rc
mulhwux	31	D	A	B	0	11	Rc
mulldx [4]	31	D	A	B	OE	233	Rc
mullwx	31	D	A	B	OE	235	Rc
negx	31	D	A	0 0 0 0 0	OE	104	Rc
subfx	31	D	A	B	OE	40	Rc
subfcx	31	D	A	B	OE	8	Rc
subfex	31	D	A	B	OE	136	Rc
subfmex	31	D	A	0 0 0 0 0	OE	232	Rc
subfzex	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

Name		D	A	B	C		Rc
faddx	63	D	A	B	0 0 0 0 0	21	Rc
faddsx	59	D	A	B	0 0 0 0 0	21	Rc
fdivx	63	D	A	B	0 0 0 0 0	18	Rc
fdivsx	59	D	A	B	0 0 0 0 0	18	Rc
fmaddx	63	D	A	B	C	29	Rc
fmaddsx	59	D	A	B	C	29	Rc
fmsubx	63	D	A	B	C	28	Rc
fmsubsx	59	D	A	B	C	28	Rc

Name							
fmulx	63	D	A	0 0 0 0 0	C	25	Rc
fmulsx	59	D	A	0 0 0 0 0	C	25	Rc
fnmaddx	63	D	A	B	C	31	Rc
fnmaddsx	59	D	A	B	C	31	Rc
fnmsubx	63	D	A	B	C	30	Rc
fnmsubsx	59	D	A	B	C	30	Rc
fresx [5]	59	D	0 0 0 0 0	B	0 0 0 0 0	24	Rc
frsqrtex [5]	63	D	0 0 0 0 0	B	0 0 0 0 0	26	Rc
fselx [5]	63	D	A	B	C	23	Rc
fsqrtx [5]	63	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsqrtsx [5]	59	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsubx	63	D	A	B	0 0 0 0 0	20	Rc
fsubsx	59	D	A	B	0 0 0 0 0	20	Rc

Table A-43. M-Form

OPCD	S	A	SH	MB	ME	Rc
OPCD	S	A	B	MB	ME	Rc

Specific Instructions

Name	0　　　5	6　7　8　9　10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
rlwimix	20	S	A	SH	MB	ME	Rc
rlwinmx	21	S	A	SH	MB	ME	Rc
rlwnmx	23	S	A	B	MB	ME	Rc

Table A-44. MD-Form

OPCD	S	A	sh	mb	XO	sh	Rc
OPCD	S	A	sh	me	XO	sh	Rc

Specific Instructions

Name	0　　　5	6　7　8　9　10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
rldicx [4]	30	S	A	sh	mb	2	sh	Rc
rldiclx [4]	30	S	A	sh	mb	0	sh	Rc
rldicrx [4]	30	S	A	sh	me	1	sh	Rc
rldimix [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	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
rldclx [4]	30		S			A			B				mb						8									Rc
rldcrx [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 in the PowerPC architecture
[6] 602-implementation specific instruction

A.5 Instruction Set Legend

Table A-46 provides general information on the PowerPC instruction set (such as the architectural level, privilege level, and form).

Key:

☐ Reserved bits   ☐ Instruction not implemented in the 602

Table A-46. PowerPC Instruction Set Legend

	UISA	VEA	OEA	Supervisor Level	64-Bit	Optional	Form
addx	√						XO
addcx	√						XO
addex	√						XO
addi	√						D
addic	√						D
addic.	√						D
addis	√						D
addmex	√						XO
addzex	√						XO
andx	√						X
andcx	√						X
andi.	√						D
andis.	√						D
bx	√						I
bcx	√						B
bcctrx	√						XL
bclrx	√						XL
cmp	√						X
cmpi	√						D
cmpl	√						X
cmpli	√						D
cntlzdx	√				√		X
cntlzwx	√						X
crand	√						XL
crandc	√						XL
creqv	√						XL

	UISA	VEA	OEA	Supervisor Level	64-Bit	Optional	Form
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
dsa [3]							X
eciwx		√				√	X
ecowx		√				√	X
eieio		√					X
eqv*x*	√						X
esa [3]							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

