IDT79RC64474 & IDT7RC64475 RISController™ 64-bit Embedded Microprocessor Hardware Reference Manual

Version 1.0
February 1999

6024 Silver Creek Valley Road, San Jose, California 95138
Telephone: (800) 345-7015 • (408) 284-8200 • FAX: (408) 284-2775
Printed in U.S.A.

About This Manual

Notes

Introduction

This hardware user's manual includes both hardware and software information on the RC64474 and RC64475, two new high-performance 64-bit microprocessors that extend the RISCore4000 family choices offered by Integrated Device Technology (IDT). Operational overviews, functional descriptions, diagrams, and flowcharts are provided to assist system developers in obtaining optimum device performance.

Additional Information

Information not included in this manual such as mechanicals, package pin-outs and electrical characteristics can be found in the data sheet for these devices, which is available from the IDT website **(www.idt.com)** as well as through your local IDT sales representative.

Content Summary

Chapter 1, "RC64474/RC64475 Overview," provides a complete introduction to the performance capabilities of these two 64-bit processors. Included in this chapter is a summary of features for the devices as well as a table that provides a feature's summary of IDT's RISCore4000 family.

Chapter 2, "CPU Instruction Set Overview," presents a general overview on the three CPU instruction set formats of the MIPS architecture. Instruction set summary tables are also provided.

Chapter 3, "RISCore4000 Pipeline," discusses the operation of the 5-stage pipeline design used in the RISCore4000 CPU core as well as exception and interlock handling.

Chapter 4, "Memory Management Unit," contains a discussion on the virtual-to-physical address translation, Translation Lookaside Buffer management and the User, Kernel, and Supervisor modes of operation.

Chapter 5, "System Control Coprocessor (CPO)," contains illustrations, definitions and descriptions of the memory management and exception processing registers. An explanation on the address translation process is also provided.

Chapter 6, "CPU Exception Processing," contains illustrattions, definitions and descriptions of the various exception processing registers and discusses the exceptions and handling processes involved.

Chapter 7, "The Floating-Point Unit," describes the RISCore4000 floating-point status and control registers and includes a programming model, single and double precision floating-point operation formats as well as information on the RISCore4000 floating-point pipeline.

Chapter 8, "Floating-Point Exceptions," provides an explanation of the floating-point unit exception types; exception trap processing; exception flags; saving and restoring state, when handling an exception, and trap handlers for IEEE Standard 754 exceptions.

Chapter 9, "Processor Signal Descriptions," describes the signals used by and in conjunction with the RC64474 and RC64475 processors. Tables provide the signal name, definition, direction and description. Each table listed is according to functional groupings.

Chapter 10, "The Clocking, Reset and Initialization Interface," contains timing diagrams and descriptions on the basic system clocks and timing parameters used in the RC64474 and RC64475 processors. A processor signal-summary table is provided that lists each signal and their possible states.

Chapter 11, "Cache Organization, Operation and Coherency," describes the on-chip cache memory attributes and accessibility. Cache states, cache line ownership and the cache locking feature are also discussed.

Notes

Chapter 12, "System Interface Overview," explains the system interface process from the standpoint of both the processor and the external agent.

Chapter 13, "The Read Interface," discusses specifics of the read interface and read operation processes. Processor read protocols are defined, request and response timing diagrams are included.

Chapter 14, "The Write Interface," discusses specifics of the Write protocol and associated operations. Timing diagrams are included.

Chapter 15, "Processor Interrupts," provides information on the hardware and software interrupt processes of the RC64474 and RC64475 processors.

Chapter 16, "Processor Error Checking," describes parity error detection as well as the error checking and correcting processes for internal transactions. A table is provided that summarizes these error checking operations.

Chapter 18, "Standard JTAG Support Interface," introduces the standard JTAG interface used for board-level debugging. A pin description table provides a description of the 6 pins added to support this feature. A description on the Test Access Port (TAP) interface and TAP controller state assignments is also included.

Appendix A, "Cache Operations' Timing," lists cycle operation counts and caveats for RC64474/RC64475 cache operations timing.

Appendix B, "Entering Standby Mode," details the power management feature of the RC64474 and RC64475 processors. A flowchart on the standby operation is included.

Appendix C," Coprocessor 0 Hazards," identifies the RC64474/475 coprocessor 0 hazards.

Table of Contents

Notes

Notes

Notes

Notes

Notes

Notes

Notes

14 The Write Interface

Notes

Notes

Notes

List of Tables

Notes

Notes

Notes

Notes

List of Figures

Notes

Notes

Notes

Notes

RC64474/RC64475 Overview

Notes

Introduction

Designed to target applications that require high bandwidth, real-time response and rapid data processing, Integrated Device Technology's (IDT) RC64474 and RC64475 processors are high performance devices that extend IDT's RISCore4000 processor family choices.

IDT's RISCore4000 is a 250MHz 64-bit execution core that utilizes a 5-stage scalar pipeline. The RISCore4000 implements the MIPS-III Instruction Set Architecture (ISA) and is upwardly compatible with applications that run on earlier generation parts. The core is capable of executing in either big- or little-endian byte ordering, without performance loss for either mode.

Implementation of the MIPS-III architecture results in 64-bit operations, improved performance for commonly used code sequences in operating kernels and faster execution of floating-point intensive applications. All resource dependencies are made transparent to the programmer, insuring transportability around implementations of the MIPS ISA.

The RISCore4000 can operate on both 32- and 64-bit data, utilizing 32 general-purpose registers (GPR) that are used for integer operations and address calculation. Also, an on-chip floating-point coprocessor adds 32 floating-point registers and a floating-point control/status register. The RC64474 and RC64475 enhance IDT's entire RISCore4000 series through the implementation of features such as boundary scan, to facilitate board-level testing; enhanced timing protocols for SyncDRAM, to simplify system implementation and improve performance; on-chip resources such as larger caches and a translation lookaside buffer (TLB), for higher level integration; and 5V tolerant I/Os, to enable interfacing with earlier generation 5V devices.

The RC64474 is packaged in a 128-pin footprint QFP package and uses a 32-bit external bus, offering the ideal combination of 64-bit processing power and 32-bit low-cost memory systems. The RC64475 is packaged in a 208-pin QFP footprint package and uses the full 64-bit external bus. The RC64475 is ideal for applications requiring 64-bit performance and 64-bit external bandwidth.

Performance

The RC64474/475 bring high performance to lower-cost systems, through separate 16KB on-chip two-way set associative caches; minimized branch and load delays through a simple streamlined pipeline; up to 4GB/s aggregate bandwidth support or system interface to 1GB; a simple input clock strategy; and facilities such as early restart for data cache misses. The RC64474/475 processors are rated at 330 Dhrystone MIPS and 125 Million floating point operations per second (MFLOP/s), at 250 Mhz. The internal cache bandwidth for these devices is 3GB/s.

Device Compatibility

The RC64474/475 are application-software compatible with IDT's entire RISController™ series of embedded microprocessors, including devices from the RISCore3000,RISCore4000, RISCore5000, and RISCore32300 families. Also, through full pin and socket compatibility, the RC64474/475 offer a direct migration path for designs based on IDT's RC4640 and RC4650 processors. All devices in the RISCore4000 family use the same bus protocols and syntax, enabling a range of designs and support chips.

Notes

RISCore4000 Family

	RC4640	RC4650	RC4700	RC64474	RC64475
CPU	64-bit RISCore4000 w/ DSP extensions	64-bit RISCore4000 w/ DSP extensions	64-bit RISCore4000	64-bit RISCore4000	64-bit RISCore4000
Performance	260MIPS @200MHz	260MIPS @200MHz	260MIPS @200MHz	330MIPS @200MHz	330MIPS @200MHz
FPA	60 mflops, single precision only	60 mflops, single precision only	100 mflops, single and double precision	125 mflops, single and double precision	125 mflops, single and double precision
Caches	8kB/8kB, 2-way, lockable	8kB/8kB, 2-way, lockable	16kB/16kB, 2-way	16kB/16kB, 2-way , lockable	16kB/16kB, 2-way, lockable
External Bus	32-bit	32- or 64-bit	64-bit	32-bit, Superset pin compatible w/RC4640	32- or 64-bit, Superset pin compatible w/ RC4650
Voltages	3.3V	3.3V	3.3V	3.3V, 5V tolerant IO	3.3V, 5V tolerant IO
Packages	128 PQFP	208 MQUAD	208 MQUAD	128 QFP	208 QFP
MMU	Base-Bounds	Base-Bounds	96 page TLB	96 page TLB	96 page TLB
Key Features	Cache locking, on-chip MAC, 32-bit external bus	Cache locking, on-chip MAC, 32-bit & 64 bit bus option	Large Primary caches	Cache locking, JTAG, syncDRAM mode, 32-bit external bus	Cache locking, JTAG, syncDRAM mode, 32- 64-bit bus option

Table 1.1 Summary of Features for the RISCore4000 Family

Summary of Features

The RC64474 and RC64475 uniquely achieve high performance levels, while providing cost-effective solutions, through features such as those listed below. In addition, an array of hardware and software tools are available to assist system designers in the rapid development of RC64474 or RC64475 based systems, which allows a wide variety of customers to take full advantage of the processor's high-performance features while addressing today's aggressive time-to-market demands.

- *High performance 64-bit microprocessor, based on the RISCore4000*
  - *Minimized branch and load delays, through streamlined 5-stage scalar pipeline.*
  - *Single and double precision floating-point unit*
  - *125 peak MFLOP/s at 250 MHz*
  - *330 Dhrystone MIPS at 250 MHz*
  - *Flexible RC4700 compatible MMU*
  - *On-chip TLB, for virtual-to-physical address mapping*
- *On-chip two-way set associative caches*
  - *16KB instruction cache (I-cache)*
  - *16KB data cache (D-cache)*
  - *Write-through and write-back support*
  - *Critical word first with early restart*
- *I-cache and D-cache locking facility, provides improved real-time support*
- *Enhanced, flexible bus interface allows simple, low-cost design*
  - *64-bit Bus Interface option, 1GB/s bandwidth support*
  - *32-bit Bus Interface option, 5GB/s bandwidth support*
  - *SDRAM timing protocols*
  - *RC4000/RC5000 family compatibility*
- *Implements MIPS-III Instruction Set Architecture (ISA)*
- *3.3V with 5V tolerant I/O*
- *Software compatible with entire RISController Series of Embedded Microprocessors*
- *Industrial temperature range support*
- *Active power management*
  - *Powers down inactive units*
  - *Ensures low device cost through lower-cost package options*

Notes

- *Superset pin compatibility between RC64474 and RC4640*
- *Superset pin compatibility between RC64475 and RC4650*
- *RC64474 available in 128-pin QFP footprint package, for 32-bit only systems*
- *RC64475 available in 208-pin PQUAD package, for full 64/32 bit systems*
- *Simplified board-level testing, through full JTAG boundary scan*
- *Windows® CE compliant*

Device Overview

Figure 1.1 contains an illustration of the key functional elements of the RC64474/RC64475 processors. An overview on these elements follows. Operational details are provided throughout the manual.

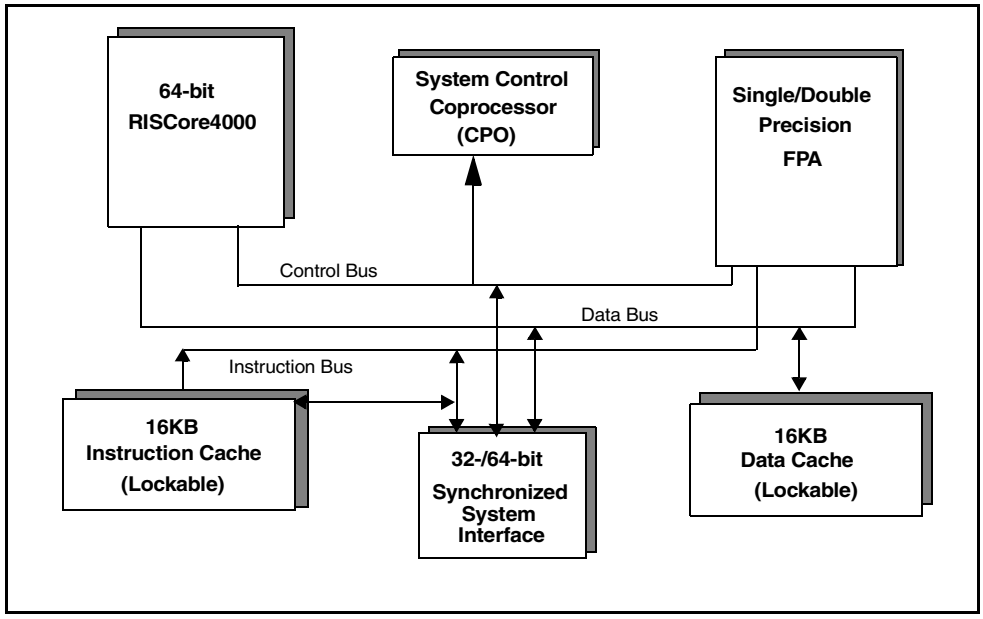


Figure 1.1 RC64474/RC64475 Functional Block Diagram

RISCore4000 Pipeline

Similar to the RISCore3000 and RISCore32300 families, the RISCore4000 (RC4000) execution core uses a 5-stage scalar pipeline, which achieves an instruction execution rate approaching one instruction per CPU cycle. The simplicity of this pipeline allows the RC64474/475 to be lower cost, lower powered processors than super-scalar or super-pipelined processors: unlike superscalar processors, applications that have large data dependencies or require a great deal of load/stores can still achieve levels that are close to the peak performance of the processor.

RISCore4000 Registers

Consistent with the MIPS-III ISA, the execution core can operate on both 32-or 64-bit data, using thirty-two 64-bit general purpose registers, which are used for scalar integer operations and address calculation. The register file consists of two read ports and one write port and is fully bypassed, to minimize operation latency within the pipeline. The CPU registers are shown in Figure 1.2.

Notes

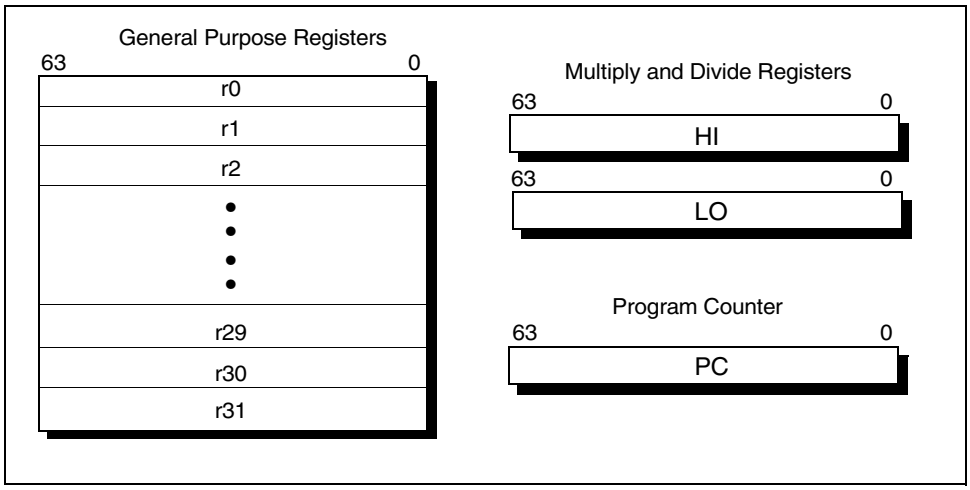


Figure 1.2 RC64474/RC64475 CPU registers

The **r0** register is hardwired to a value of zero and can be used as the target register for any instruction whose result is to be discarded. This register can also be used as a source when a zero value is needed. The Jump and Link (JAL) and BAL series of instructions use the **r31** register as an implicit return destination address register. The CPU registers also include three special purpose registers—PC, HI, and LO:

- ***PC*** *is a Program Counter register that contains the address of the instruction in the program being executed.*
- ***HI*** *is a Multiply and Divide register, higher result, and* ***LO*** *is a Multiply and Divide register, lower result. These two Multiply and Divide registers will store* **1)** *the product of integer multiply operations, or* **2)** *the quotient (in LO) and remainder (in HI) of integer divide operations.*

Note that the RC64474/475 processors do not have a Program Status Word (PSW) register. The PSW function is covered by the Status and Cause registers incorporated within the System Control Coprocessor (CP0). The CPO registers are discussed in detail in Chapter 4.

CPU Instruction Sets

MIPS-ISA instructions are 32-bits long and are grouped into three instruction formats: immediate (I-type), jump (J-type), and register (R-type). Instruction decoding is sped up through limiting the number of formats to three, and more complicated (and less frequently used) operations and addressing modes can be synthesized by the compiler, using sequences of these same basic instructions. CPU instruction sets can be divided further into the following groups:

- ***Load and Store*** *instructions move data between memory and general registers. They are all immediate (I-type) instructions, since the only addressing mode supported is base register plus 16-bit, signed immediate offset.*
- ***Computational*** *instructions perform arithmetic, logical, shift, multiply, and divide operations on values in registers. They include register (R-type, in which both the operands and the result are stored in registers) and immediate (I-type, in which one operand is a 16-bit immediate value) formats.*
- ***Jump and Branch*** *instructions change the control flow of a program. Jumps are always made to a paged, absolute address formed by combining a 26-bit target address with the high-order bits of the Program Counter (J-type format) or register address (R-type format). Branches have 16-bit offsets relative to the program counter (I-type). Jump And Link instructions save their return address in register 31.*
- ***Coprocessor*** *instructions perform operations in the coprocessors. Coprocessor load and store instructions are I-type.*
- ***Coprocessor 0*** *(system coprocessor) instructions perform operations on CP0 registers to control the memory management and exception handling facilities of the processor and the standby mode for power management.*

Notes

- ***Special*** *instructions perform system calls and breakpoint operations. These instructions are always R-type.*
- ***Exception*** *instructions cause a branch to the general exception-handling vector based upon the result of a comparison. These instructions occur in both R-type (both the operands and the result are registers) and I-type (one operand is a 16-bit immediate value) formats.*

Chapter 2 contains more details on the MIPS instruction set formats and operations mentioned above and includes instruction set summary tables.

Data Formats and Addressing

The RC64474/475 processors use four data formats: a 64-bit doubleword, a 32-bit word, a 16-bit halfword, and an 8-bit byte. Byte ordering within each of the larger data formats—halfword, word, doubleword—can be configured in either big- or little-endian[1] order.

Figure 1.3 and Figure 1.4 show the ordering of bytes within words and the ordering of words within multiple-word structures for the big-endian and little-endian conventions. As shown, when the RC64474/475 processors are configured as a big-endian system, byte 0 is the most-significant (left most) byte, thereby providing compatibility with MC 68000 and IBM 370 conventions. Figure 1.3 illustrates the big-endian configuration.

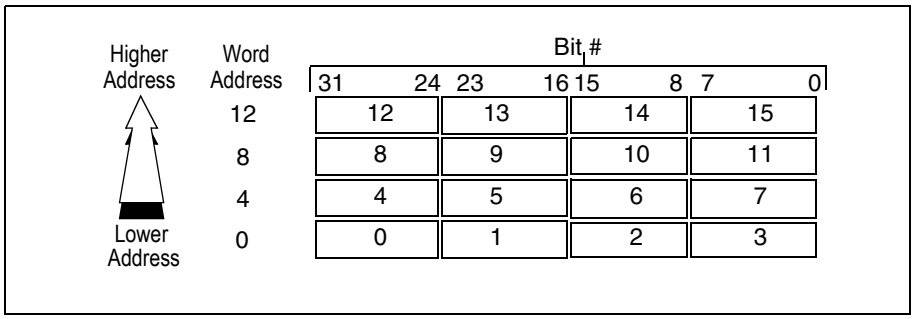


Figure 1.3 Big-Endian Byte Ordering

When configured as a little-endian system, byte 0 is always the least-significant (rightmost) byte, which is compatible with iAPX x86 and DEC VAX conventions. Figure 1.4 illustrates this configuration.

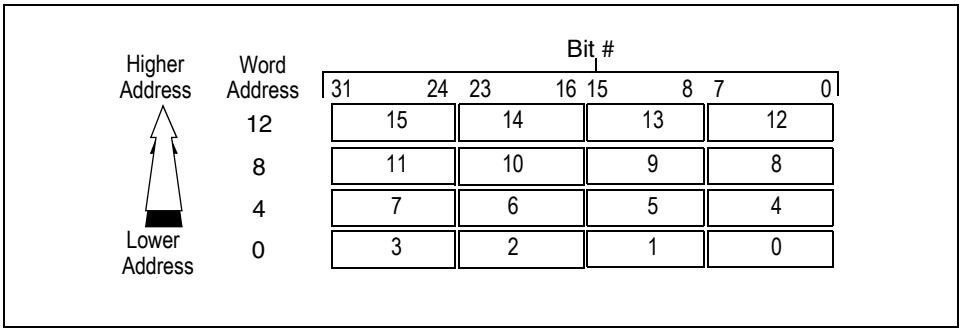


Figure 1.4 Little-Endian Byte Ordering

Throughout this text, bit 0 is always the least-significant (rightmost) bit; thus, bit designations are always little-endian (although no instructions explicitly designate bit positions within words).

Figure 1.5 and Figure 1.6 show little-endian and big-endian byte ordering in doublewords.

1. Endianness configuration refers to the location of byte 0 within the multi-byte data structure.

Notes

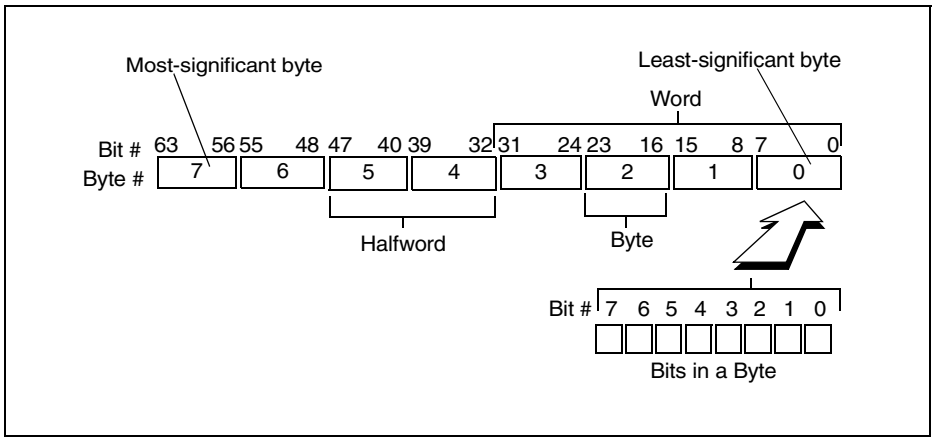


Figure 1.5 Little-Endian Data in a Doubleword

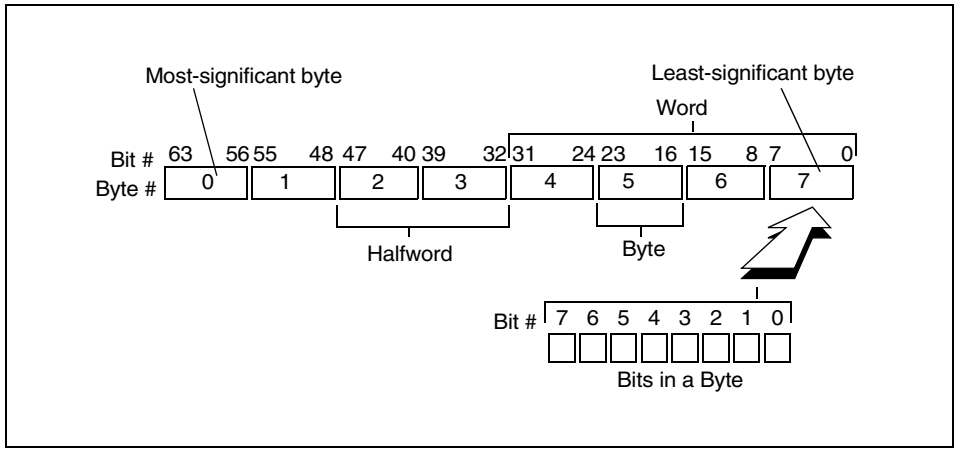


Figure 1.6 Big-Endian Data in a Doubleword

The CPU uses byte addressing for halfword, word, and doubleword accesses with the following alignment constraints:

- *Halfword accesses must be aligned on an even byte boundary (0, 2, 4...).*
- *Word accesses must be aligned on a byte boundary divisible by four (0, 4, 8...).*
- *Doubleword accesses must be aligned on a byte boundary divisible by eight (0, 8, 16...).*

The following special instructions load and store words not aligned on 4-byte (word) or 8-word (doubleword) boundaries and are used in pairs to provide addressing of misaligned words:

- *Load word left (**LWL**) and Load word right (**LWR**)*
- *Store word left (**SWL**) and Store word right (**SWR**)*
- *Load doubleword left (**LDL**) and Load doubleword right (**LDR**)*
- *Store doubleword left (**SDL**) and Store doubleword right (**SDR**)*

Addressing misaligned data incurs one additional instruction cycle over that required for addressing aligned data. This extra cycle is because of the extra instruction allowed for the instruction “pair” (for example, LWL and LWR form a pair). Also note that the CPU moves the unaligned data at the same rate as a hardware mechanism.

Figure 1.7 and Figure 1.8 show the access of a misaligned word that has byte address 3, in the big- and little-endian configurations.

Notes

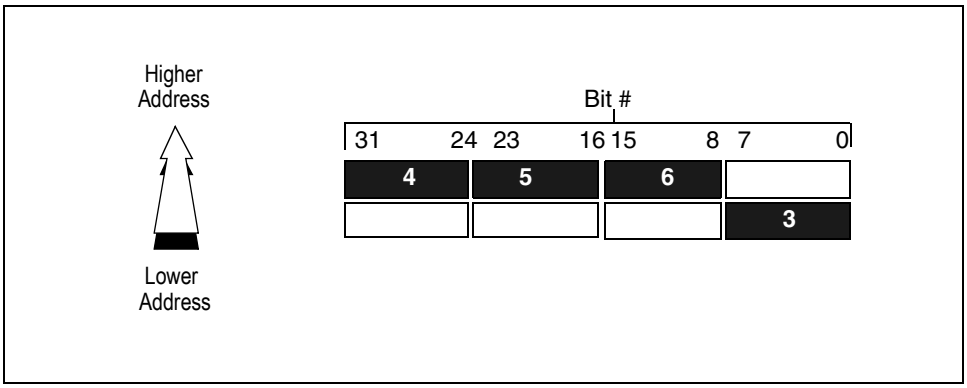


Figure 1.7 Big-Endian Misaligned Word Addressing

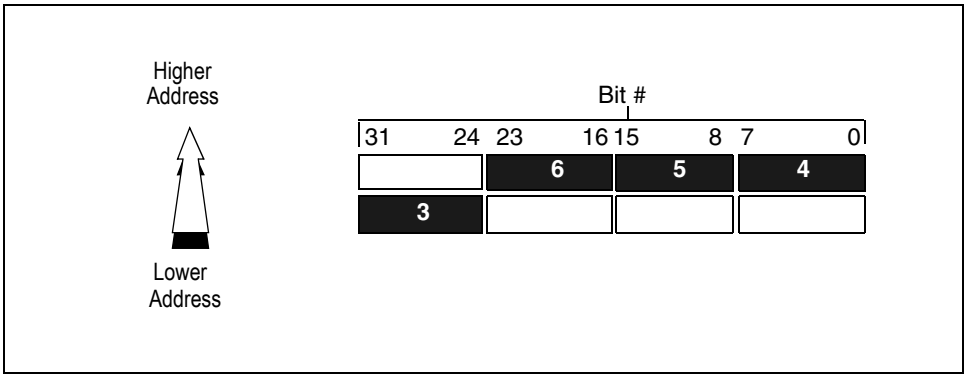


Figure 1.8 Little-Endian Misaligned Word Addressing

RISCore4000 Coprocessors

The RC64474/RC64475's execution core defines three coprocessors, designated CP0 through CP1 as follows:

- *Coprocessor 0 (**CP0**)—also referred to as the System Control Coprocessor—supports the virtual memory system and exception handling.*
- *Coprocessor 1 (**CP1**) implements the MIPS floating-point instruction set.*

System Control Coprocessor, CP0

CP0 translates virtual addresses into physical addresses and manages exceptions and transitions between kernel, user and supervisor states. CP0 also controls the cache subsystem, as well as providing diagnostic control and error recovery facilities.

CP0 is also used to control the power management system for the RC64474/RC64475. This standby mode can be used to reduce the power consumption of the CPU's internal core. Standby mode is entered by executing the WAIT instruction with the SysAD bus idle and is exited by any interrupt.

The CP0 registers described in Table 1.2 manipulate the memory management and exception handling capabilities of the CPU[1].

[1] The results of access to a reserved or undefined CP0 register are undefined. An exception may or may not result.

Notes

Number	Register Name	Register Type	Description
0	Index	Memory Management	Provides the User Instruction address space Base
1	Random	Memory Management	Pseudorandom pointer into TLB array (read only)
2	EntryLo0	Memory Management	Low half of TLB entry for even virtual page (VPN)
3	EntryLo1	Memory Management	Low half of TLB entry for odd virtual page (VPN)
4	Context	Exception Processing	Pointer to kernel virtual page table entry (PTE) for 32-bit address spaces
5	PageMask	Memory Management	TLB Page Mask
6	Wired	Memory Management	Number of wired TLB entries
7	Reserved		
8	BadVAddr	Exception Processing	Bad virtual address
9	Count	Exception Processing	Timer Count
10	EntryHi	Memory Management	High half of TLB entry
11	Compare	Exception Processing	Timer Compare
12	SR	Exception Processing	Status register
13	Cause	Exception Processing	Cause of last exception
14	EPC	Exception Processing	Exception Program Counter
15	PRId	Memory Management	Processor Revision Identifier
16	Config	Memory Management	Configuration register
17	LLAddr	Memory Management	Cache attributes control
18–19	Reserved		
20	XContext	Exception Processing	Pointer to kernel virtual PTE table for 64-bit address spaces.
21–25	Reserved		
26	ECC	Exception Processing	Secondary-cache error checking and correcting (ECC) and Primary parity
27	CacheErr	Exception Processing	Cache Error and Status register
28	TagLo	Memory Management	Cache Tag register
29	TagHi	Memory Management	Cache Tag register
30	ErrorEPC	Exception Processing	Error Exception Program Counter
31	Reserved		

Table 1.2 System Control Coprocessor (CP0) Register Definitions

Floating-Point Coprocessor, CP1

The floating-point coprocessor, CP1, includes 32 floating-point registers and a floating-point control/status register. CP1 forms a “seamless” interface with the integer unit, decoding and executing instructions in parallel with the integer unit.

Floating-Point Units

The floating-point execution units support single- and double precision arithmetic, as specified in the IEEE Standard 754. The execution unit is broken into a separate multiply unit and a combined add/convert/divide/square root unit. Overlap of multiplies and add/subtract is supported. The multiplier is partially pipelined, allowing a new multiply to begin every 6 cycles.

Notes

Fully precise floating-point exceptions are maintained while allowing both overlapped and pipelined operations. Precise exceptions are extremely important in mission-critical environments and highly desirable for debugging in any environment.

The floating-point unit's operation set includes floating-point add, subtract, multiply, divide, square root, conversion between fixed-point and floating-point format, and floating-point compare. These operations comply with IEEE Standard 754. Table 1.3 gives the latencies of some of the floating-point instructions in internal processor cycles.

Operation	Single Precision Latency	Double Precision Latency
ADD	4	4
SUB	4	4
MUL	4	8
DIV	32	61
SQRT	31	60
CMP	3	3
FIX	4	4
FLOAT	6	6
ABS	1	1
MOV	1	1
NEG	1	1
LWC1,LDC1	2	2
SWC1, SDC1	1	1

Table 1.3 Single and Double Precision Latency Cycles

Address Mapping

Virtual-to-physical address mapping is available to system software to provide a secure environment for user processes. Bits in a status register determine the mode of operation. In the RISCore4000 the **user**, **supervisor** and **kernel** modes of operation are available.

Typically used for application's programs, RISCore4000's user mode provides a single, uniform virtual address space of 256GB (2GB when Status.UX = 0). When operating in the kernel mode—typically used for exception handling and operating system kernel functions—four distinct virtual address spaces, totalling 1024GB (4GB when Status.KX = 0), are simultaneously available and are differentiated by the high-order bits of the virtual address.

The RC64474/475 processors also support a supervisor mode in which the virtual address space is 256.5GB (2.5GB when Status.SX = 0), divided into three regions based on the high-order bits of the virtual address. When the CPU uses 64-bit virtual addresses, the address space layouts are an upwardly compatible extension of the 32-bit virtual address space layout.

Joint Translation-Lookaside Buffer (JTLB)

For fast virtual-to-physical address decoding, the RC64474/RC64475 uses a large, fully associative TLB, which maps 96 virtual pages to their corresponding physical addresses. The TLB is organized as 48 pairs of even-odd entries, and maps a virtual address and address space identifier into the large, 64GB physical address space.

Notes

Two mechanisms are provided to assist in controlling the amount of mapped space, and the replacement characteristics of various memory regions. First, the page size can be configured—on a per-entry basis—to map a page size of 4KB to 16MB (in multiples of 4). A CP0 register is loaded with the page size of a mapping, and that size is entered into the TLB when a new entry is written. Thus, operating systems can provide special purpose maps; for example, a typical frame buffer can be memory mapped using only one TLB entry.

The second mechanism controls the replacement algorithm when a TLB miss occurs. The RC64474/475 provides a random replacement algorithm to select a TLB entry to be written with a new mapping; however, the processor provides a mechanism whereby a system specific number of mappings can be locked into the TLB, and thus avoid being randomly replaced. This facilitates the design of real-time systems, by allowing deterministic access to critical software.

The joint TLB also contains information to control the cache coherency protocol for each page. Specifically, each page has attribute bits to determine whether the coherency algorithm is: uncached, non-coherent write-back, non-coherent write-through write-allocate, non-coherent write-through no write-allocate, sharable, exclusive, or update. Non-coherent write-back is typically used for both code and data on the RC64474/475; the write-through modes support more efficient frame buffer accesses than the original MIPS R4000. The coherent modes are supported for R4000 compatibility and generate different transaction types on the system interface; however, hardware based cache coherency is not supported.

Instruction Translation Lookaside Buffer (ITLB)

The RISCore4000 also incorporates a 2-entry instruction TLB. Each entry maps a 4KB page. The instruction TLB improves performance by allowing instruction address translation to occur in parallel with data address translation. When a miss occurs on an instruction address translation, the least-recently used ITLB entry is filled from the JTLB. The operation of the ITLB is invisible to the user.

Data Translation (DTLB)

The RISCore4000 also incorporates a 4-entry data TLB. Each entry maps a 4KB page. The data TLB improves performance by allowing data address translation to occur in parallel with data address translation. When a miss occurs on an data address translation, the DTLB is filled from the JTLB. The DTLB refill is pseudo-LRU: the least recently used entry of the least recently used half is filled. The operation of the DTLB is invisible to the user.

Cache Memory

To keep the RISCore4000's high-performance pipeline full and operating efficiently, on-chip instruction and data caches that can be accessed in a single processor cycle are incorporated. Each cache has its own 64-bit data path and can be accessed in parallel. The cache subsystem provides the integer and floating-point units with an aggregate bandwidth of 4GB per second @ 250MHz.

Instruction Cache (I-Cache)

The RC64474/475 incorporate a two-way set associative on-chip instruction cache. This virtually indexed, physically tagged I-cache is 16KB in size and is word parity protected.

Because the cache is virtually indexed, the virtual-to-physical address translation occurs in parallel with the cache access, thus further increasing performance by allowing these two operations to occur simultaneously. The tag holds 24-bits of the physical address and valid bit and is parity protected.

The instruction cache is 64-bits wide and can be refilled or accessed in a single processor cycle. Instruction fetches require only 32 bits per cycle, for a peak instruction bandwidth of 1GB/sec at 250MHz. Sequential accesses take advantage of the 64-bit fetch to reduce power dissipation, and cache miss refill writes 64 bits per cycle to minimize the cache miss penalty. To maximize performance, the line size is eight instructions (32 bytes).

Notes

In addition, the contents of one set of the instruction cache (set “A”) can be “locked” by setting a bit in a CP0 register. Locking the set prevents its contents from being overwritten by a subsequent cache miss; refill occurs then only into “set A”. This operation effectively “locks” time critical code into one 8KB set, while allowing the other set to service other instruction streams in a normal fashion. Thus, the benefits of cached performance are achieved, while deterministic real-time response is preserved.

Data Cache (D-Cache)

For fast, single cycle data access, the RC64474/475 include a 16KB on-chip data cache that is two-way set associative with a fixed 32-byte (eight word) line size. Both the D-cache and the I-cache can be accessed each pipeline cycle; thus, the data bandwidth is 2GB/sec at 250 MHz, in addition to the 1GB/sec instruction bandwidth. The data cache is protected with byte parity and its tag is protected with a single parity bit. It is virtually indexed and physically tagged to allow simultaneous address translation and data cache access.

In addition, the contents of one set of the data cache (set “A”) can be “locked” by setting a bit in a CP0 register. Locking the set prevents its contents from being overwritten by a subsequent cache miss; refill occurs then only into “set A”. This operation effectively “locks” time critical code into one 8KB set, while allowing the other set to service other instruction streams in a normal fashion. Thus, the benefits of cached performance are achieved, while deterministic real-time response is preserved.

Associated with the data cache is the store buffer. When the RISCore4000 executes a store instruction, this single-entry buffer gets written with the store data while the tag comparison is performed. If the tag matches, then the data is written into the data cache in the next cycle that the data cache is not accessed (the next non-load cycle). The store buffer allows the RISCore4000 to execute a store every processor cycle and to perform back-to-back stores without penalty.

Write Buffer

Writes to external memory—whether they are cache miss write-backs, stores to uncached or write-through addresses—use the on-chip write buffer. The write buffer holds up to four 64-bit address and 64-bit data pairs. The entire buffer is used for a data cache write-back and allows the processor to proceed in parallel with memory updates.

System Interfaces

The RC64475[1] supports a 64-bit system interface that is compatible with the RC4650 system interface. This interface operates from the input Reference clock. The system interface consists of a 64-bit address/data bus with eight check bits and a 9-bit command bus. Eight handshake signals and six interrupt inputs are included. The interface has a simple timing specification and is capable of transferring data between the processor and memory at a peak rate of 1GB/sec.

Additionally, the RC64474/475 supports a boot-time option to run the system interface as 32 bits wide, using basically the same protocols as a 64-bit system. This feature allows the system designer to reduce the costs of the overall memory system without sacrificing computational performance.

The RC64474/475 clocking interface allows the CPU to be easily mated with external reference clocks. The CPU input clock is the bus reference clock, and can be between 25 and 125MHz.

An on-chip phase-locked-loop (PLL) generates the pipeline clock (PClock) through multiplication of the system interface clock by values of 2,3,4,5,6,7 or 8, as defined at system reset. This allows the pipeline clock to be implemented at a significantly higher frequency than the system interface clock.

System Address/Data Bus

The System Address Data (SysAD) bus is used to transfer addresses and data between the processor and the rest of the system.

1. The RC64474 supports the 32-bit system interface only.

Notes

During 64-bit operation, SysAD transfers are protected with an 8-bit parity check bus, SysADC. When initialized for 32-bit operation, the SysAD can be viewed as a 32-bit multiplexed bus that is protected by 4 parity check bits.

To allow easier interfacing to memory and I/O systems of varying frequencies, the RC64474/475 system interface is configurable. Bus frequencies and reference timings of the RC64474/475 are derived from the input clock, and the rate at which the CPU transmits data to the system interface is programmable via boot-time mode control bits. The rate at which the processor receives data is fully controlled by the external device.

Therefore, either a low cost interface—requiring no read or write buffering—or a faster, high performance interface can be designed to communicate with the RC64474/475 devices. Again, the system designer has the flexibility to make these price/performance trade-offs.

System Command Bus

The RC64474/475 interfaces have a 9-bit bidirectional System Command (SysCmd) bus, which indicates whether the SysAD bus carries an address or data item. The RC64474/475 initiates processor requests that are responded to by an external device. External requests are issued by an external device and require the RC64474/475 devices to respond.

The RC64474/475 support single data (one to eight byte) and 8-word block transfers on the SysAD bus. In the case of a single-data transfer, the low-order 3 address bits provide the byte address of the transfer, and the SysCmd bus indicates the number of bytes being transferred.

Choosing a 32- or 64-bit wide system interface dictates whether a cache line block transaction requires 4 doubleword data cycles or 8 single word cycles, as well as whether a single data transfer—larger than 4 bytes—must be divided into two smaller transfers.

SDRAM Timing Protocols

To facilitate discrete interface to SDRAM, the RC64474/475 bus interface is enhanced with a programmable delay that is inserted between the write address and the write data, during write cycles (for both block and non-block writes).

The bus delay can be defined as 0 to 7 **MasterClock** cycles and is activated and controlled through mode bit (17:15) settings selected during the reset initialization sequence. The '000' setting provides the same write operations timing as the RC4640, RC4650, RC4700 and RC5000 processors. Figure 1.9 shows a typical system using the RC64474/475. In this example, there is SDRAM and a boot EPROM.

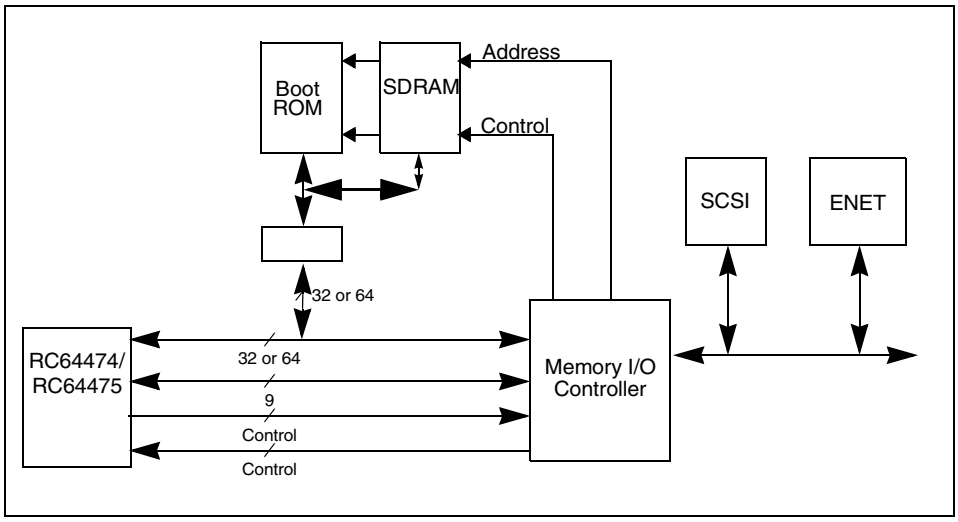


Figure 1.9 Typical System Block Diagram

Notes

Handshake Signals

The RC64474/RC64475 system interface consists of the following six handshake signals: RdRdy*, WrRdy*, ExtRqst*, Release*, ValidOut*, and ValidIn*

An external device uses RdRdy* and WrRdy* to indicate to the RC64474/475 its readiness to accept a new read or write transaction. On read and write requests, the RC64474/475 samples RdRdy* and WrRdy* before deasserting the address.

ExtRqst* and Release* are used to transfer control of the SysAD and SysCmd buses between the processor and an external device. When an external device requires control of the interface, it asserts ExtRqst*. To release the system interface to slave state, the RC64474/475 responds by asserting Release*.

The RC64474/475 and the external device, respectively, use ValidOut* and ValidIn* to indicate that there is a valid command or data on the SysAD and SysCmd buses. When the RC64474/475 is driving these buses with a valid command or data, it asserts ValidOut*. The external device drives ValidIn* when it has control of the buses and is driving a valid command or data.

Non-overlapping System Interface

The RC64474/475 require a non-overlapping system interface, compatible with the RC4640/RC4650. This means that only one processor request may be outstanding at a time and that the request must be serviced by an external device before the RC64474/475 issues another request. The RC64474/475 can issue read and write requests to an external device, and an external device can issue read and write requests to the processors.

The RC64474/475 devices assert ValidOut* while simultaneously driving the address and read command on the SysAD and SysCmd buses. If the system interface has RdRdy* or Read transactions asserted, then the processors tristate their drivers and release the system interface to slave state by asserting Release*. The external device can then begin sending data to RC64474/475.

Write Reissue and Pipeline Write

The RC64474/475 implement additional write protocols that double the effective write bandwidth. The write re-issue has a repeat rate of 2 optional cycles per write. A write issues if WrRdy* is asserted 2 cycles earlier and remains asserted at the issue cycle. If WrRdy* does not remain asserted, the last write will re-issue. Pipelined writes have the same 2-cycle per write repeat rate but can issue an additional write after WrRdy* de-asserts.

External Requests

The RC64474/475 responds to requests issued by an external device. These requests can take several forms: an external device may need to supply data in response to an RC64474/475 read request or it may need to obtain system interface bus control to access other resources that may be on that bus. Read Response and Null external requests are both supported.

Boot-Time Options

The boot-time mode control interface initializes fundamental processor modes. The boot-time mode control interface is a serial interface that operates at a very low frequency (MasterClock divided by 256). This low-frequency operation allows the initialization information to be kept in a low-cost EPROM; alternatively, the twenty-or-so bits could be generated by the system interface ASIC or a simple PAL.

To initialize all fundamental operation modes immediately after the VCCOK signal is asserted, the processor reads a serial bit stream of 256 bits. After initialization is complete, the processor continues to drive the serial clock output, but no further initialization bits are read.

Notes

CPU Instruction Set Overview

Notes

Introduction

This chapter provides a general overview on the three CPU instruction set formats of the MIPS architecture: Immediate, Jump, and Register. For details on a specific RISCore4000 instruction, refer to the *IDT MIPS Microprocessor Family Software Reference Manual*.

Instruction Formats

Each CPU instruction consists of a single 32-bit word, aligned on a word boundary. There are three instruction formats:

- *Immediate (I-type)*
- *Jump (J-type)*
- *Register (R-type)*

Limiting the number of instruction formats to three simplifies instruction decoding (thus enable higher frequency operation) and allows the compiler to synthesize more complicated (and less frequently used) operations and addressing modes. Figure 2.1 illustrates the three instruction formats of the MIPS architecture.

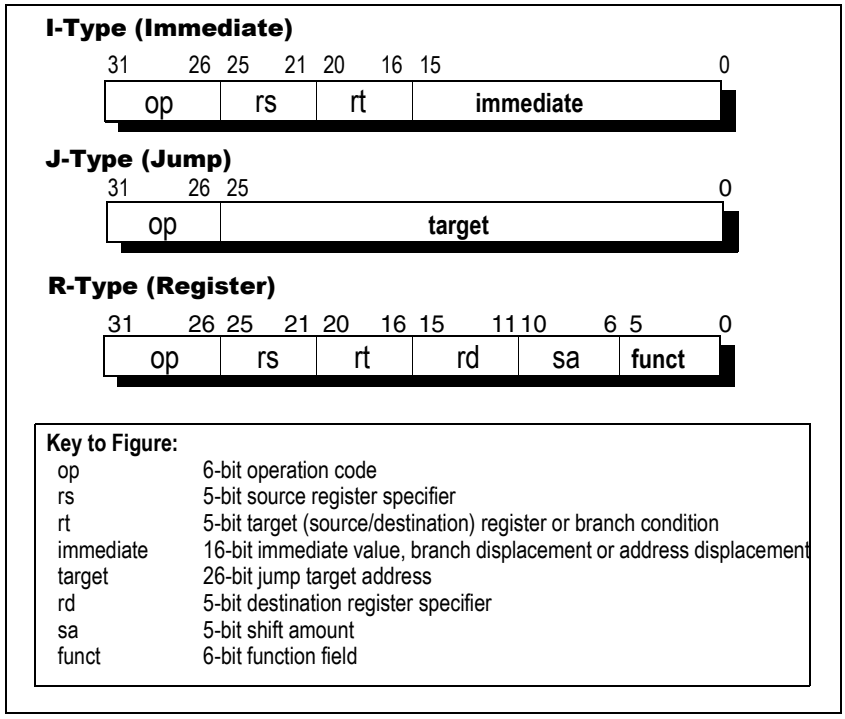


Figure 2.1 CPU Instruction Formats

Note that in the MIPS architecture, coprocessor instructions are implementation-dependent. Refer to the *IDT MIPS Microprocessor Family Software Reference Manual* for details on individual Coprocessor 0 instructions.

Load and Store Instructions (I-type)

Load and store are immediate (I-type) instructions that move data between memory and the general registers. The only addressing mode that load and store instructions directly support is *base register plus 16-bit signed immediate offset*.

Notes

Load Delay Slot Scheduling

A load instruction that does not allow its result to be used by the instruction immediately following is called a *delayed load instruction*. The instruction slot immediately following this delayed load instruction is referred to as the *load delay slot*.

In the RISCore4000, the instruction immediately following a load instruction can request the contents of the loaded register, however, in such cases, hardware interlocks insert an additional real cycles. Consequently, scheduling load delay slots can be desirable, both for performance and processor compatibility. However, the scheduling of load delay slots is not absolutely required.

Defining Access Types

Access type indicates the size of an RC64474/475 processor data item to be loaded or stored, set by the load or store instruction opcode. Memory access types are defined in Appendix A of the *IDT MIPS Microprocessor Family Software Reference Manual*.

Regardless of access type or byte ordering (endianness), the address given specifies the low-order byte in the addressed field. For a big-endian configuration, the low-order byte is the most-significant byte; for a little-endian configuration, the low-order byte is the least-significant byte.

The access type—together with the three low-order bits of the address—define the bytes accessed within the addressed doubleword, which is shown in Figure 2.2. Only the combinations shown in this table are permissible. Other combinations will cause address error exceptions.

Access Type Mnemonic (*Value*)	Low Order Address Bits			Bytes Accessed															
				Big Endian (63----------31----------0) Byte								Little Endian (63----------31----------0) Byte							
	2	1	0																
Doubleword (*7*)	0	0	0	0	1	2	3	4	5	6	7	7	6	5	4	3	2	1	0
Septibyte (*6*)	0	0	0	0	1	2	3	4	5	6			6	5	4	3	2	1	0
	0	0	1		1	2	3	4	5	6	7	7	6	5	4	3	2	1	
Sextibyte (*5*)	0	0	0	0	1	2	3	4	5					5	4	3	2	1	0
	0	1	0			2	3	4	5	6	7	7	6	5	4	3	2		
Quintibyte (*4*)	0	0	0	0	1	2	3	4							4	3	2	1	0
	0	1	1				3	4	5	6	7	7	6	5	4	3			
Word (*3*)	0	0	0	0	1	2	3									3	2	1	0
	1	0	0					4	5	6	7	7	6	5	4				
Triplebyte (*2*)	0	0	0	0	1	2											2	1	0
	0	0	1		1	2	3									3	2	1	
	1	0	0					4	5	6			6	5	4				
	1	0	1						5	6	7	7	6	5					
Halfword (*1*)	0	0	0	0	1													1	0
	0	1	0			2	3									3	2		
	1	0	0					4	5					5	4				
	1	1	0							6	7	7	6						
Byte (*0*)	0	0	0	0															0
	0	0	1		1													1	
	0	1	0			2											2		
	0	1	1				3									3			
	1	0	0					4							4				
	1	0	1						5					5					
	1	1	0							6			6						
	1	1	1								7	7							

Figure 2.2 Byte Access within a Doubleword

Notes

Computational Instructions (R-type and I-type)

Computational instructions can be in either register (R-type) format, in which both operands are registers, or immediate (I-type) format, in which one operand is a 16-bit immediate. Computational instructions perform the following operations on register values:

- *arithmetic*
- *logical*
- *shift*
- *multiply*
- *divide*

Arithmetic, logical, shift, multiply and divide operations fit into the following four categories of computational instructions:

- *ALU Immediate instructions*
- *three-Operand Register-Type instructions*
- *shift instructions*
- *multiply and divide instructions*

Operations with 32-bit Operands

Operands to 32-bit operand opcodes must be in sign-extended form. 32-bit operand opcodes include all non-doubleword operations, such as: ADD, ADDU, SUB, SUBU, ADDI, SLL, SRL, SRA, SLLV, etc. The result of operations that use incorrect sign-extended 32-bit values is unpredictable.

Cycle Timing for Multiply and Divide Instructions

If necessary, to allow complete execution of the multiply and divide instructions, RC64474/475 hardware interlocks. For example, MFHI and MFLO instructions[1] are interlocked so that any attempt to read or execute them before prior to the completion of previously issued multiply or divide instructions will be delayed.

Table 2.1 gives the number of processor cycles (PCycles) required to resolve an interlock or stall between various multiply or divide instructions and a subsequent MFHI or MFLO instruction. When reviewing Table 2.1, the following definitions apply:

- *Latency is the number of clock cycles until the result is available.*
- *Repeat is the number of clock cycles until the instruction can be repeated.*
- *Stall is the number of clock cycles the CPU will automatically stall.*

Opcode	Operand *Size	Latency	Repeat	Stall
MULT/U	16-bit	3	2	0
	32-bit	4	3	0
DMULT, DMULTU	any	6	5	0
DIV, DIVU	any	36	36	0
DDIV, DDIVU	any	68	68	0

Table 2.1 RISCore4000 Integer Multiply/Divide Operation

1. More details on CPU instruction sets are located in Appendix A of the *IDT MIPS Microprocessor Family Software Reference Manual*

Notes

Jump & Branch Instructions (J-type & R-type)

Jump and branch instructions change the control flow of a program. All jump and branch instructions occur with a delay of one instruction; that is, the instruction immediately following the jump or branch (this is known as the instruction in the *delay slot*) always executes while the target instruction is being fetched from storage[1].

Jump Instruction Overview

Subroutine calls in high-level languages are usually implemented with Jump or Jump and Link instructions, both of which are J-type instructions. In J-type format, the 26-bit target address shifts left 2 bits and combines with the high-order 4 bits of the current program counter to form an absolute address.

Returns, dispatches, and large cross-page jumps are usually implemented with the Jump Register or Jump and Link Register instructions, which are both R-type instructions that take the 32-bit or 64-bit byte address contained in one of the general purpose registers.

Branch Instruction Overview

A Branch instruction is a jump to a specified memory location and has an architectural delay of one instruction. All branch instruction target addresses are computed by adding the address of the instruction in the delay slot to the 16-bit *offset* (shifts left 2 bits and is sign-extended to 32 bits).

When a branch is taken, the instruction immediately following the branch instruction, in the branch delay slot, is executed before the branch to the target instruction takes place. There are two versions of Conditional branches, and each one treats the instruction in the delay slot differently. The “branch” instructions will execute the instruction in the delay slot, but the “branch likely” instructions do not. If a conditional branch likely is not taken, the instruction in the delay slot is nullified. For regular conditional branches, the delay slot is always executed.

Special Instructions (R-type)

Special instructions allow the software to initiate traps. Trap instructions cause exceptions conditionally based upon the result of a comparison. These special instructions are always R-type. For more information about special instructions, refer to the individual instruction as described in the *IDT MIPS Microprocessor Family Software Reference Manual*.

Exception Instructions

Exception instructions are extensions to the MIPS ISA. For more information about exception instructions, refer to the individual instruction as described in Appendix A.

Coprocessor Instructions (I-type)

To manipulate the memory management and exception handling facilities of the processor, coprocessor instructions perform operations specifically on the System Control Coprocessor registers. Coprocessor loads and stores are I-type, and coprocessor computational instructions have coprocessor-dependent formats. Individual coprocessor instructions are described in the *IDT MIPS Micrprocessor Family Software Reference Manual*.

Instruction Set Summary

Table 2.2 through Table 2.11 lists CPU instructions common to MIPS R-Series processors, along with the level in which they first appeared. The last column of each table refers to the MIPS ISA level in which the instruction first appeared. Table 2.11 shows CP0 instructions.

1. See Branch Likely for exceptions to this rule.

Notes

OpCode	Description	MIPS ISA Level†
LB	Load Byte	I
LBU	Load Byte Unsigned	I
LH	Load Halfword	I
LHU	Load Halfword Unsigned	I
LW	Load Word	I
LWL	Load Word Left	I
LWR	Load Word Right	I
SB	Store Byte	I
SH	Store Halfword	I
SW	Store Word	I
SWL	Store Word Left	I
SWR	Store Word Right	I
LD	Load Doubleword	III
LDL	Load Doubleword Left	III
LDR	Load Doubleword Right	III
LL	Load Linked	II
LLD	Load Linked Doubleword	III
LWU	Load Word Unsigned	III
SC	Store Conditional	II
SCD	Store Conditional Doubleword	III
SD	Store Doubleword	III
SDL	Store Doubleword Left	III
SDR	Store Doubleword Right	III
SYNC	Sync	II

Note: †For Tables 1.1 through 1.17 this column refers to the level in which the instruction first appeared.

Table 2.2 Instruction Set: MIPS 1 /MIPS 2/MIPS 3 Load and Store Instructions

OpCode	Description	MIPS ISA Level
ADDI	Add Immediate	I
ADDIU	Add Immediate Unsigned	I
SLTI	Set on Less Than Immediate	I
SLTIU	Set on Less Than Immediate Unsigned	I
ANDI	AND Immediate	I
ORI	OR Immediate	I
XORI	Exclusive OR Immediate	I
LUI	Load Upper Immediate	I
DADDI	Doubleword Add Immediate	III
DADDIU	Doubleword Add Immediate Unsigned	III

Table 2.3 CPU Instruction Set: MIPS 1 /MIPS 2/ MIPS 3 Arithmetic Instructions (ALU Immediate)

Notes

OpCode	Description	MIPS ISA Level
ADD	Add	I
ADDU	Add Unsigned	I
SUB	Subtract	I
SUBU	Subtract Unsigned	I
SLT	Set on Less Than	I
SLTU	Set on Less Than Unsigned	I
AND	AND	I
OR	OR	I
XOR	Exclusive OR	I
NOR	NOR	I
DADD	Doubleword Add	III
DADDU	Doubleword Add Unsigned	III
DSUB	Doubleword Subtract	III
DSUBU	Doubleword Subtract Unsigned	III

Table 2.4 CPU Instruction Set: Arithmetic (3-Operand, R-Type)

OpCode	Description	MIPS ISA Level
MULT	Multiply (result in HI/LO)	I
MULTU	Multiply Unsigned (result in HI/LO)	I
DIV	Divide	I
DIVU	Divide Unsigned	I
MFHI	Move From HI	I
MTHI	Move To HI	I
MFLO	Move From LO	I
MTLO	Move To LO	I
DMULT	Doubleword Multiply (Result in Hi/Lo)	III
DMULTU	Doubleword Multiply Unsigned (Result in Hi/Lo)	III
DDIV	Doubleword Divide	III
DDIVU	Doubleword Divide Unsigned	III
Note: †These are IDT-proprietary extensions to the MIPS instruction set.		

Table 2.5 CPU Instruction Set: MIPS 1, MIPS 2, MIPS 3 Multiply and Divide Instructions

Notes

OpCode	Description	MIPS ISA Level
J	Jump	I
JAL	Jump And Link	I
JR	Jump Register	I
JALR	Jump And Link Register	I
BEQ	Branch on Equal	I
BNE	Branch on Not Equal	I
BLEZ	Branch on Less Than or Equal to Zero	I
BGTZ	Branch on Greater Than Zero	I
BLTZ	Branch on Less Than Zero	I
BGEZ	Branch on Greater Than or Equal to Zero	I
BLTZAL	Branch on Less Than Zero And Link	I
BGEZAL	Branch on Greater Than or Equal to Zero And Link	I
BEQL	Branch on Equal Likely	II
BNEL	Branch on Not Equal Likely	II
BLEZL	Branch on Less Than or Equal to Zero Likely	II
BGTZL	Branch on Greater Than Zero Likely	II
BLTZL	Branch on Less Than Zero Likely	II
BGEZL	Branch on Greater Than or Equal to Zero Likely	II
BLTZALL	Branch on Less Than Zero And Link Likely	II
BGEZALL	Branch on Greater Than or Equal to Zero And Link Likely	II
BCzTL	Branch on Coprocessor z True Likely	II
BCzFL	Branch on Coprocessor z False Likely	II

Table 2.6 CPU Instruction Set: Jump and Branch Instruction

OpCode	Description	MIPS ISA Level
SLL	Shift Left Logical	I
SRL	Shift Right Logical	I
SRA	Shift Right Arithmetic	I
SLLV	Shift Left Logical Variable	I
SRLV	Shift Right Logical Variable	I
SRAV	Shift Right Arithmetic Variable	I
DSLL	Doubleword Shift Left Logical	III
DSRL	Doubleword Shift Right Logical	III
DSRA	Doubleword Shift Right Arithmetic	III
DSLLV	Doubleword Shift Left Logical Variable	III
DSRLV	Doubleword Shift Right Logical Variable	III
DSRAV	Doubleword Shift Right Arithmetic Variable	III
DSLL32	Doubleword Shift Left Logical + 32	III
DSRL32	Doubleword Shift Right Logical + 32	III
DSRA32	Doubleword Shift Right Arithmetic + 32	III

Table 2.7 CPU Instruction Set: Shift Instructions

Notes

OpCode	Description	MIPS ISA Level
LWCz	Load Word to Coprocessor z	I
SWCz	Store Word from Coprocessor z	I
MTCz	Move To Coprocessor z	I
MFCz	Move From Coprocessor z	I
CTCz	Move Control to Coprocessor z	I
CFCz	Move Control From Coprocessor z	I
COPz	Coprocessor Operation z	I
BCzT	Branch on Coprocessor z True	I
BCzF	Branch on Coprocessor z False	I
DMFCz	Doubleword Move From Coprocessor z	II
DMTCz	Doubleword Move To Coprocessor z	II
LDCz	Load Double Coprocessor z	II
SDCz	Store Double Coprocessor z	II

Table 2.8 Instruction Set: Coprocessor Instructions

OpCode	Description	MIPS ISA Level
SYSCALL	System Call	I
BREAK	Break	I

Table 2.9 CPU Instruction Set: Special Instructions

OpCode	Description	MIPS ISA Level
TGE	Trap if Greater Than or Equal	II
TGEU	Trap if Greater Than or Equal Unsigned	II
TLT	Trap if Less Than	II
TLTU	Trap if Less Than Unsigned	II
TEQ	Trap if Equal	II
TNE	Trap if Not Equal	II
TGEI	Trap if Greater Than or Equal Immediate	II
TGEIU	Trap if Greater Than or Equal Immediate Unsigned	II
TLTI	Trap if Less Than Immediate	II
TLTIU	Trap if Less Than Immediate Unsigned	II
TEQI	Trap if Equal Immediate	II
TNEI	Trap if Not Equal Immediate	II

Table 2.10 MIPS 2/MIPS 3 Exception Instructions

Notes

OpCode	Description	MIPS ISA Level
DMFC0	Doubleword Move From CP0	III
DMTC0	Doubleword Move To CP0	III
MTC0	Move to CP0	I
MFC0	Move from CP0	I
TLBR	Read Indexed TLB Entry	I
TLBWI	Write Indexed TLB Entry	I
TLBWR	Write Random TLB Entry	I
TLBP	Probe TLB for Matching Entry	I
CACHE	Cache Operation	RISCore4000 RISCore32300
ERET	Exception Return	RISCore4000 RISCore32300
WAIT	Enter Standby mode	RISCore4000 RISCore32300

Table 2.11 RC64474/RC64475 CP0 Instructions

Notes

RISCore4000 Pipeline

Notes

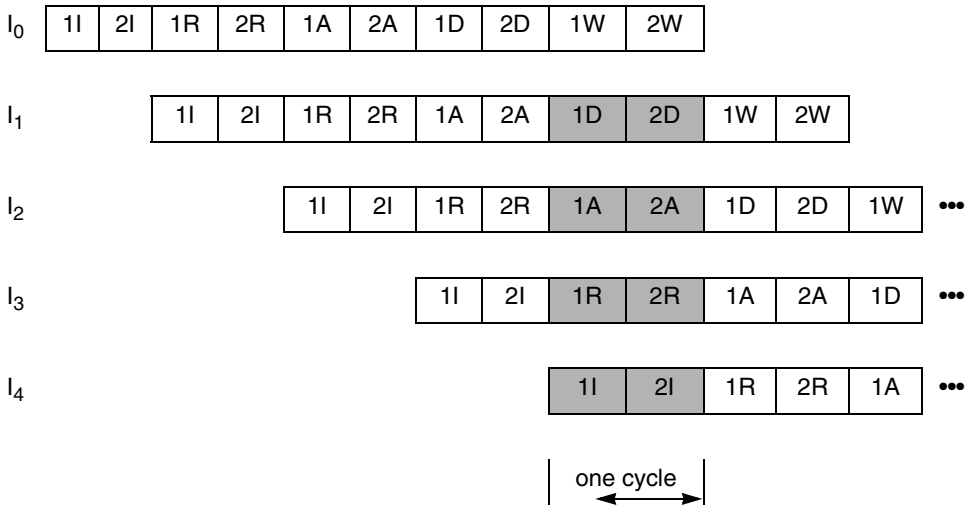
Introduction

This chapter describes some basic operations of the RISCore4000 pipeline and includes descriptions of the delay instructions[1], interruptions to the pipeline flow caused by interlocks and exceptions, and RC64474/RC64475 implementation of an uncached store buffer. The floating point unit (FPU) pipeline is described in a later chapter.

The RISCore4000 uses a 5-stage pipeline, similar to the RISCore3000 and RISCore32300 families. The simplicity of this pipeline allows the device to be lower cost and lower power than super-scalar or super-pipelined processors, allowing large data dependent applications to run close to the processor's peak performance levels.

Pipeline Operations

Once the pipeline has been filled, five instructions are executed simultaneously. Figure 3.1 shows the five stages of the instruction pipeline. Functional descriptions follow.



Key to Figure:			
1I-1R	Instruction cache access	2R	Instruction decode
1I-2I	Instruction virtual to physical address translation	1A-2A	Integer add, logical, shift
2A-2D	Data cache access and load align	1A	Data virtual address calculation
1D-2D	Data virtual to physical address translation	2A	Store align
2R	Register file read	1A	Branch decision
2R	Bypass calculation	2W	Register file write

Figure 3.1 RISCore4000 5-stage Instruction Pipeline

Each of the five pipeline stages has two phases, which are as illustrated and described below:

- *1I - Instruction Fetch, Phase one*
- *2I - Instruction Fetch, Phase two*
- *1R - Register Fetch, Phase one*

1. Delay instructions follow a branch or load instruction in the pipeline.

Notes

- *2R - Register Fetch, Phase two*
- *1A - Execution, Phase one*
- *2A - Execution, Phase two*
- *1D - Data Fetch, Phase one*
- *2D - Data Fetch, Phase two*
- *1W - Write Back, Phase one*
- *2W - Write Back, Phase two*

1I - Instruction Fetch, Phase one

During the 1I phase the instruction address translation begins in the ITLB.

2I - Instruction Fetch, Phase two

During the 2I phase, the instruction cache fetch begins and the instruction address translation in the ITLB continues.

1R - Register Fetch, Phase one

The 1R phase occurs as follows:

- *The instruction cache fetch finishes.*
- *The instruction cache tag is checked against the page frame number obtained from the ITLB.*

2R - Register Fetch, Phase two

The 2R phase occurs as follows:

- *The instruction decoder decodes the instruction.*
- *Any required operands are fetched from the register file.*
- *Make a decision to either issue or slip (for an interlock condition).*
- *For a branch, the branch address is calculated.*

1A - Execution, Phase one

The 1A phase occurs as follows:

- *Any result from the A or D stages are bypassed.*
- *The arithmetic logic unit (ALU) starts the integer arithmetic, logical or shift operation.*
- *The ALU calculates the data virtual address for load and store instructions.*
- *The ALU determines whether the branch condition is true.*

2A - Execution, Phase two

During the 2A phase, one of the following occurs:

- *The integer arithmetic, logical or shift operation will complete.*
- *A data cache access will start.*
- *Store data is shifted to the specified byte position(s).*
- *The data virtual-to-physical address translation in the DTLB will start.*

1D - Data Fetch, Phase one

During the 1D phase, one of the following occurs:

- *The data cache access will continue.*
- *The data address translation in the DTLB completes.*
- *The virtual-to-physical address translation in the JTLB will start.*

2D - Data Fetch, Phase two

During the 2D phase, one of the following occurs:

- *The data cache access will finish and the data is shifted down and extended.*

Notes

- *The virtual-to-physical address translation in the JTLB will finish.*
- *The data cache tag is checked against the PFN from the DTLB or JTLB for any data cache access.*

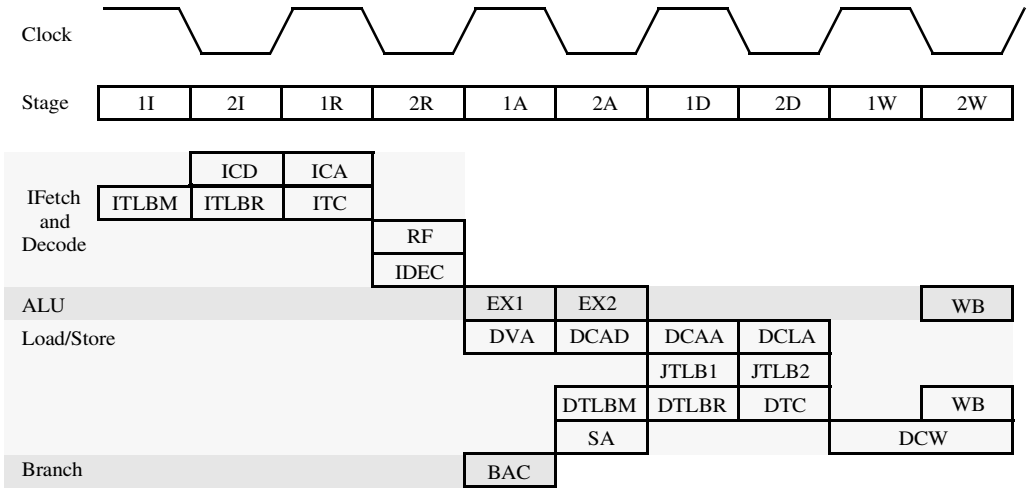
1W - Write Back, Phase one

This phase is used internally by the processor to resolve all exceptions, in preparation for the register file write.

2W - Write Back, Phase two

For register-to-register and load instructions, the result is written back to the register file during the 2W stage. Branch instructions perform no operation during this stage.

Figure 3.2 shows the activities occurring during each ALU pipeline stage, for load, store, and branch instructions.



ICD	Instruction cache address decode	ICA	Instruction cache array access
ITLBM	Instruction address translation match	ITLBR	Instrustion address translation read
ITC	Instruction tag check	RF	Register operand fetch
IDEC	Instruction decode	EX1	Operation stage 1
EX2	Operation stage 2	WB	Write back to register file
DVA	Data virtual address calculation	DCAD	Data cache address decode
DCAA	Data cache array access	DCLA	Data cache load align
JTLB1	Address translation in JTLB stage 1	JTLB2	Address translation in JTLB stage 2
DTLBM	Data address translation match	DTLMR	Data address translation read
DTC	Data tag check	SA	Store align
DCW	Data cache write	BAC	Branch address calculation

Figure 3.2 CPU Pipeline Activities

Branch Delay

The CPU pipeline has a branch delay of one cycle and a load delay of one cycle. The one-cycle branch delay is a result of the branch decision logic operating during the 1A pipeline phase of the branch instruction. This allows the branch target address calculated in the previous phase to be used for the instruction access in the following 1I phase. The pipeline will begin the fetch of the branch path as well as the fall-through path in the cycle following the delay slot. After the branch decision is made, the processor will continue with the fetch of either the branch path (for a taken branch) or the fall-through path (for the non-taken branch). Figure 3.3 illustrates the branch delay.

Notes

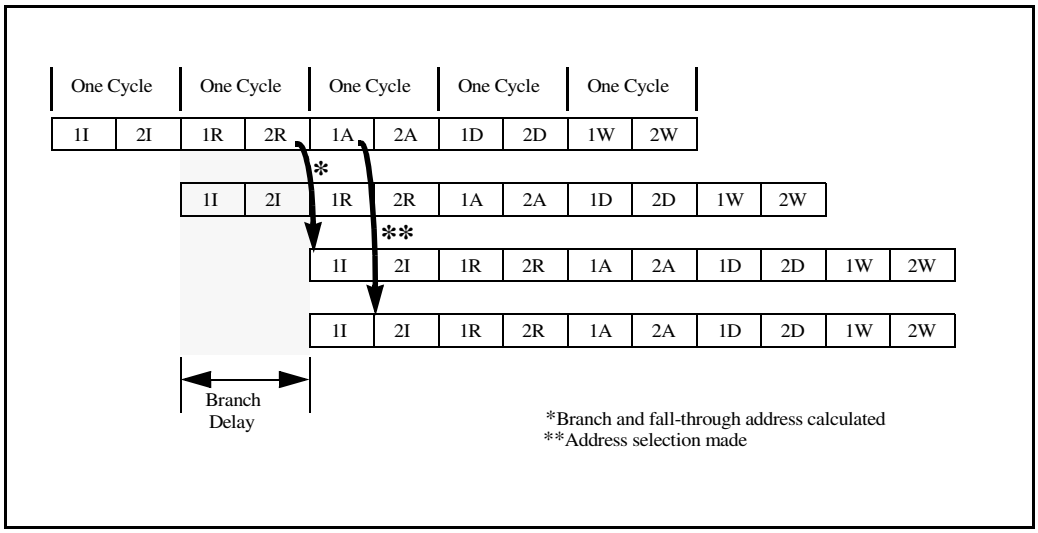


Figure 3.3 CPU Pipeline Branch Delay

Load Delay

The completion of a load at the end of the 2D pipeline phase produces an operand that is available for the 1A pipeline phase of the instruction following the load delay slot. Figure 3.4 shows the load delay of one pipeline cycle.

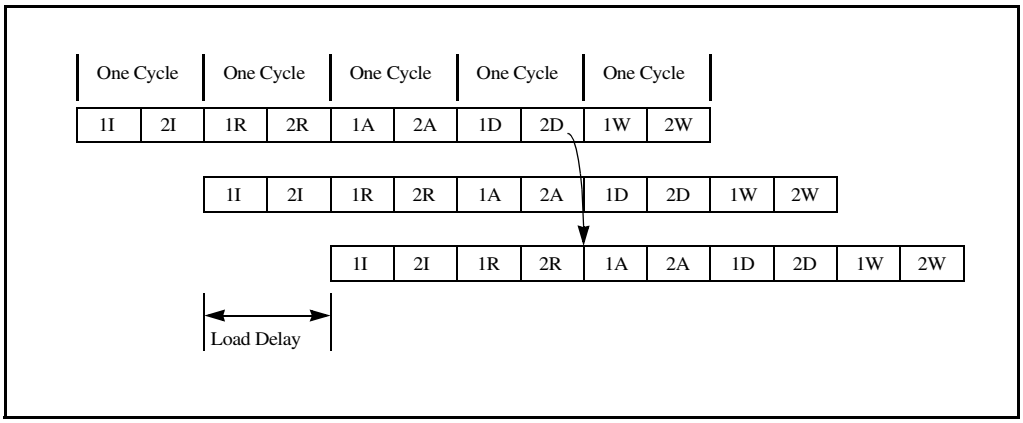


Figure 3.4 CPU Pipeline Load Delay

Interlock & Exception Handling

Smooth pipeline flow is interrupted when cache misses or exceptions occur, or when data dependencies are detected. Interruptions handled using hardware, such as cache misses, are referred to as interlocks, while those that are handled using software are called exceptions. The two types of interlocks are:

- *stalls, which are resolved by halting the pipeline*
- *slips, which require the back end of the pipeline to advance while the front end of the pipeline is held static*

At each cycle, exception and interlock conditions are checked for all active instructions. Because each exception or interlock condition corresponds to a particular pipeline stage, a condition can be traced back to the particular instruction in the exception/interlock stage, as shown in Table 3.1. For instance, a Reserved Instruction (RI) exception is raised in the execution (A) stage.

Notes

State	Pipeline Stage				
	I	R	A	D	W
Stall	ITM	ICM		DCM	
				CPE	
	I	**R**	**A**	**D**	**W**
Slip		LDI			
		MDSt			
		FCBsy			
	I	**R**	**A**	**D**	**W**
Exceptions	ITLB	IBE	RI	DBE	
		IPErr	CUn	NMI	
			BP	Reset	
			SC	DPErr	
			DTLB	OVF	
			TLBMod	Trap	
			Intr		

Table 3.1 Correspondence of Pipeline Stage to Interlock Condition

Table 3.2 and Table 3.3 describe the pipeline interlocks and exceptions listed in Table 3.1.

Exception	Description
ITLB	Instruction Translation or Address Exception
Intr	External Interrupt
IBE	Instruction Bus Error
RI	Reserved Instruction
BP	Breakpoint
SC	System Call
CUn	Coprocessor Unusable
IPErr	Instruction Parity Error
OVF	Integer Overflow
FPE	FP Interrupt
ExTrap	EX Stage Traps
DTLB	Data Translation or Address Exception
TLBMod	TLB Modified
DBE	Data Bus Error
DPErr	Data Parity Error
NMI	Non-maskable Interrupt (or Soft Reset)
Reset	Reset

Table 3.2 Pipeline Exceptions

Notes

Interlock	Description
ITM	Instruction TLB Miss
ICM	Instruction Cache Miss
CPE	Coprocessor Possible Exception
DCM	Data Cache Miss
LDI	Load Interlock
MDSt	Multiply/Divide Start
FCBsy	FP Coprocessor Busy

Table 3.3 Pipeline Interlocks

Exception Conditions

When an exception condition occurs, the relevant instruction and all those that follow it in the pipeline are cancelled. Accordingly, any stall conditions and any later exception conditions that may have referenced this instruction are inhibited; there is no benefit in servicing stalls for a cancelled instruction.

When an exceptional condition is detected for an instruction, the RC4000 will kill that instruction and all instructions that follow. When this instruction reaches the W stage, the exception flag causes it to write various CP0 registers with the exception state, change the current PC to the appropriate exception vector address and clear the exception bits of earlier pipeline stages.

This implementation allows all preceding instructions to complete execution and prevents all subsequent instructions from completing. Thus the value in the EPC is sufficient to restart execution. It also ensures that exceptions are taken in the order of execution; an instruction taking an exception may itself be killed by an instruction further down the pipeline that takes an exception in a later cycle.

Figure 3.5 shows the exception detection procedure (for example, a reserved instruction exception).

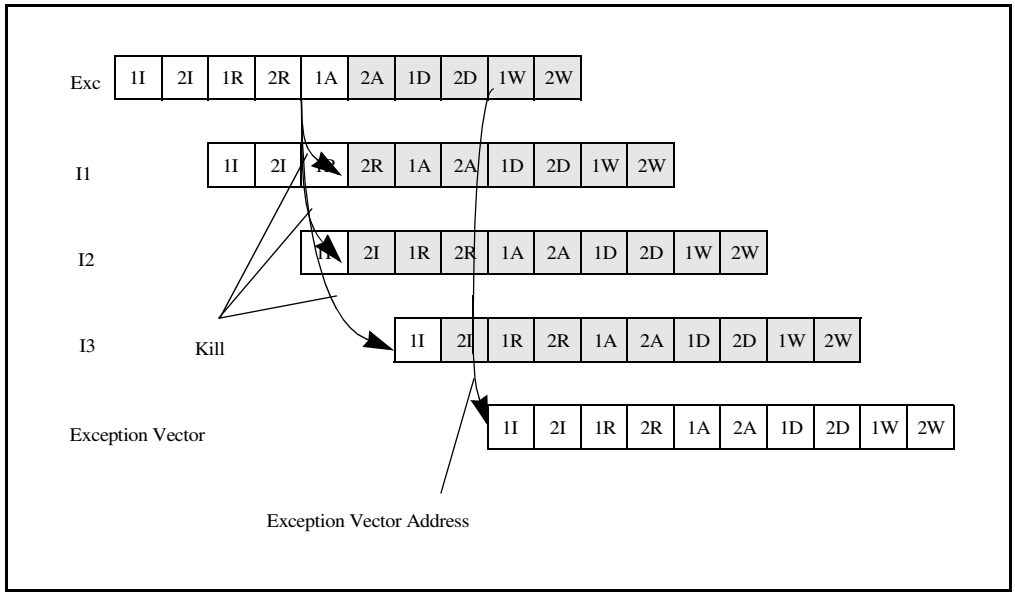


Figure 3.5 Exception Detection

Stall Conditions

Stalls are used to stop the pipeline for conditions detected after the R pipe-stage. When a stall occurs, the processor will resolve the condition and then the pipeline will continue. Figure 3.6 shows a data cache miss stall.

Notes

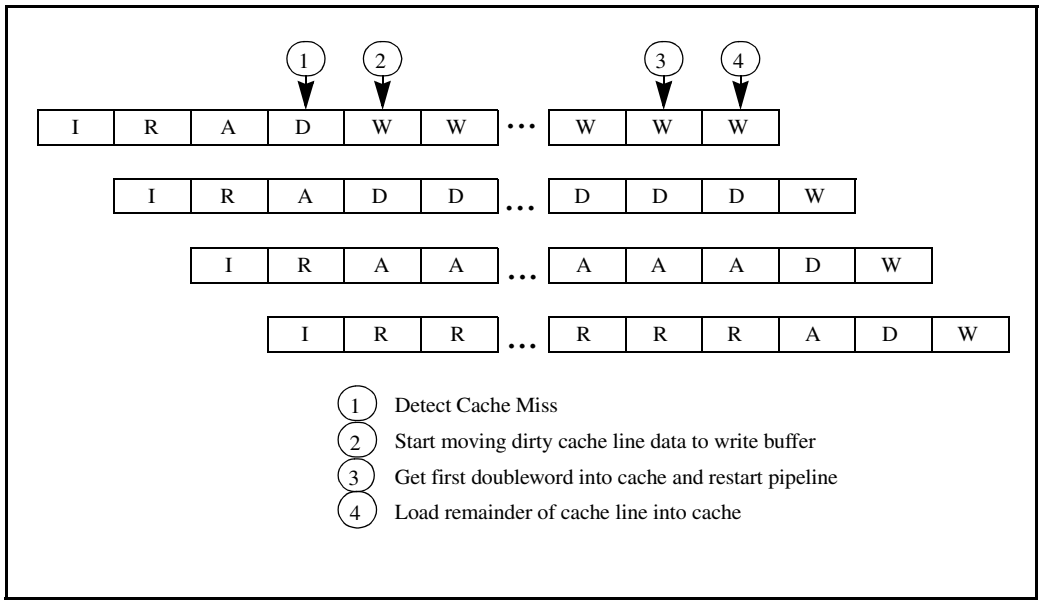


Figure 3.6 Data Cache Miss

The data cache miss is detected in the D pipe stage. If the cache line to be replaced is dirty — the W bit is set — the data is moved to the internal write buffer in the next cycle. The first doubleword of data is returned to the cache in 3 and the pipeline will then restart. The remainder of the cache line is returned in the subsequent cycles. The data to be written back will be returned to memory some time after the entire new cache line is returned.

Slip Conditions

During the 2R and 1A pipe-stages, internal logic will determine whether it is possible to start the current instruction in this cycle. If all of the source operands are available (either from the register file or via the internal bypass logic) and all the hardware resources necessary to complete the instruction will be available at the necessary time(s), then the instruction “issues”; otherwise, the instruction will “slip”. Slipped instructions are retried on subsequent cycles until they issue. The backend of the pipeline (stages D and W) will advance normally during slips in an attempt to resolve the conflict. “NOPS” will be inserted into the bubble in the pipeline. Instructions killed by branch likely instructions, ERET or exceptions will not cause slips. Figure 3.7 shows an instruction cache miss.

Notes

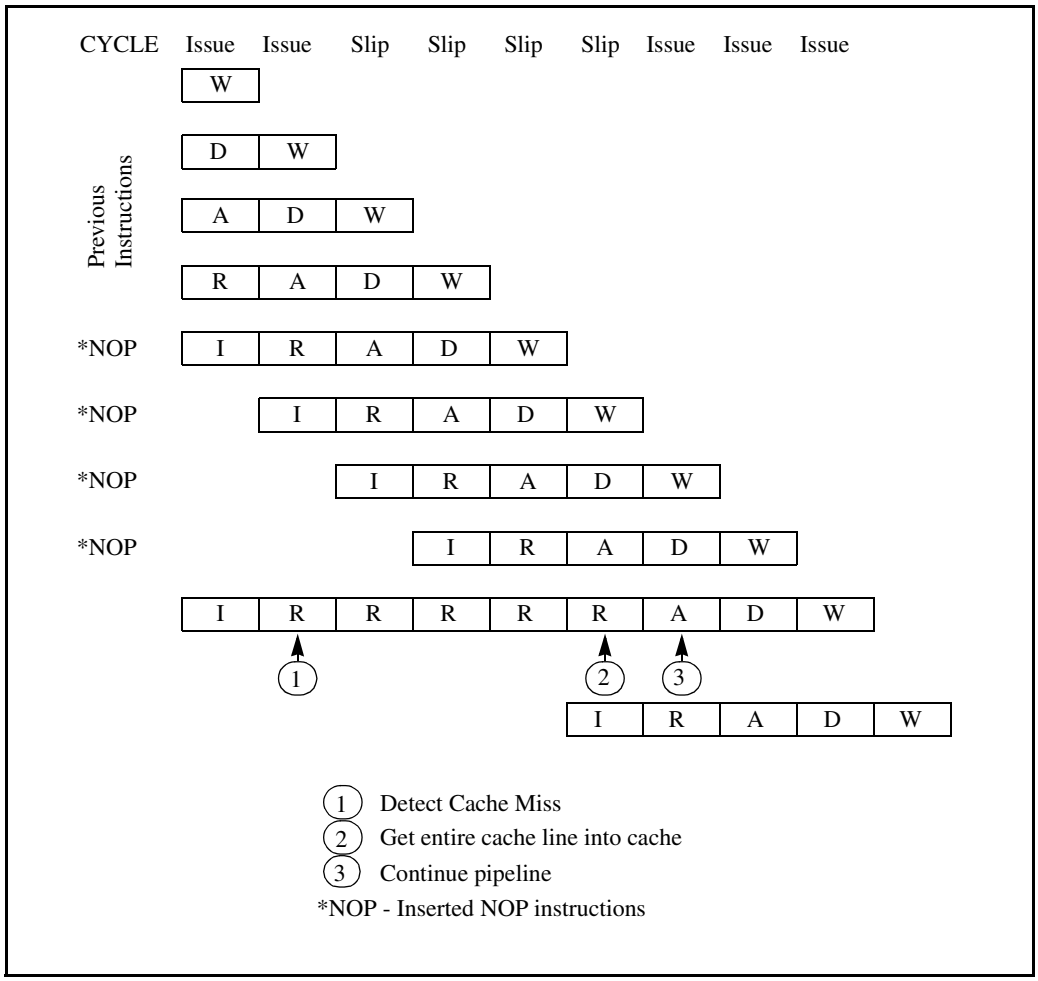


Figure 3.7 Instruction cache miss

Instruction cache misses are detected in R as shown in Figure 3.7 and the pipeline slips in its A stage. There can never be a writeback required for an instruction cache miss since dirty data can never exist in the I cache. Writes are not allowed to the I cache. Note that early restart is not employed for instruction cache misses, the requested cache line will be loaded into the cache in its entirety and, after that, the pipeline will restart.

RISCore4000 Write Buffer

The RISCore4000 contains a write buffer to improve the performance of writes to the external memory. Writes to external memory, whether cache miss writebacks or stores to uncached or write-through addresses, use this on-chip write buffer. The write buffer holds up to four 64-bit address and data pairs.

For a cache miss write-back, the entire buffer is used for the write-back data and allows the processor to proceed in parallel with the memory update. For uncached and write-through stores, the write buffer uncouples the CPU from the write to memory allowing increased performance. If the write buffer is full, additional stores will stall until there is room for them in the write buffer.

Memory Management Unit

Notes

Introduction

The RISCore4000 provides a full-featured memory management unit (MMU) that uses an on-chip Translation Lookaside Buffer (TLB), to translate virtual addresses into physical addresses. This chapter describes the processor's virtual and physical address spaces, the virtual-to-physical address translation, the operation of the TLB in making these translations, and those System Control Coprocessor (CP0) registers that provide the software interface to the TLB.

Mapped virtual addresses are translated into physical addresses using the on-chip TLB.[1] The TLB is a fully associative memory organized as 48 pairs of even/odd entries and maps 96 virtual pages to their corresponding physical addresses. When address mapping is indicated, each TLB entry is checked simultaneously for a match with the virtual address that is extended with an ASID stored in the *EntryHi* register. The address mapped to a page ranges in size from 4Kbytes to 16Mbytes, in multiples of 4—that is, 4K, 16K, 64K, 256K, 1M, 4M, 16M.

If there is a virtual address match or hit in the TLB, the physical page number is extracted from the TLB and concatenated with the offset to form the physical address (see Figure 4.1). If no match occurs (TLB miss), an exception is taken and software refills the TLB from the page table resident in memory. Software can write over a selected TLB entry or use a hardware mechanism to write into a random entry.

Note: No detection or shutdown mechanism is provided for the occurrence of multiple matches in the TLB. There is no damage possible from this condition, and the result is undefined. It is expected of the software not to allow multiple matches.

Address Spaces

The processor's virtual address space can either be 32- or 64-bits wide, depending on implementation of the user, supervisor or kernel mode of operation and the setting of the corresponding extended address bit in the Status register (UX, SX and KX).

- *For the extended address bit = 0, addresses are 32-bits wide.*
- *For the extended address bit = 1, addresses are 64-bits wide.*

Both 32-bit and 64-bit address wrap in the same way. For example, in 64-bit mode `0xffffffffffffffff` will wrap to `0x0000000000000000`.

Note: The RISCore4000 does not slip on shifts of >32-bits or other shift variables.

Figure 4.1 shows the translation of a virtual address into a physical address.

[1] There are virtual-to-physical address translations that occur outside of the TLB. For example, addresses in *kseg0* and *kseg1* spaces are unmapped translations. In these spaces, the physical address is `0x0000 0000 0 || VA[28:0]`

Notes

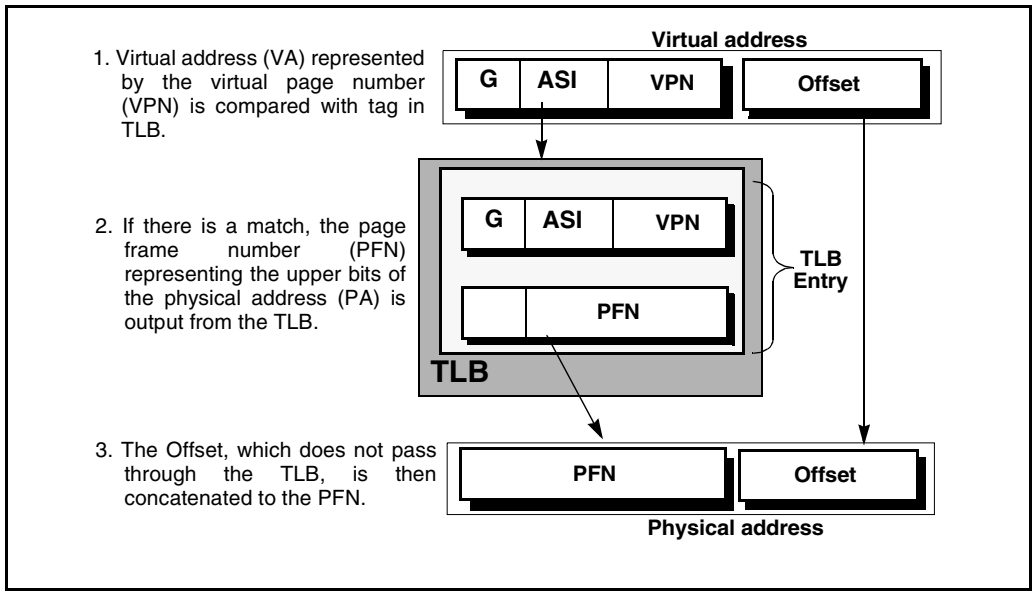


Figure 4.1 Overview of a Virtual-to-Physical Address Translation

As shown in Figure 4.2 and Figure 4.3, the virtual address is extended with an 8-bit address space identifier (ASID), which reduces the frequency of TLB flushing when switching contexts. This 8-bit ASID is in the CP0 *EntryHi* register, described later in this chapter. The *Global* bit (*G*) is in the *EntryLo0* and *EntryLo1* registers, described later in this chapter.

Physical Address Space

Using a 36-bit address, the processor physical address space encompasses 64Gigabytes. The section following describes the translation of a virtual address to a physical address.

Virtual-to-Physical Address Translation

Converting a virtual address to a physical address begins by comparing the virtual address from the processor with the virtual address in the TLB; there is a match when the virtual page number (VPN) of the address is the same as the VPN field of the entry, and either:

- *the Global (G) bit of the TLB entry is set, or*
- *the ASID field of the virtual address is the same as the ASID field of the TLB entry.*

This match is referred to as a *TLB hit*. If there is no match, a TLB Miss exception is taken by the processor and software is allowed to refill the TLB from a page table of virtual/physical addresses in memory. If there is a virtual address match in the TLB, the physical address is output from the TLB and concatenated with the *Offset*, which represents an address within the page frame space. The *Offset* does not pass through the TLB. Virtual-to-physical translation is described in greater detail throughout the remainder of this chapter; Figure 4.1 on page 4-2 is a flow diagram of the process. The following two sections provide descriptions on the 32-bit and 64-bit address translations.

32-bit Virtual Address Translation

Figure 4.2 shows the virtual-to-physical-address translation of a 32-bit virtual address.

- *The top portion of Figure 4.2 shows a virtual address with a 12-bit, or 4Kbyte, page size, labelled Offset. The remaining 20 bits of the address represent the VPN, and index the 1M-entry page table.*
- *The bottom portion of Figure 4.2 shows a virtual address with a 24-bit, or 16Mbyte, page size, labelled Offset. The remaining 8 bits of the address represent the VPN, and index the 256-entry page table.*

Notes

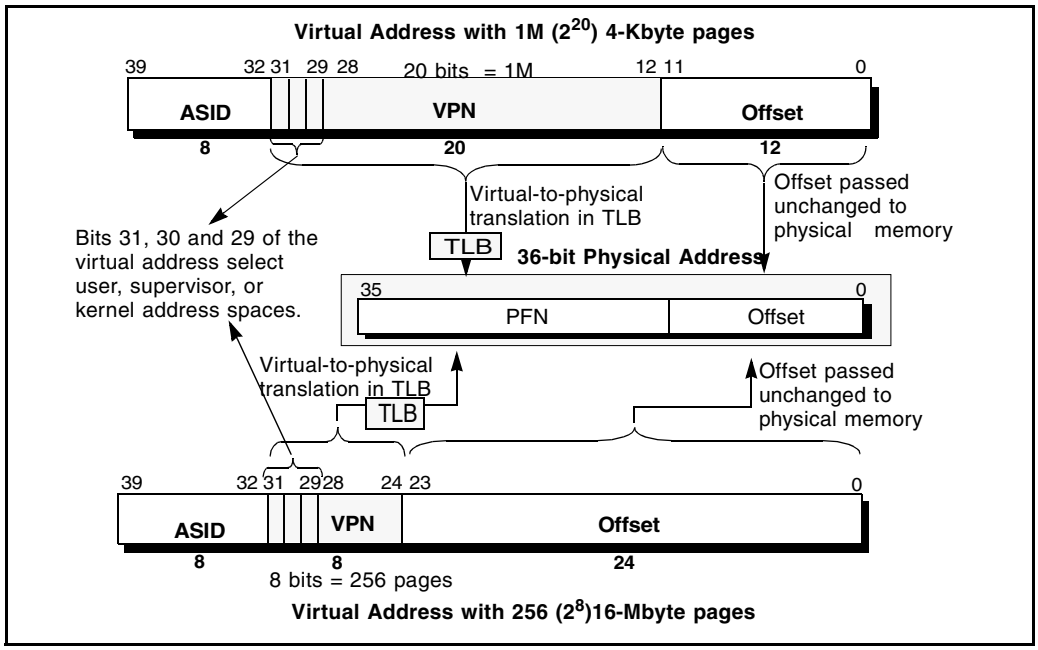


Figure 4.2 32-bit Virtual Address Translation

64-bit Virtual Address Translation

Figure 4.3 on page 4-3 shows the virtual-to-physical-address translation of a 64-bit virtual address. This figure illustrates the two extremes in the range of possible page sizes: a 4Kbyte page (12 bits) and a 16Mbyte page (24 bits).

- *The top portion of Figure 4.3 shows a virtual address with a 12-bit, or 4Kbyte, page size, labelled Offset. The remaining 28 bits of the address represent the VPN, and index the 256M-entry page table.*
- *The bottom portion of Figure 4.3 shows a virtual address with a 24-bit, or 16Mbyte, page size, labelled Offset. The remaining 16 bits of the address represent the VPN, and index the 64K-entry page table.*

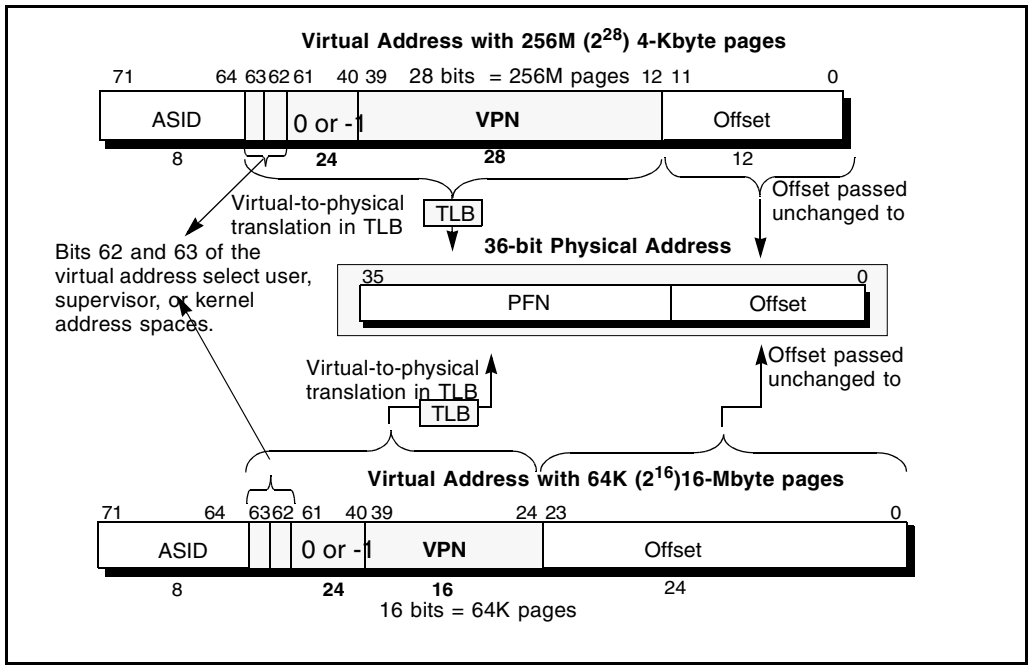


Figure 4.3 64-bit Virtual Address Translation

Notes

Operating Modes

The processor has three operating modes that function in both 32- and 64-bit operations:

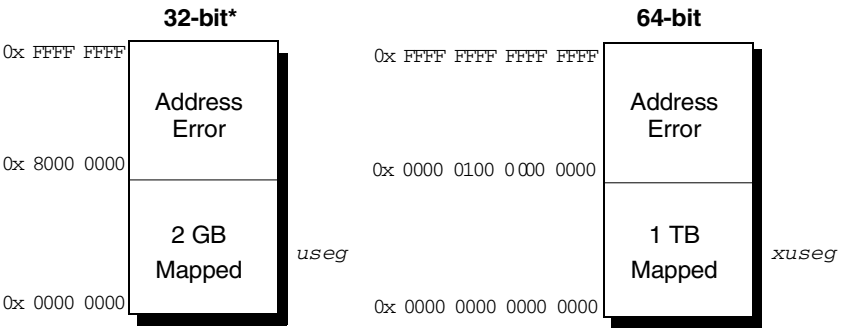
- *User mode*
- *Supervisor mode*
- *Kernel mode*

User Mode

In User mode, a single, uniform virtual address space—labelled User segment—is available; its size is:

- *2 Gbytes (2^{31} bytes) for Status.UX = 0 (useg)*
- *1 Tbyte (2^{40} bytes) for Status.UX = 1 (xuseg)*

Figure 4.4 shows the User mode virtual address space.



Note: For 32-bit virtual addresses, bit 31 is sign-extended through bits 63:32. Failure (for example, bit 31 = 1) results in an Address Error exception.

Figure 4.4 User Mode Virtual Address Space

The User segment starts at address 0 and the current active user process resides in either *useg* (32-bit virtual addressing) or *xuseg* (in 64-bit virtual addressing). The TLB identically maps all references to *useg*/*xuseg* from all modes, and controls cache accessibility.

The processor operates in **User mode** when the *Status* register contains the following bit-values:

- *KSU bits = 10_2*
- *EXL = 0*
- *ERL = 0*

In conjunction with these bits, the *UX* bit in the *Status* register selects between 32- or 64-bit User virtual addressing as follows:

- *when UX = 0, 32-bit useg space is selected*
- *when UX = 1, 64-bit xuseg space is selected*

Table 4.1 lists the characteristics of the two user mode segments, *useg* and *xuseg*.

Address Bit Values	Status Register Bit Values				Segment Name	Address Range	Segment Size
	KSU	EXL	ERL	UX			
32-bit A(31) = 0	10_2	0	0	0	useg	0x0000 0000 through 0x7FFF FFFF	2 Gbyte (2^{31} bytes)
64-bit A(63:40) = 0	10_2	0	0	1	xuseg	0x0000 0000 0000 0000 through 0x0000 00FF FFFF FFFF	1 Tbyte (2^{40} bytes)

Table 4.1 32-bit and 64-bit User Mode Segments

Notes

32-bit User Mode (*useg*)

In User mode, when Status.UX = 0, User mode virtual addressing is compatible with the 32-bit addressing model shown in Figure 4.4, and a 2-Gbyte user address space is available, labelled *useg*.

All valid User mode virtual addresses have their most-significant bit cleared to 0. Any attempt to reference an address with the most-significant bit set while in User mode causes an Address Error exception. In 32-bit User mode virtual addressing, the TLB refill exception vector is used for TLB misses. The system maps all references to *useg* through the TLB, and bit settings within the TLB entry for the page determine the cacheability of a reference.

64-bit User Mode (*xuseg*)

In User mode, when Status.UX =1, User mode virtual addressing is extended to the 64-bit model shown in Figure 4.4, and a 1-Tbyte user address space is available, labelled *xuseg*. All valid User mode virtual addresses have bits 63:40 equal to 0; an attempt to reference an address with bits 63:40 not equal to 0 causes an Address Error exception. The extended addressing TLB refill exception vector is used for TLB misses.

Supervisor Mode Operations

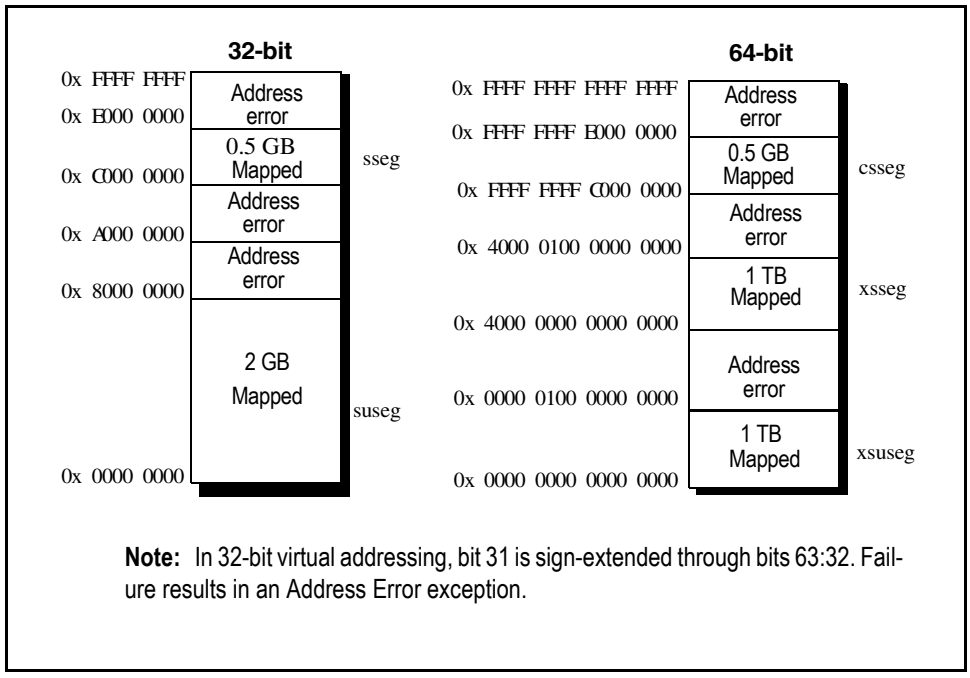
Supervisor mode is designed for layered operating systems in which a true kernel runs in RISCore4000's Kernel mode while the rest of the operating system runs in Supervisor mode. The processor operates in Supervisor mode when the *Status* register contains the following bit values:

- $KSU = 01_2$
- *EXL = 0*
- *ERL = 0*

In conjunction with these bits, the *SX* bit in the *Status* register selects between 32- or 64-bit Supervisor mode virtual addressing:

- *when SX = 0, 32-bit supervisor space virtual addressing is selected*
- *when SX = 1, 64-bit supervisor space virtual addressing is selected*

Figure 4.5 shows Supervisor mode address mapping, and Table 4.2, which follows the figure, lists the characteristics of the supervisor mode segments. Descriptions of the address spaces follow.



Note: In 32-bit virtual addressing, bit 31 is sign-extended through bits 63:32. Failure results in an Address Error exception.

Figure 4.5 Supervisor Mode Virtual Address Space

Notes

Address Bit Values	Status Register Bit Values				Segment Name	Address Range	Segment Size
	KSU	EXL	ERL	SX			
32-bit A(31) = 0	01_2	0	0	0	suseg	0x0000 0000 through 0x7FFF FFFF	2 Gbytes (2^{31} bytes)
32-bit A(31:29) = 110_2	01_2	0	0	0	sseg	0xC000 0000 through 0xDFFF FFFF	512 Mbytes (2^{29} bytes)
64-bit A(63:62) = 00_2	01_2	0	0	1	xsuseg	0x0000 0000 0000 0000 through 0x0000 00FF FFFF FFFF	1 Tbyte (2^{40} bytes)
64-bit A(63:62) = 01_2	01_2	0	0	1	xsseg	0x4000 0000 0000 0000 through 0x4000 00FF FFFF FFFF	1 Tbyte (2^{40} bytes)
64-bit A(63:62) = 11_2	01_2	0	0	1	csseg	0xFFFF FFFF C000 0000 through 0xFFFF FFFF DFFF FFFF	512 Mbytes (2^{29} bytes)

Table 4.2 32-bit and 64-bit Supervisor Mode Segments

In **Supervisor mode**, when **Status.SX = 0** and the **most-significant bit** of the 32-bit virtual address **is set to 0**, the ***suseg*** virtual address space is selected; it covers the full 2^{31} bytes (2Gbytes) of the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address. This mapped space starts at virtual address 0x0000 0000 and runs through 0x7FFF FFFF.

When **Status.SX = 0** and the **three most-significant bits** of the 32-bit virtual address **are 110_2**, the ***sseg*** virtual address space is selected; it covers 2^{29}-bytes (512Mbytes) of the current supervisor address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address. This mapped space begins at virtual address 0xC000 0000 and runs through 0xDFFF FFFF.

64-bit Supervisor Mode, User Space (*xsuseg*)

In Supervisor mode, when Status.SX = 1 and bits 63:62 of the virtual address are set to 00_2, the *xsuseg* virtual address space is selected; it covers the full 2^{40} bytes (1Tbyte) of the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address. This mapped space starts at virtual address 0x0000 0000 0000 0000 and runs through 0x0000 00FF FFFF FFFF.

64-bit Supervisor Mode, Current Supervisor Space (*xsseg*)

In Supervisor mode, when Status.SX = 1 and bits 63:62 of the virtual address are set to 01_2, the *xsseg* current supervisor virtual address space is selected. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address. This mapped space begins at virtual address 0x4000 0000 0000 0000 and runs through 0x4000 00FF FFFF FFFF.

64-bit Supervisor Mode, Separate Supervisor Space (*csseg*)

In Supervisor mode, when Status.SX = 1 and bits 63:62 of the virtual address are set to 11_2, the *csseg* separate supervisor virtual address space is selected. Addressing of the *csseg* is compatible with addressing *sseg* in 32-bit mode. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address. This mapped space begins at virtual address 0xFFFF FFFF C000 0000 and runs through 0xFFFF FFFF DFFF FFFF.

Notes

Kernel Mode Operations

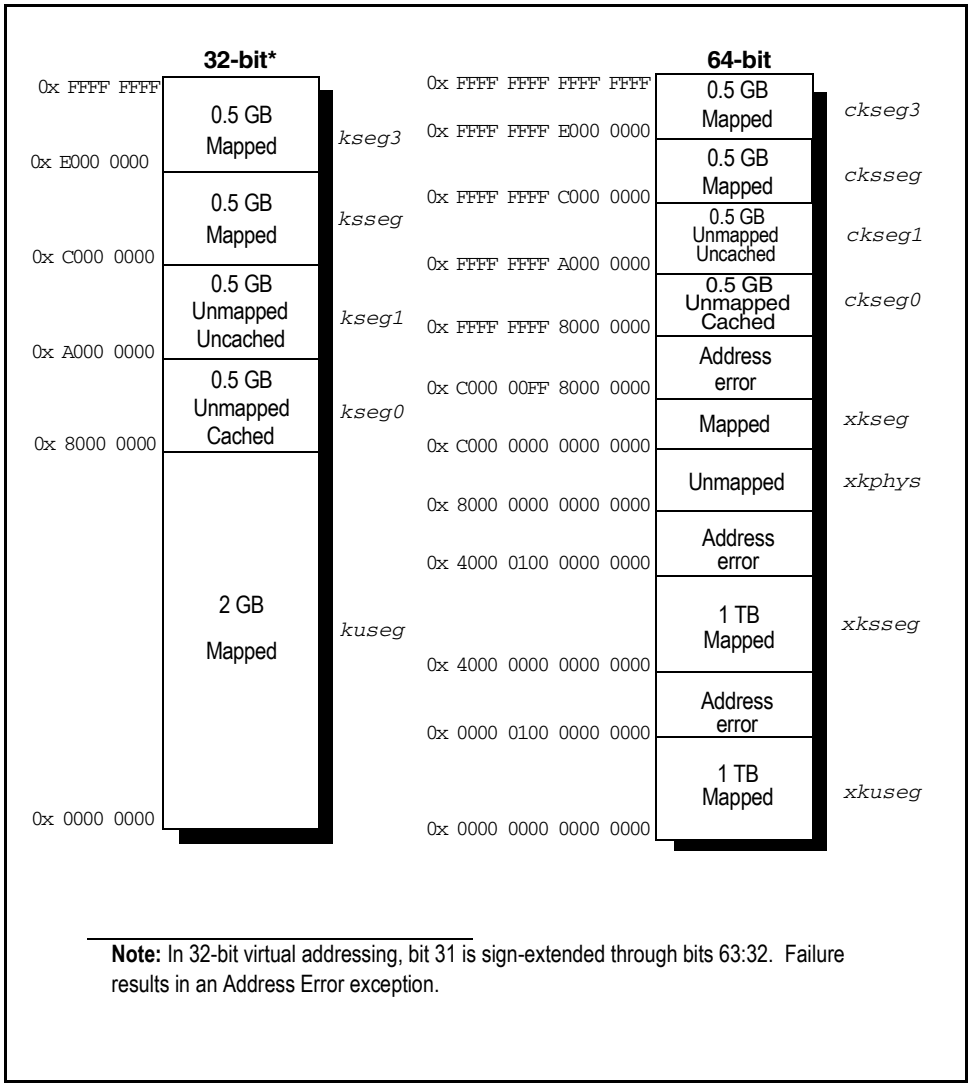
The processor operates in Kernel mode when the *Status* register contains one of the following values:

- *KSU = 00_2*
- *EXL = 1*
- *ERL = 1*

In conjunction with these bits, the *KX* bit in the *Status* register selects between 32- or 64-bit Kernel mode addressing:

- *when KX = 0, 32-bit kernel space virtual addressing is selected*
- *when KX = 1, 64-bit kernel space virtual addressing is selected*

The processor enters Kernel mode whenever an exception is detected and it remains in Kernel mode until an Exception Return (ERET) instruction is executed. The ERET instruction restores the processor to the mode existing prior to the exception. Kernel mode's virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 4.6.



Note: In 32-bit virtual addressing, bit 31 is sign-extended through bits 63:32. Failure results in an Address Error exception.

Figure 4.6 Kernel Mode Address Space

Table 4.3 lists the characteristics of the 32-bit kernel mode segments, and Table 4.4 lists the characteristics of the 64-bit kernel mode segments

Notes

Address Bit Values	Status Register Is One Of These Values				Segment Name	Address Range	Segment Size
	KSU	EXL	ERL	KX			
A(31) = 0	KSU = 00_2 or EXL = 1 or ERL =1			0	kuseg	0x0000 0000 through 0x7FFF FFFF	2 Gbytes (2^{31} bytes)
A(31:29) = 100_2				0	kseg0	0x8000 0000 through 0x9FFF FFFF	512 Mbytes (2^{29} bytes)
A(31:29) = 101_2				0	kseg1	0xA000 0000 through 0xBFFF FFFF	512 Mbytes (2^{29} bytes)
A(31:29) = 110_2				0	ksseg	0xC000 0000 through 0xDFFF FFFF	512 Mbytes (2^{29} bytes)
A(31:29) = 111_2				0	kseg3	0xE000 0000 through 0xFFFF FFFF	512 Mbytes (2^{29} bytes)

Table 4.3 32-bit Kernel Mode Segments

32-bit Kernel Mode, User Space (*kuseg*)

In Kernel mode, when Status.KX = 0, and the most-significant bit of the virtual address, A31, is cleared, the 32-bit *kuseg* virtual address space is selected; it covers the full 2^{31} bytes (2 Gbytes) of the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

32-bit Kernel Mode, Kernel Space 0 (*kseg0*)

In Kernel mode, when Status.KX = 0 and the most-significant three bits of the virtual address are 100_2, 32-bit *kseg0* virtual address space is selected; it is the current 2^{29}-byte (512-Mbyte) kernel physical space. References to *kseg0* are not mapped through the TLB; the physical address selected is defined by subtracting 0x8000 0000 from the virtual address (physical address = 0x0000 0000 0 || VA[28:0]). The *K0* field of the *Config* register, described in this chapter, controls cacheability and coherency.

32-bit Kernel Mode, Kernel Space 1 (*kseg1*)

In Kernel mode, when Status.KX = 0 and the most-significant three bits of the 32-bit virtual address are 101_2, 32-bit *kseg1* virtual address space is selected; it is the current 2^{29}-byte (512Mbyte) kernel physical space. References to *kseg1* are not mapped through the TLB; the physical address selected is defined by subtracting 0xA000 0000 from the virtual address (physical address = 0x0000 0000 0 || VA[28:0]). Caches are disabled for accesses to these addresses, and physical memory (or memory-mapped I/O device registers) are accessed directly.

32-bit Kernel Mode, Supervisor Space (*ksseg*)

In Kernel mode, when Status.KX = 0 and the most-significant three bits of the 32-bit virtual address are 110_2, the *ksseg* virtual address space is selected; it is the current 2^{29}-byte (512Mbyte) supervisor virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

32-bit Kernel Mode, Kernel Space 3 (*kseg3*)

In Kernel mode, when Status.KX = 0 and the most-significant three bits of the 32-bit virtual address are 111_2, the *kseg3* virtual address space is selected; it is the current 2^{29}-byte (512Mbyte) kernel virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

Notes

Address Bit Values	Status Register Is One Of These Values				Segment Name	Address Range	Segment Size
	KSU	EXL	ERL	KX			
$A(63:62) = 00_2$	$KSU = 00_2$ or EXL = 1 or ERL = 1			1	xkuseg	0x0000 0000 0000 0000 through 0x0000 00FF FFFF FFFF	1 Tbyte (2^{40} bytes)
$A(63:62) = 01_2$				1	xksseg	0x4000 0000 0000 0000 through 0x4000 00FF FFFF FFFF	1 Tbyte (2^{40} bytes)
$A(63:62) = 10_2$				1	xkphys	0x8000 0000 0000 0000 through 0xBFFF FFFF FFFF FFFF	8 2^{36}-byte spaces
$A(63:62) = 11_2$				1	xkseg	0xC000 0000 0000 0000 through 0xC000 00FF 7FFF FFFF	2^{44} bytes
$A(63:62) = 11_2$ $A(61:31) = -1$				1	ckseg0	0xFFFF FFFF 8000 0000 through 0xFFFF FFFF 9FFF FFFF	512 Mbytes (2^{29} bytes)
$A(63:62) = 11_2$ $A(61:31) = -1$				1	ckseg1	0xFFFF FFFF A000 0000 through 0xFFFF FFFF BFFF FFFF	512 Mbytes (2^{29} bytes)
$A(63:62) = 11_2$ $A(61:31) = -1$				1	cksseg	0xFFFF FFFF C000 0000 through 0xFFFF FFFF DFFF FFFF	512 Mbytes (2^{29} bytes)
$A(63:62) = 11_2$ $A(61:31) = -1$				1	ckseg3	0xFFFF FFFF E000 0000 through 0xFFFF FFFF FFFF FFFF	512 Mbytes (2^{29} bytes)

Table 4.4 64-bit Kernel Mode Segments

64-bit Kernel Mode, User Space (*xkuseg*)

In Kernel mode, when Status.KX = 1 and bits 63:62 of the 64-bit virtual address are 00_2, the *xkuseg* virtual address space is selected; it covers the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address. As a special feature for the ECC handler, if the *ERL* bit of the *Status* register is set, the user address region becomes a 2^{31}-byte unmapped, uncached space. This allows the ECC exception code to operate uncached using *r0* as a base register.

64-bit Kernel Mode, Current Supervisor Space (*xksseg*)

In Kernel mode, when Status.KX = 1 and bits 63:62 of the 64-bit virtual address are 01_2, the *xksseg* virtual address space is selected; it is the current supervisor virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

64-bit Kernel Mode, Physical Spaces (*xkphys*)

In Kernel mode, when Status.KX = 1 and bits 63:62 of the 64-bit virtual address are 10_2, the *xkphys* virtual address space is selected; it is a set of eight 2^{36}-byte kernel physical spaces. Accesses with address bits 58:36 not equal to 0 cause an address error. References to this space are not mapped; the physical address selected is taken from bits 35:0 of the virtual address. Bits 61:59 of the virtual address specify the cacheability and coherency attributes, as shown in Table 4.5.

Notes

Value (61:59)	Cacheability and Coherency Attributes	Starting Address
0	Cacheable, noncoherent, write-through, no write allocate	0x8000 0000 0000 0000
1	Cacheable, noncoherent, write-through, write allocate	0x8800 0000 0000 0000
2	Uncached	0x9000 0000 0000 0000
3	Cacheable, noncoherent	0x9800 0000 0000 0000
4 - 7	Reserved	0xA000 0000 0000 0000

Table 4.5 Cacheability and Coherency Attributes

64-bit Kernel Mode, Kernel Space (*xkseg*)

In Kernel mode, when Status.KX = 1 and bits 63:62 of the 64-bit virtual address are 11_2, the address space selected is one of the following:

- *kernel virtual space, xkseg, the current supervisor virtual space; the virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address*
- *one of the four 32-bit kernel compatibility spaces, as described in the next section.*

64-bit Kernel Mode, Compatibility Spaces (*ckseg1:0, cksseg, ckseg3*)

In Kernel mode, when Status.KX = 1, bits 63:62 of the 64-bit virtual address are 11_2, and bits 61:31 of the virtual address equal "–1", the lower two bytes of address, as shown in Figure 4.6, select one of the following 512-Mbyte compatibility spaces.

- *ckseg0. This 64-bit virtual address space is an unmapped region, compatible with the 32-bit address model kseg0. The K0 field of the Config register, described in this chapter, controls cacheability and coherency.*
- *ckseg1. This 64-bit virtual address space is an unmapped and uncached region, compatible with the 32-bit address model kseg1.*
- *cksseg. This 64-bit virtual address space is the current supervisor virtual space, compatible with the 32-bit address model ksseg.*
- *ckseg3. This 64-bit virtual address space is kernel virtual space, compatible with the 32-bit address model kseg3.*

System Control Coprocessor (CP0)

Notes

Introduction

The System Control Coprocessor (CP0) is implemented as an integral part of the CPU, and supports memory management, address translation, exception handling, and other privileged operations. CP0 contains the registers shown in Figure 5.1 plus a 48-entry TLB. The sections that follow describe how the processor uses each of the memory management-related registers. Each CP0 register has a unique identification number. This unique number is referred to as the *register number*. For instance, the *Page Mask* register is register number 5.

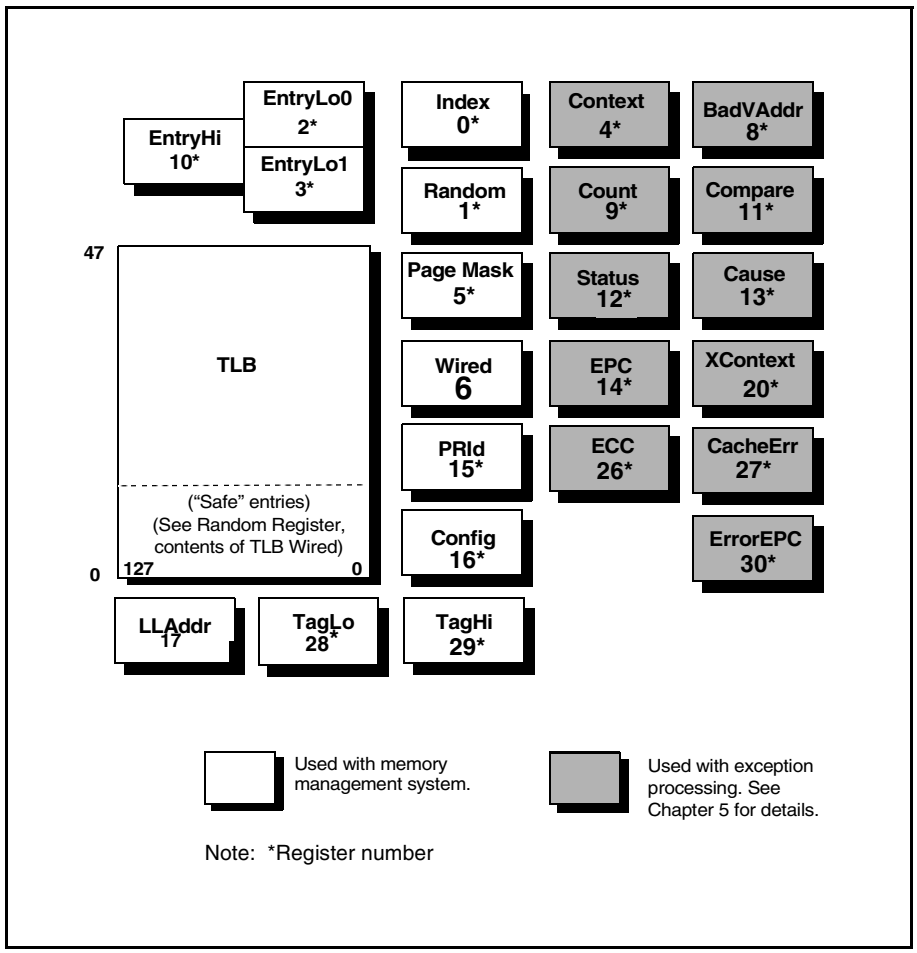


Figure 5.1 CP0 Registers and the TLB

Format of a TLB Entry

Figure 5.2 shows the TLB entry formats for both 32- and 64-bit virtual addressing. Each field of an entry has a corresponding field in the *EntryHi*, *EntryLo0*, *EntryLo1*, or *PageMask* registers, as shown in Figure 5.3 and Figure 5.4; for example the *Mask* field of the TLB entry is also held in the *PageMask* register.

Notes

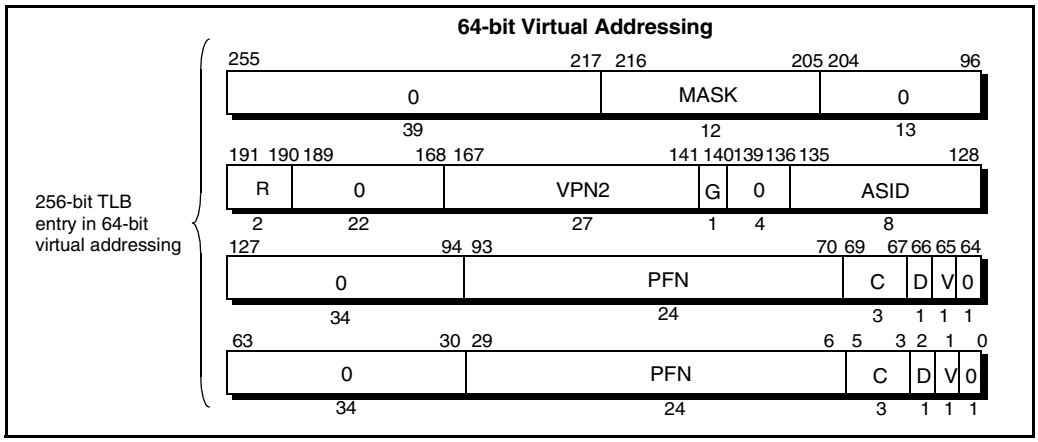
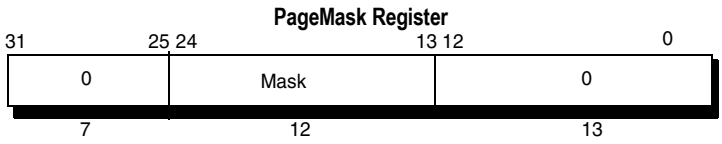
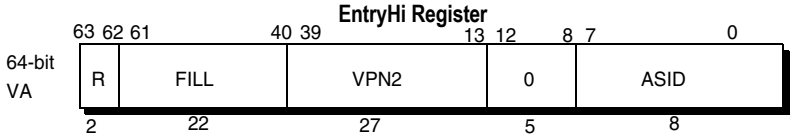


Figure 5.2 Format of a TLB Entry

The format of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers is nearly the same as the TLB entry. The one exception is the *Global* field (*G* bit), which is used in the TLB but is reserved in the *EntryHi* register. Figure 5.3 describes the TLB entry fields that are shown in Figure 5.2.

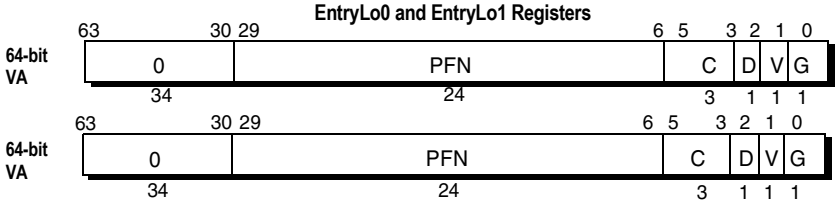


Mask ..... Page comparison mask.
0 ..... Reserved. Must be written as zeroes, and returns zeroes when read.



VPN2 ..... Virtual page number divided by two (maps to two pages).
ASID ..... Address space ID field. An 8-bit field that lets multiple processes share the TLB; each process has a distinct mapping of otherwise identical virtual page numbers.
R ..... Region. (00 Æ user, 01 Æ supervisor, 11 Æ kernel) used to match $vAddr_{63...62}$
Fill ..... Reserved. Returns zero when read, ignored on writes.
0 ..... Reserved. Must be written as zeroes, and returns zeroes when read.

Figure 5.3 Fields of the PageMask and EntryHi Registers



PFN ..... Page frame number; the upper bits of the physical address.
C ..... Specifies the TLB page coherency attribute; see Table 5.1.
D ..... Dirty. If this bit is set, the page is marked as dirty and, therefore, writable. This bit is actually a write-protect bit that software can use to prevent alteration of data.
V ..... Valid. If this bit is set, it indicates that the TLB entry is valid; otherwise, a TLBL or TLBS miss occurs.
G ..... Global. If this bit is set in both Lo0 and Lo1, then the processor ignores the ASID during TLB lookup.
0 ..... Reserved. Must be written as zeroes, and returns zeroes when read.

Figure 5.4 Fields of the EntryLo0 and EntryLo1 Registers

Notes

The TLB page coherency attribute (*C*) bits specify whether references to the page should be cached; if cached, the algorithm selects between several coherency attributes. Table 5.1 shows the coherency attributes selected by the *C* bits.

C(5:3) Value	Page Coherency Attribute
0	Cacheable, noncoherent, write-through, no write allocate
1	Cacheable, noncoherent, write-through, write allocate
2	Uncached
3	Cacheable, noncoherent, write-back
4 - 7	Reserved

Table 5.1 TLB Page Coherency (C) Bit Values

CP0 Registers

The following sections describe the CP0 registers (shown in Figure 5.1 on page 5-1) that are assigned specifically as a software interface with memory management (each register is followed by its register number in parentheses).

- *Index register (CP0 register number 0)*
- *Random register (1)*
- *EntryLo0 (2) and EntryLo1 (3) registers*
- *PageMask register (5)*
- *Wired register (6)*
- *EntryHi register (10)*
- *PRId register (15)*
- *Config register (16)*
- *LLAddr register (17)*
- *TagLo (28) and TagHi (29) registers*

Index Register (0)

The *Index* register is a 32-bit, read/write register containing six bits to index an entry in the TLB. The high-order bit of the register shows the success or failure of a TLB Probe (TLBP) instruction. The *Index* register also specifies the TLB entry affected by TLB Read (TLBR) or TLB Write Index (TLBWI) instructions. Figure 5.5 shows the format of the *Index* register. Table 5.2, which follows the figure, describes the *Index* register fields.

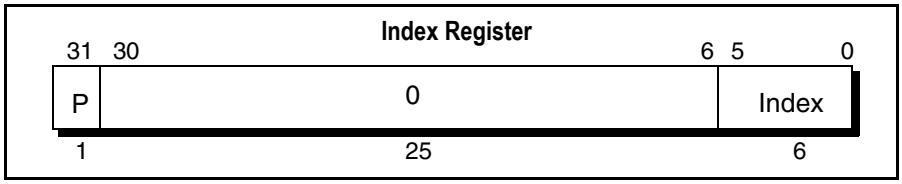


Figure 5.5 Index Register

Field	Description
P	Probe failure. Set to 1 when the previous TLBProbe (TLBP) instruction was unsuccessful.
Index	Index to the TLB entry affected by the TLBRead and TLBWrite instructions
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 5.2 Index Register Field Descriptions

Notes

Random Register (1)

The *Random* register is a read-only register of which six bits index an entry in the TLB. This register decrements as each instruction executes, and its values range between an upper and a lower bound, as follows:

- *A lower bound is set by the number of TLB entries reserved for exclusive use by the operating system (the contents of the Wired register).*
- *An upper bound is set by the total number of TLB entries. Thus the upper bound is 47 (The TLB entries are number from 0 to 47).*

The RISCore4000 counts only valid instructions. The *Random* register specifies the entry in the TLB that is affected by the TLB Write Random instruction. The register does not need to be read for this purpose; however, the register is readable to verify proper operation of the processor. To simplify testing, the *Random* register is set to the value of the upper bound upon system reset. This register is also set to the upper bound when the *Wired* register is written. Figure 5.6 shows the format of the *Random* register; Table 5.3 on page 5-4 describes the *Random* register fields.

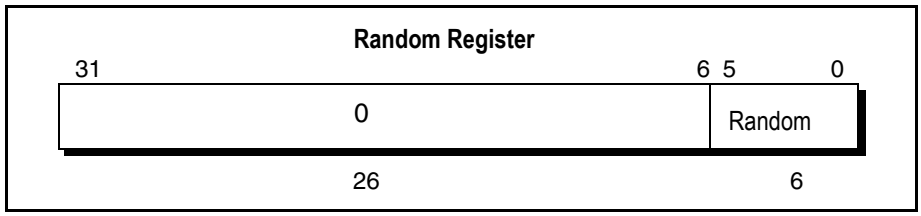


Figure 5.6 Random Register

Field	Description
Random	TLB random index
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 5.3 Random Register Field Descriptions

EntryLo0 (2), and EntryLo1 (3) Registers

The *EntryLo* register consists of two registers that have identical formats:

- *EntryLo0 is used for even virtual pages.*
- *EntryLo1 is used for odd virtual pages.*

The *EntryLo0* and *EntryLo1* registers are read/write registers. They hold the physical page frame number (PFN) of the TLB entry for even and odd pages, respectively, when performing TLB read and write operations. Figure 5.4 on page 5-2 shows the format of these registers.

PageMask Register (5)

The *PageMask* register is a read/write register used for reading from or writing to the TLB. This register holds a comparison mask that sets the variable page size for each TLB entry, as shown in Table 5.4.

TLB read and write operations use this register as either a source or a destination; when virtual addresses are presented for translation into physical address, the corresponding bits in the TLB identify which virtual address bits among bits 24:13 are used in the comparison.

When the *Mask* field is not one of the values shown in , the operation of the TLB is undefined.

Notes

Page Size	Bit											
	24	23	22	21	20	19	18	17	16	15	14	13
4 Kbytes	0	0	0	0	0	0	0	0	0	0	0	0
16 Kbytes	0	0	0	0	0	0	0	0	0	0	1	1
64 Kbytes	0	0	0	0	0	0	0	0	1	1	1	1
256 Kbytes	0	0	0	0	0	0	1	1	1	1	1	1
1 Mbyte	0	0	0	0	1	1	1	1	1	1	1	1
4 Mbytes	0	0	1	1	1	1	1	1	1	1	1	1
16 Mbytes	1	1	1	1	1	1	1	1	1	1	1	1

Table 5.4 Mask Field Values for Page Sizes

Wired Register (6)

The *Wired* register is a read/write register that specifies the boundary between the *wired* and *random* entries of the TLB, as shown in Figure 5.7. Wired entries are nonreplaceable entries, which cannot be overwritten by a TLB write random operation. Random entries can be overwritten.

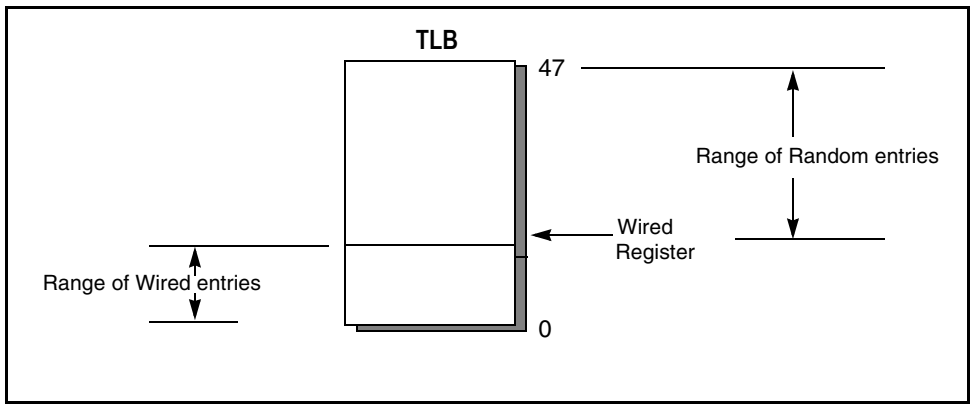


Figure 5.7 Wired Register Boundary

The *Wired* register is set to 0 upon system reset. Writing this register also sets the *Random* register to the value of its upper bound (see *Random* register, above). Figure 5.8 shows the format of the *Wired* register; Table 5.5, which follows the figure, describes the register fields.

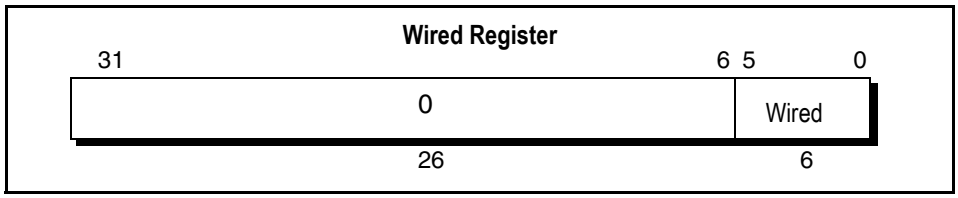


Figure 5.8 Wired Register

Field	Description
Wired	TLB Wired boundary (the number of wired TLB entries)
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 5.5 Wired Register Field Descriptions

Notes

EntryHi Register (10)

The *EntryHi* register holds the high-order bits of a TLB entry for TLB read and write operations and is accessed by the TLB Probe, TLB Write Random, TLB Write Indexed, and TLB Read Indexed instructions. Figure 5.3 shows the format of this register. When either a TLB refill, TLB invalid, or TLB modified exception occurs, the *EntryHi* register is loaded with the virtual page number (VPN2) and the ASID of the virtual address that did not have a matching TLB entry. (See Chapter 5 for more information about these exceptions.)

Processor Revision Identifier (PRId) Register (15)

The 32-bit, read-only *Processor Revision Identifier* (*PRId*) register contains information identifying the implementation and revision level of the CPU and CP0. Figure 5.9 shows the format of the *PRId* register; Table 5.6 describes the *PRId* register fields.

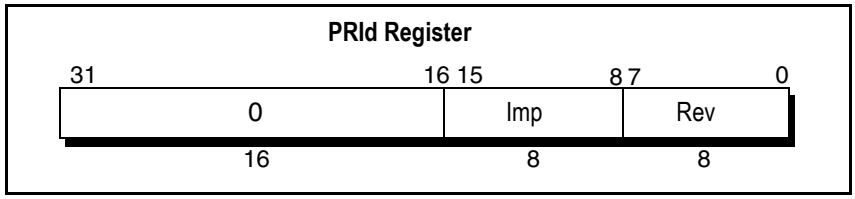


Figure 5.9 Processor Revision Identifier Register Format

Field	Description	
Imp	Implementation number	RC64474/RC64475 = 0x30
Rev	Revision number	0X00
0	Reserved. Must be written as zeroes, returns zeroes when read.	

Table 5.6 PRId Register Fields

The low-order byte (bits 7:0) of the *PRId* register is interpreted as a revision number, and the high-order byte (bits 15:8) is interpreted as an implementation number. The implementation number of the Rc64474/ RC64475 processors is 0x30. The content of the high-order halfword (bits 31:16) of the register are reserved. The revision number is stored as a value in the form *y.x*, where *y* is a major revision number in bits 7:4 and *x* is a minor revision number in bits 3:0.

The revision number can distinguish some chip revisions, however there is no guarantee that changes to the chip will necessarily be reflected in the *PRId* register, or that changes to the revision number necessarily reflect real chip changes. For this reason, these values are not listed and software should not rely on the revision number in the *PRId* register to characterize the chip. Certain attributes, such as cache size, are independent of implementation number.

Config Register (16)

The *Config* register specifies various configuration options selected on RISCore4000 processors; Table 5.7 lists these options. Some configuration options, as defined by *Config* bits 31:3, are set by the hardware during reset and are included in the *Config* register as read-only status bits for the software to access. The K0 field is the only read/write field (as indicated by *Config* register bits 2:0) and controlled by software; on reset these fields are undefined. Figure 5.10 shows the format of the *Config* register; Table 5.7, which follows the figure, describes the *Config* register fields.

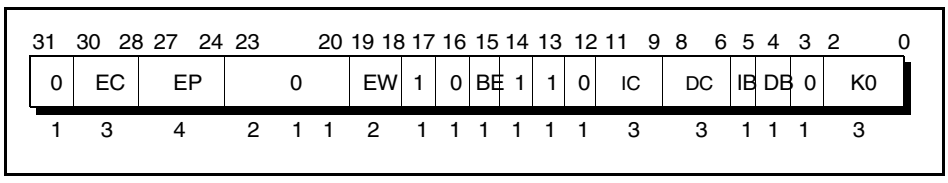


Figure 5.10 Config Register Format

Notes

Field	Description
EC	System clock ratio: 0 → processor clock frequency divided by 2 1 → processor clock frequency divided by 3 2 → processor clock frequency divided by 4 3 → processor clock frequency divided by 5 4 → processor clock frequency divided by 6 5 → processor clock frequency divided by 7 6 → processor clock frequency divided by 8 7 Reserved
EP	Writeback data rate: 0 → DDDD — Doubleword every cycle 1 → DDxDDx — 2 Doublewords every 3 cycles 2 → DDxxDDxx — 2 Doublewords every 4 cycles 3 → DxDxDxDx — 2 Doublewords every 4 cycles 4 → DDxxxDDxxx — 2 Doublewords every 5 cycles 5 → DDxxxxDDxxxx — 2 Doublewords every 6 cycles 6 → DxxDxxDxxDxx — 2 Doublewords every 6 cycles 7 → DDxxxxxDDxxxxx — 2 Doublewords every 7 cycles 8 → DxxxDxxxDxxxDxxx — 2 Doublewords every 8 cycles 9 - 15 — Reserved
EW	System bus size selection 0 → SysAd = 64 bits 1 → SysAD = 32 bits
BE	BigEndianMem 0 → Little endian 1 → Big endian
IC	Primary I-cache Size (I-cache size = 2^{12+IC} bytes). In the RC64474/475 processor, this is set to 16 Kbytes (IC = 010)
DC	Primary D-cache Size (D-cache size = 2^{12+DC} bytes). In the RC64474/475 processor, this is set to 16 Kbytes (DC = 010)
IB	Primary I-cache line size 1 → 32 bytes (8 Words)
DB	Primary D-cache line size 1 → 32 bytes (8 Words)
K0	*kseg0* coherency algorithm (see *EntryLo0* and *EntryLo1* registers)
Others	Reserved. Returns indicated values when read.

Table 5.7 Config Register Fields

Load Linked Address (LLAddr) Register (17)

The read/write *Load Linked Address* (*LLAddr*) register contains the physical address read by the most recent Load Linked instruction. This register is for diagnostic purposes only, and serves no function during normal operation.

Figure 5.11 shows the format of the *LLAddr* register; *PAddr* represents bits of the physical address, PA(35:4)

Notes

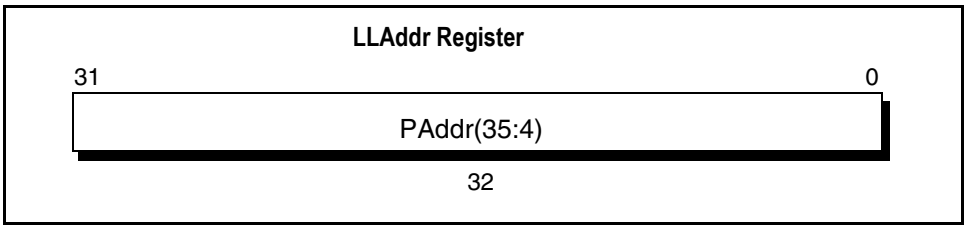


Figure 5.11 LLAddr Register Format

Cache Tag Registers [TagLo (28) and TagHi (29)]

The *TagLo* and *TagHi* registers are 32-bit read/write registers that hold the primary cache tag and parity during cache initialization, cache diagnostics, or cache error processing. The *Tag* registers are written by the CACHE and MTC0 instructions. The *P* field of these registers is ignored on Index Store Tag operations. Parity is computed by the store operation. Figure 5.12 shows the format of these registers for primary cache operations. Table 5.8 lists the field definitions of the *TagLo* and *TagHi* registers.