	UISA	VEA	OEA	Supervisor Level	64-Bit	Optional	Form
fctiwzx	√					√	X
fdivx	√						A
fdivsx	√						A
fmaddx	√						A
fmaddsx	√						A
fmrx	√						X
fmsubx	√						A
fmsubsx	√						A
fmulx	√						A
fmulsx	√						A
fnabsx	√						X
fnegx	√						X
fnmaddx	√						A
fnmaddsx	√						A
fnmsubx	√						A
fnmsubsx	√						A
fresx	√					√	A
frspx	√						X
frsqrtex	√					√	A
fselx	√					√	A
fsqrtx	√					√	A
fsqrtsx	√					√	A
fsubx	√						A
fsubsx	√						A
icbi		√					X
isync		√					XL
lbz	√						D
lbzu	√						D
lbzux	√						X
lbzx	√						X
ld	√				√		DS
ldarx	√				√		X
ldu	√				√		DS

	UISA	VEA	OEA	Supervisor Level	64-Bit	Optional	Form
ldux	√				√		X
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

	UISA	VEA	OEA	Supervisor Level	64-Bit	Optional	Form
mcrxr	√						X
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

PowerPC 602 RISC Microprocessor User's Manual

	UISA	VEA	OEA	Supervisor Level	64-Bit	Optional	Form
rldcrx	√				√		MDS
rldicx	√				√		MD
rldiclx	√				√		MD
rldicrx	√				√		MD
rldimix	√				√		MD
rlwimix	√						M
rlwinmx	√						M
rlwnmx	√						M
sc	√		√				SC
slbia			√	√	√	√	X
slbie			√	√	√	√	X
sldx	√				√		X
slwx	√						X
sradx	√				√		X
sradix	√				√		XS
srawx	√						X
srawix	√						X
srdx	√				√		X
srwx	√						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

	UISA	VEA	OEA	Supervisor Level	64-Bit	Optional	Form
stfs	√						D
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
subfx	√						XO
subfcx	√						XO
subfex	√						XO
subfic	√						D
subfmex	√						XO
subfzex	√						XO
sync	√						X
td	√				√		X
tdi	√				√		D
tlbia			√	√		√	X
tlbie			√	√		√	X
tlbld [3]				√			X
tlbli [3]				√			X
tlbsync			√	√		√	X
tw	√						X

	UISA	VEA	OEA	Supervisor Level	64-Bit	Optional	Form
twi	√						D
xorx	√						X
xori	√						D
xoris	√						D

[1] Supervisor- and user-level instruction
[2] Load and store string or multiple instruction
[3] 602-implementation specific instruction

PowerPC 602 RISC Microprocessor User's Manual

Appendix B
Instructions Not Implemented

This appendix describes the 32-bit and 64-bit PowerPC instructions that are not implemented in the PowerPC 602 microprocessor. It also provides the 32-bit SPR encoding that is not implemented by the 602. Note that any attempt to execute instructions that are not implemented on the 602 causes an illegal instruction exception.

Table B-1 provides the 32-bit PowerPC instructions that are optional to the PowerPC architecture but not implemented by the 602.

Table B-1. 32-Bit Instructions Not Implemented by the PowerPC 602 Microprocessor

Mnemonic	Instruction
fsqrt	Floating Square Root (Double-Precision)
fsqrts	Floating Square Root Single
tlbia	TLB Invalidate All

Table B-2 provides a list of 64-bit instructions that are not implemented by the 602.

Table B-2. 64-Bit Instructions Not Implemented by the PowerPC 602 Microprocessor

Mnemonic	Instruction
cntlzd	Count Leading Zeros Double Word
divd	Divide Double Word
divdu	Divide Double Word Unsigned
extsw	Extend Sign Word
fcfid	Floating Convert From Integer Double Word
fctid	Floating Convert to Integer Double Word
fctidz	Floating Convert to Integer Double Word with Round toward Zero
ld	Load Double Word
ldarx	Load Double Word and Reserve Indexed
ldu	Load Double Word with Update

**Table B-2. 64-Bit Instructions Not Implemented by the PowerPC 602
Microprocessor (Continued)**

Mnemonic	Instruction
ldux	Load Double Word with Update Indexed
ldx	Load Double Word Indexed
lwa	Load Word Algebraic
lwaux	Load Word Algebraic with Update Indexed
lwax	Load Word Algebraic Indexed
mulld	Multiply Low Double Word
mulhd	Multiply High Double Word
mulhdu	Multiply High Double Word Unsigned
rldcl	Rotate Left Double Word then Clear Left
rldcr	Rotate Left Double Word then Clear Right
rldic	Rotate Left Double Word Immediate then Clear
rldicl	Rotate Left Double Word Immediate then Clear Left
rldicr	Rotate Left Double Word Immediate then Clear Right
rldimi	Rotate Left Double Word Immediate then Mask Insert
slbia	SLB Invalidate All
slbie	SLB Invalidate Entry
sld	Shift Left Double Word
srad	Shift Right Algebraic Double Word
sradi	Shift Right Algebraic Double Word Immediate
srd	Shift Right Double Word
std	Store Double Word
stdcx.	Store Double Word Conditional Indexed
stdu	Store Double Word with Update
stdux	Store Double Word Indexed with Update
stdx	Store Double Word Indexed
td	Trap Double Word
tdi	Trap Double Word Immediate

Table B-3 provides the 64-bit SPR encoding that is not implemented by the 602.

**Table B-3. 64-Bit SPR Encoding Not Implemented by the PowerPC 602
Microprocessor**

SPR			Register Name	Access
Decimal	**spr[5–9]**	**spr[0–4]**		
280	01000	11000	ASR	Supervisor

Appendix C
Boundary-Scan Testing Support

The PowerPC 602 microprocessor provides a boundary-scan interface for board-level testing. The boundary-scan interface of 602 is not completely IEEE 1149.1-compliant. However, the 602 can be tested with commercially available board-level JTAG Automatic Test Pattern Generation (ATPG) tools provided that certain constraints are met. Section C.2, "Unimplemented IEEE 1149.1 Features," describes specific noncompliant aspects and the constraints.

The 602's boundary-scan interface implements the five test port signals required by the IEEE 1149.1 specification.

C.1 Boundary-Scan Interface Description

The interface consists of a set of five signals, two test data registers, an instruction register, and a test access port (TAP) controller, described in the following sections. A block diagram of the interface is shown in Figure C-1.

C.1.1 Boundary-Scan Signals

The 602 provides five dedicated boundary-scan signals:

- Test data input (TDI) and test data output (TDO). The TDI and TDO signals are used to input and output instructions and data to the scan registers.

- Test mode select (TMS)—The TAP controller controls boundary-scan operations through commands received by means of the TMS signal.

- Test reset ($\overline{\text{TRST}}$)—The $\overline{\text{TRST}}$ signal is used to reset the TAP controller asynchronously. Asserting the $\overline{\text{TRST}}$ signal at power-on reset assures that the boundary-scan logic does not interfere with the 602's normal operation.

- Test clock (TCK)—Boundary-scan data is latched by the TAP controller on the rising edge of the TCK signal.

Section 7.2.10, "JTAG/Scan Interface Signals," provides additional detail about the operation of these signals.
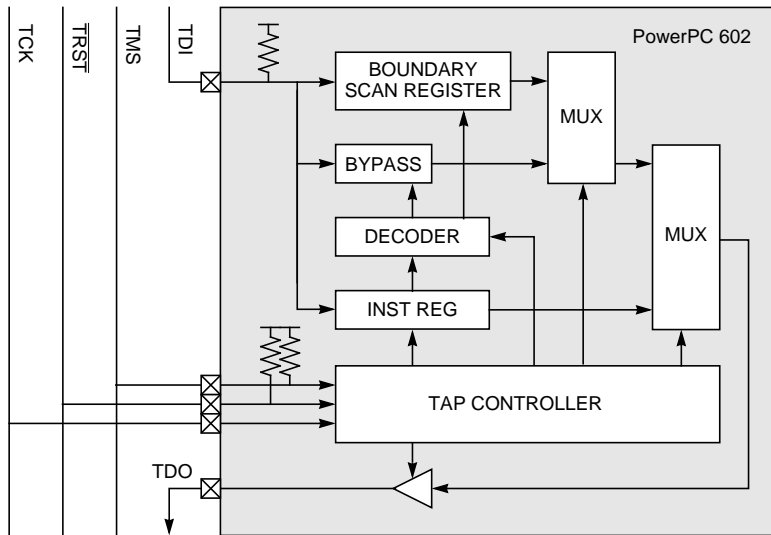
Figure C-1. Boundary-Scan Interface Block Diagram

C.1.2 Boundary-Scan Registers and Scan Chains

The 602 implements the bypass, boundary-scan, and instruction registers and their associated scan chains. These resources are described in the following sections.

C.1.2.1 Bypass Register

The bypass register is a single-stage register used to bypass the boundary-scan register of the 602 during board-level boundary-scan operations involving components other than the 602. Using the bypass register reduces the total scan string size of the boundary-scan test. The bypass register is accessed by the BYPASS instruction.