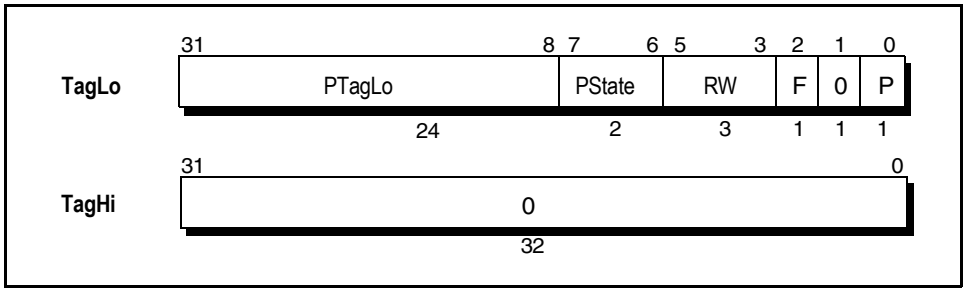


Figure 5.12 TagLo and TagHi Register (P-cache) Formats

Field	Description
PTagLo	Specifies the physical address bits 35:12
PState	Specifies the primary cache state
P	Specifies the primary tag even parity bit
F	The FIFO bit used to implement FIFO refill of the cache
RW	Read/Write bits.
0	Reserved. Must be written as zeroes; returns zeroes when read

Table 5.8 Cache Tag Register Fields

Address Translation Process

During virtual-to-physical address translation, the CPU compares the 8-bit ASID (if the Global bit, *G*, is not set) of the virtual address to the ASID of the TLB entry to see if there is a match. The following comparison is also made:

- *For the 64-bit virtual addresses, the highest 15-to-27 bits (depending upon the page size) of the virtual address are compared to the contents of the TLB virtual page number.*

If a TLB entry matches, the physical address and access control bits (*C*, *D*, and *V*) are retrieved from the matching TLB entry. While the *V* bit of the entry must be set for a valid translation to take place, it is not involved in the determination of a matching TLB entry. Figure 5.13 illustrates the TLB address translation process.

Notes

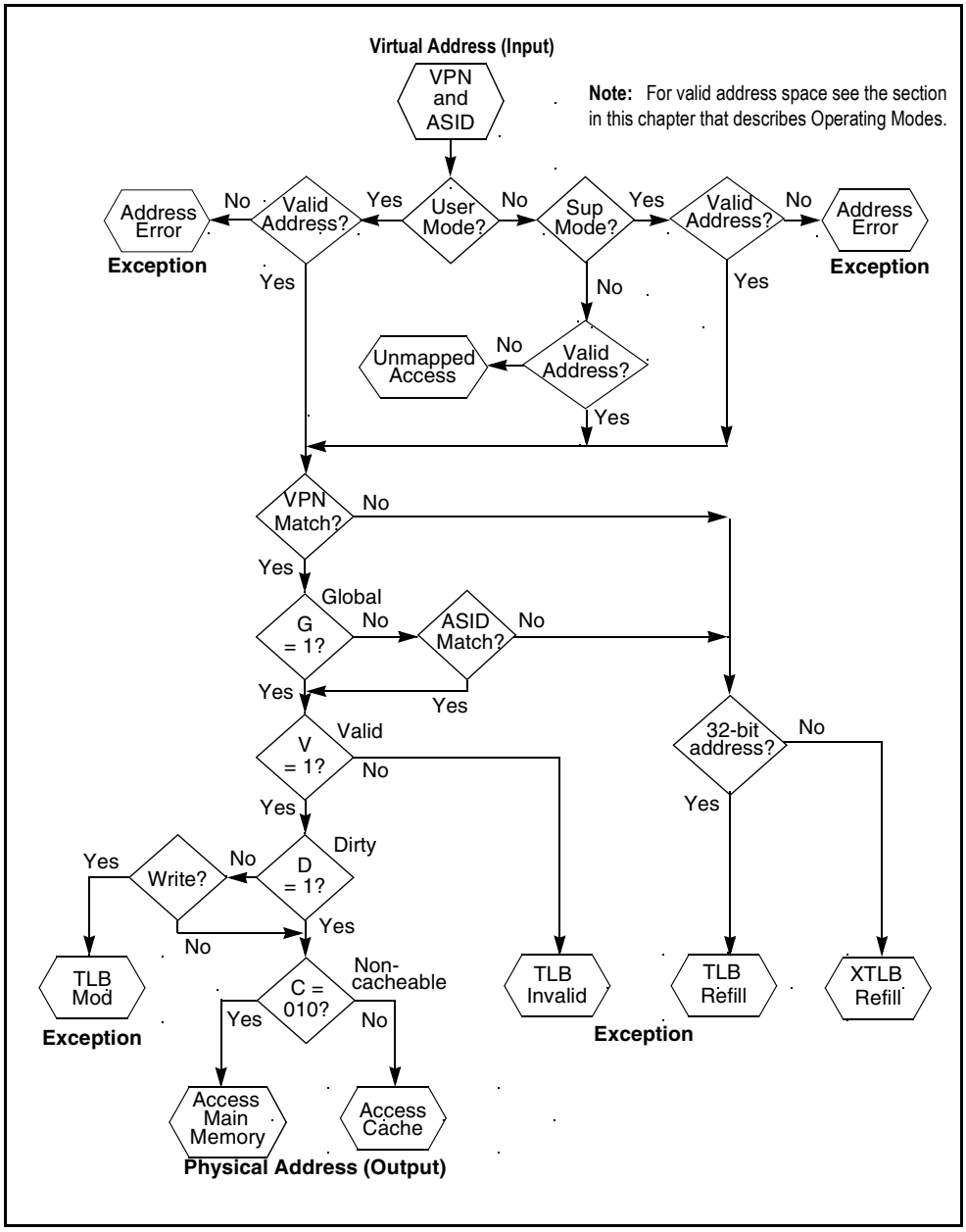


Figure 5.13 TLB Address Translation

TLB Misses

If there is no TLB entry that matches the virtual address, a TLB miss exception occurs. If the access control bits (*D* and *V*) indicate that the access is not valid, a TLB modification or TLB invalid exception occurs. If the *C* bits equal 010_2, the physical address that is retrieved accesses main memory, bypassing the cache.

TLB Instructions

Table 5.9 lists the instructions that the CPU provides for working with the TLB. Refer to the *IDT MIPS Microprocessor Family Software Reference Manual* for a detailed description of these instructions.

Notes

Op Code	Description of Instruction
TLBP	Translation Lookaside Buffer Probe
TLBR	Translation Lookaside Buffer Read
TLBWI	Translation Lookaside Buffer Write Index
TLBWR	Translation Lookaside Buffer Write Random

Table 5.9 Translation Lookaside Buffer Instructions

CPU Exception Processing

Notes

Introduction

This chapter describes the CPU exception processing, discusses the format and use of each CPU exception register and concludes with a description of each exception's cause as well as CPU service procedures. For information about Floating-Point Unit exceptions, refer to Chapter 7.

How Does Exception Processing Work?

The processor receives exceptions from a number of sources, including address translation errors, arithmetic overflows, I/O interrupts, and system calls. When the CPU detects one of these exceptions, the normal sequence of instruction execution is suspended and the processor enters Kernel mode. Refer to Chapter 4 for a description of system operating modes.

The processor then disables interrupts and forces execution of a software exception processor (called a *handler*) located at a fixed address. The handler may save the context of the processor, including the contents of the program counter, the current operating mode (User or Kernel), and the status of the interrupts (enabled or disabled). This context would be saved so it can be restored when the exception has been serviced.

When an exception occurs, the CPU loads the *Exception Program Counter* (*EPC*) register with a location where execution can restart after the exception has been serviced. The restart location in the *EPC* register is the address of the instruction that caused the exception or, if the instruction was executing in a branch delay slot, the address of the branch instruction immediately preceding the delay slot. Registers described later in the chapter assist in this exception processing by retaining address, cause and status information. For an illustration of the exception handling process, refer to the flowcharts at the end of this chapter.

Exception Processing Registers

This section describes the CP0 registers that are used in exception processing. Each register has a unique identification number called a *register number*. For example, the *ECC* register is register number 26. The remaining CP0 registers are used in memory management, as described in Chapter 4. Software examines the CP0 registers during exception processing to determine the cause of the exception and the state of the CPU at the time the exception occurred. Table 6.1 lists the register used in exception processing. A description of each register follows the table.

Register Name	Register #
Context	4
BadVAddr (Bad Virtual Address)	8
Count	9
Compare register	11
Status	12
Cause	13
EPC (Exception Program Counter)	14
XContext	20
ECC	26
CacheErr (Cache Error and Status)	27
ErrorEPC (Error Exception Program Counter)	30

Table 6.1 CP0 Exception Processing Registers

Notes

Context Register (4)

The *Context* register is a read/write register containing the pointer to an entry in the page table entry (PTE) array; this array is an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the CPU loads the TLB with the missing translation from the PTE array.

Normally, the operating system uses the *Context* register to address the current page map which resides in the kernel-mapped segment, *kseg3*. The *Context* register duplicates some of the information provided in the *BadVAddr* register, but the information is arranged in a form that is more useful for a software TLB exception handler. Figure 6.1 shows the format of the *Context* register; Table 6.2, which follows the figure, describes the *Context* register fields.

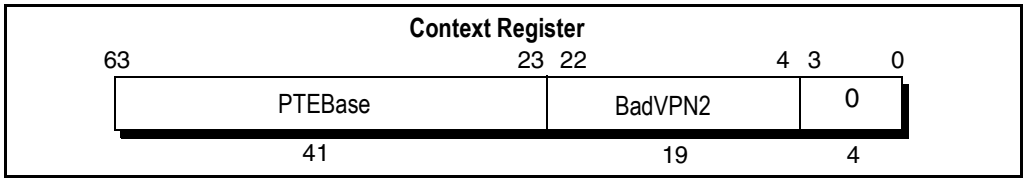


Figure 6.1 Context Register Format

Field	Description
BadVPN2	This field is written by hardware on a miss. It contains the virtual page number (VPN) of the most recent virtual address that did not have a valid translation.
PTEBase	This field is a read/write field for use by the operating system. It is normally written with a value that allows the operating system to use the *Context* register as a pointer into the current PTE array in memory.

Table 6.2 Context Register Fields

Bad Virtual Address Register (BadVAddr) (8)

The Bad Virtual Address register (*BadVAddr*) is a read-only register that displays the most recent virtual address that caused one of the following exceptions: Address Error (e.g., unaligned access), TLB Invalid, TLB Modified, TLB Refill, Virtual Coherency Data Access, or Virtual Coherency Instruction Fetch. The processor does not write to the *BadVAddr* register when the *EXL* bit in the *Status* register is set to a 1. Figure 6.2 shows the format of the *BadVAddr* register.

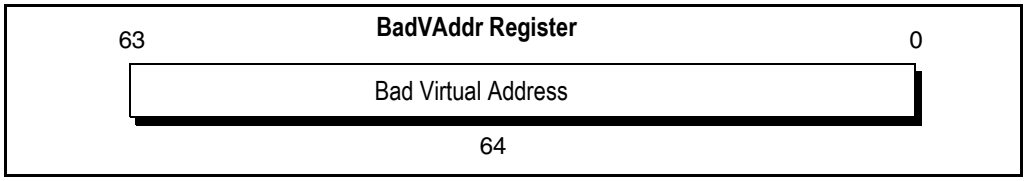


Figure 6.2 BadVAddr Register Format

Note: The *BadVAddr* register does not save any information for bus errors, since bus errors are not addressing errors.

Count Register (9)

The *Count* register acts as a timer, incrementing at a constant rate—half the maximum instruction issue rate—whether or not an instruction is executed, retired, or any forward progress is made through the pipeline. This register can be read or written. It can be written for diagnostic purposes or system initialization; for example, to synchronize processors. Figure 6.3 shows the format of the *Count* register.

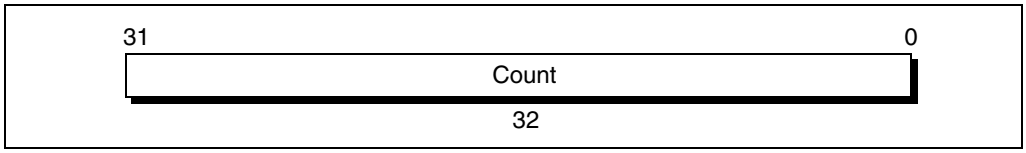


Figure 6.3 Count Register Format

Notes

Compare Register (11)

The *Compare* register acts as a timer, and (see also the *Count* register) maintains a stable value that does not change on its own. When the value of the *Count* register equals the value of the *Compare* register, interrupt bit *IP(7)* in the *Cause* register is set. If the timer interrupt was enabled at boot time, an interrupt will occur as soon as the interrupt is enabled. Writing a value to the *Compare* register, as a side effect, clears the timer interrupt. For diagnostic purposes, the *Compare* register is a read/write register. However, in normal use the *Compare* register is write-only. Figure 6.4 shows the format of the *Compare* register.

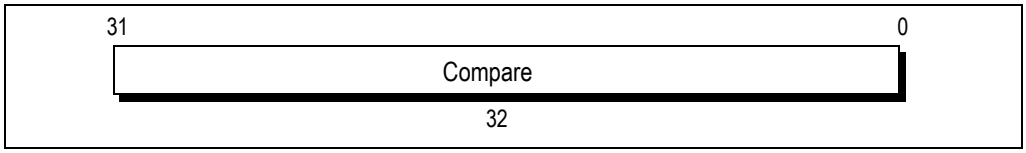


Figure 6.4 Compare Register Format

Status Register (12)

The *Status* register (SR) is a read/write register that contains theoperating mode, interrupt enabling, and the diagnostic states of the processor. The following list describes the more important *Status* register fields; Figure 6.5 show the format of the entire register, including descriptions of the fields. Some of the important fields include:

- *The 8-bit Interrupt Mask (IM) field controls the enabling of eight interrupt conditions. Interrupts must be enabled before they can cause the exception, and the corresponding bits are set in both the Interrupt Mask field of the Status register and the Interrupt Pending field of the Cause register. For more information, refer to the Interrupt Pending (IP) field of the Cause register. IM[1:0] are the masks for the two software interrupts while IM[7:2] correspond to Int[5:0].*
- *The 4-bit Coprocessor Usability (CU) field controls the usability of 4 possible coprocessors. Regardless of the CU0 bit setting, CP0 is always usable in Kernel mode. For all other cases, an instruction for or access to an unusable coprocessor causes an exception.*
- *The 9-bit Diagnostic Status (DS) field (Status[24:16]) is used for self-testing, and checks the cache and virtual memory system.*
- *The Reverse-Endian (RE) bit, bit 25, reverses the endianness of the machine. The processor can be configured as either little-endian or big-endian at system reset. This selection is always used in Kernel and Supervisor modes, and also in User mode when the RE bit is 0. Setting the RE bit to 1 inverts the User mode endianness.*

Figure 6.5 shows the format of the *Status* register. Table 6.3, which follows the figure, describes the *Status* register fields.

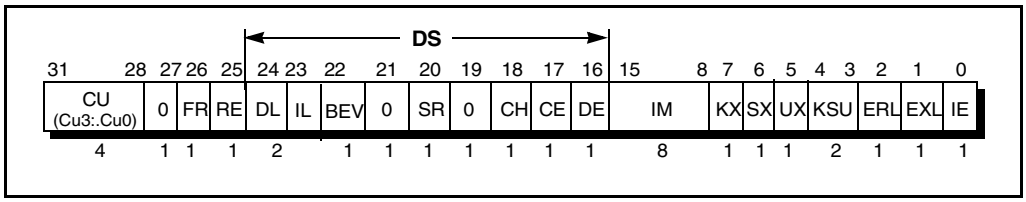


Figure 6.5 Status Register

Notes

Field	Description
CU	Controls the usability of each of the four coprocessor unit numbers. CP0 is always usable when in Kernel mode, regardless of the setting of the CU_0 bit. 1 → usable 0 → unusable
FR	Enables additional floating-point registers 0 → 16 registers 1 → 32 registers
RE	*Reverse-Endian* bit, valid in User mode.
DL	Data cache lock bit allows refill into set A when set A is invalid. Does not inhibit update of the D-cache on store operations. 0 → normal 1→ refill into set A disabled
IL	Instruction cache lock does not prevent refills into set A when set A is invalid. 0 → normal 1→ refill into set A disabled
BEV	Controls the location of TLB refill and general exception vectors. 0 → normal 1→ bootstrap
SR	1→ Indicates that a soft reset or NMI has occurred.
CH	Hit (tag match and valid state) or miss indication for last CACHE Hit Invalidate, Hit Write Back Invalidate, Hit Write Back, or Hit Set Virtual for a primary cache. 0 → miss 1 → hit
CE	Contents of the ECC register set or modify the check bits of the caches when CE = 1; see description of the *ECC* register.
DE	Specifies that cache parity errors cannot cause exceptions. 0 → parity remains enabled 1 → disables parity
0	Reserved. Must be written as zeroes, and returns zeroes when read.
IM	*Interrupt Mask*: controls the enabling of each of the external, internal, and software interrupts. An interrupt is taken if interrupts are enabled, and the corresponding bits are set in both the *Interrupt Mask* field of the *Status* register and the *Interrupt Pending* field of the *Cause* register. IM[7:2] correspond to interrupts Int[5:0] and IM[1:0] to the software interrupts. 0 → disabled 1→ enabled
KX	KX controls whether the TLB Refill Vector or the XTLB Refill Vector address is used for TLB misses on kernel addresses 0 → TLB Refill Vector 1 → XTLB Refill Vector
SX	Enables 64-bit virtual addressing and operations in Supervisor mode. The extended-addressing TLB refill exception is used for TLB misses on supervisor addresses. 0 → 32-bit 1 → 64-bit
UX	Enables 64-bit virtual addressing and operations in User mode. The extended-addressing TLB refill exception is used for TLB misses on user addresses. 0 → 32-bit 1 → 64-bit
KSU	Mode bits 10_2 → User 01_2 → Supervisor 00_2 → Kernel
ERL	Error Level 0 → normal 1 → error
EXL	Exception Level 0 → normal 1 → exception **Note:** When going from 0 to 1, IE should be disabled (0) first. This would be done when preparing to return from the exception handler, such as before executing the ERET instruction.
IE	Interrupt Enable 0 → disable interrupts 1 → enables interrupts

Table 6.3 Status Register Fields

Notes

Status Register Modes and Access States

Interrupts are enabled when all of the following conditions are true:

- *IE = 1*
- *EXL = 0*
- *ERL = 0*

If these conditions are met, the settings of the *IM* bits identify the interrupt.

Note: Setting the IE bit may be delayed by up to 3 cycles. If performing nested interrupts, re-enable the IE bit first.

Operating Modes: The following CPU *Status* register bit settings are required for User, Kernel, and Supervisor modes.

- *The processor is in User mode when KSU = 10_2, EXL = 0, and ERL = 0.*
- *The processor is in Supervisor mode when KSU = 01_2, EXL = 0, and ERL = 0.*
- *The processor is in Kernel mode when KSU = 00_2, or EXL = 1, or ERL = 1.*

32- and 64-bit Virtual Addressing: The following CPU *Status* register bit settings select 32- or 64-bit virtual addressing for User and Supervisor operating modes. Enabling 64-bit virtual addressing permits the execution of 64-bit opcodes and translation of 64-bit virtual addresses. 64-bit virtual addressing for User and Supervisor modes can be set independently but is always used for Kernel mode.

- *The KX field controls whether the TLB Refill Vector or the XTLB Refill Vector address is used for TLB misses on Kernel addresses. 64-bit opcodes are always valid in Kernel mode.*
- *64-bit addressing and operations are enabled for Supervisor mode when SX = 1.*
- *64-bit addressing and operations are enabled for User mode when UX = 1.*

Kernel Address Space Accesses: Access to the kernel address space is allowed when the processor is in Kernel mode. **Supervisor Address Space Accesses**: Access to the supervisor address space is allowed when the processor is in Kernel or Supervisor mode, as described above in the paragraph titled Operating Modes. **User Address Space Accesses**: Access to the user address space is allowed in any of the three operating modes. **Status Register Reset:** The contents of the *Status* register are undefined at reset, except for the following bits — *ERL* and *BEV* = 1. The *SR* bit distinguishes between Reset and Soft Reset (Nonmaskable Interrupt [NMI]).

Cause Register (13)

The 32-bit read/write *Cause* register describes the cause of the most recent exception. Figure 6.6 shows the fields of this register; Table 6.4, which follows the figure, describes the *Cause* register fields. A 5-bit exception code (*ExcCode*) indicates the cause of the most recent exception, as listed in Table 6.5 on page 6-6. All bits in the *Cause* register, with the exception of the *IP(1:0)* bits, are read-only; *IP(1:0)* are used for software interrupts.

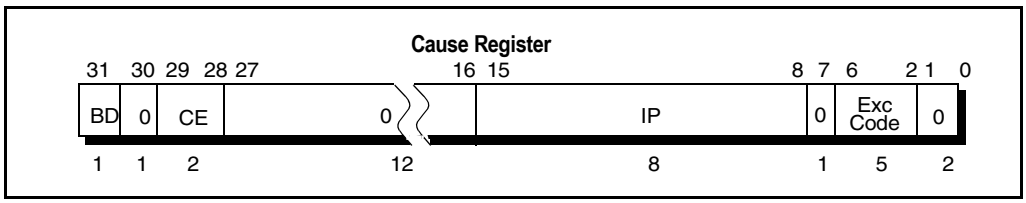


Figure 6.6 Cause Register Format

Notes

Field	Description
BD	Indicates whether the last exception taken occurred in a branch delay slot. 1 → delay slot 0→ normal
CE	Coprocessor unit number referenced when a Coprocessor Unusable exception is taken.
IP	Indicates an interrupt is pending. 1 → interrupt pending 0 → no interrupt
ExcCode	Exception code field (see Table 6.5 on page 6-6)
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 6.4 Cause Register Fields

Exception Code Value	Mnemonic	Description
0	Int	Interrupt
1	Mod	TLB modification exception
2	TLBL	TLB exception (load or instruction fetch)
3	TLBS	TLB exception (store)
4	AdEL	Address error exception (load or instruction fetch)
5	AdES	Address error exception (store)
6	IBE	Bus error exception (instruction fetch)
7	DBE	Bus error exception (data reference: load or store)
8	Sys	Syscall exception
9	Bp	Breakpoint exception
10	RI	Reserved instruction exception
11	CpU	Coprocessor Unusable exception
12	Ov	Arithmetic Overflow exception
13	Tr	Trap exception
14	—	Reserved
15	FPE	Floating-Point exception
16–31	—	Reserved

Table 6.5 Cause Register ExcCode Field

Exception Program Counter (EPC) Register (14)

The Exception Program Counter (*EPC*) is a read/write register that contains the address at which processing resumes after an exception has been serviced.

For synchronous exceptions, the *EPC* register contains either:

- *the virtual address of the instruction that was the direct cause of the exception, or*
- *the virtual address of the immediately preceding branch or jump instruction (which occurs when the instruction is in a branch delay slot, and the Branch Delay bit in the Cause register is set).*

Note: The processor does not write to the *EPC* register when the *EXL* bit in the *Status* register is set to 1. Figure 6.7 shows the format of the *EPC* register.

Notes

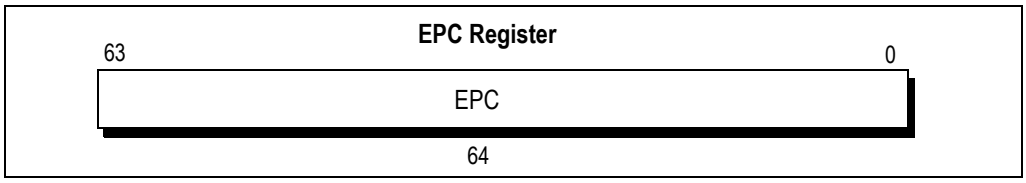


Figure 6.7 EPC Register Format

XContext Register (20)

The read/write *XContext* register contains a pointer to an entry in the page table entry (PTE) array, an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the operating system software loads the TLB with the missing translation from the PTE array. The *XContext* register duplicates some of the information provided in the *BadVAddr* register, and puts it in a form useful for a software TLB exception handler.

The *XContext* register is for use with the XTLB refill handler, which loads TLB entries for references to a 64-bit address space, and is included solely for operating system use. The operating system sets the PTE base field in the register, as needed. Normally, the operating system uses the *XContext* register to address the current page map, which resides in the kernel-mapped segment *kseg3*. Figure 6.8 shows the format of the *XContext* register; Table 6.6, which follows the figure, describes the *XContext* register fields.

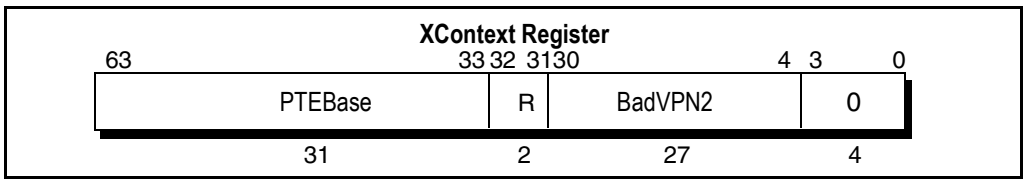


Figure 6.8 XContext Register Format

The 27-bit *BadVPN2* field has bits 39:13 of the virtual address that caused the TLB miss; bit 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4-Kbyte page size, this format may be used directly to address the pair-table of 8-byte PTEs. For other page and PTE sizes, shifting and masking this value produces the appropriate address.

Field	Description
BadVPN2	The *Bad Virtual Page Number*/2 field is written by hardware on a miss. It contains the VPN of the most recent invalidly translated virtual address.
R	The *Region* field contains bits 63:62 of the virtual address. 00_2 = user 01_2 = supervisor 11_2 = kernel.
PTEBase	The *Page Table Entry Base* read/write field is normally written with a value that allows the operating system to use the *Context* register as a pointer into the current PTE array in memory.

Table 6.6 XContext Register Fields

Error Checking and Correcting (ECC) Register (26)

The 8-bit *Error Checking and Correcting* (*ECC*) register reads or writes primary-cache data parity bits for cache initialization, cache diagnostics, or cache error processing. Tag parity is loaded from and stored to the *TagLo* register. The *ECC* register is loaded by the Index Load Tag CACHE operation. Content of the ECC register are:

- *written into the primary data cache on store instructions (instead of the computed parity) when the CE bit of the Status register is set, and*
- *substituted for the computed instruction parity for the CACHE operation Fill*

Notes

To force a cache parity value use the *Status CE* bit and the ECC register. Figure 6.9 shows the format of the *ECC* register; Table 6.7, which follows the figure, describes the register fields.

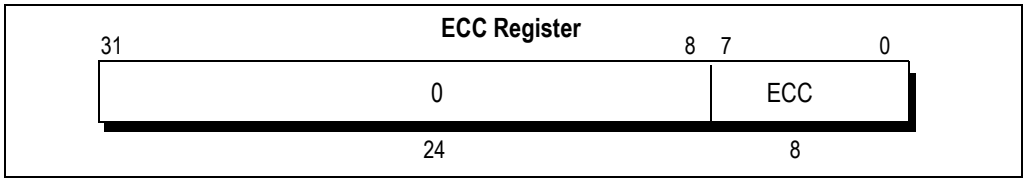


Figure 6.9 ECC Register Format

Field	Description
ECC	An 8-bit field specifying the parity bits read from or written to a primary cache.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 6.7 ECC Register Fields

Cache Error (CacheErr) Register (27)

The 32-bit read-only *CacheErr* register processes parity errors in the primary cache. Parity errors cannot be corrected. The *CacheErr* register holds cache index and status bits that indicate the source and nature of the error. It is loaded when a Cache Error exception is asserted. When a read response returns with bad parity, this exception is also asserted. Figure 6.10 shows the format of the *CacheErr* register. Table 6.8, which follows the figure, describes the *CacheErr* register fields.

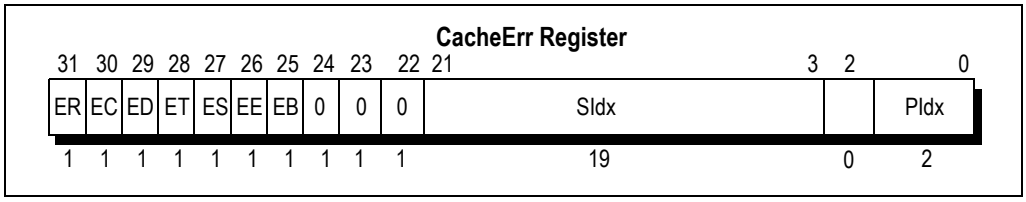


Figure 6.10 CacheErr Register Format

Notes

Field	Description
ER	Type of reference 0 → instruction 1 → data
EC	Cache level of the error 0 → primary 1 → reserved
ED	Indicates if a data field error occurred 0 → no error 1 → error
ET	Indicates if a tag field error occurred 0 → no error 1 → error
ES	Indicates the error occurred accessing processor-managed resources, in response to an external request. 0 → internal reference 1 → external reference Because the RC64474/475 doesn't have any external events that would look in a cache (which is the only processor-managed resource), this bit would not be set under normal operating conditions.
EE	Set if the error occurred on the SysAD bus. Taking a cache error exception sets/clears this bit.
EB	Set if a data error occurred in addition to the instruction error (indicated by the remainder of the bits). If so, this requires flushing the data cache after fixing the instruction error.
SIdx	Physical address 21:3 of the reference that encountered the error.
PIdx	Virtual address 13:12 of the double word in error. To be used with SIdx to construct a virtual index for the primary caches. Only the lower two bits (bits 1 and 0) are vAddr; the high bit (bit 2) is zero.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 6.8 CacheErr Register Fields

Error Exception Program Counter (Error EPC) Register (30)

The *ErrorEPC* register is similar to the *EPC* register, except that *ErrorEPC* is used on parity error exceptions. It is also used to store the program counter (PC) on Reset, Soft Reset, and nonmaskable interrupt (NMI) exceptions.

The read/write *ErrorEPC* register contains the virtual address at which instruction processing can resume after servicing an error. This address can be either:

- *the virtual address of the instruction that caused the exception*
- *the virtual address of the immediately preceding branch or jump instruction, when this address is in a branch delay slot.*

There is no branch delay slot indication for the *ErrorEPC* register.

Figure 6.11 shows the format of the *ErrorEPC* register.

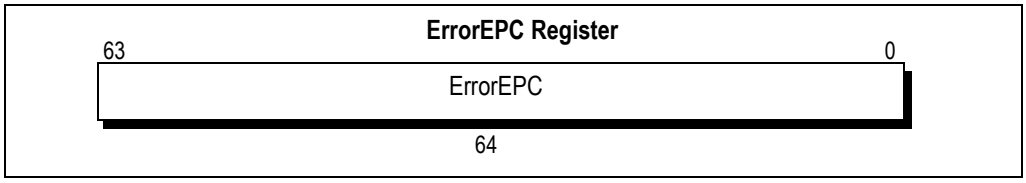


Figure 6.11 ErrorEPC Register Format

Notes

CPU Exceptions

This section describes the processor exceptions: cause of each exception, its processing by the hardware, and servicing by a handler (software). The types of exception, with exception processing operations are as follows:

- *reset*
- *soft rese*
- *nonmaskable interrupt (NMI)*
- *cache error*
- *remaining processor exceptions*

When the *EXL* bit in the *Status* register is 0, either User or Supervisor operating mode is specified by the *KSU* bits in the *Status* register. When the *EXL* bit or the *ERL* bit is a 1, the processor is in Kernel mode. When the processor takes an exception, the *EXL* bit is set to 1, which means the system is in Kernel mode. After saving the appropriate state, the exception handler typically resets the *EXL* bit back to 0. When restoring the state and restarting, the handler sets the *EXL* bit back to 1.

Returning from an exception, also resets the *EXL* bit to 0 (see the ERET instruction in Appendix A). In the following sections, sample hardware pocesses for various exceptions are shown, together with the servicing required by the handler (software).

Reset Exception Process

Figure 6.12 shows the Reset exception process.

```
T: undefined
   Random ← TLBENTRIES–1
   Wired ← 0
   Config <- 0 || EC || EP || 00000000 || BE || 110 || 010 || 010 || 1 || 1 || 0 ||
undefined^3
   ErrorEPC ← PC
   SR ← SR31:23 || 1 || 0 || 0 || SR19:3 || 1 || SR1:0
```

Figure 6.12 Reset Exception Processing

Cache Error Exception Process

Figure 6.13 shows the Cache Error exception process.

```
T: ErrorEPC ← PC
   CacheErr ← ER || EC || ED || ET || ES || EE || EB || 0^25
   SR ← SR31:3 || 1 ||SR1:0
   if SR22 = 1 then        /* What is the BEV bit setting */
      PC ← 0xFFFF FFFF BFC0 0200 + 0x100   /* access boot-PROM area
*/
   else
```

Figure 6.13 Cache Error Exception Processing

Soft Reset and NMI Exception Process

Figure 6.14 shows the Soft Reset and NMI exception process.

```
T: ErrorEPC ← PC
   SR ← SR31:23 || 1 || 0 || 1 || SR19:3 || 1 || SR1:0
```

Figure 6.14 Soft Reset and NMI Exception Processing

Notes

General Exception Process

Figure 6.15 shows the process used for exceptions other than Reset, Soft Reset, NMI, and Cache Error.

```
T: Cause ← BD || 0 || CE || 0^12 || Cause_15:8 || 0 || ExcCode || 0^2
   if SR_1 = 0 then          /* system in User or Supervisor mode with no current
exception */
      EPC ← PC
   endif
   SR ← SR_31:2 || 1 || SR0
   if SR_22 = 1 then         /* What is the BEV bit setting */
      PC ← 0xFFFF FFFF BFC0 0200 + vector   /* access to uncached space
*/
   else
      PC ← 0xFFFF FFFF 8000 0000 + vector    /* access to cached space */
   endif
```

Figure 6.15 General Exception Processing (Except Reset, Soft Reset, NMI, and Cache Error)

Exception Vector Locations

The Reset, Soft Reset, and NMI exceptions are always vectored to location `0xFFFF FFFF BFC0 0000` (virtual address), corresponding to *kseg1*. Addresses for all other exceptions are a combination of a *vector offset* and a *base address*. The base address is determined by the *BEV* bit of the *Status* register, as shown in Table 6.9. Table 6.10 shows the vector offset that is added to the base address to create the exception address.

BEV	RC64474/475 processor vector base	Cache Error base
0	`0xFFFF FFFF 8000 0000`	`0xFFFF FFFF A000 0000`
1	`0xFFFF FFFF BFC0 0200`	`0xFFFF FFFF BFC0 0200`

Table 6.9 Exception Vector Base Addresses

As shown in Table 6.9, when *BEV* = 0, the vector base for the Cache Error exception changes from *kseg0* (`0xFFFF FFFF 8000 0000`) to *kseg1* (`0xFFFF FFFF A000 0000`). When *BEV* = 1, the vector base for the Cache Error exception is `0xFFFF FFFF BFC0 0200`. This is an uncached and unmapped space, allowing the exception to bypass the cache and TLB.

Exception	RC64474/475 Processor Vector Offset
TLB refill, EXL = 0	0x000
XTLB refill, EXL = 0 (X = 64-bit TLB)	0x080
Cache Error	0x100
Others	0x180

Table 6.10 Exception Vector Offsets

Exception Priority

The remainder of this chapter describes exceptions in the order of their priority, as shown in Table 6.11. While more than one exception can occur for a single instruction, only the exception with the highest priority is reported.

Notes

Exception Priority			
1	Reset *(highest priority)*	**9**	Integer overflow, Trap, System Call, Breakpoint, Reserved Instruction, Coprocessor Unusable, or Floating-Point Exception
2	Soft Reset	**10**	Address error — Data access
3	Nonmaskable Interrupt (NMI)	**11**	TLB refill — Data access
4	Address error — Instruction fetch	**12**	TLB invalid — Data access
5	TLB refill — Instruction fetch	**13**	TLB modified — Data write
6	TLB invalid — Instruction fetch	**14**	Cache error — Data access
7	Cache error — Instruction fetch	**15**	Bus error — Data access
8	Bus error — Instruction fetch	**16**	Interrupt *(lowest priority)*

Table 6.11 Exception Priority Order

Generally speaking, the exceptions described in the following sections are handled ("processed") by hardware; these exceptions are then serviced by software.

Reset Exception

Cause: The Reset exception occurs when the ColdReset*[1] signal is asserted and then deasserted.

Processing: The CPU provides a special exception vector for this exception of: `0xFFFF FFFF BFC0 0000`. The Reset vector resides in unmapped and uncached CPU address space, so the hardware need not initialize the TLB or the cache to process this exception. It also means the processor can fetch and execute instructions while the caches and virtual memory are in an undefined state. The contents of all registers in the CPU are undefined when this exception occurs, except for the following register fields:

- *In the Status register, SR is cleared to 0, and ERL and BEV are set to 1. All other bits are undefined.*
- *The Random register is initialized to the value of its upper bound.*
- *The Wired register is initialized to 0.*
- *Some of the Config Register bits are initialized from the boot-time mode stream.*

Reset exception processing is shown in Figure 6.12 on page 6-10.

Servicing: The Reset exception is serviced by:

- *initializing all processor registers, coprocessor registers, caches, and the memory system*
- *performing diagnostic tests*
- *bootstrapping the operating system*

Maskable: No.

Soft Reset Exception

The primary purpose of the Soft Reset exception is to reinitialize the processor after a fatal error during normal operations. Unlike an NMI, all cache and bus state machines are reset by this exception. Like Reset, Soft Reset can be used on the processor in any state; the caches, TLB, and normal exception vectors need not be properly initialized. Soft Reset preserves the state of the caches and memory system, while resetting the bus state and cache state machine.

Cause: The Soft Reset exception occurs in response to the Reset* input signal, and execution begins at the Reset vector when Reset* is deasserted.

1. In the following sections (and throughout this manual) a signal followed by an asterisk, such as **Reset***, is low active.

Notes

Processing: The Reset exception vector is used for this exception, located within unmapped and uncached address space so that the cache and TLB need not be initialized to process this exception. When a Soft Reset occurs, the *SR* bit of the *Status* register is set to distinguish this exception from a Reset exception. When this exception occurs, the contents of all registers are preserved except for:

- *ErrorEPC register, which contains the restart PC*
- *ERL bit of the Status register, which is set to 1*
- *SR bit of the Status register, which is set to 1*
- *BEV bit of the Status register, which is set to 1*

Because the Soft Reset can abort cache and bus operations, cache and memory state is undefined when this exception occurs. Soft reset exception processing is shown in Figure 6.14 on page 6-10.

Servicing: The Soft Reset exception is serviced by saving the current processor state for diagnostic purposes, and reinitializing for the Reset exception.

Maskable: No

Nonmaskable Interrupt (NMI) Exception

Cause: The Nonmaskable Interrupt (NMI) exception occurs in response to the falling edge of the NMI pin, or an external write to the **Int*[6]** bit of the *Interrupt* register. NMI occurs regardless of the settings of the *EXL*, *ERL*, and the *IE* bits in the *Status* register.

Processing: The Reset exception vector is used for this exception. This vector is located within unmapped and uncached address space so that the cache and TLB need not be initialized to process an NMI interrupt. When an NMI exception occurs, the *SR* bit of the *Status* register is set to differentiate this exception from a Reset exception.

Because an NMI can occur in the midst of another exception, it is not normally possible to continue program execution after servicing an NMI. Unlike Reset and Soft Reset, NMI is taken only at instruction boundaries. The state of the caches and memory system are preserved by this exception. To terminate a pending read that has hung the best approach is to return a bus error. However, if you wish to use a CPU exception to indicate a hung read, Soft Reset is preferable to NMI.

When this exception occurs, the contents of all registers are preserved except for:

- *ErrorEPC register, which contains the restart PC*
- *ERL bit of the Status register, which is set to 1*
- *SR bit of the Status register, which is set to 1*
- *BEV bit of the Status register, which is set to 1*

NMI exception processing is shown in Figure 6.14 on page 6-10.

Servicing: The NMI exception is serviced by saving the current processor state for diagnostic purposes, and reinitializing the system for the Reset exception.

Maskable: No

Address Error Exception

Cause: The Address Error exception occurs when an attempt is made to execute one of the following:

- *load or store a doubleword that is not aligned on a doubleword boundary (except for use of special instruction)*
- *load, fetch, or store a word that is not aligned on a word boundary (except for use of special instruction)*
- *load or store a halfword that is not aligned on a halfword boundary*
- *reference the kernel address space from User or Supervisor mode*
- *reference the supervisor address space from User mode*

Notes

Processing: The common exception vector is used for this exception. The *AdEL* or *AdES* code in the *Cause* register is set, indicating whether the instruction (shown by the *EPC* register and *BD* bit in the *Cause* register) caused the exception with an instruction reference, load operation, or store operation. When this exception occurs, the *BadVAddr* register retains the virtual address that was not properly aligned or referenced protected address space. The contents of the *VPN* field of the *Context* and *EntryHi* registers are undefined, as are the contents of the *EntryLo* register.

The *EPC* register contains the address of the instruction that caused the exception, unless this instruction is in a branch delay slot. If it is in a branch delay slot, the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set as indication. Address Error exception processing is shown in Figure 6.15 on page 6-11.

Servicing: Typically the process executing at the time is handed a segmentation violation signal. This error is usually fatal to the process incurring the exception.To resume execution, the *EPC* register must be altered so that the unaligned reference instruction does not re-execute; this is accomplished by adding a value of 4 to the *EPC* register (*EPC* register + 4) before returning. If an unaligned reference instruction is in a branch delay slot, interpretation of the branch instruction is required to resume execution.

Maskable: No.

TLB Exceptions

The following three types of TLB exceptions can occur:

- *TLB Refill occurs when there is no TLB entry that matches an attempted reference to a mapped address space.*
- *TLB Invalid occurs when a virtual address reference matches a TLB entry that is marked invalid.*
- *TLB Modified occurs when a store operation virtual address reference to memory matches a TLB entry which is marked valid but is not dirty (the entry is not writable).*

TLB Refill Exception

Cause: The TLB refill exception occurs when there is no TLB entry to match a reference to a mapped address space.

Processing: There are two special exception vectors for this exception; one for references to 32-bit virtual address spaces, and one for references to 64-bit virtual address spaces. The *UX, SX,* and *KX* bits of the *Status* register determine whether the user, supervisor or kernel address spaces referenced are 32-bit or 64-bit spaces.

All references use these vectors when the *EXL* bit is set to 0 in the *Status* register. This exception sets the *TLBL* or *TLBS* code in the *ExcCode* field of the *Cause* register. This code indicates whether the instruction, as shown by the *EPC* register and the *BD* bit in the *Cause* register, caused the miss by an instruction reference, load operation, or store operation.

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers hold the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The *Random* register normally suggests a valid location in which to place the replacement TLB entry.

The contents of the *EntryLo* register are undefined. The *EPC* register contains the address of the instruction that caused the exception, unless this instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set. TLB Refill exception processing is shown in Figure 6.15 on page 6-11.

Servicing: To service this exception, the contents of the Context or XContext register are used as a virtual address to fetch memory locations containing the physical page frame and access control bits for a pair of TLB entries. The two entries are placed into the EntryLo0/EntryLo1 register; the EntryHi and EntryLo registers are written into the TLB.

Notes

It is possible that the virtual address used to obtain the physical address and access control information is on a page that is not resident in the TLB. This condition is processed by allowing a TLB refill exception in the TLB refill handler. This second exception goes to the common exception vector because the *EXL* bit of the *Status* register is set.

Maskable: No.

TLB Invalid Exception

Cause: The TLB invalid exception occurs when a virtual address reference matches a TLB entry that is marked invalid (TLB valid bit cleared).

Processing: The common exception vector is used for this exception. The *TLBL* or *TLBS* code in the *ExcCode* field of the *Cause* register is set. This indicates whether the instruction, as shown by the *EPC* register and *BD* bit in the *Cause* register, caused the miss by an instruction reference, load operation, or store operation. When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers contain the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The *Random* register normally contains a valid location in which to put the replacement TLB entry. The contents of the *EntryLo* registers are undefined.

The *EPC* register contains the address of the instruction that caused the exception unless this instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set. TLB Invalid exception processing is shown in Figure 6.15 on page 6-11.

Servicing: A TLB entry is typically marked invalid when one of the following is true:

- *a virtual address does not exist*
- *the virtual address exists, but is not in main memory (a page fault)*
- *a trap is desired on any reference to the page (for example, to maintain a reference bit or during debug)*

After servicing the cause of a TLB Invalid exception, the TLB entry is located with TLBP (TLB Probe), and replaced by an entry with that entry's *Valid* bit set.

Maskable: No.

TLB Modified Exception

Cause: The TLB modified exception occurs when a store operation virtual address reference to memory matches a TLB entry that is marked valid but is not dirty and therefore is not writable.

Processing: The common exception vector is used for this exception, and the *Mod* code in the *Cause* register is set. When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers contain the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The contents of the *EntryLo* registers are undefined. The *EPC* register contains the address of the instruction that caused the exception unless that instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set. TLB Modified exception processing is shown in Figure 6.15 on page 6-11.

Servicing: The kernel uses the failed virtual address or virtual page number to identify the corresponding access control information. The page identified may or may not permit write accesses; if writes are not permitted, a write protection violation occurs. If write accesses are permitted, the page frame is marked dirty/writable by the kernel in its own data structures. The TLBP instruction places the index of the TLB entry that must be altered into the Index register. The EntryLo register is loaded with a word containing the physical page frame and access control bits (with the D bit set), and the EntryHi and EntryLo registers are written into the TLB.

Maskable: No.

Cache Error Exception

Cause: The Cache Error exception occurs when a primary cache parity error is detected.

Notes

Processing: The processor sets the *ERL* bit in the *Status* register, saves the exception restart address in *ErrorEPC* register, and then transfers to a special vector in uncached space:

If the BEV bit = 0, the vector is `0xFFFF FFFF A000 0100`.

If the BEV bit = 1, the vector is `0xFFFF FFFF BFC0 0300`.

No other registers are changed. Cache Error exception processing is shown in Figure 6.13 on page 6-10.

Servicing: All errors should be logged. To correct cache parity errors the system uses the CACHE instruction to invalidate the cache block, overwrites the old data through a cache miss, and resumes execution with an ERET. Other errors are not correctable and are likely to be fatal to the current process.

Maskable: This exception is maskable by the *DE* bit of the *Status* register.

Bus Error Exception

Cause: A Bus Error exception is raised by board-level circuitry for events such as bus time-out, backplane bus parity errors, and invalid physical memory addresses or access types. A Bus Error exception occurs only when a cache miss refill, uncached reference, or unbuffered write occurs synchronously; a Bus Error exception resulting from a buffered write transaction must be reported using the general interrupt mechanism.

Processing: The common interrupt vector is used for a Bus Error exception. The *IBE* or *DBE* code in the *ExcCode* field of the *Cause* register is set, signifying whether the instruction (as indicated by the *EPC* register and *BD* bit in the *Cause* register) caused the exception by an instruction reference, load operation, or store operation. The *EPC* register contains the address of the instruction that caused the exception, unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set. Bus Error processing is shown in Figure 6.15 on page 6-11.

Servicing: The physical address at which the fault occurred can be computed from information available in the CP0 registers.

- *If the IBE code in the Cause register is set (indicating an instruction fetch reference), the virtual address is contained in the EPC register.*
- *If the DBE code is set (indicating a load or store reference), the instruction that caused the exception is located at the virtual address contained in the EPC register (or 4+ the contents of the EPC register if the BD bit of the Cause register is set).*

The virtual address of the load and store reference can then be obtained by interpreting the instruction. The physical address can be obtained by using the TLBP instruction and reading the *EntryLo* register to compute the physical page number. The process executing at the time of this exception is handed a bus error signal, which is usually fatal.

Maskable: No.

Integer Overflow Exception

Cause: An Integer Overflow exception occurs when an ADD, ADDI, SUB, DADD, DADDI or DSUB[1] instruction results in a 2's complement overflow.

Processing: The common exception vector is used for this exception, and the *OV* code in the *Cause* register is set. The *EPC* register contains the address of the instruction that caused the exception unless the instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set. Integer Overflow exception processing is shown in Figure 6.15 on page 6-11.

Servicing: The process executing at the time of the exception is handed a floating-point exception/integer overflow signal. This error is usually fatal to the current process.

Maskable: No.

1. See the *IDT MIPS Microprocessor FamilySoftware Reference Manual* for instruction description.

Notes

Trap Exception

Cause: The Trap exception occurs when a TGE, TGEU, TLT, TLTU, TEQ, TNE, TGEI, TGEUI, TLTI, TLTUI, TEQI, or TNEI instruction results in a TRUE condition.

Processing: The common exception vector is used for this exception, and the *Tr* code in the *Cause* register is set. The *EPC* register contains the address of the instruction causing the exception unless the instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set. Trap exception processing is shown in Figure 6.15 on page 6-11.

Servicing: The process executing at the time of a Trap exception is handed a floating-point exception/ integer overflow signal. This error is usually fatal.

Maskable: No.

System Call Exception

Cause: A System Call exception occurs during an attempt to execute the SYSCALL instruction.

Processing: The common exception vector is used for this exception, and the *Sys* code in the *Cause* register is set. The *EPC* register contains the address of the SYSCALL instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction. If the SYSCALL instruction is in a branch delay slot, the *BD* bit of the *Status* register is set; otherwise this bit is cleared. System Call exception processing is shown in Figure 6.15 on page 6-11.

Servicing: When this exception occurs, control is transferred to the applicable system routine. To resume execution, the EPC register must be altered so that the SYSCALL instruction does not re-execute; this is accomplished by adding a value of 4 to the EPC register (EPC register + 4) before returning. If a SYSCALL instruction is in a branch delay slot, a more complicated algorithm, beyond the scope of this description, may be required.

Maskable: No.

Breakpoint Exception

Cause: A Breakpoint exception occurs when an attempt is made to execute the BREAK instruction.

Processing: The common exception vector is used for this exception, and the *BP* code in the *Cause* register is set. The *EPC* register contains the address of the BREAK instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction. If the BREAK instruction is in a branch delay slot, the *BD* bit of the *Status* register is set, otherwise the bit is cleared. Breakpoint exception processing is shown in Figure 6.15 on page 6-11.

Servicing: When the Breakpoint exception occurs, control is transferred to the applicable system routine. Additional distinctions can be made by analyzing the unused bits of the BREAK instruction (bits 25:6), and loading the contents of the instruction whose address the EPC register contains. A value of 4 must be added to the contents of the EPC register (EPC register + 4) to locate the instruction if it resides in a branch delay slot.

To resume execution, the *EPC* register must be altered so that the BREAK instruction does not re-execute; this is accomplished by adding a value of 4 to the *EPC* register (*EPC* register + 4) before returning. If a BREAK instruction is in a branch delay slot, interpretation of the branch instruction is required to resume execution.

Maskable: No.

Reserved Instruction Exception

Cause: The Reserved Instruction exception occurs when one of the following conditions occurs:

- *an attempt is made to execute an instruction with an undefined major opcode (bits 31:26)*
- *an attempt is made to execute a SPECIAL instruction with an undefined minor opcode (bits 5:0)*
- *an attempt is made to execute a REGIMM instruction with an undefined minor opcode (bits 20:16)*

Notes

- *an attempt is made to execute 64-bit operations in 32-bit virtual addressing when in User or Supervisor modes*

64-bit operations are always valid in Kernel mode regardless of the value of the *KX* bit in the *Status* register. Reserved Instruction exception processing is shown in Figure 6.15 on page 6-11.

Processing: The common exception vector is used for this exception, and the *RI* code in the *Cause* register is set. The *EPC* register contains the address of the reserved instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

Servicing: No instructions in the MIPS ISA are currently interpreted. The process executing at the time of this exception is handed an illegal instruction/reserved operand fault signal. This error is usually fatal.

Maskable: No.

Coprocessor Unusable Exception

Cause: The Coprocessor Unusable exception occurs when an attempt is made to execute a coprocessor instruction for either:

- *a corresponding coprocessor unit that has not been marked usable*
- *CP0 instructions, when the unit has not been marked usable and the process executes in User mode.*

Processing: The common exception vector is used for this exception, and the *CPU* code in the *Cause* register is set. The contents of the *Coprocessor Usage Error* field of the coprocessor *Control* register indicate which of the four coprocessors was referenced. The *EPC* register contains the address of the unusable coprocessor instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction. Coprocessor Unusable exception processing is shown in Figure 6.15 on page 6-11.

Servicing: The coprocessor unit to which an attempted reference was made is identified by the Coprocessor Usage Error field, which results in one of the following situations:

- *If the process is entitled access to the coprocessor, the coprocessor is marked usable and the corresponding user state is restored to the coprocessor.*
- *If the process is entitled access to the coprocessor, but the coprocessor does not exist or has failed, interpretation of the coprocessor instruction is possible.*
- *If the BD bit is set in the Cause register, the branch instruction must be interpreted; then the coprocessor instruction can be emulated and execution resumed with the EPC register advanced past the coprocessor instruction.*
- *If the process is not entitled access to the coprocessor, the process executing at the time is handed an illegal instruction/privileged instruction fault signal. This error is usually fatal.*

Maskable: No.

Floating-Point Exception

Cause: The Floating-Point exception is used by the floating-point coprocessor.

Processing: The common exception vector is used for this exception, and the *FPE* code in the *Cause* register is set. The contents of the *Floating-Point Control/Status* register indicate the cause of this exception. Floating-Point exception processing is shown in Figure 6.15 on page 6-11.

Servicing: This exception is cleared by clearing the appropriate bit in the Floating-Point Control/Status register. For an unimplemented instruction exception, the kernel should emulate the instruction; for other exceptions, the kernel should pass the exception to the user program that caused the exception.

Maskable: No.

Interrupt Exception

Cause: The Interrupt exception occurs when one of the eight interrupt conditions is asserted. The significance of these interrupts is dependent upon the specific system implementation.

Notes

Processing: The common exception vector is used for this exception, and the *Int* code in the *Cause* register is set. The *IP* field of the *Cause* register indicates current interrupt requests. It is possible that more than one of the bits can be simultaneously set (or even *no* bits may be set if the interrupt is asserted and then deasserted before this register is read). Interrupt exception processing is shown in Figure 6.15 on page 6-11.

Servicing: If the interrupt is caused by one of the two software-generated exceptions (SW1 or SW0), the interrupt condition is cleared by setting the corresponding Cause register bit to 0. If the interrupt is hardware-generated, the interrupt condition is cleared by correcting the condition causing the interrupt pin to be asserted.

Maskble: Each of the eight interrupts can be masked by clearing the corresponding bit in the *Int-Mask* field of the *Status* register, and all of the eight interrupts can be masked at once by clearing the *IE* bit of the *Status* register.

> **Note:** Due to the write buffer, a store to an external device will not necessarily occur until after other instructions in the pipeline finish. As such, the user must ensure that the store will occur before the return from exception instruction (ERET) is executed otherwise the interrupt may be serviced again even though there should be no interrupt pending.

Handling/Servicing Flowcharts

The remainder of this chapter contains the flowcharts for the exceptions described in Table 6.12, and guidelines for their handlers.

Figure	Description
Figure 6.16, Figure 6.17	General exceptions and their exception handler
Figure 6.18, Figure 6.19	TLB/XTLB miss exception and their exception handler
Figure 6.20	Cache error exception and its handler
Figure 6.21	Reset, soft reset and NMI exceptions, and a guideline to their handler.

Table 6.12 List of Exception Flowcharts

Generally speaking, the exceptions are handled by hardware (HW), and then the exceptions are serviced by software (SW).

Notes

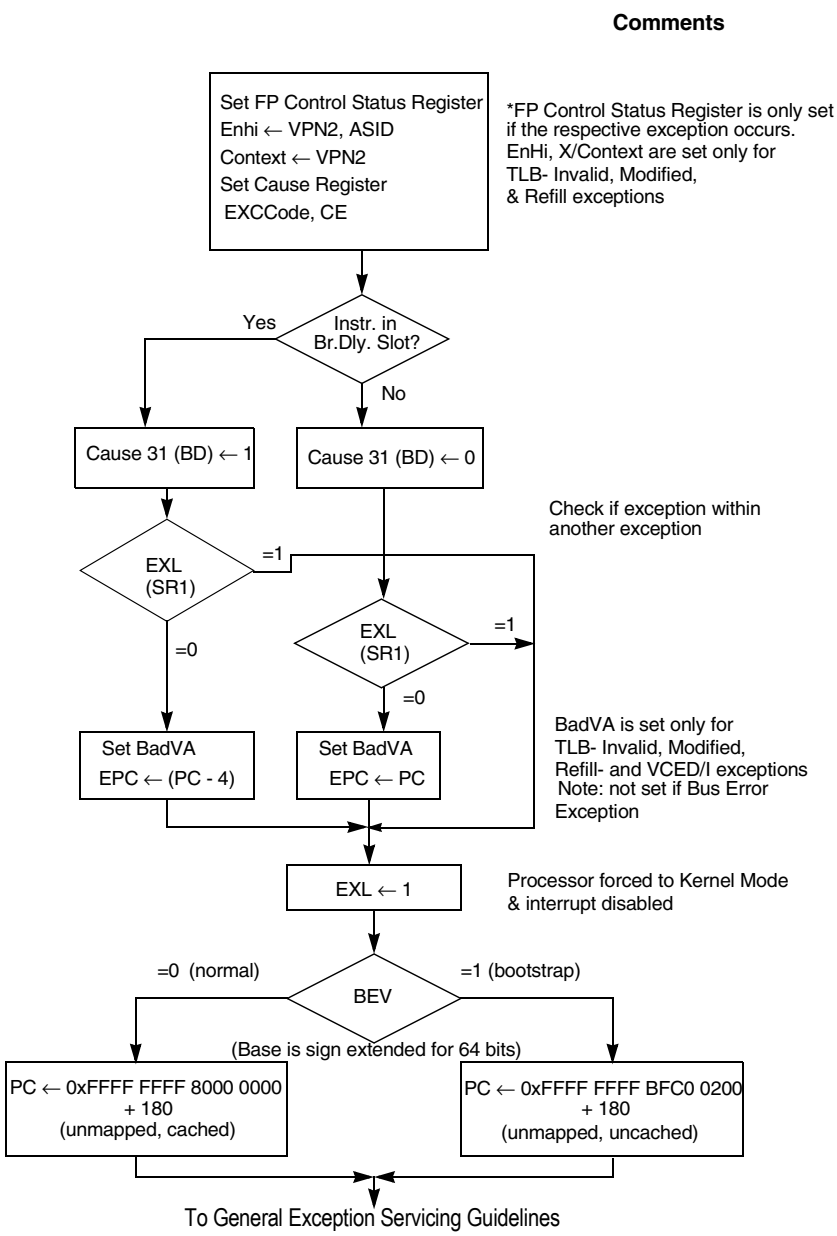


Figure 6.16 General Exception Handler (HW)

Notes

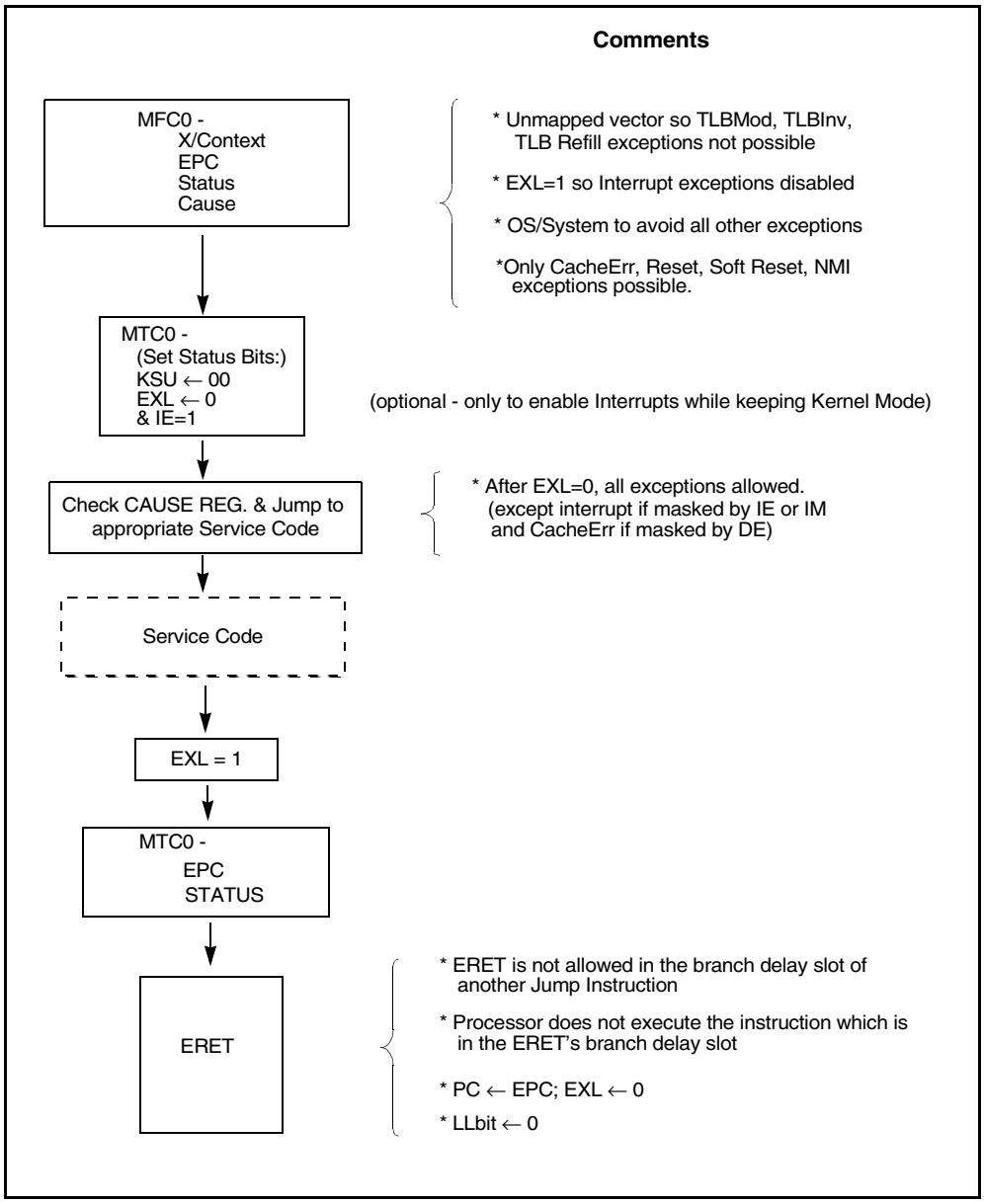


Figure 6.17 General Exception Servicing Guidelines (SW)

Notes

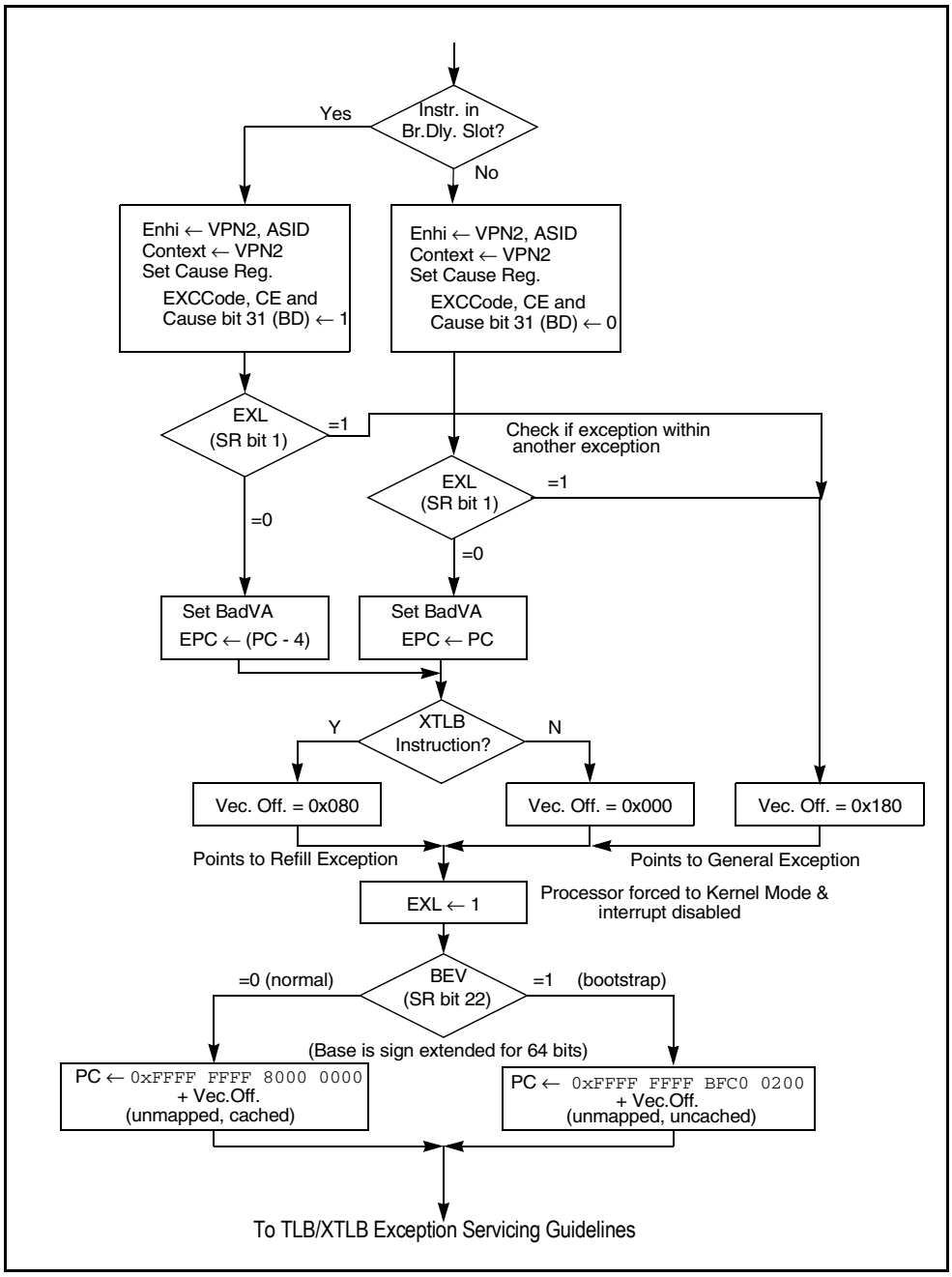


Figure 6.18 TLB/XTLB Miss Exception Handler (HW)

Notes

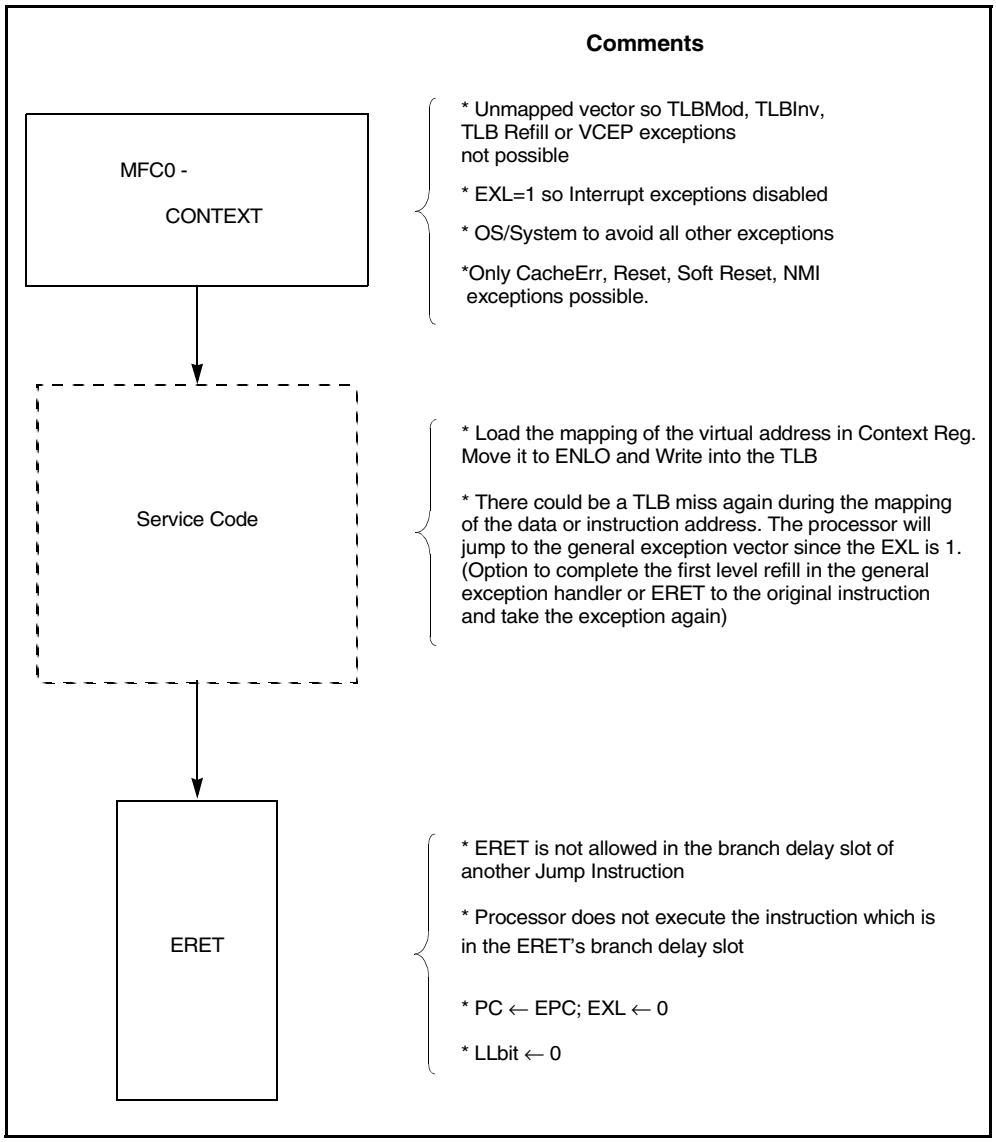


Figure 6.19 TLB/XTLB Exception Servicing Guidelines (SW)

Notes

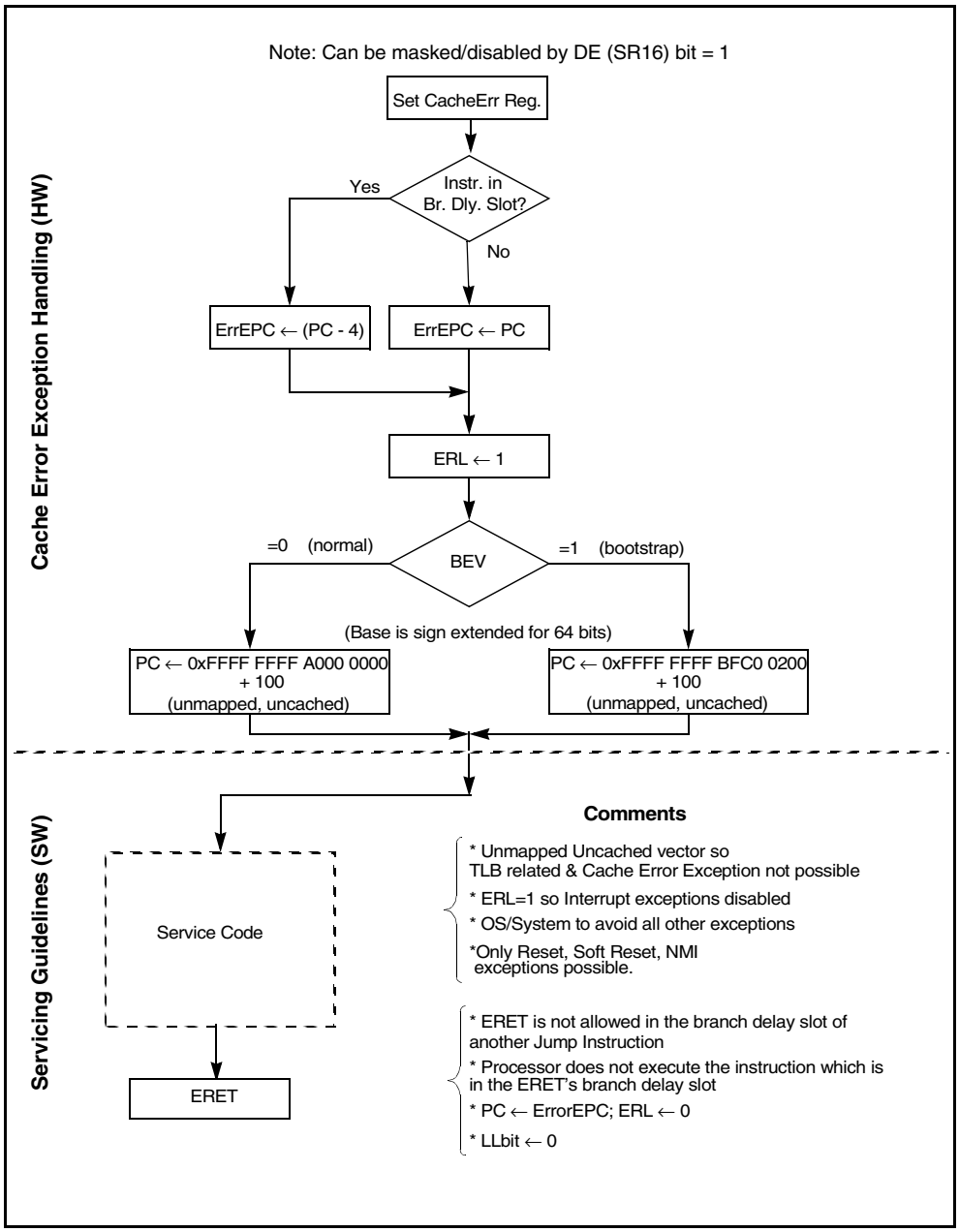


Figure 6.20 Cache Error Exception Handling (HW) and Servicing Guidelines (SW)

Notes

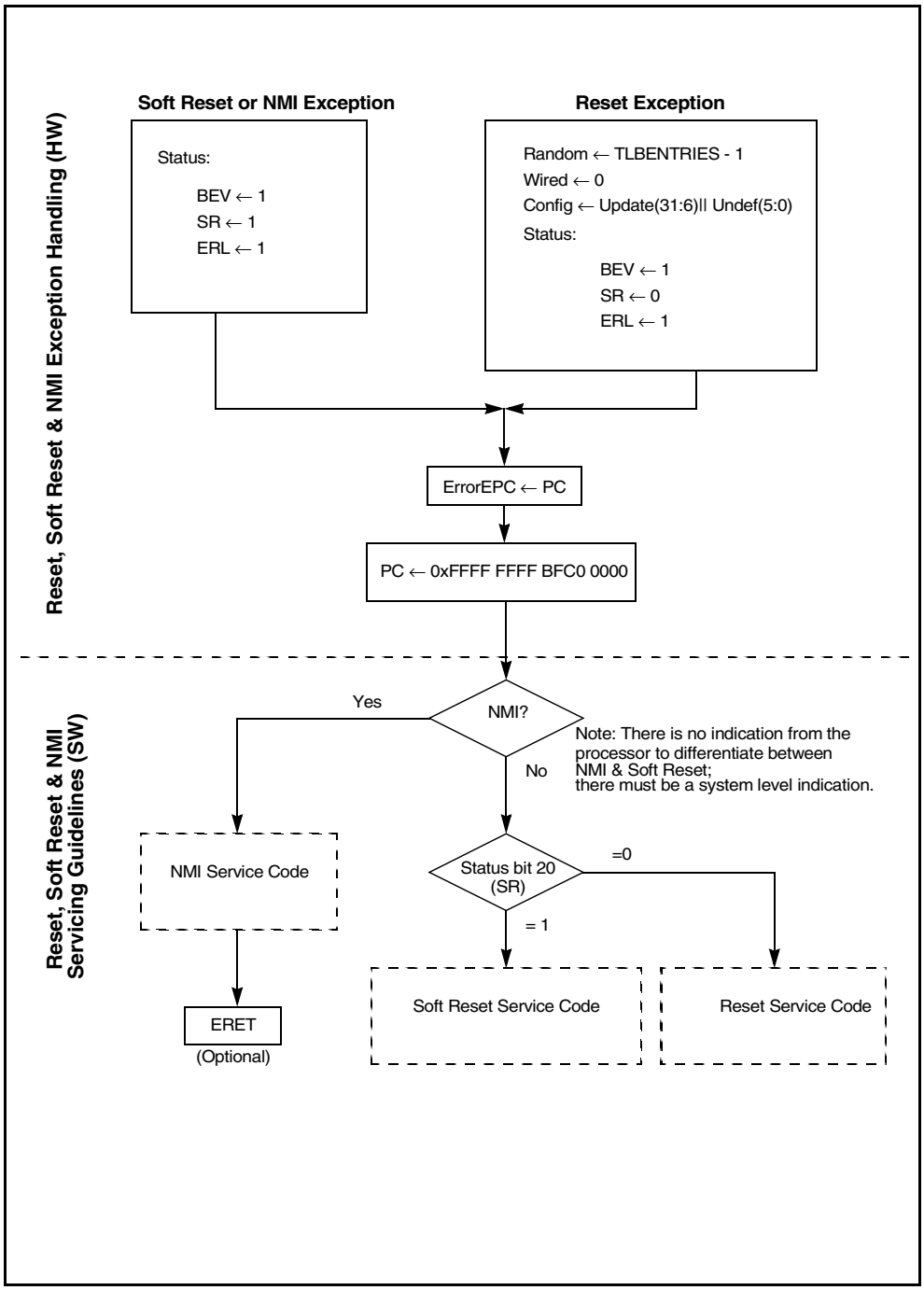


Figure 6.21 Reset, Soft Reset & NMI Exception Handling (HW) and Servicing Guidelines (SW)

Notes

The Floating-Point Unit

Notes

Introduction

This chapter describes the RISCore4000 floating-point unit (FPU) features and includes a programming model, instruction set and formats, and information on the pipeline. The FPU—with associated system software—fully conforms to the requirements of ANSI/IEEE Standard 754–1985, IEEE Standard for Binary Floating-Point Arithmetic. In addition, the MIPS architecture fully supports the recommendations of the standard and precise exceptions. The FPU operates as a coprocessor for the CPU and is assigned coprocessor label CP1. The FPU extends the CPU instruction set to perform arithmetic operations on floating-point values.

Floating-Point Coprocessor (CP1)

The RISCore4000's floating-point coprocessor has improved floating multiply operations and incorporates an entire floating-point coprocessor on chip, including a floating-point register file and execution units. The floating-point coprocessor forms a seamless interface with the integer unit, decoding and executing instructions in parallel with the integer unit.

The RISCore4000 uses the floating-point unit to perform integer multiply and divide, results are placed in the HI and LO registers. The values can then be transferred to the general purpose register file using the MFHI/MFLO instructions. The RISCore4000 performs a single-precision multiply in 4 clock cycles and a double-precision multiply in 5 clock cycles. Figure 7.1 illustrates the functional organization of the FPU.

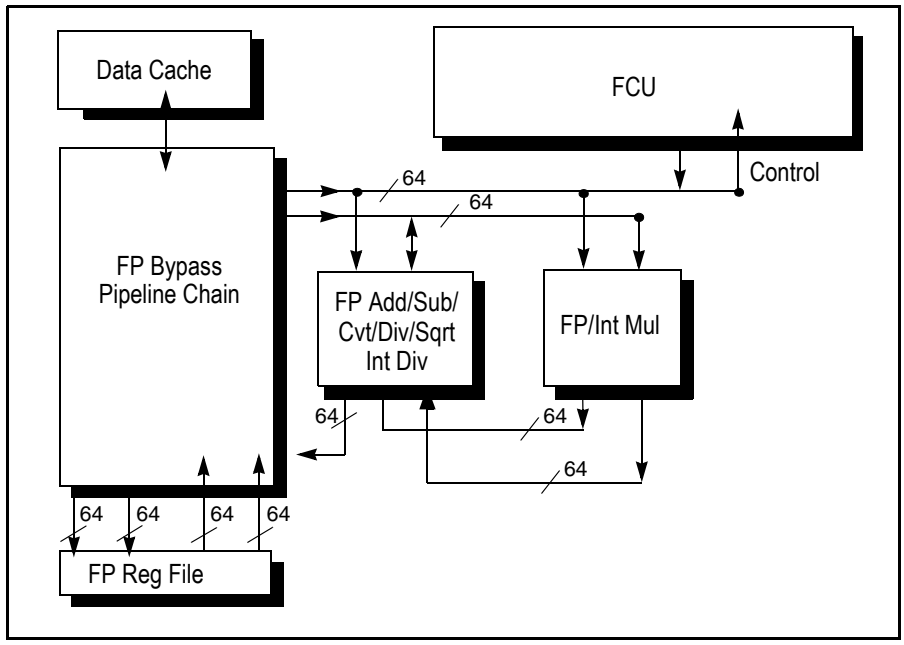


Figure 7.1 FPU Functional Block Diagram

Features

This section briefly describes the operating model, the load/store instruction set, and the coprocessor interface in the FPU. A more detailed description follows.

- ***Full 64-bit Operation****. When the FR bit in the CPU Status register equals 0, the FPU is configured for sixteen 64-bit registers for double-precision values or thirty-two 32-bit registers for single-precision values. When the FR bit in the CPU Status register equals 1, the FPU is configured for thirty-two 64-bit registers. Each register can hold single- or double-precision values. The FPU also*

Notes

includes a 32-bit Control/Status register that provides access to all IEEE-Standard exception handling capabilities.

- ***Load and Store Instruction Set**. Like the CPU, the FPU uses a load- and store-oriented instruction set, with single-cycle load and store operations. Overlap of multiply and add is supported.*
- ***Tightly Coupled Coprocessor Interface**. The FPU resides on-chip to form a tightly coupled unit with a seamless integration of floating-point and fixed-point instruction sets.*

General Registers (FGRs)

The FPU provides:

- *16 Floating-Point registers (FPRs) for Status.FR = 0 or*
- *32 Floating-Point registers (FPRs) for Status.FR = 1.*

These 64-bit registers hold floating-point values during floating-point operations and are physically formed from the *General Purpose* registers (*FGRs*). When the *FR* bit in the *Status* register equals 1, the *FPR* references a single 64-bit *FGR.* The *FPR*s hold values in either single- or double-precision floating-point format. If the *FR* bit equals 0, only even numbers (the *least* register, as shown in Figure 7.2) can be used to address *FPR*s. When the *FR* bit is set to a 1, all *FPR* register numbers are valid.

If the *FR* bit equals 0 during a double-precision floating-point operation, the general registers are accessed in double pairs. Thus, in a double-precision operation, selecting *Floating-Point Register 0* (*FPR0*) actually addresses adjacent *Floating-Point General Purpose* registers *FGR0* and *FGR1*.

The *Floating-Point General Purpose* registers *(FGR*s) can be accessed in the following ways:

- *As 32 general-purpose registers (32 FGRs), each of which is 32-bits wide when the FR bit in the CPU Status register equals 0; or as 32 general-purpose registers (32 FGRs), each of which is 64-bits wide when FR equals 1. The CPU accesses these registers through move, load, and store instructions.*
- *As 16 floating-point registers (see the next section for a description of FPRs), each of which is 64-bits wide, when the FR bit in the CPU Status register equals 0. The FPRs hold values in either single- or double-precision floating-point format. Each FPR corresponds to adjacently numbered FGRs as shown in Figure 7.2 on page 7-2.*
- *As 32 floating-point registers (see the next section for a description of FPRs), each of which is 64-bits wide, when the FR bit in the CPU Status register equals 1. The FPRs hold values in either single- or double-precision floating-point format.*

Each FPR corresponds to an FGR, as shown in Figure 7.2.

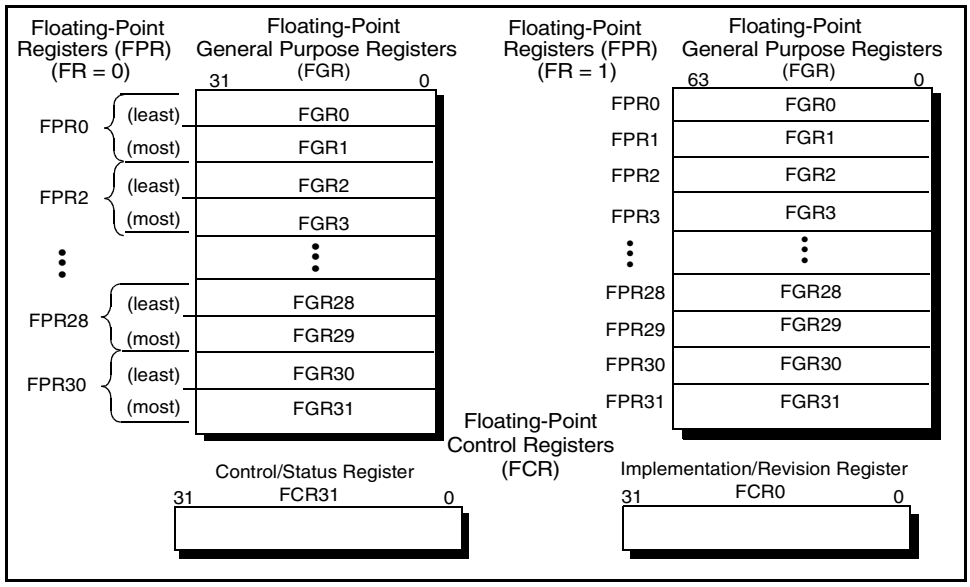


Figure 7.2 Floating-Point Unit (FPU) Registers

Notes

Control Registers (FCRs)

The FPU has 32 control registers (*FCR*s) that can only be accessed by move operations. The *FCR*s are described below:

- *The Implementation/Revision register (FCR0) holds revision information about the FPU.*
- *The Control/Status register (FCR31) controls and monitors exceptions, holds the result of compare operations, and establishes rounding modes.*
- *FCR1 to FCR30 are reserved.*

Table 7.1 lists the assignments of the *FCR*s.

FCR Number	Use
FCR0	Coprocessor implementation and revision register
FCR1 to FCR30	Reserved
FCR31	Rounding mode, cause, trap enables, and flags

Table 7.1 Floating-Point Control Register Assignments

Implementation and Revision Register, (FCR0)

The read-only *Implementation and Revision* register (*FCR0*) specifies the implementation and revision number of the FPU. This information can determine the coprocessor revision and performance level, and can also be used by diagnostic software. Figure 7.3 shows the layout of the register; Table 7.2, which follows the figure, describes the *Implementation and Revision* register (*FCR0*) fields.

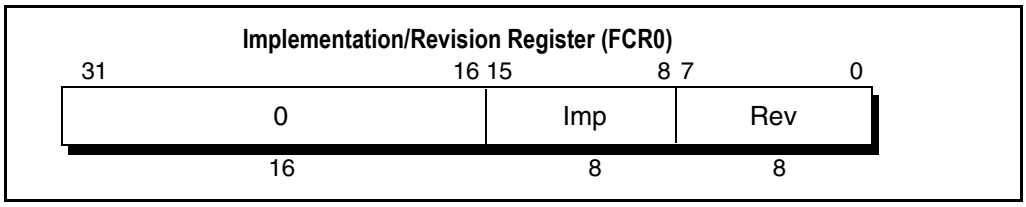


Figure 7.3 Implementation/Revision Register

Field	Description
Imp	Implementation number: 0x21
Rev	Revision number in the form of *y.x*
0	Reserved.

Table 7.2 FCR0 Register Fields

The revision number is a value of the form *y.x*, where:

- *y is a major revision number held in bits 7:4.*
- *x is a minor revision number held in bits 3:0.*

The revision number distinguishes some chip revisions; however, there is no guarantee that changes to the chip are necessarily reflected by the revision number, or that changes to the revision number necessarily reflect real chip changes. For this reason revision number values are not listed, and software should not rely on the revision number to characterize the chip.

Control/Status Register (FCR31)

The *Control/Status* register *(FCR31*) contains control and status information that can be accessed by instructions in either Kernel or User mode. *FCR31* also controls the arithmetic rounding mode and enables User mode traps, as well as identifying any exceptions that may have occurred in the most recently executed instruction, along with any exceptions that may have occurred without being trapped. Figure 7.4

Notes

on page 7-4 shows the format of the *Control/Status* register, and Table 7.3, which follows the figure, describes the *Control/Status* register fields. Figure 7.5 on page 7-4 shows the *Control/Status* register *Cause, Flag,* and *Enable* fields.

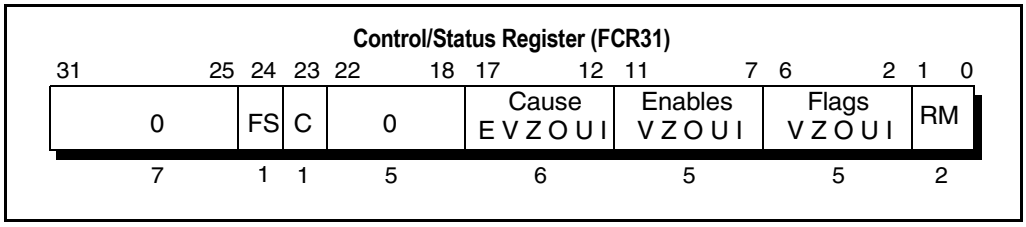


Figure 7.4 FP Control/Status Register Bit Assignments

Field	Description
FS	When set, denormalized results are flushed to 0 instead of causing an unimplemented operation exception.
C	**Condition bit.** See description of *Control/Status* register *Condition* bit.
Cause	**Cause bits**. See Figure 7.5 and the description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
Enables	**Enable bits**. See Figure 7.5 and the description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
Flags	**Flag bits**. See Figure 7.5 and the description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
RM	**Rounding mode bits**. See Table 7.4 on page 7-6 and the description of *Control/Status* register *Rounding Mode Control* bits.

Table 7.3 Control/Status Register Fields

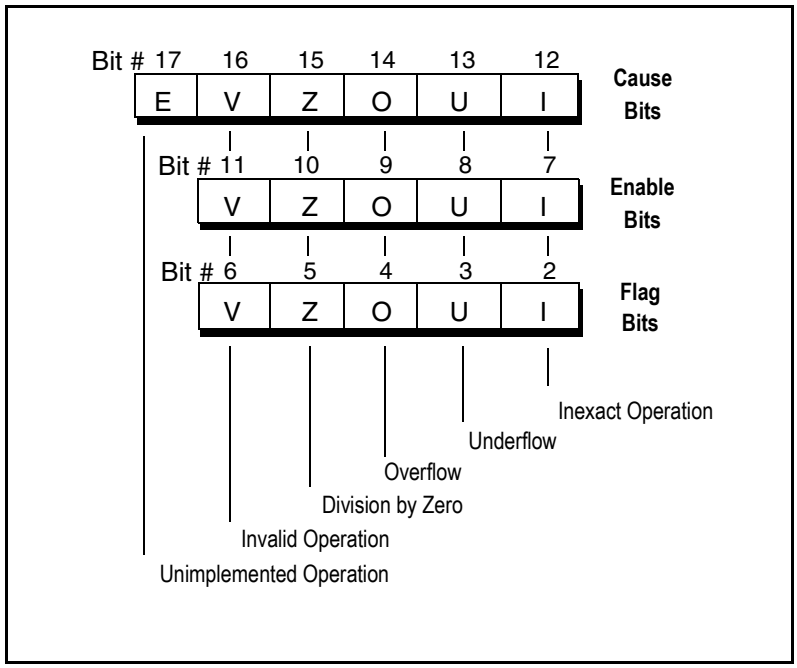


Figure 7.5 Control/Status Register Cause, Flag, and Enable Fields

When the *Control/Status* register is read by a Move Control From Coprocessor 1 (CFC1) instruction, all unfinished instructions in the pipeline are completed before the contents of the register are moved to the main processor. If a floating-point exception occurs as the pipeline empties, the FP exception is taken and the CFC1 instruction is re-executed after the exception is serviced. The bits in the *Control/Status* register can be set or cleared by writing to the register using a Move Control To Coprocessor 1 (CTC1) instruction. CTC1 is not issued until all previous floating-point operations are complete.

Notes

IEEE Standard 754

IEEE Standard 754 specifies that floating-point operations detect certain exceptional cases, raise flags, and can invoke an exception handler when an exception occurs. These features are implemented in the MIPS architecture with the *Cause*, *Enable*, and *Flag* fields of the *Control/Status* register. The *Flag* bits implement IEEE 754 exception status flags, and the *Cause* and *Enable* bits implement exception handling.

Control/Status register FS bit: When the *FS* bit is set, denormalized results are flushed to 0 instead of causing an unimplemented operation exception.

Control/Status Register Condition Bit: When a floating-point Compare operation takes place, the result is stored at bit 23, the *Condition* bit, to save or restore the state of the condition line. The *C* bit is set to 1 if the condition is true; the bit is cleared to 0 if the condition is false. Bit 23 is affected only by compare and Move Control To FPU instructions.

Control/Status Register Cause, Flag, and Enable Fields: Figure 7.5 on page 7-4 illustrates the *Cause*, *Flag*, and *Enable* fields of the *Control/Status* register.

Cause Bits: Bits 17:12 in the *Control/Status* register contain *Cause* bits, as shown in Figure 7.5 on page 7-4, which reflect the results of the most recently executed instruction. The *Cause* bits are a logical extension of the CP0 *Cause* register; they identify the exceptions raised by the last floating-point operation and raise an interrupt or exception if the corresponding enable bit is set. If more than one exception occurs on a single instruction, each appropriate bit is set.

The *Cause* bits are written by each floating-point operation (but not by load, store, or move operations). The Unimplemented Operation (*E*) bit is set to a 1 if software emulation is required, otherwise it remains 0. The other bits are set to 0 or 1 to indicate the occurrence or non-occurrence (respectively) of an IEEE 754 exception. When a floating-point exception is taken, no results are stored, and the only state affected is the *Cause* bits. Exceptions caused by an immediately previous floating-point operation can be determined by reading the *Cause* field.

Enable Bits: A floating-point operation that sets an enabled *Cause* bit forces an immediate exception, as does setting both *Cause* and *Enable* bits with CTC1. The floating-point exception or interrupt is enabled when the corresponding enable be is set. There is no enable for Unimplemented Operation (*E*). Setting Unimplemented Operation always generates a floating-point exception.

Before returning from a floating-point exception, or doing a CTC1, software must first clear the enabled *Cause* bits to prevent a repeat of the interrupt. Thus, User mode programs can never observe enabled *Cause* bits set; if this information is required in a User mode handler, it must be passed somewhere other than the *Status* register. For a floating-point operation that sets only unenabled *Cause* bits, no exception occurs and the default result defined by IEEE 754 is stored. In this case, the exceptions that were caused by the immediately previous floating-point operation can be determined by reading the *Cause* field.

Flag Bits: When an exception case is detected and the exception Enable is not set, the corresponding flag bit is set. If an exception is taken, none of the flag bits are modified. Note however that system software may set the flag bits before invoking a user exception handler.

The *Flag* bits are cumulative and indicate that an exception was raised by an operation that was executed since they were explicitly reset. *Flag* bits are set to 1 if an IEEE 754 exception is raised, otherwise they remain unchanged. The *Flag* bits are never cleared as a side effect of floating-point operations; however, they can be set or cleared by writing a new value into the *Status* register, using a Move To Coprocessor Control instruction.

Control/Status Register Rounding Mode Control Bits: Bits 1 and 0 in the *Control/Status* register constitute the *Rounding Mode* (*RM*) field. As shown in Table 7.4, these bits specify the rounding mode that the FPU uses for all floating-point operations.

Notes

Rounding Mode RM(1:0)	Mnemonic	Description
0	RN	Round result to nearest representable value; round to value with least-significant bit 0 when the two nearest representable values are equally near.
1	RZ	Round toward 0: round to value closest to and not greater in magnitude than the infinitely precise result.
2	RP	Round toward +•: round to value closest to and not less than the infinitely precise result.
3	RM	Round toward – •: round to value closest to and not greater than the infinitely precise result.

Table 7.4 Rounding Mode Bit Decoding

Floating-Point Formats

The FPU performs both 32-bit (single-precision) and 64-bit (double-precision) IEEE standard floating-point operations. The 32-bit single-precision format has a 24-bit signed-magnitude fraction field (*f+s*) and an 8-bit exponent (*e*), as shown in Figure 7.6.

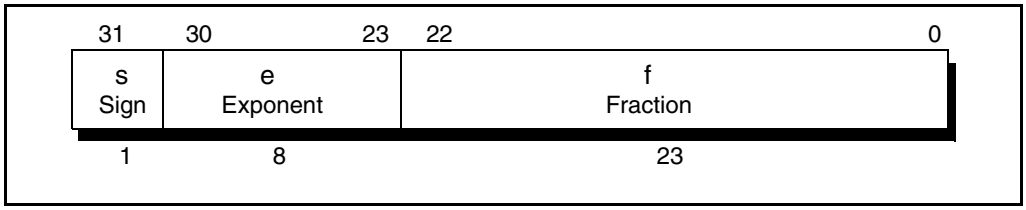


Figure 7.6 Single-Precision Floating-Point Format

The 64-bit double-precision format has a 53-bit signed-magnitude fraction field (*f+s)* and an 11-bit exponent, as shown in Figure 7.7.

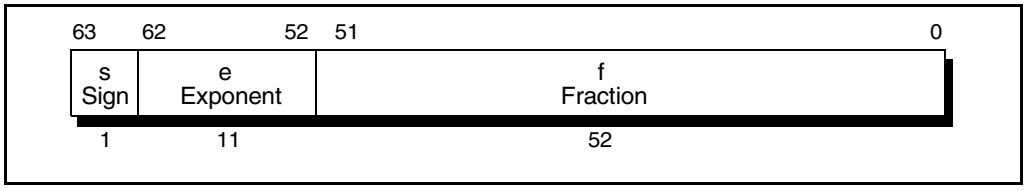


Figure 7.7 Double-Precision Floating-Point Format

As shown in the above figures, numbers in floating-point format are composed of three fields:

- *sign field, s*
- *biased exponent, e = E + bias*
- *fraction, f =.$b_1b_2....b_{p-1}$*

The range of the unbiased exponent *E* includes every integer between the two values E_{min} and E_{max} inclusive, together with two other reserved values:

- *E_{min} -1 (to encode ±0 and denormalized numbers)*
- *E_{max} +1 (to encode ±• and NaNs [Not a Number])*

For single- and double-precision formats, each representable nonzero numerical value has just one encoding. For single- and double-precision formats, the value of a number, *v*, is determined by the equations shown in .

Notes

No.	Equation
(1)	if E = E_{max}+1 and *f* ¼ 0, then *v* is NaN, regardless of *s*
(2)	if E = E_{max}+1 and f = 0, then *v* = $(-1)^s$ •
(3)	if E_{min} £ E £ E_{max}, then *v* = $(-1)^s 2^E (1.f)$
(4)	if E = E_{min}–1 and f ¼ 0, then *v* = $(-1)^s 2^{Emin} (0.f)$
(5)	if E = E_{min}–1 and f = 0, then *v* = $(-1)^s 0$

Table 7.5 Equations for Calculating Values in Single and Double-Precision Floating-Point Format

For all floating-point formats, if *v* is NaN, the most-significant bit of *f* determines whether the value is a signaling or quiet NaN: *v* is a signaling NaN if the most-significant bit of *f* is set, otherwise, *v* is a quiet NaN. Table 7.7 defines the values for the format parameters.

Minimum and maximum floating-point values are given in Table 7.7.

Parameter	Format	
	Single	Double
f	24	53
E_{max}	+127	+1023
E_{min}	–126	–1022
Exponent *bias*	+127	+1023
Exponent width in bits	8	11
Integer bit	hidden	hidden
Fraction width in bits	24	53
Format width in bits	32	64

Table 7.6 Floating-Point Format Parameter Values

Type	Value
Float Minimum	1.40129846e–45
Float Minimum Norm	1.17549435e–38
Float Maximum	3.40282347e+38
Double Minimum	4.9406564584124654e–324
Double Minimum Norm	2.2250738585072014e–308
Double Maximum	1.7976931348623157e+308

Table 7.7 Minimum and Maximum Floating-Point Values

Binary Fixed-Point Format

Binary fixed-point values are held in 2's complement format. Unsigned fixed-point values are not directly provided by the floating-point instruction set. Figure 7.8 illustrates binary fixed-point format; Table 7.8, which follows the figure, lists the binary fixed-point format fields.

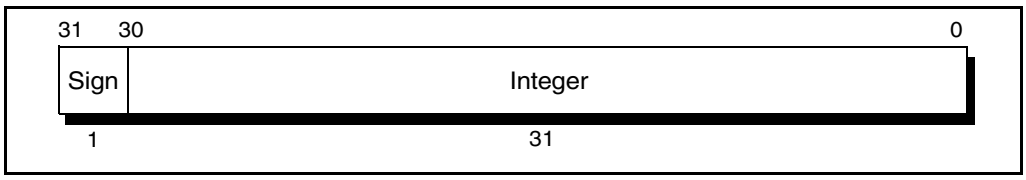


Figure 7.8 Binary Fixed-Point Format

Notes

Field	Description
sign	sign bit
integer	integer value

Table 7.8 Binary Fixed-Point Format Fields

FPU Instruction Set Overview

All FPU instructions are 32-bits long, aligned on a word boundary and can be divided into the following groups:

- ***Load, Store, and Move*** *instructions move data between memory, the main processor, and the FPU General Purpose registers.*
- ***Conversion*** *instructions perform conversion operations between the various data formats.*
- ***Computational*** *instructions perform arithmetic operations on floating-point values in the FPU registers.*
- ***Compare*** *instructions perform comparisons of the contents of registers and set a conditional bit based on the results.*
- ***Branch on FPU Condition*** *instructions perform a branch to the specified target if the specified coprocessor condition is met.*

Table 7.12 lists the instruction set of the FPU. A complete description of each instruction is provided in the *IDT MIPS Microprocessor Family Software Reference Manual*. In the instruction formats shown in Table 7.12, the *fmt* appended to the instruction opcode specifies the data format: *s* specifies single-precision binary floating-point, *d* specifies double-precision binary floating-point, and *w* specifies binary fixed-point.

OpCode	Description
LWC1	Load Word to FPU
SWC1	Store Word from FPU
LDC1	Load Doubleword to FPU
SDC1	Store Doubleword From FPU
MTC1	Move Word To FPU
MFC1	Move Word From FPU
CTC1	Move Control Word To FPU
CFC1	Move Control Word From FPU
DMTC1	Doubleword Move To FPU
DMFC1	Doubleword Move From FPU

Table 7.9 FPU Instruction Summary: Load, Move and Store Instructions

OpCode	Description
CVT.S.fmt	Floating-point Convert to Single FP
CVT.D.fmt	Floating-point Convert to Double FP
CVT.W.fmt	Floating-point Convert to Single Fixed Point
ROUND.w.fmt	Floating-point Round
TRUNC.w.fmt	Floating-point Truncate
CEIL.w.fmt	Floating-point Ceiling
FLOOR.w.fmt	Floating-point Floor

Table 7.10 FPU Instruction Summary: Conversion Instructions

Notes

OpCode	Description
ADD.fmt	Floating-point Add
SUB.fmt	Floating-point Subtract
MUL.fmt	Floating-point Multiply
DIV.fmt	Floating-point Divide
ABS.fmt	Floating-point Absolute Value
MOV.fmt	Floating-point Move
NEG.fmt	Floating-point Negate
SQRT.fmt	Floating-point Square Root

Table 7.11 FPU Instruction Summary: Computational Instructions

OpCode	Description
C.cond.fmt	Floating-point Compare
BC1T	Branch on FPU True
BC1F	Branch on FPU False
BC1TL	Branch on FPU True Likely
BC1FL	Branch on FPU False Likely

Table 7.12 FPU Instruction Summary: Compare and Branch Instructions

Load, Store, and Move Instructions

This section discusses the manner in which the FPU uses the load, store and move instructions listed in Table 7.9 on page 7-8. Detailed descriptions of each instruction can be found in the *IDT MIPS Microprocessor Family Software Reference Manual.*

Transfers Between FPU and Memory

All data movement between the FPU and memory is accomplished by using one of the following instructions:

- *Load Word To Coprocessor 1 (LWC1) or Store Word To Coprocessor 1 (SWC1) instructions, which reference a single 32-bit word of the FPU general registers*
- *Load Doubleword (LDC1) or Store Doubleword (SDC1) instructions, which reference a 64-bit doubleword.*

These load and store operations are unformatted; no format conversions are performed and therefore no floating-point exceptions can occur due to these operations. With the LDC1 and SDC1 instructions the RISCore4000 floating-point unit can take advantage of the 64-bit wide data cache and issue a coprocessor load or store double-word instruction with every cycle.

Transfers Between FPU and CPU

Data can also be moved directly between the FPU and the CPU by using one of the following instructions:

- *Move To Coprocessor 1 (MTC1)*
- *Move From Coprocessor 1 (MFC1)*
- *Doubleword Move To Coprocessor 1 (DMTC1)*
- *Doubleword Move From Coprocessor 1 (DMFC1)*

Like the floating-point load and store operations, MTC1, MFC1, DMTC1 and DMFC1 operations perform no format conversions and do not cause floating-point exceptions.

Notes

Load Delay and Hardware Interlocks

The instruction immediately following a load can use the contents of the loaded register. In such cases the hardware interlocks, requiring additional real cycles; for this reason, scheduling load delay slots is desirable, although it is not required for functional code.

Data Alignment: All coprocessor loads and stores reference the following aligned data items:

- *For word loads and stores, the access type is always WORD, and the low-order 2 bits of the address must always be 0.*
- *For doubleword loads and stores, the access type is always DOUBLEWORD, and the low-order 3 bits of the address must always be 0.*

Endianness: Regardless of byte-numbering order (endianness) of the data, the address specifies the byte that has the smallest byte address in the addressed field. For a big-endian system, it is the leftmost byte; for a little-endian system, it is the rightmost byte.

Floating-Point Conversion Instructions

Conversion instructions perform conversions between the various data formats such as single- or double-precision and fixed- or floating-point formats. Table 7.10 on page 7-8 lists conversion instructions.

Floating-Point Computational Instructions

Computational instructions perform arithmetic operations on floating-point values, in registers. Table 7.11 on page 7-9 lists the computational instructions. As shown below, there are two categories of computational instructions:

- *3-Operand Register-Type instructions, which perform floating-point addition, subtraction, multiplication, division, and square root.*
- *2-Operand Register-Type instructions, which perform floating-point absolute value, move, and negate.*

Branch on FPU Condition Instructions

Table 7.12 on page 7-9 lists the Branch on FPU (coprocessor unit 1) condition instructions that can test the result of the FPU compare (C.cond) instructions. For more details, refer to the IDT

Floating-Point Compare Operations

The floating-point compare (C.fmt.cond) instructions interpret the contents of two FPU registers (*fs, ft*) in the specified format (*fmt*) and arithmetically compare them. A result is determined based on the comparison and conditions (*cond*) specified in the instruction. Table 7.12 on page 7-9 lists the compare instructions.

Table 7.13 on page 7-10 lists the mnemonics for the compare instruction conditions.

Mnemonic	Definition	Mnemonic	Definition
F	False	T	True
UN	Unordered	OR	Ordered
EQ	Equal	NEQ	Not Equal
UEQ	Unordered or Equal	OLG	Ordered or Less Than or Greater Than
OLT	Ordered Less Than	UGE	Unordered or Greater Than or Equal
ULT	Unordered or Less Than	OGE	Ordered Greater Than
OLE	Ordered Less Than or Equal	UGT	Unordered or Greater Than
ULE	Unordered or Less Than or Equal	OGT	Ordered Greater Than
SF	Signaling False	ST	Signaling True

Table 7.13 Mnemonics and Definitions of Compare Instruction Conditions (Part 1 of 2)

Notes

Mnemonic	Definition	Mnemonic	Definition
NGLE	Not Greater Than or Less Than or Equal	GLE	Greater Than, or Less Than or Equal
SEQ	Signaling Equal	SNE	Signaling Not Equal
NGL	Not Greater Than or Less Than	GL	Greater Than or Less Than
LT	Less Than	NLT	Not Less Than
NGE	Not Greater Than or Equal	GE	Greater Than or Equal
LE	Less Than or Equal	NLE	Not Less Than or Equal
NGT	Not Greater Than	GT	Greater Than

Table 7.13 Mnemonics and Definitions of Compare Instruction Conditions (Part 2 of 2)

Instruction Pipeline Overview

The RISCore4000 floating-point unit provides an instruction pipeline that parallels the CPU instruction pipeline and shares the same five-stage pipeline architecture (see Chapter 3).

However, unlike the CPU pipeline—which executes almost all instructions in a single cycle—more time may be required to execute FPU instructions. The FPU pipeline allows for the longer execution time requirements, which are listed in Table 7.14 on page 7-12. The 5-stage FPU pipeline is shown in Figure 7.9.

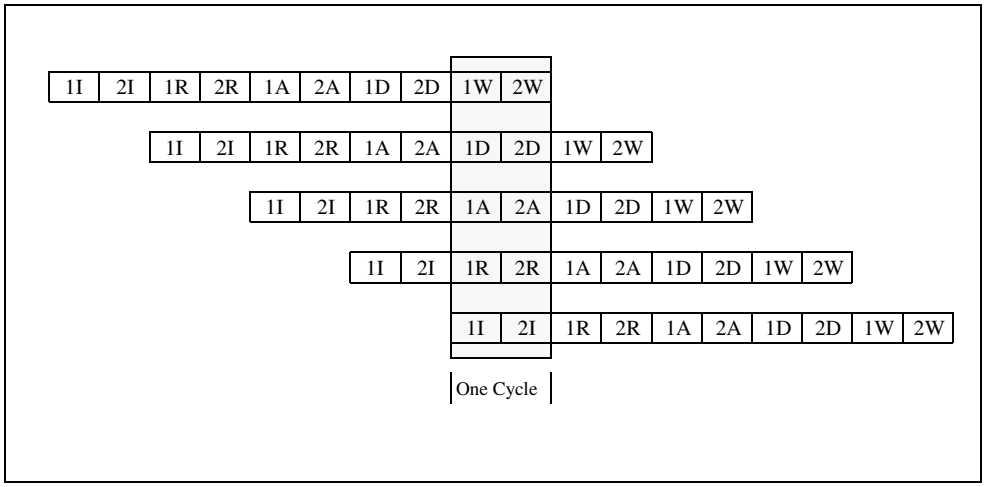


Figure 7.9 FPU Instruction Pipeline

Figure 7.9 on page 7-11 assumes that one instruction is completed every PCycle. Most FPU instructions, however, require more than one cycle in the EX stage. This means the FPU must stall the pipeline if an instruction execution cannot proceed because of register or resource conflicts. Floating-point operations proceed in parallel with non-floating-point operations and are not allowed to overlap each other, with the following two exceptions:

- *An add operation may start 2 cycles after the start of a multiply and thus will be completely overlapped by the multiply.*
- *A new multiply may start 4 cycles after another multiply (for both single and double precision).*

Non-floating-point operations, as well as other integer operations, may be executed in parallel with the floating-point operations. All of this is handled automatically by the RISCore4000's internal hardware.

Notes

Operation	Pipeline Cycles		Operation	Pipeline Cycles	
	Single	Double		Single	Double
ADD.fmt	4	4	**BC1T**	1	
SUB.fmt	4	4	**BC1F**	1	
MUL.fmt	4	5	**BC1TL**	1	
DIV.fmt	32	61	**BC1FL**	1	
SQRT.fmt	31	60	**LWC1, LDC1**	2	
ABS.fmt	1	1	**SWC1, SDC1**	1	
MOV.fmt	1	1	**TRUNC.W.fmt**	4	4
NEG.fmt	1	1	**MTC1, DMTC1**	2	
ROUND.W.fmt	4	4	**MFC1, DMFC1**	2	
CEIL.W.fmt	4	4	**CTC1**	3	
FLOOR.W.fmt	4	4	**CFC1**	2	
CVT.S.fmt	[1]	4	CMP	3	3
CVT.D.fmt	2	[1]	FIX	4	4
CVT.W.fmt	4	4	FLOAT	6	6
C.fmt.cond	3	3			

Table 7.14 Floating-Point Operation Latencies

1. These operations are illegal.

The **FPU resource scheduler** only issues instructions to the FPU op units (adder and multiplier) when no hardware use conflicts will occur. In addition, some overlap possibilities are disallowed to keep the scheduler simple (and/or increase performance). The **FPU multiplier constraints** are more fully pipelined in the RISCore4000, allowing a new multiply to begin every 4 cycles. The FPU scheduler may issue an add operation (ADD.fmt or SUB.fmt) 2 cycles after a multiply (MUL.fmt).

Resource Scheduling Rules

The FPU Resource Scheduler issues instructions, while adhering to the rules described below. These scheduling rules optimize op unit executions; if the rules are not followed, the hardware interlocks to guarantee correct operation.

DIV.[S,D] can start only when all of the following conditions are met in the 1A phase.

- *The adder is idle (division is performed in the adder).*
- *The multiplier is idle.*

MUL.[S,D] can start only when all of the following conditions are met in the 1A phase.

- *The multiplier is one of the following:*
  - *idle.*
  - *Started execution at least 6 cycles earlier on the current multiply*
- *The adder is idle.*

SQRT.[S,D] can start when the following conditions are met in the 1A phase.

- *The adder is idle.*
- *The multiplier must be idle.*

CVT.fmt instructions can only start when all of the following conditions are met in the 1A phase.

- *The adder is idle.*
- *The multiplier is idle.*

Notes

ADD.[S,D] or **SUB.[S,D]** can start only when all of the following conditions are met in the 1A phase.

- *The adder is idle*
- *The multiplier is either:*
  - *idle.*
  - *started execution of the current multiply at least 2 cycles earlier.*

NEG.[S,D] or **ABS.[S,D]** can start only when all of the following conditions are met in the 1A phase.

- *The adder is idle.*
- *The multiplier is idle.*

C.COND.[S,D] can start only when all of the following conditions are met in the 1A phase.

- *The adder is idle.*
- *The multiplier is idle.*

Notes

Floating-Point Exceptions

Notes

Introduction

Whenever the FPU is unable to handle either the operands or the results of a floating-point operation through normal methods, a floating-point exception occurs. The FPU then responds by generating an exception to initiate a software trap or by setting a status flag. These exceptions—including FPU exception types, exception trap processing, exception flags, saving and restoring state when handling an exception, and trap handlers for IEEE Standard 754 exceptions—are discussed in this chapter.

Exception Types

The FP *Control/Status* register described contains an *Enable* bit for each exception type; exception *Enable* bits determine whether an exception will cause the FPU to initiate a trap or set a status flag. **If a trap is taken,** the FPU remains in the state found at the beginning of the operation and a software exception handling routine executes. **If a trap is not taken**, an appropriate value is written into the FPU destination register and execution continues. The FPU supports the following five IEEE Standard 754 exceptions:

- *Inexact (I)*
- *Underflow (U)*
- *Overflow (O)*
- *Division by Zero (Z)*
- *Invalid Operation (V)*

The Cause, Enables, and Flag bits (status flags) are used. The FPU adds a sixth exception type, Unimplemented Operation (E). This exception indicates the use of a software implementation. The Unimplemented Operation exception has no Enable or Flag bit; whenever this exception occurs, an unimplemented exception trap is taken. Figure 8.1 illustrates the *Control/Status* register bits that support exceptions.

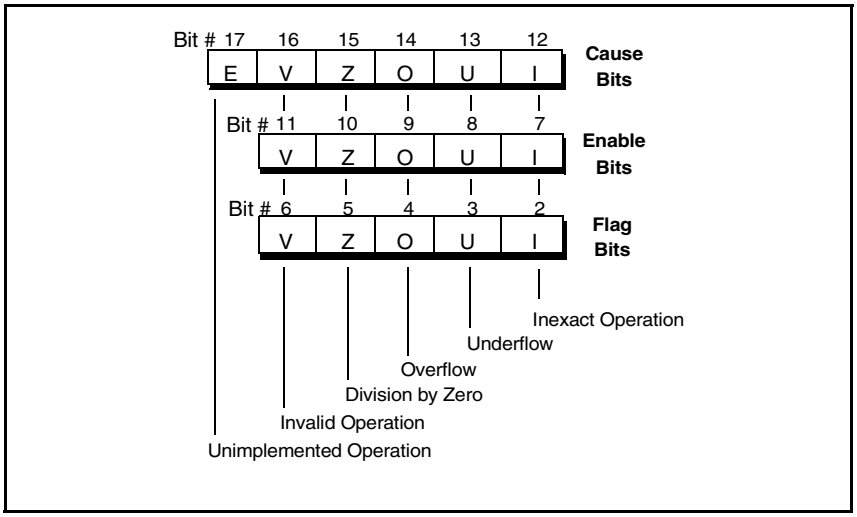


Figure 8.1 Control/Status Register Exception/Flag/Trap/Enable Bits

Each of the five IEEE Standard 754 exceptions (V, Z, O, U, I) is associated with a trap under user control, and is enabled by setting one of the five *Enable* bits. When an exception occurs and its corresponding Enable bit is not set, both the corresponding Cause and Flag bits are set. When an exception occurs and its corresponding Enable bit is set, the corresponding Cause bit is set and the subsequent exception processing allows a trap to be taken.

Notes

Exception Trap Processing

When a floating-point texception trap is taken, the *Cause* register indicates the floating-point coprocessor that is the cause of the exception trap. The Floating-Point Exception (FPE) code is used, and the *Cause* bits of the floating-point *Control/Status* register indicate the reason for the floating-point exception. In effect, these bits are an extension of the system coprocessor Cause register.

Flags

A *Flag* bit is provided for each IEEE exception. This *Flag* bit is set to a 1 on the assertion of its corresponding exception, with no corresponding exception trap signaled. The *Flag* bit is reset by writing a new value into the *Status* register. Flags can be saved and restored by software, either individually or as a group.

When no exception trap is signaled, the floating-point coprocessor takes a default action, providing a substitute value for the exception-causing result of the floating-point operation. The particular default action taken depends upon the type of exception. Table 8.1 lists the default action taken by the FPU for each of the IEEE exceptions.

Field	Description	Rounding Mode	Default action
I	Inexact exception	Any	Supply a rounded result
U	Underflow exception	Any	Take unimplemented unless FCSR.FS bit is set.
O	Overflow exception	RN	Modify overflow values to • with the sign of the intermediate result
		RZ	Modify overflow values to the format's largest finite number with the sign of the intermediate result
		RP	Modify negative overflows to the format's most negative finite number; modify positive overflows to + •
		RM	Modify positive overflows to the format's largest finite number; modify negative overflows to – •
Z	Division by zero	Any	Supply a properly signed •
V	Invalid operation	Any	Supply a quiet Not a Number (NaN)

Table 8.1 Default FPU Exception Actions

The FPU detects the eight exception causes internally. When the FPU encounters one of these unusual situations, it causes either an IEEE exception or an Unimplemented Operation exception (E).

Table 8.2 lists the exception-causing conditions of the IEEE Standard 754

.

FPA Internal Result	IEEE Standard 754	Trap Enable	Trap Disable	Notes
Inexact result	I	I	I	Loss of accuracy
Exponent overflow	O,I[a]	O,I	O,I	Normalized exponent > E_{max}
Division by zero	Z	Z	Z	Zero is (exponent = E_{min}-1, mantissa = 0)
Overflow on convert	V	E	E	Source out of integer range
Signaling NaN source	V	V	V	Signaling NaN source produces quiet NaN result
Invalid operation	V	V	V	0/0, etc.
Exponent underflow	U	E	E	Normalized exponent < E_{min}
Denormalized source	None	E	E	Exponent = E-1 and mantissa <> 0

Note: [a]The IEEE Standard 754 specifies an inexact exception on overflow only if the overflow trap is disabled.

Table 8.2 FPU Exception-Causing Conditions

Notes

FPU Exception Types

The following sections describe the conditions that cause the FPU to generate each of its exceptions, and details the FPU response to each exception-causing condition.

Inexact Exception (I)

The FPU generates the Inexact exception if the rounded result of an operation is not exact or if it overflows. The FPU usually examines the operands of floating-point operations before execution actually begins, to determine (based on the exponent values of the operands) if the operation can *possibly* cause an exception. If there is a possibility of an instruction causing an exception trap, the FPU uses a coprocessor stall to execute the instruction.

It is impossible, however, for the FPU to predetermine if an instruction will produce an inexact result. If Inexact exception traps are enabled, the FPU uses the coprocessor stall mechanism to execute all floating-point operations that require more than two cycles. Since this mode of execution can impact performance, Inexact exception traps should be enabled only when necessary.

Trap Enabled Results: If Inexact exception traps are enabled, the result register is not modified and the source registers are preserved. **Trap Disabled Results:** The rounded or overflowed result is delivered to the destination register if no other software trap occurs.

Invalid Operation Exception (V)

The Invalid Operation exception is signaled if one or both of the operands are invalid for an implemented operation. When the exception occurs without a trap, the MIPS ISA defines the result as a quiet Not a Number (NaN). The invalid operations are:

- *Addition or subtraction: magnitude subtraction of infinities, such as: (+ ∞) + (– ∞) or (– ∞) – (– ∞)*
- *Multiplication: 0 times ∞, with any signs*
- *Division: 0/0, or ∞/∞, with any signs*
- *Comparison of predicates involving* **<** *or* **>** *without* **?**, *when the operands are unordered*
- *Any arithmetic operation on a signaling NaN. A move (MOV) operation is not considered to be an arithmetic operation, but absolute value (ABS) and negate (NEG) are considered to be arithmetic operations and cause this exception if one or both operands is a signaling NaN.*
- *Square root: $\sqrt{x}$, where x is less than zero*

Software can simulate the Invalid Operation exception for other operations that are invalid for the given source operands. Examples of these operations include IEEE Standard 754-specified functions implemented in software, such as Remainder: *x* REM *y*, where *y* is 0 or *x* is infinite; conversion of a floating-point number to a decimal format whose value causes an overflow, is infinity, or is NaN; and transcendental functions, such as ln (–5) or cos–1(3). Refer to Appendix B for examples or for routines to handle these cases.

Trap Enabled Results: The original operand values are undisturbed. **Trap Disabled Results:** The FPU sets the Invalid Operation Exception flag and a quiet NaN is delivered to the destination register.

Division-by-Zero Exception (Z)

The Division-by-Zero exception is signaled on an implemented divide operation if the divisor is zero and the dividend is a finite nonzero number. Software can simulate this exception for other operations that produce a signed infinity, such as ln(0), sec(π/2), csc(0), or 0^{-1}.

Trap Enabled Results: The result register is not modified, and the source registers are preserved. **Trap Disabled Results:** The result, when no trap occurs, is a correctly signed infinity.

Overflow Exception (O)

The Overflow exception is signaled when the magnitude of the rounded floating-point result, with an unbounded exponent range, is larger than the largest finite number of the destination format. (This exception also sets the Inexact exception and *Flag* bits.)

Notes

Trap Enabled Results: The result register is not modified, and the source registers are preserved. **Trap Disabled Results:** The result, when no trap occurs, is determined by the rounding mode and the sign of the intermediate result.

Underflow Exception (U)

Two related events contribute to the Underflow exception:

- *creation of a tiny nonzero result between $\pm 2^{Emin}$ which can cause some later exception because it is so tiny*
- *extraordinary loss of accuracy during the approximation of such tiny numbers by denormalized numbers.*

IEEE Standard 754 allows a variety of ways to detect these events, but requires they be detected the same way for all operations.

Tinniness can be detected by one of the following methods:

- *after rounding (when a nonzero result, computed as though the exponent range were unbounded, would lie strictly between $\pm 2^{Emin}$)*
- *before rounding (when a nonzero result, computed as though the exponent range and the precision were unbounded, would lie strictly between $\pm 2^{Emin}$).*

The MIPS architecture requires that tininess be detected after rounding. Loss of accuracy can be detected by one of the following methods:

- *denormalization loss (when the delivered result differs from what would have been computed if the exponent range were unbounded)*
- *inexact result (when the delivered result differs from what would have been computed if the exponent range and precision were both unbounded).*

The MIPS architecture requires that loss of accuracy be detected as an inexact result.

Trap Enabled Results: When an underflow trap is enabled, underflow is signaled when tininess is detected regardless of loss of accuracy. If underflow traps are enabled, the result register is not modified, and the source registers are preserved. **Trap Disabled Results:** When an underflow trap is not enabled and FCSR.FS is clear, then take an unimplemented exception. When an underflow trap is not enabled and FCSR.FS is set, raise Inexact and return either 0 or $\pm 2^{Emin}$, as appropriate for the current rounding mode.

Unimplemented Instruction Exception (E)

Any attempt to execute an instruction with an operation code or format code that has been reserved for future definition sets the *Unimplemented* bit in the *Cause* field in the FPU *Control/Status* register and traps. The operand and destination registers remain undisturbed and the instruction is emulated in software. Any of the IEEE Standard 754 exceptions can arise from the emulated operation, and these exceptions in turn are simulated. The Unimplemented Instruction exception can also be signaled when unusual operands or result conditions are detected that the implemented hardware cannot handle properly. These include:

- *Denormalized operand*
- *Quiet NaN operand*
- *Underflow*
- *Reserved opcodes*
- *Unimplemented formats*
- *Conversion of a floating-point number to a fixed point format when an overflow occurs or the source operand value is Infinity or a NaN.*
- *Operations which are invalid for their format (for instance, CVT.S.S)*

Denormalized and NaN operands are only trapped if the instruction is a convert or computational operation. Moves and compares do not trap if their operands are either denormalized or NaNs. Use of this exception for such conditions is optional; most of these conditions are newly developed and are not expected to

Notes

be widely used in early implementations. Loopholes are provided in the architecture so that these conditions can be implemented with assistance provided by software, maintaining full compatibility with the IEEE Standard 754.

Trap Enabled Results: The original operand values are undisturbed. **Trap Disabled Results:** This trap cannot be disabled.

Saving and Restoring State

Sixteen or thirty-two doubleword coprocessor load or store operations save or restore the coprocessor floating-point register state in memory. The remainder of control and status information can be saved or restored through Move To/From Coprocessor Control Register instructions, and saving and restoring the processor registers. Normally, the *Control/Status* register is saved first and restored last.

When the coprocessor *Control/Status* register (*FCR31*) is read—and the coprocessor is executing one or more floating-point instructions—the instruction(s) in progress are either completed or reported as exceptions. The architecture requires that no more than one of these pending instructions can cause an exception. Information indicating the type of exception is placed in the *Control/Status* register. When state is restored, state information in the status word indicates that exceptions are pending. Writing a zero value to the *Cause* field of *Control/Status* register clears all pending exceptions, permitting normal processing to restart after the floating-point register state is restored.

The *Cause* field of the *Control/Status* register holds the results of only one instruction; the FPU examines source operands before an operation is initiated to determine if this instruction can possibly cause an exception. If an exception is possible, the FPU executes the instruction in stall mode to ensure that no more than one instruction (that might cause an exception) is executed at a time.

Trap Handlers

The IEEE Standard 754 strongly recommends that users be allowed to specify a trap handler for any of the five standard exceptions that can compute; the trap handler can either compute or specify a substitute result to be placed in the destination register of the operation. By retrieving an instruction using the processor *Exception Program Counter (EPC*) register, the trap handler determines:

- *exceptions occurring during the operation*
- *the operation being performed*
- *the destination format*

On Overflow or Underflow exceptions (except for conversions), and on Inexact exceptions, the trap handler gains access to the correctly rounded result by examining source registers and simulating the operation in software.

On Overflow or Underflow exceptions encountered on floating-point conversions, and on Invalid Operation and Divide-by-Zero exceptions, the trap handler gains access to the operand values by examining the source registers of the instruction. If enabled, the IEEE Standard 754 recommends that the overflow and underflow traps take precedence over a separate inexact trap. This prioritization is accomplished in software; hardware sets the bits for both the Inexact exception and the Overflow or Underflow exception.

Notes

Processor Signal Descriptions

Notes

Introduction

This chapter describes the signals used by and in conjunction with the RC64474 and R64475 processors. Signals include the System interface, the Clock/Control interface, the Interrupt interface, the Initialization interface, the handshake signals and JTAG Interface.

Signal names are listed in bold, and active-low signals have a trailing asterisk. For example, the low-active Read Ready signal is **RdRdy***. The signal description also tells if the signal is an input (the processor receives it) an output (the processor sends it out) or both. Figure 9.1 illustrates the functional groupings of the processor signals.

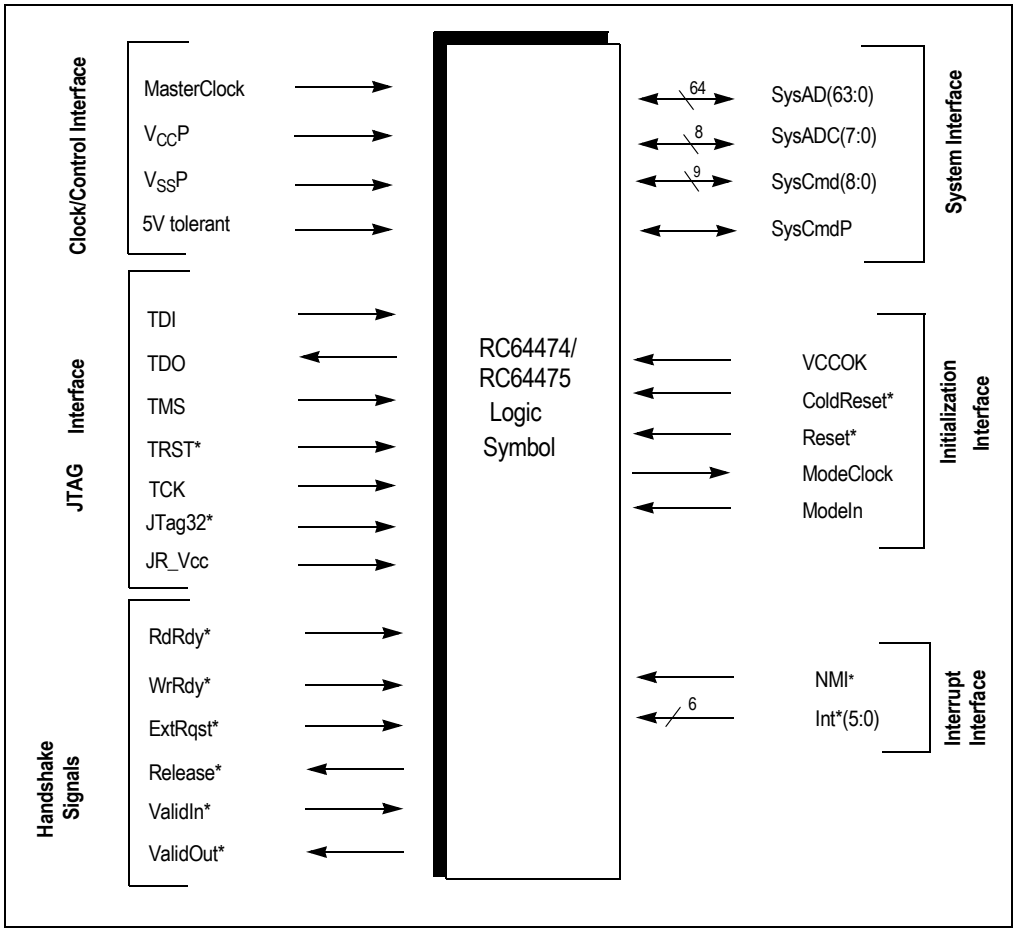


Figure 9.1 RC64474/RC64475 Logic Diagram

System Interface Signals

System interface signals provide the connection between the RC64474 or RC64475 processors and the other system components. The RC64475 supports a 64-bit system interface that is bus compatible with the RC4650 system interface. To ensure socket compatibility between the RC4640 and the RC64474 or the RC4650 and the RC64475, several pin changes are required as shown in the *RC64474/RC64475 RISController Embedded 64-bit Microprocessor* advance data sheet. The data sheet is available at **www.idt.com**. Table 9.1 lists the 64-bit system interface signals of the RC64475. Table 9.2 lists the system interface signals for the RC64474 or the RC64475 in 32-bit mode.

Notes

Name	Definition	Direction	Description
ExtRqst*	External request	Input	An external agent asserts **ExtRqst*** to request use of the System interface. The processor grants the request by asserting **Release***.
Release*	Release interface	Output	In response to the assertion of **ExtRqst*** or a CPU read request, the processor asserts **Release***, signalling to the requesting device that the System interface is available.
RdRdy*	Read ready	Input	The external agent asserts **RdRdy*** to indicate that it can accept a processor read request.
WrRdy*	Write ready	Input	An external agent asserts **WrRdy*** when it can accept a processor write request.
ValidIn*	Valid input	Input	The external agent asserts **ValidIn*** when it is driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.
ValidOut*	Valid output	Output	The processor asserts **ValidOut*** when it is driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.
SysAD(63:0)	System address/ data bus	Input/Output	A 64-bit address and data bus for communication between the processor and an external agent. During address phases only, **SysAd(35:0)** contains invalid address information. The remaining SysAD(63:36) pins are not used. The whole 64-bit SysAD(63:0) are used during the data transfer phase.
SysADC(7:0)	System address/ data check bus	Input/Output	An 8-bit bus containing parity check bits for the SysAD bus during data bus cycles.
SysCmd(8:0)	System command/ data identifier bus	Input/Output	A 9-bit bus for command and data identifier transmission between the processor and an external agent.
SysCmdP	System Command Parity	Input/Output	A single, even-parity bit for the Syscmd bus. This signal is always driven low

Table 9.1 RC64475 System Interface Signals

Table 9.2 lists the system interface signals that apply when the RC64475 is in 32-bit system interface mode or for the RC64474 device. In 32-bit mode, **SysAD (63:32)** and **SysADC (7:6)** are not used, regardless of Endianness.

Name	Definition	Direction	Description
ExtRqst*	External request	Input	An external agent asserts **ExtRqst*** to request use of the System interface. The processor grants the request by asserting **Release***.
Release*	Release interface	Output	In response to the assertion of **ExtRqst*** or a CPU read request, the processor asserts **Release***, signalling to the requesting device that the System interface is available.
RdRdy*	Read ready	Input	The external agent asserts **RdRdy*** to indicate that it can accept a processor read request.
WrRdy*	Write ready	Input	An external agent asserts **WrRdy*** when it can accept a processor write request.
ValidIn*	Valid input	Input	The external agent asserts **ValidIn*** when it is driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.
ValidOut*	Valid output	Output	The processor asserts **ValidOut*** when it is driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.

Table 9.2 RC64474 or RC64475 32-Bit Mode System Interface Signals (Part 1 of 2)

Notes

Name	Definition	Direction	Description
SysAD(31:0)	System address/ data bus	Input/Output	In 32-bit mode and in the RC64474, SysAD(63:32) is not used, regardless of Endianness. A 32-bit address and data communication between processor and external agent is performed via SysAD(31:0).
SysADC(3:0)	System address/ data check bus	Input/Output	An 8-bit bus containing check bits for the **SysAD** data bus cycles.
SysCmd(8:0)	System command/ data identifier	Input/Output	A 9-bit bus for command and data identifier transmission between the processor and an external agent.
SysCmdP	System command parity	Input/Output	A single, even-parity bit for the **SysCmd** bus. This signal is always driven low.

Table 9.2 RC64474 or RC64475 32-Bit Mode System Interface Signals (Part 2 of 2)

Clock/Control Interface

The Clock/Control interface signals make up the interface for clocking and maintenance. Table 9.3 lists the Clock/Control interface signals for the RC64474 and RC64475 devices. These same clock signals are used for both 32-bit and 64-bit system interface modes.

Name	Definition	Direction	Description
Master-Clock	Master clock	Input	Master clock input establishes the processor and bus operating frequency. It is multiplied internally by 2, 3, 4, 5, 6, 7, or 8 to generate the pipeline clock (**PClock**). This clock must be driven by 3.3V (Vcc) clock signals, regardless of the 5V tolerant pin setting.
$V_{CC}P$	**Quiet V_{CC} for PLL**	Input	Quiet V_{CC} for the internal phase locked loop.
$V_{SS}P$	Quiet V_{SS} for PLL	Input	Quiet V_{SS} for the internal phase locked loop.
5V Tolerant	5V Tolerant I/O	Input	This pin is used to convert the I/O ring of the RC4740/50 to 5V tolerant. In pure 3.3V systems no 5V signals will be driven in the CPU, this pin must be driven with Vcc (3.3V). In systems where the RC4740/50 is expected to be driven with 5V signals, this input must be driven with 5V.

Table 9.3 Clock/Control Interface Signals

Interrupt Interface

The Interrupt interface signals make up the interface that is used by external agents to interrupt the RC64474 and RC64475 processor. Six hardware interrupts (**Int*(5:0)**) and one NMI are available on these devices. Table 9.4 lists the Interrupt interface signals. These same signals are used for both 32-bit and 64-bit system interface modes.

Name	Definition	Direction	Description
Int*(5:0)	Interrupt	Input	Six general processor interrupts, bit-wise OR'd with bits 5:0 of the interrupt register.
NMI*	Nonmaskable interrupt	Input	Nonmaskable interrupt, OR'd with bit 6 of the interrupt register.

Table 9.4 Interrupt Interface Signals

Notes

Initialization Interface

The Initialization interface signals make up the interface by which an external agent initializes the processor operating parameters. Table 9.5 lists the Initialization interface signals. These same signals are used for both the 32-bit and 64-bit system interface modes.

Name	Definition	Direction	Description
ColdReset*	Cold reset	Input	This signal must be asserted for a power on reset or a cold reset. **ColdReset*** must be deasserted synchronously with **MasterClock**.
Reset*	Reset	Input	This signal must be asserted for any reset sequence. It can be asserted synchronously or asynchronously for a cold reset, or synchronously to initiate a warm reset. **Reset*** must be deasserted synchronously with **MasterClock**.
V_{CC}Ok	V_{CC} is OK	Input	When asserted, this signal indicates to the processor that $V_{CC} > V_{CC}$-min for more than 100 milliseconds and will remain stable. The assertion of **V_{CC}Ok** initiates the initialization sequence.
ModeClock	Boot-mode clock	Output	Serial boot-mode data clock output; runs at the MasterClock frequency divided by 256: (**MasterClock**/256).
ModeIn	Boot-mode data in	Input	Serial boot-mode data input.

Table 9.5 Initialization Interface Signals

JTAG Interface

The standard JTAG boundary scan is used for on-chip board-level debugging, during Run-time mode. To support the implementation of this interface, the six pins listed in Table 9.6 are available on the RC64474 and RC64475 devices.

RC64475 Signal Summary

Name	Definition	Direction	Description
TDI	JTAG Data In	Input	On the rising edge of TCK, serial input data are shifted into either the Instruction register or Data register, depending on the TAP controller state.
TDO	JTAG Data Out	Output	On the falling edge of TCK, the TDO is serial data shifted out from either the instruction or data register. When no data is shifted out, the TDO is tri-stated (high impedance).
TCK	JTAG Clock Input	Input	An input test clock used to shift into or out of the boundary-scan register cells. TCK is independent of the system and processor clock with nominal 40-60% duty cycle.
TMS	JTAG Command Select	Input	The logic signal received at the TMS input is decoded by the TAP controller to control test operation. TMS is sampled on the rising edge of TCK.
TRST*	JTAG Reset to Reset Circuitry	Input	The TRST* pin is an active-low signal used for asynchronous reset of the debug unit, independent of the processor logic. During normal CPU operation, the JTAG controller will be held in the reset mode, asserting this active low pin. When asserted low, this pin will also cause the TDO into tri-state mode.
JTAG32*	JTAG 32-bit scan	Input	This pin is used to control length of the scan chain for SYsAD (32-bit or 64-bit) for the JTAG mode. When set to Vss, 32-bit bus mode is selected. In this mode, only SysAD(31:0) are part of the scan chain. When set to Vcc, 64-bit bus mode is selected. In this mode, SysAD(63:0) are part of the scan chain. This pin has a built-in pull-down device to guarantee 32-bit scan, if it is left uncovered.

Table 9.6 JTAG Interface Signals

Notes

Table 9.7 lists the RC64475's processor signals and their possible states in 64-bit system interface mode.

Description	Name	I/O	Asserted State	3-State	Reset State
System address/data bus	**SysAD(63:0)**	I/O	High	Yes	a
System address/data check bus	**SysADC(7:0)**	I/O	High	Yes	a
System command/data identifier bus	**SysCmd(8:0)**	I/O	High	Yes	a
System command/data identifier bus parity	**SysCmdP**	I/O	High	Yes	a
Valid input	**ValidIn***	I	Low	No	NA
Valid output	**ValidOut***	O	Low	Yes	b
External request	**ExtRqst***	I	Low	No	NA
Release interface	**Release***	O	Low	Yes	b
Read ready	**RdRdy***	I	Low	No	NA
Write ready	**WrRdy***	I	Low	No	NA
Interrupts	**Int*(5:0)**	I	Low	No	NA
Nonmaskable interrupt	**NMI***	I	Low	No	NA
Boot mode data In	**ModeIn**	I	High	No	NA
Boot mode clock	**ModeClock**	O	High	No	c
Master clock	**MasterClock**	I	High	No	NA
V_{CC} is OK	**VCCOk**	I	High	No	NA
Cold reset	**ColdReset***	I	Low	No	NA
Reset	**Reset***	I	Low	No	NA
JTAG Data In	**TDI**	I	High	No	NA
JTAG Data Out	**TDO**	O	High	Yes	d
JTAG Clock Input	**TCK**	I	High	No	NA
JTAG Command Select	**TMS**	I	High	No	NA
JTAG Reset	**TRST***	I	Low	No	NA
JTAG 32-bit scan	**JTAG32***	I	Low	No	NA
JTAG Vcc	**JR_Vcc***	I	Low	No	NA
5V Tolerant I/O	**5V tolerant**	I	5V	No	NA

Key to Reset State Column:

d All I/O pins (SysAD[63:0], SysADC[7:0], etc.) remain 3-stated until the Reset* signal deasserts.
e All output only pins (ValidOut*, Release*, etc.), except the clocks, are 3-stated until the ColdReset* signal deasserts.
f ModeClock is always driven.
g Control by TRST* only.
NA Not applicable to input pins.

Table 9.7 RC64475 Processor Signal Summary

Notes

RC64474 or RC64475 Signal Summary

Table 9.8 lists the RC64474—or the RC64475 processor signals in 32-bit mode—and their possible states while in 32-bit system interface mode. In this mode **SysADC(63:32)** and **SysADC(7:4)** are not defined.

Description	Name	I/O	Asserted State	3-State	Reset State
System address/data bus	**SysAD(31:0)**	I/O	High	Yes	a
System address/data check bus	**SysADC(3:0)**	I/O	High	Yes	a
System command/data identifier bus	**SysCmd(8:0)**	I/O	High	Yes	a
System command/data identifier bus parity	**SysCmdP**	I/O	High	Yes	a
Valid input	**ValidIn***	I	Low	No	NA
Valid output	**ValidOut***	O	Low	Yes	b
External request	**ExtRqst***	I	Low	No	NA
Release interface	**Release***	O	Low	Yes	b
Read ready	**RdRdy***	I	Low	No	NA
Write ready	**WrRdy***	I	Low	No	NA
Interrupts	**Int*(5:0)**	I	Low	No	NA
Nonmaskable interrupt	**NMI***	I	Low	No	NA
Boot mode data in	**ModeIn**	I	High	No	NA
Boot mode clock	**ModeClock**	O	High	No	c
Master clock	**MasterClock**	I	High	No	NA
V_{CC} is OK	**VCCOk**	I	High	No	NA
Cold reset	**ColdReset***	I	Low	No	NA
Reset	**Reset***	I	Low	No	NA
JTAG Data In	**TDI**	I	High	No	NA
JTAG Data Out	**TDO**	O	High	Yes	d
JTAG Clock Input	**TCK**	I	High	No	NA
JTAG Command Select	**TMS**	I	High	No	NA
JTAG Reset	**TRST***	I	Low	No	NA
JTAG 32-bit scan	**JTAG32***	I	Low	No	NA
JTAG Vcc	**JR_Vcc***	I	Low	No	NA
5V Tolerant I/O	**5V tolerant**	I	5V	No	NA

Key to Reset State Column:

a All I/O pins (SysAD[63:0], SysADC[7:0], etc.) remain 3-stated until the Reset* signal deasserts.
b All output only pins (ValidOut*, Release*, etc.), except the clocks, are 3-stated until the ColdReset* signal deasserts.
c ModeClock is always driven.
d Control by TRST* only.
NA Not applicable to input pins.

Table 9.8 RC64474 & RC64475 Processor Signal Summary

The Clocking, Reset and Initialization Interface

Notes

Introduction

This chapter describes the clock signals (“clocks”) used in the RC64474 and RC64475 processors, including basic system clocks and system timing parameters. Note that throughout the manual, when describing signal transitions, the following terminology is used:

- *Rising edge indicates a low-to-high transition.*
- *Falling edge indicates a high-to-low transition.*
- *Clock-to-Q delay is the amount of time it takes for a signal to move from the input of a device (clock) to the output of the device (Q).*

Figure 10.1 and Figure 10.2 illustrate these terms.

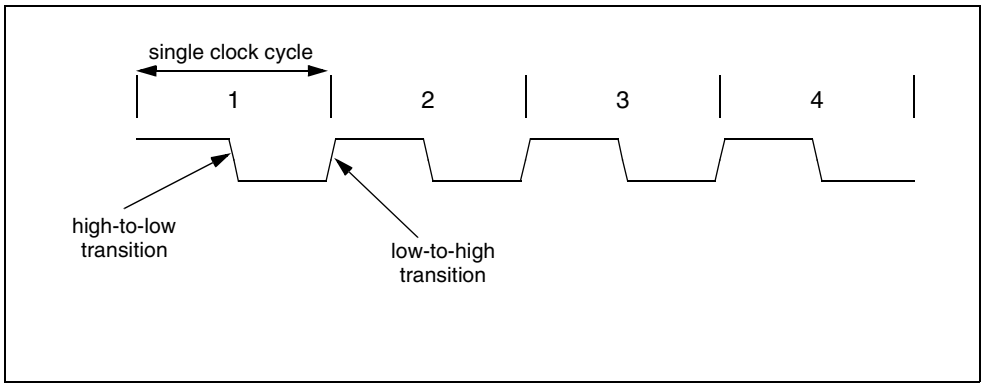


Figure 10.1 Signal Transitions

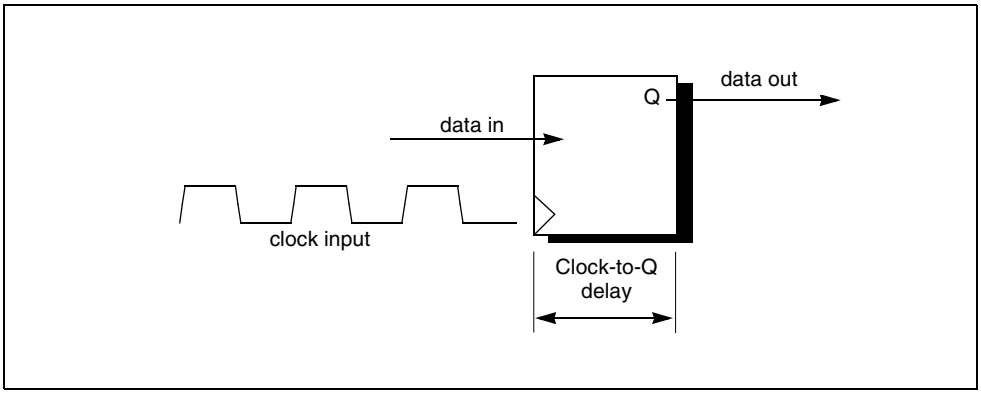


Figure 10.2 Clock-to-Q Delay

System Clocks

The RC64474/RC64475 processors have a single input clock, **MasterClock**, and no output clocks. **MasterClock** is used to base all internal and external clocking, sample data at the system interface, and clock data into the processor system interface output register. The external agent should use **MasterClock** for the global system clock and for clocking the output registers of an external agent.

The processor multiplies **MasterClock** by 2,3,4,5,6,7, or 8 to generate **PClock.** All internal registers and latches (except for **ModeClock, which is part of the initialization interface)** use **PClock**, which is the pipeline clock rate.

Notes

Figure 10.3 shows the clocks data setup, output and hold timing[1] for the basic system clocks.

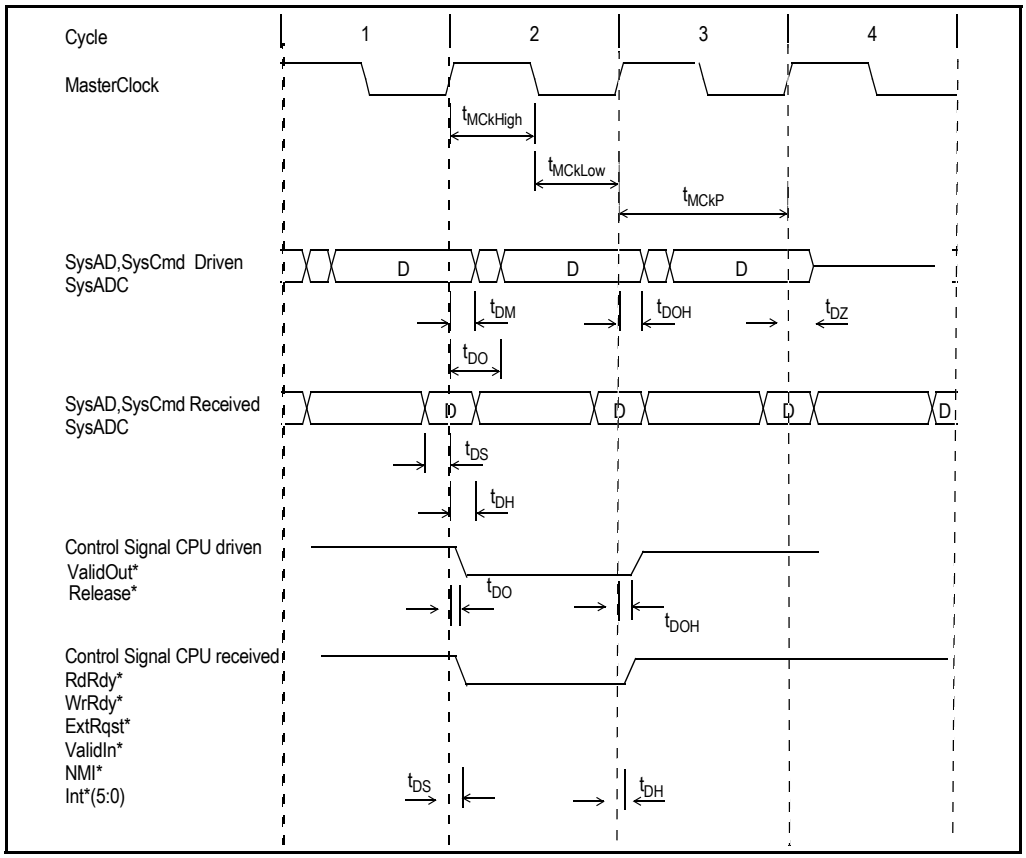


Figure 10.3 RC64474/RC64475 System Clocks Data Setup, Output, and Hold timing

System Timing Parameters

As shown in Figure 10.3, data provided to the processor must be stable a minimum of t_{DS} nanoseconds (ns) before the rising edge of **MasterClock** and be held valid for a minimum of t_{DH} ns after the rising edge of **MasterClock**.

Alignment to MasterClock

Processor data becomes stable a minimum of t_{DM} ns and a maximum of t_{DO} ns after the rising edge of **MasterClock**. This drive-time is the sum of the maximum delay through the processor output drivers together with the maximum clock-to-Q delay of the processor output registers. Processor data is held constant for a minimum of t_{DOH} ns after the rising edge of **MasterClock**. All processor inputs (including **VCCOk**, **ColdReset***, and **Reset***) are sampled based on **MasterClock**, and all outputs are based on **MasterClock**.

Phase-Locked Loop (PLL)

The processor aligns and generates **PClock** with internal phase-locked loop (PLL) circuits. By their nature, PLL circuits are only capable of generating aligned clocks for **MasterClock** frequencies within a limited range.

Clocks generated using PLL circuits contain some inherent inaccuracy, or *jitter*; a clock aligned with **MasterClock** by the PLL can lead or trail **MasterClock** by as much as the related maximum jitter specified in the data sheet.

1. Values for AC & DC parameters are listed in the RC64474/RC64475 *RISController Embedded 64-bit Microprocessor* data sheet.

Notes

PLL Components and Operation

The storage capacitor required for the Phase Locked Loop circuit is contained in the RC64474/RC64475 devices. However, it is recommended that the system designer provide a filter network of passive components for the PLL power supply.

Passive Components

The Phase Locked Loop circuit requires several passive components for proper operation, which are connected to **Vcc**, **Vss**, **VccP**, and **VssP**, as illustrated in 10.4.

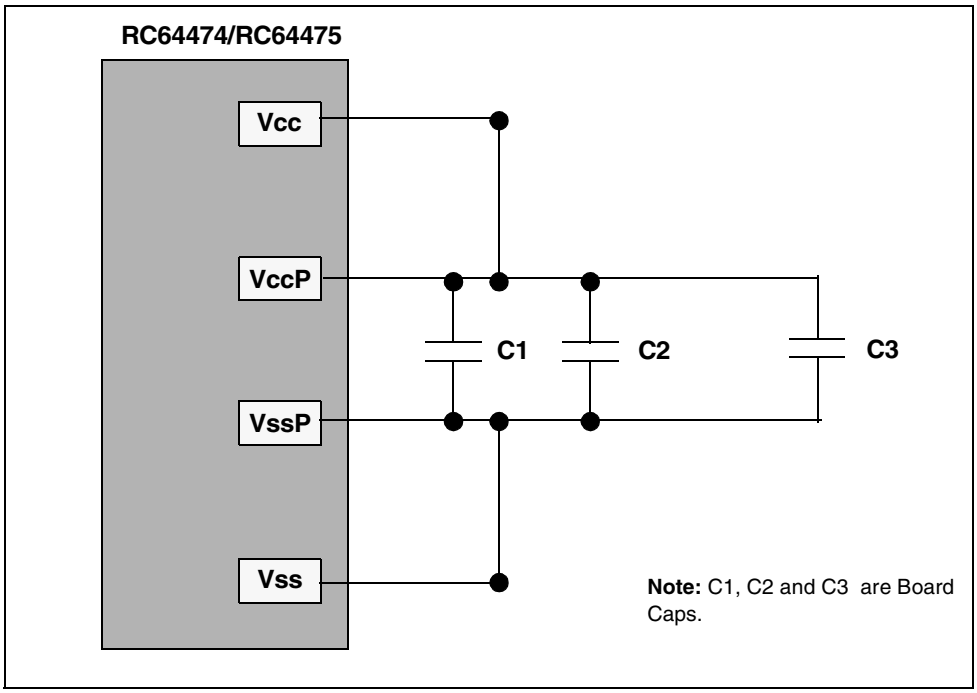


Figure 10.4 PLL Passive Components

It is essential to isolate the analog power and ground for the PLL circuit (**VccP**/**VssP**) from the regular power and ground (**Vcc**/**Vss**). Various evaluations have yielded good results with the following values:

C1 = 1 nF
C2 = 3.3 µF
C3 = 10 µF

Since the optimum values for the filter components depend upon the application and the system noise environment, these values should be considered as starting points for further experimentation within your specific application.

Connecting to an External Agent

MasterClock is used to drive both the processor and the external agent. The RC64474/ uses **MasterClock** to drive its output buffer and to sample the input buffer. Similarly, the external agent should use **MasterClock** to sample its input buffers, drive its output buffer, and act as the system clock. In such a system, the delivery of data and data sampling have common characteristics, even if the processor and external agent have different delay values. For example, *transmission time* (the amount of time a signal takes to move from the processor to external agent to another along a trace on the board) can be calculated from the following equation:

Transmission Time = (MasterClock period)
– (t_{DO} for processor or external agent)
– (t_{DS} for external agent or processor)

Notes

Figure 10.5 shows a block-level diagram of a system using the R4650 processor.

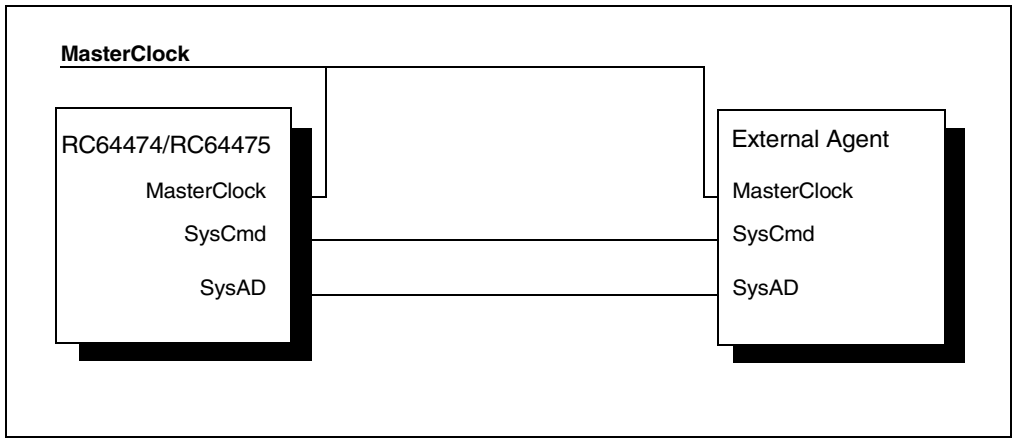


Figure 10.5 RC64474/RC64475 Processor System

Initialization and Reset Interface

The **initialization/reset interface** is a serial interface that operates at the frequency of **MasterClock** divided by 256 (**MasterClock**/256). This low-frequency operation allows the initialization information to be stored in a low-cost Serial EEPROM. The timing of these reset operations is shown in Figure 10.6 on page 10-7, Figure 10.7 on page 10-7, and Figure 10.8 on page 10-7. The RC64474/RC64475 processors have the following three reset types:

- ***Power-on reset****: Starts when the power supply is turned on and completely reinitializes the internal state machine of the processor without saving any state information.*
- ***Cold reset****: Restarts all clocks, but the power supply remains stable. A cold reset completely reinitializes the internal state machine of the processor without saving any state information.*
- ***Warm reset****: Restarts processor, but does not affect clocks. A warm reset preserves the processor internal state.*

Each reset uses the **VCCOk**, **ColdReset***, and **Reset*** input signals, which are summarized in the following section.

Signal Descriptions

This section describes the three reset signals, **VCCOk, ColdReset***, and **Reset*** as well as the two initialization signals, **ModeIn** and **ModeClock**.

- ***VCCOk:*** *When asserted*[1]*,* ***VCCOk*** *indicates to the processor that Vcc has been above the minimum Vcc for more than 100 milliseconds (ms) and is expected to remain stable. The assertion of* ***VCCOk*** *initiates the reading of the boot-time mode control serial stream. This is described in the subsection "Initialization Sequence" on page 10-6.*
- ***ColdReset*:*** *The* ***ColdReset**** *signal must be asserted (low) for either a power-on reset or a cold reset. ColdReset* must be de-asserted synchronously with* ***MasterClock****.*
- ***Reset*:*** *The* ***Reset**** *signal must be asserted for any reset sequence. It can be asserted synchronously or asynchronously for a cold reset, or synchronously to initiate a warm reset.* ***Reset**** *must be de-asserted synchronously with* ***MasterClock.***
- ***ModeIn:*** *Serial boot mode data in.*
- ***ModeClock:*** *Serial boot mode data out, at the* ***MasterClock*** *frequency divided by 256.*

Table 10.1 lists the processor signals and their possible states.

1. *Asserted* means the signal is true, or in its valid state. For example, the low-active **Reset*** signal is said to be asserted when it is in a low (true) state; the high-active **VCCOk** signal is true when it is asserted high.

Notes

Description	Name	I/O	Asserted State	Tri-State	Reset State
System address/data bus	**SysAD(63:0)**	I/O	High	Yes	a
System address/data check bus	**SysADC(7:0)**	I/O	High	Yes	a
System command/data identifier bus	**SysCmd(8:0)**	I/O	High	Yes	a
System command/data identifier bus parity	**SysCmdP**	I/O	High	Yes	a
Valid input	**ValidIn***	I	Low	No	NA
Valid output	**ValidOut***	O	Low	Yes	b
External request	**ExtRqst***	I	Low	No	NA
Release interface	**Release***	O	Low	Yes	b
Read ready	**RdRdy***	I	Low	No	NA
Write ready	**WrRdy***	I	Low	No	NA
Interrupts	**Int*(5:0)**	I	Low	No	NA
Nonmaskable interrupt	**NMI***	I	Low	No	NA
Boot mode data in	**ModeIn**	I	High	No	NA
Boot mode clock	**ModeClock**	O	High	No	d
Master clock	**MasterClock**	I	High	No	NA
Vcc is within specified range	**VCCOk**	I	High	No	NA
Cold reset	**ColdReset***	I	Low	No	NA
Reset	**Reset***	I	Low	No	NA
JTAG Data In	**TDI**	I	High	No	NA
JTAG Data Out	**TDO**	O	High	Yes	d
JTAG Clock Input	**TCK**	I	High	No	NA
JTAG Command Select	**TMS**	I	High	No	NA
JTAG Reset	**TRST***	I	Low	No	NA
JTAG 32-bit scan	**JTAG32***	I	Low	No	NA
JTAG Vcc	**JR_Vcc***	I	Low	No	NA
5V Tolerant I/O	**5V tolerant**	I	5V	No	NA

Key to Reset State Column:

a All I/O pins (**SysAD[63:0]**, **SysADC[7:0]**, etc.) remain 3-stated until the **Reset*** signal deasserts.
b All output only pins (**ValidOut***, **Release***, etc.), except the clocks, are 3-stated until the **ColdReset*** signal deasserts.
c All clocks, except **ModeClock**, are 3-stated until **VCCOk** asserts.
d **ModeClock** is always driven.
NA Not applicable to input pins.

Table 10.1 RC64474/RC64475 Processor Signal Summary

Notes

Power-On Reset

Figure 10.6 on page 10-7, Figure 10.7 on page 10-7 and Figure 10.8 on page 10-7 illustrate the power-on, cold, and warm resets. For a power-on reset, the sequence is as follows:

1. Power-on reset applies a stable Vcc of at least the Vcc minimum value to the processor. During this time, **VCCOk** is deasserted, **ColdReset*** and **Reset*** are asserted and the **MasterClock** input oscillates.
2. After at least 100 ms of stable Vcc and **MasterClock**, the **VCCOk** signal is asserted to the processor. The assertion of **VCCOk** begins the initialization of the processor. After the mode bits have been read in, the processor allows its internal phase locked loop to lock, stabilizing the processor internal clock, **PClock**.
3. **ColdReset*** is asserted for at least 64K (or 216) clock cycles after the assertion of VCCOk. Once the processor reads the boot-time mode control serial data stream, **ColdReset*** can be deasserted. **ColdReset*** must be deasserted synchronously with **MasterClock**.
4. After **ColdReset*** is deasserted synchronously, **Reset*** is deasserted to allow the processor to begin running. Reset* must be held asserted for at least 64 **MasterClock** cycles after the deassertion of **ColdReset***. **Reset*** must be deasserted synchronously with **MasterClock**.

Cold Reset[1]

A cold reset can begin anytime after the processor has read the initialization data stream, causing the processor to start with the Reset exception. A cold reset requires the same sequence as a power-on reset except that the power is presumed to be stable before the assertion of the reset inputs and the deassertion of **VCCOk**. To begin the reset sequence, **VCCOk** must be deasserted for a minimum of 100 ms before reassertion.

Warm Reset

To execute a warm reset, the **Reset*** input is asserted synchronously with **MasterClock**. It is then held asserted for at least 64 **MasterClock** cycles before being deasserted synchronously with MasterClock. The processor internal clock, **PClock**, is not affected by a warm reset. The boot-time mode control serial data stream is not read by the processor on a warm reset. A warm reset forces the processor to start with a Soft Reset exception. **MasterClock** generates any reset-related signals for the processor that must be synchronous with **MasterClock**.

After a power-on reset, cold reset, or warm reset, all processor internal state machines are reset, and the processor begins execution at the reset vector. All processor internal states are preserved during a warm reset, although the precise state of the caches depends on whether or not a cache miss sequence has been interrupted by resetting the processor state machines.

Initialization Sequence

The boot-mode initialization sequence begins immediately after **VCCOk** is asserted. As the processor reads the serial stream of 256 bits through the **ModeIn** pin, the boot-mode bits initialize all fundamental processor modes. (The signals used are described in Chapter 8). The initialization sequence is as follows:

1. The system deasserts the **VCCOk** signal. The **ModeClock** output is held asserted.
2. The processor synchronizes the **ModeClock** output at the time **VCCOk** is asserted. The first rising edge of **ModeClock** occurs at least 256 **MasterClock** cycles after **VCCOk** is asserted. There could be more clock cycles due to internal delays on the **VccOK** signal. After the first rising edge, each additional rising edge will be 256 master clock cycles.
3. Each bit of the initialization stream is presented at the **ModeIn** pin after each rising edge of the **ModeClock**. The processor samples 256 initialization bits from the **ModeIn** input.

1. **ColdReset*** must be asserted when **VCCOk** asserts. The behavior of the processor is undefined if **VCCOk** asserts while **ColdReset*** is deasserted

Notes

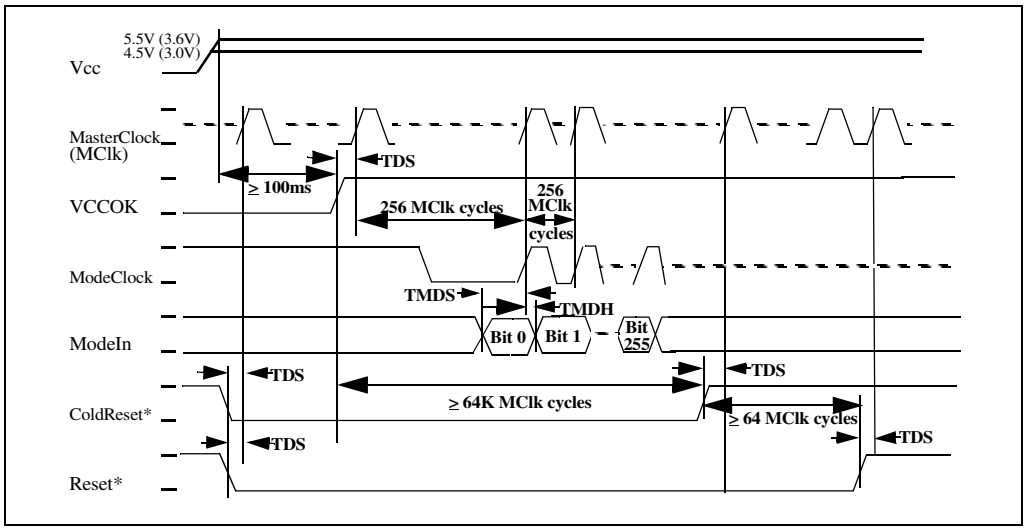


Figure 10.6 Power-on Reset

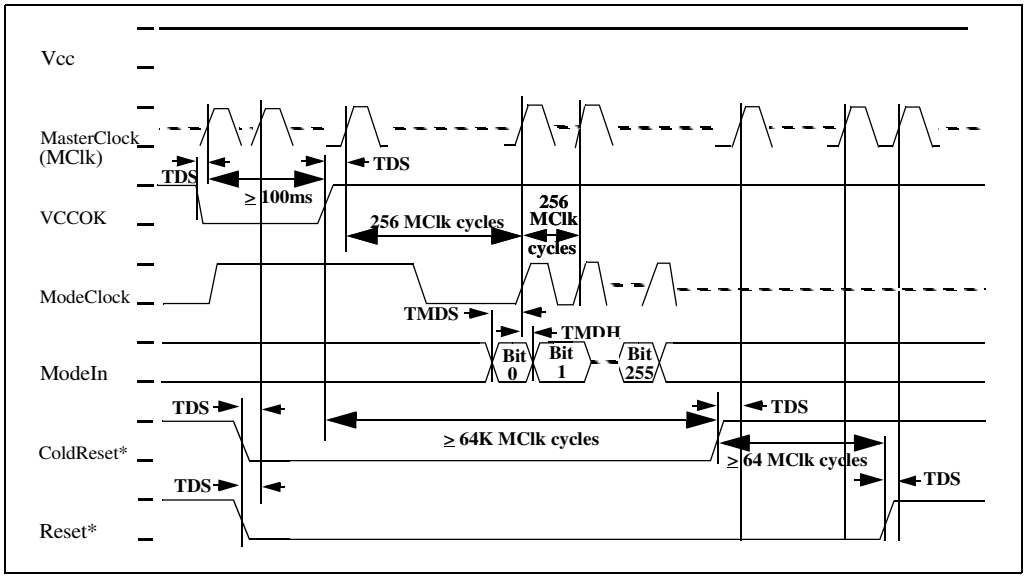


Figure 10.7 Cold Reset

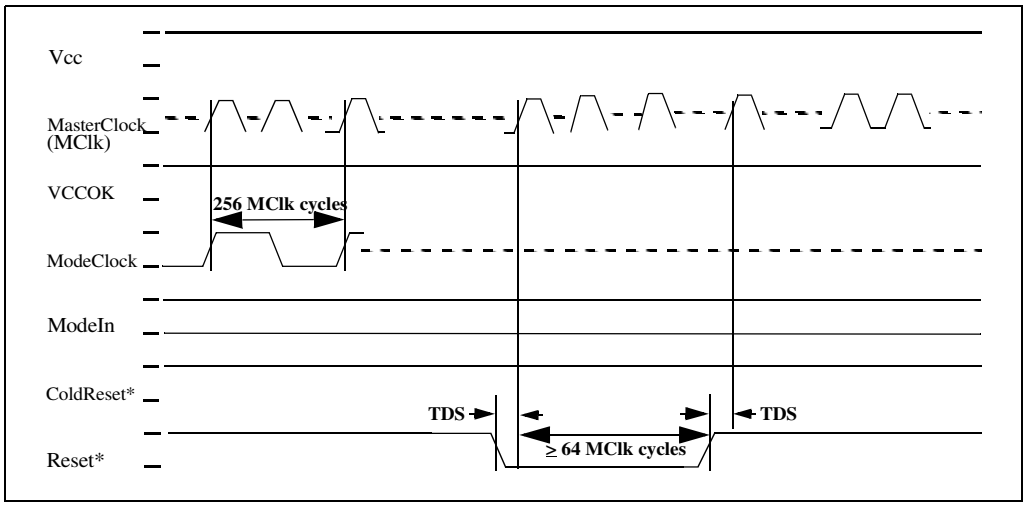


Figure 10.8 Warm Reset

Notes

Boot-Mode Settings

A number of processor operational parameters are determined statically at boot time. These include:

- *Output driver slew rate*
- *Data writeback pattern*
- *System byte ordering*
- ***MasterClock** to **PClock** ratio*
- *Bus interface width.*

Table 10.2 lists the processor boot-mode settings. The following rules apply to the settings in the table:

- *Bit 0 of the stream is presented to the processor when **VCCOk** is first asserted.*
- *Selecting a reserved value results in undefined processor behavior.*
- *Bits 15 to 255 are reserved bits.*
- *Zeros must be scanned in for all reserved bits.*

Serial Bit	Description	Value & Mode Setting
0	Reserved	Must be zero
4:1	**Writeback data rate** System interface data rate for block writes only: **bit 4 is MSB**	64-bit: 0→ ΔΔΔΔ 1→ ΔΔξΔΔξ 2 → ΔΔξξΔΔξξ 3 → ΔξΔξΔξ 4 → ΔΔξξξΔΔξξξ 5 → ΔΔξξξΔΔξξξ 6 → ΔξξΔξξΔξξΔξξ 7→ ΔΔξξξξξξΔΔξξξξξξ 8 → ΔξξξΔξξξΔξξξΔξξξ 9:15 Reserved
		32-bit: 0→ ΩΩΩΩΩΩΩΩ 1→ ΩΩξΩΩξΩΩξΩΩξ 2→ ΩΩξξΩΩξξΩΩξξΩΩξξ 3→ ΩξΩξΩξΩξΩξΩξΩξ 4→ ΩΩξξξΩΩξξξΩΩξξξΩΩξξξ 5→ ΩΩξξξΩΩξξξΩΩξξξΩΩξξξ 6→ ΩξξΩξξΩξξΩξξΩξξΩξξΩξξ 7→ ΩΩξξξξξξΩΩξξξξξξΩΩξξξξξξΩΩξξξξξξ 8→ ΩξξξΩξξξΩξξξΩξξξΩξξξΩξξξΩξξξωξξξ 9:15 Reserved
7:5	**Clock Multiplier** MasterClock is multiplied internally to generate PClock	Clock multiplier: 0 = Μυλτιπλψ βψ 2 1 = Μυλτιπλψ βψ 3 2 = Μυλτιπλψ βψ 4 3 = Μυλτιπλψ βψ 5 4 = Μυλτιπλψ βψ 6 5 = Μυλτιπλψ βψ 7 6 = Μυλτιπλψ βψ 8 7 = Reserved

Table 10.2 Boot-time Mode Stream (Part 1 of 2)

Notes

Serial Bit	Description	Value & Mode Setting
8	**EndBit** Specifies byte ordering	0 → Little endian 1 → Big endian
10:9	**Non-block write** Selects non-block write type. **Bit 10 is MSB.**	00 → RC4x00 compatible 01 → Reserved 10 → pipelined writes 11 → write re-issue
11	**TmrIntEn** Disables the timer interrupt on Int*[5]	0 → Enabled Timer Interrupt 1 → Disabled Timer Interrupt
12	**System interface bus width**	0 → 64-bit system interface 1 → 32-bit system interface
14:13	**Drv_Out** output driver slew rate control. **Bit 14 is MSB.** Affects only non-clock outputs.	Output driver strength: 10 → 100% strength (fastest) 11 → 83% strength 00 → 67% strength 01 → 50% strength (slowest)
17:15	**WAdrWData_Del** Write address to write data delay in MasterClock cycles.	000 → 0 cycles 001 → 1 cycle 010 → 2 cycles 011 → 3 cycles 100 → 4 cycles 101 → 5 cycles 110 → 6 cycles 111 → 7 cycles
255:18	Reserved	Must be 0

Table 10.2 Boot-time Mode Stream (Part 2 of 2)

Notes

Cache Organization, Operation and Coherency

Notes

Introduction

Caches are high speed memories used to buffer the central processing unit from slower, larger storage devices such as those found in main memory. Caches are used to store the data or instructions that a program is currently using while the majority of the data remains in the slower memory, thus providing quick, temporary storage.

In the logical memory hierarchy, caches reside between the CPU and main memory. The increased memory access speed made possible through caches is usually transparent to the programmer. Each functional block, shown in Figure 11.1, has the capacity to hold more data than the block above it. For example, physical main memory has a larger capacity than the primary cache. However, each functional block requires longer access times than any block above it; therefore, it takes longer to access data in main memory than in the CPU on-chip registers.

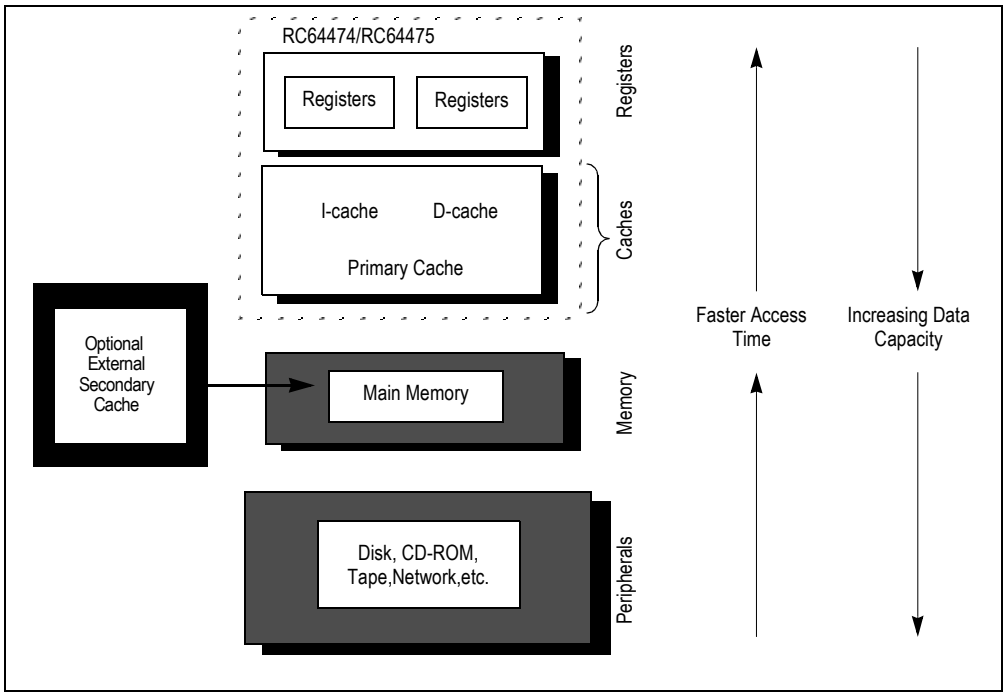


Figure 11.1 Logical Hierarchy of Memory

Cache Operation Overview

To support high-performance RISC designs, the primary cache is made up of an Instruction cache (holds instructions) and a Data cache (holds data). This arrangement allows the processor simultaneous access to both instructions and data, thereby doubling the effective cache-memory bandwidth.

In general, during cache operations, the processor accesses cache-resident instructions/data when the on-chip cache controller detects valid information in the cache by an address match. Figure 11.5 shows the primary cache lookup sequence. If valid instruction or data is present, the processor retrieves it from cache memory and is then known as a primary-cache *hit.* If the instruction/data is not present, a cache *miss* has occurred. The cache line[1] must then be retrieved from slower main memory.

1. Cache Line Size is the smallest unit of information that can be fetched from memory to be filled into the cache. In the RC64474/475, a primary cache line is 8 words in length and is represented by a single tag.

Notes

For a cache hit, the processor retrieves the instruction/data from the (high-speed) primary cache and the operation continues. In the case of a cache miss, the processor can restart the pipeline after the first doubleword is retrieved (the one at the miss address) and continues the cache line refill in parallel.

It is possible for the same data to simultaneously reside in both main memory and primary cache. The data is kept consistent through the use of either a write-back or a write-through methodology. For a write-back cache, the modified data is not written back to memory until the cache line is replaced. In a write-through cache, the data is written to memory as the cached data is modified (with a possible delay due to the write buffer).

RC64474/RC64475 Cache Attributes

In the RC64474 and RC64475 on-chip instruction (I-cache) and data caches (D-cache) have been incorporated. Each cache has its own data path and can be accessed in the same single pipeline clock cycle.

The 16KB two-way set associative I-cache is virtually indexed, physically tagged and word parity protected. Because this cache is virtually indexed, the virtual-to-physical address translation occurs in parallel with the cache access, further increasing performance by allowing both operations to occur simultaneously. The 16KB two-way set associative D-cache is byte parity protected and has a fixed 32-byte (eight words) line size. Its tag is protected with a single parity bit. To allow simultaneous address translation and data cache access, the D-cache is virtually indexed and physically tagged.

To lock critical sections of code and/or data into the caches for quick access, a "cache locking" feature has been implemented. Once enabled, a cache is said to be locked when a particular piece of code or data is loaded into the cache and that cache location will not be selected later for refill by other data. This feature locks a set (8KB) of Instruction and/or Data. Operational details on the cache locking feature are provided later in this chapter. The cache attributes overviewed above are listed in Table 11.1.

Characteristics	Instruction	Data
Size	16KB	16KB
Organization	2-way set associative	2-way set associative
Line size	32B	32B
read unit	32-bits	64-bits
write policy	na	write-back, write-through with or without write-allocate
Line transfer order	sub-block order, for load	sub-block order, for load sequential order, for store
Miss restart after transfer of:	entire line	miss word
Parity	per word	per byte
Cache locking	per set	per set

Table 11.1 RC64474/RC64475 Cache Attributes

Organization and Accessibility

This section describes the organization of the primary cache, including the manner in which it is mapped, the addressing used to index the cache, and composition of the cache lines. The primary instruction and data caches are indexed with a virtual address (VA).

Notes

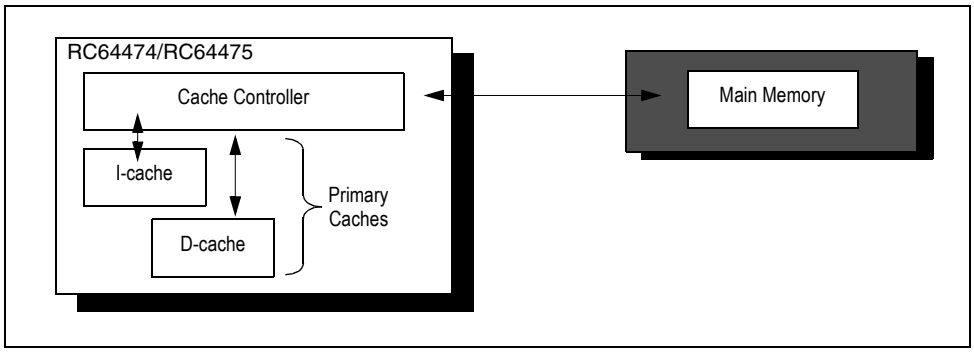


Figure 11.2 Cache Support in the RC64474/RC64475

Primary Instruction Cache (I-Cache) Organization

Each line of primary I-cache data (although it is actually an instruction, it is referred to as data to distinguish it from its tag) has an associated 24-bit tag that contains a 20-bit physical address, a single valid bit, a reserved bit, a single parity bit and the FIFO replacement bit. Word parity is used on I-cache data. Figure 11.3 shows the format of a primary I-cache line. As discussed earlier, the primary I-cache of the RC64474/RC64475 devices has the following characteristics:

- *two-way set associative*
- *indexed with a virtual address*
- *checked with a physical tag*
- *organized with 8-word (32-byte) cache line*
- *lockable on a per-set basis*

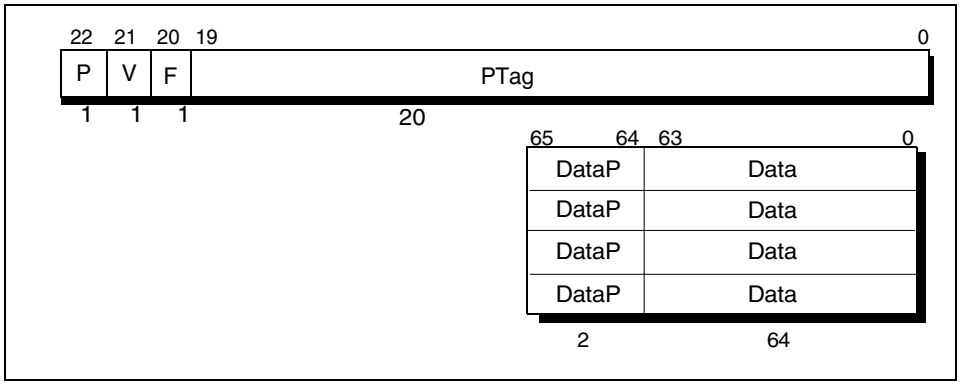


Figure 11.3 RC64474/RC64475 Primary I-Cache Line Format

Field	Description
PTag	Physical tag (bits 31:12 of the physical address)
V	Valid bit
F	FIFO Replacement Bit. Complemented on refill.
P	Even parity for the PTag and V fields
DataP	Even parity; 1 parity bit per word of data
Data	Cache data

Table 11.2 Primary I-Cache Field Descriptions

Notes

Primary Data Cache (D-Cache) Organization

Each line of primary D-cache data has an associated 26-bit tag that contains a 20-bit physical address, 2-bit cache line state, a write-back bit, a parity bit for the physical address and cache state fields, a parity bit for the write-back bit, and the FIFO replacement bit. Figure 11.4 shows the format of a primary D-cache line. The processor's primary D-cache has the following characteristics:

- *write-back or write-through on a per-page basis*
- *two-way set associative*
- *indexed with a virtual address*
- *checked with a physical tag*
- *organized with 8-word (32-byte) cache line*
- *Lockable on a per-set basis*

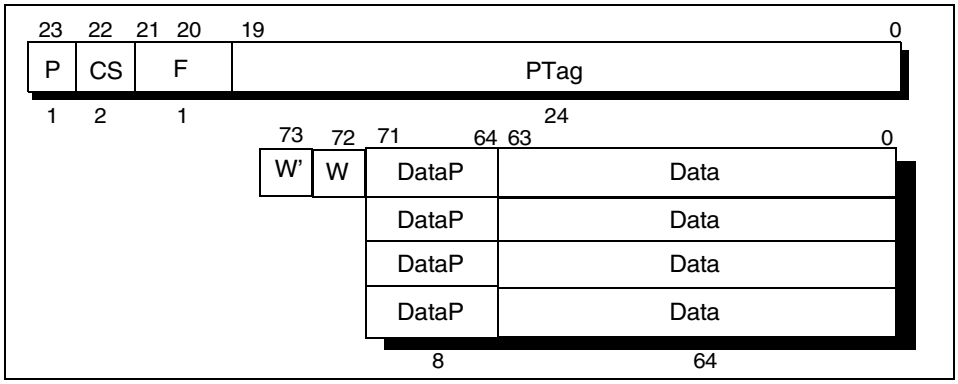


Figure 11.4 Primary D-Cache Line Format

Field	Description
F	FIFO Replacement Bit
W'	Even parity for the write-back bit
W	Write-back bit (set if cache line has been written)
P	Even parity for the PTag and CS fields
CS	Primary cache state: 0 = Invalid, 1 = Shared, 2 = Clean Exclusive, 3 = Dirty Exclusive
PTag	Physical tag (bits 35:12 of the physical address)
DataP	Even parity for the data; 1-bit per byte
Data	Cache data

Table 11.3 D-Cache Field Descriptions

In the RC64474/RC64475 devices, the *W* (write-back) bit—not the cache state—indicates whether or not the primary cache contains modified data that must be written back to memory.

Note: There is no hardware support for cache coherency. The only cache states used are Dirty Exclusive and Invalid.

Notes

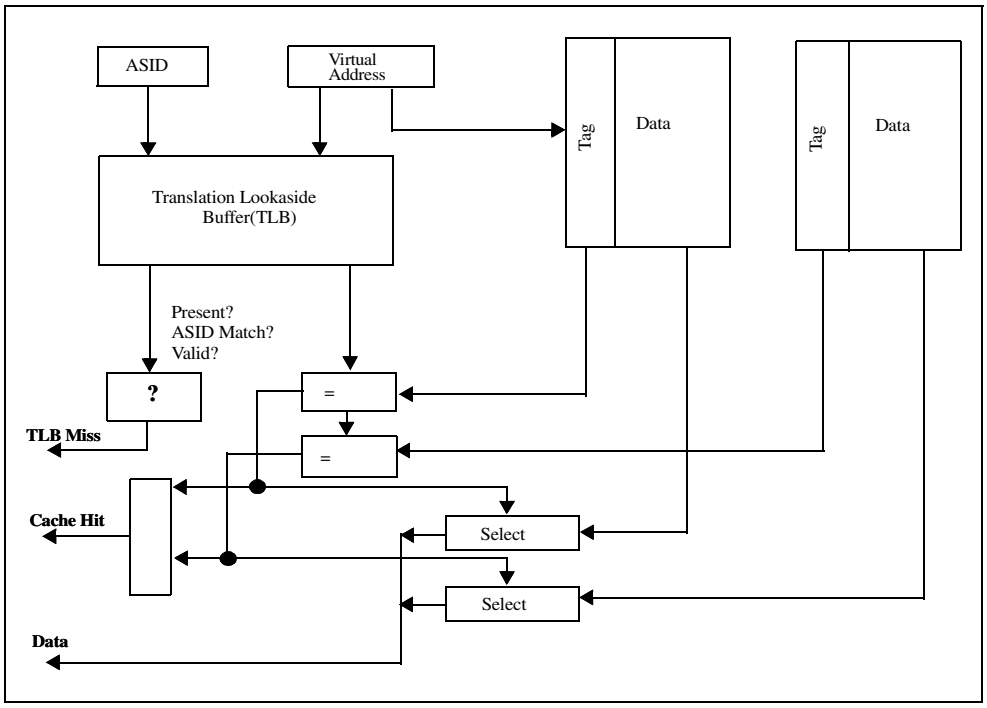


Figure 11.5 Conceptual Primary Cache Lookup Sequence

Accessing Primary Caches

Figure 11.6 shows the virtual address (VA) index into the primary caches. Each instruction and data cache size is 16 Kbytes.

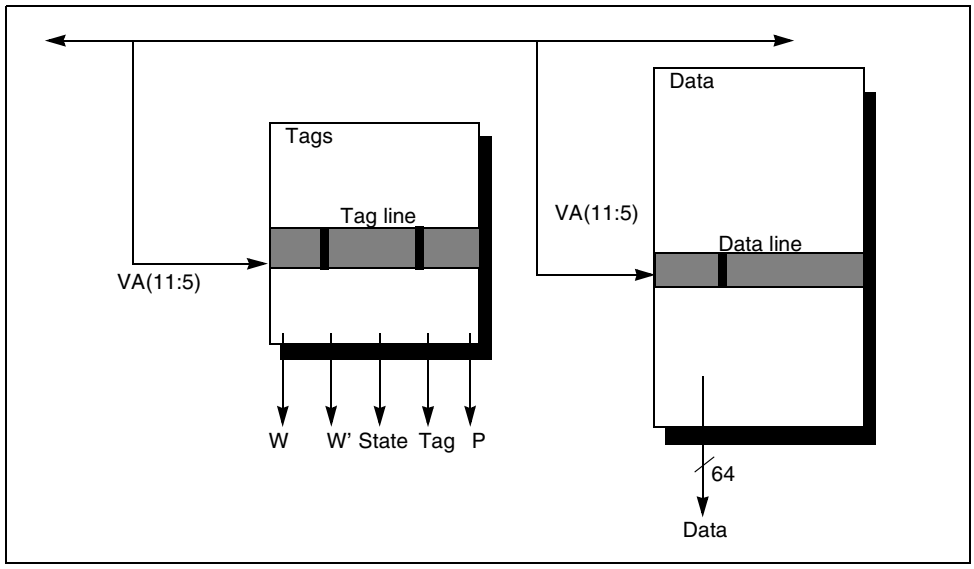


Figure 11.6 Primary Cache Data and Tag Organization

Cache States

The terms below are used to describe the *state* of a cache line:

- ***Exclusive****: a cache line that is present in exactly one cache in the system is exclusive. This is always the case for the RC64474/RC64475. All cache lines are in an exclusive state.*
- ***Dirty****: a cache line that contains data that has changed since it was loaded from memory is dirty.*

Notes

- ***Clean****: a cache line that contains data that has not changed since it was loaded from memory is clean.*
- ***Shared****: a cache line that is present in more than one cache in the system. The RC64474/RC64475 does not provide for hardware cache coherency. This state will never happen in normal operations.*

The RC64474/RC64475 processors only supports the four cache states as shown in Table 11.4 on page 11-6. The only states that will occur in the devices, under normal operations are the Dirty Exclusive and Invalid states.

Note: Although valid data is in the Dirty Exclusive state, it may still be consistent with memory. One must look at the dirty bit, W, to determine if the cache line is to be written back to memory when it is replaced. Each primary cache line in the RC64474/RC64475 system is in one of the states described in Table 11.4.

Cache Line State	Description
Invalid	A cache line that does not contain valid information must be marked invalid, and cannot be used. A cache line in any other state than invalid is assumed to contain valid information.
Shared	A cache line that is present in more than one cache in the system is shared. This state will not occur for normal operations.
Clean Exclusive	A clean exclusive cache line contains valid information and this cache line is not present in any other cache. The cache line is consistent with memory and is not owned by the processor (see “Cache-Line Ownership” on page 11-6 in this chapter). This state will not occur for normal operations.
Dirty Exclusive	A dirty exclusive cache line contains valid information and is not present in any other cache. The cache line may or may not be consistent with memory and is owned by the processor (see “Cache-Line Ownership” on page 11-6 in this chapter). Use the W bit to determine if the line must be written back on replacement.

Table 11.4 Primary Cache States

Note that of the above primary cache states, each primary **data cache line** is normally in an **invalid** or **dirty exclusive** cache state. Each primary **instruction cache line** is normally in an **invalid** or **valid** cache state.

Cache-Line Ownership

The processor is the owner of a cache line when it is in the dirty exclusive state and is also responsible for the contents of that line. There can only be one owner for each cache line. The ownership of a cache line is set and maintained through the rules described below.

- *A processor assumes ownership of the cache line if the state of the primary cache line is dirty exclusive.*
- *A processor that owns a cache line is responsible for writing the cache line back to memory if the line is replaced during the execution of a Write-back or Write-back Invalidate cache instruction if the line is in a write-back page.*
- *Memory always owns clean cache lines*
- *The processor gives up ownership of a cache line when the state of the cache line changes to invalid.*

Therefore, based on these rules and that any valid data cache line is in the Dirty Exclusive state (under normal operating conditions), the processor is considered to be the owner of the cache line.

Notes

Cache Write Policy

The RC64474/RC64475 processors manage their primary data cache by using either a write-back or a write-through policy, determined by settings in the “C” field in the TLB entry. In a write-back cache, the data is not written back to memory until the cache line is replaced. A write-through policy means the store data is written to the cache and to memory. The write of the data to memory may not occur at the same time as the write to cache due to the write buffer.

For a write-back entry, if the cache line is valid and has been modified (the *W* bit is set), the processor writes this cache line back to memory when the line is replaced, either in the course of satisfying a cache miss or during the execution of a Write-back or Write-back Invalidate CACHE instruction.

For a write-through entry, whenever a store hits in the cache line, the data is also written to memory via the write buffer. The store will not set or clear the *W* bit for a write-through cache line. This allows a different virtual address that maps to the same physical address and with a write-back policy to set the *W* bit. For a miss to a write-through line, the action taken is determined by the write-allocation policy. For a write-allocate entry, the cache line is first retrieved from memory and the store continues. A no write-allocate entry posts the write to the system interface via the write buffer, in the same manner as an uncached write.

When the processor writes a cache line back to memory, it does not ordinarily retain a copy of the cache line, and the state of the cache line is changed to invalid. However, there are exceptions. For example, the processor retains a copy of the cache line if a cache line is written back by the Hit Write-back cache instruction. If the W bit is set, the cache line is written back and the W bit is cleared. The processor signals this line retention during a write by setting **SysCmd(2)** to a 1.

Cache State-Transition Diagrams

The following sections describe the cache state diagrams that illustrate the cache state transitions for the primary cache. Figure 11.7 shows the state diagram of the primary cache.

When an external agent supplies a cache line, it need not return the initial state of the cache line, for normal operations (refer to Chapter 12 for a definition of an external agent). This is because the only read request the processors should issue are for non-coherent data and the lower three bits for the data identifier are reserved. The initial state will automatically be set to DE by the RC64474/RC64475. Otherwise, the processor changes the state of the cache line during one of the following events:

- *A store to a dirty exclusive line remains in a dirty exclusive state.*
- *The state is changed to invalid for:*
  - *a Cache invalidate operation*
  - *if the line is replaced*

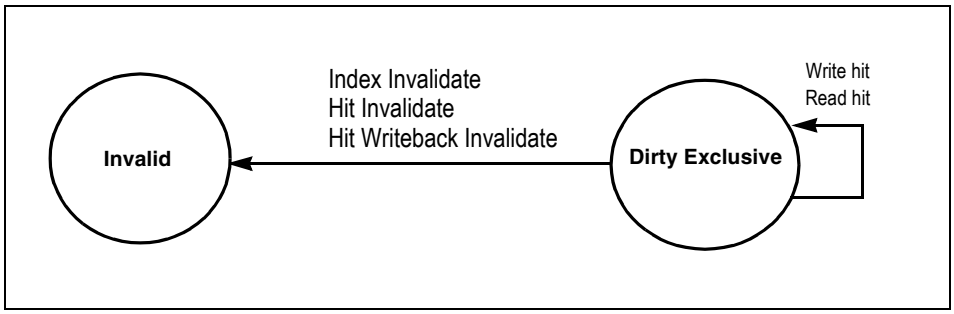


Figure 11.7 Primary Data Cache State Diagram

Cache Coherency Overview

Systems using more than one master must have a mechanism to maintain data consistency throughout the system. This mechanism is called a cache coherency protocol. The RC64474/RC64475 do not provide any hardware cache coherency. Cache coherency must be handled with software.

Notes

Cache Coherency Attributes

Cache coherency attributes are necessary to ensure the consistency of data throughout the system. Bits in the CAlg register control coherency according to the virtual address. Specifically, the CAlg register contains 3 bits per entry that provide two possible coherency attribute types; they are listed below and described more fully in the following sections.

- *uncached*
- *noncoherent (includes 3 attribute values)*

Table 11.5 summarizes the behavior of the processor on load misses and store misses for each of the coherency attribute types listed above. The following sections describe in detail these coherency attribute types.

Attribute Type	Load Miss	Store Miss
Uncached	Main memory read	Main memory write
Noncoherent	Noncoherent read	Noncoherent read (write-allocate page) Main memory write (no write-allocate page)

Table 11.5 Coherency Attributes and Processor Behavior

Uncached

Lines within an *uncached* page are never in a cache. When a virtual address has the uncached coherency attribute, the processor issues a doubleword, partial-doubleword, word, or partial-word read or write request directly to main memory (bypassing the cache) for any load or store to a location within that page.

Noncoherent

Lines with a *noncoherent* attribute type can reside in a cache; a load miss causes the processor to issue a noncoherent block read request to a location within the cached page. For a store miss to a write-allocate page, the processor issues a noncoherent block read request to a location within the cached page and then does the write-through. If the virtual address has the no write-allocate attribute, a store miss will generate a write to the memory as in the uncached case.

Note: The RC64474/RC64475 processors only support the no-secondary-cache mode (only uncached and noncoherent coherency attributes are applicable) of RC4400 operation.

Cache Locking

These processors implement a feature referred to as “cache locking.” That is, the kernel may set status register control bits that inhibit the cache refill process from displacing valid contents in set “A” of either cache. Note that these bits do not inhibit caches from being changed by any of the following operations or conditions:

- *cache operations*
- *store operations to D-cache*
- *if they are invalid*

A cache is said to be *locked* when a particular piece of code or data is loaded into the cache and that cache location will not be automatically elected later for refill by other data. Cache locking is useful when:

- *a portion of code has to reside in cache permanently (e.g. time critical exception vectors) for real-time performance*
- *a given section of code is executed frequently and can fit inside the instruction cache*
- *a given section of data is accessed frequently and can fit inside the data cache (e.g. tables containing routing information in an embedded network application)*

Notes

Each 16KB cache is two-way set associative, with set A and set B. The size of each set is 8KB. On reset, both sets A and B are unlocked. By setting the DL or IL bit in the Status register of CP0, set A of the appropriate cache can be prevented from being chosen for refill on a cache miss, thus effectively locking the contents of the cache. The restriction on only set A being lockable is only for deterministic performance.

If both sets are invalid, the CPU always chooses set A. Similarly, data store operations to locked data update the D-cache contents; as above, locking merely prevents the cache line contents from being replaced by the contents of a different physical location. Otherwise, if a set is locked, its contents will not be changed. An invalid line in a locked set will still be chosen for refill on a cache miss. Once refilled (and thus valid), this line will not be selected for refill until the appropriate lock bit is reset. This understanding, along with knowledge of Coprocessor 0 (CP0) hazards, can be used to develop a small and efficient algorithm for cache locking in the RC64474/RC64475.

The basic algorithm presented here consists of the following three steps. Two examples follow.

1. Invalidate the cache(s).
2. Set the appropriate cache lock bit(s).
3. Load the critical code/data into the cache(s).

Data Cache Locking Example

Assume an example application in which there is a table that must always be kept in cache. In the startup code, after initialization of data structures, flushing of caches, etc., is done, the user can perform reads through cached addresses to load the data into the data cache, and then set the DL bit in the Status register to lock set A of the data cache. A sample code fragment follows:

```
        .set noreorder
        jal     flush_cache         /* Flush caches */
        nop
        la      t0, critical_table  /* This table should always be in cache */
        li      t1, table_size      /* Size of table in bytes */
        li      t2, 0               /* Number of bytes read into cache */
1:      lw      a0, 0(t0)
        addiu   t2, 4
        bneq    t2, t1, 1b          /* Loop back till done */
        addiu   t0, 4               /* bump read address */

        mfc0    a0, C0_SR           /* Get old SR value */
        li      a1, SR_DL           /* SR_DL = 0x00100000 */
        or      a0, a0, a1
        mtc0    a0, C0_SR           /* Set the Lock bit for data cache */
        nop
        nop
        nop                         /* 3 nops: safety against CP0 hazard */
```

Instruction Cache Locking

Assume an example application in which there is a critical function that must always be kept in cache. Also assume that the size of the function is known. (If not known, you can find out the size by generating a disassembly of the object file.)

In the startup code, after initializing data structures, flushing of caches, etc., is done, you can perform the FILL operation in the CACHE instruction to fill the instruction cache with the critical function, and then set the IL bit in the Status register to lock set A of the instruction cache. A sample code fragment follows:

Notes

```
        .set noreorder
        la      t0, 1f              /* Get address of label '1' */
        li      t1, 0xA0000000
        or      t0, t0, t0
        jr       t0                 /* Uncached execution from now onwards */
        nop
1:      jal      flush_cache
        nop
        la      t0, func_start_addr /* Start address of critical code */
        li      t1, func_size       /* Critical code size */
        li      t2, 0               /* Number of words read into cache */
2:      cache    Fill_I, 0(t0)      /* Fill Operation */
        addiu   t2, 4
        bneq    t2, t1, 1b          /* Loop back till done */
        addiu   t0, 4               /* bump read address */

        mfc0     a0, C0_SR          /* Get old SR value */
        li       a1, SR_IL          /* SR_IL = 0x00080000 */
        or       a0, a0, a1
        mtc0     a0, C0_SR          /* Set Lock bit for instruction cache */
        nop
        nop
        nop
        nop
        nop                         /* 5 nops: safety against CP0 hazard */
la              v0, 3f
jr              v0
nop

3:                                  /* Resume execution in mode as linked */
```

Synchronization Support

In a multiprocessor system, it is essential that two or more processors working on a common task can execute without corrupting each other's subtasks. *Synchronization*, an operation that guarantees an orderly access to shared memory, must be implemented for a properly functioning multiprocessor system. Two of the more widely used methods are discussed in this section: test-and-set, and counter. Even though the processors do not support symmetric multi-processing (SMP), these are useful for multi-master and heterogenous multi-processing.

Test-and-Set

Test-and-set uses a variable called the *semaphore*, which protects data from being simultaneously modified by more than one processor. In other words, a processor can lock out other processors from accessing shared data when the processor is in a *critical section*, a part of program in which no more than a fixed number of processors is allowed to execute. In the case of test-and-set, only one processor can enter the critical section.

Notes

Figure 11.8 illustrates a test-and-set synchronization procedure that uses a semaphore; when the semaphore is set to 0, the shared data is unlocked, and when the semaphore is set to 1, the shared data is locked.

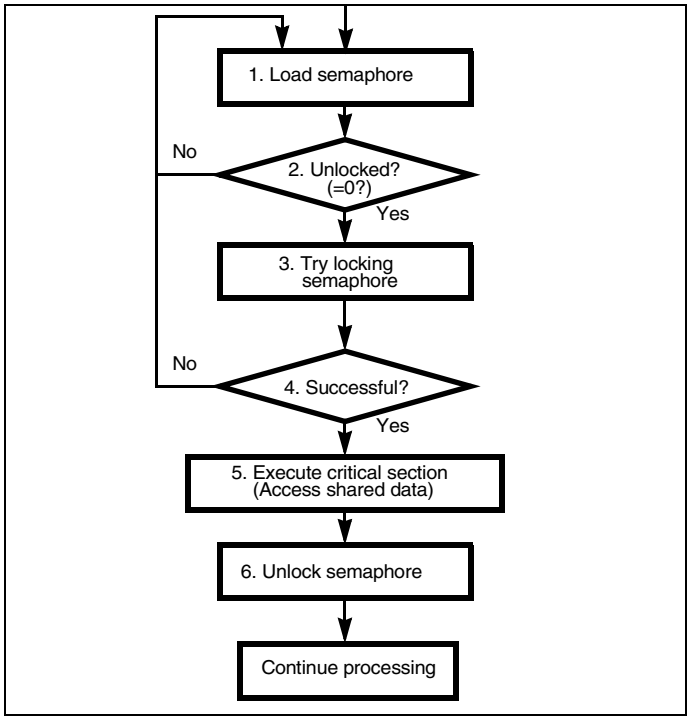


Figure 11.8 Synchronization with Test-and-Set

The processor begins by loading the semaphore and checking to see if it is unlocked (set to 0) in steps 1 and 2. If the semaphore is not 0, the processor loops back to step 1. If the semaphore is 0, indicating the shared data is not locked, the processor next tries to lock out any other access to the shared data (step 3). If not successful, the processor loops back to step 1, and reloads the semaphore. If the processor is successful at setting the semaphore (step 4), it executes the critical section of code (step 5) and gains access to the shared data, completes its task, unlocks the semaphore (step 6), and continues processing.

Counter

Another common synchronization technique uses a *counter. A counter is* a designated memory location that can be incremented or decremented. In the test-and-set method, only one processor at a time is permitted to enter the critical section. Using a counter, up to *N* processors are allowed to concurrently execute the critical section. All processors after the *N*th processor must wait until one of the *N* processors exits the critical section and a space becomes available.

The counter works by not allowing more than one processor to modify it at any given time. Conceptually, the counter can be viewed as a variable that counts the number of limited resources (for example, the number of processes, or software licenses, etc.).

Figure 11.9 illustrates this process.

Notes

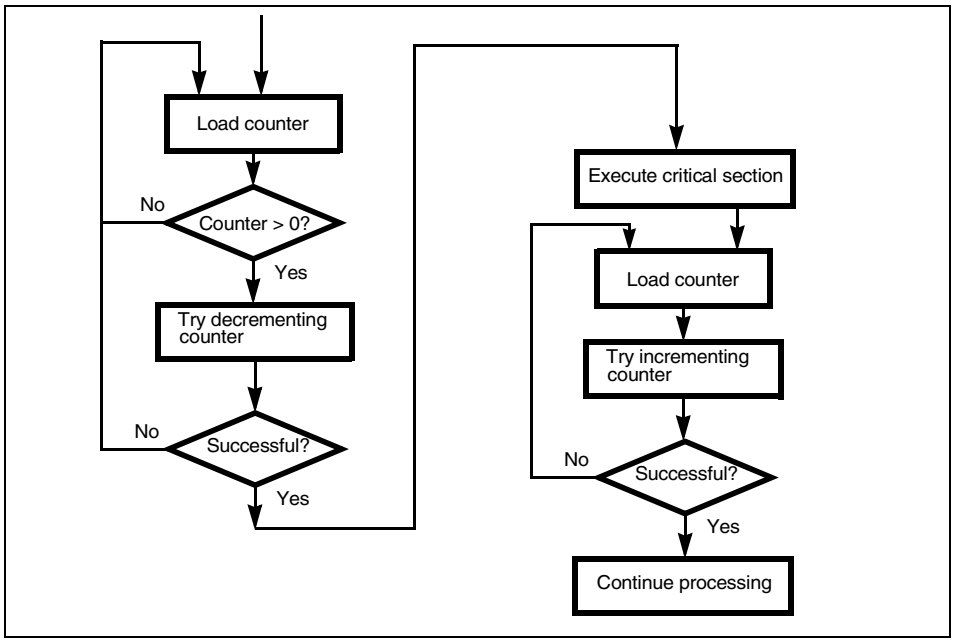


Figure 11.9 Synchronization Using a Counter

Load Linked and Store Conditional

The RC64474/RC64475 instructions *Load Linked* (LL) and *Store Conditional* (SC) provide support for processor synchronization. These two instructions work very much like their simpler counterparts, load and store. The LL instruction, in addition to doing a simple load, has the side effect of setting a bit called the *link bit*. This link bit forms a breakable link between the LL instruction and the subsequent SC instruction. The SC performs a simple store if the link bit is set when the store executes. If the link bit is not set, then the store fails to execute. The success or failure of the SC is indicated in the target register of the store.

The link is broken upon completion of an ERET (return from exception) instruction. The most important features of LL and SC are that:

- *they provide a mechanism for generating all of the common synchronization primitives including test-and-set, counters, sequencers, etc., with no additional overhead*
- *when they operate, bus traffic is generated only if the state of the cache line changes; lock words stay in the cache until some other processor takes ownership of that cache line*

Examples Using LL and SC

Figure 11.10 shows how to implement test-and-set using LL and SC instructions.

Notes

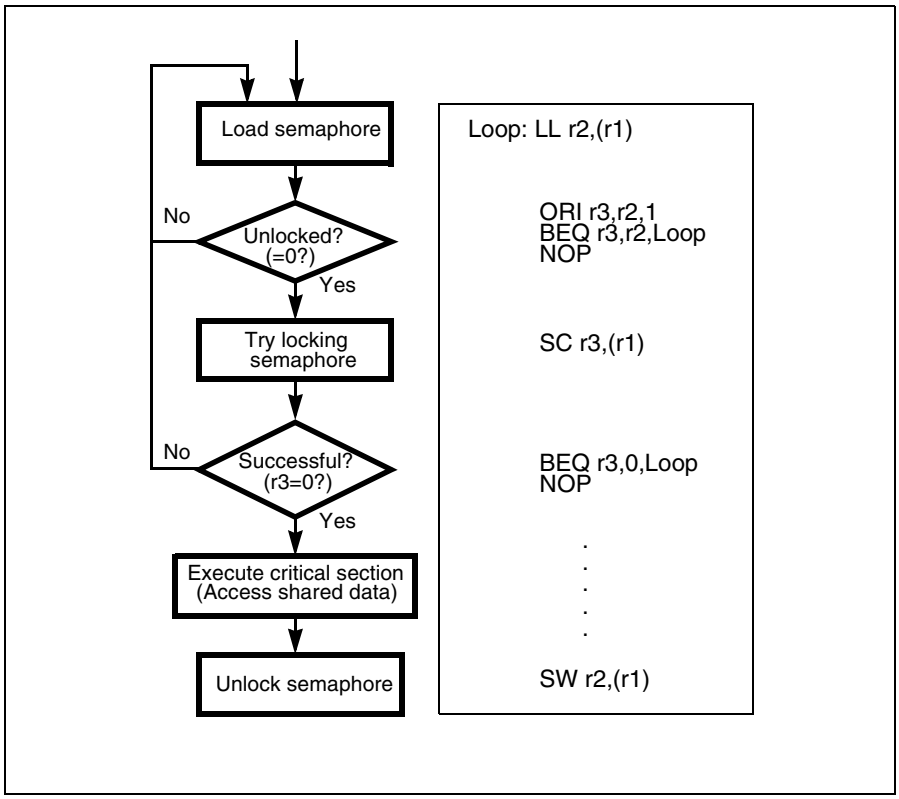


Figure 11.10 Test-and-Set using LL and SC

Figure 11.11 shows synchronization using a counter.

Notes

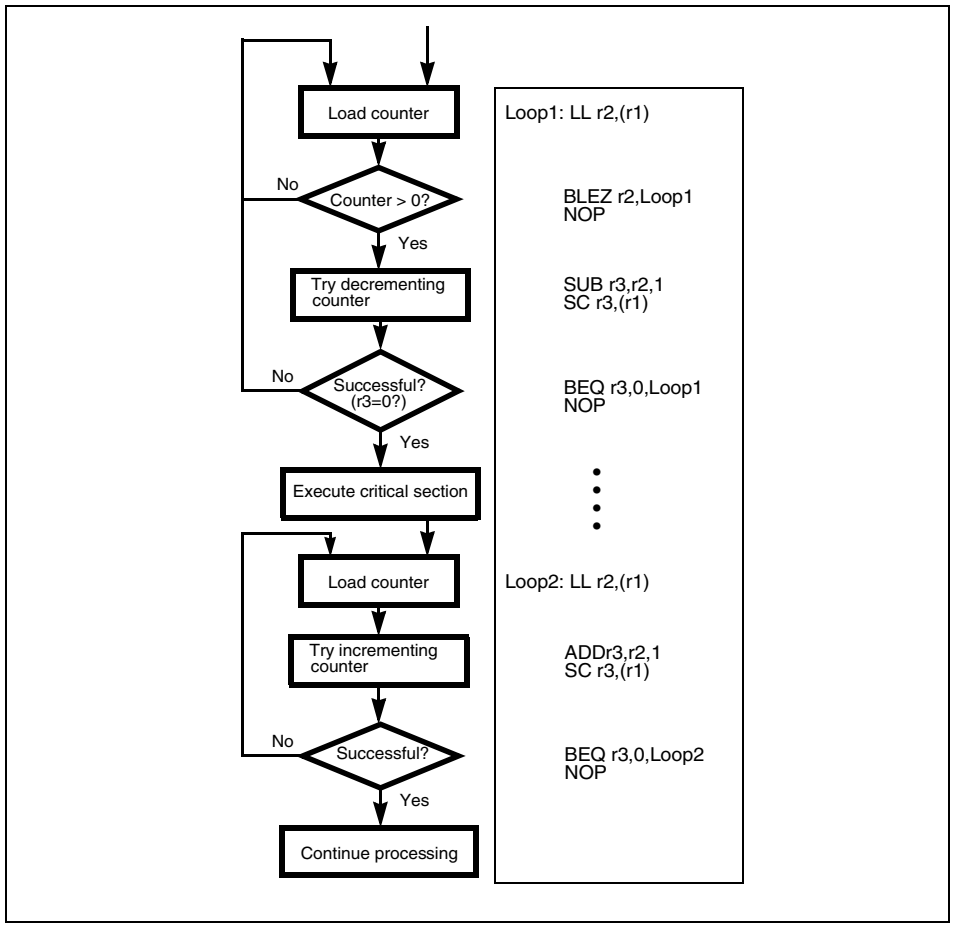


Figure 11.11 Counter Using LL and SC

System Interface Overview

Notes

Introduction

The System interface allows the processor to access external resources that are needed to satisfy cache misses and uncached operations, while permitting an external agent access to some of the processor internal resources. This chapter describes the system interface from the point of view of both the processor and the external agent.

Terminology

The following terms are used in this chapter:

- *An external agent is any logic device connected to the processor over the system interface that allows the processor to issue requests.*
- *A system event is an event that occurs within the processor and requires access to external system resources.*
- *Sequence refers to the precise series of requests that a processor generates to service a system event.*
- *Protocol refers to the cycle-by-cycle signal transitions that occur on the system interface pins to assert a processor or external request.*
- *Syntax refers to the precise definition of bit patterns on encoded buses, such as the command bus.*

System Interface Description

The RC64475 supports a 64-bit address/data interface that can construct a simple uniprocessor with main memory and can be configured for a 32-bit external address/data interface as well. The RC64474 supports a 32-bit address/data interface only. The System interface for these processors consists of the following buses and signals:

- *64-bit address and data bus,* ***SysAD***
- *8-bit SysAD check bus,* ***SysADC*** *(even parity only)*
- *9-bit command bus,* ***SysCmd***
- *Six handshake signals:*

  RdRdy *, **WrRdy***

  ExtRqst *, **Release***

  ValidIn *, **ValidOut***

The processor uses the system interface to access external resources in order to service processor requests such as cache misses, cache line write-backs, write-through stores and uncached operations.

Interface Buses

Figure 12.1 shows the primary communication paths for the system interface: a 64-bit address and data bus, **SysAD(63:0)**, and a 9-bit command bus, **SysCmd(8:0)**. These **SysAD** and the **SysCmd** buses are bidirectional; that is, they are driven by the processor to issue a processor request, and by the external agent to issue an external request. A request through the system interface consists of:

- *an address*
- *a System interface command that specifies the precise nature of the request*
- *a series of data elements if the request is for a write or read response.*

Notes

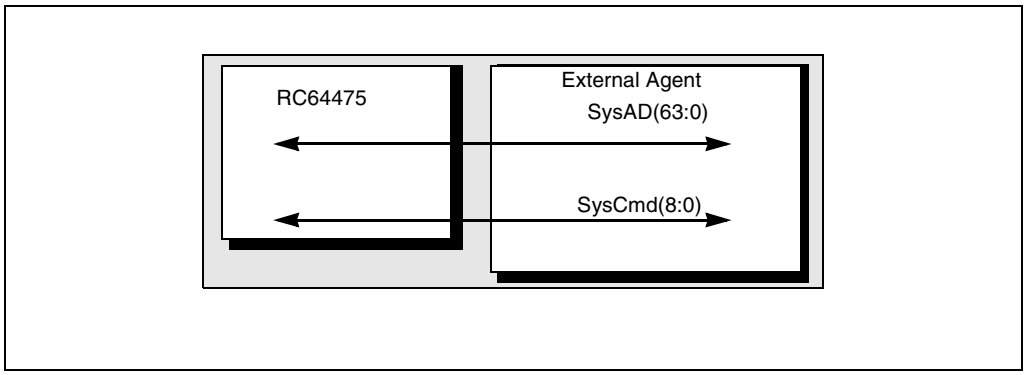


Figure 12.1 System Interface Buses

Address and Data Cycles

Cycles in which the **SysAD** bus contains a valid address are called *address cycles*. Cycles in which the **SysAD** bus contains valid data are called *data cycles*. Validity is determined by the state of the **ValidIn*** and **ValidOut*** signals.

The **SysCmd** bus identifies the contents of the **SysAD** bus during any cycle in which it is valid. The most significant bit of the **SysCmd** bus is always used to indicate whether the current cycle is an address cycle or a data cycle.

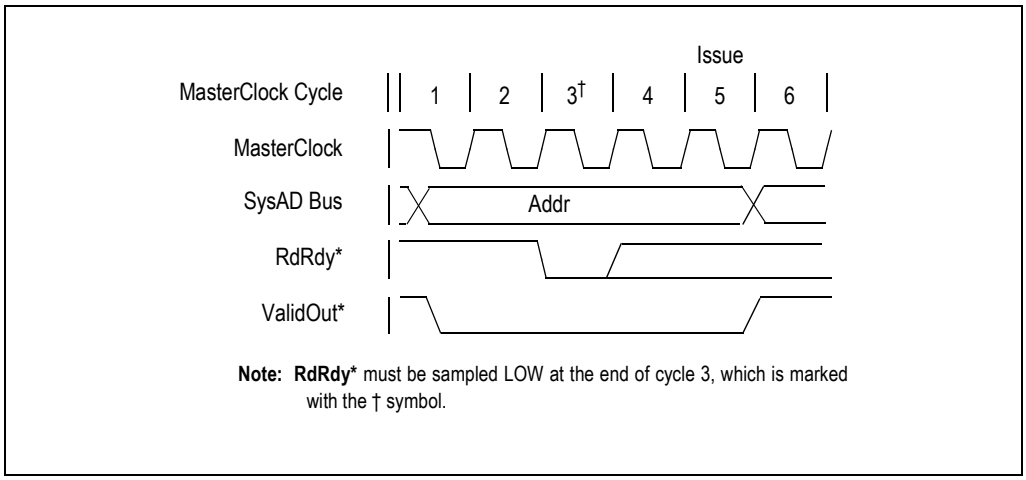
- *During address cycles [**SysCmd(8)** = 0], the remainder of the **SysCmd** bus, **SysCmd(7:0)**, contains a System interface command.*
- *During data cycles [**SysCmd(8)** = 1], the remainder of the **SysCmd** bus, **SysCmd(7:0)**, contains a data identifier .*

Issue Cycles

The issue cycle is defined as the cycle when the external agent can accept the address issued from the processor. There are two types of processor issue cycles:

- *processor read request issue cycles*
- *processor write request issue cycles.*

The processor samples the signal **RdRdy*** to determine the *issue cycle* for a processor read request; the processor samples the signal **WrRdy*** to determine the *issue cycle* of a processor write request. As shown in Figure 12.2, **RdRdy*** must be asserted for one clock cycle, two cycles prior to the address cycle of the processor read request to define the address cycle as the issue cycle (cycle 5 in Figure 12.2). **RdRdy*** does not need to be asserted during the issue cycle.



Note: **RdRdy*** must be sampled LOW at the end of cycle 3, which is marked with the † symbol.

Figure 12.2 State of RdRdy* Signal for Read Requests

Notes

As shown in Figure 12.3, **WrRdy*** must be asserted for one clock cycle, two cycles prior to the first address cycle of the processor write request to define the address cycle as the issue cycle (cycle 5 in Figure 12.3). **WrRdy*** does not need to be asserted during the issue cycle.

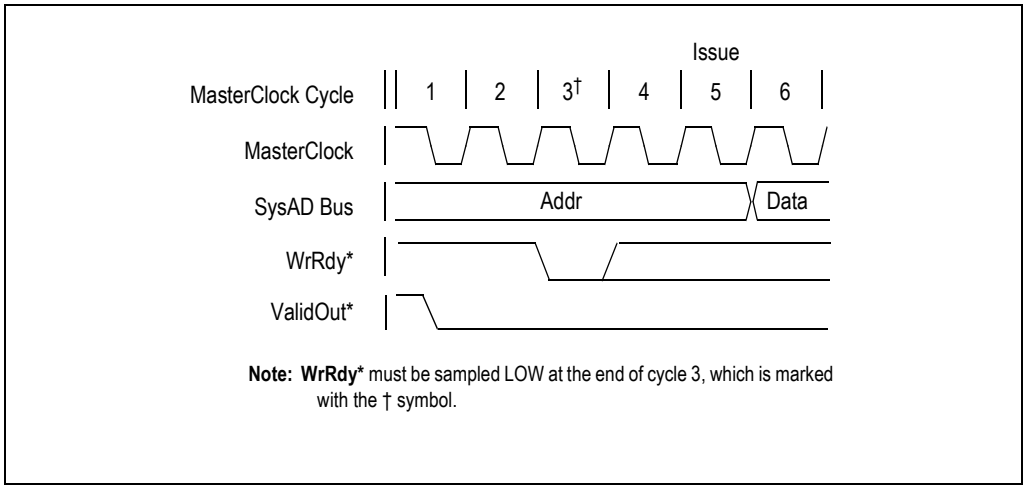


Figure 12.3 State of WrRdy* Signal for Write Requests

The processor repeats the address cycle for the request (that is, asserts the valid address and the **ValidOut*** signal) until the conditions for a valid issue cycle are met. After the issue cycle, if the processor request requires data to be sent, the data transmission begins. There is only one issue cycle for any processor request.

The processor accepts external requests, even while attempting to issue a processor request, by releasing the system interface to slave state in response to an assertion of **ExtRqst*** by the external agent. Note that the rules governing the issue cycle of a processor request are strictly applied to determine the action the processor takes. The processor will either:

- *complete the issuance of the processor request in its entirety before the external request is accepted, or*
- *release the system interface to slave state without completing the issuance of the processor request.*

In the latter case, the processor issues the processor request (provided the processor request is still necessary) after the external request is complete. The rules governing an issue cycle again apply to the processor request.

Handshake Signals

The processor manages the flow of requests through the following six control signals:

- ***RdRdy**, **WrRdy** are used by the external agent to indicate when it can accept a new read (**RdRdy**) or write (**WrRdy**) transaction.*
- ***ExtRqst**, **Release** are used to transfer control of the **SysAD** and **SysCmd** buses. **ExtRqst** is used by an external agent to indicate a need to control the interface. **Release** is asserted by the processor when it transfers the mastership of the system interface to the external agent.*
- *The RC64474/RC64475 processors use **ValidOut** and the external agent uses **ValidIn** to indicate valid command and data on the **SysCmd** and **SysAD** buses.*

Notes

System Interface Protocols

Figure 12.4 illustrates that the system interface operates from register to register. That is, processor outputs come directly from output registers and begin to change with the rising edge of **MasterClock.**[1] Processor inputs are fed directly to input registers that latch these input signals with the rising edge of **MasterClock**. This allows the system interface to run at the highest possible clock frequency.

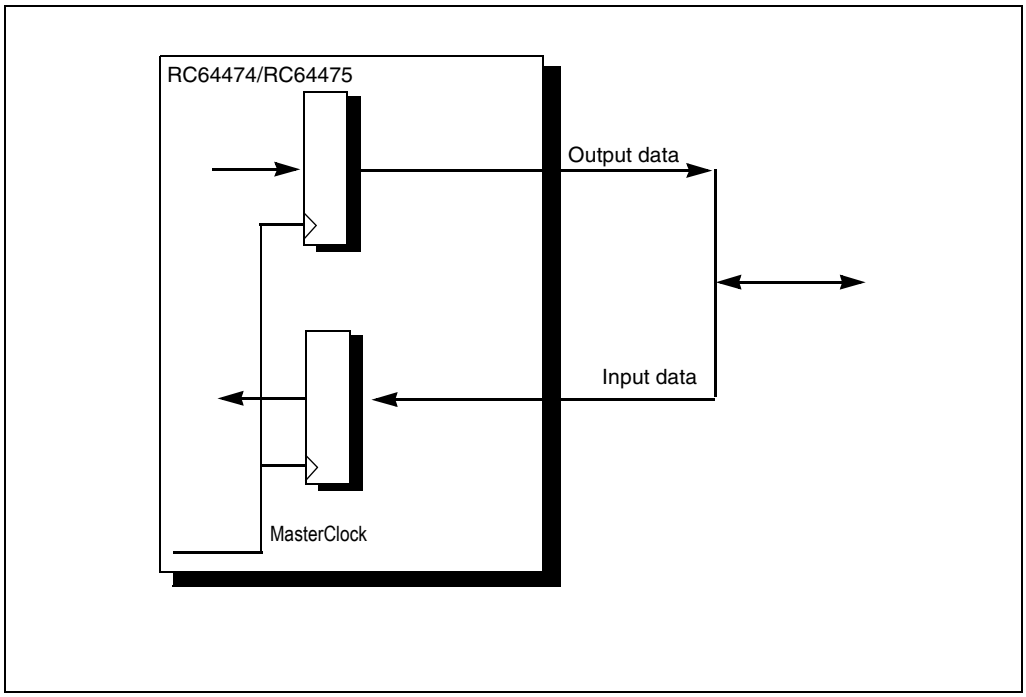


Figure 12.4 System Interface Register-to-Register Operation

Master and Slave States

When the processor is driving the **SysAD** and **SysCmd** buses, the system interface is in *master state*. When the external agent is driving the **SysAD** and **SysCmd** buses, the system interface is in *slave state*. In master state, the processor drives the **SysAD** and **SysCmd** buses and will assert the signal **ValidOut*** whenever the information on these buses is valid. In slave state, the external agent drives the **SysAD** and **SysCmd** buses and asserts the signal **ValidIn*** whenever the information on these buses is valid.

Moving from Master to Slave State

The system interface remains in master state unless one of the following occurs:

- *The external agent requests and is granted the system interface (external arbitration).*
- *The processor issues a read request and performs an uncompelled change to slave state.*

External Arbitration

For the external agent to issue an external request through the system interface, the system interface must be in slave state. The transition from master state to slave state is arbitrated by the processor using the system interface handshake signals **ExtRqst*** and **Release***. This transition is described by the following procedure:

1. An external agent signals that it wishes to issue an external request by asserting **ExtRqst***.
2. When the processor is ready to release bus mastership and accept an external request it asserts **Release*** for one cycle, which releases the system interface from master to slave state.
3. The system interface returns to master state as soon as the external request issue is complete.

[1] **MasterClock** is the input clock to the processor.

Notes

Uncompelled Change to Slave State

An *uncompelled* change to slave state is the transition of the system interface from master state to slave state, initiated by the processor when a processor read request is pending. **Release*** is asserted automatically after a read request. An uncompelled change to slave state occurs during the issue cycle of a read request. After an uncompelled change to slave state, the processor returns to master state at the end of the next external request. This can be a read response, or some other type of external request.

An external agent must note that the processor has performed an uncompelled change to slave state and begin driving the **SysAD** bus along with the **SysCmd** bus. As long as the system interface is in slave state, the external agent can begin a single external request without arbitrating for the system interface; that is, without asserting **ExtRqst***. After the external request, the system interface returns to master state.

Whenever a processor read request is pending, after the issue of a read request, the processor automatically switches the system interface to slave state, even though the external agent is not arbitrating to issue an external request. This transition to slave state allows the external agent to quickly return read response data.

Processor and External Requests

There are two broad categories of requests: *processor requests* and *external requests*. These two categories are described in this section. When a system event occurs, the processor issues either a single request or a series of requests—called *processor requests*—through the system interface, to access an external resource and service the event. For this to work, the processor system interface must be connected to an external agent that is compatible with the system interface protocol, and can coordinate access to system resources. An external agent requesting access to a processor status register generates an *external request*. This access request passes through the system interface. System events and request cycles are shown in Figure 12.5.

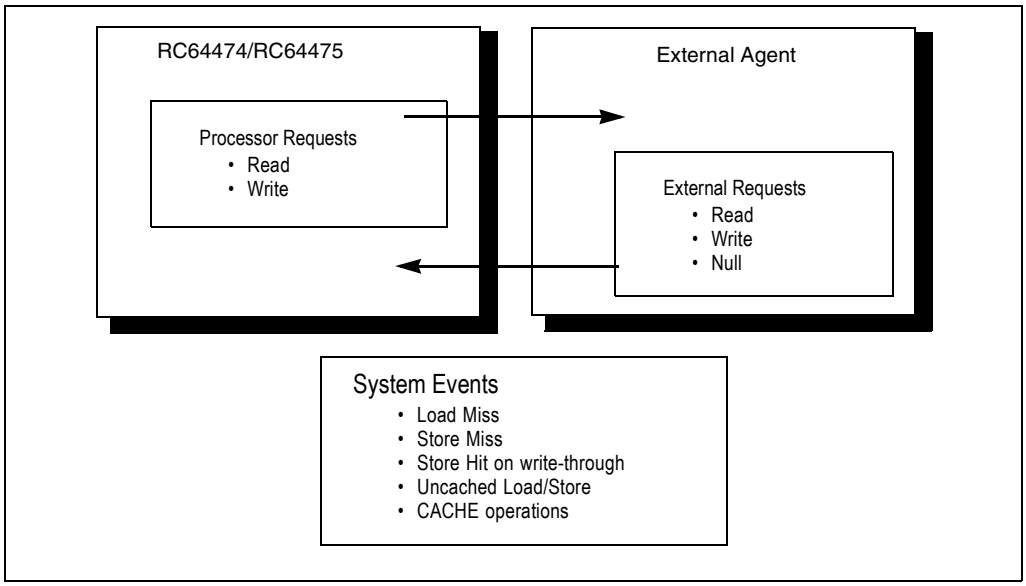


Figure 12.5 Requests and System Events

Processor Request Rules

The following rules apply to processor requests:

- *After issuing a processor read request, the processor cannot issue a subsequent read request until it has received a read response.*
- *After the processor has issued a write request in R4x00 compatible write mode (set at boot time), the processor cannot issue a subsequent request until at least four cycles after the issue cycle of the write request. This means back-to-back write requests with a single data cycle are separated by two unused system cycles, as shown in Figure 12.6.*

Notes

After the processor has issued a write request in either of the two new write modes, write reissue and pipelined writes, the processor can issue a subsequent write immediately provided the **WrRdy*** requirement is met. In Chapter 14, this is discussed in more detail.

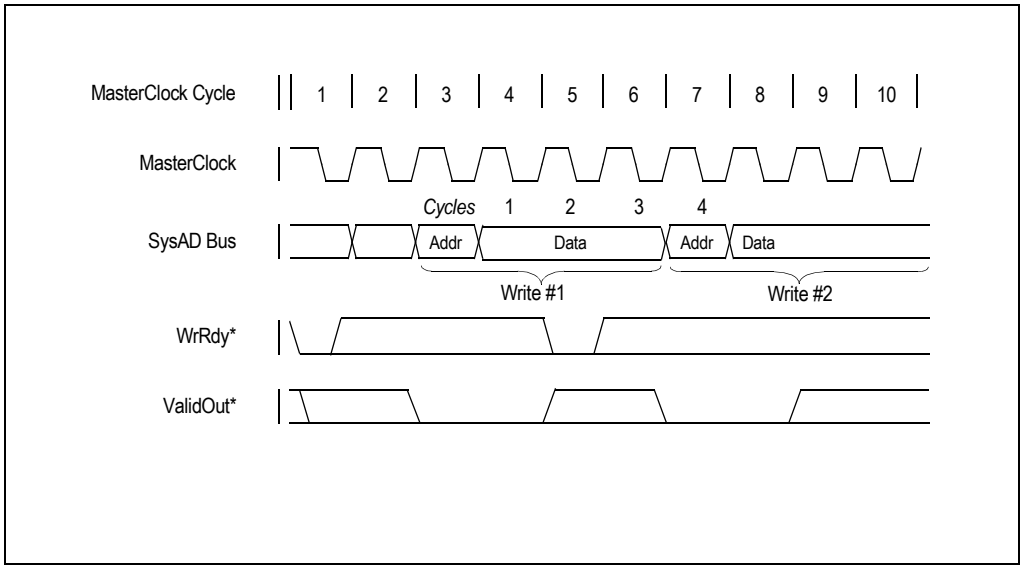


Figure 12.6 Back-to-Back Write Cycle Timing (RISCore4000 family)

Processor Requests

A processor request is a request or a series of requests, through the system interface, to access some external resource. As shown in Figure 12.7, processor requests include only reads and writes.

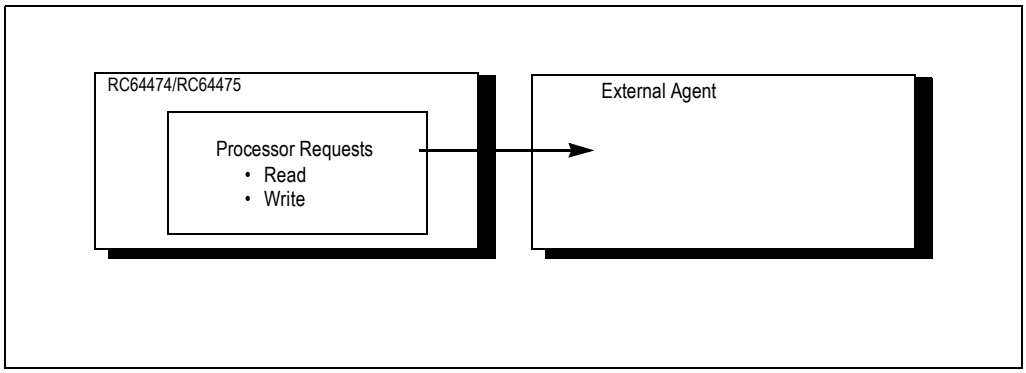


Figure 12.7 Processor Requests

Read request asks for a block, doubleword, partial doubleword, word, or partial word of data either from main memory or from another system resource. *Write request* provides a block, doubleword, partial doubleword, word, or partial word of data to be written either to main memory or to another system resource.

Processor requests are managed by the processor in the *no-secondary-cache mode*. The processor issues requests in a strict sequential fashion; that is, the processor is only allowed to have one request pending at any time. For example, the processor issues a read request and waits for a read response before issuing any subsequent requests. The processor submits a write request only if there are no read requests pending. The processor has the input signals **RdRdy*** and **WrRdy*** to allow an external agent to manage the flow of processor requests. **RdRdy*** controls the flow of processor read requests, while **WrRdy*** controls the flow of processor write requests. The processor request cycle sequence is shown in Figure 12.8.

Notes

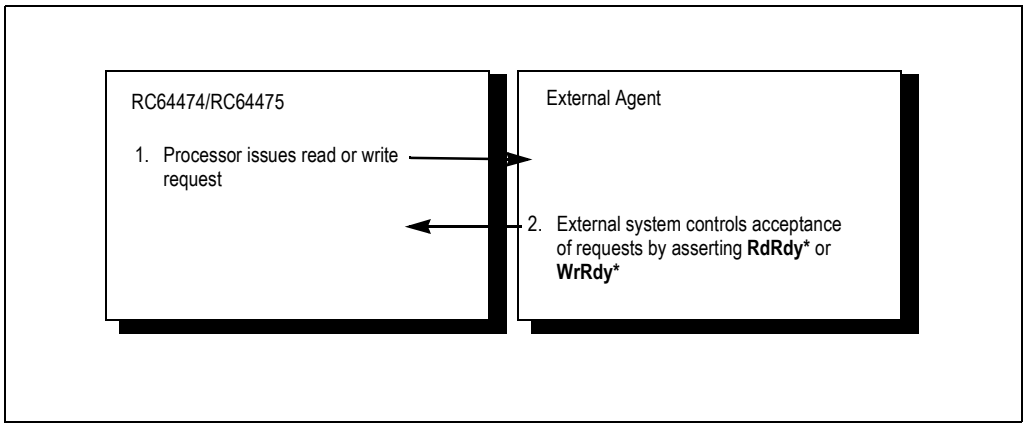


Figure 12.8 Processor Request

Processor Read Request

When a processor issues a read request, the external agent must access the specified resource and return the requested data. A processor read request can be split from the external agent's return of the requested data; in other words, the external agent can initiate an unrelated external request before it returns the response data for a processor read.

A processor read request is completed after the last word of response data has been received from the external agent. Note that the data identifier associated with the response data can signal that the returned data is erroneous, causing the processor to take a bus error. Processor read requests that have been issued, but for which data has not yet been returned, are said to be *pending*. A read remains pending until the requested read data is returned. The external agent must be capable of accepting a processor read request any time the following two conditions are met:

- *There is no processor read request pending.*
- *The signal **RdRdy*** has been asserted for one clock cycle, two cycles before the issue cycle.*

Processor Write Request

When a processor issues a write request, the specified resource is accessed and the data is written to it. A processor write request is complete after the last word of data has been transmitted to the external agent. The external agent must be capable of accepting a processor write request any time the following two conditions are met:

- *No processor read request is pending.*
- *The signal **WrRdy*** has been asserted for one clock cycle, two cycles before the issue cycle.*

Two new modes to enhance the throughput of non-block writes have been added to the RC64474/ RC64475 devices. These modes allow for 2 cycle throughput on back-to-back non-block writes. The actual protocol is discussed in Chapter 13, "The Write Interface." The external agent must be capable of accepting a processor write request in these modes under the same conditions as for the RISCore4000 family (except as explained in Chapter 13, "The Write Interface").

External Requests

External requests include read, write and null requests, as shown in Figure 12.9. This section also includes a description of read response, a special case of an external request.

Notes

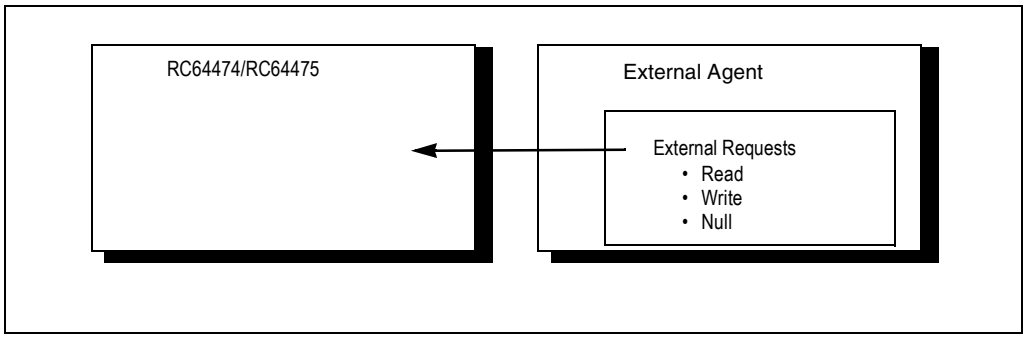


Figure 12.9 External Requests

***Read* request** asks for a word of data from the processor's internal resource. ***Write* request** provides a word of data to be written to the processor's internal resource. ***Null* request** requires no action by the processor; it provides a mechanism for the external agent to return control of the system interface to the master state without affecting the processor.

The processor controls the flow of external requests through the arbitration signals **ExtRqst*** and **Release***, as shown in Figure 12.10. The external agent must acquire mastership of the system interface before it is allowed to issue an external request; the external agent arbitrates for mastership of the system interface by asserting **ExtRqst*** and then waiting for the processor to assert **Release*** for one cycle.

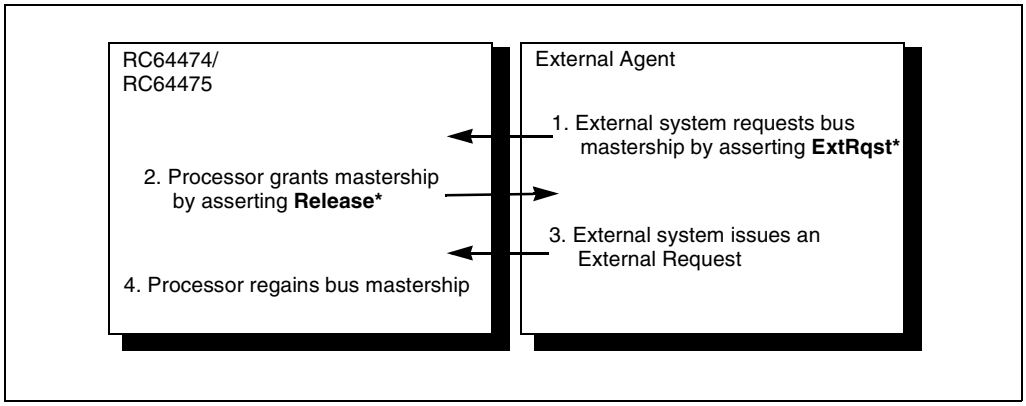


Figure 12.10 External Requests

Mastership of the system interface always returns to the processor after an external request is issued. The processor does not accept a subsequent external request until it has completed the current request. If there are no processor requests pending, the processor decides, based on its internal state, whether to accept the external request, or to issue a new processor request. The processor can issue a new processor request even if the external agent is requesting access to the system interface.

The external agent asserts **ExtRqst*** indicating that it wishes to begin an external request. The external agent then waits for the processor to signal that it is ready to accept this request by asserting **Release***. The processor signals that it is ready to accept an external request based on the criteria listed below.

- *The processor completes any processor request that is in progress.*
- *While waiting for the assertion of **RdRdy*** to issue a processor read request, the processor can accept an external request if the request is delivered to the processor one or more cycles before **RdRdy*** is asserted.*
- *While waiting for the assertion of **WrRdy*** to issue a processor write request, the processor can accept an external request provided the request is delivered to the processor one or more cycles before **WrRdy*** is asserted.*
- *If waiting for the response to a read request after the processor has made an uncompelled change to a slave state, the external agent can issue an external request before providing the read response data.*

Notes

External Read Request

In contrast to a processor read request, data is returned directly in response to an external read request; no other requests can be issued until the processor returns the requested data. An external read request is complete after the processor returns the requested word of data.

The data identifier associated with the response data can signal that the returned data is erroneous, causing the processor to take a bus error.

Note: The RC64474/RC64475 processors do not contain any resources that are readable by an external read request; in response to an external read request the processor returns undefined data and a data identifier with its *Erroneous Data* bit, **SysCmd(5)**, set. Thus, the processor will take a bus error at the completion of the external read request.

External Write Request

When an external agent issues a write request, the specified resource is accessed and the data is written to it. An external write request is complete after the word of data has been transmitted to the processor. The only processor resource available to an external write request is the IP field of the Cause register.

System Interface Endianness

The endianness of the system interface is programmed at boot time through the boot-time mode control interface (see Chapter 9, “Initialization Interface” for specifics), and remains fixed until the next time the processor boot-time mode bits are read. Software cannot change the endianness of the system interface and the external system; software can set the reverse endian bit to reverse the interpretation of endianness inside the processor, but the endianness of the system interface remains unchanged.

System Interface Cycle Time

The processor specifies minimum and maximum cycle counts for various processor transactions and for the processor response time to external requests. Processor requests themselves are constrained by the system interface request protocol, and request cycle counts can be determined by examining the protocol. The following system interface interactions can vary within minimum and maximum cycle counts:

- *waiting period for the processor to release the system interface to slave state in response to an external request (release latency)*
- *response time for an external request that requires a response (external response latency).*

The remainder of this section describes and tabulates the minimum and maximum cycle counts for these system interface interactions.

Release Latency

Release latency is generally defined as the number of cycles the processor can wait to release the system interface to slave state for an external request. When no processor requests are in progress, internal activity can cause the processor to wait some number of cycles before releasing the system interface. Release latency is therefore more specifically defined as the number of cycles that occur between the assertion of **ExtRqst*** and the assertion of **Release***. There are three categories of release latency:

- *Category 1: When the external request signal is asserted two cycles before the last cycle of a processor request.*
- *Category 2: When the external request signal is not asserted during a processor request, or is asserted during the last cycle of a processor request.*
- *Category 3: When the processor makes an uncompelled change to slave state.*

Table 12.1 summarizes the minimum and maximum release latencies for requests that fall into categories 1, 2 and 3.

Note: The maximum and minimum cycle count values are subject to change.

Notes

Category	Minimum PCycles	Maximum PCycles
1	4	6
2	4	24
3	0	0

Table 12.1 Release Latency for External Requests

The differences in the minimum and maximum times are due to internal conditions not readily observable externally. The relationship between **PClock** and **MasterClock** will dictate when the **Release*** signal is seen externally.

64-bit System Interface Addresses

System interface addresses are full 36-bit physical addresses presented on the least-significant 36 bits (bits 35 through 0) of the **SysAD** bus during address cycles; the remaining bits of the **SysAD** bus are unused during address cycles.

Addressing Conventions for 64-bit Wide Interface

Addresses associated with doubleword, partial doubleword, word, or partial word transactions, are aligned for the size of the data element. The system uses the following address conventions:

- *Addresses associated with block requests are aligned to double-word boundaries; that is, the low-order 3 bits of address are 0.*
- *Doubleword requests set the low-order 3 bits of address to 0.*
- *Word requests set the low-order 2 bits of address to 0.*
- *Halfword requests set the low-order bit of address to 0.*
- *Byte, tribyte, quintibyte, sextibyte, and septibyte requests use the byte address.*

32-bit System Interface Addresses

System interface addresses are 32-bit physical addresses presented on the least-significant 32 bits (bits 31 through 0) of the **SysAD** bus during address cycles; the remaining bits of the **SysAD** bus are unused during address cycles. Internal address bits above Addr(31) are truncated in 32-bit mode.

Addressing Conventions for 32-bit Wide Interface

Addresses associated with doubleword, partial doubleword, word, or partial word transactions, are aligned for the size of the data element. The system uses the following address conventions:

- *Addresses associated with block requests are aligned to word boundaries; that is, the low-order 2 bits of address are 0.*
- *Word requests set the low-order 2 bits of address to 0.*
- *Halfword requests set the low-order bit of address to 0.*
- *Byte and tribyte requests use the byte address.*

The Read Interface

Notes

Introduction

When a processor issues a read request, the external agent must access the specified resource and return the requested data. A processor read request can be split from the external agent's return of the requested data; in other words, the external agent can initiate an unrelated external request before it returns the response data for a processor read. A processor read request is completed after the last word of response data has been received from the external agent. Note that the data identifier associated with the response data can signal that the returned data is erroneous, causing the processor to take a bus error. Processor read requests that have been issued, but for which data has not yet been returned, are said to be *pending*. A read remains pending until the requested read data is returned.

The external agent must be capable of accepting a processor read request any time the following two conditions are met:

- *There is no processor read request pending.*
- *The signal **RdRdy*** has been asserted for one clock cycle, two cycles before the issue cycle.*

Read Response

A *read response* returns data in response to a processor read request, as shown in Figure 13.1. While a read response is technically an external request, it has one characteristic that differentiates it from all other external requests—it does not perform system interface arbitration. For this reason, read responses are handled separately from all other external requests, and are simply called read responses. When a read response comes back with bad parity for the first data, a cache error exception results.

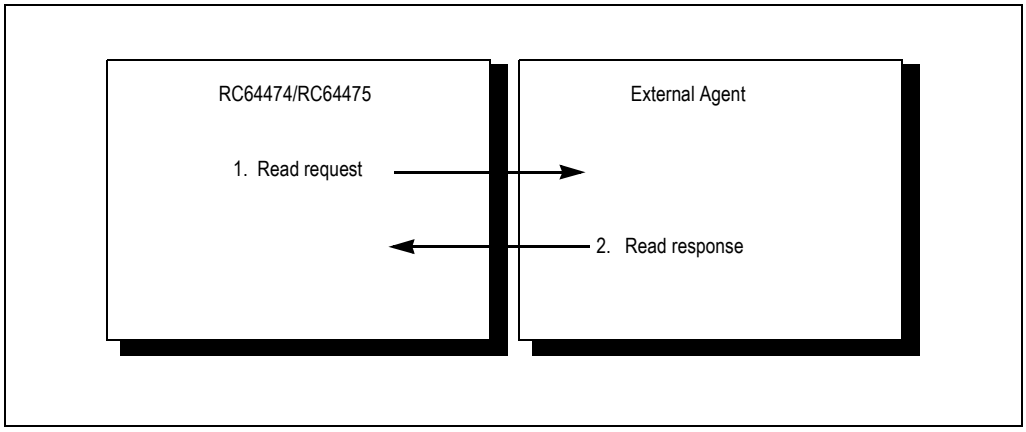


Figure 13.1 Read Response

Handling Requests

This section details the *sequence*, *protocol*, and *syntax* of both processor and external requests. The following system events are discussed:

- *load miss*
- *store miss*
- *store hit*
- *uncached loads/stores*
- *CACHE operations*
- *load linked store conditional*

Notes

Load Miss

When a processor load misses in the primary cache, before the processor can proceed it must obtain the cache line that contains the data element to be loaded from the external agent. If the new cache line replaces a current cache line with a W bit set, the current cache line must be written back. The processor examines the coherency attribute in the CAlg register for the memory region that contains the requested cache line, and executes a noncoherent read request; the coherency attribute is *noncoherent.* Table 13.1 shows the actions taken on a load miss to primary cache.

Page Attribute	State of Data Cache Line Being Replaced	
	Clean/Invalid	Dirty (W=1)
Noncoherent	NCR	NCR/W
NCR Processor noncoherent block read request NCR/W Processor noncoherent block read request followed by processor block write request		

Table 13.1 Load Miss to Primary Cache

If the cache line must be written back on a load miss, the read request is issued and completed before the write request is handled. The processor takes the following steps:

1. The processor issues a noncoherent read request for the cache line that contains the data element to be loaded.
2. The processor then waits for an external agent to provide the read response.
3. The processor will restart the pipeline after the first doubleword (the data that missed is fetched first). The rest of the data cache line will be placed into the cache in parallel.

If the current cache line must be written back, the processor issues a write request to save the dirty cache line in memory. In 64-bit bus mode a block transfer (read or write) is equivalent to 4 data transfer to/from the memory. In 32-bit mode a block transfer (read or write) is equivalent to 8 data transfer to/from the memory.

Store Miss

When a processor store misses in the primary cache, the processor may request, from the external agent, the cache line that contains the target location of the store for pages that are either write-back or write-through with write-allocate only. The processor examines the coherency attribute in the CAlg register for the memory region that contains the requested cache line to see if the line is write-allocate or no-write-allocate. The processor then executes one of the following requests:

- *If the coherency attribute is noncoherent, write-back or noncoherent, write-through with write-allocate, a noncoherent block read request is issued.*
- *If the coherency attribute is noncoherent, write-through with no write-allocate, the processor issues a non-block write request.*
- *Table 13.2 shows the actions taken on a store miss to the primary cache.*

Page Attribute	State of Data Cache Line Being Replaced	
	Clean/Invalid	Dirty (W=1)
Noncoherent, write-back or Noncoherent, write-through with write-allocate	NCR	NCR/W
Noncoherent, write-through with no write-allocate	NCW	NA
Table Legend: NCR Processor noncoherent block read request NCR/W Processor noncoherent block read request followed by processor block write request NCW Processor noncoherent write request		

Table 13.2 Store Miss to Primary Cache

Notes

If the coherency attribute is write-back or write-through with write-allocate, the processor issues a read request for the cache line that contains the data element to be loaded, then waits for the external agent to provide read data in response to the read request. Then, if the current cache line must be written back, the processor issues a write request for the current cache line. For a write-through, no write-allocate store miss, the processor issues a write request only.

If the new cache line replaces a current cache line whose *Write back* (*W*) bit is set, the current cache line moves to an internal write buffer before the new cache line is loaded in the primary cache. In 64-bit bus mode a block transfer (read or write) is equivalent to 4 data transfer to/from the memory. In 32-bit mode a block transfer (read or write) is equivalent to 8 data transfer to/from the memory.

Store Hit

This section describes store hits in no-secondary-cache mode for both write-back and write-through lines. The action on the system interface will be determined by whether the line is write-back or write-through. All lines that use a write-back policy are set to the dirty exclusive cache state and there is no bus transaction generated. For lines with a write-through policy, the store will also generate a processor write request for the store data. In 64-bit bus mode this is equivalent to 4 data transfer to the memory. In 32-bit mode this is equivalent to 8 data transfer to the memory.

Uncached Loads

When the processor performs an uncached load, it issues a noncoherent word read request (the actual access can be for a doubleword, word, partial word or byte, but the request is called a word read request to differentiate it from the block read request).

In 64-bit mode the CPU expects valid parity and data in the full **SysAD** bus (all 64 bits), even if it is looking for less than a double word. If a partial word is returned the correct parity for the full 64-bit must be returned, or the CPU must be informed not to check parity. In 32-bit bus mode the CPU expects valid parity and data in the full **SysAD** bus (all 32 bits), even if it is looking for less than a word. If a partial word is returned the correct parity for the full 32-bit must be returned, or the CPU must be informed not to check parity.

All writes by the processor will be buffered from the system interface by the 4-deep write buffer. The write requests are sent to the system interface when there are no other requests in progress. If the write buffer contains any entries when a block request is needed, the write buffer is first flushed before any read request will occur (cache miss or uncached load). Both a data cache miss and an uncached data load will flush the write buffer.

CACHE Operations

The processor provides a variety of CACHE operations to maintain the state and contents of the primary cache. During the execution of the CACHE operation instructions, the processor can issue write or read requests.

Load Linked/Store Conditional Operation

Generally, the execution of a Load Linked/Store Conditional instruction sequence is not visible at the system interface; that is, no special requests are generated due to the execution of this instruction sequence.

However, there is one situation in which the execution of a Load Linked/Store Conditional instruction sequence is visible, as indicated by the *link address retained* bit during a processor read request, as programmed by the **SysCmd(2)** bit. This occurs when the data location targeted by a Load-Linked-Store-Conditional instruction sequence maps to the same cache line to which the instruction area containing the Load Linked/Store Conditional code sequence is mapped. In this case, immediately after executing the Load Linked instruction, the cache line that contains the link location is replaced by the instruction line containing the code. The link address is kept in a register separate from the cache, and remains active as long as the *link* bit, set by the Load Linked instruction, is set.

Notes

The *link* bit, which is set by the load linked instruction, is cleared by a change of cache state for the line containing the link address, or by a Return From Exception. For more information, refer to Chapter 11, or see the specific Load Linked and Store Conditional instructions described in the *IDT MIPS Microprocessor Family Software Reference Manual*.

Processor Read Protocols

The following sections contain a cycle-by-cycle description of the bus arbitration protocols for the processor read request. Table 13.3 lists the abbreviations and definitions for each of the buses used in the timing diagrams that follow.

Scope	Abbreviation	Meaning
Global	Unsd	Unused
SysAD bus	Addr	Physical address
	Data<n>	Data element number n of a block of data
SysCmd bus	Cmd	An unspecified system interface command
	Read	A processor or external read request command
	Write	A processor or external write request command
	SINull	A system interface release external null request command
	NData	A noncoherent data identifier for a data element other than the last data element
	NEOD	A noncoherent data identifier for the last data element

Table 13.3 System Interface Requests

Processor Read Request

In the timing diagrams in this section note that the two closely spaced, wavy vertical lines (for example, MasterClock Cycle 2 in Figure 13.5 on page 13-8) indicate one or more identical cycles.

Processor Read Request Protocol Steps

The following sequence describes the protocol for a processor read request. This protocol is the same for either 32-bit bus mode or 64-bit bus mode. The numbered steps in this list correspond to the numbers in Figure 13.2.

1. **RdRdy*** is asserted low, indicating the external agent is ready to accept a read request.
2. With the system interface in master state, a processor read request is issued by driving a read command on the **SysCmd** bus and a read address on the **SysAD** bus.
3. At the same time, the processor asserts **ValidOut*** for one cycle, indicating valid data is present on the **SysCmd** and the **SysAD** buses.

   Note: Only one processor read request can be pending at a time. **ValidOut*** is asserted every time the CPU is driving valid information on **SysAD** and **SysCmd** bus. In the case of read request, this means as long as the address is driven and will be deasserted at the end of the bus cycle.
4. The processor makes an uncompelled change to slave state at the issue cycle of the read request by asserting the **Release*** signal for one cycle.

   Note: The external agent must not assert the signal **ExtRqst*** for the purposes of returning a read response, but rather must wait for the uncompelled change to slave state. The signal **ExtRqst*** can be asserted before or during a read response to perform an external request other than a read response.
5. The processor releases the **SysCmd** and the **SysAD** buses one **MasterClock** cycle after the assertion of **Release***.
6. The external agent drives the **SysCmd** and the **SysAD** buses within two cycles after the assertion of **Release***.

Notes

Once in slave state (starting at cycle 5 in Figure 13.2), the external agent can return the requested data through a read response. The read response can return the requested data or, if the requested data could not be successfully retrieved, an indication that the returned data is erroneous. If the returned data is erroneous, the processor takes a bus error exception.

Note: For read response data the processor only checks the error bits for the first doubleword in 64-bit bus mode, and the first word in 32-bit bus mode. All other error bits are ignored. **WrRdy*** is not checked during processor read requests.

Figure 13.2 illustrates a processor read request, coupled with an uncompelled change to slave state.

Note: Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

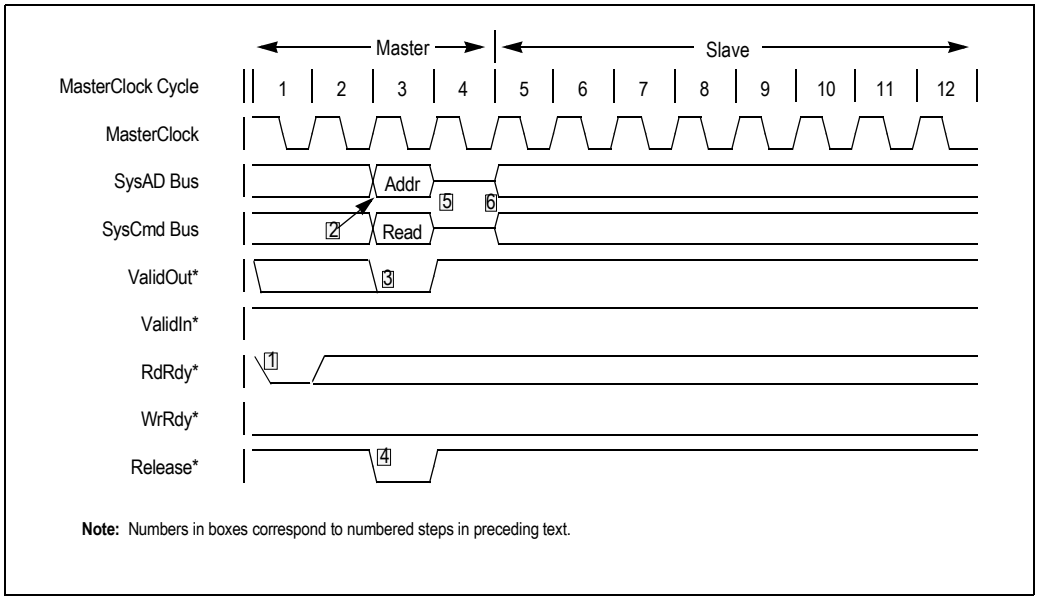


Figure 13.2 Processor Read Request Protocol

The assertion of **Release*** indicates either an uncompelled change to slave state, or a response to the assertion of **ExtRqst***, whereupon the processor accepts either a read response, or any other external request. If any external request other than a read response is issued, the processor performs another uncompelled change to slave state after processing the external request by asserting release for one clock cycle. The actual read response, where the external agent returns the requested data, is shown later in this chapter.

External Instruction Read Response Time

The RC64474/RC64475 accesses the external bus due to instruction cache miss or an uncached reference. The length of time for an external read is based on the overhead at the beginning and end of the read along with the time to drive the address and get the response data.

Instruction Read Latency Steps for System Clock

The read latency for a system clock in the multiply-by-two mode is as follows:

1. The startup overhead is one to two pipeline cycles (PCycle) for the CPU to transfer the address to the pads to be output. The second PCycle is needed if the miss is detected on a PCycle not aligned with the rising edge of **MasterClock**.
2. The CPU drives the address on the **SysAD** bus for two PCycles.
3. The CPU tri-states the **SysAD** bus for two PCycles.
4. The CPU waits for the main memory to return the data. This is expressed as *n* x 2 PCycles.
5. The first double word is driven in the **SysAD** from the main memory for two PCycles.
6. The remaining three double words of instruction are driven on **SysAD** for 3*2 PCycles.

Notes

Note That:

– *For instruction misses, the pipeline starts after all the instructions are returned.*
– *n is the total number of idle cycles (even between double word instruction). For zero wait state systems, n = 0.*

Example of Instruction Block Read with Zero Wait-State

Table 13.4 shows an instruction block read with a zero wait state (n=0):

Step	Description	PCycles
1	CPU overhead for cache miss detection	1-2
2	Address driven on **SysAD** bus	2
3	**SysAD** bus tri-stated	2
4	Memory latency to return the data (nx2)	0*2
5	First double word driven on SysAD bus	2
6	Remaining three instructions returned	2*3=6
	Total PCycles:	13-14

Table 13.4 Steps for Single Read With Zero Wait-State

External Data Read Response Time

The RC64474/RC64475 access the external bus due to data cache miss or an uncached reference. The length of time for an external read is based on the overhead at the beginning and end of the read along with the time to drive the address and get the response data.

Data Read Latency Steps for System Clock

The read latency for a system clock that is in the multiply-by-two mode is as follows:

1. The startup overhead is one to two pipeline cycles (PCycle) for the CPU to generate the parity for the address to be output. The second PCycle is needed if the miss is detected or a PCycle not aligned with the rising edge of SClock.
2. The CPU drives the address on the **SysAD** bus for two PCycles.
3. The CPU tri-states the **SysAD** bus for two PCycles.
4. The CPU waits for the main memory to return the data. This is expressed as *n* x 2 PCycles where *n* is the number of MasterClock cycles for the first data to be returned in a block read, or the latency for the single read. For zero wait state memory system *n* should be zero.
5. The first double word is driven in the **SysAD** from the main memory for two PCycles.
6. The end of the overhead is two PCycles: one to transfer the data from the pads and generate the parity, and one to write to the register (or cache, if it is cacheable data).

Note the Following:

- *If n=0 and the line being replaced is dirty, the CPU takes one to two additional PCycles of overhead to move the dirty data into the write buffer.*
- *The additional latency for returning the remaining three data elements should be added in a manner similar to the instruction read latency.*
- *If cache line needs to be written back, the read request is posted first and then the write is completed.*

Example of Data Single Read with Zero Wait-State

Table 13.5 shows a data block read with a zero wait state (n=0):

Notes

Step	Description	PCycles
1	CPU overhead for cache miss detection	1-2
2	Address driven on **SysAD** bus	2
3	**SysAD** bus tri-stated	2
4	Memory latency to return the data (nx2)	0*2
5	First double word driven on **SysAD** bus	2
6	CPU overhead to write the data cache, do the fixup, and then restart	2
	Total PCycles:	9-10

Table 13.5 Steps for Data Block Read With Zero Wait-State

External Cycles for Read Latency

The external cycles to get the response data will look similar to 13.3. For a larger “multiply-by” it will take longer to get the response data.

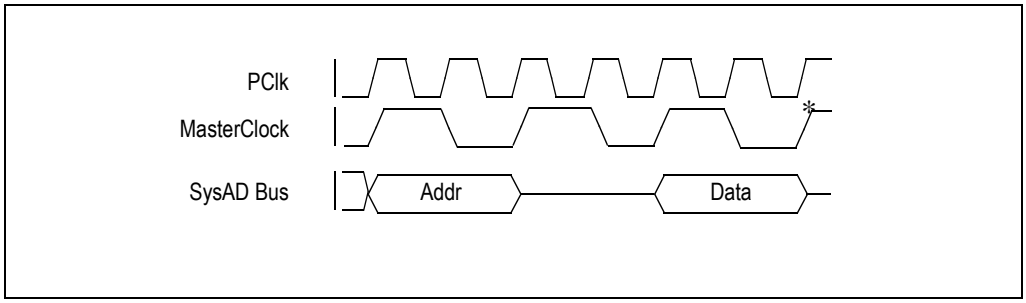


Figure 13.3 Uncached Read—External Cycles

The same operation is shown in greater detail in Figure 13.4. These figures assume the following:

- *Data is returned immediately after **Release*** is asserted, and after the bus turnaround cycle (when the CPU tri-states the bus to allow the external agent to drive it).*
- *The data meets the setup and hold requirements for the rising edge of **MasterClock** that is identified in the preceding and following figures with an asterisk.*

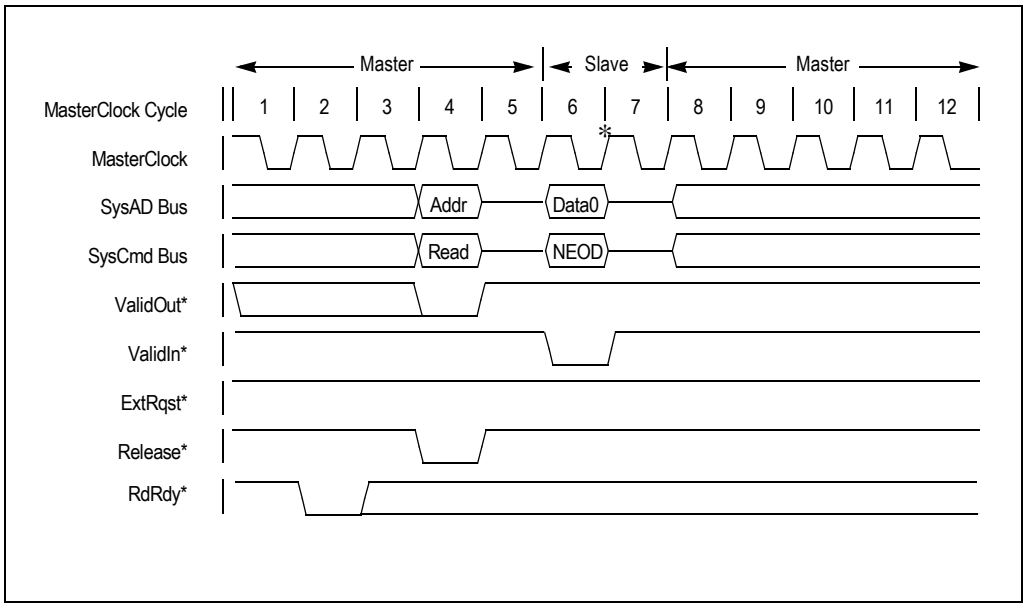


Figure 13.4 Processor Read Cycle

Notes

Read Response Protocol

An external agent must return data to the processor in response to a processor read request by using a read response protocol. A read response protocol consists of the following steps:

1. The external agent waits for the processor to perform an uncompelled change to slave state.
2. The external agent returns the data through a single data cycle or a series of data cycles.
3. After the last data cycle is issued, the read response is complete and the external agent sets the **SysCmd** and **SysAD** buses to a tri-state.
4. The system interface returns to master state.

   Note: The processor always performs an uncompelled change to slave state in the same cycle that it issues a read request.
5. The data identifier for data cycles must indicate the fact that this data is *response data*.
6. The data identifier associated with the last data cycle must contain a *last data cycle* indication.

For read responses to non-coherent block read requests (which is the only read request for normal operations of the RC64474/RC64475,) the response data will not need to identify an initial cache state. The cache state will automatically be assigned as dirty exclusive by the processor.

The data identifier associated with a data cycle can indicate that the data transmitted during that cycle is erroneous; however, an external agent must return a data block of the correct size regardless of the fact that the data may be in error. The processor only checks the error bit for the first data of a block, while the other error bits for the block of data are ignored. If an initial erroneous data cycle is detected, the processor takes a bus error at the completion of the data transfer.

Read response data must only be delivered to the processor when a processor read request is pending. The behavior of the processor is undefined when a read response is presented to it and there is no processor read pending.

Figure 13.5 illustrates a processor word read request followed by a word read response. Figure 13.6 illustrates a read response for a processor block read with the system interface already in slave state. Figure 13.7 illustrates a block read transaction with one wait state.

Note: Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

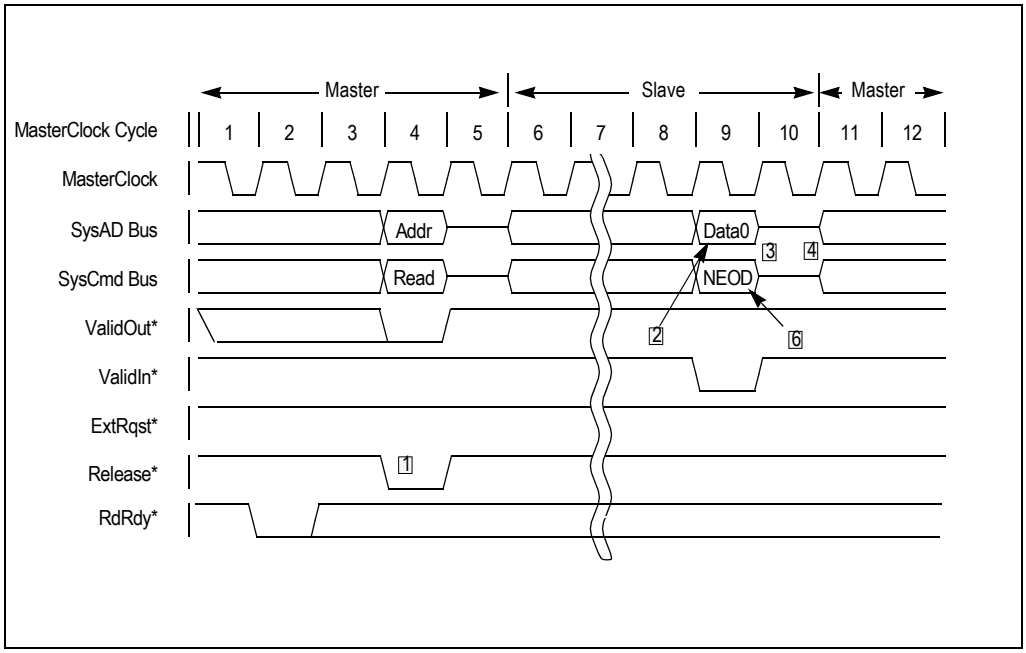


Figure 13.5 Processor Word Read Request Followed by a Word Read Response (64-bit bus interface)

Notes

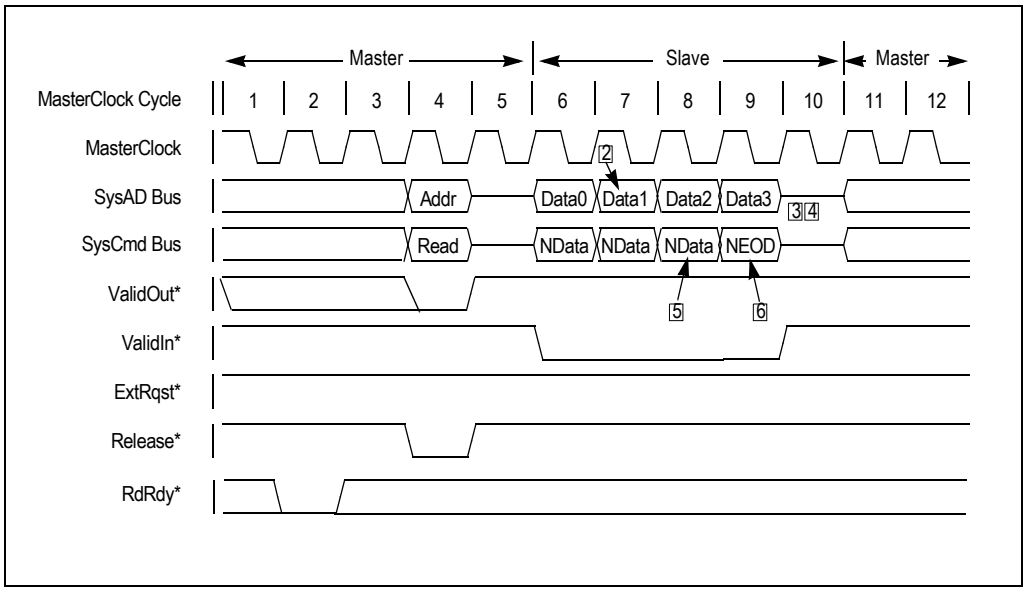


Figure 13.6 Block Read Response With Zero Wait-State (64-bit bus interface)

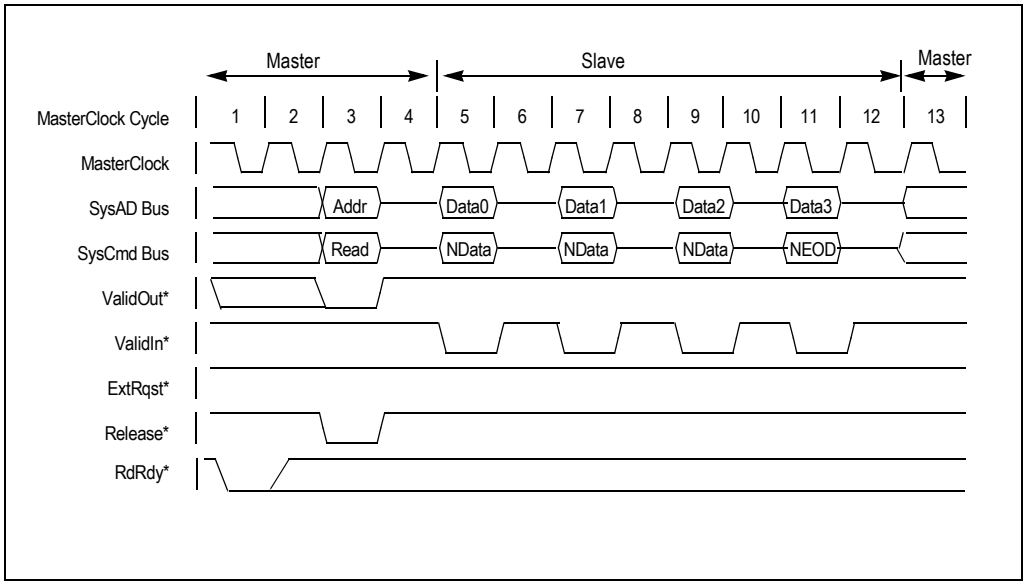


Figure 13.7 Block Read Transaction With One Wait-State (64-bit bus interface)

Data Rate Control

The system interface supports a maximum data rate of one doubleword per cycle in 64-bit bus mode and one word per cycle in 32-bit bus mode. The data rate the processor can support is directly related to the rate at which the external agent can return data.

Read Data Pattern

The rate at which data is delivered to the processor can be determined by the external agent—for example, the external agent can drive data and assert **ValidIn*** every *n* cycles, instead of every cycle. An external agent can deliver data at any rate it chooses, but must not deliver data to the processor any faster than the processor is capable of receiving it.

The processor only accepts cycles as valid when **ValidIn*** is asserted and the **SysCmd** bus contains a data identifier. If the external agent sends more data items then requested (e.g., a fifth doubleword of read response data with **ValidIn*** asserted in 64-bit bus mode) or the last data (i.e., the fourth doubleword in 64-

Notes

bit bus mode) of a block read is not tagged as the last data item, it is an error and the resulting actions of the processor for these cases will be undefined. Figure 13.8 shows a read response with reduced data rate and with the system interface in slave state.

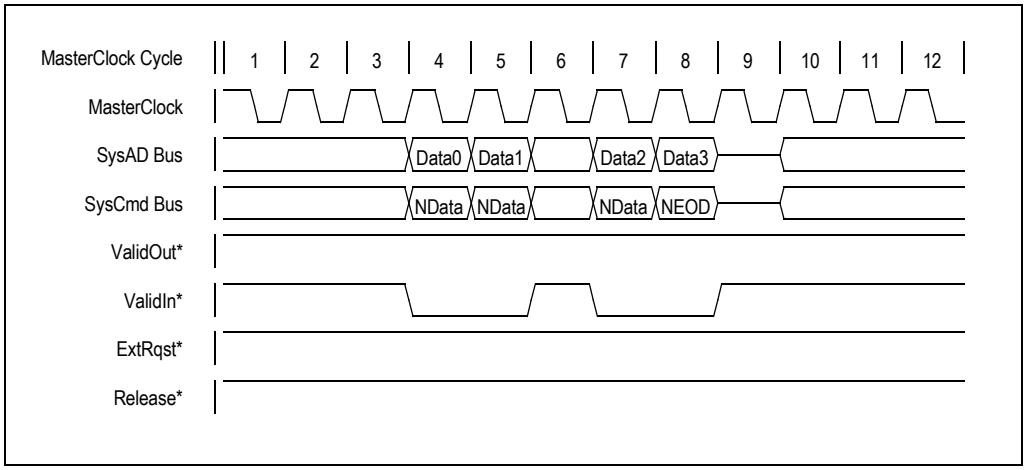


Figure 13.8 Read Response, Reduced Data Rate, System Interface in Slave State (64-bit bus interface)

64-bit & 32-bit Bus Modes

The bus interface of the RC64475 can be configured during reset to be either 64-bit wide or 32-bit wide. The same bus protocol explained earlier in this chapter applies for both modes. In 32-bit bus mode, the internal execution core is still a full 64-bit engine. Only the bus interface unit can be configured as either 64-bit or 32-bit interface. The bus width mode is a static feature of the device. This means that the bus width has to be configured once during reset. This feature should not be thought of as dynamic bus width interface where the bus width is 64-bit in one access and 32-bit wide in the other access.

64-bit Bus Mode

In 64-bit bus mode, the RC64475 supports 64-bit address/data system interface that consists of:

- *64-bit address and data,* ***SysAD(63:0)***
- *8-bit SysAD check bus,* ***SysADC(7:0)*** *(even parity)*
- *9-bit command bus,* ***SysCmd(8:0)***
- *Six handshake signals:*
  RdRdy*,* ***WrRdy****
  ExtReq*,* ***Release****
  ValidIn*,* ***ValidOut****

64-bit Bus Mode block Read Operation

In 64-bit bus mode, the RC64475 issues a single block read request for the entire cache line (4 double words). The external agent should return all four double words as explained in the read protocol section earlier. Figure 13.9 illustrates the timing diagram for a block read operation in 64-bit bus mode. The address issued by the RC64475 is double word (64-bit) aligned.

Notes

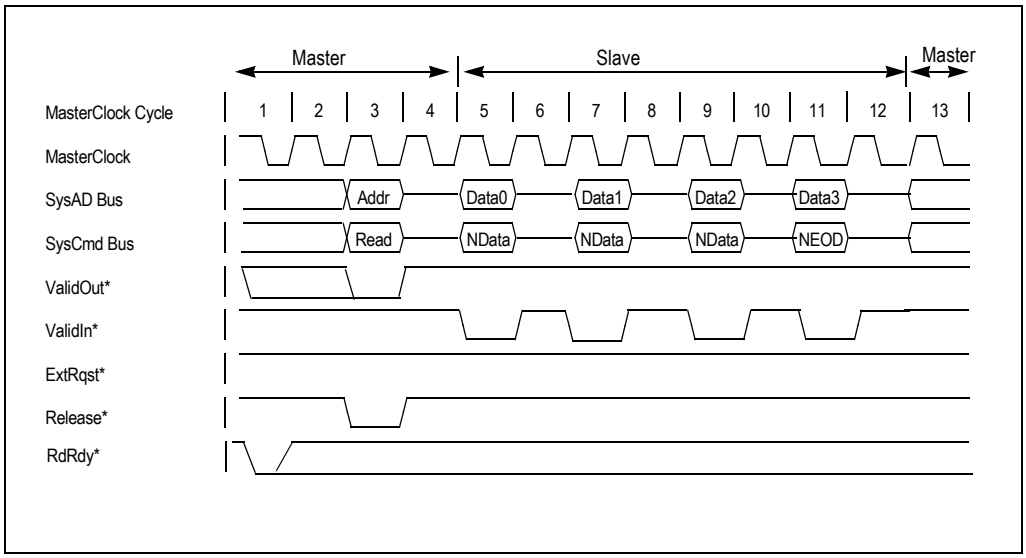


Figure 13.9 Block Read Transaction With One Wait-State

64-bit Bus Mode Single (Uncached) Read Operation

In 64-bit bus mode, the RC64475 issues a single uncached read request using a doubleword (64-bit) aligned address. The actual access can be for a doubleword, word, partial word, or byte, but the request is called a word read request to differentiate it from the block read request. Figure 13.10 illustrates the timing for an uncached read operation.

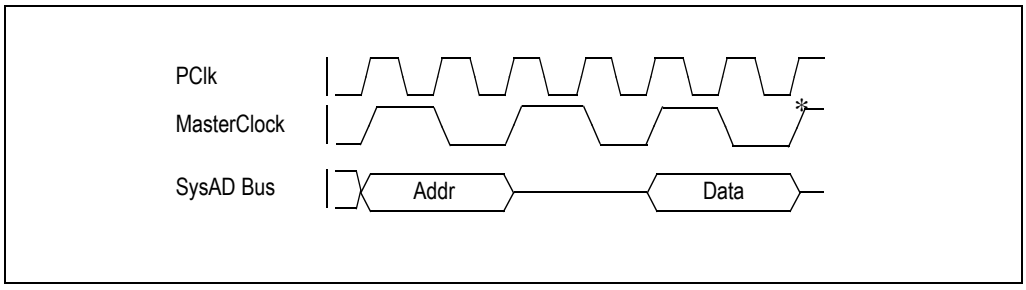


Figure 13.10 64-Bit Uncached Read—External Cycles

32-bit Bus Mode

In 32-bit bus mode, the RC64474/RC64475 support a 32-bit address/data system interface that consists of the following:

- *The 32-bit address & data (**SysAD** (31:0)) and the 4-bit **SysAD** check bus (**SysADC** (3:0), even parity). **SysAD** (63:32) and **SysADC** (7:4) are undefined.*
- *9-bit command bus, **SysCmd(8:0)***
- *Six handshake signals:*

  RdRdy*, **WrRdy***

  ExtReq*, Release*

  ValidIn*, ValidOut*

It is important to note that in the 32-bit bus mode **SysAd(31:0)** and **SysADC(3:0)** are always used regardless of the Endianness of the system. It is also important to note that the encoding of **SysCmd(8:0)** is the same for both 64-bit and 32-bit bus modes. This means that the RC64474/RC64475 does not inform the external agent about the bus width mode. It is expected that this mode is programmed during reset and that the external agent is configured to interface to the RC64474/RC64475 in either 64-bit or 32-bit bus mode.

Notes

32-bit Bus Mode Block Read Operation

In 32-bit bus mode, the RC64474 or RC64475 will issue a single block read request for the entire cache line (4 double words). since the bus interface is configured to be 32-bit wide, the processor issues a single address that is word (32-bit) aligned. The external agent should return 8 single words, as explained in the read protocol section.

Figure 13.11 illustrates the timing diagram for a block read operation in 32-bit bus mode. This means that a block read request is not divided into two requests. The external agent is responsible for returning all 8 single words to the RC64474 or RC64475 .

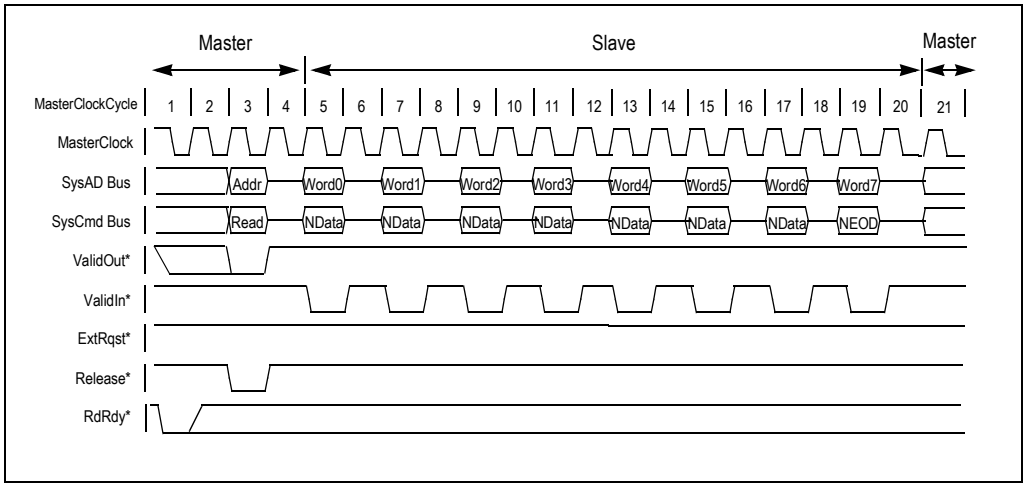


Figure 13.11 Block Read Transaction With One Wait-State

The RC64474/RC64475 combine the words internally to generate a double word data to be used by the execution core. This implies that the order of the words in a double word data will be endian-dependent. On little-endian machines bits 31:0 will be transferred first and bits 63:32 transferred second; on a big- endian machine the order will be reversed.

32-bit Bus Mode Single (Uncached) Read Operation

In 32-bit bus mode, the RC64475 or the RC64474 will issue a single uncached read request using a word (32-bit) aligned address (the actual access could be for a word, partial word or a byte). If the internal core requests an uncached data that is larger than a word, the external request is then broken into two external requests. The first request will transfer 4 bytes and the second will transfer up to 4 bytes. Figure 13.12 illustrates the timing for an uncached read operation of one word.

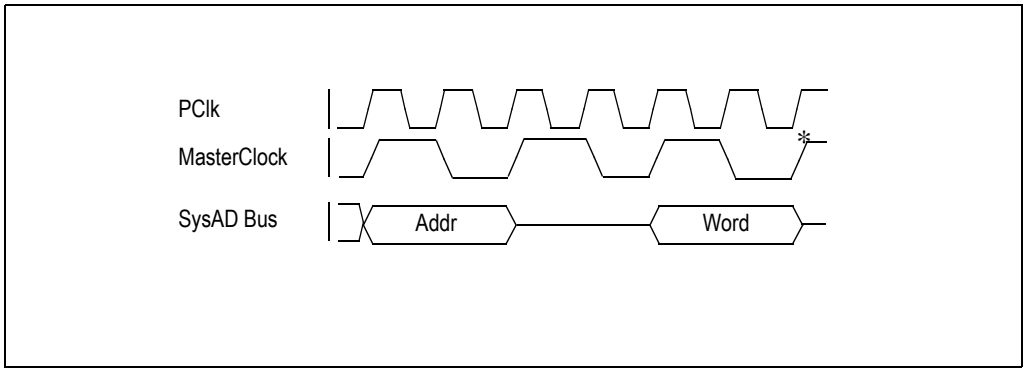


Figure 13.12 32-Bit Bus Mode Uncached Read for Single Word

Figure 13.13 illustrates the timing diagram for an uncached read operation of a double word value.

Notes

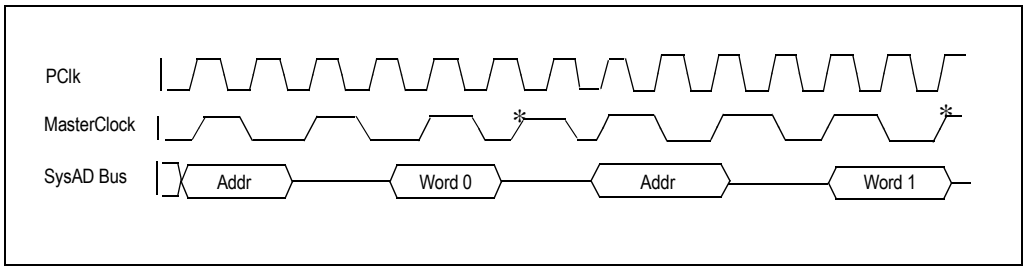


Figure 13.13 32-Bit Bus Mode Uncached Read for Double Word

The processor combines the word internally to generate a double word data to be used by the execution core. This implies that the order of the words in a double word data will be endian dependent. On little-endian machines, bits 31:0 will be transferred first, with bits 63:32 transferred second. On a big-endian machine, the order will be reversed.

Subblock Ordering

The order in which data is returned in response to a processor block read request is *subblock ordering*. In subblock ordering, the processor delivers the address of the requested doubleword (in 64-bit bus mode) or word (in 32-bit bus mode) within the block. An external agent must return the block of data using subblock ordering, starting with the addressed doubleword or word.

In general, a block of data elements (whether bytes, halfwords, words, or doublewords) can be retrieved from storage in two ways: in sequential order, or using a subblock order. This section describes these retrieval methods, with an emphasis on subblock ordering. Note that only subblock ordering is used for block reads.

Sequential Ordering Example

Sequential ordering retrieves the data elements of a block in serial, or sequential, order. Figure 13.14 shows a sequential order in which doubleword 0 (DW0) is taken first and doubleword 3 (DW3) is taken last.

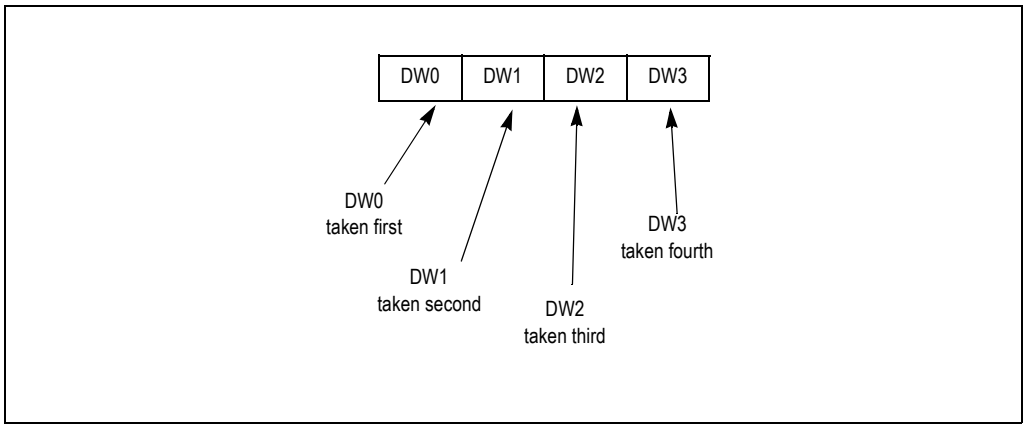


Figure 13.14 Retrieving a Data Block in Sequential Order

Subblock Ordering Examples

In 64-bit bus mode the smallest data element of a block transfer is a doubleword; in 32-bit bus mode, a single word. Figure 13.15 shows the retrieval of a block of data that consists of four doublewords in 64-bit bus mode, with doubleword 2 taken first. Cache line size is 8 words.

Using the subblock ordering shown in Figure 13.15, the doubleword at the target address is retrieved first (doubleword 2), followed by the remaining doubleword (doubleword 3) in this quadword. Next, the quadword that fills out the octalword are retrieved in the same order as the prior quadword (in this case doubleword 0 is followed by doubleword 1).

Notes

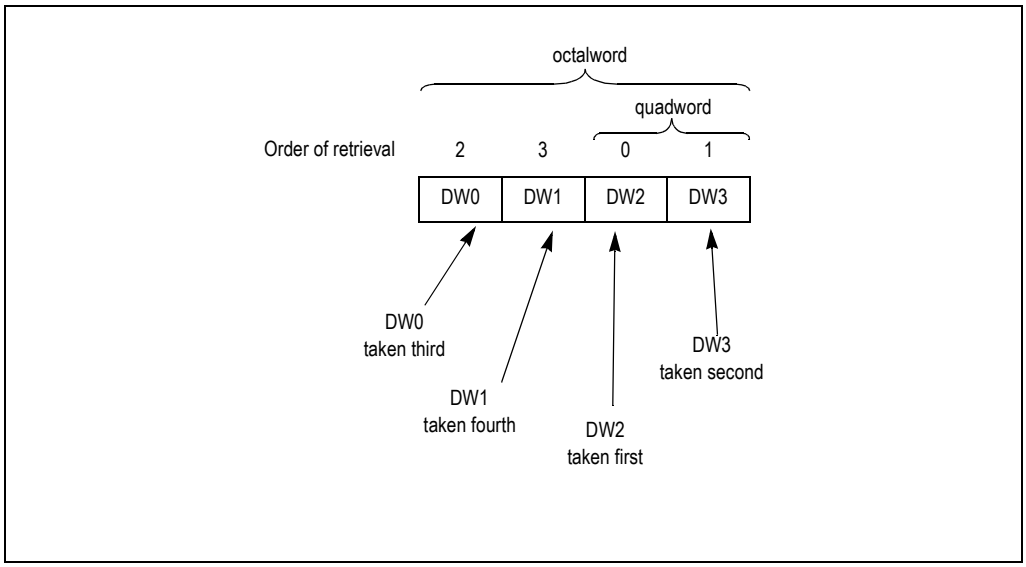


Figure 13.15 Retrieving Data in a Subblock Order

Figure 13.16 shows the retrieval of a block of data that consists of 8 words in 32-bit bus mode, with word 2 taken first. Cache-line size is 8 words.

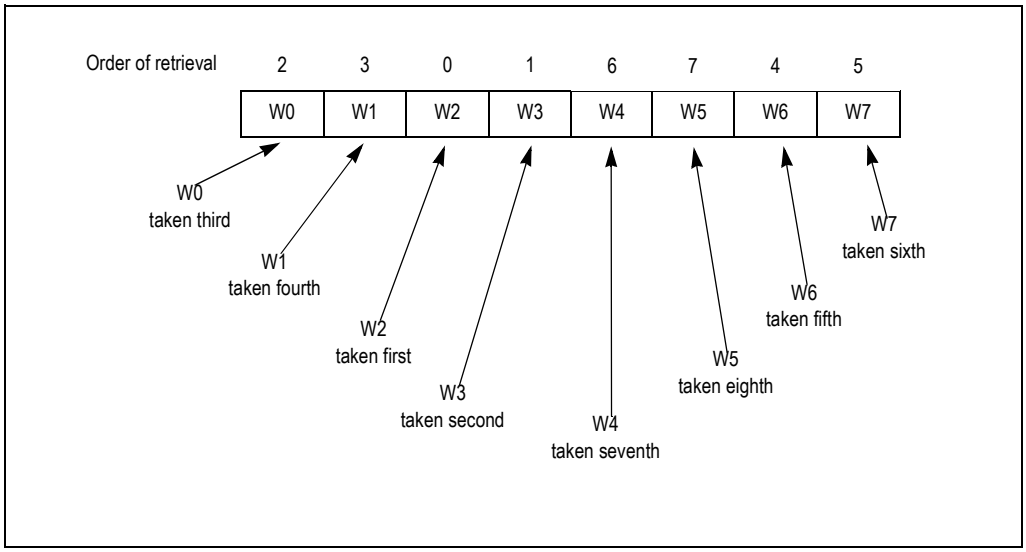


Figure 13.16 Retrieving Data in a Subblock Order

Using the subblock ordering shown in Figure 13.16, the word at the target address, in this case word 2, is retrieved first, followed by word 3. Next, word 6 is followed by word 7, then word 4, followed by word 5. Word 0 is then followed by word 1. A simpler way to understand subblock ordering would be to take a look at the method used for generating the address of each doubleword or word as it is retrieved. The subblock ordering logic generates this address by executing a bit-wise exclusive-OR (XOR) of the starting block address with the output of a binary counter that increments with each doubleword or word, starting at doubleword 0 (00_2) or word 0 (000_2).

Generating Subblock Order of Doublewords

Using this scheme, Table 13.6, Table 13.7, and Table 13.8 list the subblock ordering of doublewords for an 8-word block, based on three different starting-block addresses: 10_2, 11_2, and 01_2. The subblock ordering is generated by an XOR of the subblock address (either 10_2, 11_2, or 01_2) with the binary count of the doubleword (00_2 through 11_2).

Notes

Thus, the third doubleword retrieved from a block of data with a starting address of 10_2 is determined by taking the XOR of address 10_2 with the binary count of doubleword 2, 10_2. The result is 00_2, or doubleword 0, as shown in Table 13.6).

Cycle	Starting Block Address	Binary Count	Double Word Retrieved
1	10	00	10
2	10	01	11
3	10	10	00
4	10	11	01

Table 13.6 Sequence of Doublewords Transferred Using Subblock Ordering: Address 10_2

Cycle	Starting Block Address	Binary Count	Double Word Retrieved
1	11	00	11
2	11	01	10
3	11	10	01
4	11	11	00

Table 13.7 Sequence of Doublewords Transferred Using Subblock Ordering: Address 11_2

Cycle	Starting Block Address	Binary Count	Double Word Retrieved
1	01	00	01
2	01	01	00
3	01	10	11
4	01	11	10

Table 13.8 Sequence of Doublewords Transferred Using Subblock Ordering: Address 01_2

Generating Subblock Order of Words

Using the same scheme, Table 13.9 and Table 13.10 list the subblock ordering of words for an 8-word block, based on two different starting-block addresses: 010_2 and 011_2. The subblock ordering is generated by an XOR of the subblock address (either 010_2 or 011_2) with the binary count of the word (000_2 through 111_2). Therefore, the third word retrieved from a block of data with a starting address of 010_2 is determined by taking the XOR of address 010_2 with the binary count of word 2, 010_2. The result is 000_2, or word 0, as shown in Table 13.9.

Cycle	Starting Block Address	Binary Count	Word Retrieved
1	010	000	010
2	010	001	011
3	010	010	000
4	010	011	001
5	010	100	110
6	010	101	111
7	010	110	100
8	010	111	101

Table 13.9 Sequence of Words Transferred Using Subblock Ordering: Address 010_2

Notes

Cycle	Starting Block Address	Binary Count	Word Retrieved
1	011	000	011
2	011	001	010
3	011	010	001
4	011	011	000
5	011	100	111
6	011	101	110
7	011	110	101
8	011	111	100

Table 13.10 Sequence of Words Transferred Using Subblock Ordering: Address 110_2

Interface Commands & Data Identifiers

System interface commands specify the nature and attributes of any system interface request; this specification is made during the address cycle for the request. System interface data identifiers specify the attributes of data transmitted during a system interface data cycle.

The following sections describe the syntax, that is, the bitwise encoding, of system interface commands and data identifiers. The same **SysCmd** encoding is used for both 32-bit and 64-bit bus mode. The selection of 64-bit versus 32-bit is not dynamic and should be done only once during Reset. The RC64474/RC64475 does not indicate externally whether the bus is configured as 32-bit or 64-bit.

Reserved bits and reserved fields in the command or data identifier should be set to 1 for system interface commands and data identifiers associated with external requests. For system interface commands and data identifiers associated with processor requests, reserved bits and reserved fields in the command and data identifier are undefined.

Command and Data Identifier Syntax

System interface commands and data identifiers are encoded in 9 bits and are transmitted on the **SysCmd** bus from the processor to an external agent, or from an external agent to the processor, during address and data cycles. Bit 8 (the most-significant bit) of the **SysCmd** bus determines whether the current content of the **SysCmd** bus is a command or a data identifier and, therefore, whether the current cycle is an address cycle or a data cycle. For system interface commands, **SysCmd(8)** must be set to 0. For system interface data identifiers, **SysCmd(8)** must be set to 1.

System Interface Command Syntax

This section describes the **SysCmd** bus encoding for system interface commands. Figure 13.17 shows a common encoding used for all system interface commands.

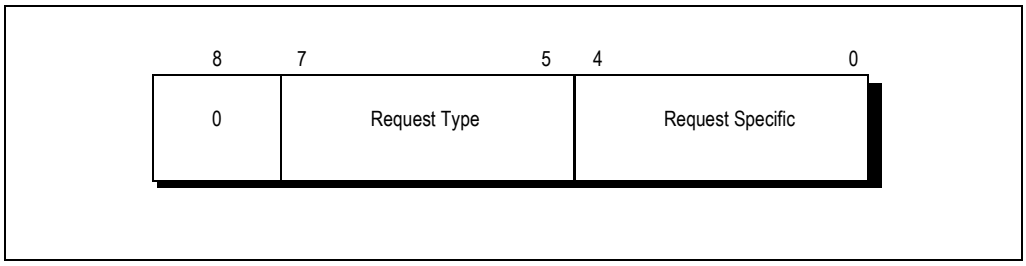


Figure 13.17 System Interface Command Syntax Bit Definition

SysCmd(8) must be set to 0 for all system interface commands. **SysCmd(7:5)** specify the system interface request type which may be read, write or null; Table 13.11 illustrates the types of requests encoded by the **SysCmd(7:5)** bits.

Notes

SysCmd(7:5)	Command
0	Read Request
1	Reserved
2	Write Request
3	Null Request
4 - 7	Reserved

Table 13.11 Encoding of SysCmd (7:5) for System Interface Commands

SysCmd(4:0) are specific to each type of request and are defined in each of the following sections.

Read requests

Figure 13.18 shows the format of a **SysCmd** read request.

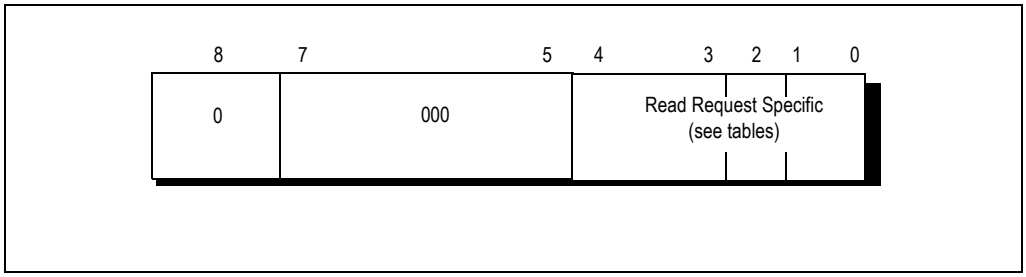


Figure 13.18 Read Request SysCmd Bus Bit Definition

Table 13.12, Table 13.13, and Table 13.14 list the encoding of **SysCmd(4:0)** for read requests.

SysCmd(4:3)	Read Attributes
0 - 1	Reserved
2	Noncoherent block read
3	64-bit mode:Doubleword, partial doubleword, word, or partial word 32-bit bus mode: Word or partial word.

Table 13.12 Encoding of SysCmd (4:3) for Read Requests

SysCmd(2)	Link Address Retained Indication
0	Link address not retained
1	Link address retained
SysCmd(1:0)	**Read Block Size**
0	Reserved
1	8 words (64-bit or 32-bit bus modes)
2 - 3	Reserved

Table 13.13 Encoding of SysCmd (2:0) for Block Read Request

Notes

SysCmd(2:0)	Read Data Size
	64-bit or 32-bit bus mode:
0	1 byte valid (Byte)
1	2 bytes valid (Halfword)
2	3 bytes valid (Tribyte)
3	4 bytes valid (Word)
	64-bit mode only:
4	5 bytes valid (Quintibyte)
5	6 bytes valid (Sextibyte)
6	7 bytes valid (Septibyte)
7	8 bytes valid (Doubleword)

Table 13.14 Doubleword, Word, or Partial-Word Read Request Data Size Encoding of SysCmd (2:0)

System Interface Data Identifier Syntax

This section defines the encoding of the **SysCmd** bus for system interface data identifiers. Figure 13.19 shows a common encoding scheme used for all system interface data identifiers.

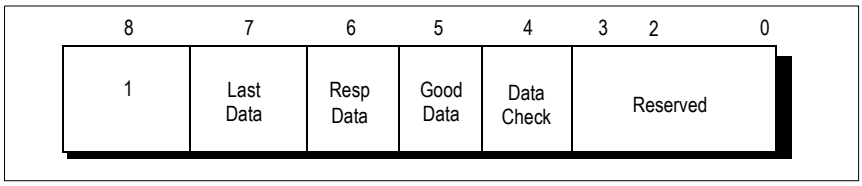


Figure 13.19 Data Identifier SysCmd Bus Bit Definition

SysCmd(8) must be set to 1 for all system interface data identifiers. System interface data identifiers use the format for noncoherent data.

Noncoherent Data

Noncoherent data is defined as follows:

- *data that is associated with processor block write requests and processor doubleword, partial doubleword, word, or partial word write requests*
- *data that is returned in response to a processor noncoherent block read request or a processor doubleword, partial doubleword, word, or partial word read request*
- *data that is associated with external write requests*
- *data that is returned in response to an external read request*

Data Identifier Bit Definitions

SysCmd(7) marks the last data element and **SysCmd(6)** indicates whether or not the data is response data, for both processor and external coherent and noncoherent data identifiers. Response data is data returned in response to a read request. **SysCmd(5)** indicates whether or not the data element is error free. Erroneous data contains an uncorrectable error and is returned to the processor, forcing a bus error.

The processor delivers data with the good data bit deasserted if a primary parity error is detected for a transmitted data item. **SysCmd(4)** indicates to the processor whether to check the data and check bits for this data element. **SysCmd(3)** is reserved for external data identifiers. **SysCmd(4:3)** are reserved for noncoherent processor data identifiers. **SysCmd(2:0)** are reserved for noncoherent data identifiers.

Table 13.15 lists the encoding of **SysCmd(7:3)** for processor data identifiers.

Notes

SysCmd(7)	Last Data Element Indication
0	Last data element
1	Not the last data element
SysCmd(6)	**Response Data Indication**
0	Data is response data
1	Data is not response data
SysCmd(5)	**Good Data Indication**
0	Data is error free
1	Data is erroneous
SysCmd(4:3)	Reserved

Table 13.15 Processor Data Identifier Encoding of SysCmd (7:3)

Table 13.16 lists the encoding of **SysCmd(7:3)** for external data identifiers.

SysCmd(7)	Last Data Element Indication
0	Last data element
1	Not the last data element
SysCmd(6)	**Response Data Indication**
0	Data is response data
1	Data is not response data
SysCmd(5)	**Good Data Indication**
0	Data is error free
1	Data is erroneous
SysCmd(4)	**Data Checking Enable**
0	Check the data and check bits
1	Do not check the data and check bits
SysCmd(3)	Reserved

Table 13.16 External Data Identifier Encoding of SysCmd (7:3)

During data cycles in 64-bit bus mode, the valid byte lanes depend upon the position of the data with respect to the aligned doubleword (this may be a byte, halfword, tribyte, quadbyte/word, quintibyte, sextibyte, septibyte, or an octalbyte/doubleword). For example, in little-endian mode, on a byte request where the address modulo 8 is 0, **SysAD(7:0)** are valid during the data cycles.

Table 13.17 shows the byte lanes used for partial word transfers for both little and big endian in 64-bit bus mode.

Notes

# Bytes SysCmd(2:0)	Address Mod 8	SysAD Byte Lanes Used (Big Endian) 63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
1 (000)	0	x							
	1		x						
	2			x					
	3				x				
	4					x			
	5						x		
	6							x	
	7								x
2 (001)	0	x	x						
	2			x	x				
	4					x	x		
	6							x	x
3 (010)	0	x	x	x					
	1		x	x	x				
	4					x	x	x	
	5						x	x	x
4 (011)	0	x	x	x	x				
	4					x	x	x	x
5 (100)	0	x	x	x	x	x			
	3				x	x	x	x	x
6 (101)	0	x	x	x	x	x	x		
	2			x	x	x	x	x	x
7 (110)	0	x	x	x	x	x	x	x	
	1		x	x	x	x	x	x	x
8 (111)	0	x	x	x	x	x	x	x	x
		7:0	15:8	23:16	31:24	39:32	47:40	55:48	63:56
		SysAD Byte Lanes Used (Little Endian)							

Table 13.17 Partial Word Transfer Byte Lane Usage—64-Bit Mode

During data cycles in 32-bit bus mode, the valid byte lanes depend upon the position of the data with respect to the aligned word, which may be a byte, halfword, tribyte, or word. For example, in little-endian mode, on a byte request where the address modulo 4 is 0, **SysAD(7:0)** are valid during the data cycles. Table 13.8 shows the byte lanes used for partial word transfers for both little and big endian in 32-bit bus mode.

Notes

# Bytes	Address	SysAD Byte Lanes Used (Big Endian)			
SysCmd(2:0)	Mod 4	31:24	23:16	15:8	7:0
1 (000)	0	x			
	1		x		
	2			x	
	3				x
2 (001)	0	x	x		
	2			x	x
3 (010)	0	x	x	x	
	1		x	x	x
4 (011)	0	x	x	x	x
		0:7	8:15	16:23	24:31
		SysAD Byte Lanes Used (Little Endian)			

Table 13.18 Partial Word Transfer Byte Lane Usage—32-Bit Mode

Notes

The Write Interface

Notes

Introduction

This chapter discusses the Write protocol and associated operations. When a processor issues a write request, the specified resource is accessed and the data is written to it. A processor write request is complete after the last word of data has been transmitted to the external agent. In no-secondary-cache mode, the external agent must be capable of accepting a processor write request any time **WrRdy*** has been asserted for one clock cycle, two cycles before the issue cycle.

To enhance the throughput of non-block writes, two new modes have been added that allow for 2 cycle throughput on back-to-back non-block writes. The external agent must be capable of accepting a processor write request in these modes under the same conditions as for the RISCore4000 family compatibility mode (except as noted later in this chapter).

Special Notation: The bus interface of the RC64474 and RC64475 devices has been enhanced with the introduction of a programmable delay inserted between the write address and the write data during write cycles (for both block and non-block writes). This bus delay can be defined to vary between 0 and 7 **MasterClock** cycles, as shown in Table 9.2 of chapter 9. This system enhancement facilitates discrete interface to SDRAM and is activated and controlled via mode bits (17:15), during the reset initialization. The '000' setting will maintain write operation's timing compatibility between the RC64474/RC46475 and the RC4640/RC4650 based systems.

Processor Write Protocols

The following sections contain a cycle-by-cycle description of the bus arbitration protocols for the processor write request. Table 14.1 describes the buses that appear in the timing diagrams that follow.

Scope	Abbreviation	Description
Global	Unsd	Unused
SysAD bus	Addr	Physical address
	Data<n>	Data element number n of a block of data
SysCmd bus	Cmd	An unspecified system interface command
	Read	A processor or external read request command
	Write	A processor or external write request command
	SINull	A system interface release external null request command
	NData	A noncoherent data identifier for a data element other than the last data element
	NEOD	A noncoherent data identifier for the last data element

Table 14.1 System Interface Requests

The RC64474/RC64475 have three write protocols:

- *R4000-family-compatible mode*
- *Pipeline write*
- *Write reissue*

These protocols apply to both single and block write and to 32-bit and 64-bit interface mode. This means, for example, that for pipeline write a single write can be followed immediately by a block write that the external agent must accept. The write protocol is selected through the reset vector, along with the bus width interface. The selection of the write protocol is static, which means that it should be selected once

Notes

during reset. In R4000 family compatible write, a single write access takes four clock cycles, while in pipeline write or write reissue a single write access takes two clock cycles.

Processor Write-Request Protocol

Processor write requests are issued using one of two protocols:

- *Doubleword, partial doubleword, word, or partial word writes use a word[1] write request protocol.*
- *Block writes use a block write request protocol.*

Processor word write requests are issued with the system interface in master state, as described in the following steps. These steps apply to both 64-bit and 32-bit bus interface modes.

1. A processor single word write request is issued by driving a write command on the **SysCmd** bus and a write address on the **SysAD** bus.
2. The processor asserts **ValidOut***.
3. The processor drives a data identifier on the **SysCmd** bus and data on the **SysAD** bus.
4. The data identifier associated with the data cycle must contain a last data cycle indication. At the end of the cycle, **ValidOut*** is deasserted.

Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively. Figure 14.1 shows a processor noncoherent word write request cycle.

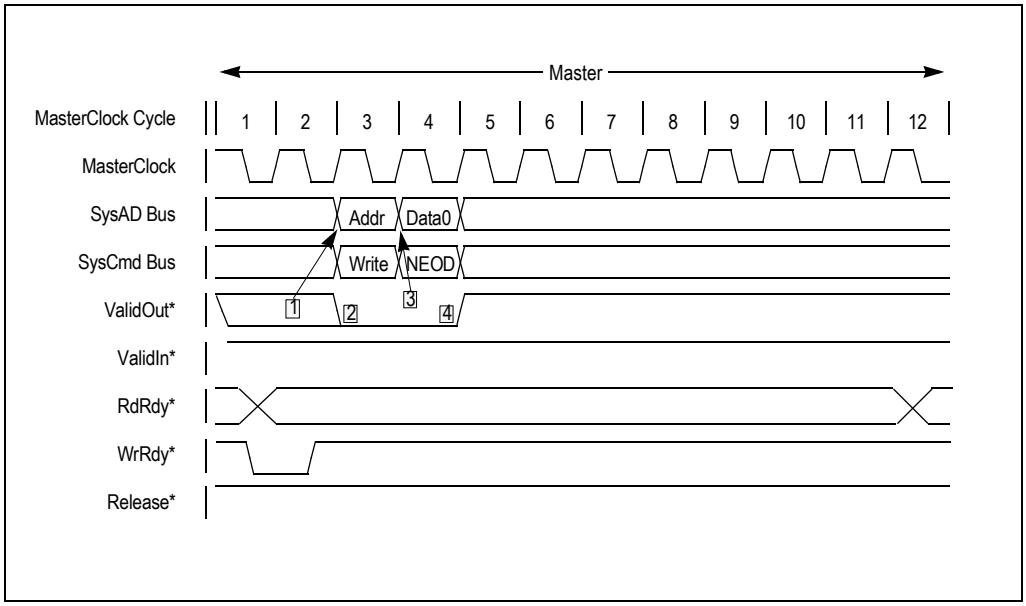


Figure 14.1 Processor Noncoherent Word Write Request Protocol

Processor Single-Write Request

There are three types of processor single write requests, as follows:

- *R4000 compatible writes*
- *Write reissue*
- *Pipelined writes*

R4000 Compatible Write Mode

In R4000 family compatible write mode a single write operation takes four clock cycles. The address is asserted for one clock cycle, followed by one clock cycle of data and then two unused clock cycles. This applies to both 64-bit and 32-bit bus modes, and is illustrated in Figure 14.2.

1. Called *word* to distinguish it from *block* request protocol. Data transferred can actually be doubleword, partial doubleword, word, or partial word.

Notes

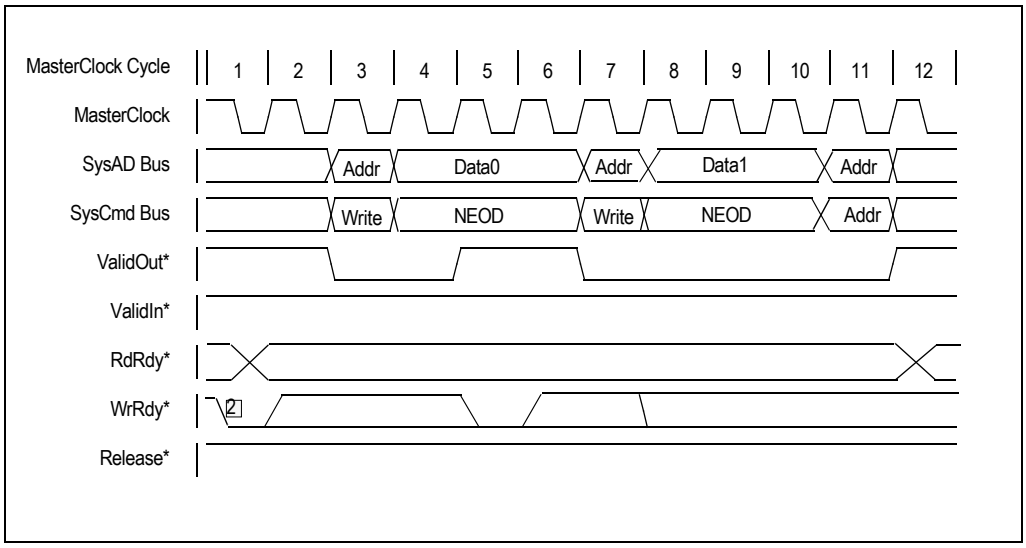


Figure 14.2 R4000 Compatible Write Mode

The RC64474/RC64475 interface requires that **WrRdy*** be asserted two system cycles prior to the issue of a write, for one clock cycle. An external agent that deasserts **WrRdy*** immediately upon receiving the write that fills its buffer will stop a subsequent write for four system cycles in RC4000 non-block write compatible mode. This leaves two null system cycles after a write address/data pair to give the external agent time to stop the next write.

An Address/data pair every four system cycles is not sufficiently high performance for all applications. For this reason, the RC64474/RC64475 provide two new protocol options that modify the R4000 back-to-back write protocol to allow an address/data pair every two system cycles. The first protocol, called write reissue, allows **WrRdy*** to be deasserted during the address cycle and forces a write to be reissued. The second, called pipelined writes, leaves the sample point of **WrRdy*** unchanged and requires that the external agent accept one more write than the RC4000 protocol.

Write Reissue

In Write Reissue mode, writes issue when **WrRdy*** is asserted both for 1 clock cycle, two cycles prior to the address cycle and during the address cycle. The write reissue protocol is shown in Figure 14.3. For this figure, note the following:

- *For Addr0/Data0 the write will issue because **WrRdy*** is sampled LOW at *0 and at *1, which is the issue cycle.*
- *Addr1/Data1 will not issue because **WrRdy*** is sampled HIGH at *2, which is the possible issue cycle.*
- *This address/data pair will then be reissued to the system interface, and will issue as indicated in 14.3 because **WrRdy*** is sampled LOW at *3 and at *4.*

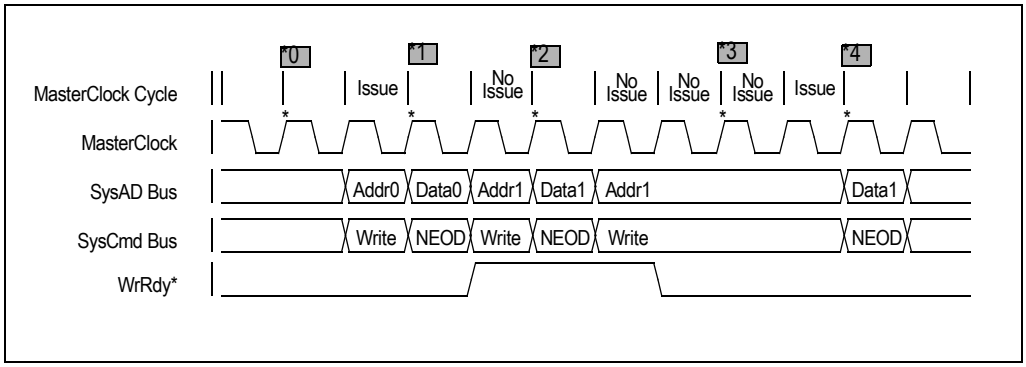


Figure 14.3 Write Reissue

Notes

Pipelined Write

The pipelined write protocol maintains the RC4000 write issue rule (which is, issue if **WrRdy*** is asserted two cycles prior to the address cycle, for one clock cycle), and eliminates the two null cycles between writes. The external agent may be required to accept one more write after it deasserts **WrRdy***.

This protocol is shown in Figure 14.4. For this figure note the following:

- *Addr0/Data0 issues because **WrRdy*** was asserted at *0.*
- *Addr1/Data1 will be issued because **WrRdy*** was asserted at *1.*
- *Addr2/Data2 will not issue at first because **WrRdy*** is sampled HIGH at *2. It will issue as indicated in the figure because **WrRdy*** was sampled LOW at *3.*

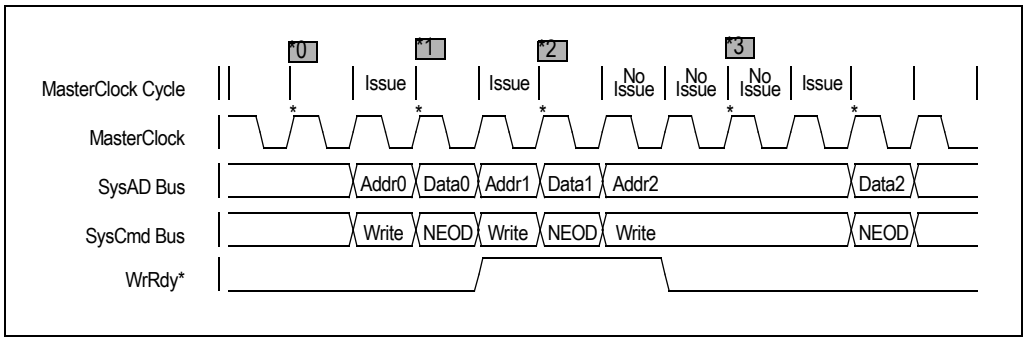


Figure 14.4 Pipelined Writes

All three write protocols apply for both single write and block writes. This means that in pipeline write, for example, a single write can be followed immediately by a block write that the external agent must accept.

Processor Block-Write Request

Processor block write requests are issued with the system interface in master state, as described below. The protocol is the same for either 64-bit or 32-bit bus mode. A processor noncoherent block request for eight words of data in 64-bit bus mode is illustrated in Figure 14.5.

1. The processor issues a write command on the **SysCmd** bus and a write address on the **SysAD** bus
2. The processor asserts **ValidOut***.
3. The processor drives a data identifier on the **SysCmd** bus and data on the **SysAD** bus.
4. The processor asserts **ValidOut*** for a number of cycles sufficient to transmit the block of data.
5. The data identifier associated with the last data cycle must contain a last data cycle indication.

Figure 14.5 illustrates a processor noncoherent block request for eight words of data with a data pattern of DDDD in 64-bit bus mode.

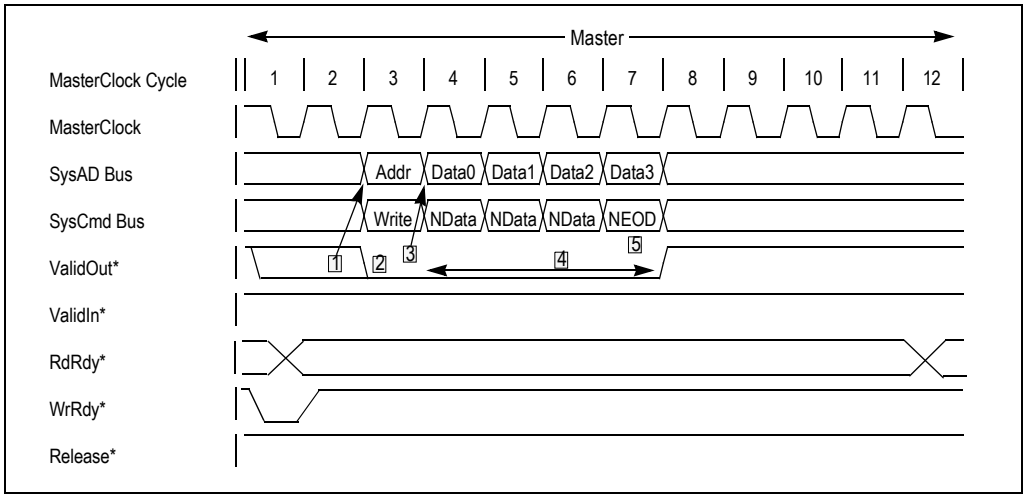


Figure 14.5 Processor Noncoherent Block Write Request Protocol

Notes

Write Data Transfer Patterns

The write data pattern specifies the pattern the processor uses when writing a block to the external agent. This pattern is specified once through the mode bits during reset. A data pattern is a sequence of letters indicating the *data* and *unused* cycles that repeat to provide the appropriate data rate. For example, the data pattern **DDxx** specifies a repeatable data rate of two doublewords every four cycles, with the last two cycles unused.

Table 14.2 lists the maximum processor data rate and the data pattern for each data rate in 64-bit mode. Data patterns are specified using the characters *D* and *x; D* indicates a doubleword data cycle and *x* indicates an unused cycle. During the unused cycles, the data bus will maintain the last doubleword data value (D).

Maximum Data Transmit Rate Block Writes	Data Pattern
1 Double/1 MasterClock Cycle	DDDD
2 Doubles/3 MasterClock Cycles	DDxDDx
1 Double/2 MasterClock Cycles	DDxxDDxx
1 Double/2 MasterClock Cycles	DxDxDxDx
2 Doubles/5 MasterClock Cycles	DDxxxDDxxx
1 Double/3 MasterClock Cycles	DDxxxxDDxxxx
1 Double/3 MasterClock Cycles	DxxDxxDxxDxx
1 Double/4 MasterClock Cycles	DDxxxxxxDDxxxxxx
1 Double/4 MasterClock Cycles	DxxxDxxxDxxxDxxx

Table 14.2 Transmit Data Rates and Patterns in 64-Bit Mode

Table 14.3 lists the maximum processor data rate and the data pattern for each data rate in 32-bit mode. Data patterns are specified using the characters *W* and *x*; W indicates a word data cycle and *x* indicates an unused cycle. During the unused cycles, the data bus will maintain the last word data value (D).

Maximum Data Transmit Rate Block Writes	Data Pattern
1 Double/1 MasterClock Cycle	WWWWWWWW
2 Doubles/3 MasterClock Cycles	WWxWWxWWxWWx
1 Double/2 MasterClock Cycles	WWxxWWxxWWxxWWxx
1 Double/2 MasterClock Cycles	WxWxWxWxWxWxWxWx
2 Doubles/5 MasterClock Cycles	WWxxxWWxxxWWxxxWWxxx
1 Double/3 MasterClock Cycles	WWxxxxWWxxxxWWxxxxWWxxxx
1 Double/3 MasterClock Cycles	WxxWxxWxxWxxWxxWxxWxxWxx
1 Double/4 MasterClock Cycles	WWxxxxxxWWxxxxxxWWxxxxxxWWxxxxxx
1 Double/4 MasterClock Cycles	WxxxWxxxWxxxWxxxWxxxWxxxWxxxWxxx

Table 14.3 Transmit Data Rates and Patterns in 32-Bit Mode

Notes

Processor Request & Flow Control

To control the flow of processor write requests, the external agent uses **WrRdy***. These are the steps that occur:

1. The processor samples the signal **WrRdy*** to determine if the external agent is capable of accepting a read request.
2. The processor does not complete the issue of a read request, until it issues an address cycle in response to the request for which the signal **RdRdy*** was asserted two cycles earlier.
3. The processor does not complete the issue of a write request until it issues an address cycle in response to the write request for which the signal **WrRdy*** was asserted two cycles earlier.

Figure 14.6 illustrates two processor write requests in which the issue of the second is delayed for the assertion of **WrRdy***. These steps apply for both 64-bit and 32-bit bus modes.

Note: Timings for the **SysADC** and **SysCmdP** buses are the same as for the **SysAD** and **SysCmd** buses, respectively.

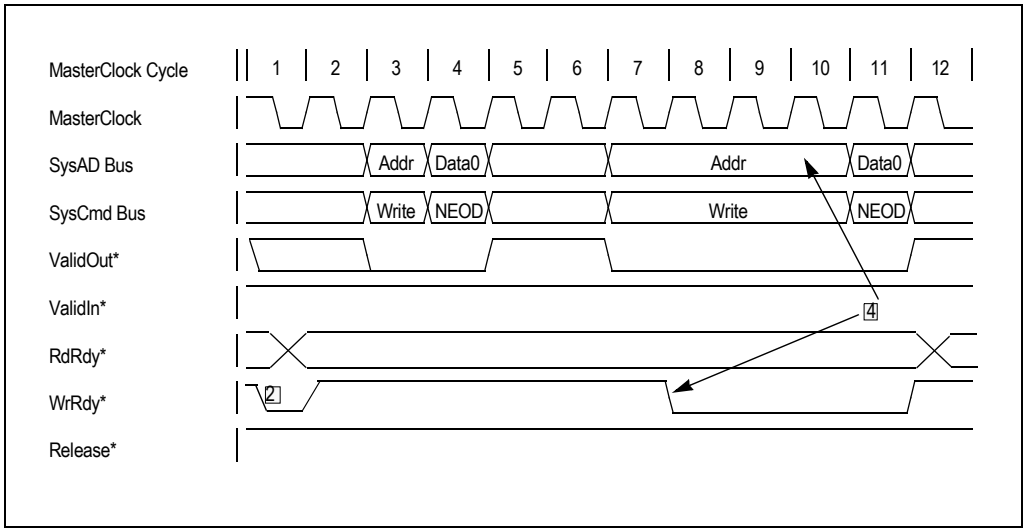


Figure 14.6 Two Processor Write Requests, Second Write Delayed for the Assertion of WrRdy*

64-Bit and 32-Bit Bus Modes

The bus interface of the RC64475 can be configured during reset to be either 64-bit wide or 32-bit wide. The same bus protocol explained earlier in this chapter applies for both modes. In 32-bit bus mode, the internal execution core is still a full 64-bit engine. Only the bus interface unit can be configured as either 64-bit or 32-bit interface.

The bus width mode is a static feature of the device. This means that the bus width has to be configured once during reset. This feature should not be thought of as dynamic bus width interface where the bus width is 64-bit in one access and 32-bit wide in the other access.

64-bit Bus Mode

In 64-bit bus mode, the RC64475 supports 64-bit address/data system interface that consist of:

- *64-bit address and data,* ***SysAD(63:0)***
- *8-bit SysAD check bus,* ***SysADC(7:0)*** *(even parity)*
- *9-bit command bus,* ***SysCmd(8:0)***
- *Six handshake signals:*
  RdRdy*,* ***WrRdy****
  ExtReq*,* ***Release****
  ValidIn*,* ***ValidOut****

Notes

64-bit Bus Mode Block Write Operation

In 64-bit bus mode, the RC64475 issues a single block write request for the entire cache line (4 double words). The external agent should return all four double words as explained in the write protocol section earlier. Figure 14.7 illustrates the timing diagram for a block write operation in 64-bit bus mode. The address issued by the RC64475 is double word (64-bit) aligned.

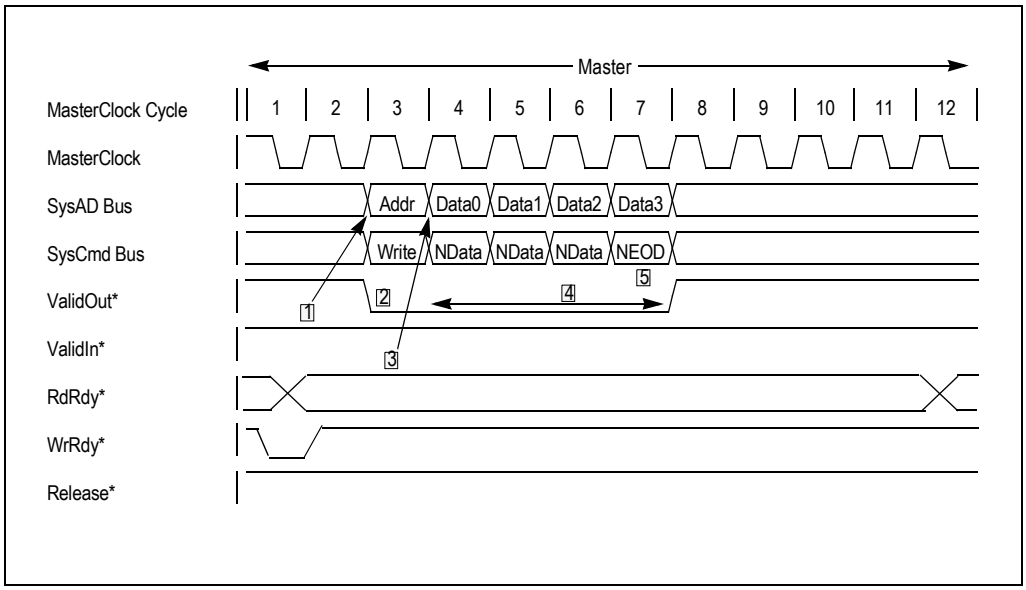


Figure 14.7 Processor Noncoherent Block Write Request Protocol

64-bit Bus Mode Single (Uncached) Write Operation

In 64-bit bus mode, the RC64475 will issue a single uncached write request using a doubleword (64-bit) aligned address. The actual access can be for a doubleword, word, partial word, or byte, but the request is called a *word write request* to differentiate it from the block write request.

R4000 Family Compatible Write Mode

While in the R4000 family compatible write mode, a single write operation takes four clock cycles. The address is asserted for one clock cycle, followed by one clock cycle of data and then two unused clock cycles. This is illustrated in Figure 14.8.

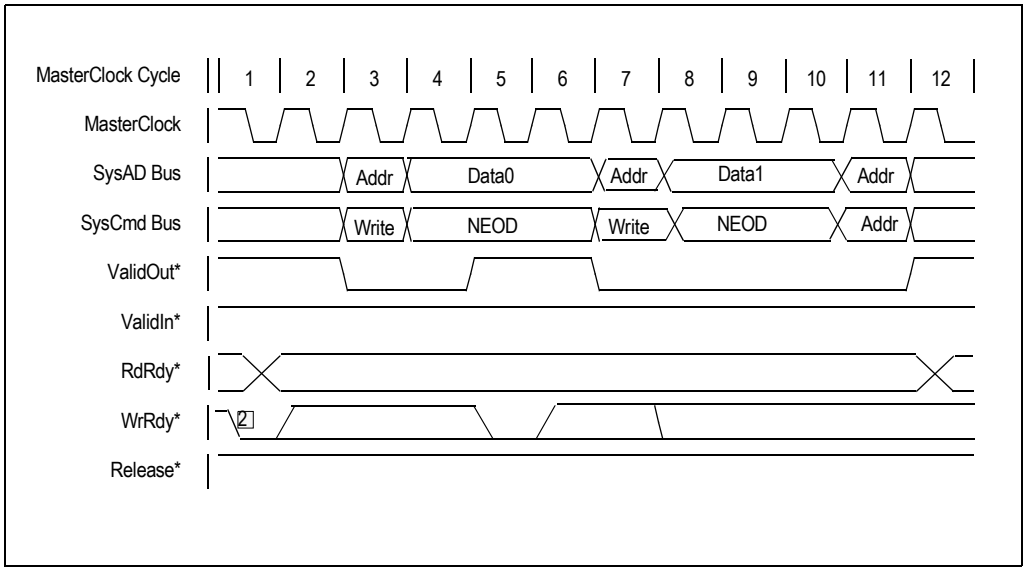


Figure 14.8 R4000 Family Compatible Write Mode

Notes

The RC64474/RC64475 interface requires that **WrRdy*** be asserted two system cycles prior to the issue of a write, for one clock cycle. An external agent that deasserts **WrRdy*** immediately upon receiving the write that fills its buffer will stop a subsequent write for four system cycles in RC4000 non-block write compatible mode. This leaves two null system cycles after a write address/data pair to give the external agent time to stop the next write.

Write Reissue

Writes issue when **WrRdy*** is asserted both for 1 clock cycle, two cycles prior to the address cycle and during the address cycle. The write reissue protocol is shown in Figure 14.9. For this figure note the following:

- *For Addr0/Data0 the write will issue because **WrRdy*** is sampled LOW at *0 and at *1, which is the issue cycle.*
- *Addr1/Data1 will not issue because **WrRdy*** is sampled HIGH at *2, which is the possible issue cycle.*
- *This address/data pair will then be reissued to the system interface, and will issue as indicated in Figure 14.9 because **WrRdy*** is sampled LOW at *3 and at *4.*

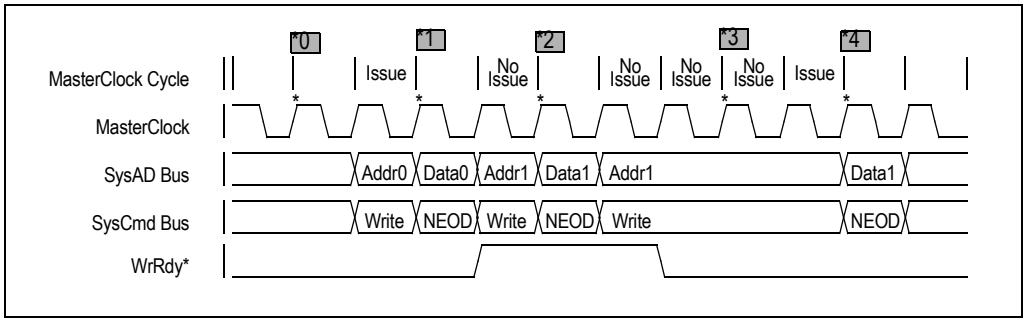


Figure 14.9 Write Reissue

Pipelined Writes

The pipelined write protocol maintains the R4000 write issue rule (which is, issue if **WrRdy*** is asserted two cycles prior to the address cycle, for one clock cycle), and eliminates the two null cycles between writes. The external agent may be required to accept one more write after it deasserts **WrRdy***. This protocol is shown in Figure 14.10. For this figure note the following:

- *Addr0/Data0 issues because **WrRdy*** was asserted at *0.*
- *Addr1/Data1 will be issued because **WrRdy*** was asserted at *1.*
- *Addr2/Data2 will not issue at first because **WrRdy*** is sampled HIGH at *2. It will issue as indicated in the figure because **WrRdy*** was sampled LOW at *3.*

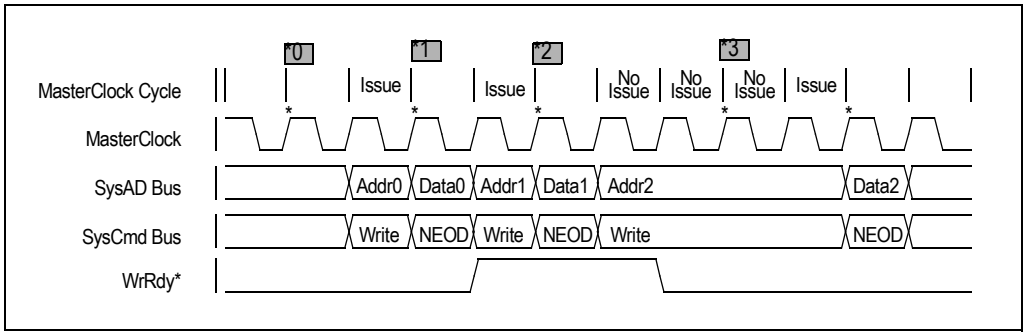


Figure 14.10 Pipelined Writes

All three write protocols apply for both single write and block writes. For example,this means that in pipeline write a single write can be followed immediately by a block write that the external agent must accept.

Notes

32-bit Bus Mode

In 32-bit bus mode, the RC64474 or RC64475 supports a 32-bit address/data system interface that consists of the following:

- *The 32-bit address & data (**SysAD (31:0)**) and the 4-bit **SysAD** check bus (**SysADC(3:0)'**, even parity). **SysAD(63:31)** and **SysADC(7:4)** are undefined.*
- *9-bit command bus, **SysCmd(8:0)***
- *Six handshake signals:*

  RdRdy*, **WrRdy***

  ExtReq*, Release*

  ValidIn*, ValidOut*

It is important to note that in the 32-bit bus mode **SysAd(31:0)** and **SysADC(3:0)** are always used regardless of the Endianness of the system. It is also important to note that the encoding of **SysCmd(8:0)** is the same for both 64-bit and 32-bit bus modes. This means that the processor does not inform the external agent about the bus width mode. It is expected that this mode is programmed during reset and that the external agent is configured to interface to the processor in either 64-bit or 32-bit bus mode.

32-Bit Bus Mode Block Write Operation

In 32-bit bus mode, the RC64474 or RC64475 will issue a single block write request for the entire cache line (4 double words). since the bus interface is configured to be 32-bit wide, the processor then issues a single address that is word (32-bit) aligned, followed by 8 single words to the RC64474/RC64475.

Figure 14.11 illustrates the timing diagram for a block write operation in 32-bit bus mode. This means that a block write request is not divided into two requests. The external agent is responsible for accepting all 8 single word from the processor.

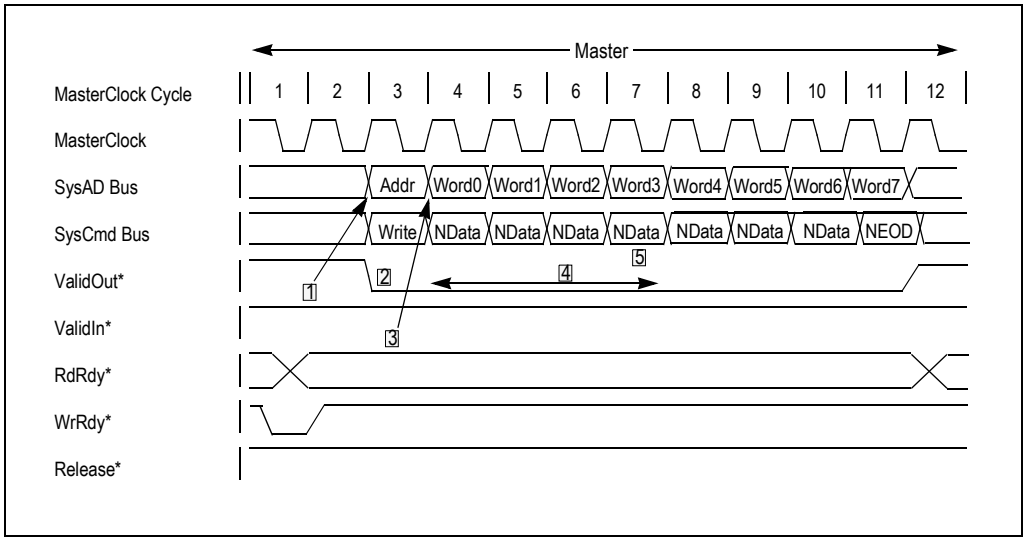


Figure 14.11 Processor Noncoherent Block Write Request Protocol

The order of the words in a double word datum will be endian-dependent. On little-endian machines bits 31:0 will be transferred first and bits 63:32 transferred second, while on a big-endian machine the order will be reversed.

32-bit Bus Mode Single (Uncached) Write Operation

In 32-bit bus mode, the RC64474 or RC64475 will issue a single uncached write request using a word (32-bit) aligned address (the actual access could be for a word, partial word or a byte). If the internal core writes an uncached datum that is larger than a word, the external request is then broken into two external requests. The first request will transfer 4 bytes and the second will transfer up to 4 bytes. The order of the

Notes

words in a double word datum will be endian dependent. On little-endian machines, bits 31:0 will be transferred first, with bits 63:32 transferred second. On a big-endian machine, the order will be reversed.

R4000 Family Compatible Write Mode

In the R4000 family compatible write mode, a single write operation takes four clock cycles. The address is asserted for one clock cycle, followed by one clock cycle of data and then two unused clock cycles. This is illustrated in Figure 14.12.

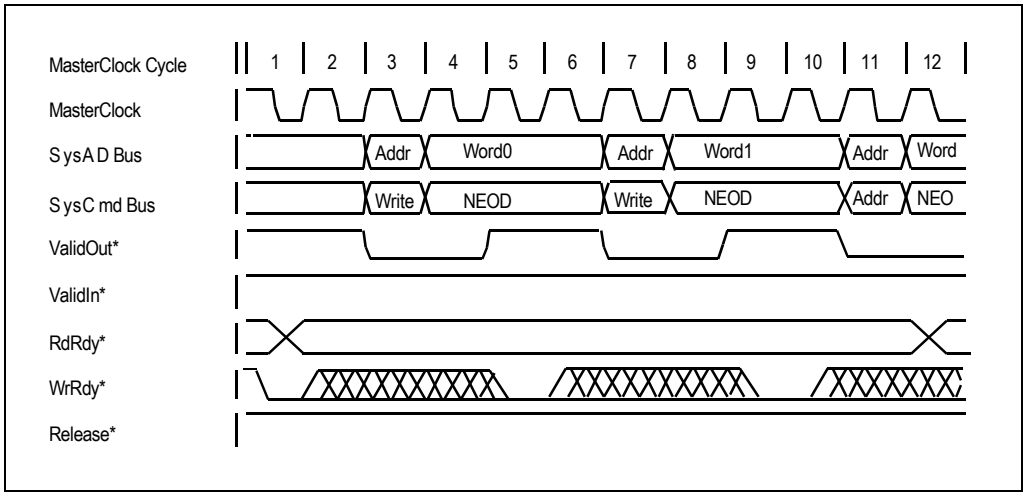


Figure 14.12 R4000 Family Compatible Write Protocol

The RC64474/RC64475 interface requires that **WrRdy*** be asserted two system cycles prior to the issue of a write, for one clock cycle. An external agent that deasserts **WrRdy*** immediately upon receiving the write that fills its buffer will stop a subsequent write for four system cycles in RC4000 non-block write compatible mode. This leaves two null system cycles after a write address/data pair to give the external agent time to stop the next write.

Write Reissue

Writes issue when **WrRdy*** is asserted both for 1 clock cycle, two cycles prior to the address cycle and during the address cycle. A 64-bit transfer is broken into 2 word transfers. The write reissue protocol is shown in Figure 14.13. For this figure, note the following:

- *For Addr0/Word0 the write will issue because **WrRdy*** is sampled LOW at *0 and at *1, which is the issue cycle.*
- *Addr1/Word1 will not issue because **WrRdy*** is sampled HIGH at *2, which is the possible issue cycle.*
- *This address/word pair will then be reissued to the system interface, and will issue as indicated in Figure 14.13 because **WrRdy*** is sampled LOW at *3 and at *4.*

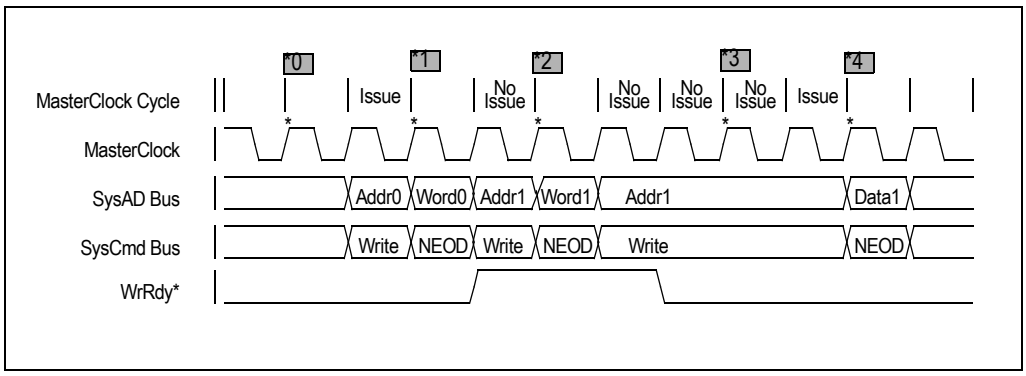


Figure 14.13 Write Reissue

Notes

Pipelined Writes

The pipelined write protocol maintains the RC4000 write issue rule (which is, issue if **WrRdy*** is asserted two cycles prior to the address cycle, for one clock cycle), and eliminates the two null cycles between writes. The external agent may be required to accept one more write after it deasserts **WrRdy***.

The pipeline write protocol is shown in Figure 14.14. For this figure, note the following:

- *Addr0/Word0 issues because **WrRdy*** was asserted at *0.*
- *Addr1/Word1 will be issued because **WrRdy*** was asserted at *1.*
- *Addr2/Word2 will not issue at first because **WrRdy*** is sampled HIGH at *2. It will issue as indicated in the figure because **WrRdy*** was sampled LOW at *3.*

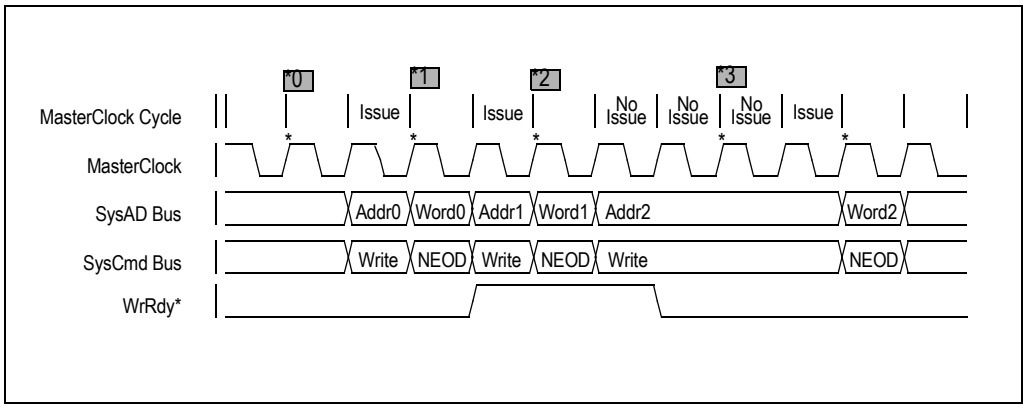


Figure 14.14 Pipelined Writes

All three write protocols apply for both single write and block writes. This means that in pipeline write, for example, a single write can be followed immediately by a block write that the external agent must accept.

Note: In 32-bit bus mode and pipeline write mode a single write can be followed by a block write of eight words. This means that the external agent must be capable of accepting all nine words both: a) in a sequential fashion, and b) at the speed of the data transmission pattern selected during reset.

Sequential Ordering

For block write requests in 64-bit bus mode, the processor always delivers the address of the doubleword at the beginning of the block. The processor delivers data beginning with the doubleword at the beginning of the block and progresses sequentially through the doublewords that form the block.

For block write requests in 32-bit bus mode, the processor always delivers the address of the word at the beginning of the block. The processor delivers data beginning with the word at the beginning of the block and progresses sequentially through the words that form the block. Note that sequential ordering begins the first data element's address in the same order as for sub-block ordering, which allows some simplification of the memory controller.

Sequential Ordering Example

Sequential ordering transfers the data elements of a block in serial, or sequential, order. Figure 14.15 shows a sequential order in which doubleword 0 (DW0) is transferred first and doubleword 3 (DW3) is transferred last.

Notes

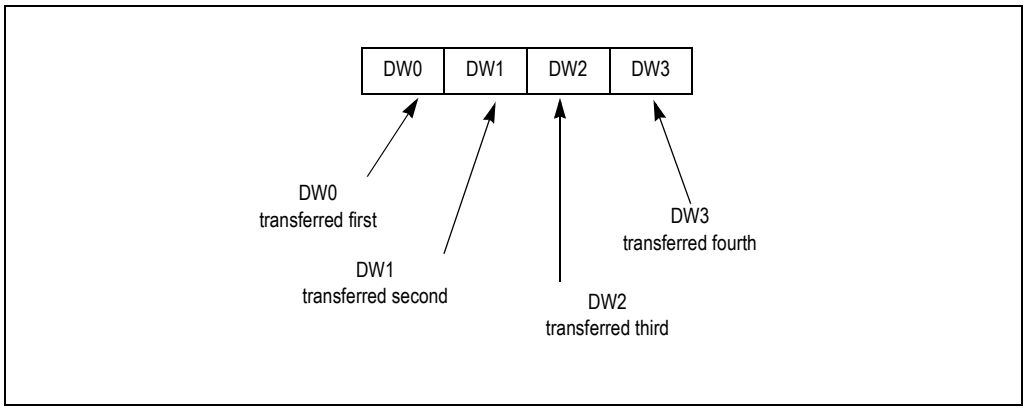


Figure 14.15 Transferring a Data Block in Sequential Order

Figure 14.16 shows a sequential order in which Word0 (W0) is transferred first and Word 7 (W7) is transferred last.

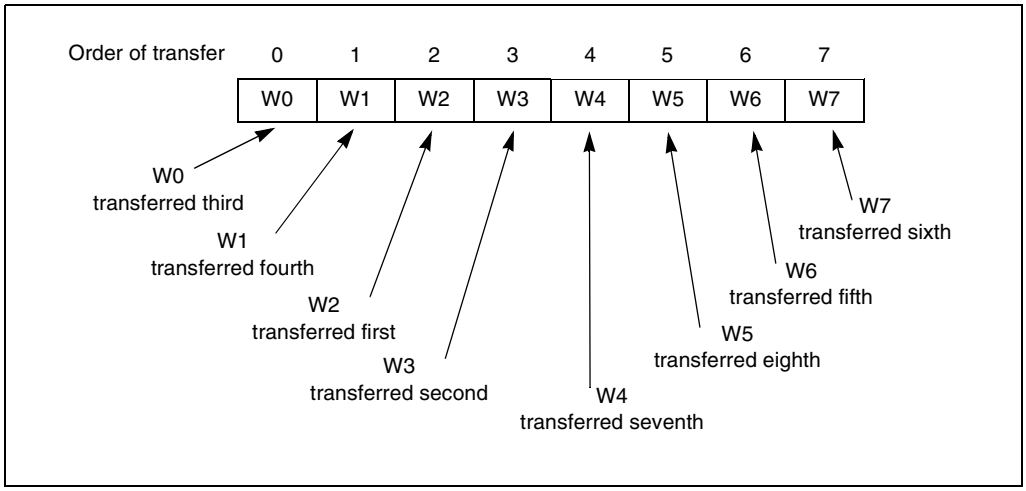


Figure 14.16 Transferring Data in a Subblock Order

Table 14.4 shows the byte lanes used for 64-bit bus mode partial word transfers for both little and big endian.

Notes

# Bytes SysCmd(2:0)	Address Mod 8	SysAD byte lanes used (big endian) 63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
1 (000)	0	x							
	1		x						
	2			x					
	3				x				
	4					x			
	5						x		
	6							x	
	7								x
2 (001)	0	x	x						
	2			x	x				
	4					x	x		
	6							x	x
3 (010)	0	x	x	x					
	1		x	x	x				
	4					x	x	x	
	5						x	x	x
4 (011)	0	x	x	x	x				
	4					x	x	x	x
5 (100)	0	x	x	x	x	x			
	3				x	x	x	x	x
6 (101)	0	x	x	x	x	x	x		
	2			x	x	x	x	x	x
7 (110)	0	x	x	x	x	x	x	x	
	1		x	x	x	x	x	x	x
8 (111)	0	x	x	x	x	x	x	x	x
		7:0	15:8	23:16	31:24	39:32	47:40	55:48	63:56
		SysAD byte lanes used (little endian)							

Table 14.4 Partial Word Transfer Byte Lane Usage

Table 14.5 shows the byte lanes used for 32-bit bus mode partial word transfers for both little and big endian.

Notes

# Bytes SysCmd(2:0)	Address Mod 4	SysAD Byte Lanes Used (Big Endian) 31:24	23:16	15:8	7:0
1 (000)	0	x			
	1		x		
	2			x	
	3				x
2 (001)	0	x	x		
	2			x	x
3 (010)	0	x	x	x	
	1		x	x	x
4 (011)	0	x	x	x	x
		0:7	8:15	16:23	24:31
		SysAD Byte Lanes Used (Little Endian)			

Table 14.5 Partial Word Transfer Byte Lane Usage—32-Bit Mode

During data cycles, the valid byte lines depend upon the position of the data with respect to the aligned doubleword (this may be a byte, halfword, tribyte, quadbyte/word, quintibyte, sextibyte, septibyte, or an octalbyte/doubleword). For example, in little-endian mode, on a byte request where the address modulo 8 is 0, **SysAD(7:0)** are valid during the data cycles.

Interface Commands & Data Identifiers

System interface commands specify the nature and attributes of any system interface request; this specification is made during the address cycle for the request. System interface data identifiers specify the attributes of data transmitted during a system interface data cycle.

The sections that follow describe the syntax, that is, the bitwise encoding, of system interface commands and data identifiers. The same **SysCmd** encoding is used for both 32-bit and 64-bit bus mode. The selection of 64-bit versus 32-bit is not dynamic and should be done only once during **Reset**. The RC64474/RC64475 do not indicate externally whether the bus is configured as 32-bit or 64-bit.

Reserved bits and reserved fields in the command or data identifier should be set to 1 for system interface commands and data identifiers associated with external requests. For system interface commands and data identifiers associated with processor requests, reserved bits and reserved fields in the command and data identifier are undefined.

Command and Data Identifier Syntax

System interface commands and data identifiers are encoded in 9 bits and are transmitted on the **SysCmd** bus from the processor to an external agent, or from an external agent to the processor, during address and data cycles. Bit 8 (the most-significant bit) of the **SysCmd** bus determines whether the current content of the **SysCmd** bus is a command or a data identifier and, therefore, whether the current cycle is an address cycle or a data cycle. For system interface commands, **SysCmd(8)** must be set to 0. For system interface data identifiers, **SysCmd(8)** must be set to 1.

Notes

System Interface Command Syntax

This section describes the **SysCmd** bus encoding for system interface commands. Figure 14.17 shows a common encoding used for all system interface commands.

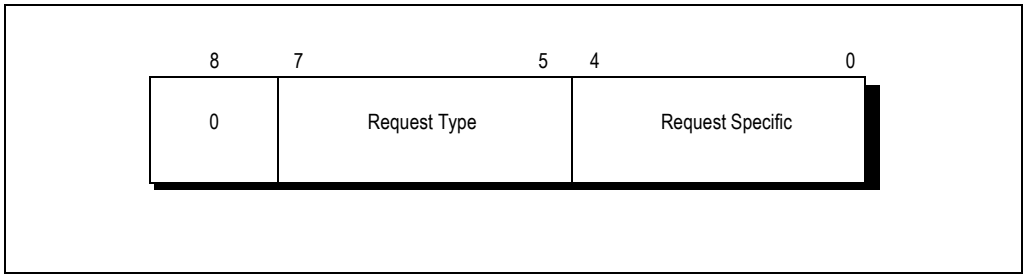


Figure 14.17 System Interface Command Syntax Bit Definition

SysCmd(8) must be set to 0 for all system interface commands. **SysCmd(7:5)** specify the system interface request type which may be read, write or null. Table 14.6 shows the types of requests encoded by the **SysCmd(7:5)** bits. **SysCmd(4:0)** are specific to each type of request.

SysCmd(7:5)	Command
0	Read Request
1	Reserved
2	Write Request
3	Null Request
4 - 7	Reserved

Table 14.6 Encoding of SysCmd (7:5) for System Interface Commands

Write Requests

Figure 14.18 shows the format of a **SysCmd** write request.

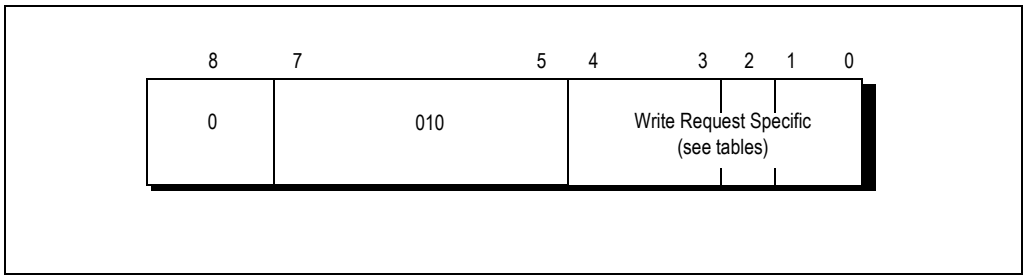


Figure 14.18 Write Request SysCmd Bus Bit Definition

Table 14.7 lists the write attributes encoded in bits **SysCmd(4:3)**.

SysCmd(4:3)	Write Attributes
0	Reserved
1	Reserved
2	Block write
3	64-bit mode: Doubleword, partial doubleword, word, or partial word 32-bit bus mode: Word or partial word.

Table 14.7 Write Request Encoding of SysCmd (4:3)

Notes

Table 14.8 lists the block write replacement attributes encoded in bits **SysCmd(2:0)**.

SysCmd(2)	Cache Line Replacement Attributes
0	Cache line replaced
1	Cache line retained
SysCmd(1:0)	**Write Block Size**
0	Reserved
1	8 words
2 - 3	Reserved

Table 14.8 Block Write Request Encoding of SysCmd (2:0)

Table 14.9 lists the write request bit encoding in **SysCmd(2:0)**.

SysCmd(2:0)	Read Data Size
0 1 2 3	64-bit or 32-bit bus mode: 1 byte valid (Byte) 2 bytes valid (Halfword) 3 bytes valid (Tribyte) 4 bytes valid (Word)
4 5 6 7	64-bit mode only: 5 bytes valid (Quintibyte) 6 bytes valid (Sextibyte) 7 bytes valid (Septibyte) 8 bytes valid (Doubleword)

Table 14.9 Doubleword, Word, or Partial-Word Write Request Data Size Encoding of SysCmd (2:0)

System Interface Data Identifier Syntax

This section defines the encoding of the SysCmd bus for system interface data identifiers. Figure 14.19 shows a common encoding scheme that is used for all system interface data identifiers.

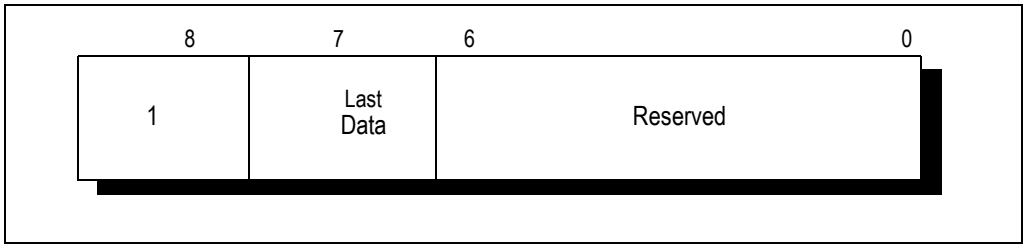


Figure 14.19 Data Identifier SysCmd Bus Bit Definition

Data Identifier Bit Definitions

As shown in Table 14.10, when set to 0, SysCmd(7) marks the last data element.

SysCmd(7)	Last Data Element Indication
0	Last data element
1	Not the last data element
SysCmd(6:0)	Reserved

Table 14.10 Processor Data Identifier Encoding of SysCmd(7)

The External Request Interface

Notes

Introduction

An external request includes reads, writes and null requests, as shown in Figure 15.1. ***Read*** requests ask for a word of data from the processor's internal resource. ***Write*** requests provide a word of data to be written to the processor's internal resource. The ***Null*** request requires no action by the processor; it provides a mechanism for the external agent to return control of the system interface to the master state without affecting the processor. In addition to the read, write and null requests, this chapter includes a description of the processor read response, a special case external request.

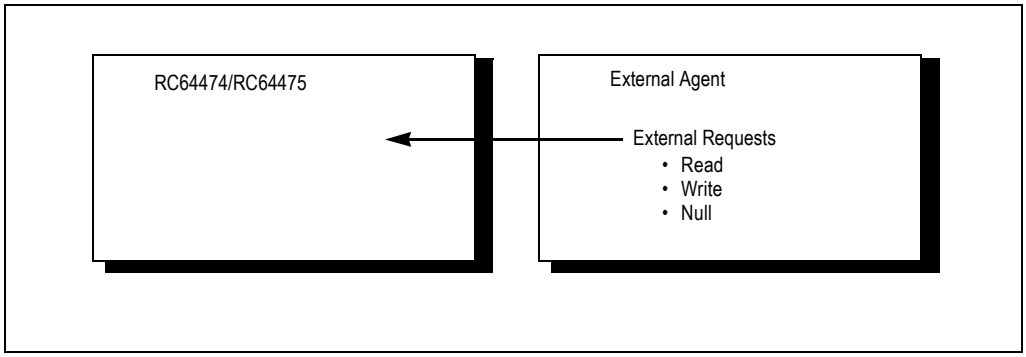


Figure 15.1 External Requests

The processor controls the flow of these external requests through the arbitration signals **ExtRqst*** and **Release***, as shown in Figure 15.2. The external agent must acquire mastership of the system interface before it is allowed to issue an external request; the external agent arbitrates for mastership of the system interface by asserting **ExtRqst*** and then waiting for the processor to assert **Release*** for one cycle.

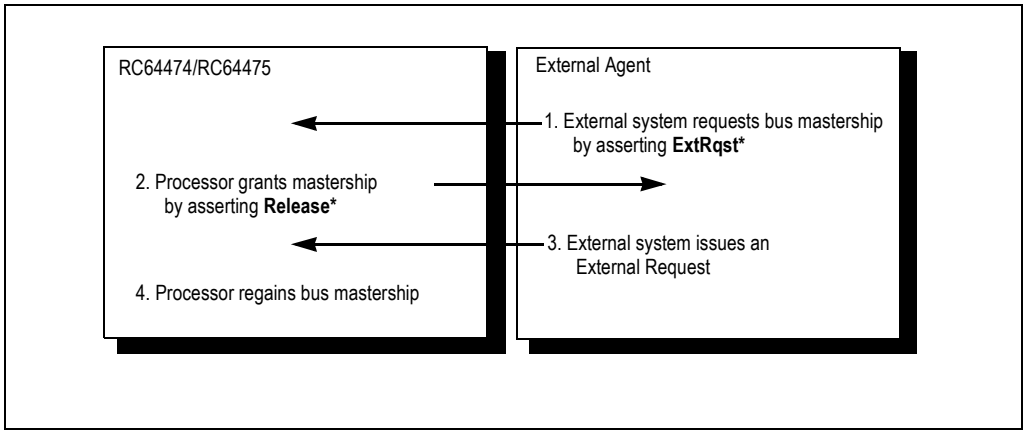


Figure 15.2 Processor Control of External Request, through arbitration signals

Mastership of the system interface always returns to the processor after an external request is issued. The processor does not accept a subsequent external request until it has completed the current request. If there are no processor requests pending, the processor decides, based on its internal state, whether to accept the external request, or to issue a new processor request. The processor can issue a new processor request even if the external agent is requesting access to the system interface.

The external agent asserts **ExtRqst*** indicating that it wishes to begin an external request. The external agent then waits for the processor to signal that it is ready to accept this request by asserting **Release***.

Notes

The processor signals that it is ready to accept an external request based on the following criteria:

- *The processor completes any processor request that is in progress.*
- *While waiting for the assertion of **RdRdy*** to issue a processor read request, the processor can accept an external request if the request is delivered to the processor one or more cycles before **RdRdy*** is asserted.*
- *While waiting for the assertion of **WrRdy*** to issue a processor write request, the processor can accept an external request provided the request is delivered to the processor one or more cycles before **WrRdy*** is asserted.*
- *If waiting for the response to a read request after the processor has made an uncompelled change to a slave state, the external agent can issue an external request before providing the read response data.*

External Read Request

In contrast to a processor read request, data is returned directly in response to an external read request; no other requests can be issued until the processor returns the requested data. An external read request is complete after the processor returns the requested word of data. The data identifier associated with the response data can signal that the returned data is erroneous, causing the processor to take a bus error.

Note: The RC64474/RC64475 does not contain any resources that are readable by an external read request; in response to an external read request the processor returns undefined data and a data identifier with its *Erroneous Data* bit, **SysCmd(5)**, set.

External Write Request

When an external agent issues a write request, the specified resource is accessed and the data is written to it. An external write request is complete after the word of data has been transmitted to the processor. The only processor resource available to an external write request is the IP field of the Cause register.

Read Response

A *read response* returns data in response to a processor read request, as shown in Figure 15.3. While a read response is technically an external request, it has one characteristic that differentiates it from all other external requests—it does not perform system interface arbitration. For this reason, read responses are handled separately from all other external requests, and are simply called read responses. When a read response comes back with bad parity for the first datum, a cache error exception results.

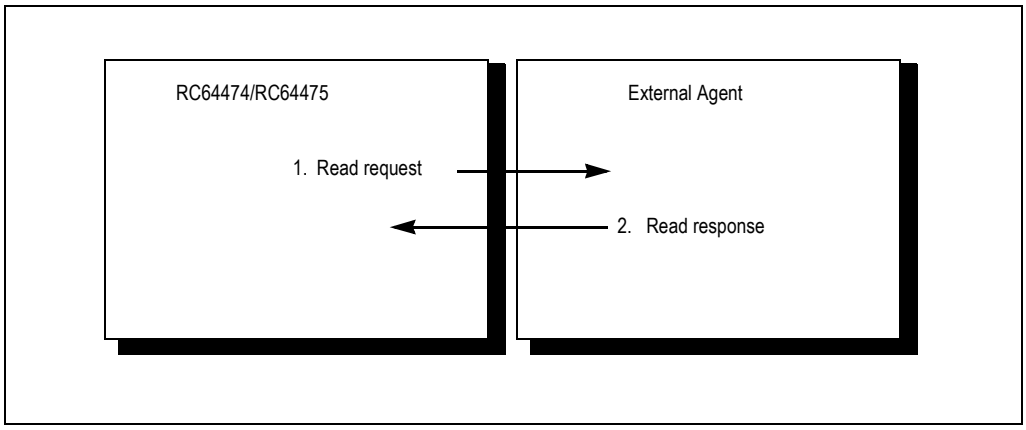


Figure 15.3 Read Response

Processor and External Request Protocols

The following sections contain a cycle-by-cycle description of the bus arbitration protocols for each type of processor and external request. Table 15.1 lists the abbreviations and definitions for each of the buses that are used in the timing diagrams that follow.

Notes

Scope	Abbreviation	Meaning
Global	Unsd	Unused
SysAD bus	Addr	Physical address
	Data<n>	Data element number n of a block of data
SysCmd bus	Cmd	An unspecified system interface command
	Read	A processor or external read request command
	Write	A processor or external write request command
	SINull	A system interface release external null request command
	NData	A noncoherent data identifier for a data element other than the last data element
	NEOD	A noncoherent data identifier for the last data element

Table 15.1 System Interface Requests

External Request Protocols

This section describes the following external request protocols:

- *read*
- *null*
- *write*
- *read response*

External requests can only be issued with the system interface in slave state. An external agent asserts **ExtRqst*** to arbitrate (see the External Arbitration Protocol) for the system interface, then waits for the processor to release the system interface to slave state by asserting **Release*** before the external agent issues an external request. If the system interface is already in slave state (that is, the processor has previously performed an uncompelled change to slave state due to a read operation) the external agent can begin an external request immediately.

After issuing an external request, the external agent must return the system interface to master state. If the external agent does not have any additional external requests to perform, **ExtRqst*** must be deasserted two cycles after the cycle in which **Release*** was asserted. For a string of external requests, the **ExtRqst*** signal is asserted until the last request cycle, whereupon it is deasserted two cycles after the cycle in which **Release*** was asserted.

The processor continues to handle external requests as long as **ExtRqst*** is asserted; however, the processor cannot release the system interface to slave state for a subsequent external request until it has completed the current request. As long as **ExtRqst*** is asserted, the string of external requests is not interrupted by a processor request. The protocol is the same for either 64-bit or 32-bit bus interface mode.

External Arbitration Protocol

System interface arbitration uses the signals **ExtRqst*** and **Release*** as described above. Figure 15.4 is a timing diagram of the arbitration protocol, in which slave and master states are shown.

The arbitration cycle consists of the following steps:

1. The external agent asserts **ExtRqst*** when it wishes to submit an external request.
2. The processor waits until it is ready to handle an external request, whereupon it asserts **Release*** for one cycle.
3. The processor sets the **SysAD** and **SysCmd** buses to tri-state.
4. The external agent must begin driving the **SysAD** bus and the **SysCmd** bus two cycles after the assertion of **Release***.
5. The external agent deasserts **ExtRqst*** two cycles after the assertion of **Release***, unless the external agent wishes to perform an additional external request.
6. The external agent sets the **SysAD** and the **SysCmd** buses to tri-state at the completion of an external request.

Notes

The processor can start issuing a processor request one cycle after the external agent sets the bus to tri-state.

Note: Timings for the **SysADC** and **SysCmdP** buses are the same as those for the **SysAD** and **SysCmd** buses, respectively. The protocol is the same for 64-bit and 32-bit bus interface mode.

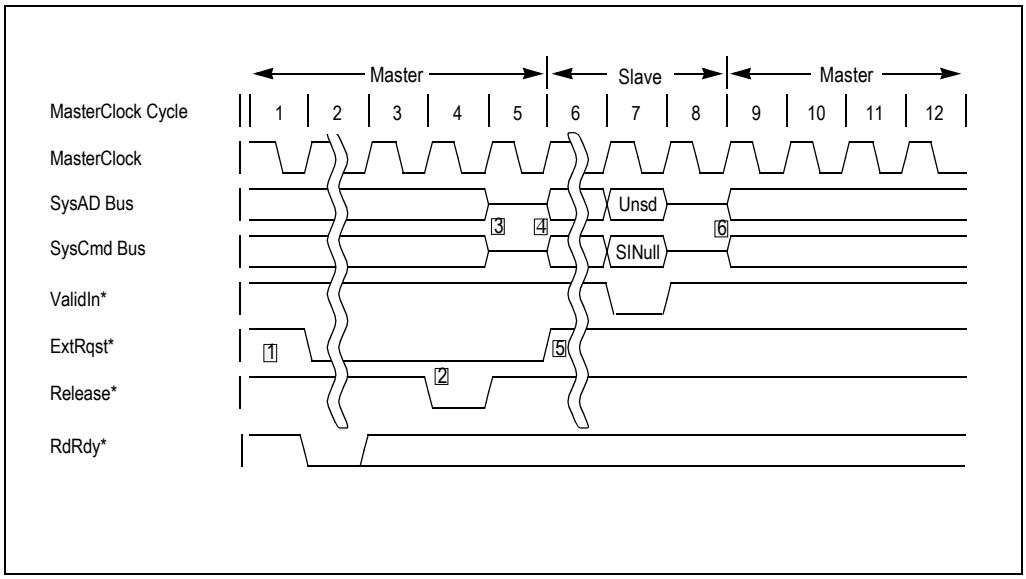


Figure 15.4 Arbitration Protocol for External Requests

External Read Request Protocol

External reads are requests for a word of data from a processor internal resource, such as a register. External read requests cannot be split; that is, no other request can occur between the external read request and its read response.

Figure 15.5 shows a timing diagram of an external read request, which consists of the following steps:

1. An external agent asserts **ExtRqst*** to arbitrate for the system interface.
2. The processor releases the system interface to slave state by asserting **Release*** for one cycle and then deasserting **Release***.
3. After **Release*** is deasserted, the **SysAD** and **SysCmd** buses are set to a tri-state for one cycle.
4. The external agent drives a read request command on the **SysCmd** bus and a read request address on the **SysAD** bus and asserts **ValidIn*** for one cycle.
5. After the address and command are sent, the external agent releases the **SysCmd** and **SysAD** buses by setting them to tri-state and allowing the processor to drive them. The processor, having accessed the data that is the target of the read, returns this data to the external agent. The processor accomplishes this by driving a data identifier on the **SysCmd** bus, the response data on the **SysAD** bus, and asserting **ValidOut*** for one cycle. The data identifier indicates that this is last-data-cycle response data.
6. The system interface is in master state. The processor continues driving the **SysCmd** and **SysAD** buses after the read response is returned.

Note: Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

External read requests are only allowed to read a (32-bit) word of data from the processor. The processor response to external read requests is undefined for any data element other than a word. In 64-bit or 32-bit bus mode this operation is only a single external read request to the processor. In both modes **SysAD(31:0)** provides the address of the internal resource that is to be read.

Note: The processor does not contain resources that are readable by an external read request. In response to an external read request the processor returns undefined data and a data identifier that has its *erroneous data bit*, **SysCmd(5)**, set. This will also cause the CPU to take an error data exception.

Notes

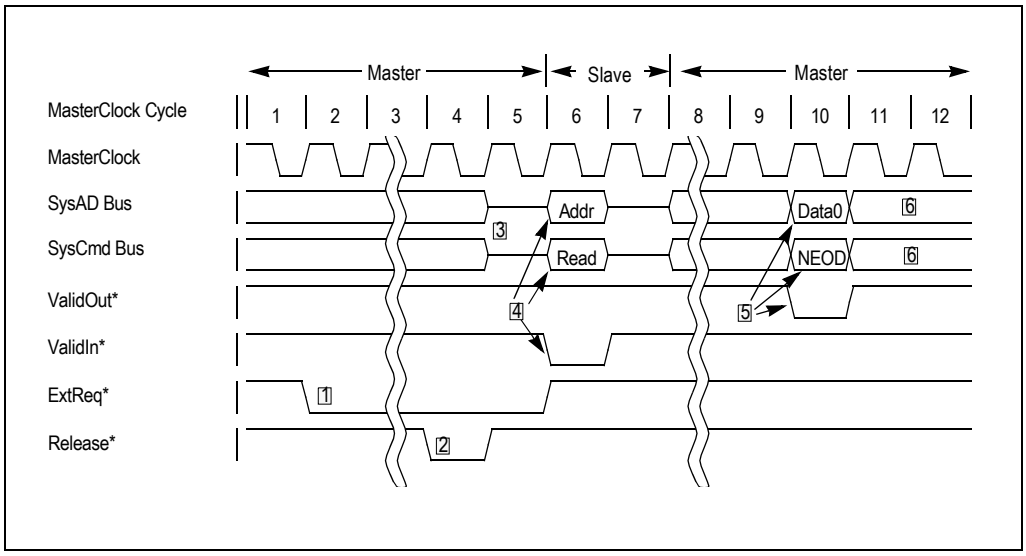


Figure 15.5 External Read Request, System Interface in Master State

External Null Request Protocol

The RC64474/RC64475 only support one external null request. A *system interface release external null request* returns the system interface to master state from slave state without otherwise affecting the processor.

External null requests require no action from the processor other than to return the system interface to master state.

Figure 15.6 show timing diagram of the external null request cycle, which consist of the following steps:

1. The external agent asserts **ExtRqst*** to arbitrate for the system interface.
2. The processor releases the system interface to slave state by asserting **Release***.
3. The external agent drives a system interface release external null request command on the **SysCmd** bus, and asserts **ValidIn*** for one cycle to return the system interface back to master state.
4. The **SysAD** bus is unused (does not contain valid data) during the address cycle associated with an external null request.
5. After the address cycle is issued, the null request is complete.

For a *system interface release external null request*, the external agent releases the **SysCmd** and **SysAD** buses, and expects the system interface to return to master state. This protocol is the same for both 64-bit and 32-bit bus modes.

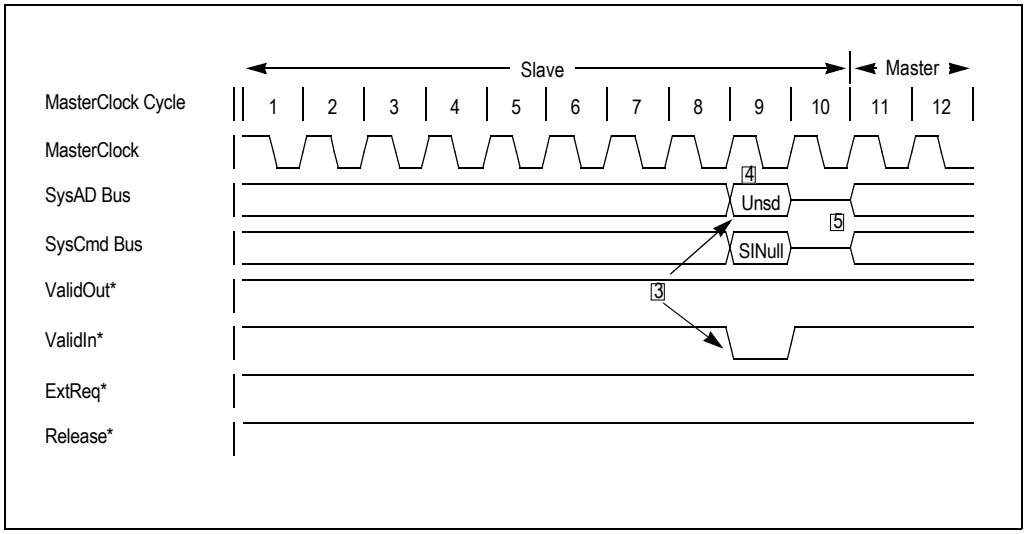


Figure 15.6 System Interface Release External Null Request

Notes

External Write Request Protocol

External write requests use a protocol identical to the processor single word write protocol except the **ValidIn*** signal is asserted instead of **ValidOut***. Figure 15.7 on page 15-6 shows a timing diagram of an external write request, which consists of the following steps:

1. The external agent asserts **ExtRqst*** to arbitrate for the system interface.
2. The processor releases the system interface to slave state by asserting **Release***.
3. The external agent drives a write command on the **SysCmd** bus, a write address on the **SysAD** bus, and asserts **ValidIn***.
4. The external agent drives a data identifier on the **SysCmd** bus, data on the **SysAD** bus, and asserts **ValidIn***.
5. The data identifier associated with the data cycle must contain a coherent or noncoherent last data cycle indication.
6. After the data cycle is issued, the write request is complete and the external agent sets the **SysCmd** and **SysAD** buses to a tri-state, allowing the system interface to return to master state. Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

External write requests are only allowed to write a (32-bit) word of data to the processor. Processor behavior in response to an external write request for any data element other than a word is undefined. In 64-bit and 32-bit bus mode **SysAD(31:0)** is used for both the address and the data portions of the external write request, regardless of the "endianness" of the system.

Note: The interrupt register is the only processor internal resource available for write access by an external request.

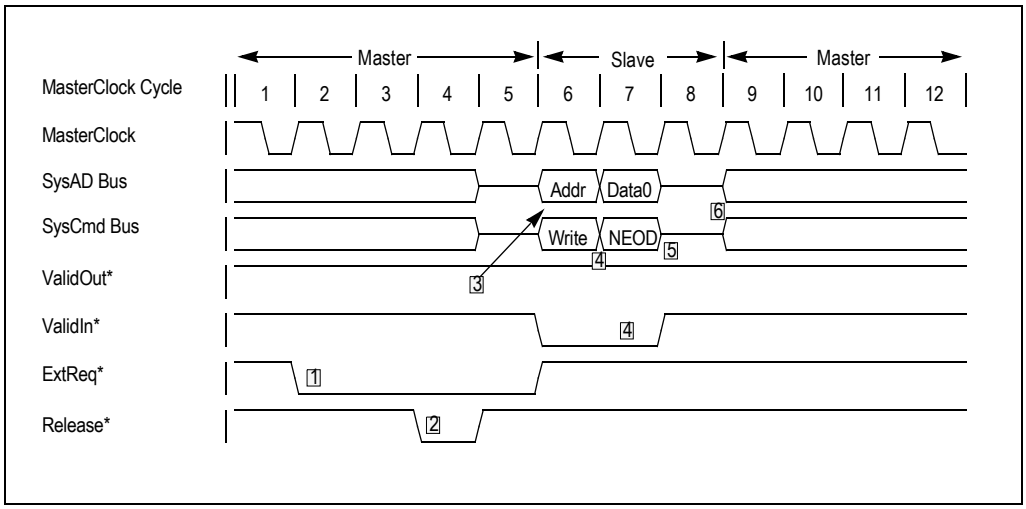


Figure 15.7 External Write Request, with System Interface Initially in Master State

Read Response Protocol

An external agent must return data to the processor in response to a processor read request by using a read response protocol. The read response protocol is discussed in detail in Chapter 12, "The Read Interface."

Interface Commands & Data Identifiers

System interface commands specify the nature and attributes of any system interface request; this specification is made during the address cycle for the request. System interface data identifiers specify the attributes of data transmitted during a system interface data cycle.

The following sections describe the syntax, that is, the bitwise encoding, of system interface commands and data identifiers. The same **SysCmd** encoding is used for both 32-bit and 64-bit bus mode. The selection of 64-bit versus 32-bit is not dynamic and should be done only once during **Reset**. The RC64474/RC64475 do not indicate externally whether the bus is configured as 32-bit or 64-bit.

Notes

Reserved bits and reserved fields in the command or data identifier should be set to 1 for system interface commands and data identifiers associated with external requests. For system interface commands and data identifiers associated with processor requests, reserved bits and reserved fields in the command and data identifier are undefined.

Command and Data Identifier Syntax

System interface commands and data identifiers are encoded in 9 bits and are transmitted on the **SysCmd** bus from the processor to an external agent, or from an external agent to the processor, during address and data cycles. Bit 8 (the most-significant bit) of the **SysCmd** bus determines whether the current content of the **SysCmd** bus is a command or a data identifier and, therefore, whether the current cycle is an address cycle or a data cycle. For system interface commands, **SysCmd(8)** must be set to 0. For system interface data identifiers, **SysCmd(8)** must be set to 1.

System Interface Command Syntax

This section describes the **SysCmd** bus encoding for system interface commands. Figure 15.8 shows a common encoding used for all system interface commands.

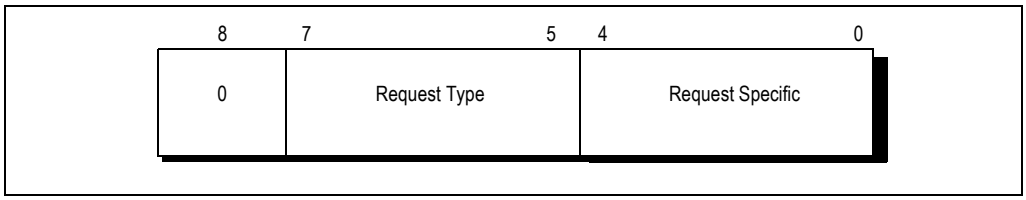


Figure 15.8 System Interface Command Syntax Bit Definition

SysCmd(8) must be set to 0 for all system interface commands. **SysCmd(7:5)** specify the system interface request type which may be read, write or null; Table 15.2 lists the encoding of **SysCmd(7:5)**. Table 15.2 shows the types of requests encoded by the **SysCmd(7:5)** bits.

SysCmd(7:5)	Command
0	Read Request
1	Reserved
2	Write Request
3	Null Request
4 - 7	Reserved

Table 15.2 Encoding of SysCmd (7:5) for System Interface Commands

SysCmd(4:0) are specific to each type of request and are defined in each of the following sections.

Null Requests

Figure 15.9 shows the format of a **SysCmd** null request.

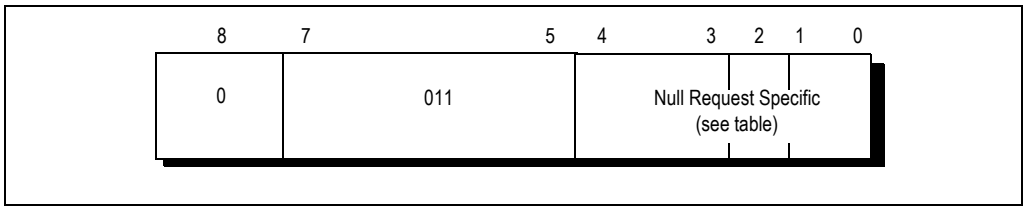


Figure 15.9 Null Request SysCmd Bus Bit Definition

System interface release external null requests use the null request command. Table 15.3 lists the encoding of **SysCmd(4:3)** for external null requests. **SysCmd(2:0)** are reserved for both instances of null requests.

Notes

SysCmd(4:3)	Null Attributes
0	System Interface release
1 - 3	Reserved

Table 15.3 External Null Request Encoding of SysCmd (4:3)

System Interface Data Identifier Syntax

This section defines the encoding of the **SysCmd** bus for system interface data identifiers. Figure 15.10 shows a common encoding scheme used for all system interface data identifiers.

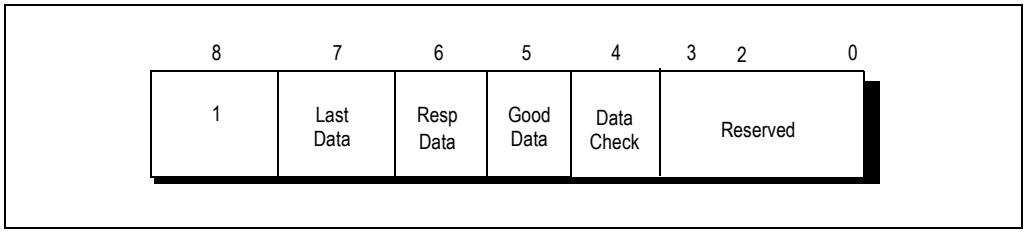


Figure 15.10 Data Identifier SysCmd Bus Bit Definition

Note: SysCmd(8) must be set to 1 for all system interface data identifiers. System interface data identifiers use the format for noncoherent data.

Noncoherent Data

Noncoherent data is defined as follows:

- *data that is associated with processor block write requests and processor doubleword, partial doubleword, word, or partial word write requests*
- *data that is returned in response to a processor noncoherent block read request or a processor doubleword, partial doubleword, word, or partial word read request*
- *data that is associated with external write requests*
- *data that is returned in response to an external read request*

Data identifier Bit Definitions

- ***SysCmd(7)*** *marks the last data element and* ***SysCmd(6)*** *indicates whether or not the data is response data, for both processor and external coherent and noncoherent data identifiers. Response data is data returned in response to a read request.*
- ***SysCmd(5)*** *indicates whether or not the data element is error free. Erroneous data contains an uncorrectable error and is returned to the processor, forcing a bus error. The processor delivers data with the good data bit deasserted if a primary parity error is detected for a transmitted data item.*
- ***SysCmd(4)*** *indicates to the processor whether to check the data and check bits for this data element.*
- ***SysCmd(3)*** *is reserved for external data identifiers.*
- ***SysCmd(4:3)*** *are reserved for noncoherent processor data identifiers.*
- ***SysCmd(2:0)*** *are reserved for noncoherent data identifiers.*

Table 15.4 lists the encoding of **SysCmd(7:3)** for processor data identifiers.

Notes

SysCmd(7)	Last Data Element Indication
0	Last data element
1	Not the last data element
SysCmd(6)	**Response Data Indication**
0	Data is response data
1	Data is not response data
SysCmd(5)	**Good Data Indication**
0	Data is error free
1	Data is erroneous
SysCmd(4:3)	Reserved

Table 15.4 Processor Data Identifier Encoding of SysCmd (7:3)

Table 15.5 lists the encoding of **SysCmd(7:3)** for external data identifiers.

SysCmd(7)	Last Data Element Indication
0	Last data element
1	Not the last data element
SysCmd(6)	**Response Data Indication**
0	Data is response data
1	Data is not response data
SysCmd(5)	**Good Data Indication**
0	Data is error free
1	Data is erroneous
SysCmd(4)	**Data Checking Enable**
0	Check the data and check bits
1	Do not check the data and check bits
SysCmd(3)	Reserved

Table 15.5 External Data Identifier Encoding of SysCmd (7:3)

System Interface Addresses

System interface addresses are full 32-bit physical addresses presented on the least-significant 32 bits (bits 31 through 0) of the **SysAD** bus during address cycles; the remaining bits of the **SysAD** bus are unused during address cycles.

Addressing Conventions

Addresses associated with doubleword, partial doubleword, word, or partial word transactions, are aligned for the size of the data element. The system uses the following address conventions:

- *Addresses associated with block requests are aligned to double-word boundaries; that is, the low-order 3 bits of address are 0.*
- *Doubleword requests set the low-order 3 bits of address to 0.*
- *Word requests set the low-order 2 bits of address to 0.*

Notes

- *Halfword requests set the low-order bit of address to 0.*
- *Byte, tribyte, quintibyte, sextibyte, and septibyte requests use the byte address.*

Processor Internal Address Map

External reads and writes provide access to processor internal resources that may be of interest to an external agent. The processor decodes bits **SysAD(6:0)** of the address associated with an external read or write request to determine which processor internal resource is the target.

However, the RC64474/RC64475 do not contain resources that are *readable* through an external read request. In response to an external read request the processor returns 1) undefined data, 2) a data identifier that has its *Erroneous Data* bit, **SysCmd(5)**, set, and then 3) takes an exception. The *Interrupt* register is the only processor internal resource available for *write* access by an external request. The *Interrupt* register is accessed by an external write request with an address of 000_2 on bits 6:4 of the **SysAD** bus. The interrupt register is described in detail in Chapter 15, “Processor Interrupts.”

Processor Interrupts

Notes

Introduction

The RC64474/475 processors support six hardware interrupts, one internal “timer interrupt,” two software interrupts, and one unmasked/nonmaskable enabled interrupt. The processor takes an exception on any interrupt. This chapter describes the six hardware and single nonmaskable interrupts. A description of the software and the timer interrupts can be found in Chapter 5. CPU exception processing is also described in Chapter 6. Floating-point exception processing is described in Chapter 7.

The six CPU hardware interrupts can either be caused by external write requests to the RC64474/475 or through dedicated interrupt pins, which are latched into an internal register by the rising edge of **MasterClock**. The nonmaskable interrupt (NMI) is caused either by an external write request to the RC64474/475 or by a dedicated pin in the RC64474/475. This pin is also latched into an internal register by the rising edge of **MasterClock**.

Asserting Interrupts

External writes to the CPU are directed to various internal resources, based on an internal address map of the processor. When **SysAD[6:0]** = 0 during an ADDR cycle of external write request, an external write to any address writes to an architecturally transparent register called the *Interrupt* register; this register is available for external write cycles, but not for external reads.

During a data cycle, **SysAD[22:16]** are the write enables for the seven individual *Interrupt* register bits (0 = disabled, 1 = enabled) and **SysAD[6:0]** are the values to be written into these bits (0 = no interrupt, 1 = interrupt). This allows any subset of the *Interrupt* register to be set or cleared with a single write request. Figure 16.1 shows the mechanics of an external write to the *Interrupt* register.

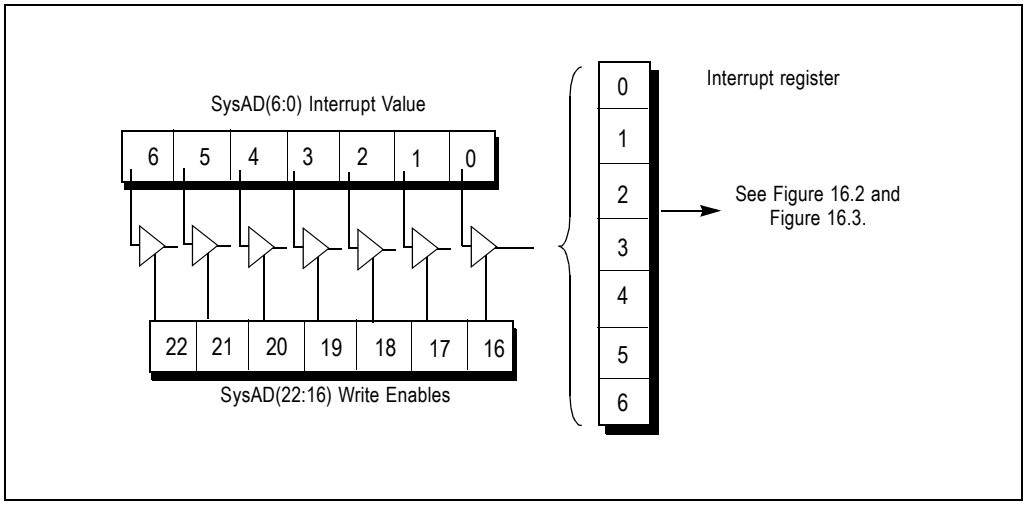


Figure 16.1 Interrupt Register Bits and Enables

Figure 16.2 shows how the RC64474/RC64475 interrupts are readable through the *Cause* register. The interrupt bits, **Int*(5:0)**, are latched into the internal register by the rising edge of **MasterClock**.

- *Bit 5 of the Interrupt register in the RC64474/RC64475 is ORed with the **Int*(5)** pin and then multiplexed with the internal **TimerInterrupt** signal. This result is directly readable as bit 15 of the Cause register.*
- *Bits 4:0 of the Interrupt register are bit-wise ORed with the current value of the interrupt pins **Int*[4:0]** and the result is directly readable as bits 14:10 of the Cause register.*

Notes

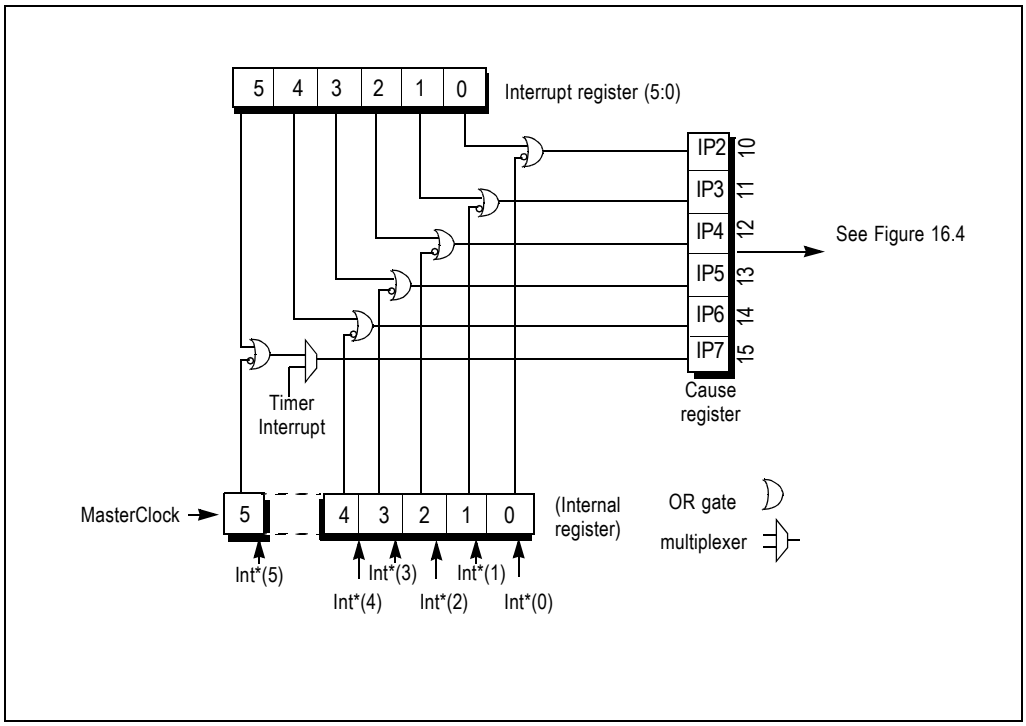


Figure 16.2 RC64474/RC64475 Interrupt Signals

Figure 16.3 shows the internal derivation of the nonmaskable (**NMI)** signal, for the RC64474/RC64475 processor. The **NMI*** pin is latched into an internal register by the rising edge of **MasterClock**. Bit 6 of the *Interrupt* register is then ORed with the inverted value of **NMI*** to form the nonmaskable interrupt. Only the one falling edge of the latched signal will cause the NMI.

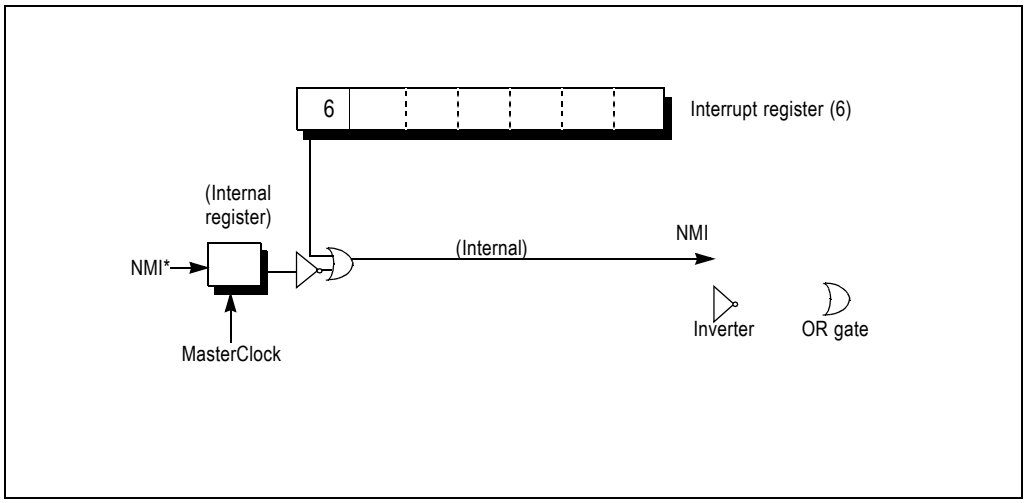


Figure 16.3 RC64474/RC64475 Nonmaskable Interrupt Signal

Figure 16.4 shows the masking of the RC64474/RC64475 interrupt signal.

- *Cause register bits 15:8 (IP7-IP0) are AND-ORed with Status register interrupt mask bits 15:8 (IM7-IM0) to mask individual interrupts.*
- *Status register bit 0 is a global Interrupt Enable (IE). It is ANDed with the output of the AND-OR logic to produce the RC64474/RC64475 interrupt signal.*

Notes

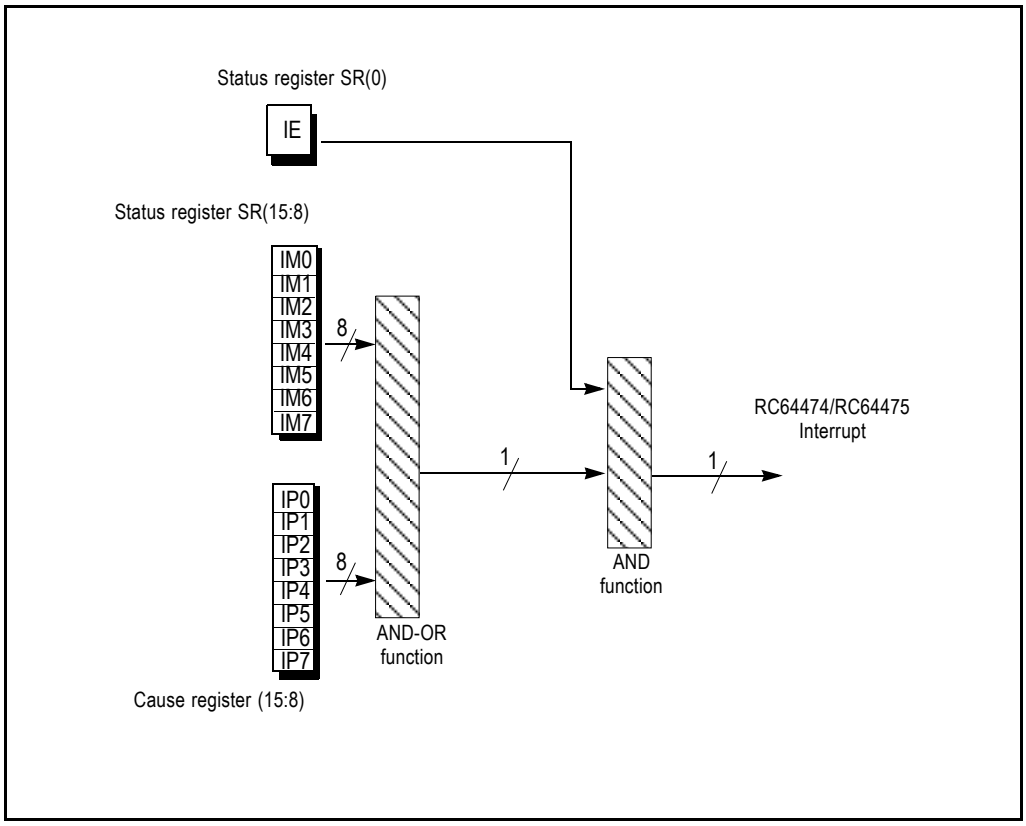


Figure 16.4 Masking of the RC64474/RC64475 Interrupts

Notes

Processor Error Checking

Notes

Introduction

Error checking codes allow the processor to detect and sometimes correct errors made when moving data from one place to another. Two major types of data errors can occur in data transmission:

- *hard errors, which are permanent, arise from broken interconnects, internal shorts, or open leads*
- *soft errors, which are transient, are caused by system noise, power surges, and alpha particles.*

Hard errors must be corrected by physical repair of the damaged equipment and restoration of data from backup. Soft errors can be corrected by using error checking and correcting codes. **For error checking only, both the RC64474 and RC64475 use even parity**.

Parity Error Detection

Parity is the simplest error detection scheme. By appending a bit to the end of an item of data—called a *parity bit*—single bit errors can be detected; however, these errors cannot be corrected.

There are two types of parity:

- ***Odd Parity** adds 1 to any even number of 1s in the data, making the total number of 1s odd (including the parity bit).*
- ***Even Parity** adds 1 to any odd number of 1s in the data, making the total number of 1s even (including the parity bit).*

Odd and even parity values are as follows:

Data(3:0)	Odd Parity Bit	Even Parity Bit
0010	0	1

Table 17.1 shows a single bit in **Data(3:0)** with a value of 1; this bit is **Data(1)**.

- *In even parity, the parity bit is set to 1. This makes 2 (an even number) the total number of bits with a value of 1.*
- *Odd parity makes the parity bit a 0 to keep the total number of 1-value bits an odd number—in the case shown above, the single bit **Data(1)**.*

Data(3:0)	Odd Parity Bit	Even Parity Bit
0110	1	0
0000	1	0
1111	1	0
1101	0	1

Table 17.1 Odd and Even Parity Bits for Various Data Values

Parity allows single-bit error detection, but it does not indicate which bit is in error. For example, suppose an odd-parity value of 00011 arrives. The last bit is the parity bit, and since odd parity demands an odd number (1,3,5) of 1s, this data is in error: it has an even number of 1s. However, it is impossible to tell *which* bit is in error. The processor does verify data correctness by using even parity as it passes data from/to the system interface to/from the primary caches.

Notes

System Interface

The processor generates correct check bits for doubleword, word, or partial-word data transmitted to the system interface. As it checks for data correctness, the processor passes data check bits from the primary cache, directly without changing the bits, to the system interface. The processor does not check data received from the system interface for external writes. By setting the *NChck* bit in the data identifier, it is possible to prevent the processor from checking read response data from the system interface.

For cache refill, if the NChck bit is set, the CPU will generate correct parity before placing data into the cache. The RC64474/RC64475 only checks parity for the first double word returned on a block instruction fetch, that is, for the double word that contains the instruction that was missed on in the cache. This double word is checked just as if it had been read out of the cache. This parity check is done as a byte parity check. For single read, and with the NChck bit set, the CPU will check parity for all 64-bit, even if the transfer size is less than that.

When the RC64474/RC64475 is checking parity, it does not actually regenerate the word parity, but rather turns the byte parity supplied by the system into word parity. It XORS the bits in groups of four. As a result, if bad byte parity is supplied by the system, bad word parity will get written into the cache. This is done to be consistent with what happens in the DCache.

The processor does not check addresses received from the system interface and does not generate correct check bits for addresses transmitted to the system interface. The processor does not contain a data corrector; instead, the processor takes a cache error exception when it detects an error based on data check bits. Software is responsible for error handling.

System Interface Command Bus

In the RC64474/RC64475 processor, the system interface command bus has no parity. **SysCmdP** always drives zero out for CPU valid cycles and is not checked when the system interface is in slave state. Error checking operations are summarized in Table 17.2 and Table 17.3.

Bus	Uncached Load	Uncached Store	Primary Cache Load from System Interface	Primary Cache Write to System Interface	Cache Instruction
Processor Data	From System Interface	Not Checked	From System Interface unchanged	Checked; Trap on Error	Check on cache writeback; Trap on Error
System Interface Address/ Command and Check Bits: Transmit	Not Generated	Not Generated	Not Generated	Not Generated	Not Generated
System nterface Address/ Command and Check Bits: Receive	Not Checked	NA	Not Checked	NA	NA
System Interface Data	Checked; Trap on Error	From Processor	Checked; Trap on Error	From Primary Cache	From Primary Cache
System Interface Data Check Bits	Checked; Trap on Error	Generated	Checked; Trap on Error	From Primary Cache	From Primary Cache

Table 17.2 Error Checking and Correcting Summary for Internal Transactions

Notes

Bus	Read Request	Write Request
Processor Data	NA	NA
System Interface Address, Command, and Check Bits: Transmit	Generated	NA
System Interface Address, Command, and Check Bits: Receive	Not Checked	Not Checked
System Interface Data	From Processor	Checked; Trap on Error
System Interface Data Check Bits	Generated	Checked; Trap on Error

Table 17.3 Error Checking and Correcting Summary for External Transactions

Notes

Standard JTAG Support Interface

Notes

Introduction

The standard JTAG boundary scan is used for on-chip debugging during Run-time mode. On the RC64474/RC64475, the following six additional pins—TDI, TDO, TMS, TCK, TRST* and JTAG32—have been added to support the implementation of this interface.

Note: The JTAG boundary scan should only be used when the RC64474/RC64475 is in the reset mode.

TDI	I	**JTAG Data In** On the rising edge of TCK, serial input data are shifted into either the Instruction register or Data register, depending on the TAP controller state.
TDO	O	**JTAG Data Out** On the falling edge of TCK, the TDO is serial data shifted out from either the instruction or data register. When no data is shifted out, the TDO is tri-stated (high impedance).
TCK	I	**JTAG Clock Input** An input test clock used to shift into or out of the boundary-scan register cells. TCK is independent of the system and processor clock with nominal 40-60% duty cycle.
TMS	I	**JTAG Command Select** The logic signal received at the TMS input is decoded by the TAP controller to control test operation. TMS is sampled on the rising edge of TCK.
TRST*	I	**JTAG Reset to Reset Circuitry** The TRST* pin is an active-low signal used for asynchronous reset of the debug unit, independent of the processor logic. During normal CPU operation, the JTAG controller will be held in the reset mode, asserting this active low pin. When asserted low, this pin will also cause the TDO into tristate mode.
JTAG32*	I	**JTAG 32-bit scan** This pin is used to control length of the scan chain for SYsAD (32-bit or 64-bit) for the JTAG mode. When set to Vss, 32-bit bus mode is selected. In this mode, only SysAD(31:0) are part of the scan chain. When set to Vcc, 64-bit bus mode is selected. In this mode, SysAD(63:0) are part of the scan chain. This pin has a built-in pull-down device to guarantee 32-bit scan, if it is left uncovered.
JR_Vcc	I	**JTag VCC** This pin has an internal pull-down to continuously reset the JTAG controller (if left unconnected) bypassing the TRst* pin. When supplied with Vcc, the TRst* pin will be the primary control for the JTAG reset.

Table 18.1 JTAG Interface Pin Descriptions and Type

The JTAG interface consists of the following four basic elements:

- *Test Access Port (TAP)*
- *TAP controller*
- *Instruction Register (IR)*
- *Data Register Port (DR)*

A brief description of each element is provided in the sections that follow. For more complete descriptions, refer to IEEE Standard Test Access port (IEEE Std. 1149.1-1990). Figure 18.1 is an illustration of the standard boundary scan architecture.

Notes

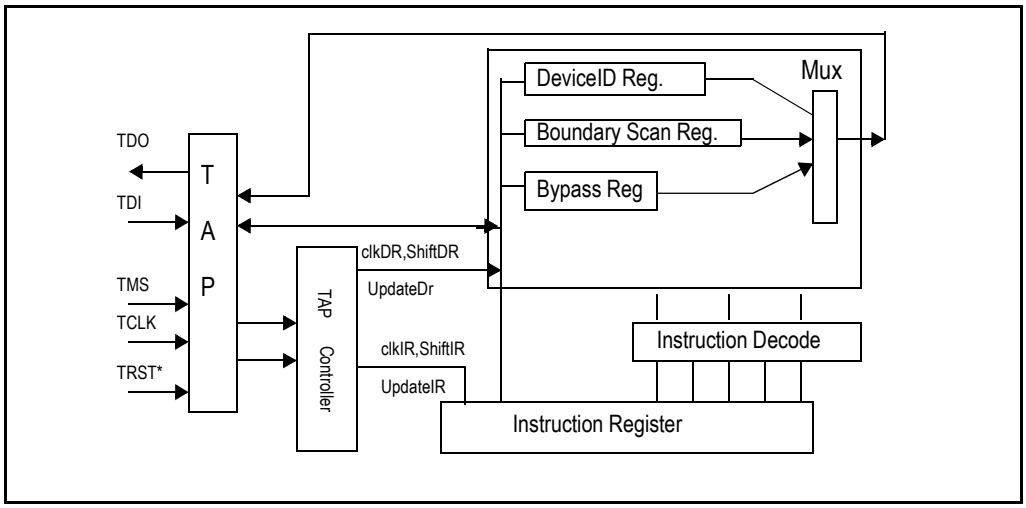


Figure 18.1 Standard Boundary Scan Architecture

Test Access Port (TAP) Interface

The TAP interface is a general-purpose port that provides internal access to the processor. It consists of four input ports (TCLK, TMS, TDI, TRST*) and one output port (TDO). For descriptions of these signals, refer to Table 18.1 on page 18-1.

The Tap Controller

The Tap controller is a synchronous, finite state machine that responds to TMS and TCLK signals, to generate clock and control signals to the Instruction and Data registers as well as other parts of the debug unit. Within the TAP controller, all state transitions occur at the rising edge of the TCLK pulse and, depending on the TMS signal level (0 or 1), it then proceeds to the next state.

The state diagram for the TAP Controller is shown in Figure 18.2.

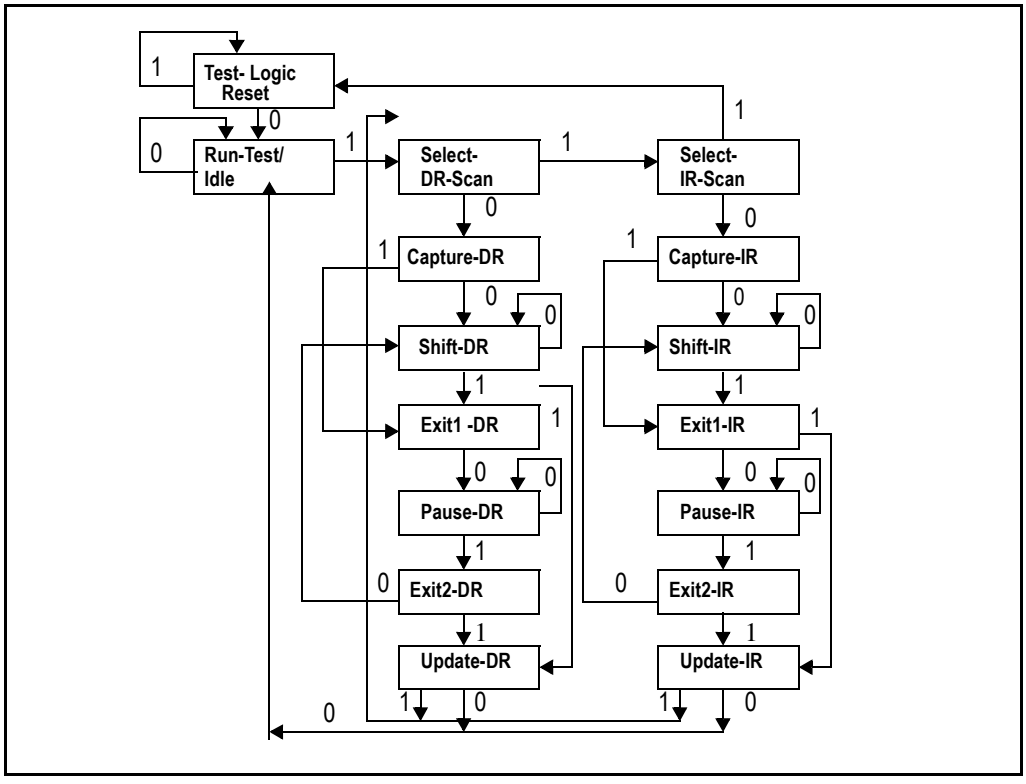


Figure 18.2 TAP Controller State Diagram

Notes

Refer to IEEE Standard Test Access port (IEEE Std. 1149.1), for the full state diagram.

TAP Controller State Assignments

All state transitions within the TAP controller occur at the rising edge of the TCLK pulse and—depending on the TMS signal level (0 or 1)—it proceeds to the next state.

- *Test-Logic-Reset*
  - *The test logic is disabled so that normal operation of the on-chip system logic can continue unhindered.*
- *Run-Test/Idle*
  - *A controller state between scan operations.*
- *Select-DR-Scan*
  - *This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state.*
- *Capture-DR*
  - *In this controller state, data may be parallel-loaded into test data registers selected by the current instruction on the riding edge of TCLK.*
- *Shift-DR*
  - *In this controller state, the test data register connected between TDI and TDO, as a result of the current instruction, shifts data one stage towards its serial output on each rising edge of TCLK. The test data register content is being shifted out serially, LSB first, at the falling edge of TCLK towards the TDO output.*
- *Exit1-DR*
  - *This is a temporary controller state. If TMS is held high, a rising edge applied to TCLK while in this state causes the controller to enter the Update-DR state, which terminates the scanning process. If TMS is held low and a rising edge is applied to TCLK, the controller enters the Pause-DR state.*
- *Pause-DR*
  - *This controller state allows shifting of the test data register in the serial path between TDI and TDO to be temporarily halted.*
- *Exit2-DR*
  - *This is a temporary controller state. If TMS is held high and a rising edge is applied to TCLK while in this state, the scanning process terminates and the TAP controller enters the Update-DR state.*
- *Update-DR*
  - *Data is latched onto the parallel output of these test data registers from the shift-register path on the falling edge of TCLK.*
- *Select-IR-Scan*
  - *This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state.*
- *Capture-IR*
  - *In this controller state, the shift-register contained in the instruction register loads a pattern of fixed logic values on the rising edge to TCLK.*
- *Shift-IR*
  - *In this controller state, the shift-register contained in the instruction register is connected between TDI and TDO and shifts data one stage towards its serial output on each rising edge to TCLK. The instruction shift register content is being shifted out serially, LSB first, at the falling edge of TCLK towards the TDO output.*
- *Exit1-IR*
  - *This is a temporary controller state. While in this state, if TMS is held high, a rising edge applied to TCLK causes the controller to enter the Update-DR state, which terminates the scanning process. If TMS is held low and a rising edge is applied to TCLK, the controller enters the Pause-DR state.*
- *Pause-IR*
  - *This controller state allows shifting of the instruction register to be halted temporarily.*

Notes

- *Exit2-IR*
  - *This is a temporary controller state. While in this state, if TMS is held high and rising edge is applied to TCLK termination of the scanning process occurs. The TAP controller then enters the Update-IR controller state. If TMS is held low and a rising edge is applied to TCLK, the controller enters the Shift-IR state.*
- *Update-IR*
  - *The instruction shifted into the instruction register is latched to the parallel output from the shift-register path on the falling edge of TCLK, in this controller state. Once the new instruction has been latched, it becomes the current instruction.*

Instruction Register (IR)

The Instruction register allows an instruction to be shifted serially into the processor at the rising edge of TCLK. The Instruction is used to select the test to be performed or the test data register to be accessed, or both. The instruction shifted into the register is latched at the completion of the shifting process when the TAP controller is at the *Update-IR* state.

The Instruction register must contain at least two shift-register-based cells that can hold instruction data. These mandatory cells are located near the serial outputs and are the least significant bits. The values of the bits are 0 and 1 (1 is the least significant bit). This register is decoded to perform the following functions:

- *To select test data registers that may operate while the instruction is current. The other test data registers should not interfere with chip operation and selected data registers.*
- *To define the serial test data register path used to shift data between TDI and TDO during data register scanning.*

The Instruction Register is comprised of IR4, IR3, IR2, IR1 and IR0 to decode 32 different possible instructions as follows:

Hex Value	Instruction and Definition	Function
0x00	**EXTEST** Mandatory instruction provided for external circuitry and board-level interconnection checks.	Select Boundary Scan Register
0x01	**IDCODE** Provided to select Device Identification to read out manufacturers identity, part and version number.	Select Chip Identification Data Register
0x02	**Sample/Preload** Mandatory instruction that allows data values to be loaded onto the latched parallel output of the boundary-scan shift register prior to selection of the other boundary-scan test instruction. The Sample instruction allows a snapshot of data flowing from the system pins to the on-chip logic or vice versa.	Select Boundary Scan Register
0x05	**HI-Z** This instruction would place all of the device's output pins into a high impedance state. An external ICE can drive all the pins and would not damage on-chip logic as well as the input pins.	JTAG
0x06	**CLAMP** This instruction allows the state of the signals driven from IC pins to be determined from the boundary-scan register while the bypass register is selected as the serial path between TDI and TDO.	JTAG
0x07	**BYPASS** Bypass mode.	Bypass mode
0x1F	**BYPASS** Bypass mode.	Bypass mode

Table 18.2 Instruction Register Bit Definitions

Note: Any unused instruction is defaulted to the BYPASS instruction.

Notes

Test Data Register (DR)

The Test Data register contains three test data registers:

- *The Bypass register*
- *The Boundary Scan registers*
- *The Device ID register*

As shown in Figure 18.2 on page 18-2, these registers are connected in parallel between a common serial input and a common serial data output. The following sections provide a brief description of these registers. For a complete description, refer to IEEE Standard Test Access port (IEEE Std. 1149.1-1990).

Bypass Register

The Bypass Register is used to allow test data to flow through the device from TDI to TDO. It contains a single-stage shift register for a minimum length in serial path. When an instruction selects the bypass register and the TAP controller is in the *Capture-DR* state, the shift register stage is set to a logic zero on the rising edge of TCLK. Bypass register operations should not have any effect on the device's operation in response to the BYPASS instruction.

Boundary-Scan Register

The Boundary Scan Register allows serial data to be loaded into or read out of the processor input/output ports. The Boundary Scan Register is a part of the IEEE 1149.1-1990 Standard JTAG Implementation. The shifting order of the JTAG's Boundary Scan Register is implemented using pin number direction for the RC64474/RC64475. The Boundary-Scan Register contains the following registers:

- *Enable I/O Buffer Register*
- *Data Register*

Note: A boundary scan device language (BSDL) file is available from Integrated Device Technology. This file can be obtained from the IDT website at www.idt.com.

Device Identification Register

The Device Identification Register is a read only 32-bit register used to specify the manufacturer, part number and version of the processor to be determined through the TAP in response to the IDCODE instruction. The IDT JEDEC identification number is 0xB3, which translates to 0x33 when the parity bit is dropped in the 11-bit Manufacturer ID field.

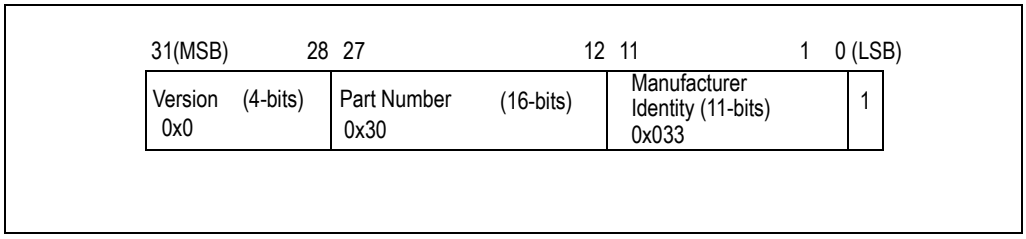


Figure 18.3 Device Identification Register Format

Notes

Cache Operations' Timing

Notes

Introduction

This appendix lists cycle operation counts and caveats for RC64474/RC64475 cache operations timing.

Caveats About Cache Operations

- *All cycle counts are in processor cycles.*
- *All cache ops have lower priority than cache misses, write backs and external requests. If the write back buffer contains unwritten data when a cache op is executed, the write back buffer will be retired before the cache op is begun.*

If an instruction cache miss occurs at the same time as a cache op is executed, the instruction cache miss will be handled first. Cache ops are mutually exclusive with respect to data cache misses. External requests will be completed before beginning a cache op.

- *For all data cache ops the cache op machine waits for the store buffer and response buffer to empty before beginning the cache op. This can add 3 cycles to any data cache op if there is data in the response buffer or store buffer. The response buffer contains data from the last data cache miss that has not yet been written to the data cache. The store buffer contains delayed store data waiting to be written to the data cache.*
- *Cache ops of the form xxxx_Writeback_xxxx may perform a write back which will fill the write back buffer. Write backs can affect subsequent cache ops, since they will stall until the write back buffer is written back to memory. Cache ops which fill the write back buffer are noted as (writeback) in the following tables.*
- *All cycle counts are best case assuming no interference from the mechanisms described above.*

Cache Operations Tables

Table A.1 and Table A.2 show data cache and instruction cache operations information. A detailed explanation of the Fill_I equation follows Table A.2.

Code[1]	Name	Number of Cycles
0	Index_Writeback_Invalidate_D	10 cycles if the cache line is clean. 12 cycles if the cache line is dirty (Writeback).
1	Index_Load_Tag_D	7 cycles.
2	Index_Store_Tag_D	8 cycles.
3	Create_Dirty_Exclusive_D	10 cycles for a cache hit. 13 cycles for a cache miss if the cache line is clean. 15 cycles for a cache miss if the cache line is dirty (Writeback).
4	Hit_Invalidate_D	7 cycles for a cache miss. 9 cycles for a cache hit.
5	Hit_Writeback_Invalidate_D	7 cycles for a cache miss. 12 cycles for a cache hit if the cache line is clean. 14 cycles for a cache hit if the cache line is dirty (Writeback).
7	Hit_Writeback_D	7 cycles for a cache miss. 10 cycles for a cache hit if the cache line is clean. 14 cycles for a cache hit if the cache line is dirty (Writeback).

Table A.1 Primary Data Cache Operations

1. Code number corresponds to the code column of the CACHE instruction in the *IDT Mips Microprocessor Family Software Reference Manual*.

Notes

Code[1]	Name	Number of Cycles
0	Index_Invalidate_I	7 cycles.
1	Index_Load_Tag_I	7 cycles.
2	Index_Store_Tag_I	8 cycles.
3	n/a	n/a
4	Hit_Invalidate_I	7 cycles for a cache miss. 9 cycles for a cache hit.
5	Fill_I	Cycle number must be calculated based on the system response to a memory access, because Fill_I causes an instruction cache refill from memory. This equation yields the number of processor cycles for a Fill_I cache op:[2] Number_of_cycles_for_a_Fill_I_CacheOp = 10 + {0 - (SYSDIV - 1)} + (2 x SYSDIV) + (ML x SYSDIV) + (D x SYSDIV)[3]
6	Hit_Writeback_I	7 cycles for a cache miss. 20 cycles for a cache hit (Writeback).

Table A.2 Primary Instruction Cache Operations

1. Code number corresponds to the code column of the CACHE instruction in Appendix A.

2. For definitions and discussion of the Fill_I equation variables refer to the subsection "Details of the Fill_I Equation," which follows this table.

3. The term {0 - (SYSDIV - 1) has a value between 0 and (SYSDIV - 1), depending on the alignment of the execution of the cache op with the system clock.

Fill_I Equation Definitions

These are the definitions for the Hit_Writeback_I equation in Table A.2:

SYSDIV: Number of processor cycles per system cycle; ranges from 2 - 8.

ML: Number of system cycles of memory latency, defined as the number of cycles the SysAD bus is driven by the external agent before the first double word of data appears.

D: Number of system cycles required to return the block of data, defined as the number of cycles beginning when the first double word of data appears on the SysAD bus and ending when the last double word of data appears on the SysAD bus, inclusive.

Standby Mode Operation

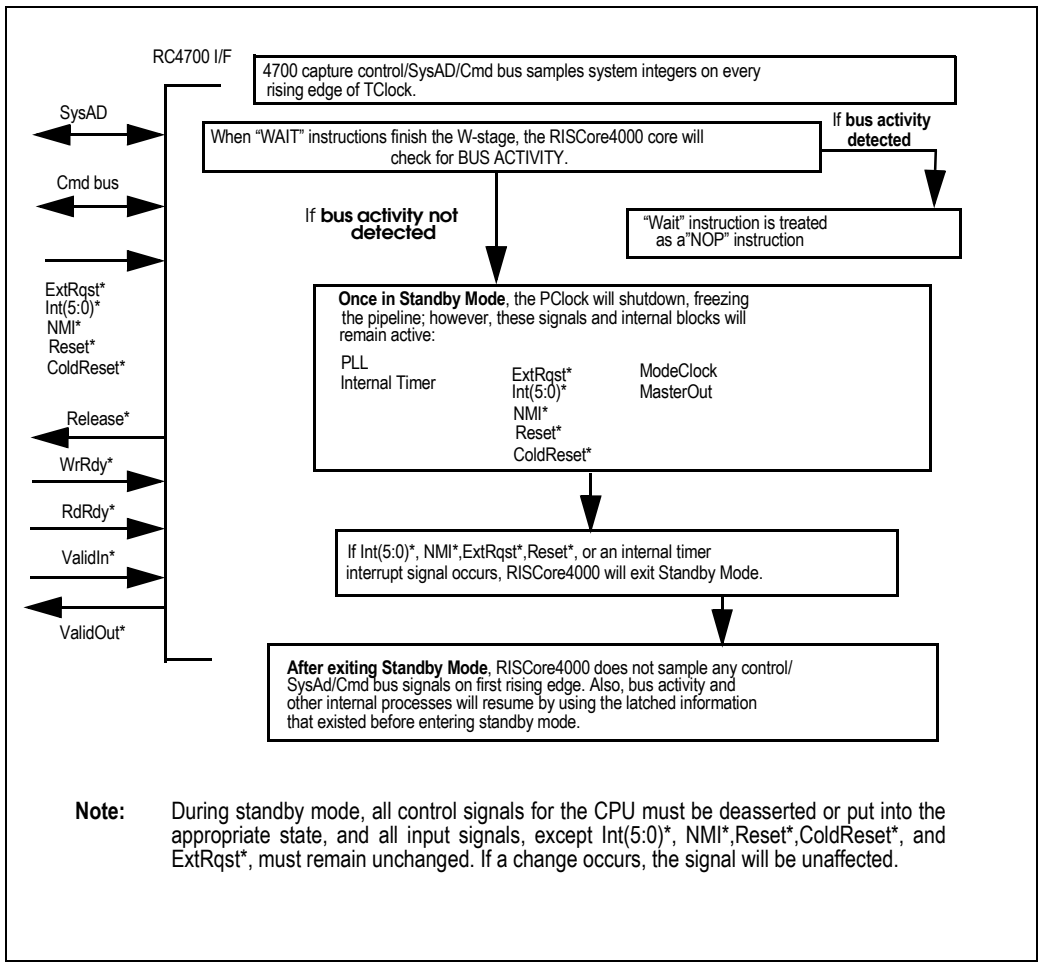
Notes

Introduction

The Standby Mode operation is a means of reducing the internal core's power consumption when the CPU is in a "standby" state. In this section, the Standby Mode operation is discussed.

Entering Standby Mode

To enter standby mode, first execute the WAIT instruction. When the WAIT instruction finishes the W pipe-stage, if the **SysAD** bus is currently idle, the internal clocks will shut down, thus freezing the pipeline. The PLL, internal timer, some of the input pin clocks (**Int[5:0]***, **NMI***, **ExtRqst***, **Reset*** and **ColdReset***), and the output clock (**ModeClock**) will continue to run. If the conditions are not correct when the WAIT instruction finishes the W pipe-stage (i.e., the **SysAD** bus is not idle), the WAIT is treated as a NOP. Once the CPU is in standby mode, any interrupt, including **ExtRqst*** or **Reset***, will cause the CPU to exit standby mode. Figure B.1 on page B-1 illustrates the Standby Mode Operation.



Note: During standby mode, all control signals for the CPU must be deasserted or put into the appropriate state, and all input signals, except Int(5:0)*, NMI*,Reset*,ColdReset*, and ExtRqst*, must remain unchanged. If a change occurs, the signal will be unaffected.

Figure B.1 Standby Mode Operation

Notes

Coprocessor 0 Hazards

Notes

Introduction

This appendix identifies the RC64474/475 Coprocessor 0 hazards. In Table C.1 the number of instructions required between instruction A (which places a value in a CP0 register) and instruction B (which uses the same register as a source) is computed using the following formula:

(destination stage of A) - (source stage of B) - 1

Operation	SOURCE Name	SOURCE Stage	DESTINATION Name	DESTINATION Stage
MTC0	gpr rt	2(A)	cpr rd	4(W)a
MFC0	cpr rd	2(A)	gpr rt	4(W)a
TLBR	Index, TLB	2(A)	PageMask, EntryHi,EntryLo0, EntryLo1	4(W)
TLBWI TLBWR	Index or Random, PageMask, EntryHi, EntryLo0, EntryLo1	2(A)	TLB	3(D)β
TLBP	PageMask, EntryHi	2(A)	Index	4(W)
ERET	EPC or ErrorEPC, Status.ERL	2(A)	Status.EXL, Status.ERL	4(W)γ
			LLbit	4(W)
CACHE Index Load Tag			TagLo, TagHi, ECC	3(D)
CACHE Index Store Tag	TagLo, TagHi, ECC	3(D)		
Instruction fetch	EntryHi.ASID, Status.KSU, Status.RE, Config.K0C, TLB	0(I)		
	Status.ERL, Status.EXL	0(I)γ		
Instruction fetch exception			EPC, Status, Cause	4(W)
			BadVAddr, Context, EntryHi	1(I)δ
Coprocessor usable test	Status.CU, Status.KSU, Status.EXL, Status.ERL	1(R)		
Interrupt	Cause.IP, Status.IM, Status.IE, Status.EXL, Status.ERL	2(A)		
Load/Store	EntryHi.ASID, Status.KSU, Status.RE, Status.ERL, Status.EXL Config.K0C, TLB	2(A)		
Load/Store exception			EPC, Status, Cause, BadVAddr, Context, EntryHi	4(W)

Notes:
a There must be at least one instruction between a MTC0 and a MFC0.
bTLBW_ instructions will cause a one cycle slip.
gInstructions fetches following an ERET will see a change in EXL or ERL in Stage 2 of the ERET in anticipation of the completion of the ERET. If the ERET does not complete, these instructions are killed before they commit changes in state other than noted by d. The pipestage corresponding to the stage field is given in parentheses.

Table C.1 Coprocessor 0 Hazards

Notes

Certain combinations of instructions are not permitted because the results of executing such combinations are unpredictable in the face of the events such as pipeline delays, cache misses, interrupts, and exceptions.

Most hazards result from instructions modifying and reading state in different pipeline stages. Such hazards are defined between pairs of instructions, not on a single instruction in isolation. Other hazards are associated with restartability of instructions in the presence of exceptions.

Index

IMPORTANT NOTICE AND DISCLAIMER

RENESAS ELECTRONICS CORPORATION AND ITS SUBSIDIARIES (“RENESAS”) PROVIDES TECHNICAL SPECIFICATIONS AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT OF THIRD-PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for developers who are designing with Renesas products. You are solely responsible for (1) selecting the appropriate products for your application, (2) designing, validating, and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. Renesas grants you permission to use these resources only to develop an application that uses Renesas products. Other reproduction or use of these resources is strictly prohibited. No license is granted to any other Renesas intellectual property or to any third-party intellectual property. Renesas disclaims responsibility for, and you will fully indemnify Renesas and its representatives against, any claims, damages, costs, losses, or liabilities arising from your use of these resources. Renesas' products are provided only subject to Renesas' Terms and Conditions of Sale or other applicable terms agreed to in writing. No use of any Renesas resources expands or otherwise alters any applicable warranties or warranty disclaimers for these products.

(Disclaimer Rev.1.01)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact Information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit www.renesas.com/contact-us/.

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.