C.1.2.2 Boundary-Scan Registers

The boundary-scan interface provides a chain of registers dedicated to boundary-scan operations. These registers are not shared with any of the 602's functional registers. The boundary-scan register chain includes registers that control the direction of the output drivers and registers that reflect the signal value received or driven.

The boundary-scan registers capture the input or output state of the 602's signals during a Capture_DR TAP controller state. When a data scan is initiated following the Capture_DR state, the sampled values are shifted out through the TDO output while new boundary-scan register values are shifted in through the TDI input. At the end of the data scan operation, the boundary-scan registers are updated with the new values during an Update_DR TAP controller state.

C.1.2.3 Compliance-Enable Signals

Note that the $\overline{\text{LSSD\_MODE}}$, $\overline{\text{L1\_TEST\_CLK}}$, and $\overline{\text{L2\_TEST\_CLK}}$ signals (used for factory testing) are not included in the boundary-scan register chain. These signals, along with $\overline{\text{HRESET}}$ and $\overline{\text{CKSTP\_IN}}$, are compliance-enable signals for boundary-scan testing.

C.1.3 Instruction Register

The 8-bit instruction register serves as an instruction and status register. As TAP controller instructions are scanned in through the TDI input, the TAP controller status bits are scanned out through the TDO output.

C.1.4 TAP Controller

The 602 provides a TAP controller that controls instruction and data scan operations. The TMS signal controls the state transitions of the TAP controller.

C.2 Unimplemented IEEE 1149.1 Features

The 602 supports IEEE 1149.1 JTAG definition with the following exceptions.

1. A hard reset sequence with $\overline{\text{HRESET}}$ as well as the test reset with $\overline{\text{TRST}}$, is required before using the EXTEST or SAMPLE/PRELOAD instructions. As specified by IEEE 1149.1, the $\overline{\text{TRST}}$ signal must completely reset all logic circuits that can affect the boundary-scan operation. The 602 $\overline{\text{TRST}}$ signal does not completely reset all boundary-scan-related logic, and as a consequence, power-on-reset must be executed before executing the boundary-scan instructions.

2. Asserting $\overline{\text{HRESET}}$ and $\overline{\text{CKSTP\_IN}}$ input signals can cause the system logic to interfere with the operation of the IEEE 1149.1 EXTEST and SAMPLE/PRELOAD instructions. As a workaround for this noncompliant feature, the boundary-scan description language (BSDL) file provided for the 602 defines the $\overline{\text{HRESET}}$ and $\overline{\text{CKSTP\_IN}}$ signals as compliance-enable signals with a compliance pattern with the signals held high (unasserted). The input boundary cells originally associated with the two signals are defined as internal cells. These two signals cannot be interconnection tested by boundary-scan ATPG tools.

3. Internally-generated checkstop conditions interfere with the operation of EXTEST and SAMPLE/PRELOAD. The workaround to this issue is to execute a power-on-reset, or otherwise ensure that no internal machine checkstop has occurred prior to the execution of the boundary-scan instructions.

4. Asserting $\overline{\text{HRESET}}$ causes $\overline{\text{BR}}$, $\overline{\text{QREQ}}$, and $\overline{\text{RESETO}}$ to be driven at high-impedance; however, the boundary-scan chain does not include any registers to control the enable of the output drivers for these signals.

All other boundary-scan features are compliant with the IEEE 1149.1 specification.

C.3 Boundary-Scan Instructions

The 602 supports the mandatory IEEE 1149.1 instructions, BYPASS, SAMPLE/ PRELOAD, and EXTEST except for the noncompliant features described in Section C.2, "Unimplemented IEEE 1149.1 Features." As long as the board-level test includes a hard reset sequence, interconnections should be able to be tested by using board-level boundary-scan ATPG tools.

All instruction opcodes except the three mandatory instructions are reserved as private instructions.

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. Modified cache blocks that are written back to memory when a snoop miss causes the least-recently used cache block to be replaced.

Context synchronization. Context synchronization as the result of the execution 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. Floating-point load instructions move data from memory to FPRs, and floating-point 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 significant digits of a floating-point number. The placement of the binary point is determined by the value of the exponent.

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   **Out-of-order.** Not occurring in strict program sequence; speculative. An operation is said to be out-of-order when its results are not guaranteed to be required by the sequential execution model, such as the execution of an instruction that follows another instruction that may alter the instruction flow. For example, execution of instructions in an unresolved branch are considered out-of-order, as is the execution of an instruction behind another instruction that may yet cause an exception. The results of operations that are performed out-of-order are not committed to architected resources until it can be ensured that these results adhere to the in-order, or sequential, execution model.

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 a group of events (for example instruction execution) into distinct steps so that multiple steps can be performed at the same time.

Precise exceptions. An exception mechanism by which the instruction pipeline can be stopped so the instructions that preceded the faulting instruction or event can complete, no results of subsequent instructions will have affected architected resources, and execution can resume with the next instruction in program order. A PowerPC-based system is precise unless one of the imprecise modes for invoking the floating-point enabled exception is in effect.

Q   **Quiet NaNs (QNaNs)**. Represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when invalid. QNaNs propagate through almost every arithmetic operation without signaling exceptions.

S   **Signaling NaNs (SNaNs)**. NaNs that 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 only can 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.

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.

PowerPC 602 RISC Microprocessor User's Manual

INDEX

INDEX

INDEX

Changed (C) bit maintenance
    recording, 5-17, 5-29–5-32
Checkstop
    signal, 7-24
    state, 4-22
$\overline{CI}$ signal, 7-13
Classes of instructions, 2-30
Clean block operation, 3-20
Clock signals
    CLK_OUT, 2-14, 7-29
    PLL_CFG0–PLL_CFG3, 7-30, 8-5
    SYSCLK, 7-29, 8-5
cmp, 2-37
cmpi, 2-37
cmpl, 2-37
cmpli, 2-37
cntlzw, 2-38
Compare instructions, 2-45, A-18
Complete/writeback stage, 6-4
Completion considerations, 6-11
Context synchronization, 2-34
COP/scan interface, 7-27
Copy-back mode, 6-18
CR logical instructions, 2-55, A-24
crand, 2-55
crandc, 2-55
creqv, 2-55
crnand, 2-55
crnor, 2-55
cror, 2-55
crorc, 2-55
crxor, 2-55

D

D0–D63, 7-20, 8-14, 8-23
Data bus
    basic transactions
        32-bit, 8-37
        64-bit, 8-29
    data tenure, 8-7
    data transfer, 7-20, 8-18–8-19
    data transfer termination, 7-22
    data transfers, alignment, 2-27
    optional 32-bit mode, 8-5
    $\overline{T32}$ signal, 32-/64-bit mode, 8-28
Data cache
    basic operations, 3-7
    cache control, 3-6
    configuration, 3-2
    DCFI, DCE, DLOCK bits, 3-6
    single-ported tags, 8-2
Data cache fill, 3-7
Data phase signals, 7-20, 8-14, 8-23

Data storage interrupt (DSI) *see* DSI exception
Data TLB miss on load exception, 4-33
Data TLB miss on store exception, 4-34
Data transfer signals
    D0–D63, 7-20, 8-14, 8-23
    $\overline{T32}$, 3-8, 7-22, 8-28
Data transfer termination signals
    $\overline{TA}$, 7-22, 8-24
    $\overline{TEA}$, 7-23, 8-24
dcbf, 2-61, 3-23
dcbi, 2-64
dcbst, 2-61, 3-23
dcbt, 2-60, 3-22
dcbtst, 2-60, 3-22
dcbz, 2-61, 3-22
DCMP and ICMP registers (602-specific), 2-11,
    2-15, 5-43
Decode stage, 6-3
Decrementer interrupt, 4-30, 9-1
Defined instruction class, 2-31
Denormalized number, support, 2-40
Dispatch considerations, 6-11
divw, 2-37
divwu, 2-37
DMISS and IMISS registers (602-specific), 2-11,
    2-15, 5-43
dsa (602-specific), 2-62, 2-66, 2-72, 4-1
DSI exception, 4-23

E

Effective address calculation
    address translation, 5-8
    branches, 2-33, 2-54
    loads and stores, 2-33, 2-47, 2-51
eieio, 2-59, 3-24
Emulation trap exception, 4-39
eqv, 2-38
esa (602-specific)
    description, 2-62, 2-67
    esa access and MMU, 5-6, 5-62
    supervisor-level access, 1-26, 2-71–2-75
    system call exception, differences in use, 2-75
ESASRR register (602-specific), 2-11, 2-19
Exceptions
    alignment exception, 4-26
    causing conditions, 4-4
    data TLB miss on load, 4-33
    data TLB miss on store, 4-34
    decrementer interrupt, 4-30
    DSI exception, 4-23
    emulation trap exception, 4-39
    enabling and disabling, 4-14
    exception classifications, 4-2

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX