79RC64574/RC64575
User Reference Manual

Version 1.0
April 2000

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 RC32134, a high performance system controller chip that supports IDT's RISCore32300 CPU family. The RC32134 offers a direct connection to IDT's RC32364 32-bit embedded microprocessor and provides the system logic for boot memory, main memory, I/O, and PCI. It also includes on-chip peripherals such as DMA channels, reset circuitry, interrupts, timers, and UARTs. Together, the RC32364 CPU and the RC32134 system controller form a complete CPU subsystem for embedded designs.

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 this device, which is available from the IDT website **(www.idt.com)** as well as through your local IDT sales representative.

Content Summary

Chapter 1, "RC64574/RC64575 Device 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 RISCore4000 and RISCore5000 families.

Chapter 2, "Central Processing Unit (CPU)," discusses how the central processing unit (CPU) executes integer and system instructions in the MIPS instruction-set architecture.

Chapter 3, "Instruction Pipeline," discusses the operation of the five-stage instruction pipeline with two parallel paths, one for integer (CPU) instructions and the other floating-point (FPU) instructions.

Chapter 4, "Memory Management Unit," describes the processor 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.

Chapter 5, "System Control Coprocessor," describe how the processor uses the memory management-related registers.

Chapter 6, "Integer (CPU) Exceptions," describes the integer exception processing done by the CPU, including an explanation of exception processing, followed by the format and use of each CPU exception register.

Chapter 7, "The Floating-Point Unit," describes the floating-point unit (FPU) operational functions consisting of an adder, multiplier, and divider. The FPU, with associated system software, conforms fully to the ANSI/IEEE Standard 754–1985, *IEEE Standard for Binary Floating-Point Arithmetic*.

Chapter 8, Floating-Point (FPU) Exceptions," describes how the FPU, when it cannot handle either the operands or the results of a floating-point operation in its normal way, responds to conditions by generating an exception to initiate a software trap or by setting a status flag.

Chapter 9, "Primary Caches," describes the on-chip primary cache, the individual operations of the primary cache, and the organization and operations of the on-chip secondary cache controller.

Chapter 10, "Processor Signal Descriptions Introduction," describes the signals used by and in conjunction with the RC64574 and R64575 processors. Signals include the System interface, the Clock/Control interface, the Interrupt interface, the Initialization interface, the handshake signals and JTAG Interface.

Chapter 11, "System Interface Overview," describes the system interface from the point of view of both the processor and the external agent.

Notes

Chapter 12, “The Clocking, Reset and Initialization Interface,” describes the clock signals (“clocks”) used in the RC64574 and RC64575 processors, including basic system clocks and system timing parameters.

Chapter 13, "The Read Interface," describes the Read protocol and associated operations.

Chapter 14, "The Write Interface," discusses the Write protocol and associated operations.

Chapter 15, "The External Request Interface," describes the external request process, including reads, writes and null requests.

Chapter 16, "Processor Interrupts," describes the six hardware interrupts, one internal “timer interrupt,” two software interrupts, and one unmasked/nonmaskable enabled interrupt.

Chapter 17, "Processor Error Checking," describes the error checking codes which allow the processor to detect and sometimes correct errors made when moving data from one place to another.

Chapter 18, "Standard JTAG Support Interface," describes the standard JTAG boundary scan that is used for on-chip debugging during Run-time mode.

Appendix A, "Cache Operations’ Timing," lists the cycle operation counts and caveats for RC64574/RC64575 cache operations timing.

Appendix B, "Standby Mode Operation," describes the Standby Mode operation.

Appendix C, "Coprocessor 0 Hazards," identifies the RC64574/575 Coprocessor 0 hazards

Revision History

October 1999: Initial publication.

March 2000: Corrected bullet statement in Boot-Mode Settings on page 12-8 to read: Bits 27 to 256 are reserved bits.

April 2000: Revised explanation of Config register bits in Config Register (16) section of Chapter 5 and added description for fields BE and IC in Table 5.7.

Table of Contents

Notes

Notes

Notes

Notes

Notes

Notes

Notes

13 The Read Interface

Notes

Notes

Notes

List of Tables

Notes

Notes

Notes

Notes

List of Figures

Notes

Notes

Notes

Notes

RC64574/RC64575 Device Overview

Notes

Introduction

Designed to service applications that require high bandwidth, real-time response and rapid data processing, Integrated Device Technology's (IDT) RC64574 and RC64575 processors are high performance devices that extend IDT's family of 64-bit RISController microprocessors. The RC64574 and RC64575 RISControllers are based on an enhanced RISCore5000, providing performance leadership at a dramatically reduced price point. Importantly, these devices continue IDT's pin compatibility strategy, enabling a simple upgrade migration path from RISCore4000 class microprocessors.

IDT's RISCore64500 is a 333MHz 64-bit execution core that utilizes a dual issue 5-stage scalar pipeline. The instruction pipeline has two parallel paths, one for integer (CPU) instructions and the other floating-point (FPU) instructions. Each stage in the CPU-instruction path takes one processor clock.

The RISCore64500 is a superset of the MIPS-IV 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-IV 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 RISCore64500 can operate on both 32- and 64-bit data, utilizing 32 general-purpose registers (GPR) that are used for integer operations and address calculations. Also, an on-chip floating-point coprocessor adds 32 floating-point registers and a floating-point control/status register.

Enhancements beyond the feature set of the RC5000 include:

1. Addition of DSP instructions previously implemented on the RC4640/RC4650 and RC32364 microprocessors
2. Inclusion of boundary scan to facilitate board-level testing
3. Enhanced timing protocols for SyncDRAM, which allows the address cycle and data cycles of writes to be separated and enables SDRAM protocol processing.
4. I-cache and D-cache locking capability (per line), providing improved real-time performance for mission critical applications
5. An ability to run SysAd 32-bit wide
6. Pin compatibility with other IDT CPUs.

The RC64574 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 RC64575 is packaged in a 208-pin QFP footprint package and uses the full 64-bit external bus. The RC64575 is ideal for applications requiring 64-bit performance and 64-bit external bandwidth. Table 1.1 summarizes the features of the 64-bit RISController Family.

Performance

The RC64574/RC64575 bring high performance to lower-cost systems, through dual instruction issue per clock cycle, dual 32KB on-chip two-way set associative caches; minimized branch and load delays through a simple streamlined pipeline; up to 6.3GB/s aggregate bandwidth support with a 1GB/s system interface; and facilities such as early restart for data cache misses. The RC64574/RC64575 processors are rated at 440 Dhrystone MIPS and 666 Million floating point operations per second (MFLOP/s), at 333 Mhz. The internal cache bandwidth for these devices is 5.3GB/s.

Notes

Device Compatibility

The RC64574/RC64575 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 RC64574/RC64575 offer a direct migration path for designs based on IDT's RC4640, RC4650, RC64474 and RC64475 microprocessors. Devices in the 64-bit RISController product families use the same bus protocols and syntax, enabling a range of designs and support chips.

64-bit RISController4000 and RISController5000

	RC4640	RC4650	RC64474	RC64475	RC64574	RC64575
CPU	64-bit RISCore4000 w/ DSP extensions	64-bit RISCore4000 w/ DSP extensions	64-bit RISCore4000	64-bit RISCore4000	64-bit RISCore 64500	64-bit RISCore 64500
Perfor-mance	350MIPS @267MHz	350MIPS @267MHz	330MIPS @250MHz	330MIPS @250MHz	440MIPS @333MHz	440MIPS @333MHz
FPA	89 mflops, single precision only	89 mflops, single precision only	125 mflops, single and double precision	125 mflops, single and double precision	666 mflops single and double precision	666 mflops single and double precision
Caches	8kB/8kB, 2-way, lockable per set	8kB/8kB, 2-way, lockable per set	16kB/16kB, 2-way, lockable per set	16kB/16kB, 2-way, lockable per set	32kB/32kB, 2-way, lockable per line	32kB/32kB, 2-way, lockable per line
External Bus	32-bit.	32- or 64-bit	32-bit, pin compatible w/ RC4640 and RC64574	32- or 64-bit, pin compatible w/ RC4650 and RC64575	32-bit Superset pin compatible w/RC4640 and RC64474	32- or 64-bit Superset pin compatible w/ RC4650 and RC64475
Voltage	3.3V	3.3V	3.3V	3.3V	2.5V, 3.3V tolerant I/O	2.5V, 3.3V tolerant I/O
Packages	128 PQFP	208 Power QUAD	128 Power-QUAD	208 PowerQUAD	128 Power-QUAD	208 PowerQUAD
MMU	Base-Bounds	Base-Bounds	96 page TLB	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	Cache locking, JTAG, syncDRAM mode, 32-bit external bus	Cache locking, JTAG, syncDRAM mode, 32-bit/ 64- bit bus option	DSP ISA extensions, JTAG, sync-DRAM mode, 32-bit external bus	DSP ISA extensions, JTAG, sync-DRAM mode, 32-bit/ 64- bit bus option

Table 1.1 Summary of Features for the 64-bit RISController Family

Notes

Summary of Features

The RC64574 and RC65475 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 RC64574 or RC64575 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 Embedded Microprocessor*
  - *MIPS-IV Instruction Set Architecture (ISA), with integer DSP and 3-operand integer multiply extensions*
  - *Dual-issue microarchitecture*
  - *High-performance floating-point capability*
- *Compatible with RC4640 and RC32364 DSP instruction set extensions*
  - *DSP Instructions, for consumer applications*
  - *2-cycle repeat rate, on atomic Multiply-add*
  - *Multiply-subtract (MSUB) support, for complex number processing*
  - *Count-leading-zero/one support, for string searches and normalization*
  - *3 Operand Multiply*
- *High-performance cache subsystem*
  - *32kB, two-set associative instruction cache*
  - *32kB, two-set associative data cache*
  - *Write-through and write-back data cache operations*
  - *High-performance cache-ops, bandwidth management*
  - *Cache-locking per line, for real-time applications*
- *Big- or Little-endian support*
- *RC5000 compatible memory management*
  - *On-chip 48-entry, 96-page TLB, for advanced operating system support*
  - *Compatible with major operating systems: Windows®CE, VxWorks, and others*
- *Bus compatible with IDT 64-bit microprocessor families*
  - *Pipeline runs at 2 to 8 times the bus frequency*
  - *Bus speeds to 125MHz*
  - *32-bit bus option, for lower cost systems*
  - *Enhanced timing protocol for SyncDRAM systems (compatible with RC64474/RC64475)*
- *RC64574:*
  - *32-bit SysAd bus, for low-cost systems*
  - *Pin compatible with RC4640 and RC64474*
  - *128-pin PowerQuad package*
- *RC64575:*
  - *64-bit SysAd bus interface*
  - *Pin compatible with RC4650 and RC64475*
  - *208-pin PowerQuad package*
- *JTAG Boundary Scan Interface*
- *2.5V operation with 3.3V compatible I/O*

Notes

Device Overview

Figure 1.1 contains an illustration of the key functional elements of the RC64574/RC64575 processors. An overview on these elements follows. Operational details are provided throughout the manual.

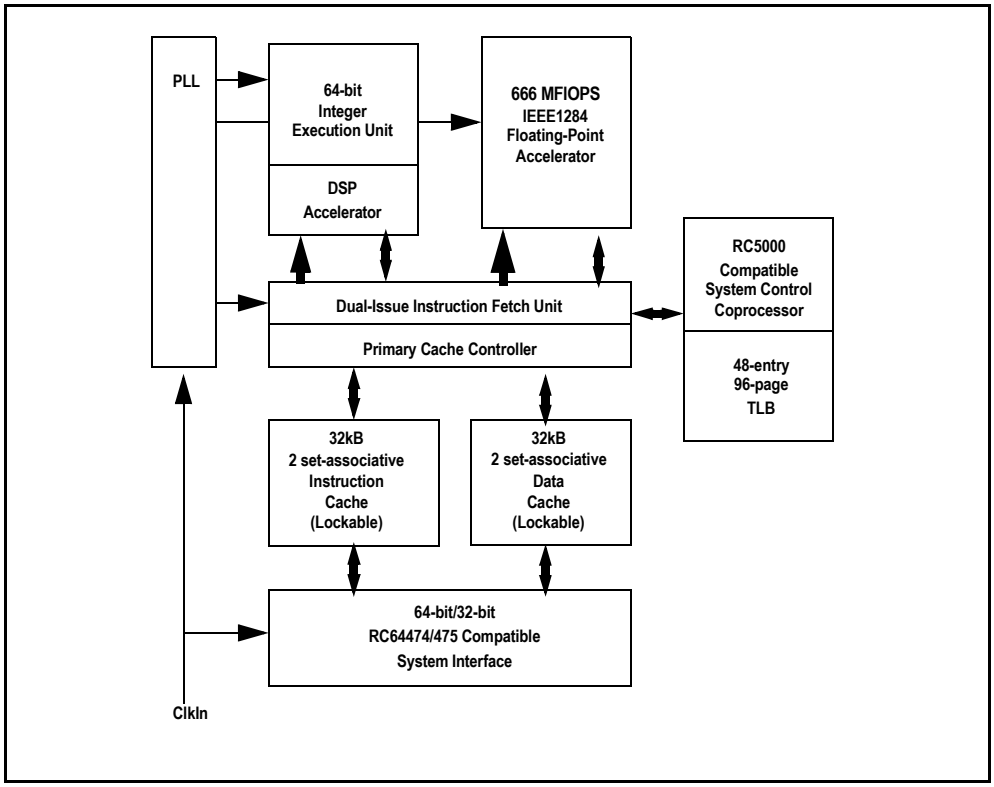


Figure 1.1 RC64574/RC64575 Functional Block Diagram

Instruction Pipeline

The RC64574/575 has a dual-issue, five-stage instruction pipeline that operates at a multiple of the frequency of the input system clock. The pipeline has two parallel paths, one for integer (CPU) instructions and the other floating-point (FPU) instructions. Each stage in the CPU-instruction path takes one processor clock.

Dual Issue

Dual-issue instruction pairing in a given clock can consist of:

- *1 floating-point ALU instruction*
- *1 instruction of any other type*

These two instruction classes are pre-decoded as they are brought on-chip. The pre-decoded information is stored in the instruction cache. If there are no pending resource conflicts, the RC64574/575 can issue one instruction per class per pipeline clock cycle. Long-latency operations, such as floating-point DIV or SQRT, or integer (CPU) divide, can slow the issue of instructions. The RC64574/575 does not perform out-of-order or speculative execution; instead, the pipeline slips until the required resource becomes available.

There are no alignment restrictions on dual-issue instruction pairs. However, since the RC64574/575 performs aligned fetches, at two instructions per cycle from the instruction cache, compilers should attempt to align branch targets to allow dual-issue on the first target cycle, in order to optimize performance.

Notes

Integer (CPU) Pipeline

The RC64574/575 implements a traditional five-stage pipeline, as shown in Figure 1.2 and Table 1.2:

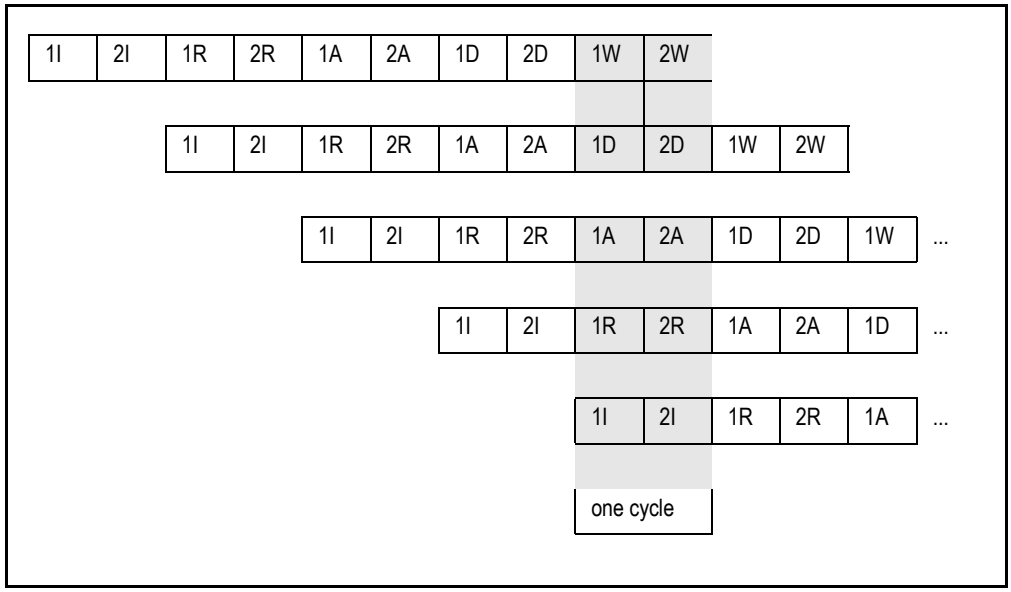


Figure 1.2 Integer (CPU) Pipeline

1I - Instruction Fetch, Phase One

2I - Instruction Fetch, Phase Two

1R - Register Read, Phase One

2R - Register Read, Phase Two

1A - Execution, Phase One

2A - Execution, Phase Two

1D - Data Load/Store, Phase One

2D - Data Load/Store, Phase Two

1W - Write Back, Phase One

2W - Write Back, Phase Two

Stage	Function
1I to 1R	Instruction-cache access
2I	Instruction virtual-to-physical address translation
2R	Register-file read, bypass calculation, instruction decode, branch-address calculation
1A	Issue or slip decision, data virtual-address calculation, branch decision
1A to 2A	Integer add, logical, shift
2A	Store align
2A to 2D	Data-cache access and load align
1D to 2D	Data virtual-to-physical address translation
1W	Resolve exceptions
2W	Register-file write

Table 1.2 Key to Integer Pipeline

Notes

Typical integer (CPU) execution latencies are shown in Table 1.3. The RC64574/575's short pipeline keeps the load and branch latencies low. The caches allow any combination of loads and stores to execute in back-to-back cycles without requiring pipeline slips or stalls if the operation hits in the cache. For entries marked with *, please refer to "Fast Multiply" note below.

Operation		Latency	Repeat
Load		2	1
Store		2	1
MULT, MAD	16-bit	3[1]	2[1]
	32-bit	4[1]	3[1]
MULTU, MADU	16-bit	3[1]	2[1]
	32-bit	4[1]	3[1]
MUL	16-bit	3[1]	2[1]
	32-bit	4[1]	3[1]
DMULT, DMULTU		7[2]	6[2]
DIV, DIVU		36	36
DDIV, DDIVU		68	68
Other Integer ALU		1	1
Branch		2	2
Jump		2	2

Table 1.3 Example Integer (CPU) Instruction Latencies

1. "Fast Multiply." To insure that the maximum frequency of operation is not limited by the speed of the multiplier unit, the RC64574/RC64575 features a "Fast Multiply" disable reset mode bit. If this bit is asserted, each multiply operation in the table above has its latency and repeat rate increased by one cycle. It is expected that this bit may be disabled for 250MHz and slower operation.

2. Similar to footnote 1, but **increased by two cycles**.

Floating-Point Unit (FPU) Pipeline

The on-chip floating-point unit (FPU) coprocessor includes a 64-bit floating-point register file. The FPU forms a seamless interface with the integer (CPU) pipeline, decoding and executing instructions in parallel with the CPU.

The FPU supports single- and double-precision arithmetic, as specified in the IEEE Standard 754. It also supports fully precise floating-point exceptions while allowing both overlapped and pipelined operations. Precise exceptions are extremely important in mission-critical environments and are highly desirable for debugging in any environment.

As described above, two instructions can be issued in a given clock if one is a floating-point ALU instruction and the another is any type other than floating-point ALU. Figure 1.3 shows a simplified diagram of this dual-issue mechanism.

Notes

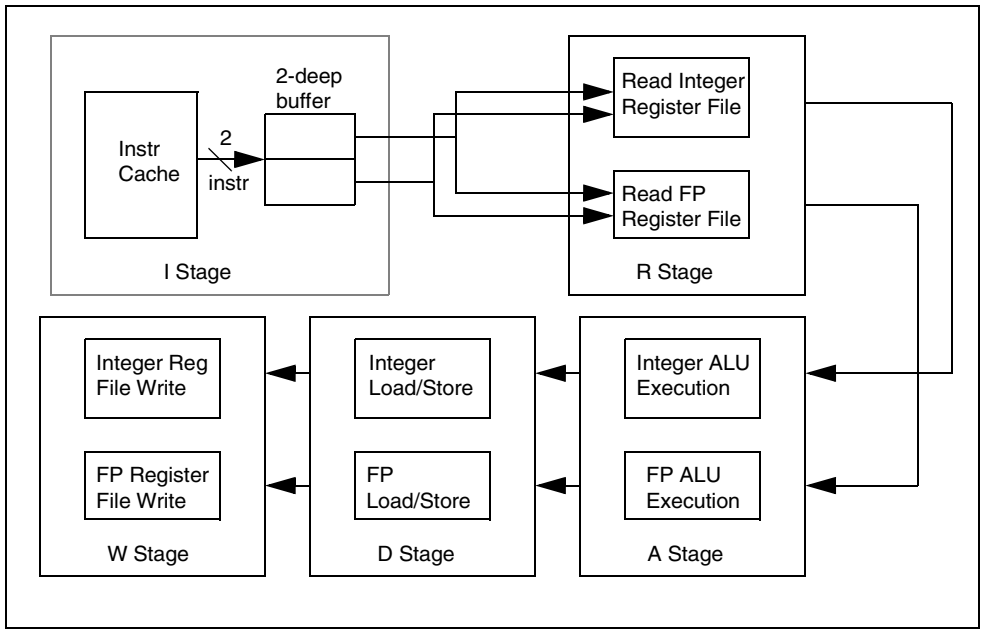


Figure 1.3 Dual-Issue Mechanism, Showing CPU and FPU Pipelines

Virtual-to-Physical Address Mapping

The RC64574/575 provides three modes of operation:

- *user mode*
- *supervisor mode*
- *kernel mode*

This mechanism is available to system software to provide a secure environment for user processes. Bits in a status register determine the mode of operation. When operating in the kernel mode, up to four distinct virtual-address spaces totalling 1024Gbytes are simultaneously available and are differentiated by the high-order bits of the virtual address. The RC64574/575 also supports a supervisor mode in which the virtual address space is 256Gbytes, divided into three regions based on the high-order bits of the virtual address. When the RC64574/575 uses 64-bit virtual addresses, the address space layouts are an upward-compatible extension of the 32-bit virtual address space layout.

Joint TLB

For fast virtual-to-physical address decoding, the RC64574/575 uses a fully associative joint 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 64Gbyte physical address space.

Two mechanisms 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, at 4Kbytes, 16Kbytes, 64Kbytes, 256Kbytes, 1Mbyte, 4Mbytes, or 16Mbytes. A system control 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 RC64574/575 supports 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.

Notes

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 RC64574/575. The write-through modes support more efficient frame-buffer accesses than the R4000 family. The coherent modes are supported for R4000 compatibility and generate different transaction types on the system interface; however, cache coherency is not supported.

Cache

The RC64574/575 incorporates on-chip instruction and data caches that can be accessed in a single processor cycle. The processor also includes an on-chip secondary cache controller for simple interfacing to large, high-speed second-level cache SRAM.

Both of the on-chip primary caches are 32KB in size, two-way set associative, virtually indexed, and physically tagged. Because the caches are virtually indexed, virtual-to-physical address translation occurs in parallel with the cache access. Each cache has its own 64-bit data path and can be accessed in parallel each pipeline cycle. The cache subsystem provides the integer unit (CPU) and floating-point unit (FPU) with an aggregate bandwidth of 5.3Gbytes per second at a pipeline clock frequency of 333MHz.

Instruction Cache

The instruction cache is 64-bits wide. Cache lines are eight instructions (32 bytes). Instruction fetches are 8 bytes per cycle, for a peak instruction-cache bandwidth of 2.7Gbytes/sec @ 333MHz. The instruction cache is protected with word parity. The tag holds a 24-bit physical address and valid bit, and is parity protected.

Data Cache

The data cache is 64-bits wide. Cache lines are 32 bytes. Data loads are 8 bytes per cycle, for a peak data-cache bandwidth of 2.7Gbytes/sec @ 333MHz (in addition to the 2.7Gbytes/sec instruction-cache 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 accesses. The normal write policy is writeback. Software can, however, select write-through on a per-page basis, such as for frame buffers.

The data cache has an associated store buffer. When the RC64574/575 executes a store instruction, this single-entry buffer gets written with the store data while the tag comparison is performed. If the tag matches, 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 RC64574/575 to execute a store every processor cycle and to perform back-to-back stores without penalty.

Write Buffer

Writes to external memory, whether cache-miss writebacks or stores to uncached or write-through addresses, use the on-chip write buffer. The write buffer holds up to four 64-bit address and data pairs or one cache line to be written back. The entire buffer is used for a data-cache writeback and allows the processor to proceed in parallel with memory update.

Clocks

The RC64574/575 uses the system interface clock as its input clock. The pipeline speed is derived from this clock using a PLL to multiply up the input reference. It is assumed that the system designer manages the system clock distribution to fit the needs of the system. Thus, the RC64574/575 does not output a system reference clock, but rather operates in synchronization with the input clock.

The RC64574/575 outputs one low-frequency reference clock: the Mode Clock. This clock operates at 1/256 the rate of the input clock, and it is used to clock-in the serial initialization stream during reset.

Notes

System Interface

The RC64574/575 supports a 64-bit multiplexed system interface that is compatible with the R4xxx system interface. The interface consists of a 64-bit address/data bus with 8 check bits and a 9-bit command bus. In addition, there are 8 handshake signals and 6 interrupt inputs. The interface has a simple timing specification and is capable of transferring data between the processor and memory at a peak rate of 1000Mbytes/sec at 125MHz. Figure 1.4 shows a typical system using the RC64574/575.

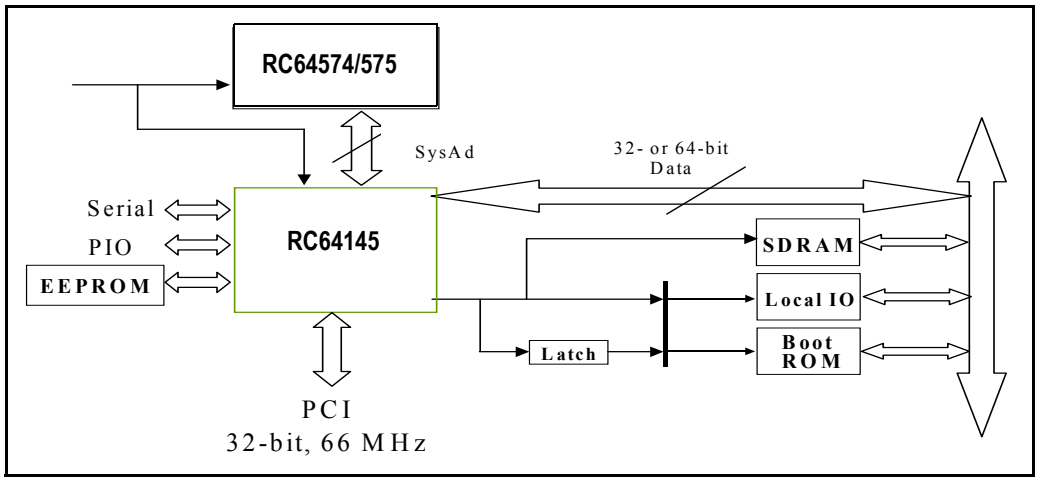


Figure 1.4 Typical RC64574/575 System Block Diagram

Notes

Central Processing Unit (CPU)

Notes

Introduction

In the MIPS instruction-set architecture, the central processing unit (CPU) executes integer and system instructions, and the floating-point unit (FPU) coprocessor executes floating-point instructions. This chapter describes only the CPU. Chapter 3 describes the FPU.

CPU Registers

The RC64574/575 integer unit has thirty-two general purpose registers. These registers 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 in the pipeline. Figure 2.1 shows the RC64574/575 CPU registers.

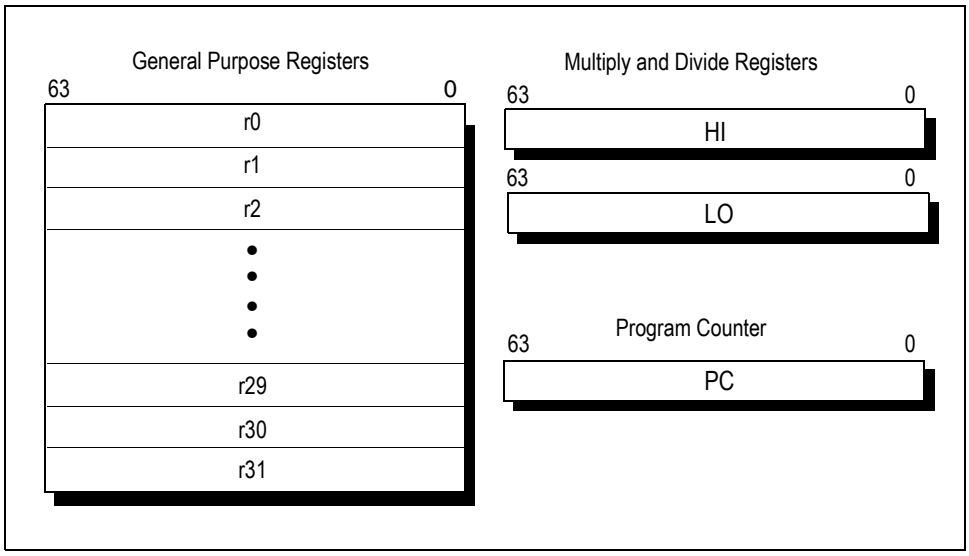


Figure 2.1 RC64574/575 CPU Registers

Two of the CPU general purpose registers have assigned functions:

- *r0 is hardwired to a value of zero, and can be used as the target register for any instruction whose result is to be discarded. r0 can also be used as a source when a zero value is needed.*
- *r31 is used as an implicit return destination address register by the JAL series of instructions.*

The CPU has three special purpose registers:

- *PC — Program Counter register*
- *HI — Multiply and Divide register, higher result*
- *LO — Multiply and Divide register, lower result*

The two Multiply and Divide registers (HI, LO) store:

- *the product of integer multiply operations, or*
- *the quotient (in LO) and remainder (in HI) of integer divide operations.*

The RC64574/575 has no Program Status Word (PSW) register as such; this is covered by the *Status* and *Cause* registers incorporated within the System Control Coprocessor (CP0), as described in the next section, below.

Notes

Coprocessors (CP0-CP2) and Their Registers

The MIPS IV instruction set architecture defines three coprocessors, designated CP0, CP1, and CP2. The RC64574/575 implements the first two:

- *Coprocessor 0 (CP0) supports the virtual memory system and exception handling. CP0 is also referred to as the System Control Coprocessor, and is described below.*
- *Coprocessor 1 (CP1) implements the MIPS floating-point instruction set and is used by the FPU. The registers associated with CP1 are described in Chapter 3.*
- *Coprocessor 2 (CP2) is reserved for future use.*

The registers associated with CP0 are shown in Figure 2.2 and described in Table 2.1. CP0 translates virtual addresses into physical addresses and manages exceptions and transitions between kernel, supervisor, and user states. CP0 also controls the cache subsystem, controls power management, and provides diagnostic control and error recovery facilities. Access to reserved or undefined CP0 register results are undefined. An exception may or may not result.

Power management is implemented in CP0 with the standby mode, which reduces power consumption by the CPU core. The standby mode is entered by executing the WAIT instruction with the SysAD bus idle, and it is exited by any interrupt.

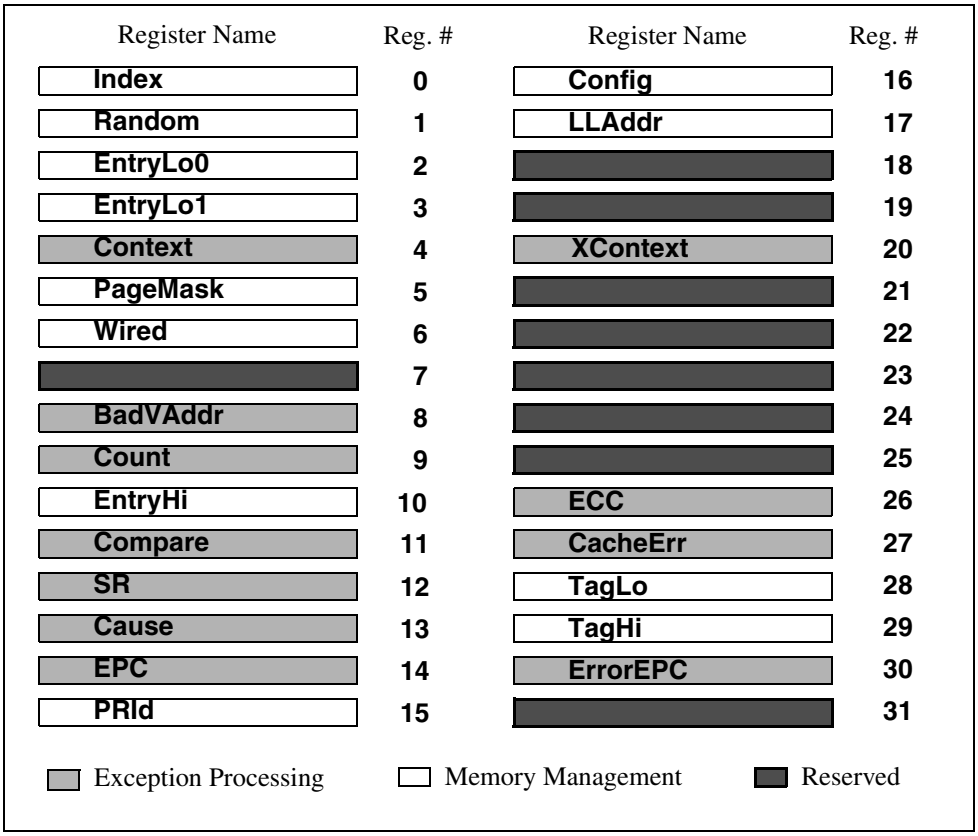


Figure 2.2 CP0 Registers

Notes

Number	Register	Description
0	Index	Programmable pointer into TLB array
1	Random	Pseudo-random pointer into TLB array *(read only)*
2	EntryLo0	Low half of TLB entry for even virtual page (VPN)
3	EntryLo1	Low half of TLB entry for odd virtual page (VPN)
4	Context	Pointer to kernel virtual page table entry (PTE) for 32-bit address spaces
5	PageMask	TLB page mask
6	Wired	Number of wired TLB entries
7	—	Reserved
8	BadVAddr	Bad virtual address
9	Count	Timer count
10	EntryHi	High half of TLB entry
11	Compare	Timer compare
12	SR	Status register
13	Cause	Cause of last exception
14	EPC	Exception program counter
15	PRId	Processor revision identifier
16	Config	Configuration register
17	LLAddr	Load linked address
18 - 19	—	Reserved
20	XContext	Pointer to kernel virtual PTE table for 64-bit address spaces
21–25	—	Reserved
26	ECC	Secondary-cache error checking and correcting (ECC) and primary parity
27	CacheErr	Cache error and status register
28	TagLo	Cache tag register
29	TagHi	Cache tag register
30	ErrorEPC	Error exception program counter
31	—	Reserved

Table 2.1 System Control Coprocessor (CPO) Register Definitions

CPU Data Formats and Addressing

The RC64574/575 processor uses four data formats: a 64-bit doubleword, a 32-bit word, a 16-bit halfword, and an 8-bit byte. Byte ordering within the halfword, word, and doubleword data formats can be configured in either big-endian or little-endian order. Endianness refers to the location of byte 0 within the multi-byte data structure. Figure 2.3 and Figure 2.4 show the ordering of bytes within words and the ordering of words within multiple-word structures for the big-endian and little-endian conventions.

When the RC64574/575 processor is 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 2.3 illustrates this configuration.

Notes

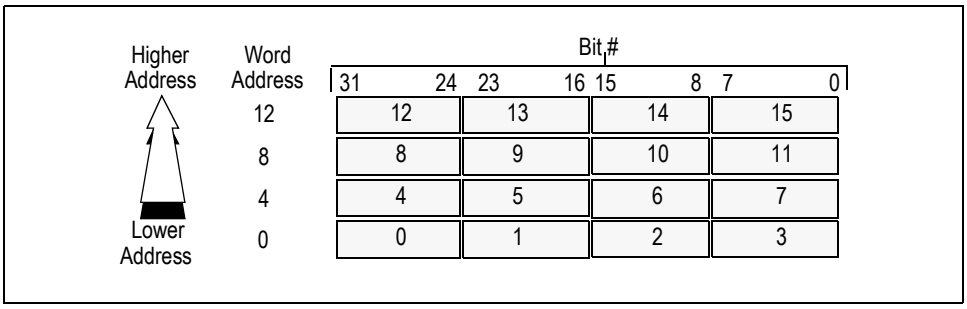


Figure 2.3 Big-Endian Byte Ordering

When configured as a little-endian system, byte 0 is always the least-significant (right-most) byte, which is compatible with x86 and DEC VAX conventions. Figure 2.4 illustrates this configuration.

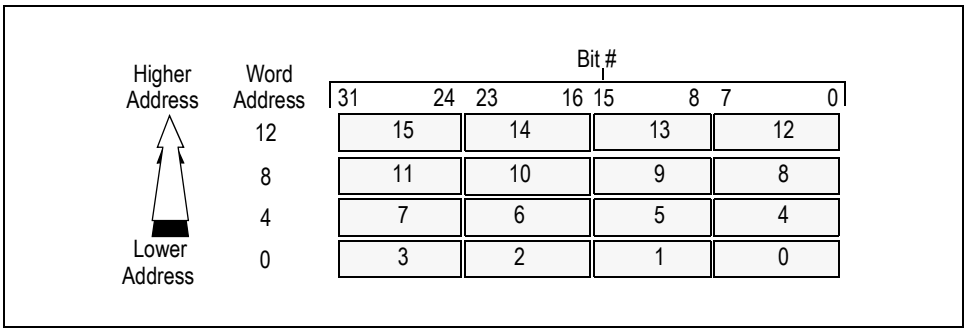


Figure 2.4 Little-Endian Byte Ordering

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

Figure 2.5 and Figure 2.6 show little-endian and big-endian byte ordering in doublewords.

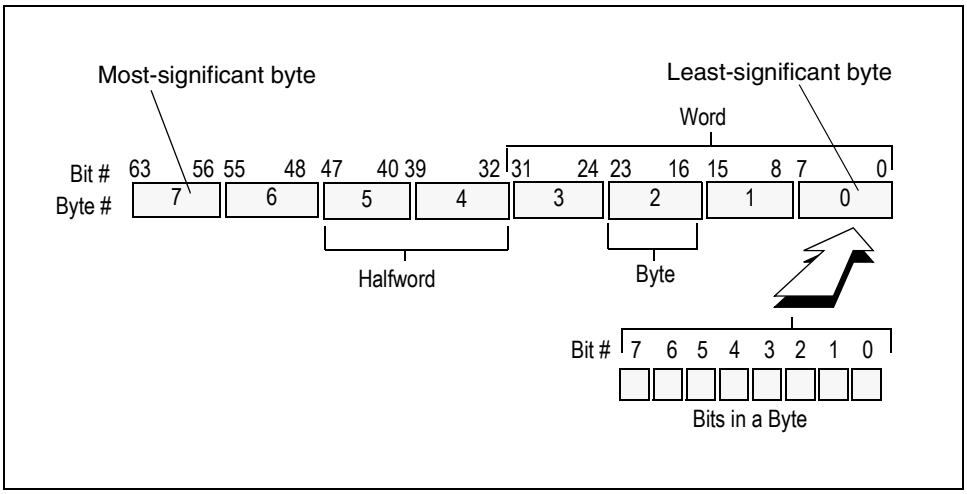


Figure 2.5 Little-Endian Data in a Doubleword

Notes

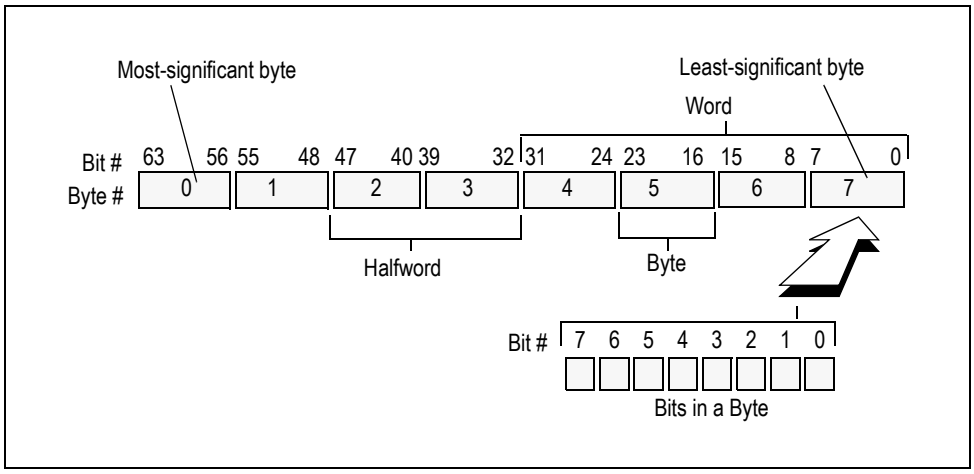


Figure 2.6 Big-Endian Data in a Doubleword

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 that are not aligned on 4-byte (word) or 8-word (doubleword) boundaries:

LWL	LWR	SWL	SWR
LDL	LDR	SDL	SDR

These instructions are used in pairs to provide addressing of misaligned words. Addressing misaligned data incurs one additional instruction cycle over that required for addressing aligned data. This extra cycle is the result of an extra instruction for the “pair” (e.g., LWL and LWR form a pair). Also note that the CPU moves the unaligned data at the same rate as a hardware mechanism.

Figure 2.7 and Figure 2.8 show the access of a misaligned word that has byte address 3.

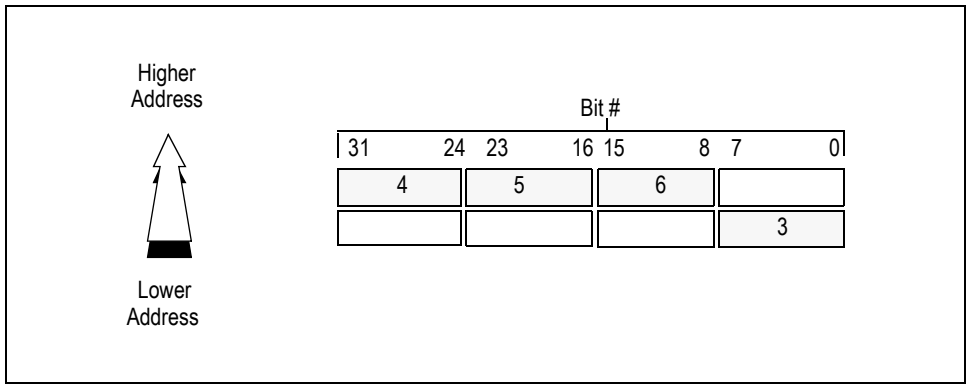


Figure 2.7 Big-Endian Misaligned Word Addressing

Notes

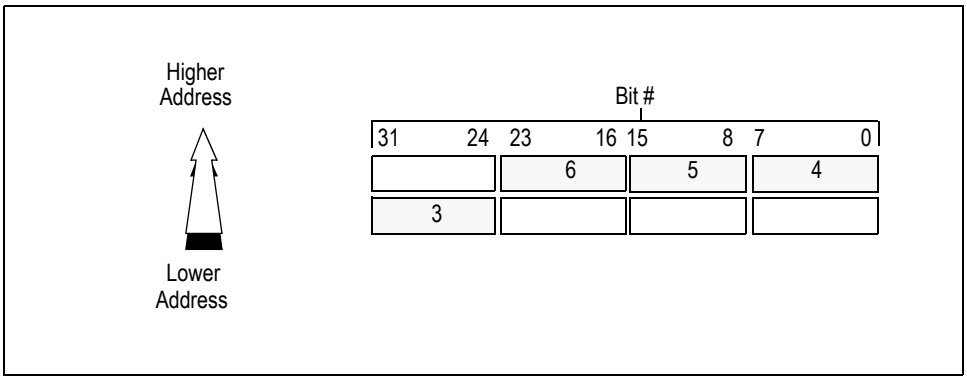


Figure 2.8 Little-Endian Misaligned Word Addressing

CPU Instruction Set Summary

The RC64574/575 executes the MIPS IV instruction set, which is a superset of the MIPS III instruction set and is backward-compatible with MIPS III. 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), and register (R-type). The use of a small number of instruction formats simplifies instruction decoding, allowing the compiler to synthesize more complicated (and less frequently used) operations and addressing modes from these three formats as needed.

Table 2.2 gives an overview of RC64574/575 CPU-instruction latencies. A summary of the MIPS IV instruction set additions is given in the remainder of this section, along with a brief explanation of each instruction. For more information on the MIPS IV instruction set, refer to the IDT Microprocessor Family Software Manual.

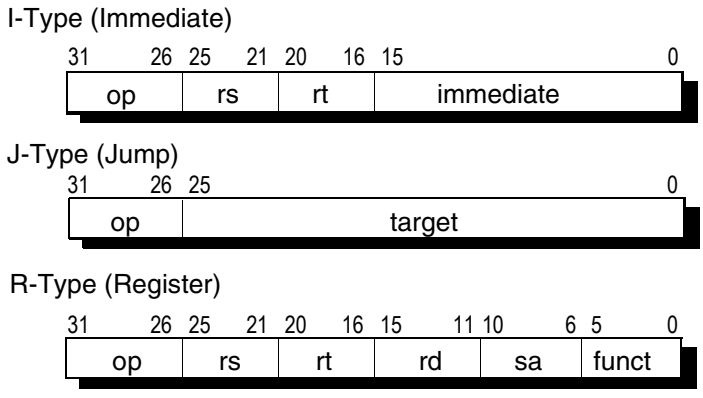
Instruction Group	Latency	Repeat
Arithmetic and Logical	1	1
Shift	1	1
Load	2	1
Store	N/A	1
Multiply (32-bit)	5	4
Multiply (64-bit)	9	8
Divide (32-bit)	36	36
Divide (64-bit)	68	68

Table 2.2 Some Integer-Instruction Latencies

Instruction Formats

The formats of the three instruction types are shown in Figure 2.9.

Notes



op	6-bit operation code
rs	5-bit source register specifier
rt	5-bit target (source/destination) register or branch condition
immediate	16-bit immediate value, branch displacement or address displacement
target	26-bit jump target address
rd	5-bit destination register specifier
sa	5-bit shift amount
funct	6-bit function field

Figure 2.9 Instruction Formats

Instruction Types

The CPU instruction set includes the following types of instructions:

- ***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.*
- ***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.*

Notes

Load and Store Instructions

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*.

Table 2.3 lists the load and store instructions.

OpCode	Description	MIPS ISA Level
LB	Load Byte	I
LBU	Load Byte Unsigned	I
LD	Load Doubleword	III
LDL	Load Doubleword Left	III
LDR	Load Doubleword Right	III
LH	Load Halfword	I
LHU	Load Halfword Unsigned	I
LL	Load Linked	II
LLD	Load Linked Doubleword	III
LW	Load Word	I
LWL	Load Word Left	I
LWR	Load Word Right	I
LWU	Load Word Unsigned	III
PREF[1]	Prefetch, Register + Offset	IV
PREFX[1]	Prefetch Indexed, Register + Register	IV
SB	Store Byte	I
SC	Store Conditional	II
SCD	Store Conditional Doubleword	III
SD	Store Doubleword	III
SDL	Store Doubleword Left	III
SDR	Store Doubleword Right	III
SH	Store Halfword	I
SW	Store Word	I
SWL	Store Word Left	I
SWR	Store Word Right	I
SYNC	Sync	II

Table 2.3 Load and Store Instructions

1. Prefetch is not implemented in the RC64574/575; these instructions are executed as no-ops.

Scheduling a Load Delay Slot

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

Notes

In the RC64574/575, the instruction immediately following a load instruction can reference the contents of the loaded register, but hardware interlocks insert additional real cycles. Consequently, scheduling load delay slots can be desirable, both for performance and RC30xx processor compatibility. However, the scheduling of load delay slots is not required for this part.

Defining Access Types

Access type indicates the size of a data item to be loaded or stored. The access type is determined by the load/store instruction opcode. Regardless of access type or byte ordering (endianness), the address 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 process endianness and the three low-order bits of the address, define the bytes accessed within the addressed doubleword (shown in Table 2.4). Only the combinations shown in Table 2.4 are permissible; other combinations 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							

Table 2.4 Byte Access Within a Doubleword

Notes

Computational Instructions

Computational instructions can be either in register (R-type) format, in which both operands are registers, or in 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*

These operations fit in the following four categories of computational instructions:

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

Table 2.5 through Table 2.8 list the computational instructions.

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

Table 2.5 Arithmetic Instructions - ALU Immediate

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

Table 2.6 Arithmetic - 3-Operand, R-Type (Part 1 of 2)

Notes

OpCode	Description	MIPS ISA Level
SLTU	Set on Less Than, Unsigned	I
SUB	Subtract	I
SUBU	Subtract, Unsigned	I
XOR	Exclusive OR	I

Table 2.6 Arithmetic - 3-Operand, R-Type (Part 2 of 2)

OpCode	Description	MIPS ISA Level
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
SLL	Shift Left Logical	I
SLLV	Shift Left Logical Variable	I
SRA	Shift Right Arithmetic	I
SRAV	Shift Right Arithmetic Variable	I
SRL	Shift Right Logical	I
SRLV	Shift Right Logical Variable	I

Table 2.7 Shift Instructions

OpCode	Description	MIPS ISA Level
DIV	Divide	I
DIVU	Divide Unsigned	I
DMULT	Doubleword Multiply	III
DMULTU	Doubleword Multiply Unsigned	III
DDIV	Doubleword Divide	III
DDIVU	Doubleword Divide Unsigned	III
MFHI	Move From HI	I
MTHI	Move To HI	I
MFLO	Move From LO	I
MOVF	Move Conditional on Condition Code False	IV
MOVN	Move on Register Not Equal to Zero	IV
MOVT	Move Conditional on Condition Code True	IV

Table 2.8 Multiply and Divide Instructions (Part 1 of 2)

Notes

OpCode	Description	MIPS ISA Level
MOVZ	Move on Register Equal to Zero	IV
MTLO	Move To LO	I
MULT	Multiply	I
MULTU	Multiply Unsigned	I

Table 2.8 Multiply and Divide Instructions (Part 2 of 2)

64-bit Operations

When operating in 64-bit mode, 32-bit operands must be sign-extended. 32-bit operand opcodes include all non-doubleword operations, such as ADD, ADDU, SUB, SUBU, ADDI, SLL, SRA, SLLV, etc. The result of operations that use incorrect sign-extended 32-bit values is unpredictable.

Cycle Timing for Multiply and Divide Instructions

MFHI and MFLO instructions are interlocked so that any attempt to read them before prior instructions complete, delays the execution of these instructions until the prior instructions finish. Table 2.9 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.

.

Opcodes		Condition	Latency	Repeat	Stall
MULT, MAD	16-bit	$-2^{15} \le rt \le 2^{15}-1$	3[1]	2[1]	0
	32-bit	$rt < -2^{15}$ or $rt > 2^{15}-1$	4[1]	3[1]	0
MULTU, MADU	16-bit	$0 \le rt \le 2^{15}-1$	3[1]	2[1]	0
	32-bit	$rt > 2^{15}-1$	4[1]	3[1]	0
MUL	16-bit	$-2^{15} \le rt \le 2^{15}-1$	3[1]	2[1]	1[1]
	32-bit	$rt < -2^{15}$ or $rt > 2^{15}-1$	4[1]	3[1]	2[1]
DMULT, DMULTU		any	7[2]	6[2]	0
DIV, DIVU		any	36	36	0
DDIV, DDIVU		any	68	68	0

Table 2.9 RC64574/RC64575 Integer Multiplier Performance

1. "Fast Multiply." To insure that the maximum frequency of operation is not limited by the speed of the multiplier unit, the RC64574/RC64575 features a "Fast Multiply" disable reset mode bit. If this bit is asserted, each multiply operation in the table above has its latency and repeat rate increased by one cycle. It is expected that this bit may be disabled for 250MHz and slower operation.

2. Similar to footnote 1, but **increased by two cycles**.

Jump and Branch Instructions

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 (the instruction in the *delay slot*) always executes while the target instruction is being fetched from storage.

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. Both are R-type instructions that take the 32-bit or 64-bit byte address contained in one of the general purpose registers.

Notes

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). All branches occur with a delay of one instruction. If a conditional branch is not taken, the instruction in the delay slot is nullified.

Table 2.10 lists the jump and branch instructions.

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

Table 2.10 Jump and Branch Instructions

Special Instructions

Special instructions allow the software to initiate traps. They are always R-type. Table 2.11 lists the special instructions.

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

Table 2.11 Special Instructions

Notes

Coprocessor Instructions

Coprocessor instructions perform operations in their respective coprocessors. Coprocessor loads and stores are I-type, and coprocessor computational instructions have coprocessor-dependent formats. CP0 instructions perform operations specifically on the System Control Coprocessor registers to manipulate the memory management and exception handling facilities of the processor.

Table 2.12 and Table 2.13 list the coprocessor instructions. Table 2.14 lists the instructions used for exception processing.

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

Table 2.12 Coprocessor Instructions

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

Table 2.13 CPO Instructions

Notes

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

Table 2.14 Exception Instructions

IDT ISA Enhancements

Instruction extensions are implemented in the RC64574 and RC64575 to accelerate execution of DSP algorithms. Those DSP instructions are the same as the one introduced in RC32364 (MUL, MAD, MADU, MSUB, MSUBU, CLZ, CLO. For more information about these instructions, refer to RC32364 Reference Manual Appendix A-4).

The instruction execution speed is tabulated in Table 2.15.

Operation	Latency	Repeat
MULT/MULTU	See section below	See section below
DMULT/DMULTU	See section below	See section below
DIV/DIVU	See section below	See section below
DDIV/DDIVU	See section below	See section below
MAD/MADU	See section below	See section below
MSUB/MSUBU	See section below	See section below
MUL	See section below	See section below
CLZ/CLO	1	1

Table 2.15 RC64574/RC64575 Instruction Enhancements Execution Rate

The short pipeline of the RC64574/RC64575 keeps the latency of loads and branches very low. Special logic in the caches allow any combination of loads and stores to execute in back to back cycles without requiring pipeline slips or stalls (presuming of course that the operation does not miss in the cache).

In addition, branch latencies are kept very low. The RC64574/RC64575 does not implement dynamic branch prediction, based on the following:

- *The MIPS ISA specifies that the instruction immediately following a branch or jump be executed, regardless of the outcome of the branch condition. This matches the branch latency of the RC64574/RC64575 pipeline, so there is no penalty for "mis-predicted" branches.*
- *The MIPS-ISA implements compiler driven branch prediction through the Branch-likely forms of branches, if the compiler cannot find a useful instruction to place in the branch-delay slot of traditional branches.*

Notes

- *The MIPS-IV ISA implements "conditional move" operations to minimize the number of branches in programs.*

Special Instruction Notes

There are a few special notes on instruction execution rules for the RC64574/RC64575:

- *Indexed CP1 loads and stores (LDXC1, LWXC1, PREFX, SDXC1, SWXC1) are two cycle instructions. This keeps the index register out of the address calculation critical path.*
- *MOVZ.fmt and MOVN.fmt are considered floating point load/store class instructions. Thus, they may dual-issue with floating point ALU operations.*
- *Integer multiply performance is sensitive to operand value and the setting of the "Fast Multiply" disable bit, taken from the boot time mode data stream.*
- *"Fast Multiply" must be disabled for operation above 250MHz.*
- *The PREF instructions are executed as NOPs in the RC64574/RC64575, as permitted in the MIPS-IV ISA.*
- *The RC64574/RC64575 implements the "Wait" instruction found in a variety of IDT microprocessors, including the RC5000.*

RC64574/RC64575 Computational Units

The RC64574/RC64575 contains the following computational units:

- *Integer ALU. The RC64574/RC64575 implements a full, single-cycle 64-bit ALU for all integer ALU functions other than multiply and divide. Bypassing is used to support back-to-back ALU operations at the full pipeline rate, without requiring stalls for data dependencies.*
  - *Note that in the RC64574/RC64575, the integer ALU has been extended to support a 32-bit count-leading-zeros and count-leading-ones operation, compatible with the RC32364.*
- *Integer Divide Unit. This unit is separated from the primary ALU, to allow these longer latency operations to run in parallel with other operations. The pipeline stalls only if an attempt to access the HI or LO registers is made before the operation completes.*
- *Integer Multiplier Unit. This unit is separated from the primary ALU, to allow these longer latency operations to run in parallel with other operations. The pipeline stalls only if an attempt to access the HI or LO registers is made before the operation completes, or if the functional unit is unavailable. This unit performs the following operations: MUL, DMULT/U, MULT/U, MADD/U, MSUB/U.*
- *Floating point ALU. This unit is responsible for all CP1/CP1X ALU operations other than DIV/SQRT. The unit is pipelined to allow a single-cycle repeat rate for single-precision operations*
- *Floating point DIV/SQRT unit. This unit is separated from the other floating-point ALU, so that these long latency operations do not prevent the issue of other floating point operations.*

In addition, the RC64574/RC64575 implements separate logical units to implement loads, stores, and branches.

Notes

RC64574/RC575 Multiply Performance

The performance of integer multiply and divide is summarized in Table 2.16. For entries marked with *, please refer to “Fast Multiply” note below.

Opcodes		Condition	Latency	Repeat	Stall
MULT, MAD	16-bit	$-2^{15} \le rt \le 2^{15}-1$	3[1]	2[1]	0
	32-bit	$rt < -2^{15}$ or $rt > 2^{15}-1$	4[1]	3[1]	0
MULTU, MADU	16-bit	$0 \le rt \le 2^{15}-1$	3[1]	2[1]	0
	32-bit	$rt > 2^{15}-1$	4[1]	3[1]	0
MUL	16-bit	$-2^{15} \le rt \le 2^{15}-1$	3[1]	2[1]	1[1]
	32-bit	$rt < -2^{15}$ or $rt > 2^{15}-1$	4[1]	3[1]	2[1]
DMULT, DMULTU		any	7[2]	6[2]	0
DIV, DIVU		any	36	36	0
DDIV, DDIVU		any	68	68	0

Table 2.16 RC64574/RC64575 Integer Multiplier Performance

1. “Fast Multiply.” To insure that the maximum frequency of operation is not limited by the speed of the multiplier unit, the RC64574/RC64575 features a “Fast Multiply” disable reset mode bit. If this bit is asserted, each multiply operation in the table above has its latency and repeat rate increased by one cycle. It is expected that this bit may be disabled for 250MHz and slower operation.

2. Similar to footnote 1, but **increased by two cycles**.

New Instructions: Opcodes and Operation

The RC64574/RC64575 adds new instructions to the MIPS-IV ISA, intended to enhance the performance of certain types of DSP algorithms.

Specifically, enhancements in the multiplier have been added to allow fast fused multiply-adds and multiply-subtracts, as shown in Figures 2.10 through 2.13. In addition, RC64574/RC64575 adds the three operand multiply operations originally found in the RC32364 and adds instructions to help normalize values (count-leading-1’s or 0’s, also found in the RC32364). See Figures 2.14 through 2.16.

Multiply Add

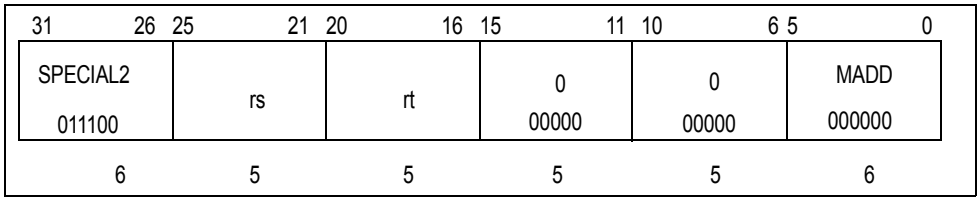


Figure 2.10 Multiply-Add Instruction Format

Format: MAD rs, rt

Description: The content of general registers rs and rt are multiplied— treating both operands as 32-bit two’s complement values—and the result is added to HI/LO. Overflow exceptions do not occur under any circumstances.

Once the operation is complete, the low-order word of the double result is loaded in LO, and the high-order word of the double result is loaded in HI.

Notes

Operation:

```
T:      temp <-- (HI || LO) + GPR[rs] * GPR[rt]
        LO <-- temp31..0
        HI <-- temp63..32
```

Exception: None

Multiply Add Unsigned

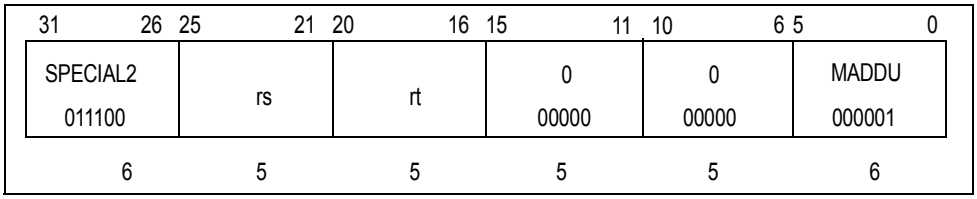


Figure 2.11 Multiply-Add Unsigned Instruction Format

Format: MADU rs,rt

Description: The content of general registers rs and rt are multiplied, treating both operands as 32-bit unsigned values, and the result is added to HI/LO. No overflow exception occur under any circumstances.

When the operation completes, the low-order word of the double result is loaded in LO, and the high-order word of the double result is loaded in HI.

Operation:

```
T:      temp <-- (HI || LO) + (0||GPR[rs]) * (0||GPR[rt])
        LO <-- temp31..0
        HI <-- temp63..32
```

Exception: None

Multiply Subtract

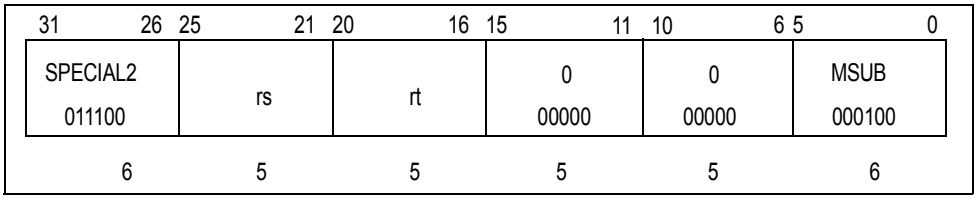


Figure 2.12 Multiply-Subtract Instruction Format

Format: MSUB rs,rt

Description: The content of general registers rs and rt are multiplied, treating both operands as 32-bit two's complement values, and the result is subtracted from HI/LO. No overflow exception occurs under any circumstances.

When the operation is complete, the low-order word of the double result is loaded in LO, and the high-order word of the double result is loaded into HI.

Notes

Operation:

```
T:      temp <-- (HI || LO) - GPR[rs] * GPR[rt]
        LO <-- temp31..0
        HI <-- temp63..32
```

Exception: None

Multiply Subtract Unsigned

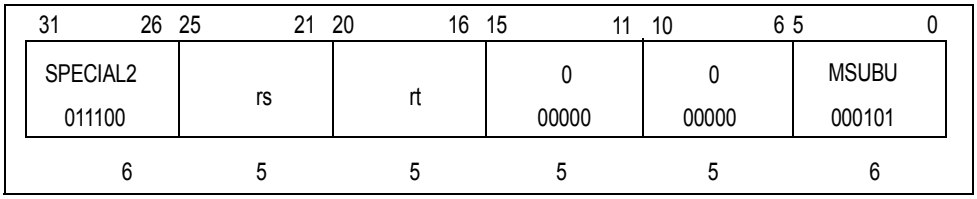


Figure 2.13 Multiply-Subtract Unsigned Instruction Format

Format: MSUBU rs,rt

Description: The content of general registers rs and rt are multiplied, treating both operands as 32-bit unsigned values, and the result is subtracted from HI/LO. No overflow exception occurs under any circumstances.

When the operation completes, the low-order word of the double result is loaded in LO, and the high-order word of the double result is loaded in HI.

Operation:

```
T:      temp <-- (HI || LO) - (0||GPR[rs]) * (0||GPR[rt])
        LO <-- temp31..0
        HI <-- temp63..32
```

Exception: None

Count Leading Zeros

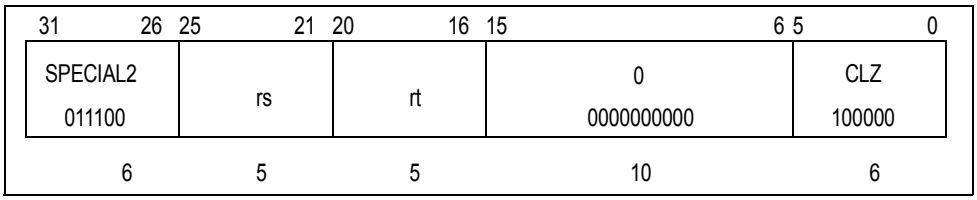


Figure 2.14 Count Leading Zeros Instruction Format

Format: CLZ rt,rs

Description: The contents of the lower 32-bits of general register rs is scanned from bit 31 to the least significant bit, and the number of leading zeros is written into general register rt. If no bits were set in general register rs (i.e. rs=0) the content of general register rt is 32.

Operation:

```
T:      rt <-- Leading_zeros(rs31..0)
```

Exception: None

Notes

Count Leading Ones

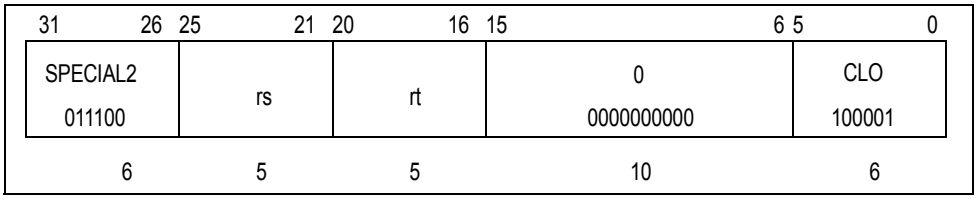


Figure 2.15 Count Leading Ones Instruction Format

Format: CLO rt,rs

Description: The contents of the lower 32-bits of general register rs is scanned from bit 31 to the least significant bit, the number of leading ones is written into general register rt. If no bits were cleared in general register rs (i.e. rs=0xffffffff) the content of general register rt is 32.

Operation:

```
T:      rt <-- Leading_ones(rs31..0)
```

Exception: None

Multiply (3 operand)

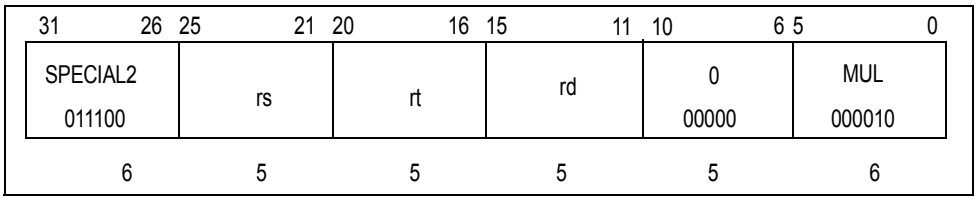


Figure 2.16 Multiple (3 operand) Format

Format: MUL rd, rs, rt

Description: The content of general registers rs and rt are multiplied, treating both operands as 32-bit unsigned values, and the result is placed in rd. No overflow exception occur under any circumstances.

Operation:

```
MUL rd, rs, rt    temp <-- rs31..0 x rt31..0
                  rd <-- (temp31)32 || temp31..0
                  HI <-- undefined
                  LO <-- undefined
```

Exception: None

Instruction Pipeline

Notes

Introduction

The RC64574/575 processor has a dual-issue, five-stage instruction pipeline with two parallel paths, one for integer (CPU) instructions and the other floating-point (FPU) instructions. Each stage in the CPU-instruction path takes one PCycle (one cycle of the processor clock, which runs at a multiple of the frequency of the system clock, SysClock). Thus, the execution of each CPU instruction takes at least five PCycles. A CPU instruction can take longer—for example, if the required data is not in the cache, the data must be retrieved from main memory. In the FPU-instruction path, most FPU instructions require more than one PCycle in the execution stage.

Once the pipeline has been filled, five instructions can be executed simultaneously. Figure 3.1 shows the five stages of the instruction pipeline.

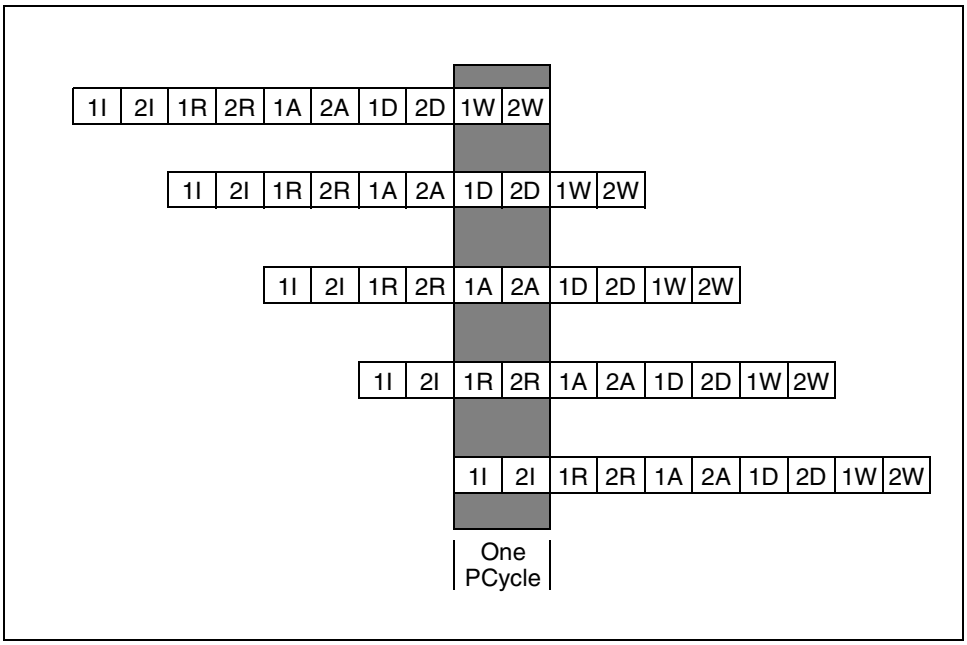


Figure 3.1 Instruction Pipeline Stages

Instruction Pipeline Stages

1I - Instruction Fetch, Phase One

2I - Instruction Fetch, Phase Two

1R - Register Read, Phase One

2R - Register Read, Phase Two

1A - Execution, Phase One

2A - Execution, Phase Two

1D - Data Load/Store, Phase One

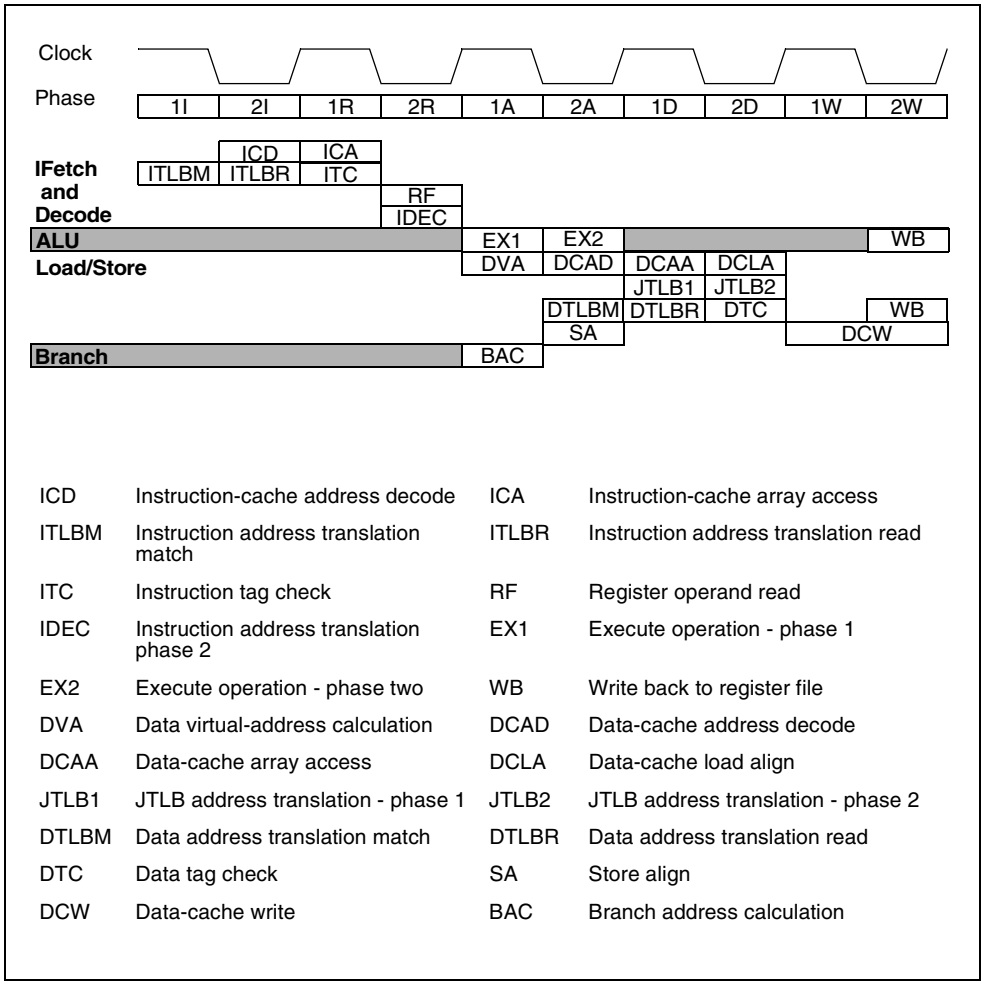
2D - Data Load/Store, Phase Two

1W - Write Back, Phase One

2W - Write Back, Phase Two

Notes

Figure 3.2 shows the CPU pipeline 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	Instruction address translation read
ITC	Instruction tag check	RF	Register operand read
IDEC	Instruction address translation phase 2	EX1	Execute operation - phase 1
EX2	Execute operation - phase two	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	JTLB address translation - phase 1	JTLB2	JTLB address translation - phase 2
DTLBM	Data address translation match	DTLBR	Data address translation read
DTC	Data tag check	SA	Store align
DCW	Data-cache write	BAC	Branch address calculation

Figure 3.2 Integer (CPU) Pipeline Activities

Notes

Dual Issue

The RC64574/575 dual-issue mechanism allows two instructions to be dispatched per processor cycle (PCycle) under the following condition: a floating-point ALU operation can be dispatched along with any other type of instruction, as long as the other instruction is not another floating-point ALU operation. In this context, “any other type of instruction” includes all integer instructions as well as floating-point loads and stores.

Figure 3.3 shows a simplified diagram of the dual issue mechanism.

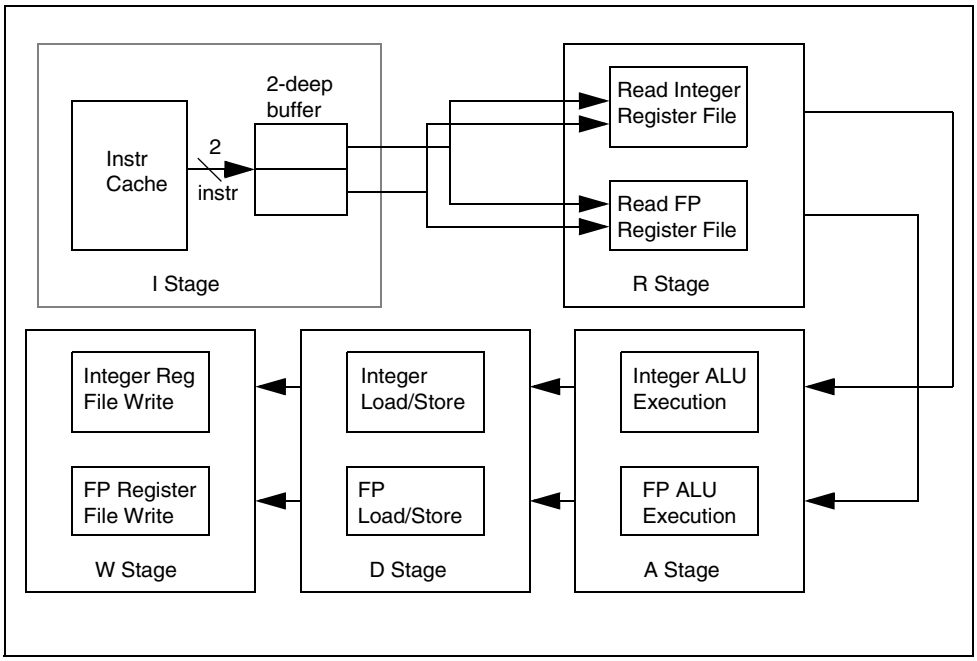


Figure 3.3 Dual-Issue Mechanism, Showing CPU and FPU Pipelines

The events that occur in each stage are:

I - Stage

Two instructions are fetched from the instruction cache and placed in a 2-deep instruction buffer. Issue logic determines the type of instruction and which pipeline the instruction is routed to. Also, the instruction cache tag is checked against the page frame number (PFN) obtained from the ITLB.

R - Stage

Any required operands are fetched from the appropriate register file, and the decision is made to either proceed or slip the instruction based on any interlock conditions. For branch instruction, the branch address is calculated.

A - Stage

The appropriate ALU begins the arithmetic, logical, or shift operation. The data virtual address is calculated for any load or store instructions. The appropriate ALU determines whether the branch condition is true. The data cache access is started.

D - Stage

The data cache access is completed. Data is shifted down and extended. Data address translation in the DTLB completes. The virtual to physical address translation in the JTLB is performed. The data cache tag is checked against the PFN from the DTLB or JTLB for any data cache access.

W - Stage

The processor resolves all exceptions. For register-to-register and load instructions, the result is written back to the appropriate register file.

Notes

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 comparison logic operating during the 1A pipeline stage of the branch. This allows the branch-target address calculated in the previous stage to be used for the instruction access in the following 1I stage. Figure 3.4 illustrates the branch delay.

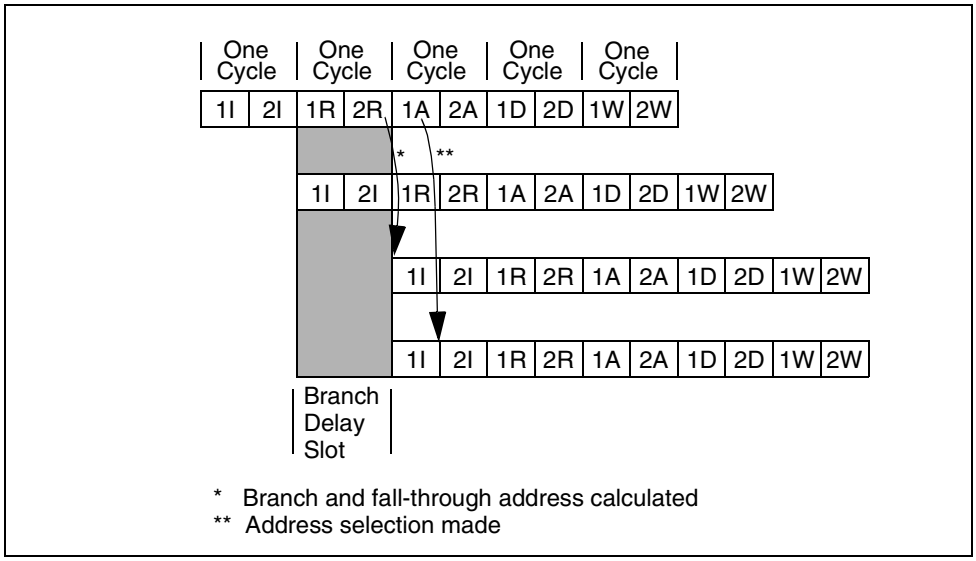


Figure 3.4 CPU-Pipeline Branch Delay

Load Delay

The completion of a load at the end of the 2D pipeline stage produces an operand that is available for the 1A pipeline stage of the subsequent instruction following the load delay slot. Figure 3.5 shows the load delay.

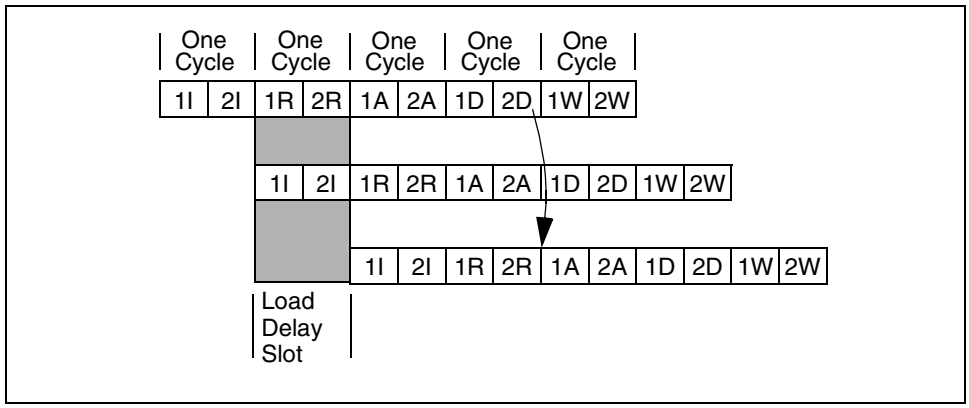


Figure 3.5 CPU-Pipeline Load Delay

Interlock and 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*.

There are two types of interlocks:

- *Stalls, which are resolved by halting the pipeline.*
- *Slips, which require one part of the pipeline to advance while another part of the pipeline is held static.*

Notes

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. See Table 3.1 For instance, a Reserved Instruction (RI) exception is raised in the execution (A) stage. Table 3.2 and Table 3.3 show CPU pipeline exceptions and CPU pipeline interlocks, respectively.

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

Table 3.1 Relationship of CPU-Pipeline Stage to Interlock Condition

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
DTLB	Data Translation or Address Exception
DTMod	TLB Modified
DBE	Data Bus Error

Table 3.2 CPU-Pipeline Exceptions

Notes

Exception	Description
DPErr	Data Parity Error
NMI	Non-maskable Interrupt
Reset	Reset

Table 3.2 CPU-Pipeline Exceptions

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
FCBusy	FP Busy

Table 3.3 CPU-Pipeline Interlocks

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 exception condition is detected, the processor aborts the instruction which caused the exception, as well as all subsequent instructions. When this instruction reaches the W stage, three events occur;

- *The exception flag causes the instruction to write various CP0 registers with the exception state,*
- *The current PC is changed to the appropriate exception vector address,*
- *The exception bits of earlier pipeline stages are cleared.*

This implementation allows all instructions which occurred before the exception to complete, and all instructions which occurred after the instruction to be aborted. Hence the value of the EPC is such that execution can be restarted. In addition, all exceptions are guaranteed to be taken in order. Figure 3.6 illustrates the exception detection mechanism for a Reserved Instruction (RI) exception.

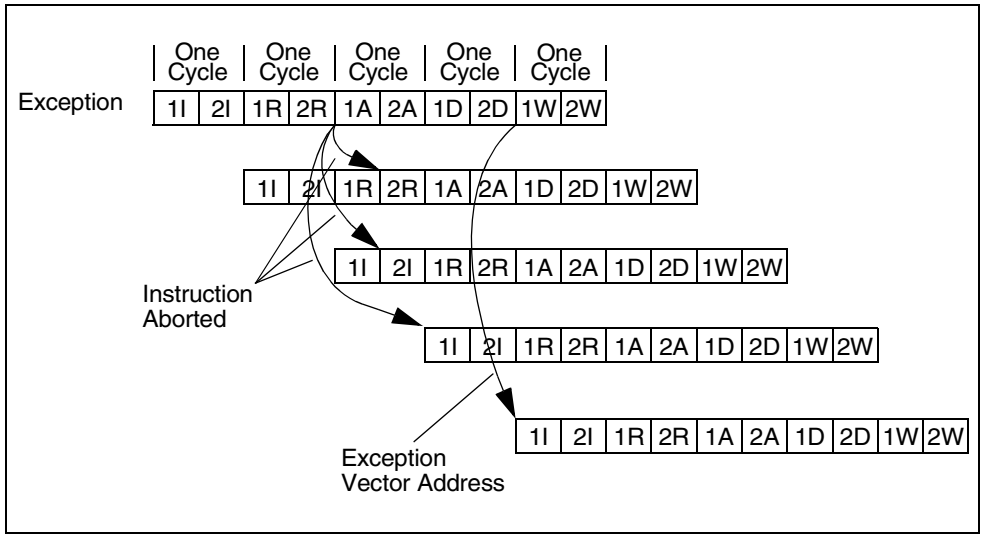


Figure 3.6 CPU-Pipeline Exception Detection Mechanism

Notes

Stall Conditions

A stall condition is used to suspend the pipeline for conditions detected after the R pipeline stage. When a stall occurs, the processor resolves the condition and then restarts the pipeline. Once the interlock is removed, the restart sequence begins two cycles before the pipeline resumes execution. The restart sequence reverses the pipeline overrun by inserting the correct information into the pipeline. Figure 3.7 shows a data cache miss stall.

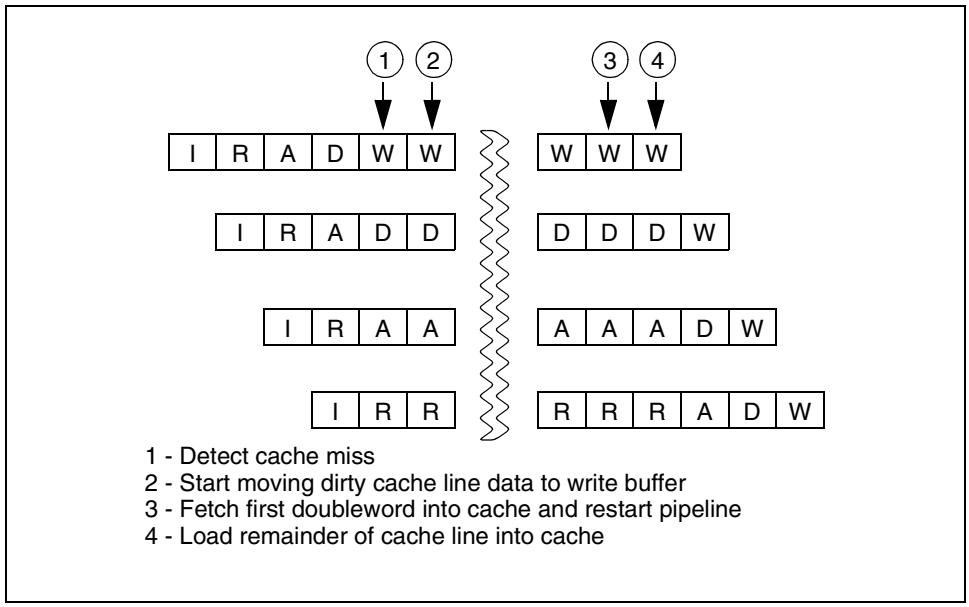


Figure 3.7 CPU-Pipeline Servicing of Data Cache Miss

The data cache miss is detected in the D stage of the pipeline. If the cache line to be replaced is dirty, the W bit is set and data is moved to the internal write buffer in the next cycle. The squiggly line in Figure 3.7 indicates the memory access. Once the memory is accessed and the first doubleword of data is returned, the pipeline is restarted. The remainder of the cache line is returned in subsequent cycles. The dirty data in the write buffer is written out to memory after the cache-line fill is completed.

Slip Conditions

During the 2R and 1A pipeline stages, internal logic determines whether it is possible to start the current instruction in this cycle. If all required source operands are available, as well as all hardware resources needed to complete the operation, the instruction is issued. Otherwise, the instruction slips. Slipped cycles are retried on subsequent cycles until they are issued. Pipeline stages D and W advance normally during slips in an attempt to resolve the conflict. NOPs are automatically inserted into the bubbles which are created in the pipeline. Instructions caused by “branch likely” instructions, ERET, or exceptions do not cause slips.

Figure 3.8 shows how an instruction can slip during an instruction-cache miss.

Notes

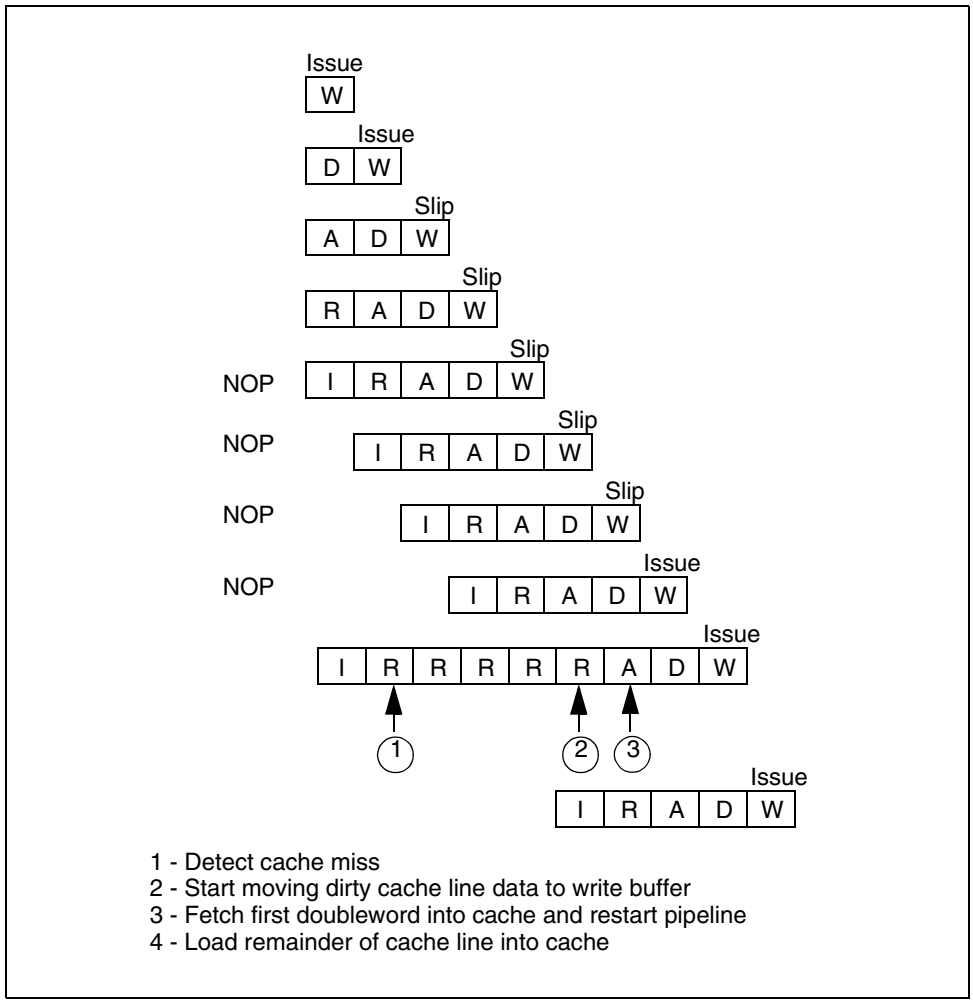


Figure 3.8 Slips During Instruction-Cache Miss

Instruction-cache misses are detected in the R-stage of the pipeline. Slips are detected in the A stage. Instruction-cache misses never require a writeback operation because writes are not allowed to the instruction cache. Unlike the data cache, early restart, where the pipeline is restarted after only a portion of the cache-line fill has occurred, is not implemented for the instruction cache. The requested cache line is loaded into the instruction cache in its entirety before the pipeline is restarted.

Write Buffer

The processor has a write buffer which improves the performance of write operations to external memory. All write cycles use the write buffer. The write buffer holds up to four 64-bit address and data pairs.

On a cache miss requiring a 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 decouples the CPU from the write to memory. If the write buffer is full, additional stores are stalled until there is room for them in the write buffer.

Memory Management Unit

Notes

Introduction

The processor provides a full-featured memory management unit (MMU) which uses an on-chip translation lookaside buffer (TLB) to translate virtual addresses into physical addresses.

This chapter describes the processor 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.

Address Spaces

This section describes the virtual and physical address spaces and the manner in which virtual addresses are translated into physical addresses in the TLB.

Virtual Address Space

The processor virtual address can be either 32 or 64 bits wide, depending on whether the processor is operating in 32-bit or 64-bit mode.

- *In 32-bit mode (extended address bit = 0), addresses are 32 bits wide. The maximum user process size is 2 gigabytes (2^{31}).*
- *In 64-bit mode (extended address bit = 1), addresses are 64 bits wide. The maximum user process size is 1 terabyte (2^{40}).*

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

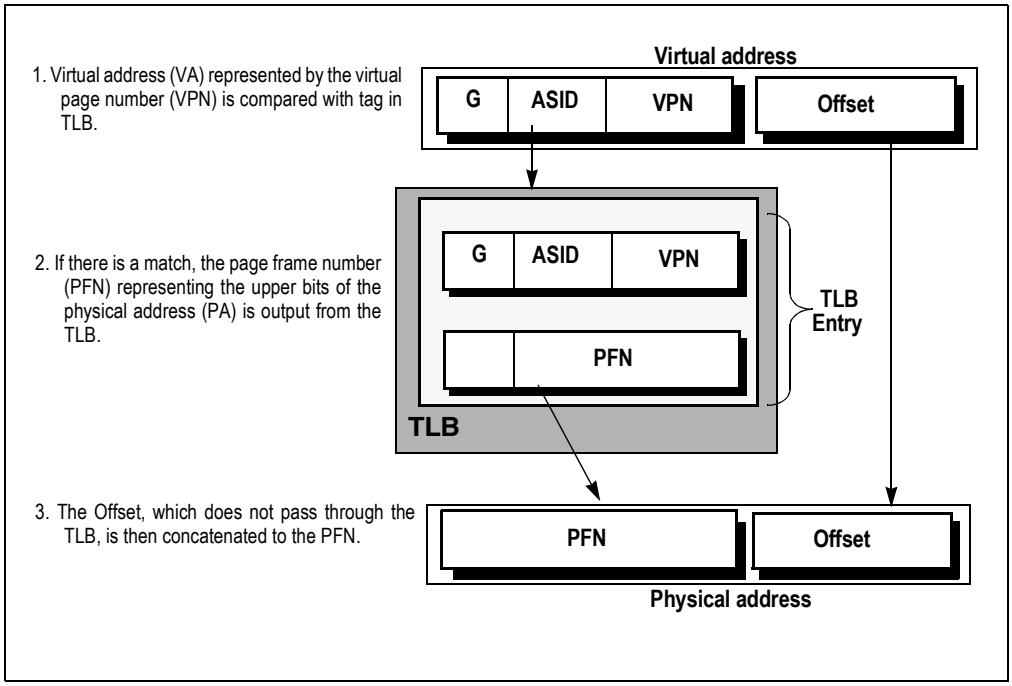


Figure 4.1 Overview of a Virtual-to-Physical Address Translation

As shown in Figures 4.2 and 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. The *Global* bit (*G*) is in the *EntryLo0* and *EntryLo1* registers.

Notes

Physical Address Space

Using a 36-bit address, the processor supports a physical address space of 64 gigabytes. The following section 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 addresses 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. The next two sections describe the 32-bit and 64-bit address translations.

32-bit Mode Virtual Address Translation

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

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

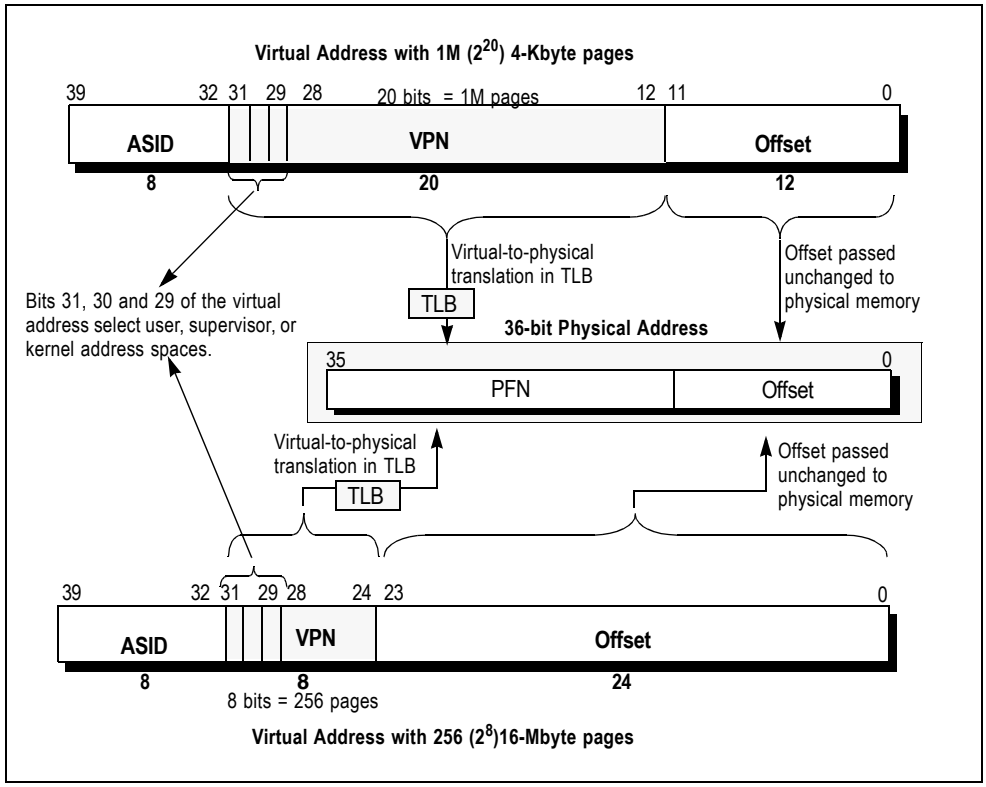


Figure 4.2 32-bit Mode Virtual Address Translation

Notes

64-bit Mode Virtual Address Translation

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

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

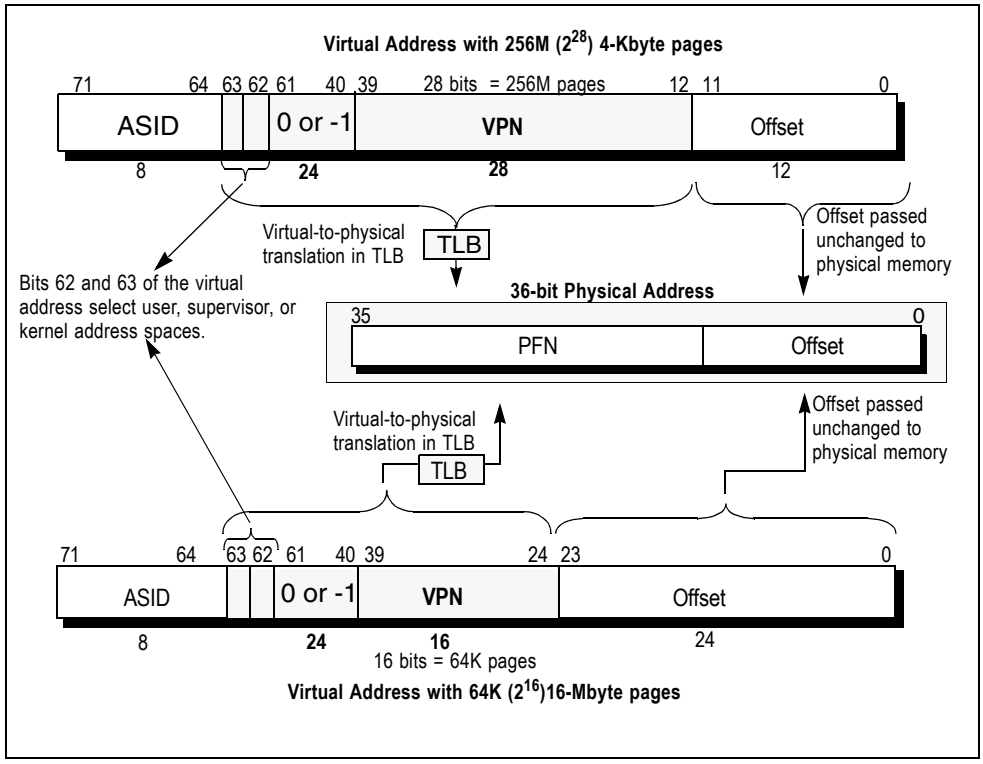


Figure 4.3 64-bit Mode Virtual Address Translation

Operating Modes

The processor has three operating modes—user, supervisor, and kernel—that function in both 32- and 64-bit operations. These modes are described in the next three sections.

User Mode Operations

Figure 4.4 shows User mode virtual address space. In User mode, a single, uniform virtual address space—labelled User segment—is available; its size is:

- *2 Gbytes (2^{31} bytes) in 32-bit mode. UX = 0 (useg)*
- *1 Tbyte (2^{40} bytes) in 64-bit mode. UX = 1 (xuseg)*

Notes

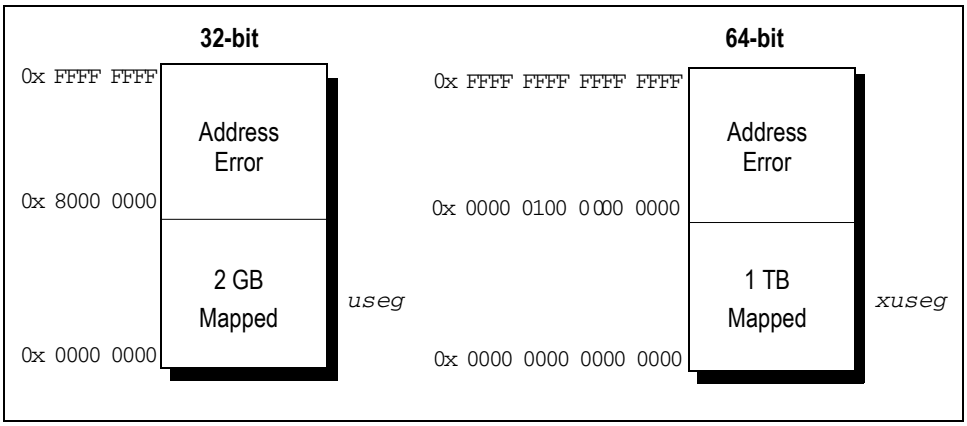


Figure 4.4 User Mode Virtual Address Space

The User segment starts at address 0 and the current active user process resides in either useg (in 32-bit mode) or xuseg (in 64-bit mode). 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 mode 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

32-bit User Mode (useg)

In User mode, when *UX* = 0 in the *Status* register, User mode 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*. 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. 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.

Notes

64-bit User Mode (*xuseg*)

In User mode, when *UX* = 1 in the *Status* register, User mode addressing is extended to 64-bits. In 64-bit User mode, the processor provides a single, uniform address space of 2^{40} bytes, 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.

Supervisor Mode Operations

Supervisor mode is designed for layered operating systems in which a true kernel runs in Kernel mode, and 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 addressing:

- *when SX = 0, 32-bit supervisor space is selected and TLB misses are handled by the 32-bit TLB refill exception handler*
- *when SX = 1, 64-bit supervisor space is selected and TLB misses are handled by the 64-bit XTLB refill exception handler. Figure 4.5 shows Supervisor mode address mapping. Table 4.2 lists the characteristics of the supervisor mode segments; descriptions of the address spaces follow.*

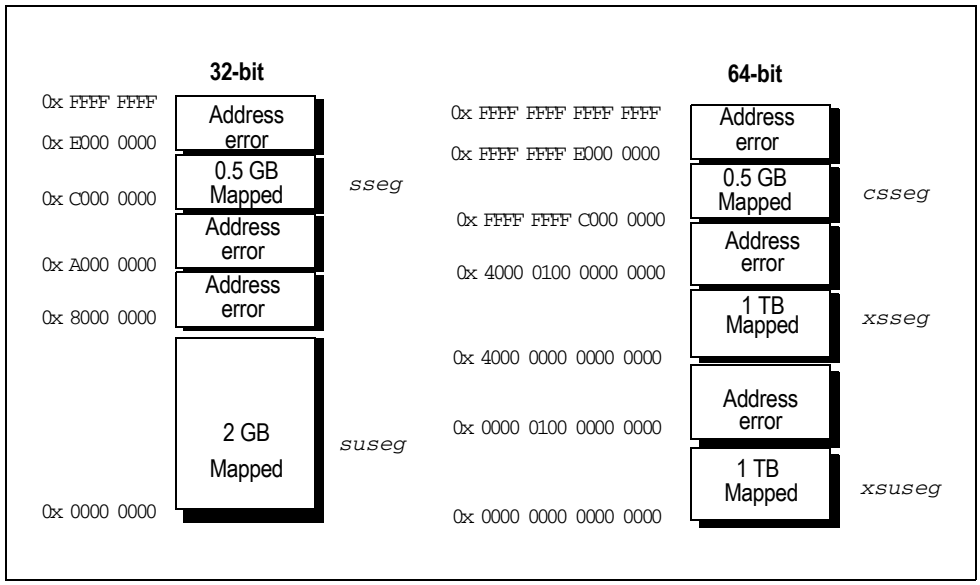


Figure 4.5 Supervisor Mode Address Space

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	ssseg	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

32-bit Supervisor Mode, User Space (*suseg*)

In Supervisor mode, when *SX* = 0 in the *Status* register 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 (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.

This mapped space starts at virtual address 0x0000 0000 and runs through 0x7FFF FFFF.

32-bit Supervisor Mode, Supervisor Space (*sseg*)

In Supervisor mode, when *SX* = 0 in the *Status* register 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 (512 Mbytes) 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 *SX* = 1 in the *Status* register 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 (1 Tbyte) 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 *SX* = 1 in the *Status* register 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.

Notes

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 *SX* = 1 in the *Status* register 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.

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 is selected.*
- *when KX = 1, 64-bit kernel space 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 virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 4.6. 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

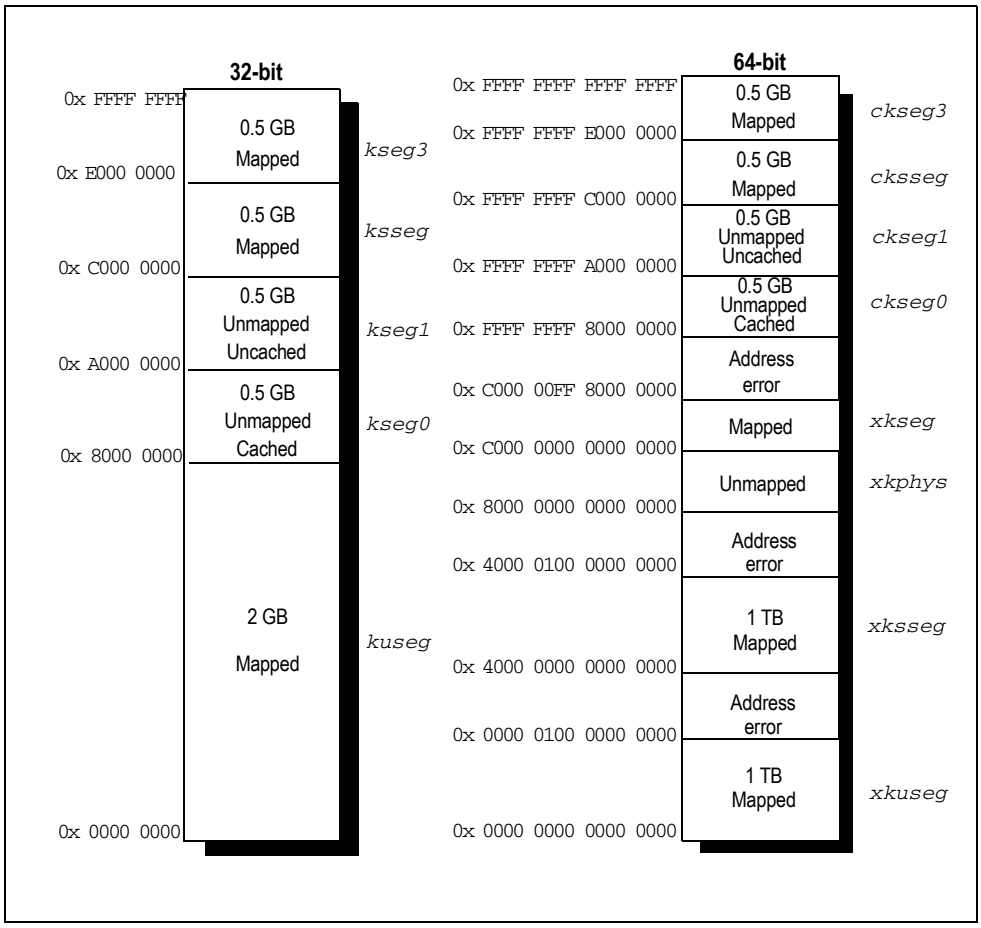


Figure 4.6 Kernel Mode Address Space

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

Notes

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

In Kernel mode, when *KX* = 0 in the *Status* register, 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 *KX* = 0 in the *Status* register and the most-significant three bits of the virtual address are 100_2, 32-bit *kseg0* virtual address space is selected; it is the 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. 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 *KX* = 0 in the *Status* register 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 2^{29}-byte (512-Mbyte) 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. 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 *KX* = 0 in the *Status* register 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 (512-Mbyte) 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 *KX* = 0 in the *Status* register 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 (512-Mbyte) 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	xksuseg	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^{40}–2^{31}) 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 *KX* = 1 in the *Status* register 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. When ERL = 1 in the *Status* register, the user address region becomes a 2^{31}-byte unmapped (that is, mapped directly to physical addresses) uncached address space.

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

In Kernel mode, when *KX* = 1 in the *Status* register 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 *KX* = 1 in the *Status* register 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 *KX* = 1 in the *Status* register 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 kernel 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

In Kernel mode, when *KX* = 1 in the *Status* register, bits 63:62 of the 64-bit virtual address are 11_2, and bits 61:31 of the virtual address equal –1. The lower 4 bytes of address, as shown in Figure 4.4, 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 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.*

Notes

System Control Coprocessor

Notes

System Control Coprocessor

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, 96-page TLB. The sections that follow describe how the processor uses the memory management-related registers.

Each CP0 register has a unique number that identifies it; this number is referred to as the *register number*. For instance, the *Page Mask* register is register number 5.

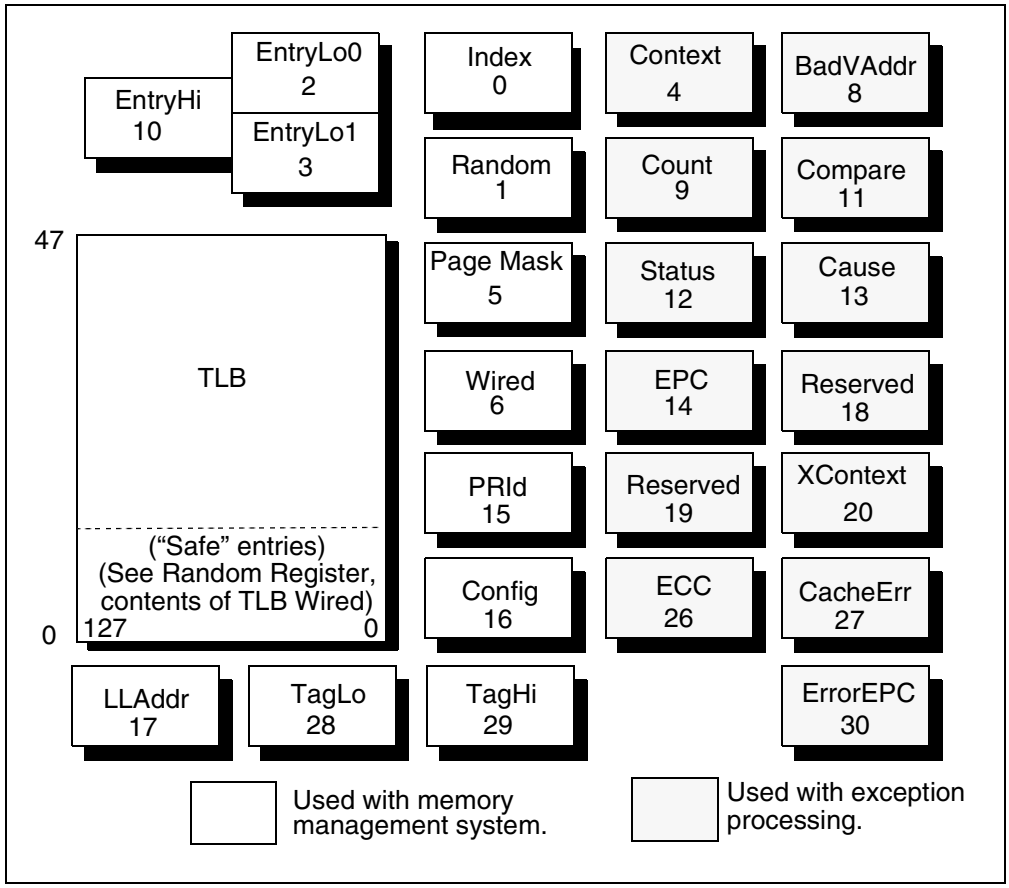


Figure 5.1 CP0 Registers and the TLB

Translation Lookaside Buffer (TLB)

Mapped virtual addresses are translated into physical addresses using an on-chip TLB.[1] The TLB is a fully associative memory that holds 48 entries, which provide mapping to 48 odd/even page pairs (96 pages). 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 page size can be configured, on a per-entry basis, at 4Kbytes, 16Kbytes, 64Kbytes, 256Kbytes, 1Mbytes, 4Mbytes, or 16Mbytes

1. There are virtual-to-physical address translations that occur outside of the TLB. For example, addresses in *the kseg0* and *kseg1* spaces are unmapped translations. In these spaces the physical address is 0x000_0000_0 || VA[28:0].

Notes

Format of a TLB Entry

Figure 5.2 shows the TLB entry formats for both 32- and 64-bit modes. Each field of an entry has a corresponding field in the *EntryHi*, *EntryLo0*, *EntryLo1*, or *PageMask* registers. Figure 5.3 and Figure 5.4 show the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers. The formats of these registers are nearly the same as the TLB-entry formats. The one exception is the *Global* field (*G* bit), which is used in the TLB, but is reserved in the *EntryHi* register and is taken from *EntryLo register*.

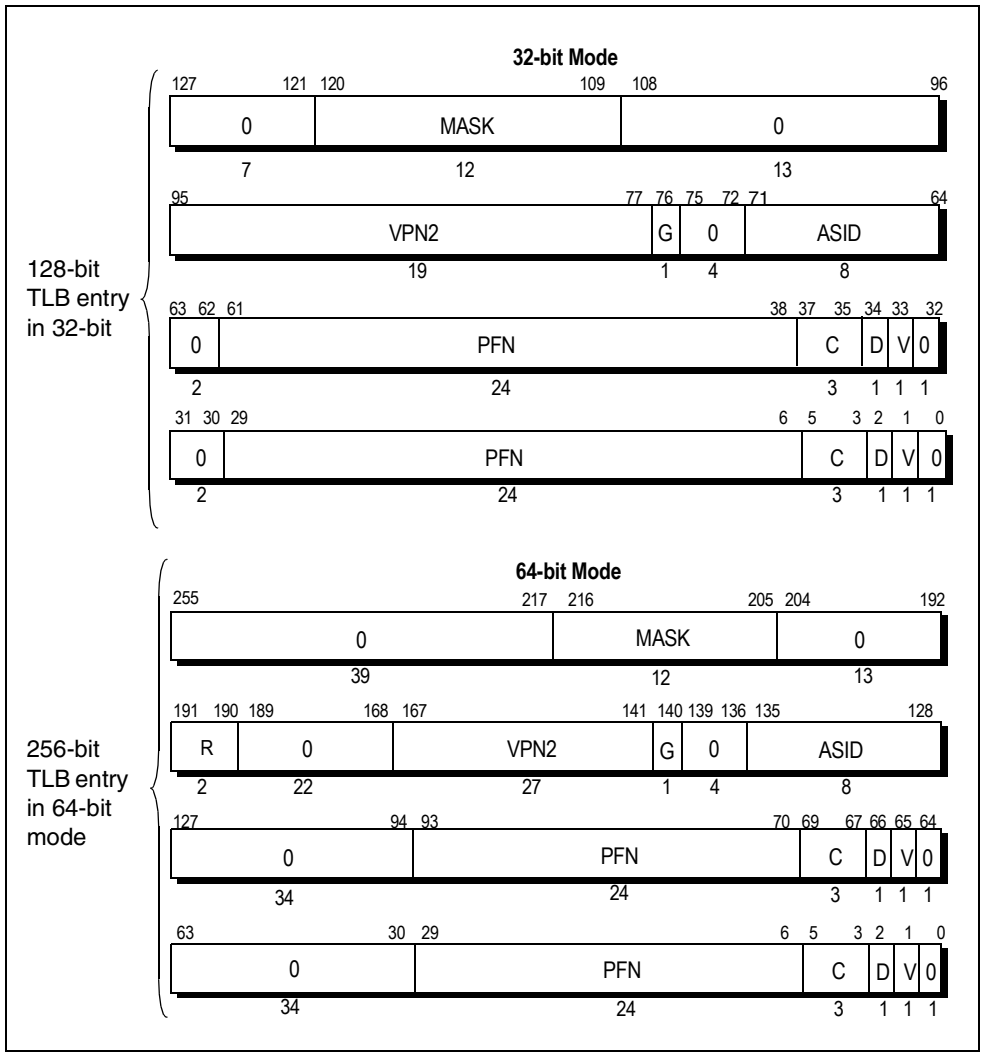
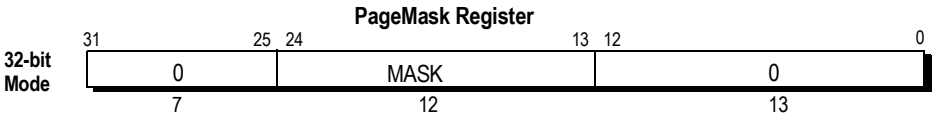
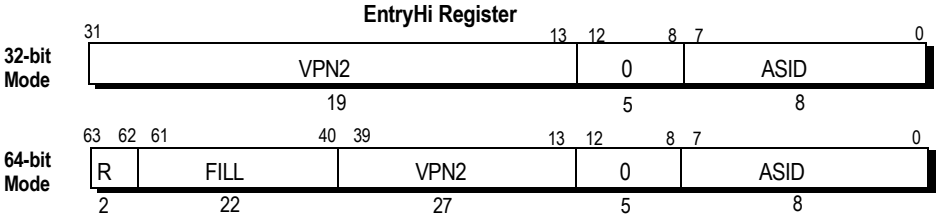


Figure 5.2 Format of a TLB Entry

Notes

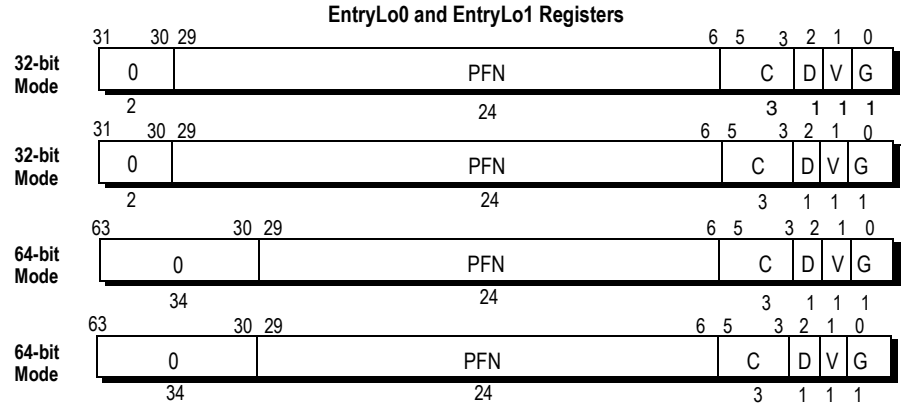


Mask..... Page comparison mask.
0..... Reserved. Must be written as zeroes, and returns zeroes when read.



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 → user, 01 → supervisor, 11 → kernel) used to match $vAddr_{63...62}$
Fill.....Reserved. 0 on read; ignored on write.
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

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.

Notes

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 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 describes the *Index* register fields.

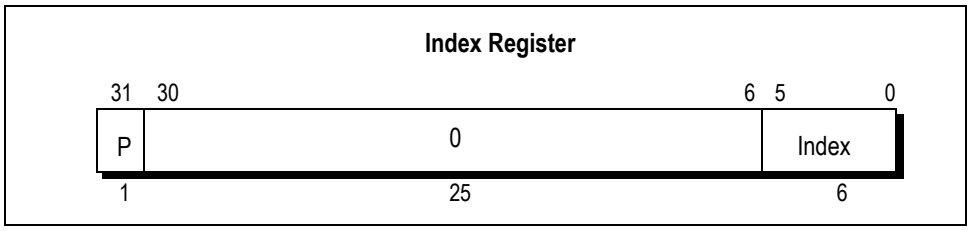


Figure 5.5 Index Register

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 at one less than the total number of TLB entries (47 maximum).*

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 describes the *Random* register fields.

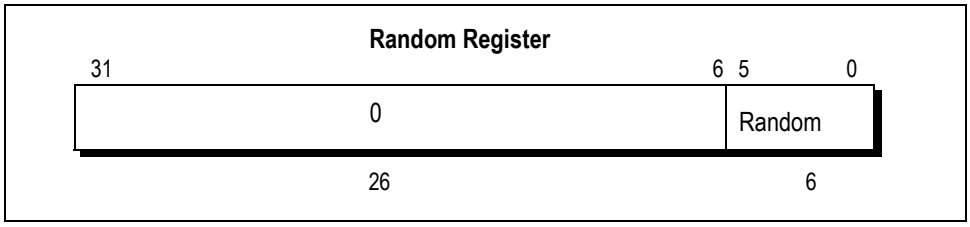


Figure 5.6 Random Register

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 5-3 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; it holds a comparison mask that sets the variable page size for each TLB entry. 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 Table 5.4, 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 fixed, nonreplaceable entries, which cannot be overwritten by a TLB write operation. Random entries can be overwritten.

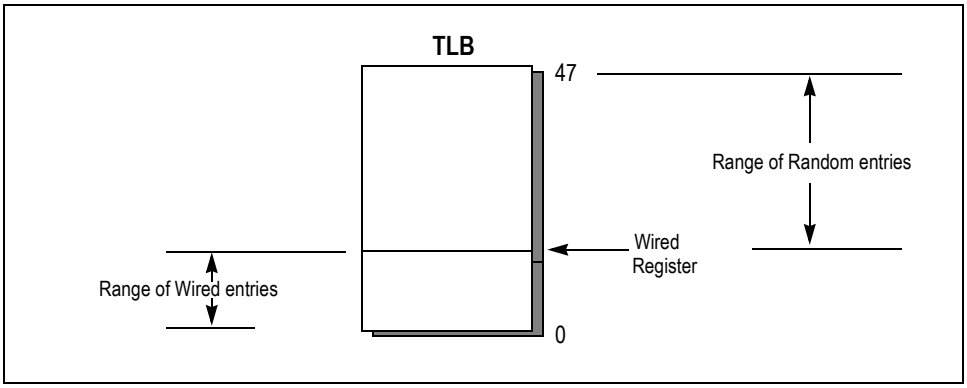


Figure 5.7 Wired Register Boundary

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 describes the register fields.

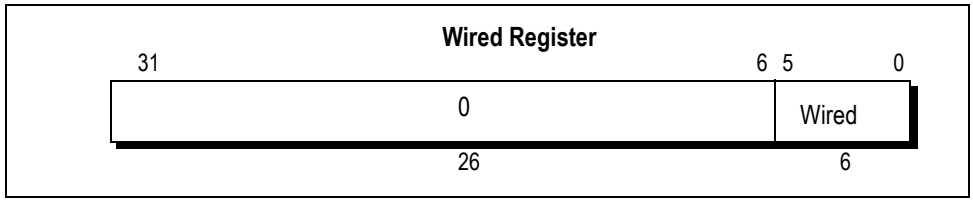


Figure 5.8 Wired Register

Field	Description
Wired	TLB Wired boundary
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 5.5 Wired Register Field Descriptions

Notes

EntryHi Register (CP0 Register 10)

The *EntryHi* register holds the high-order bits of a TLB entry for TLB read and write operations. The *EntryHi* register is accessed by the TLB Probe, TLB Write Random, TLB Write Indexed, and TLB Read Indexed instructions. 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. Figure 5.3 on page 5-3 shows the format of the EntryHi Register.

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.

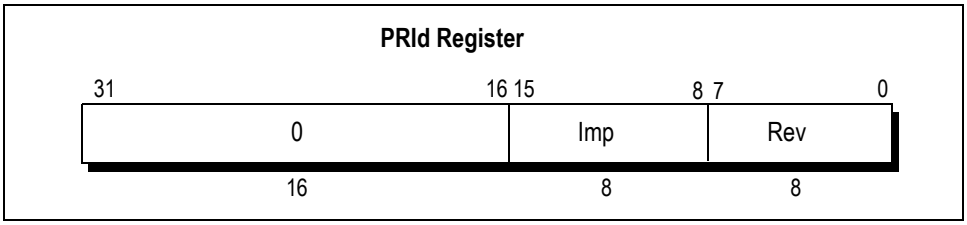


Figure 5.9 Processor Revision Identifier Register Format

Field	Description
Imp	Implementation number Imp=0x15
Rev	Revision number. May vary.
0	Reserved. Must be written as zeroes, and 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 RC64574/575 processor is 0x15. 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.

Config Register (16)

The *Config* register specifies various configuration options. Some configuration options, as defined by *Config* bits 31:13 and 11:3, are set by the hardware during Reset and are included in the *Config* register as read-only status bits. Other configuration options are read/write (as indicated by *Config* register bits 2:0) and controlled by software; on Reset, this field is undefined. Certain configurations have restrictions. The *Config* register should be initialized by software before caches are used. Caches should be re-initialized after any change is made.

Figure 5.10 shows the format of the *Config* register; Table 5.7 describes the *Config* register fields.

Notes

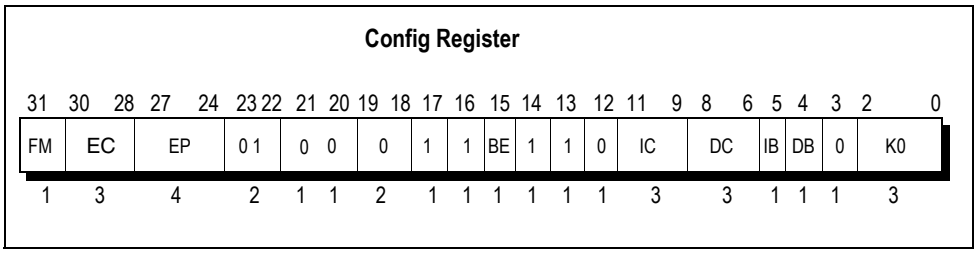


Figure 5.10 Config Register Format

Field	Description
FM	Fast Multiply 0 → regular multiply (default to root) 1 → fast multiply
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	Transmit data pattern (pattern for write-back data): 0 → D — 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 → DDxxxxxxDDxxxxxx — 2 Doublewords every 8 cycles 8 → DxxxDxxxDxxxDxxx — 2 Doublewords every 8 cycles
BE	Big Endian Mode: 0 → Little Endian 1 → Big Endian
IC	Primary I-cache Size (I-cache size = 2^{12+IC} bytes). In the RC64574/575 processor, this is set to 32 Kbytes (IC=3).
DC	Primary D-cache Size (D-cache size = 2^{12+DC} bytes). In the RC64574/575 processor, this is set to 32 Kbytes (DC=3).
IB	Primary I-cache line size. In the RC64574/575 processor, this is set to 32 bytes (IB=1). 0 → Reserved 1 → 32 bytes
DB	Primary D-cache line size. In the RC64574/575 processor, this is set to 32 bytes (DB=1). 0 → Reserved 1 → 32 bytes
K0	*kseg0* coherency algorithm (see *EntryLo0* and *EntryLo1* registers and the *C* field of Table 5.1)

Table 5.7 Config Register Fields

Notes

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).

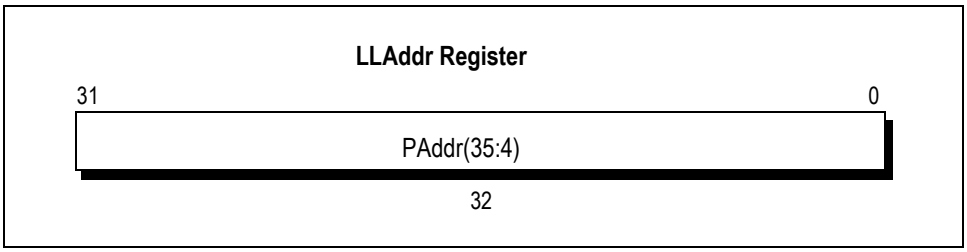


Figure 5.11 LLAddr Register Format

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

The *TagLo* and *TagHi* registers are 32-bit read/write registers that hold either the primary cache tag and parity, or the secondary cache tag and ECC during cache initialization, cache diagnostics, or cache error processing. The *Tag* registers are written by the CACHE and MTC0 instructions.

The *P* and *ECC* fields of these registers are ignored on Index Store Tag operations. Parity and ECC are computed by the store operation. Avoid using Instruction Index Store Tag operations except during primary-cache initialization, because the IType field determines the instruction type and problems occur if this specified incorrectly.

Figure 5.12 shows the format of these registers for primary cache operations. Figure 5.13 shows the format of these registers for secondary cache operations. Table 5.8 lists the field definitions of these registers.

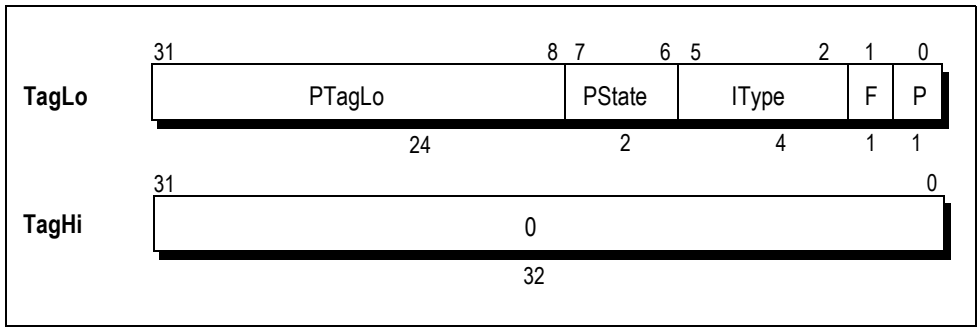


Figure 5.12 TagLo and TagHi Register (P-cache) Formats

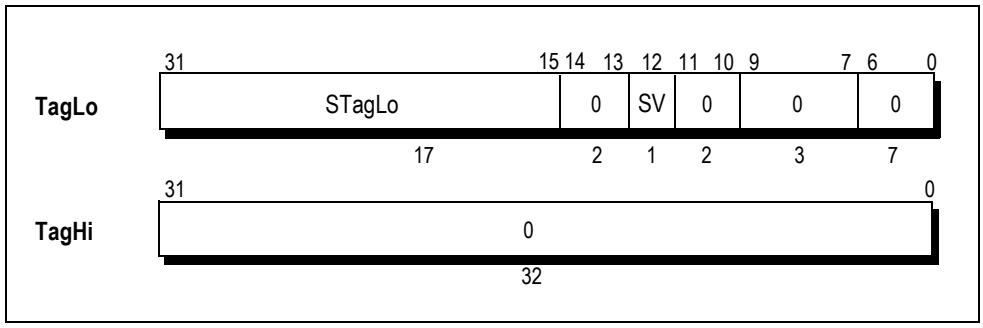


Figure 5.13 TagLo and TagHi Register (S-cache) Formats

Notes

Field	Description
PTagLo	Specifies the physical address bits 35:12.
PState	Specifies the primary cache state.
IType	Instruction-type bits: (28) MS instruction, (25) LS instruction. Specifies the even-word instruction type as integer or floating-point.
F	The FIFO bit, used to implement FIFO refill of the cache.
P	Specifies the primary tag even parity bit.
STagLo	Specifies the physical address bits 35:19.
SV	Specifies the Valid bit for secondary cache.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 5.8 Cache Tag Register Fields

Virtual-to-Physical 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. One of the following comparisons are also made:

- *In 32-bit mode, the highest 7-to-19 bits (depending upon the page size) of the virtual address are compared to the contents of the TLB virtual page number.*
- *In 64-bit mode, 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 5.14 illustrates the TLB address translation process.

Notes

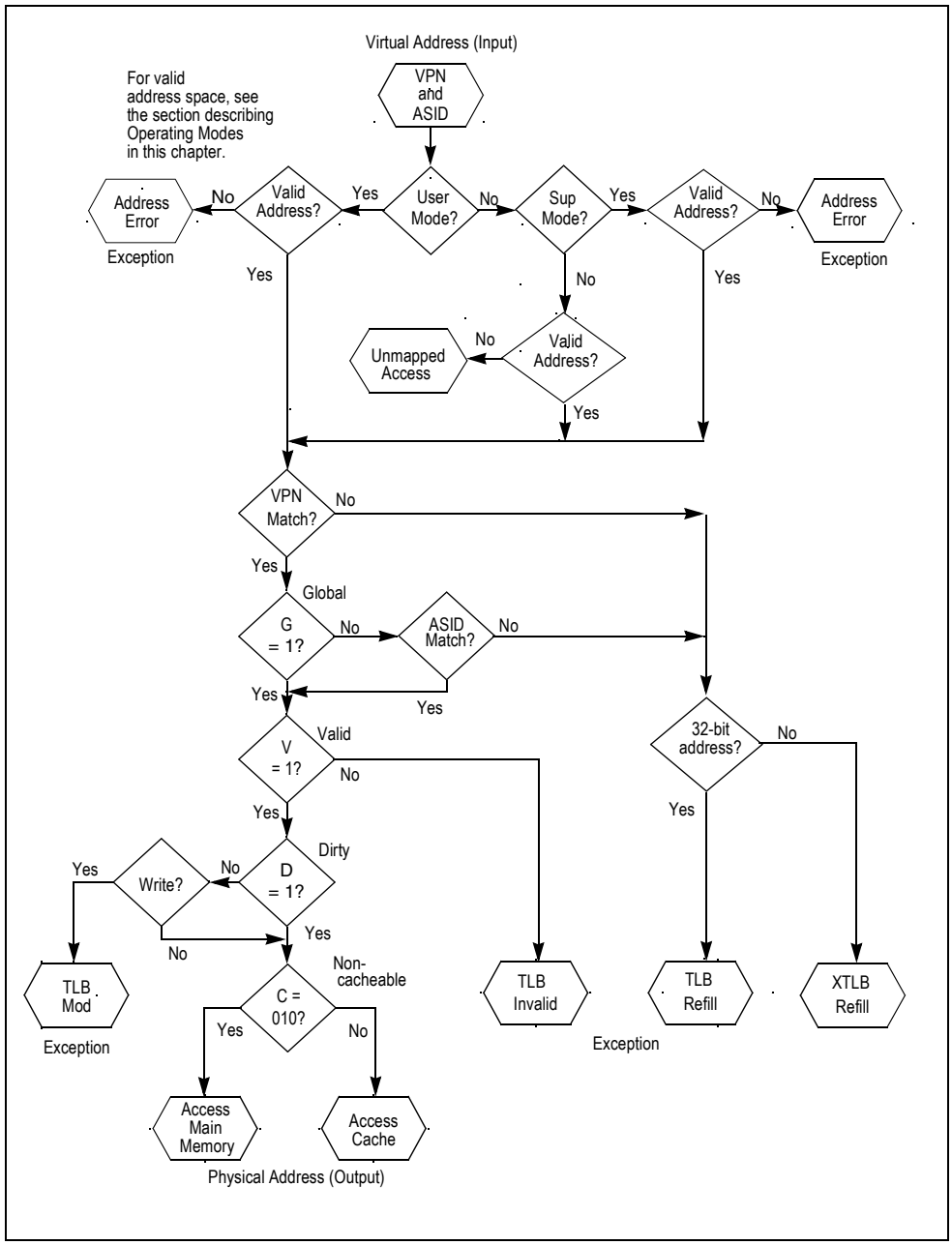


Figure 5.14 TLB Address Translation

TLB Hits and Misses

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. If there is no TLB entry that matches the virtual address, a TLB miss exception occurs 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.

Multiple TLB Matches

The processor does not provide any detection or shutdown mechanism for multiple matches in the TLB. The result of this condition is undefined, and software is expected to never allow this to occur.

Notes

Invalid TLB Accesses

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.

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 TLB Instructions

Integer (CPU) Exceptions

Notes

Introduction

This section describes the integer exception processing done by the CPU, including an explanation of exception processing, followed by the format and use of each CPU exception register. FPU exception processing is described in a later chapter.

The processor receives exceptions from a number of sources, including translation lookaside buffer (TLB) misses, 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. The processor then disables interrupts and forces execution of a software exception processor (called a *handler*) located at a fixed address. The handler typically saves the context of the processor, including the contents of the program counter, the current operating mode (User or Supervisor), and the status of the interrupts (enabled or disabled). This context is 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.

Exception Processing Registers

The *System Control Coprocessor (CP0)* registers are used in exception processing. Table 6.1 lists these registers and their unique *register numbers*. For instance, the *ECC* register is register number 26. The remaining CP0 registers are used in memory management and are described in Chapter 5.

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. The registers in Table 6.1 are used in exception processing, and are described in the sections that follow.

Register Name	R/W	Register Number
Context	R/W	4
BadVAddr (Bad Virtual Address)	R	8
Count	R/W	9
Compare register	W[1]	11
Status	R/W	12
Cause	R/W	13
EPC (Exception Program Counter)	R/W	14
XContext	R/W	20
ECC	R/W	26
CacheErr (Cache Error and Status)	R	27
ErrorEPC (Error Exception Program Counter)	R/W	30

Table 6.1 CPO Exception Processing Registers

1. For diagnostics, this register is R/W.

Notes

CPU general registers are interlocked and the result of an instruction can normally be used by the next instruction; if the result is not available right away, the processor stalls until it is available. CP0 registers and the TLB are not interlocked, however; there may be some delay before a value written by one instruction is available to following instructions. This delay may need to be explicitly coded in software.

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 describes the *Context* register fields.

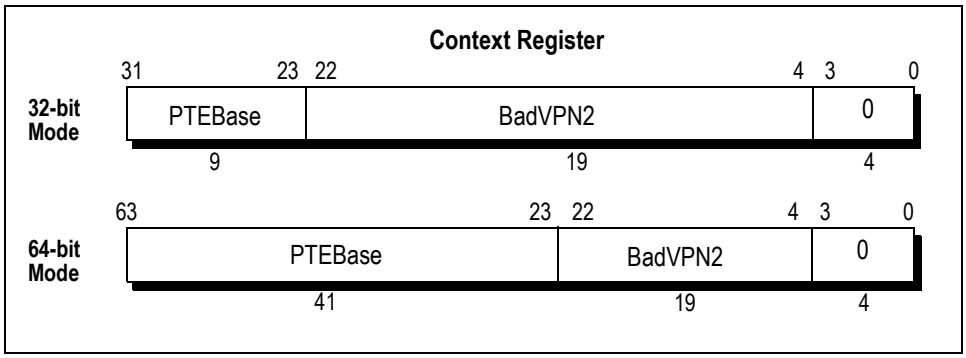


Figure 6.1 Context Register Format

Field	R/W	Description
BadVPN2	R	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	R/W	This field is 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

The 19-bit *BadVPN2* field contains bits 31:13 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 can directly address the pair-table of 8-byte PTEs. For other page and PTE sizes, shifting and masking this value produces the appropriate address.

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: TLB Invalid, TLB Modified, TLB Refill, Virtual Coherency Data Access, or Virtual Coherency Instruction Fetch. Figure 6.2 shows the format of the *BadVAddr* register. The *BadVAddr* register does not save any information for bus errors, since bus errors are not addressing errors.

Notes

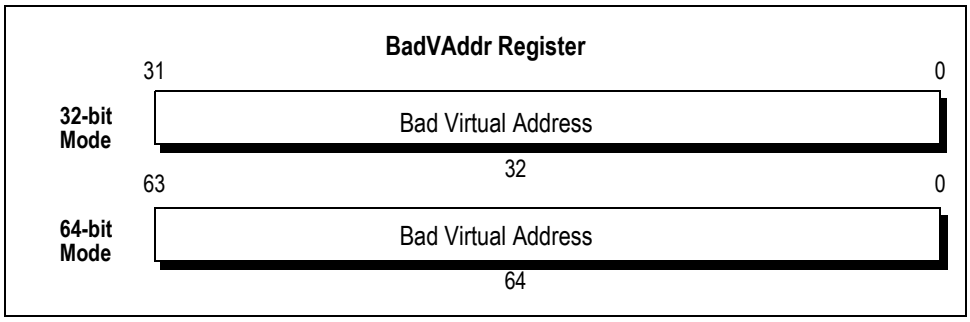


Figure 6.2 BadVAddr Register Format

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.

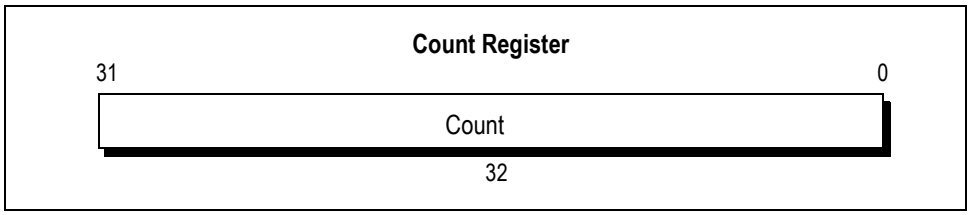


Figure 6.3 Count Register Format

Compare Register (11)

The *Compare* register acts as a timer (see also the *Count* register); it 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. This causes an interrupt 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. In normal use however, the *Compare* register is write-only. Figure 6.4 shows the format of the *Compare* register.

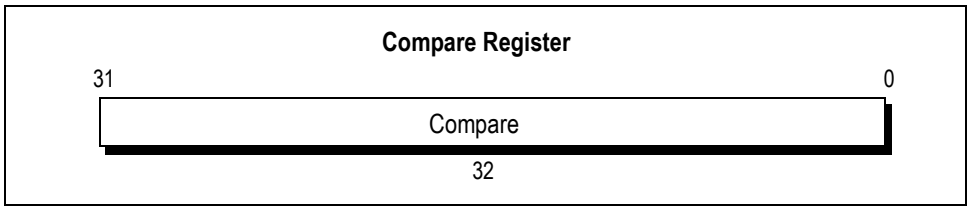


Figure 6.4 Compare Register Format

Status Register (12)

The *Status* register (SR) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. Figure 6.5 shows the format of the *Status* register.

Notes

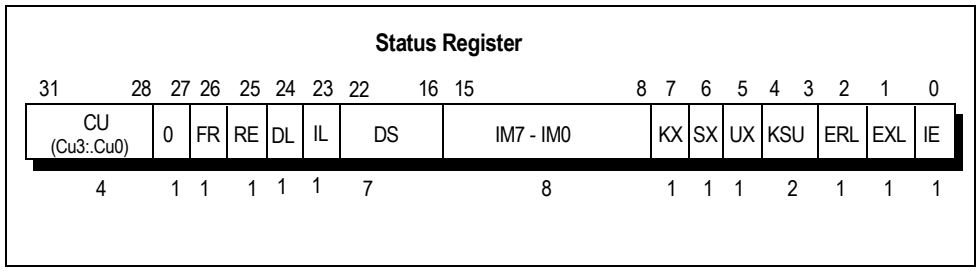


Figure 6.5 Status Register

Table 6.3 describes the *Status* register 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 be asserted, 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[1:0] are software interrupt masks, 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 access to an unusable coprocessor causes an exception.*
- *The 9-bit Diagnostic Status (DS) field 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; reverse-endian selection is used in Kernel and Supervisor modes, and in the User mode when the RE bit is 0. Setting the RE bit to 1 inverts the User mode endianness.*
- *The Data Cache Lock (DL) bit, bit 24: when set to one, it locks the data cache line currently being written, and when set to zero it does not lock the data cache line.*
- *The Instruction Cache Lock (IL) bit, bit 23: when set to one, it locks the instruction cache line currently being written, and when set to zero it does not lock the instruction cache line.*

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. Setting CU_3 enables the MIPS IV instruction set. 1 → usable 0 → unusable
0	Reserved. Set to 0.
FR	Enables additional floating-point registers 0 → 16 registers 1 → 32 registers
RE	*Reverse-Endian* bit, valid in User mode.
DS	*Diagnostic Status* field (see Figure 6.6).
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. 0 → disabled 1→ enabled

Table 6.3 Status Register Fields (Part 1 of 2)

Notes

Field	Description
KX	Enables 64-bit addressing in Kernel mode. The extended-addressing TLB refill exception is used for TLB misses on kernel addresses. 0 → 32–bit 1 → 64–bit
SX	Enables 64-bit 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 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; set by the processor when Reset, Soft Reset, NMI, or Cache Error exception are taken. 0 → normal 1 → error
EXL	Exception Level; set by the processor when any exception other than Reset, Soft Reset, NMI, or Cache Error exception are taken. 0 → normal 1 → exception
IE	Interrupt Enable 0 → disable interrupts 1 → enable interrupts

Table 6.3 Status Register Fields (Part 2 of 2)

Figure 6.6 and Table 6.4 provide additional information on the *Diagnostic Status* (*DS*) field. All bits in the *DS* field except *TS* are readable and writable.

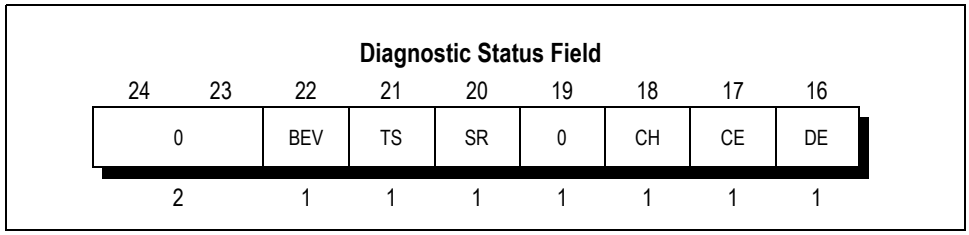


Figure 6.6 Status Register DS Field

Fields of the *Status* register set the following modes and access states:

- ***Interrupt Enable:*** *Interrupts are enabled by the settings of the IM bits when all of the following conditions are true:*
  - *IE = 1*
  - *EXL = 0*
  - *ERL = 0*
- ***Operating Modes:*** *The following CPU Status register bit settings are required for User, Kernel, and Supervisor modes.*
  - *User Mode: KSU = 10_2, EXL = 0, and ERL = 0.*
  - *Supervisor Mode: KSU = 01_2, EXL = 0, and ERL = 0.*
  - *Kernel Mode: KSU = 00_2, or EXL = 1, or ERL = 1.*

Notes

- ***32- and 64-bit Modes:*** *The following CPU Status register bit settings select 32- or 64-bit operation for User, Kernel, and Supervisor operating modes. Enabling 64-bit operation permits the execution of 64-bit opcodes and translation of 64-bit addresses. 64-bit operation for User, Kernel and Supervisor modes can be set independently.*
  - *64-bit addressing for Kernel mode is enabled when KX = 1. 64-bit operations 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.*

Bit	Description
BEV	Controls the location of TLB refill and general exception vectors. Refer to the Exception Vector Location section later in this chapter. 0 → normal 1→ bootstrap
0	Reserved. Must be written as zeroes. Returns zeroes when read.
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, Hit Set Virtual, or Create Dirty Exclusive for a secondary 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 or ECC errors cannot cause exceptions. 0 → parity/ECC remain enabled 1 → disables parity/ECC
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 6.4 Status Register Diagnostic Status Bits

Access to the kernel address space is allowed when the processor is in Kernel mode. Access to the supervisor address space is allowed when the processor is in the Kernel or Supervisor operating mode. Access to the user address space is allowed in any of the three operating modes.

The contents of the *Status* register are undefined at reset, except for the *ERL* and *BEV* bits, which are set to 1. The *SR* bit distinguishes between the Reset exception and the Soft Reset exception (caused either by Reset* or Nonmaskable Interrupt [NMI]).

Cause Register (13)

The 32-bit read/write *Cause* register describes the cause of the most recent exception. Figure 6.7 shows the fields of this register. Table 6.5 describes the *Cause* register fields. 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. Table 6.6 describes the cause register execution code fields.

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.

Table 6.5 Cause Register Fields

Notes

Field	Description
IP (7:0)	Indicates an interrupt is pending. 1 → interrupt pending 0 → no interrupt
ExcCode	Exception code field (see Table 6.6)
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 6.5 Cause Register Fields

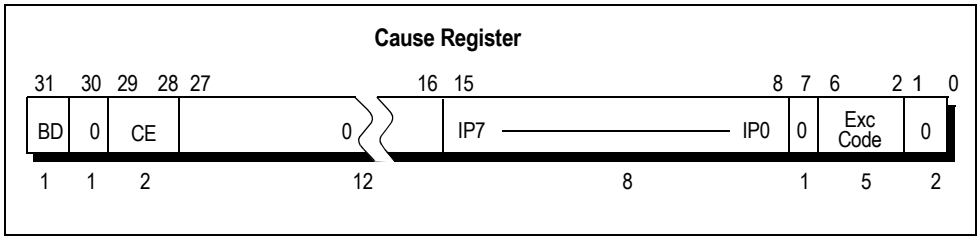


Figure 6.7 Cause Register Format

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 6.6 Cause Register ExcCode Fields

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*

Notes

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

The processor does not write to the *EPC* register when the *EXL* bit in the *Status* register is set to a 1. Figure 6.8 shows the format of the *EPC* register.

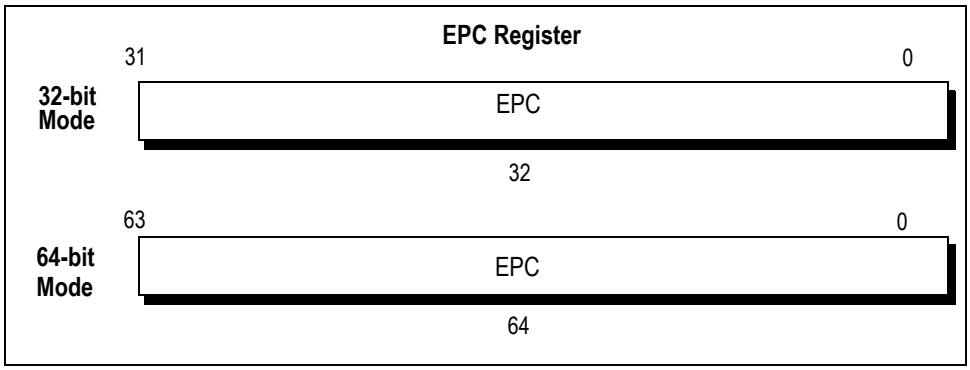


Figure 6.8 EPC Register Format

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 *Context* register to address the current page map, which resides in the kernel-mapped segment *kseg3*. Figure 6.9 shows the format of the *XContext* register; Table 6.7 describes the *XContext* register fields.

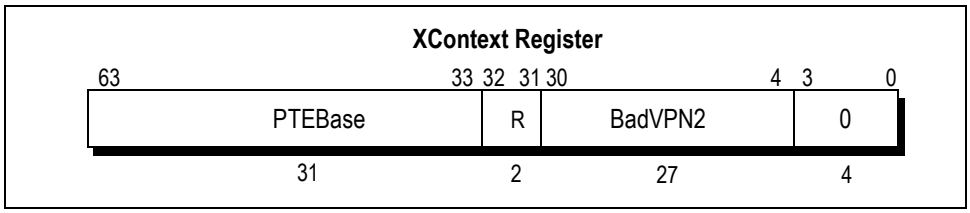


Figure 6.9 XContext Register Format

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.

Notes

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 6.7 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 ECC and parity are loaded from and stored to the *TagLo* register.) Figure 6.10 shows the format of the *ECC* register; Table 6.8 describes the register fields.

The *ECC* register is loaded by the Data Cache Index Load Tag operation. The content of the ECC register is:

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

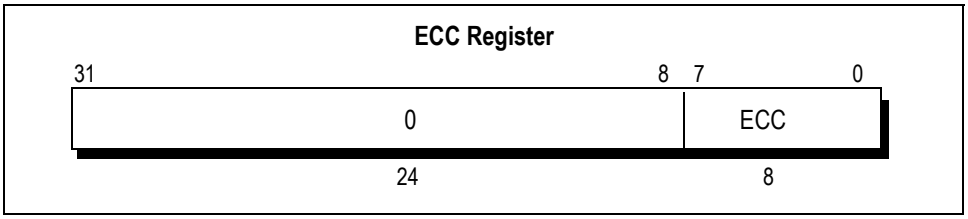


Figure 6.10 ECC Register Format

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 6.8 ECC Register Fields

Cache Error (CacheErr) Register (27)

The 32-bit read-only *CacheErr* register processes ECC errors in the secondary cache and parity errors in the primary cache. The 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. Parity errors cannot be corrected.

Figure 6.11 shows the format of the *CacheErr* register and Table 6.9 describes the *CacheErr* register fields.

Notes

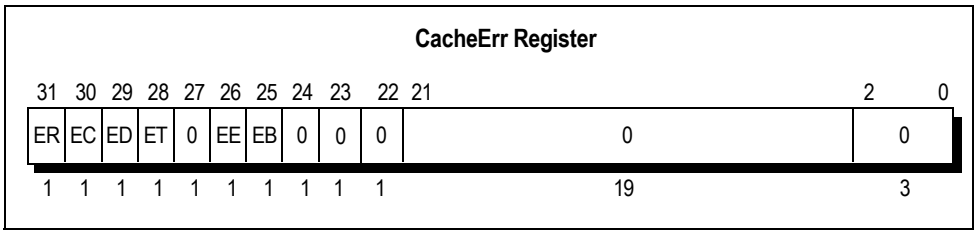


Figure 6.11 CacheErr Register Format

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
EE	This bit is set if the error occurred on the SysAD bus.
EB	This bit is 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.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 6.9 Cache Register Fields

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

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

- *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 6.12 shows the format of the *ErrorEPC* register.

Notes

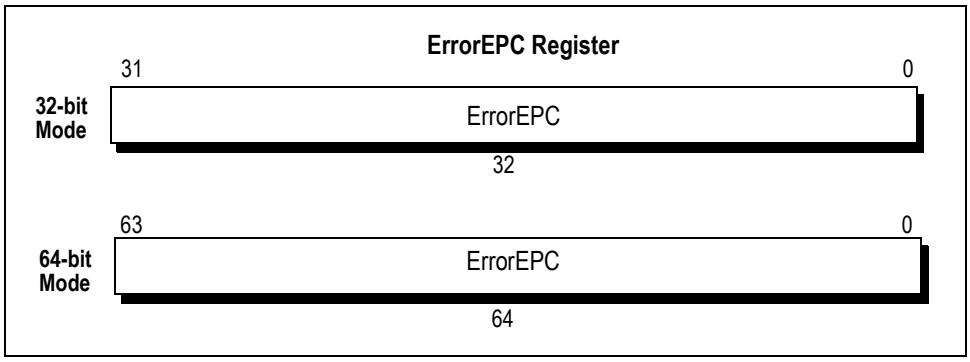


Figure 6.12 ErrorEPC Register Format

Overview of Exception Types and Handling

When the *EXL* bit in the *Status* register is 0, either User, Supervisor, or Kernel operating mode is specified by the *KSU* bits in the *Status* register. When the *EXL* 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 changes *KSU* to Kernel mode and resets the *EXL* bit back to 0. When restoring the state and restarting, the handler restores the previous value of the *KSU* field and sets the *EXL* bit back to 1. Returning from an exception also resets the *EXL* bit to 0.

Sample Hardware Processes For Various Exceptions

In the following sections, sample hardware processes for various exceptions are shown, together with the servicing required by the handler (software).

Reset

Figure 6.13 shows the Reset exception process.

```
T:  undefined
    Random     ← TLBENTRIES–1
    Wired      ← 0
    Config     ← 0 || EC || EP || 00000000 || BE || 110 || 010 || 1 || 1 || 0 || undefined
                 || DC || undefined[6]
    ErrorEPC     PC
    SR         ← SR31:23 || 1 || 0 || 0 || SR19:3 || 1 || SR1:0
    PC         ← 0xFFFF FFFF BFC0 0000
```

Figure 6.13 Reset Exception Processing

Cache Error

Figure 6.14 shows the Cache Error exception process.

Notes

```
T:  ErrorEPC   ← PC
    CacheErr   ← ER || EC || ED || ET || ES || EE || ED || 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
      PC       ← 0xFFFF FFFF A000 0000 + 0x100 /*Access main memory area*/
    endif
```

Figure 6.14 Cache Error Exception Processing

Soft Reset and NMI

Figure 6.15 shows the Soft Reset and NMI exception process.

```
T:  ErrorEPC   ← PC
    SR         ← SR31:23 || 1 || 0 || 1 || SR19:3 || 1 || SR1:0
    PC         ← 0xFFFF FFFF BFC0 0000
```

Figure 6.15 Soft Reset and NMI Exception Processing

General Exceptions

Figure 6.16 shows the process used for exceptions, other than Reset, Soft Reset, NMI, and Cache Error.

```
T:   Cause ¨ BD || 0 || CE || 0^12 || Cause15:8 || ExcCode || 0^2
if SR1 = 0 then   / * System is in User or Supervisor mode with no current exception */
                  EPC ¨ PC
    endif
    SR ¨ SR31:2 || 1 || SR0
    if SR22 = 1 then
      PC ¨ 0xFFFF FFFF BFC0 0200 + vector /*access to uncached space*/
    else
      PC ¨ 0xFFFF FFFF 8000 0000 + vector /*access to cached space*/
    endif
```

Figure 6.16 General Exception Processing

Exception Vector Locations

The Reset, Soft Reset, and NMI exceptions are always vectored to location 0xFFFF_FFFF_BFC0_0000. Addresses for all other exceptions are a combination of a *vector offset* and a *base address*. The vector associated with a *general exception* is called the *common exception vector;* its base address is determined by the BEV bit of the *Status* register.

Table 6.10 shows the 64-bit-mode vector base address for all exceptions; the 32-bit mode address is the low-order 32 bits (for instance, the base address for NMI in 32-bit mode is 0xBFC0 0000). Table 6.11 shows the vector offset added to the base address to create the exception address. When BEV = 0, the vector base address for the cache error exception changes from *kseg0* (0xFFFF FFFF 8000 0000) to *kseg1* (0xFFFF FFFF A000 0000). This change indicates that the caches are initialized and that the vector can be cached. 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 the TLB.

Notes

BEV Bit	RC64574/RC64575 Processor Vector Base Address
0	0xFFFF FFFF 8000 0000
1	0xFFFF FFFF BFC0 0200

Table 6.10 Exception Vector Base Addresses

Exception	RC64574/RC64575 Processor Vector Offset
TLB refill, EXL = 0	0x000
XTLB refill, EXL = 0 (X = 64-bit TLB)	0x080
Cache Error	0x100
Others	0x180
Reset, Soft Reset, NMI	none

Table 6.11 Exception Vector Offsets

Priority of Exceptions

Table 6.12 describes exceptions in the order of highest to lowest priority. While more than one exception can occur for a single instruction, only the exception with the highest priority is reported. In generally, the exceptions described in the following sections are first processed by hardware, then serviced by software.

Reset *(highest priority)*
Soft Reset
Nonmaskable Interrupt (NMI)
Address error — Instruction fetch
TLB refill — Instruction fetch
TLB invalid — Instruction fetch
Cache error — Instruction fetch
Bus error — Instruction fetch
Integer overflow, Trap, System Call, Breakpoint, Reserved Instruction, Coprocessor Unusable, or Floating-Point Exception
Address error — Data access
TLB refill — Data access
TLB invalid — Data access
TLB modified — Data write
Cache error — Data access
Bus error — Data access
Interrupt *(lowest priority)*

Table 6.12 Exception Priority Order

Notes

Causes, Hardware Processing, and Software Servicing of Exceptions

Reset Exception

Cause

The Reset exception occurs when the ColdReset* signal is asserted and then deasserted. This exception is not maskable.

Processing

The CPU provides a special interrupt vector for this exception:

- *location 0xFFFF_FFFF_BFC0_0000 in 64-bit mode.*

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:

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

See Figure 6.13 for additional information on this process.

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*

Soft Reset Exception

Cause

The Soft Reset exception occurs in response to assertion of the Reset* input signal. Execution begins at the Reset vector when the Reset* signal is negated. The Soft Reset exception is not maskable.

Processing

The Reset vector is used for this exception. The Reset vector is located within uncached and unmapped address space. Hence, the cache and TLB need not be initialized in order to process the exception. Regardless of the cause, when this exception occurs the *SR* bit of the *Status* register is set, distinguishing this exception from a Reset exception. Cache and memory states are undefined when the Soft Reset exception occurs because the Soft Reset can abort cache and bus operations.

The primary purpose of the Soft Reset exception is to reinitialize the processor after a fatal error during normal operation. Unlike an NMI, all cache and bus state machines are reset by this exception. When the Soft Reset exception occurs, all register contents are preserved with the following exceptions:

- *ErrorEPC register, which contains the restart PC.*
- *ERL, BEV, and SR bits of the Status Register, each of which is set to 1.*

See Figure 6.15 for additional information on this process.

Servicing

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

Notes

Non Maskable Interrupt (NMI) Exception

Cause

The Non Maskable Interrupt exception occurs in response to the falling edge of the NMI signal, or an external write to the Int*[6] bit of the *Interrupt* Register. The NMI interrupt is not maskable and occurs regardless of the settings of the *EXL, ERL*, and *IE* bits in the *Status* Register.

Processing

The Reset vector is used for this exception. The Reset vector is located within uncached and unmapped address space. Hence, the cache and TLB need not be initialized in order to process the exception. Regardless of the cause, when this exception occurs the *SR* bit of the *Status* register is set, distinguishing this exception from a Reset exception. Because the NMI can occur in the midst of another exception, it is typically not possible to continue program execution after servicing an NMI. An NMI exception is taken only at instruction boundaries. The state of the caches and memory system are preserved.

When the NMI exception occurs, all register contents are preserved with the following exceptions:

- *ErrorEPC register, which contains the restart PC.*
- *ERL, BEV, and SR bits of the Status Register, each of which is set to 1.*

See Figure 6.15 for additional information on this process.

Servicing

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

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*
- *load, fetch, or store a word that is not aligned on a word boundary*
- *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*

This exception is not maskable.

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 caused the exception with an instruction reference, load operation, or store operation shown by the *EPC* register and *BD* bit in the *Cause* register.

When this exception occurs, the *BadVAddr* register retains the virtual address that was not properly aligned or that 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.

Servicing

The process executing at the time is handed a segmentation violation signal. This error is usually fatal to the process incurring the exception.

TLB Exceptions

Three types of TLB exceptions can occur:

- *TLB Refill occurs when there is no TLB entry that matches an attempted reference to a mapped*

Notes

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).*

The following three sections describe these TLB exceptions.

TLB Refill Exception

Cause

The TLB refill exception occurs when there is no TLB entry to match a reference to a mapped address space. This exception is not maskable.

Processing

There are two special exception vectors for this exception; one for references to 32-bit address spaces, and one for references to 64-bit 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 contains 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.

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 E*ntryLo0/EntryLo1* register; the *EntryHi* and *EntryLo* registers are written into the TLB.

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.

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). This exception is not maskable.

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* register is undefined.

Notes

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.

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)*

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.

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. This exception is not maskable.

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* register is 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.

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.

Cache Error Exception

Cause

The Cache Error exception occurs when either a primary or secondary cache parity error is detected. This exception is maskable by the *DE* bit in the Status Register.

Processing

The processor sets the *ERL* bit in the *Status* register, saves the exception restart address in the *ErrorEPC* register, and then transfers the information to one of the following special vectors in uncached space:

- *If BEV = 0, the vector is 0xFFFF FFFF A000 0100.*
- *If BEV = 0, the vector is 0xFFFF FFFF BFC0 0300.*

See Figure 6.14 for additional information on this process.

Servicing

Notes

All errors should be logged. To correct parity errors the system uses the CACHE instruction to invalidate the cache block, overwrite 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.

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. This exception is not maskable. A Bus Error exception occurs when a cache-miss refill, uncached reference, or an 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.

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.

Integer Overflow Exception

Cause

An Integer Overflow exception occurs when an ADD, ADDI, SUB, DADD, DADDI or DSUB instruction results in a 2's complement overflow. This exception is not maskable.

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.

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.

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. This exception is not maskable.

Notes

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.

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.

System Call Exception

Cause

A System Call exception occurs during an attempt to execute the SYSCALL instruction. This exception is not maskable.

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.

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.

Breakpoint Exception

Cause

A Breakpoint exception occurs when an attempt is made to execute the BREAK instruction. This exception is not maskable.

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.

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.

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).*

Notes

- *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).*
- *an attempt is made to execute 64-bit operations in 32-bit mode 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. This exception is not maskable.

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.

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, or*
- *CP0 instructions, when the unit has not been marked usable and the process executes in either User or Supervisor mode.*

This exception is not maskable.

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.

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.*

Floating-Point Exception

Cause

The Floating-Point exception is used by the floating-point coprocessor. This exception is not maskable.

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.

Notes

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.

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. 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.

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.

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.

Due to the on-chip write buffer, a store to an external device may not occur until after other instructions in the pipeline finish. Hence, 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 is no actual interrupt pending.

Exception Handling and Servicing Flowcharts

The remainder of this section contains flowcharts (Figures 6.17 through 6.22) for the following exceptions and guidelines for their handlers:

- *general exceptions and their exception handler*
- *TLB/XTLB miss exception and their exception handler*
- *cache error exception and its handler*
- *reset, soft reset and NMI exceptions, and a guideline to their handler.*

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

Notes

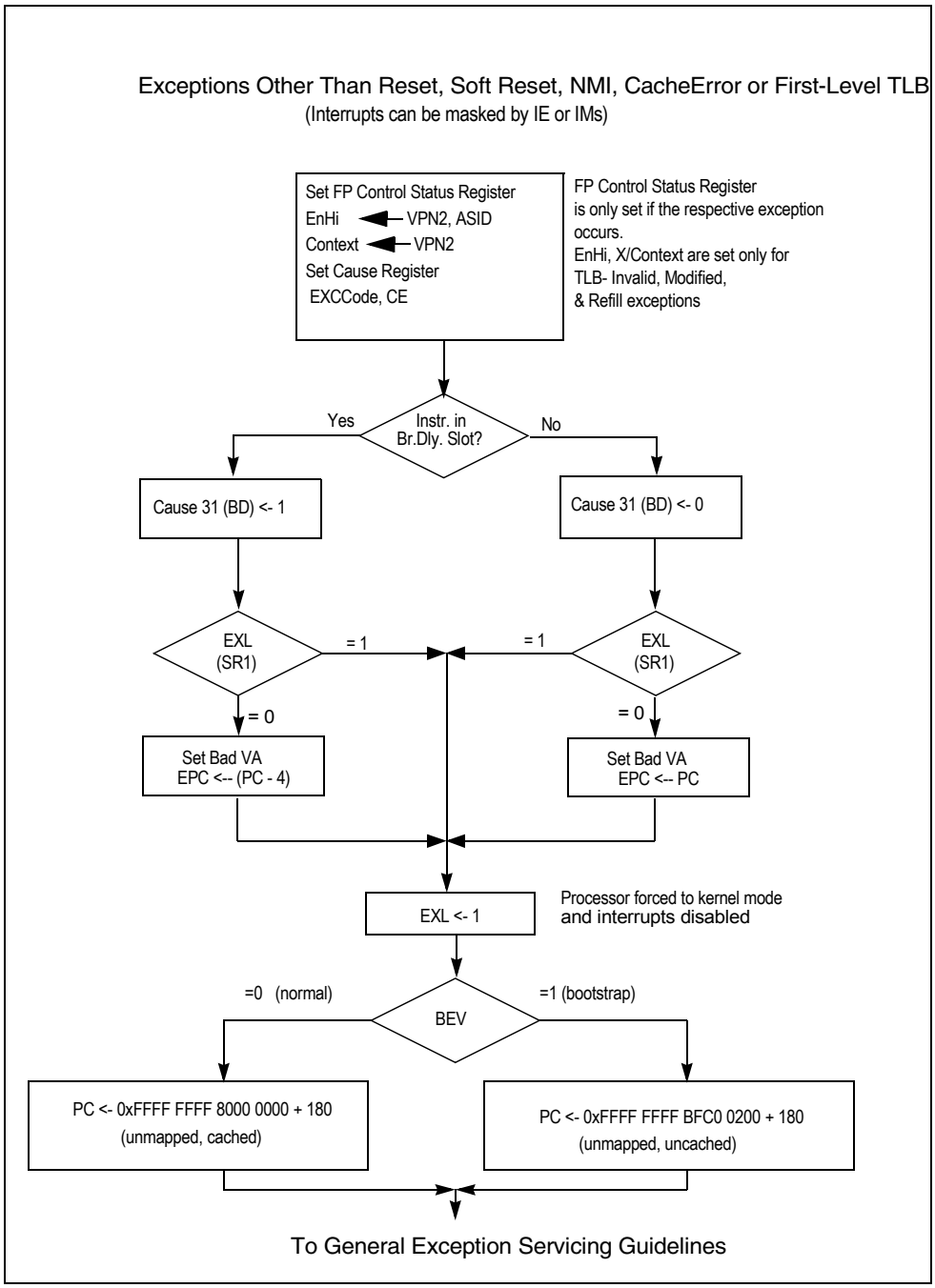


Figure 6.17 General Exception Handler (HW)

Notes

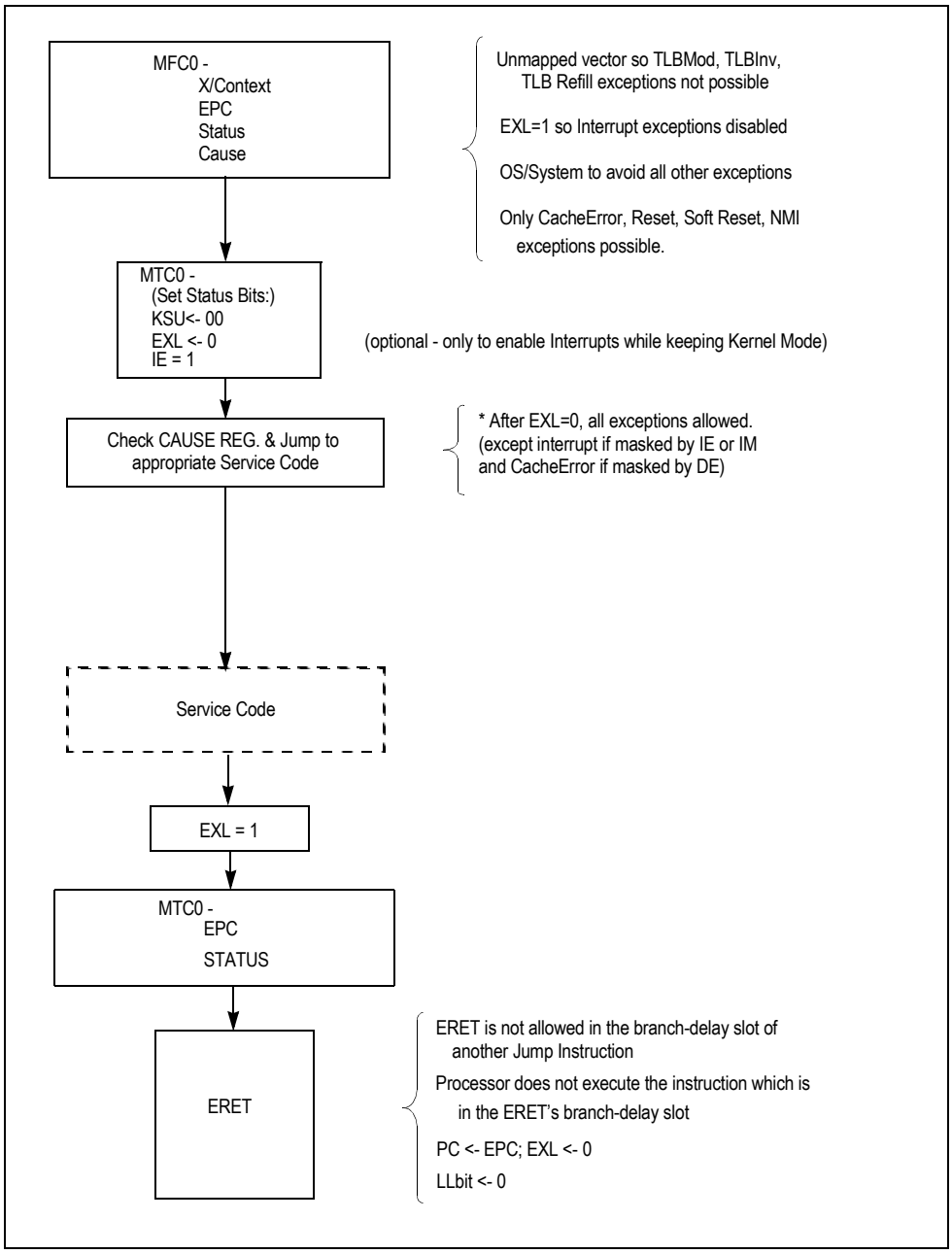


Figure 6.18 General Exception Servicing Guidelines (SW)

Notes

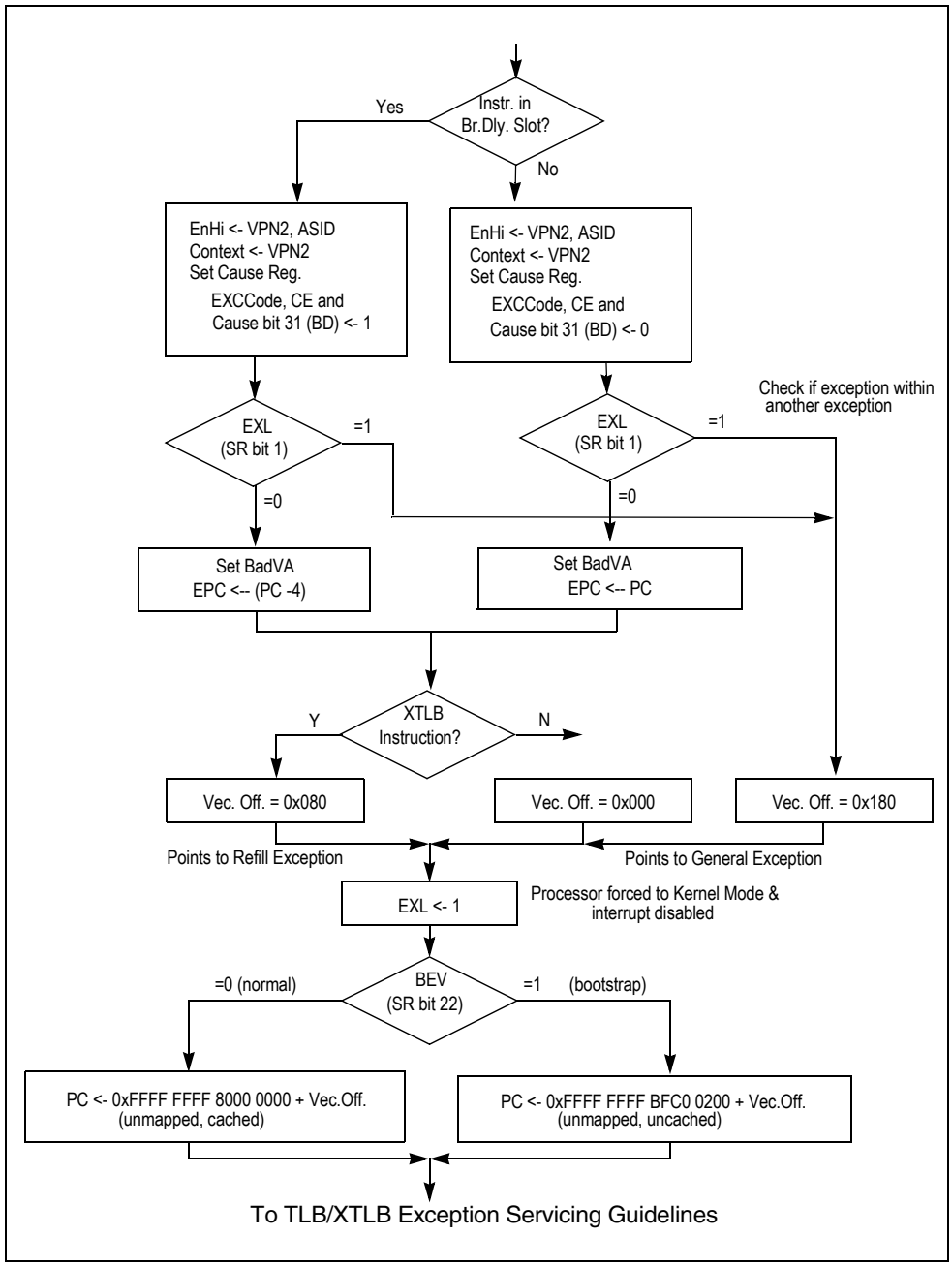


Figure 6.19 TLB/XTLB Miss Exception Handler (HW)

Notes

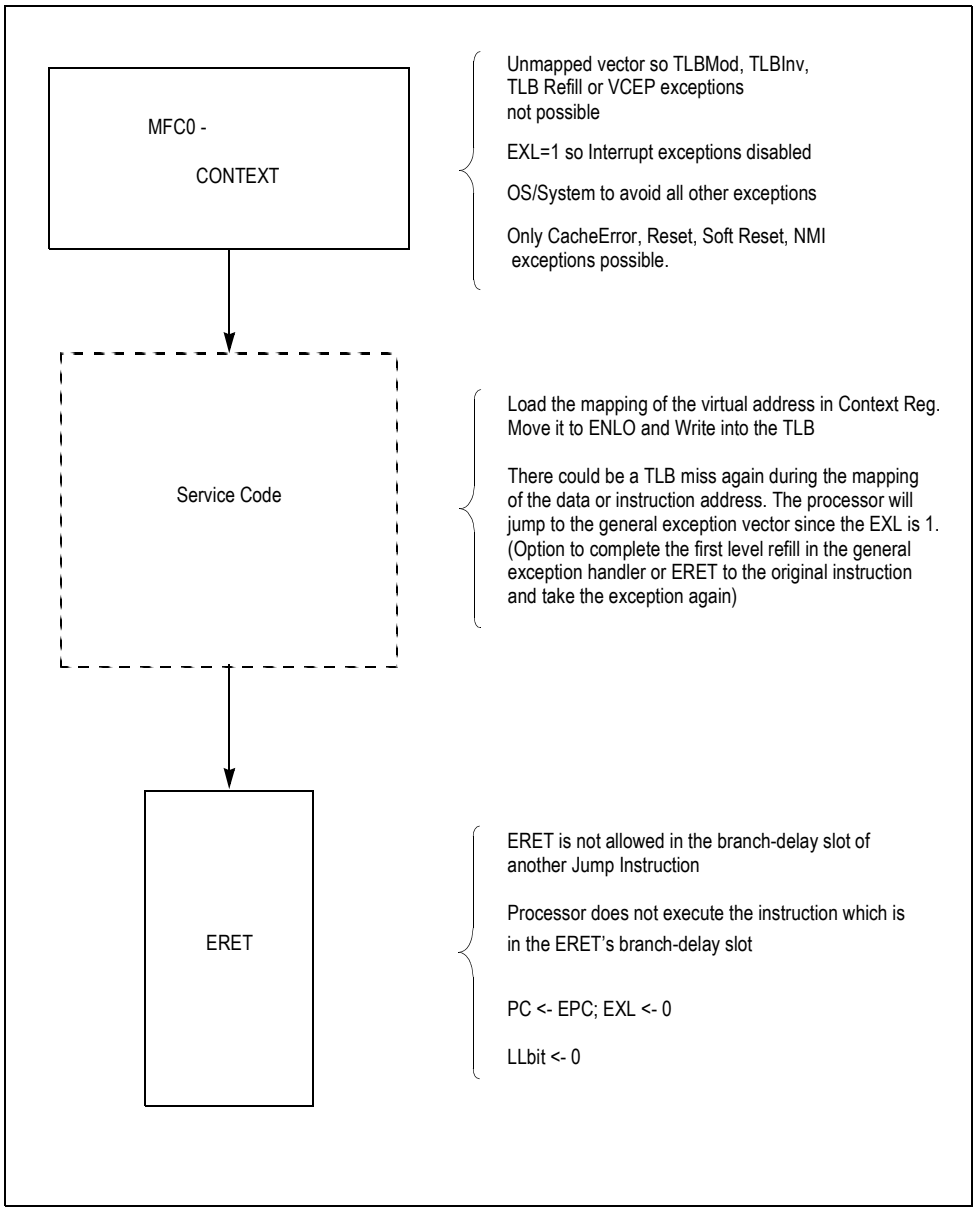


Figure 6.20 TLB/XTLB Exception Servicing Guidelines (SW)

Notes

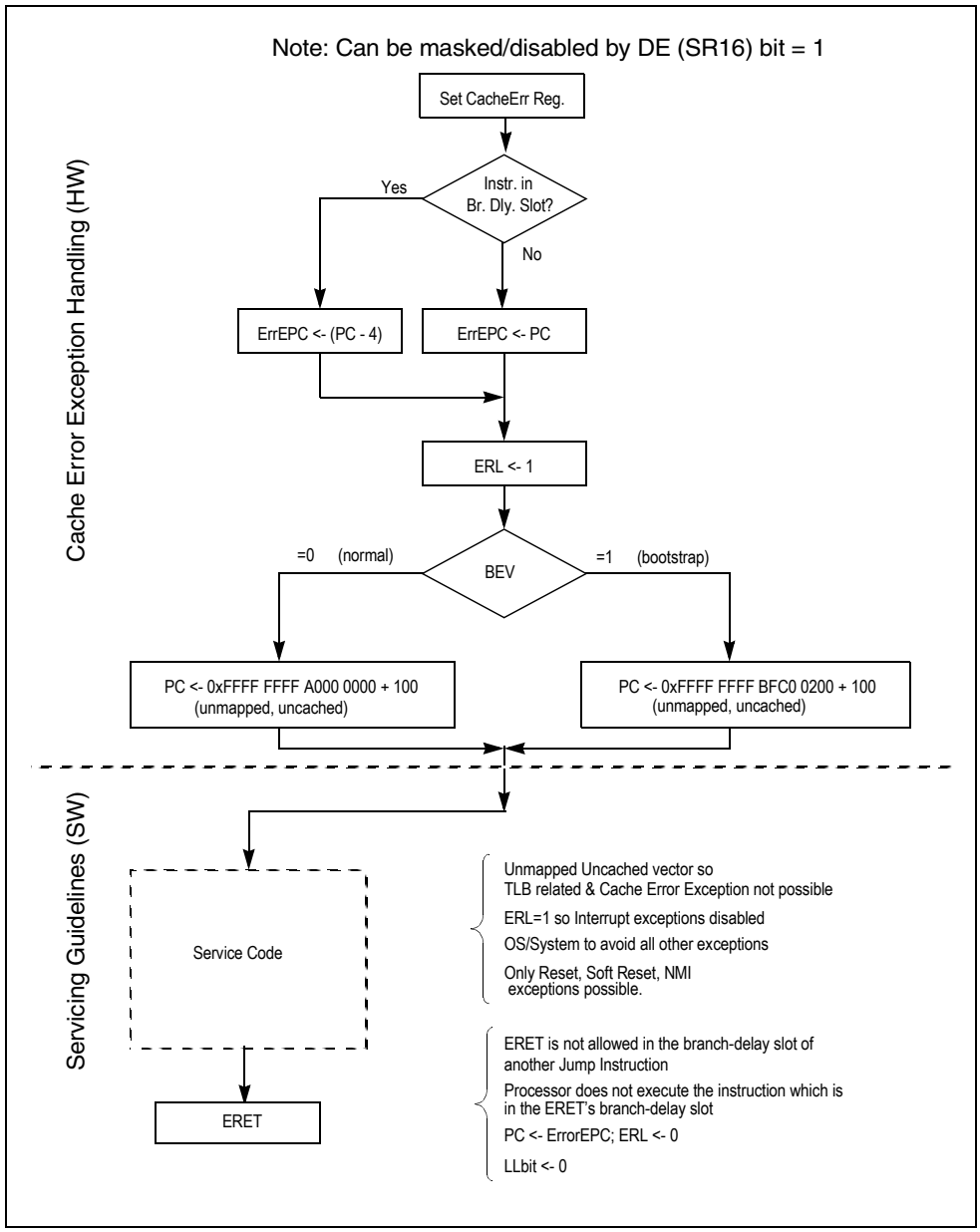


Figure 6.21 Cache Error Exception Handling (HW) and Servicing Guidelines

Notes

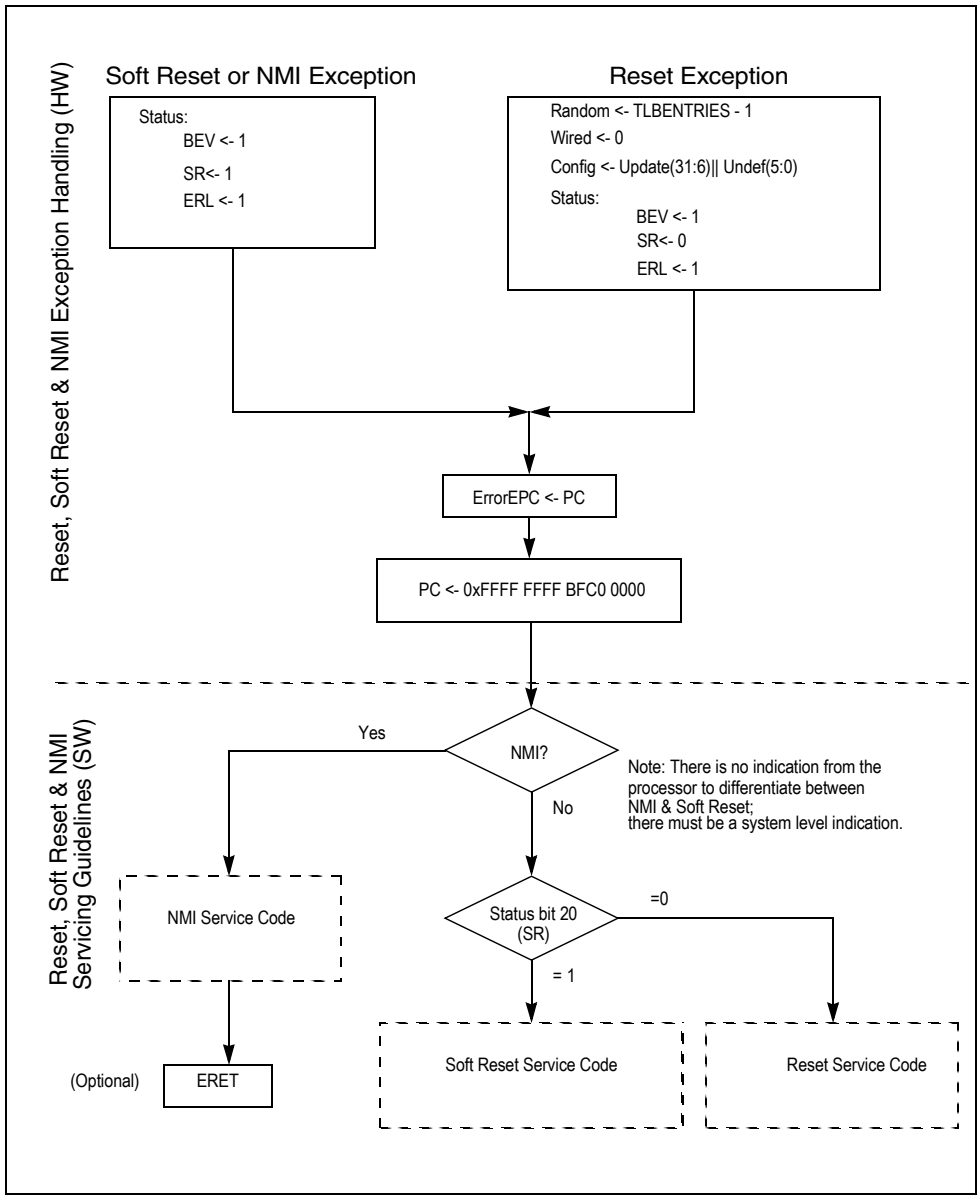


Figure 6.22 Reset, Soft Reset & NMI Exception Handling

Notes

Floating-Point Unit (FPU)

Notes

Introduction

The RC64574/RC64575 floating-point unit (FPU) operational functions consist of an adder, multiplier, and divider. The FPU, with associated system software, conforms fully to the ANSI/IEEE Standard 754–1985, *IEEE Standard for Binary Floating-Point Arithmetic*. In addition, the MIPS architecture fully supports the recommendations of the standard for precise exceptions.

The FPU operates as a coprocessor for the CPU (it is assigned coprocessor label *CP1*), and extends the CPU instruction set to perform operations on floating-point values. It has the following basic features:

- ***32-bit or 64-bit Operation***. *The FR bit in the CPU Status register controls the selection of 32-bit 64-bit mode. Each register can hold single- or double-precision values.*
- ***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.*
- ***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. Since each unit receives and executes instructions in parallel, some floating-point instructions can execute at the same single-cycle-per-instruction rate as fixed-point instructions.*

Figure 7.1 illustrates the functional organization of the FPU.

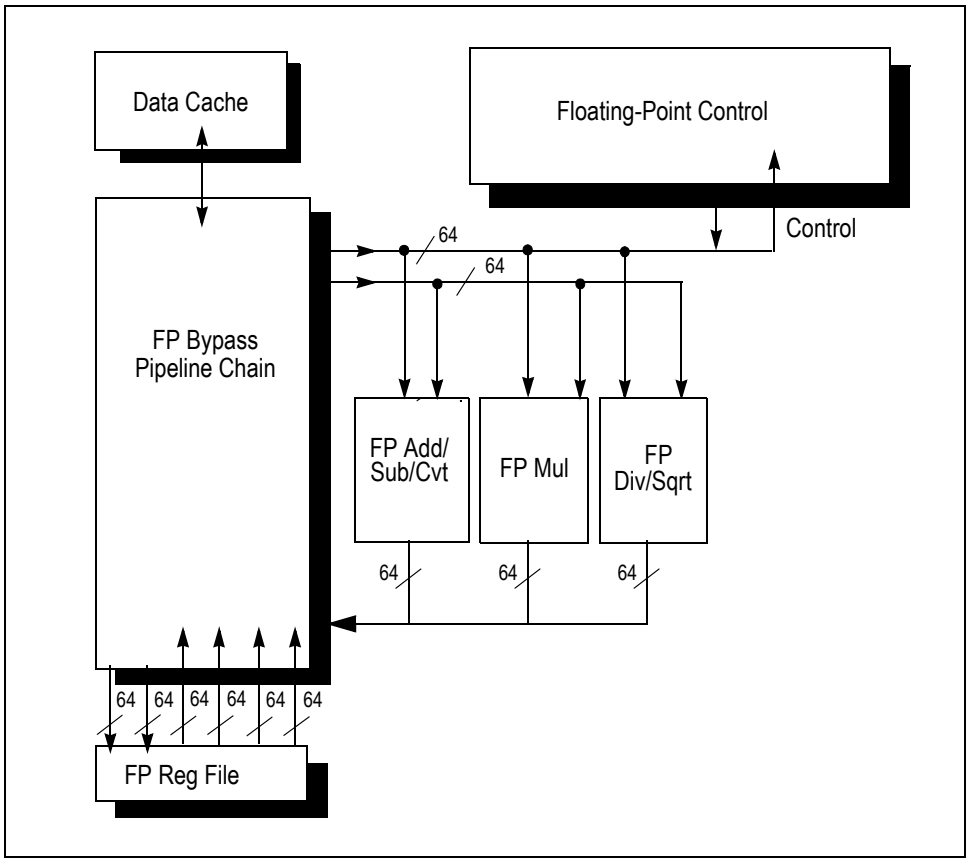


Figure 7.1 FPU Functional Block Diagram

Notes

Floating-Point General Registers (FGRs)

The FPU has a set of *Floating-Point General Registers (FGR*s) that 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.*
- *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.*

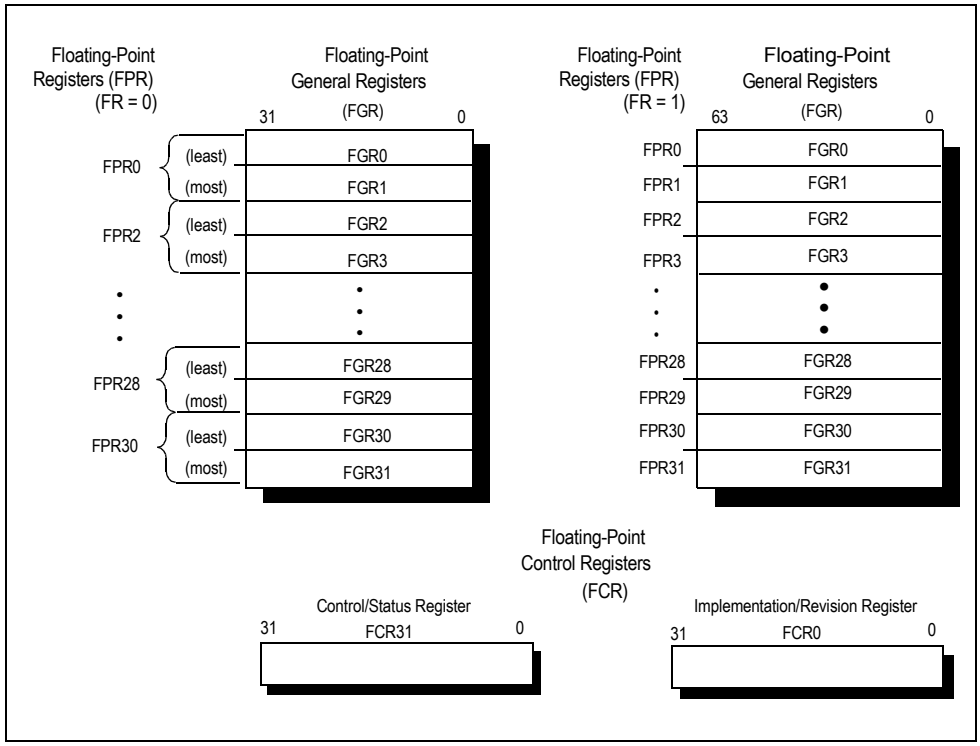


Figure 7.2 FPU Registers

Floating-Point Registers (FPRs)

The FPRs are shown in Figure 7.2. These 64-bit registers hold floating-point values during floating-point operations and are physically formed from the *Floating-Point General Registers (FGRs*).

When the *FR* bit is set to a 1, the *FPR* references a single 64-bit *FGR* and all *FPR* register numbers are valid. 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. 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*.

Notes

Floating-Point Control Registers (FCRs)

Table 7.1 lists the floating-point control registers (*FCR*s). These can only be accessed by Move operations. The *FCR*s include:

FCR Number	Use
FCR0	Implementation/Revision register: holds revision information about the FPU.
FCR1 to FCR30	Reserved
FCR31	Control/Status register: controls and monitors exceptions, holds the result of compare operations, and establishes rounding modes.

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 describes the register fields.

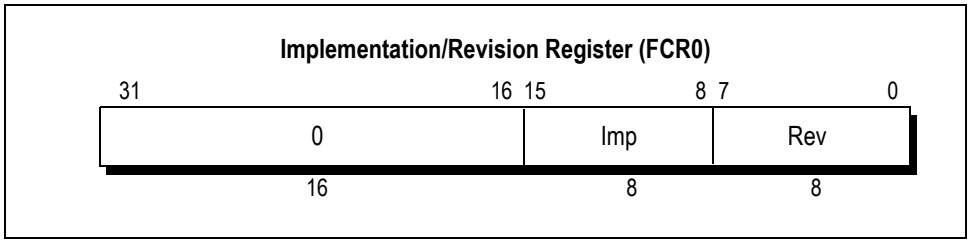


Figure 7.3 Implementation/Revision Register

Field	Description
Imp	Implementation number (0x23)
Rev	Revision number in the form of *y.x*
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 7.2 FCRO 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, IDT does not guarantee that changes to its chips 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 shows the format of the *Control/Status* register, and Table 7.3 describes the register fields.

Notes

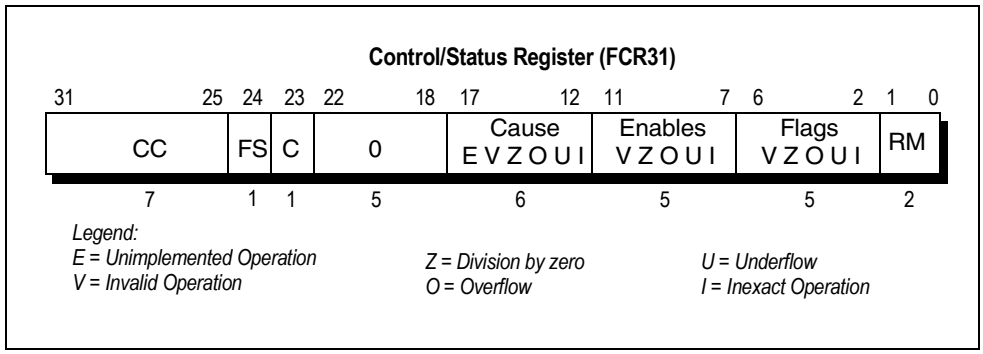


Figure 7.4 FP Control/Status Register Bit Assignments

Field	Description
CC	Condition code.
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 description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
Enables	Enable bits. See description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
Flags	Flag bits. See description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
RM	Rounding mode bits. See description of *Control/Status* register *Rounding Mode Control* bits.

Table 7.3 Control/Status Register Fields

Accessing the Control/Status Register

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. *FCR31* must only be written to when the FPU is not actively executing floating-point operations; this can be ensured by reading the contents of the register to empty the pipeline.

IEEE Standard 754

IEEE Standard 754 specifies that floating-point operations detect certain exceptional cases, raise flags, and 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.

Notes

Control/Status Register Cause, Flag, and Enable Fields

Figure 7.5 illustrates the *Cause*, *Flag*, and *Enable* fields of the *Control/Status* register.

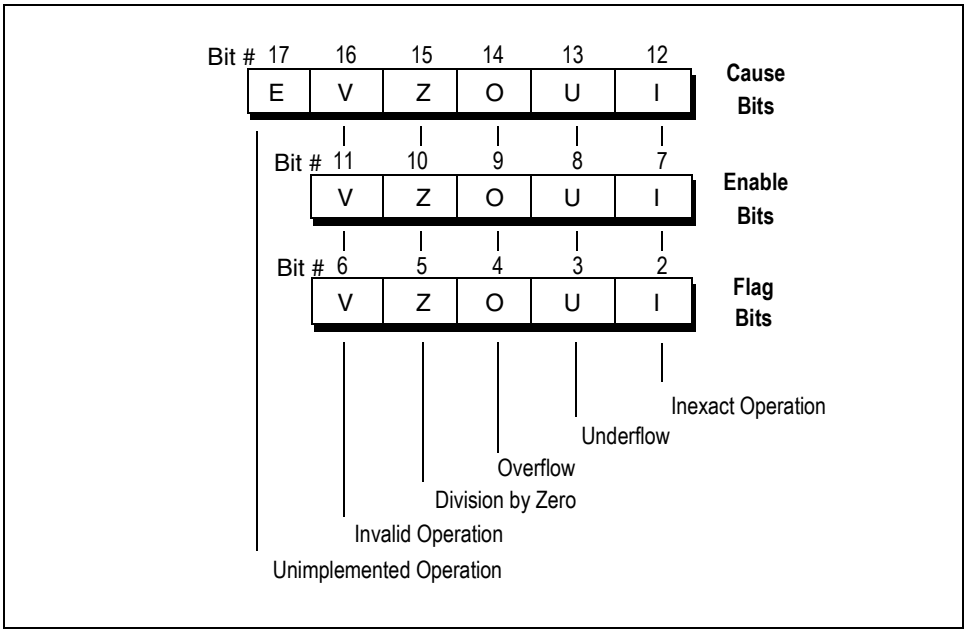


Figure 7.5 Control/Status Register Cause, Flag, and Enable Fields

Cause Bits

Bits 17:12 in the *Control/Status* register contain *Cause* bits, as shown in Figure 7.5. The *Cause* bits reflect the results of the most recently executed instruction. The 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 1 if software emulation is required, otherwise it remains 0. The other bits are set to 1 or cleared to 0 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* bit.

Enable Bits

A floating-point exception is generated any time a *Cause* bit and the corresponding *Enable* bit are set. A floating-point operation that sets an enabled *Cause* bit forces an immediate exception, as does setting both *Cause* and *Enable* bits with CTC1.

There is no enable for Unimplemented Operation (*E*). Setting Unimplemented Operation always generates a floating-point exception.

Before returning from a floating-point exception, software must first clear the enabled *Cause* bits with a CTC1 instruction 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.

Notes

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.

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

FPU Data Formats

The FPU supports both floating-point and fixed-point data formats. The floating-point formats can be single-precision binary or double-precision binary. The fixed-point formats can be 32- or 64-bit binary.

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.

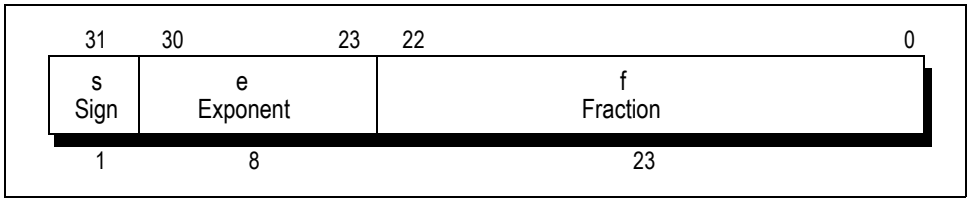


Figure 7.6 Single-Precision Floating-Point Format

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.

Notes

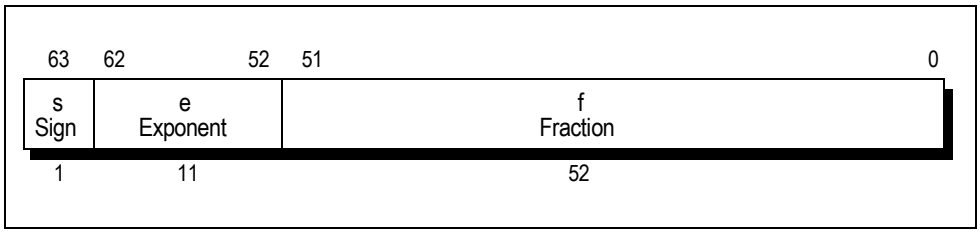


Figure 7.7 Double-Precision Floating-Point Format

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 non-zero 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 Table 7.5.

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* = $(-1)^s2^E(1.f)$
(4)	if E = E_{min}–1 and f ¼ 0, then *v* = $(-1)^s2^{Emin}(0.f)$
(5)	if E = E_{min}–1 and f = 0, then *v* = $(-1)^s0$

Table 7.5 Calculating Values in Single and Double-Precision Formats

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 7.6 defines the values for the format parameters; minimum and maximum floating-point values are given in Table 7.7.

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

Table 7.6 Floating-Point Format Parameter Values

Notes

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 two'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 lists the binary fixed-point format fields.

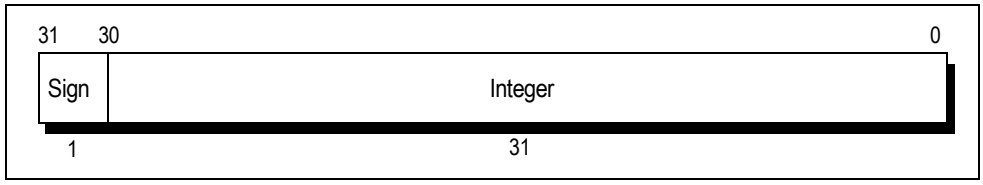


Figure 7.8 Binary Fixed-Point Format

Field	Description
sign	sign bit
integer	integer value

Table 7.8 Binary Fixed-Point Format Fields

Floating-Point Instruction Set Summary

All FPU instructions are 32 bits long, aligned on a word boundary. They 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.*

In the instruction formats shown in Table 7.9 through 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, *W* specifies 32-bit binary fixed-point, and *L* specifies 64-bit (long) binary fixed-point.

Notes

OpCode	Description
LWC1	Load Word to FPU
LWXC1	Load Word Indexed to FPU
SWC1	Store Word from FPU
SWXC1	Store Word Indexed from FPU
LDC1	Load Doubleword to FPU
LDXC1	Load Doubleword Indexed to FPU
SDC1	Store Doubleword From FPU
SDXC1	Store Doubleword Indexed 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
PREF	Prefetch - Register + Offset
PREFX	Prefetch Indexed - Register + Register

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 32-bit Fixed Point
CVT.L.fmt	Floating-Point Convert to 64-bit Fixed Point
ROUND.W.fmt	Floating-Point Round to 32-bit Fixed Point
ROUND.L.fmt	Floating-Point Round to 64-bit Fixed Point
TRUNC.W.fmt	Floating-Point Truncate to 32-bit Fixed Point
TRUNC.L.fmt	Floating-Point Truncate to 64-bit Fixed Point
CEIL.W.fmt	Floating-Point Ceiling to 32-bit Fixed Point
CEIL.L.fmt	Floating-Point Ceiling to 64-bit Fixed Point
FLOOR.W.fmt	Floating-Point Floor to 32-bit Fixed Point
FLOOR.L.fmt	Floating-Point Floor to 64-bit Fixed Point

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
RECIP	Floating-Point Reciprocal
RSQRT	Floating-Point Reciprocal 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

Floating-Point 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.

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 From 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.

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).*

Notes

Like the floating-point load and store operations, these operations perform no format conversions and never cause floating-point exceptions.

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.

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, fixed- or floating-point formats.

Floating-Point Computational Instructions

Computational instructions perform arithmetic operations on floating-point values, in registers. There are two categories of computational instructions:

- *3-operand register-type instructions, which perform floating-point addition, subtraction, multiplication, and division.*
- *2-operand register-type instructions, which perform floating-point absolute value, move, negate, and square-root operations.*

For a detailed description of each instruction, refer to the IDT MIPS Microprocessor Family Software Manual.

Branch on FPU Condition Instructions

The Branch on FPU (coprocessor unit 1) condition instructions that can test the result of the FPU compare (C.cond) instructions. For a detailed description of each instruction, refer to the IDT MIPS Microprocessor Family Software Manual.

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 7.13 lists the mnemonics for the compare-instruction conditions.

Notes

Mnemonic	Definition	Mnemonic	Definition
T	True	**F**	False
OR	Ordered	**UN**	Unordered
NEQ	Not Equal	**EQ**	Equal
OLG	Ordered or Less Than or Greater Than	**UEQ**	Unordered or Equal
UGE	Unordered or Greater Than or Equal	**OLT**	Ordered Less Than
OGE	Ordered Greater Than	**ULT**	Unordered or Less Than
UGT	Unordered or Greater Than	**OLE**	Ordered Less Than or Equal
OGT	Ordered Greater Than	**ULE**	Unordered or Less Than or Equal
ST	Signaling True	**SF**	Signaling False
GLE	Greater Than, or Less Than or Equal	**NGLE**	Not Greater Than or Less Than or Equal
SNE	Signaling Not Equal	**SEQ**	Signaling Equal
GL	Greater Than or Less Than	**NGL**	Not Greater Than or Less Than
NLT	Not Less Than	**LT**	Less Than
GE	Greater Than or Equal	**NGE**	Not Greater Than or Equal
NLE	Not Less Than or Equal	**LE**	Less Than or Equal
GT	Greater Than	**NGT**	Not Greater Than

Table 7.13 Mnemonics of Compare-Instruction Conditions

MIPS IV Instruction Set Additions to FPU Instructions

The following additions to MIPS III FPU instruction set are included in the MIPS IV FPU instruction set.

Indexed Floating-Point Load

- *LWXC1 - Load word indexed to Coprocessor 1.*
- *LDXC1 - Load doubleword indexed to Coprocessor 1.*

The two Indexed Floating-Point Load instructions transfer floating-point data types from memory to the floating-point registers using register + register addressing mode. The contents of the general register specified by the base is added to the contents of the general register specified by the index to form a virtual address. The contents of the word or doubleword specified by the effective address are loaded into the floating-point register specified in the instruction. There are no indexed loads to general registers.

The region bits (63:62) of the effective address must be supplied by the base. If the addition alters these bits an address exception occurs. Also, if the address is not aligned, an address exception occurs.

Indexed Floating-Point Store

- *SWXC1 - Store word indexed to Coprocessor 1.*
- *SDXC1 - Store doubleword indexed to Coprocessor 1.*

The two Indexed Floating-Point Store instructions transfer floating-point data types from the floating-point registers to memory using register + register addressing mode. The contents of the general register specified by the base is added to the contents of the general register specified by the index to form a virtual address. The contents of the floating-point register specified in the instruction is stored to the memory location specified by the effective address.

The region bits (63:62) of the effective address must be supplied by the base. If the addition alters these bits an address exception occurs. Also, if the address is not aligned, an address exception occurs.

Notes

Branch on Floating-Point Coprocessor

- *BC1T - Branch on FP Condition True*
- *BC1F - Branch on FP Condition False*
- *BC1TL - Branch on FP Condition True Likely*
- *BC1FL - Branch on FP Condition False Likely*

The four Branch on Floating-Point Coprocessor instructions are extensions of the branch instructions in various prior MIPS instruction sets, with which they are upward-compatible. The BC1T and BC1F instructions are extensions of MIPS I. BC1TL and BC1FL are extensions of MIPS III. These instructions test one of eight floating-point condition codes. If no condition code is specified, condition code bit zero is selected. This encoding is downward-compatible with previous MIPS architectures.

The branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended to 64 bits. If the contents of the floating-point condition code specified in the instruction are equal to the test value, the target address is branched to with a delay of one instruction. If the conditional branch is not taken and the nullify delay bit in the instruction is set, the instruction in the branch delay slot is nullified.

Floating-Point Multiply-Add/Subtract

- *MADD - Floating-Point Multiply-Add*
- *MSUB - Floating-Point Multiply-Subtract*
- *NMADD - Floating-Point Negative Multiply-Add*
- *NMSUB - Floating-Point Negative Multiply-Subtract*

The four Floating-Point Multiply-Add/Subtract instructions compute two floating-point operations with one instruction. Each of the instructions performs intermediate rounding.

Floating-Point Compare

- *C.cond - Compare*
- *C.cond - Implies cc=0*

The two Floating-Point Compare instructions are upward-compatible extensions of the floating-point compare instructions of the MIPS I instruction set and produce a boolean result which is stored in one of the condition codes.

The contents of the two FP source registers specified in the instruction are interpreted and arithmetically compared. A result is determined based on the comparison and the conditions specified in the instruction. If one of the values is not a number and the high order bit of the condition field is set, an invalid operations trap occurs. Comparisons are exact and neither overflow or underflow.

The implication for compiler code scheduling is that a compare instruction may be immediately followed by a dependent floating-point conditional move instruction, but may not be immediately followed by a dependent branch on floating-point coprocessor condition instruction or a dependent integer conditional move instruction. This restriction applies only to the condition code specified in the 3-bit condition code specifier of the instruction. All other condition codes are unaffected.

Floating-Point Conditional Moves

- *MOVT.fmt - Floating-Point Conditional Move on condition code true*
- *MOVF.fmt - Floating-Point Conditional Move on condition code false*
- *MOVN.fmt - Floating-Point Conditional Move on register not equal to zero*
- *MOVZ.fmt - Floating-Point Conditional Move on register equal to zero*

The four Floating-Point Conditional Move instructions are used to test a condition code or a general register and then conditionally perform a floating-point move. The value of the floating-point condition code specified by the 3-bit condition code specifier, or the value of the register indicated by the 5-bit general register specifier, is compared to zero. If the result indicates that the move should be performed, the

Notes

contents of the specified source register is copied into the specified destination register. All of these conditional floating-point move operations are non-arithmetic. Consequently, no IEEE-754 exceptions occur as a result of these instructions.

Reciprocals

- *RECIP.fmt - Reciprocal Approximation*
- *RSQRT.fmt - Reciprocal Square Root Approximation*

The Reciprocal Approximation instruction performs a reciprocal approximation on a floating-point value. The reciprocal of the value in the floating-point source register is approximated and placed in a destination register. The numerical accuracy of this operation is implementation-dependent, based on the rounding mode used.

The Reciprocal Square Root Approximation instruction performs a reciprocal square root approximation on a floating-point value. The reciprocal of the positive square root of a value in the floating-point source register is approximated and placed in a destination register. The numerical accuracy of this operation is implementation-dependent, based on the rounding mode used.

The approximation is due to the fact that neither of these instructions meets IEEE accuracy requirements. In both cases a small amount of precision has been sacrificed, thereby significantly reducing execution time. For example, in the case of a RECIP instruction, X/Y is computed by taking the reciprocal of Y and multiplying that result by X. The reduced execution time of the reciprocal operation allows a RECIP followed by a MUL (multiply) instruction to be executed faster than a single DIV (divide) instruction. The performance difference between a RSQRT instruction and a SQRT followed by a DIV instruction is implementation-dependent.

On the RC64574/RC64575, the RECIP instruction has the same latency as a DIV instruction, but a RSQRT is faster than a SQRT followed by a RECIP.

Notes

FPU-Instruction Latencies

Table 7.14 shows the execution-stage latencies and repeat throughput for FPU instructions. The values assume the result of the operation is immediately used in a succeeding operation.

Instruction Group	Latency	Repeat
Absolute	1	1
Add	4	1
BC1T	1	1
BC1F	1	1
BC1TL	1	1
BC1FL	1	1
CEIL.w	4	1
CEIL.l	4	1
CFC1	2	1
Compare	1	1
CTC1	3	3
CVT.s.d	4	1
CVT.s.w	6	3
CVT.s.l[2]	6	3
CVT.d.s	4	1
CVT.d.w	4	1
CVT.d.l[2]	4	1
CVT.w.s	4	1
CVT.w.d	4	1
CVT.l.s	4	1
CVT.l.d	4	1
DIV.s	21	19
DIV.d	36	34
DMFC1	2	1
DMTC1	2	1
FLOOR.w	4	1
FLOOR.l	4	1
LDC1	2	2
Load	2	1
Load Indexed	3	2
LWC1	1	1

1 Trap on greater than 53 bits of significance
2 Trap on greater than 52 bits of significance.

Table 7.14 Floating-Point Instruction Latencies (Part 1 of 2)

Notes

Instruction Group	Latency	Repeat
MADD.s	4	1
MADD.d	5	2
MFC1	2	1
Move	1	1
Move Conditional	1	1
MSUB.s	4	1
MSUB.d	5	2
MTC1	2	1
MUL.s	4	1
MUL.d	5	2
Negative	1	1
NMADD.s	4	1
NMADD.d	5	2
NMSUB.s	4	1
NMSUB.d	5	2
Prefetch	N/A	1
Prefetch Indexed	N/A	2
RECIP.s	21	19
RECIP.d	36	34
ROUND.w	4	1
ROUND.l[1]	4	1
RSQRT.s	38	36
RSQRT.d	68	66
SDC1	1	1
SQRT.s	21	19
SQRT.d	36	34
Store	N/A	1
Store Indexed	N/A	2
Subtract	4	1
SWC1	1	1
TRUNC.w	4	1
TRUNC.l	4	1

1 Trap on greater than 53 bits of significance
2 Trap on greater than 52 bits of significance.

Table 7.14 Floating-Point Instruction Latencies (Part 2 of 2)

Notes

Floating Point Architecture/Performance

Note that the MIPS-IV ISA specifies certain compound floating point operations, such as multiply-add, multiply-subtract, and reciprocal-square root. In the RC64574/RC64575, the multiply-add operation uses an intermediate round to support full IEEE compliance.

Note that the PREF operations of the RC64574/RC64575 are implemented as NOPs, as allowed by the MIPS-IV ISA. Refer to chapter 3 of 79RV5000 Reference Manual for more details.

Floating point performance is specified in Table 7.15.

Operation	.S		.D		.W		.L		Other.	
	Latency	Repeat	Latency	Repeat	Latency	Repeat	Latency	Repeat	Latency	Repeat
LWC1/LDC1									2	1
LWXC1/LDXC1									2	1
PREFX									0	2
SWC1/SDC1									2	1
SWXC1/SDXC1									3	2
MTC1/DMTC1									2	1
MFC1/DMFC1									2	1
CTC1									3	1
CFC1									2	1
BC1									2	2
ABS/NEG	1	1	1	1						
C.cond	2	1	2	1						
MOV-	1	1	1	1						
MADD/MSUB	4	1	5	2						
ADD/SUB	4	1	4	1						
MUL	4	1	5	2						
ROUND.W/ TRUNC.W	4	1	4	1						
ROUND.L/ TRUNC.L	4	1	4	1						
CEIL.W/ FLOOR.W	4	1	4	1						
CEIL.L/ FLOOR.L	4	1	4	1						
CVT.S			4	1	6	3	6	3		
CVT.D	4	1			4	1	4	1		
CVT.W	4	1	4	1						
CVT.L	4	1	4	1						
DIV	15	15	30	30						
SQRT	21	19	36	34						
RECIP	21	19	36	34						
RCPSQRT	38	36	68	66						

Table Notes:
Round.L, Trunc.L, Ceil.L, Floor.L each trap on greater than 52 bits of significance.
CVT.D.L traps on greater than 53 bits of significance.

Table 7.15 RC64574/RC64575 Floating Point Unit Execution Rate

Notes

Floating-Point (FPU) Exceptions

Notes

Introduction

A floating-point exception occurs whenever the FPU cannot handle either the operands or the results of a floating-point operation in its normal way. The FPU responds by generating an exception to initiate a software trap or by setting a status flag.

Exception Types

The FP *Control/Status* register described in Chapter 3 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 no trap is taken, an appropriate value is written into the FPU destination register and execution continues.*

The FPU supports the five IEEE Standard 754 exceptions:

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

Cause bits, *Enables*, and *Flag* bits (status flags) are used. The FPU adds a sixth exception type, *Unimplemented Operation (E)*, to use when the FPU cannot implement the standard MIPS floating-point architecture, including cases in which the FPU cannot determine the correct exception behavior. 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 (if the FPU interrupt input to the CPU is enabled).

Figure 8.1 illustrates the Control/Status register bits that support exceptions.

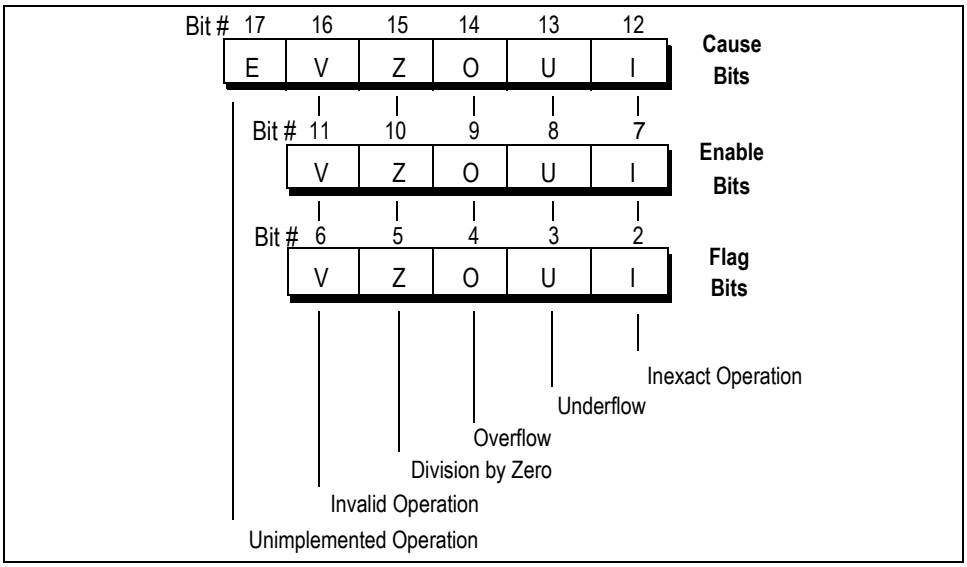


Figure 8.1 Control/Status Register Exception/Flag/Trap/Enable Bits

Notes

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, the corresponding *Cause* bit is set. If the corresponding *Enable* bit is not set, the *Flag* bit is also set. If the corresponding *Enable* bit is set, the *Flag* bit is not set and the FPU generates an interrupt to the CPU. Subsequent exception processing allows a trap to be taken.

Exception Trap Processing

When a floating-point exception trap is taken, the *Cause* register indicates the floating-point coprocessor 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. These bits are, in effect, an extension of the system coprocessor *Cause* register.

Trap Handlers for IEEE Standard 754 Exceptions

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.

The IEEE Standard 754 recommends that, if enabled, the overflow and underflow traps take precedence over a separate inexact trap. This prioritization is accomplished in software; hardware sets the bits for both.

Flags

A *Flag* bit is provided for each IEEE exception. This *Flag* bit is set to 1 on the assertion of its corresponding exception, with no corresponding exception trap signaled. The *Flag* bit is reset (cleared to 0) 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, 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.

Notes

Field	Description	Rounding Mode	Default action
I	Inexact exception	Any	Supply a rounded result
U	Underflow exception	RN	Modify underflow values to 0 with the sign of the intermediate result
		RZ	Modify underflow values to 0 with the sign of the intermediate result
		RP	Modify positive underflows to the format's smallest positive finite number; modify negative underflows to -0
		RM	Modify negative underflows to the format's smallest negative finite number; modify positive underflows to 0
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 Exceptions Actions

Table 8.2 lists the exception-causing situations and contrasts the behavior of the FPU with the requirements 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[1]	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	
Invalid operation	V	V	V	0/0, etc.
Exponent underflow	U	E	E	Normalized exponent < E_{min}
Denormalized or QNaN	None	E	E	Denormalized is (exponent = E_{min}-1 and mantissa <> 0)

Table 8.2 FPU Exception-Causing Conditions

1. The IEEE Standard 754 specifies an inexact exception on overflow only if the overflow trap is disabled.

Notes

FPU Exceptions

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

Inexact Exception (I)

The FPU generates the Inexact exception if one of the following occurs:

- *the rounded result of an operation is not exact.*
- *the rounded result of an operation overflows.*
- *the rounded result of an operation underflows and both the Underflow and Inexact Enable bits are not set and the FS bit is set.*

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 one cycle. Since this mode of execution can impact performance, Inexact exception traps should be enabled only when necessary.

The enabling of Inexact exception traps have the following results:

- *Trap Enabled: The result register is not modified and the source registers are preserved.*
- *Trap Disabled: 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.*
- *Comparison or a Convert From Floating-point Operation on a signaling NaN.*
- *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).

The enabling of traps have the following results:

- *Trap Enabled: The original operand values are undisturbed.*
- *Trap Disabled: A quiet NaN is delivered to the destination register if no other software trap occurs.*

Notes

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(p/2), csc(0), or 0–1.

The enabling of traps have the following results:

- *Trap Enabled: The result register is not modified, and the source registers are preserved.*
- *Trap Disabled: 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.)

The enabling of traps have the following results:

- *Trap Enabled: The result register is not modified, and the source registers are preserved.*
- *Trap Disabled: 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. Tininess 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.

The enabling of traps have the following results:

- *Trap Enabled: If Underflow or Inexact traps are enabled, or if the FS bit is not set, then an Unimplemented exception (E) is generated, and the result register is not modified.*
- *Trap Disabled: If Underflow and Inexact traps are not enabled and the FS bit is set, the result is determined by the rounding mode and the sign of the intermediate result (as listed in Table 8.1).*

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.

Notes

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, except for Compare instruction*
- *Quiet Not a Number operand, except for Compare instruction*
- *Denormalized result or Underflow, when either Underflow or Inexact Enable bits are set or the FS bit is not set.*
- *Reserved opcodes*
- *Unimplemented formats*
- *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 do not trap if their operands are either denormalized or NaNs.

The use of this exception for such conditions is optional; most of these conditions are newly developed and are not expected to 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.

The enabling of traps have the following results:

- *Trap Enabled: The original operand values are undisturbed.*
- *Trap Disabled: 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. If the pending instruction cannot be completed, this instruction is placed in the *Exception* register, if present. 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.

Primary Caches

Notes

Introduction

This chapter describes the on-chip primary cache, the individual operations of the primary cache, and the organization and operations of the on-chip secondary cache controller.

Figure 9.1 shows the RC64574/RC64575 system memory hierarchy. In the logical memory hierarchy, caches lie between the CPU execution core and main memory. They are designed to make the speedup of memory accesses transparent to the user. Each functional block in Figure 9.1 has the capacity to hold more data than the block above it. At the same time, each functional block takes longer to access than any block above it. To improve, the speed of access to stored instructions and data, the processor has two on-chip primary caches, one for instruction and one for data.

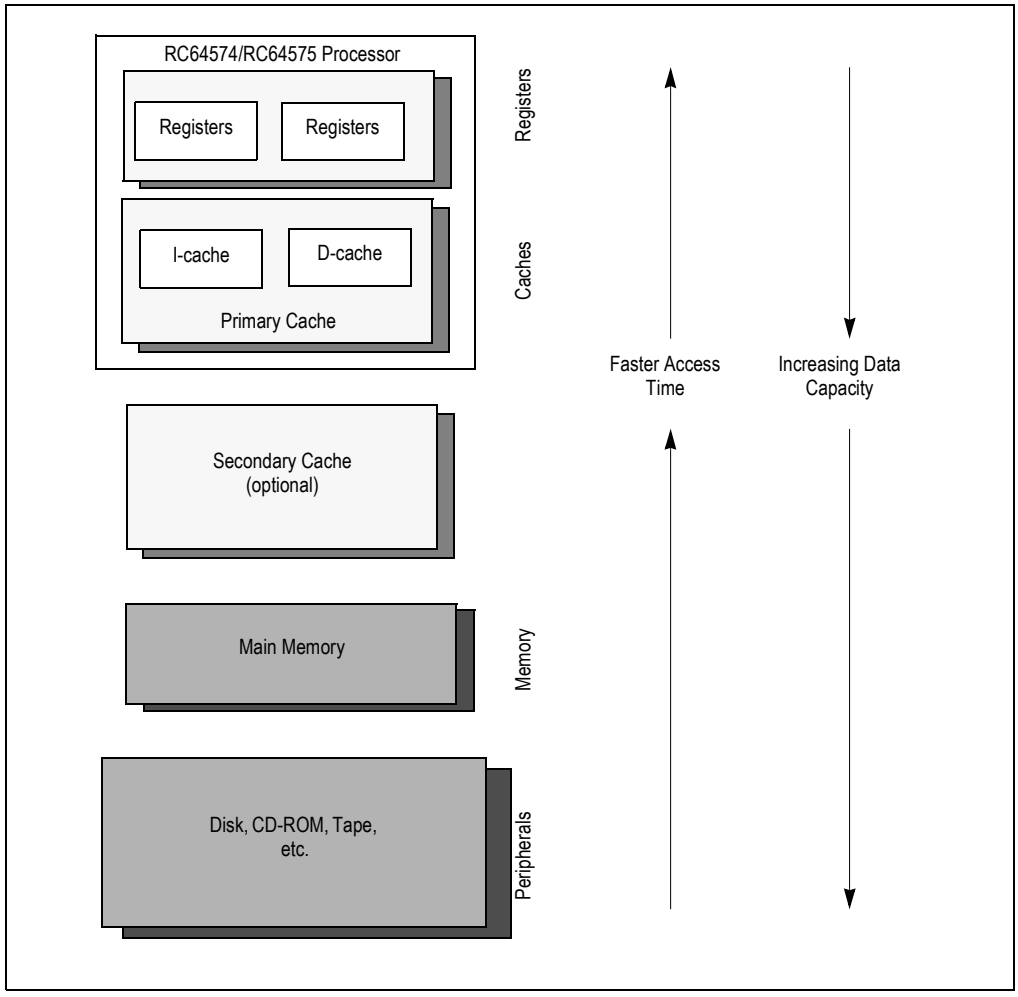


Figure 9.1 Logical Hierarchy of Memory

Caches provide fast, temporary storage, and they make the speedup of memory accesses transparent to the user. In general, the processor attempts to access the next-required instruction or datum in the primary cache. A successful access is called a primary-cache hit. If the instruction/data is not present in the primary cache, it is retrieved as a cache line from secondary cache (if available) or memory and is written into the primary cache. For a data cache miss, the processor can restart the pipeline after the first double-word (the one at the miss address) is retrieved and continues the cache line refill in parallel.

Notes

It is possible for the same data to be in two places simultaneously: main memory, and primary cache. This data is kept consistent through the use of either a write-back or a write-through protocol. For a write-back cache, the modified data is not written to memory until the cache line is replaced. In a write-through cache, the data is written to memory when the cached data is modified (with a possible delay due to the write buffer).

Primary Caches

This section describes the organization of on-chip primary caches.

Cache Line Size

A *cache line* is the smallest unit of information that can be fetched from memory to be filled into the cache. A primary cache line is 8 words (32 bytes) in length and is represented by a single tag. Upon a cache miss in the primary cache, the missing cache line is loaded from memory into the primary cache.

RC64574/RC64575 Cacheability/Coherency Attributes

In the RC64574/RC64575, TLB fields, certain CP0 register fields, and the caches have a 3-bit field describing the cacheability/coherency of virtual pages. Refer to Table 9.1.

Value	Attributes
0	Cacheable, non-coherent, write-through, no write-allocate
1	Cacheable, non-coherent, write-allocate
2	Uncacheable
3	Cacheable, non-coherent, write-back, write-allocate
5-7	Reserved. Must not be used.

Table 9.1 RC64574/RC64575 Memory Coherency/Cacheability Attributes

Cache Organization, Operation and Coherency

Attributes of the RC64574/RC64575 caches are shown in Table 9.2.

Attribute	Instruction	Data
size	32KB	32KB
organization	2-way set associative	2-way set associative
lockability	per line	per line
line size	32B	32B
read unit	64 bits	64 bits
write policy	n.a.	Write-back, write-through with/without allocate, uncached (determined on a “per page” basis)
line transfer order	sub-block order	sub-block order
miss restart after transfer of	entire line	miss word
parity	per-word	per-byte
write-back protocol	n.a.	write-back after new line read

Table 9.2 RC64574/RC64575 Primary Cache Attributes

The RC64574/RC64575 does not implement cache-coherency mechanisms for multi-processing. There are no mechanisms for external agents to “snoop” or otherwise intervene in the primary cache.

Notes

The RC64574/RC64575 uses mechanisms similar to those found in the RC5000. These include:

- *Cache operations which specify an index into the cache will use a high-order cache index bit to specify the target cache set. For the 32KB caches of the RC64574/RC64575, Address bit 14 specifies the set of the cache for cache operations.*
- *A "FIFO" bit is used to maintain the LRU replacement algorithm used in set selection.*
- *The cache-line replacement algorithm works as follows:*
  - *if the indexed line in both sets are invalid, set "A" will be selected;*
  - *if the indexed line of one set is "locked", the other set will be selected;*
  - *if the indexed line of both sets are locked, set B will be selected, and will be unlocked;*
  - *otherwise, the "F" bit will be used to select the least-recently-used set for replacement.*

Organization of the Primary Instruction Cache (I-Cache)

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 32-bit tag that contains a 24-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. Instruction-type bits determine whether the even-word instruction is an integer instruction or a floating-point instruction. A four-bit "IType" field is created to be written with the tag. The four bits correspond to each of the four instruction pairs associated with this tag, with the most-significant bit corresponding to the most-significant instruction. Figure 9.2 shows the instruction line cache format.

Each cache line can be separately locked to ensure that critical instructions, such as interrupt service routines, are always available to the CPU. Up to half the I-caches can be locked.

The primary I-cache 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 per line.*

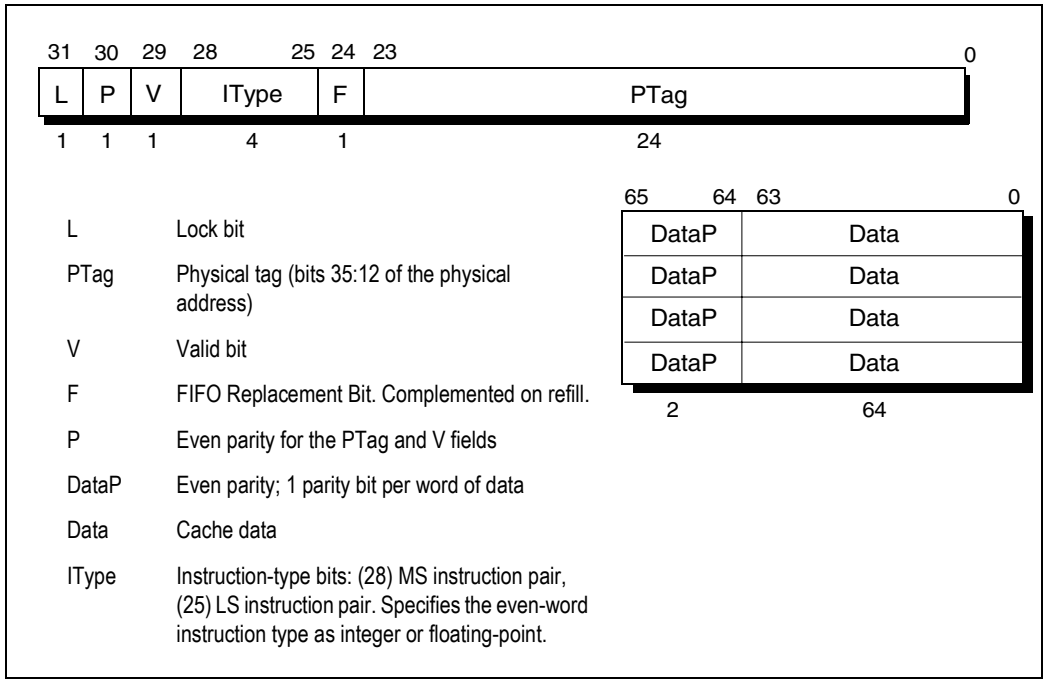


Figure 9.2 RC64574/RC64575 Instruction Cache Line Format

Notes

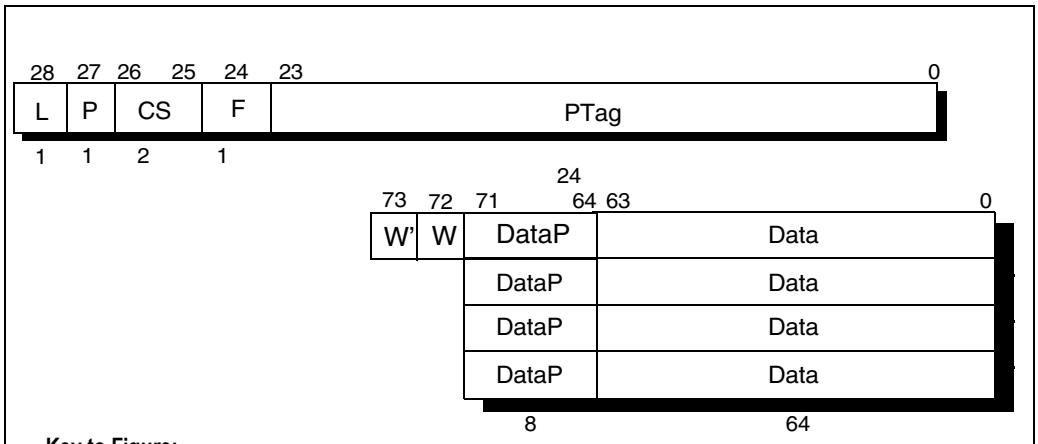
Organization of the Primary Data Cache (D-Cache)

Each line of primary D-cache data has an associated 28-bit tag that contains a 24-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 9.3 shows the data cache line format.

The 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 per line.*

The format of a primary D-cache line is shown below. In the RC64574/RC64575, 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. *There is no hardware support for cache coherency. The only cache states used are Dirty Exclusive and Invalid.* Each cache line can be separately locked to ensure that critical data elements are always available to the CPU. Up to half the D-caches can be locked.



Key to Figure:

L	Lock bit
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)
Data P	Even parity for the data; 1-bit per byte
Data	Cache data

Figure 9.3 RC64574/RC64575 Data Cache Line Format

Notes

Cache Locking

The RC64574/RC64575 also supports a cache-locking feature that can be used to lock critical sections of code and/or data into on-chip caches to guarantee quick access.

A portion of a cache is said to be *locked* when a particular piece of code or data is loaded into a cache location that will not be selected later for refill by other data. The locking feature of the RC64574/RC64575 is on a per-line basis; that is, the kernel may set status register control bits that allow individual cache lines to be locked in the cache.

Locked cache lines can be changed by any of the following operations or conditions:

- *cache operations*
- *store operations to cached virtual address*
- *if they become valid*

When To Use Cache Locking

Cache locking is useful in the following cases:

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

In the RC64574/RC64575, both the Instruction and Data cache are two-way set associative, with set A and set B. By setting the DL or IL bit in the Status register of CP0, a refilled cache line of a selected set, at that time, can be locked in the appropriate cache; therefore, a future fill into this cache line will always use the other set. Furthermore, if one set of a cache line has already been locked, the second attempt to lock this cache line will be ignored.

As previously noted, a Data store operation to locked data will update the D-cache contents; locking merely prevents the cache line contents from being replaced by the contents of a different physical location. The locked cache line can be unlocked by using a Cache operation to invalidate that line. Anytime the *valid* bit of a cache line is cleared, the *lock* bit is cleared simultaneously.

Example: Data Cache Locking

For this example, assume an application in which a table must be kept in cache. After completing the initialization of data structures, etc., in the start-up code, the DL bit in the Status register can be set to enable the cache line locking, perform reads through cached addresses to load the data into the data cache, and then—to prevent further cache locking— clear the DL bit. A sample code fragment for the Data Cache Locking operation follows:

```
        .set noreorder
jal     flush_cache         /* Flush the cache */
mfc0    a0, C0_SR           /* Get old SR value */
li      a1, SR_SET_DL       /* SR_SET_DL = 0x01000000 */
or      a0, a0, a1
mtc0    a0, CO_SR           /* Set the Lock bit for data cache */
nop
nop
nop                         /* 3 nops: safety against CP0 hazard */

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 */
```

Notes

```
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_CLR_DL      /* SR_CLR_DL = 0xfeffffff */
      and    a0, a0, a1
mtc0   a0, C0_SR         /* Clear the Lock bit for data cache */
nop
nop
      nop                /* 3 nops: safety against CP0 hazard */
```

Example: Instruction Cache Locking

For this example, assume an application in which a critical function must be kept in cache. Also assume that the size of the function is known. (If not known, the size can be determined by generating a disassembly of the object file.)

After completing the initialization of data structures, etc., in the start-up code, the IL bit of the Status register can be set to enable cache line locking, perform the FILL operation in the CACHE instruction that will fill the instruction cache with the critical function, and then—to prevent further cache locking—clear the IL bit.

A sample code fragment for the Instruction Cache Locking operation follows:

```
      .set noreorder
jal                                    flush_cache       /* Flush the cache */
      la       t0, 1f          /* Get address of label '1' */
      li       t1, 0xA0000000
      or       t0, t0, t1
      jr    t0            /* Uncached execution from now onwards */
      nop

1:    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 */
      mfc0     a0, C0_SR          /* Get old SR value */
      li       a1, SR_SET_IL      /* SR_SET_IL = 0x00800000 */
      or       a0, a0, a1
mtc0   a0, C0_SR        /* Set Lock bit for instruction cache */
      nop
      nop
      nop

2:                                 cache   Fill_I, 0(t0)      /* Fill Operation */
      addiu    t2, 4
      bneq     t2, t1, 2b         /* Loop back till done */
      addiu    t0, 4             /* bump read address */

      mfc0     a0, C0_SR          /* Get old SR value */
```

Notes

```
        li      a1, SR_CLR_IL       /* SR_CLR_IL = 0xff7fffff */
        and     a0, a0, a1
        mtc0    a0, C0_SR           /* Clear 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 */
```

Accessing the Primary Caches

Figure 9.4 shows the virtual address (VA) index into the primary caches. Each instruction and data cache size is 32 Kbytes.

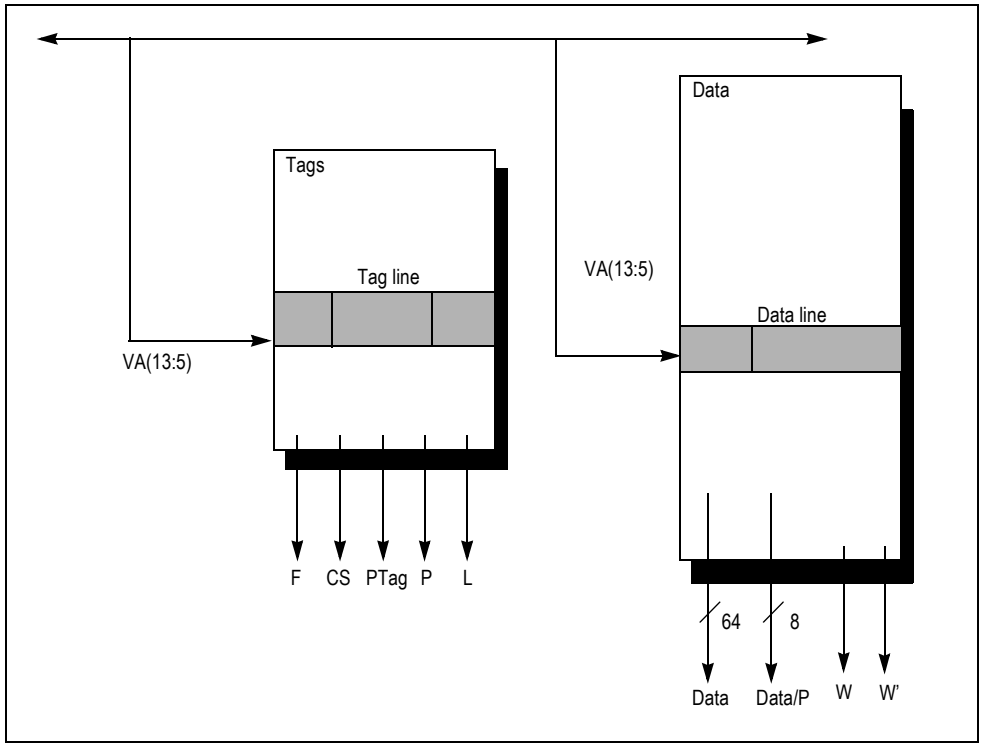


Figure 9.4 Primary Cache Data and Tag Organization

Cache-Line 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 RC64574/RC64575. All cache lines are in an exclusive state.*
- ***Dirty****: a cache line containing data that has changed since it was loaded from memory.*
- ***Clean****: a cache line containing data that has not changed since it was loaded from memory.*
- ***Shared****: a cache line present in more than one cache in the system. The RC64574/RC64575 does not provide for hardware cache coherency. This state will never happen in normal operations.*

Notes

The RC64574/RC64575 supports the four data-cache states shown in Table 9.3. Under normal operations, however, the only cache-line states that will occur in the data cache are the *Dirty Exclusive* and *Invalid* states. Each primary instruction cache line is either valid or invalid.

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.

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 Coherency” on page 9-10). 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 9-8). Use the W bit to determine if the line must be written back on replacement.

Table 9.3 Cache States

Cache-Line Ownership

The processor is the owner of a cache line when it is in the *dirty exclusive* state, and is 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. The cache instruction is explained further in the IDT MIPS Microprocessor Family Software Reference Manual.*
- *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.

Cache Write Policy

The RC64574/RC64575 manages its primary data cache by using either a write-back or a write-through policy on a per-page basis. Refer to Table 9.4. Four policies are supported:

- *Uncached*
- *Writeback*
- *Write-Through With Write Allocation*
- *Write-Through With No Write Allocation.*

Uncached writes do not modify the cache and are written directly to main memory.

Notes

A write-back policy does not write data to main memory until the cache line is replaced or the CACHE instruction flushes and possibly invalidates the dirty cache line. If a write hits in cache, the data is stored in cache and the cache line is marked dirty: no main memory access occurs. If a write misses in cache, the replaced cache line is written to memory, if dirty. The new cache line is read, the data is written to cache, and the cache line is marked dirty.

A write-through policy writes the data to main memory, immediately. If a write hits in cache, the data is stored in cache and written to memory. The cache line is *not* marked dirty, since cache and main memory have the same value. If the write misses in the cache and the policy is write-through with write allocation, the replaced cache line is written to memory, if dirty; the new cache line is read, the data is stored in cache and written to memory. If the policy is write-through with no write allocate, the data is written to memory and does not modify the cache.

Cache Write Policy	Cache Hit	Cache Miss
Uncached	N/A	Write to main memory
Write-back	Store in cache Cache line marked dirty	Flush cache line (if dirty) Read new cache line Store in cache Cache line marked dirty
Write-Through Write Allocate	Store in cache Write to main memory	Flush cache line (if dirty) Read new cache line Store in cache Write to main memory
Write-Through No Write Allocate	Store in cache Write to main memory	Write to main memory

Table 9.4 CPU Cache Write Policy

Cache-State Transitions

Figure 9.5 shows the cache-state transitions for 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 10 for a definition of an external agent). This is because the only read request the RC64574/RC64575 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 dirty exclusive by the RC64574/RC64575. 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:*
  - *for a Cache invalidate operation*
  - *if the line is replaced*

.

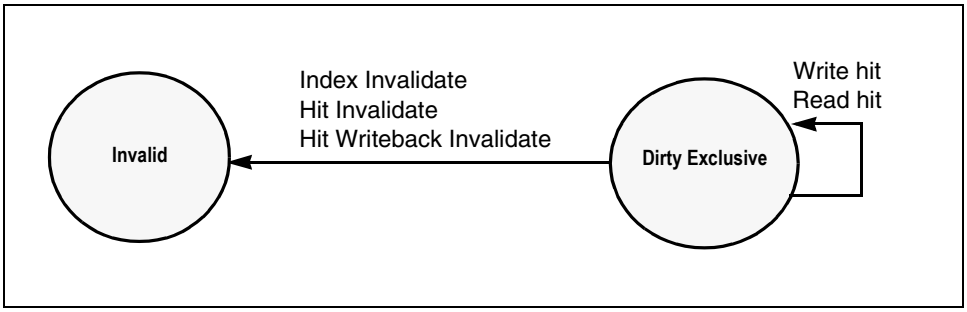


Figure 9.5 Primary Data Cache State Diagram

Notes

Cache Coherency

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 RC64574/RC64575 does not provide any hardware cache coherency. Cache coherency must be handled with software.

Cache Coherency Attributes

The TLB contains 3 bits per entry that provide two possible cache-coherency attribute types, *uncached* and *noncoherent*. These attribute types can be used to control cache coherency on a per-page basis. Table 9.5 lists the two attribute types and summarizes the behavior of the processor on load misses and store misses for each type.

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 9.5 Coherency Attributes and Processor Behavior

Uncached Attribute

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 Attribute

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.

Multiprocessor 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 RC64574/RC64575 does not support symmetric multi-processing (SMP), these are useful for multi-master and heterogeneous 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.

Figure 9.6 illustrates a test-and-set synchronization procedure that uses a semaphore; when the semaphore is cleared to 0, the shared data is unlocked, and when the semaphore is set to 1, the shared data is locked.

Notes

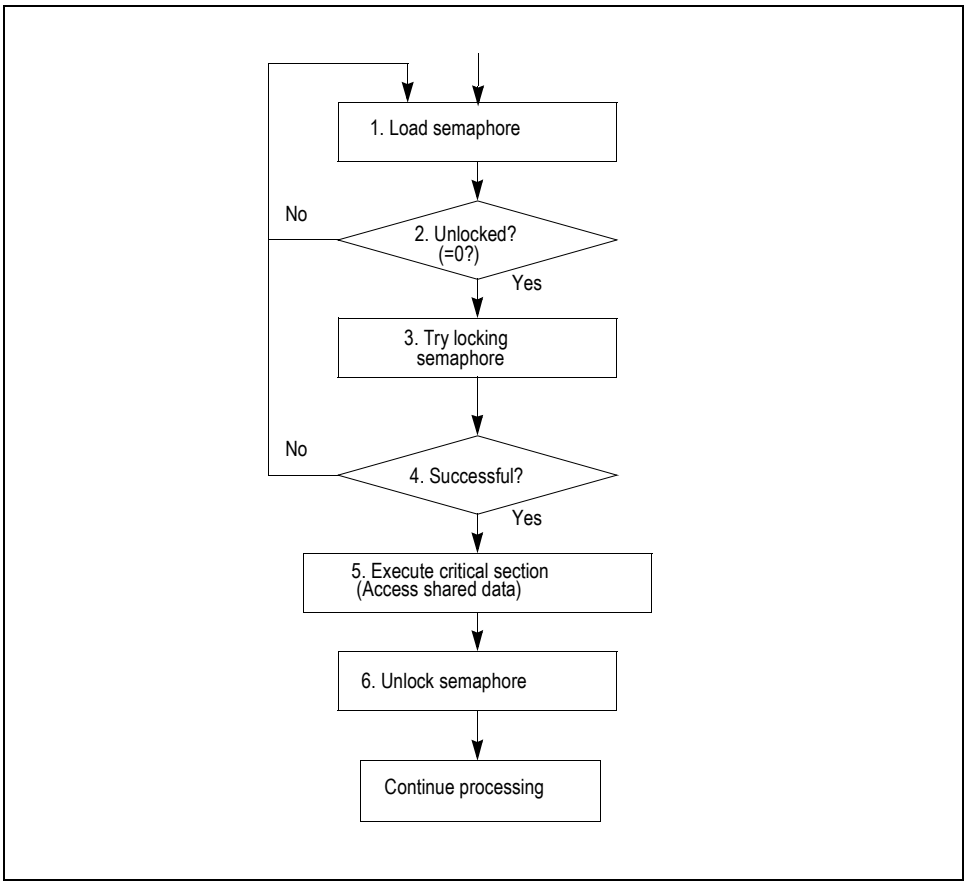


Figure 9.6 Synchronization with Test-and-Set

The processor begins by loading the semaphore and checking to see if it is unlocked (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 9.7 shows this process.

Notes

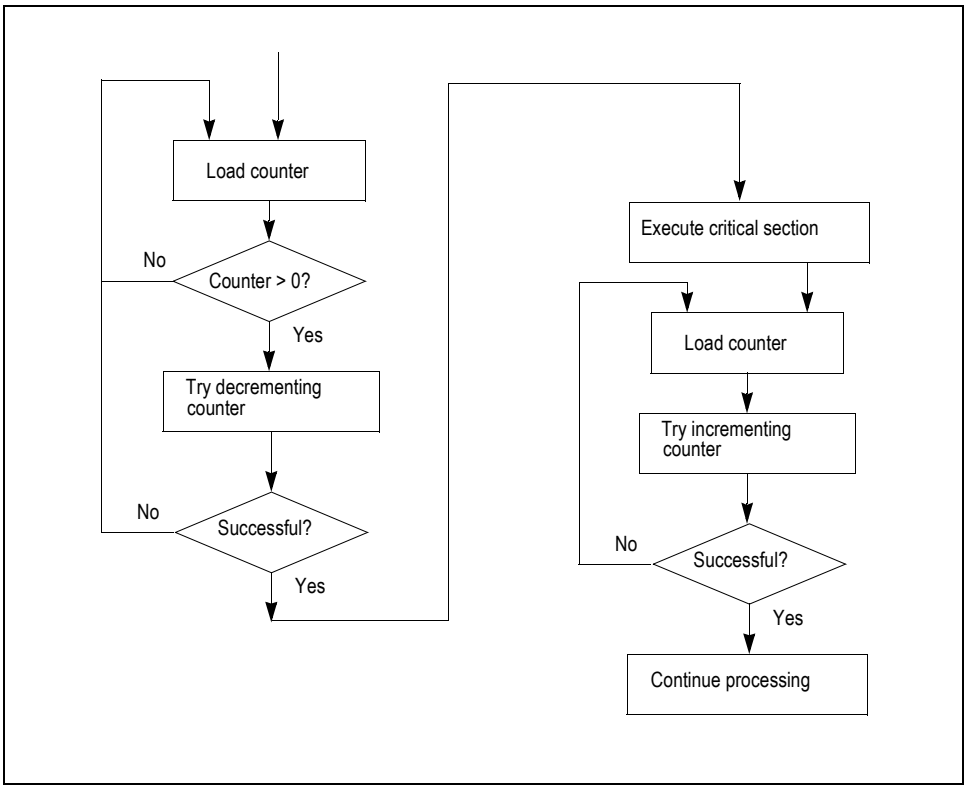


Figure 9.7 Synchronization Using a Counter

Load Linked and Store Conditional

The RC64574/RC64575 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*

Figure 9.8 shows how to implement test-and-set using LL and SC instructions, and Figure 9.9 shows synchronization using a counter.

Notes

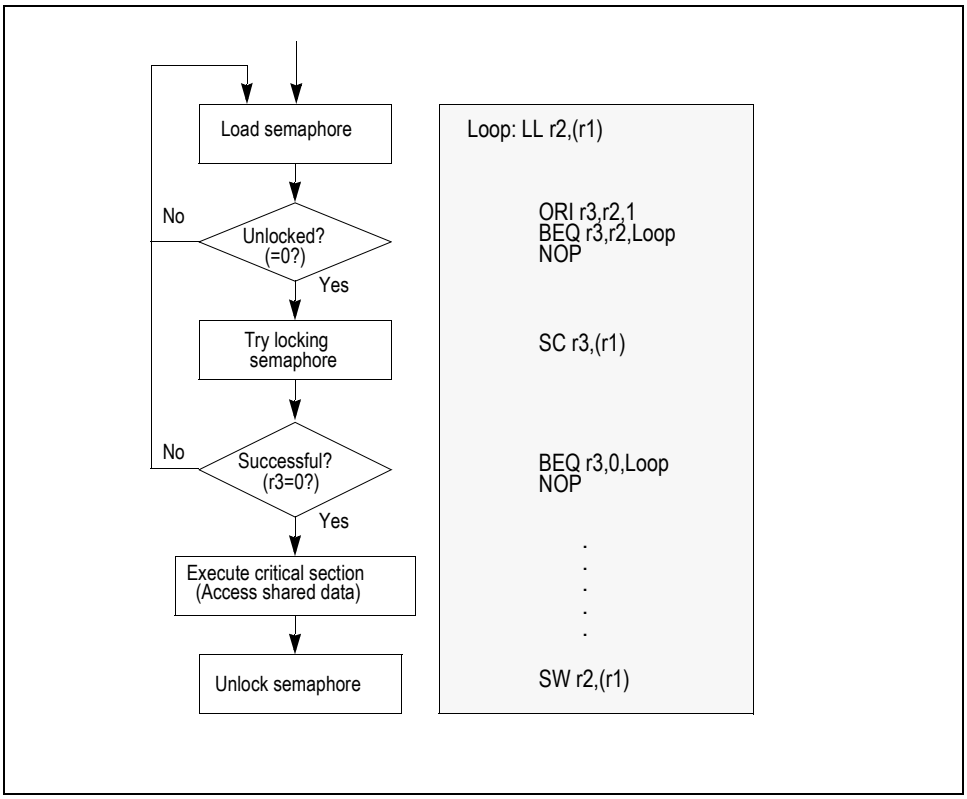


Figure 9.8 Test-and-Set using LL and SC

Notes

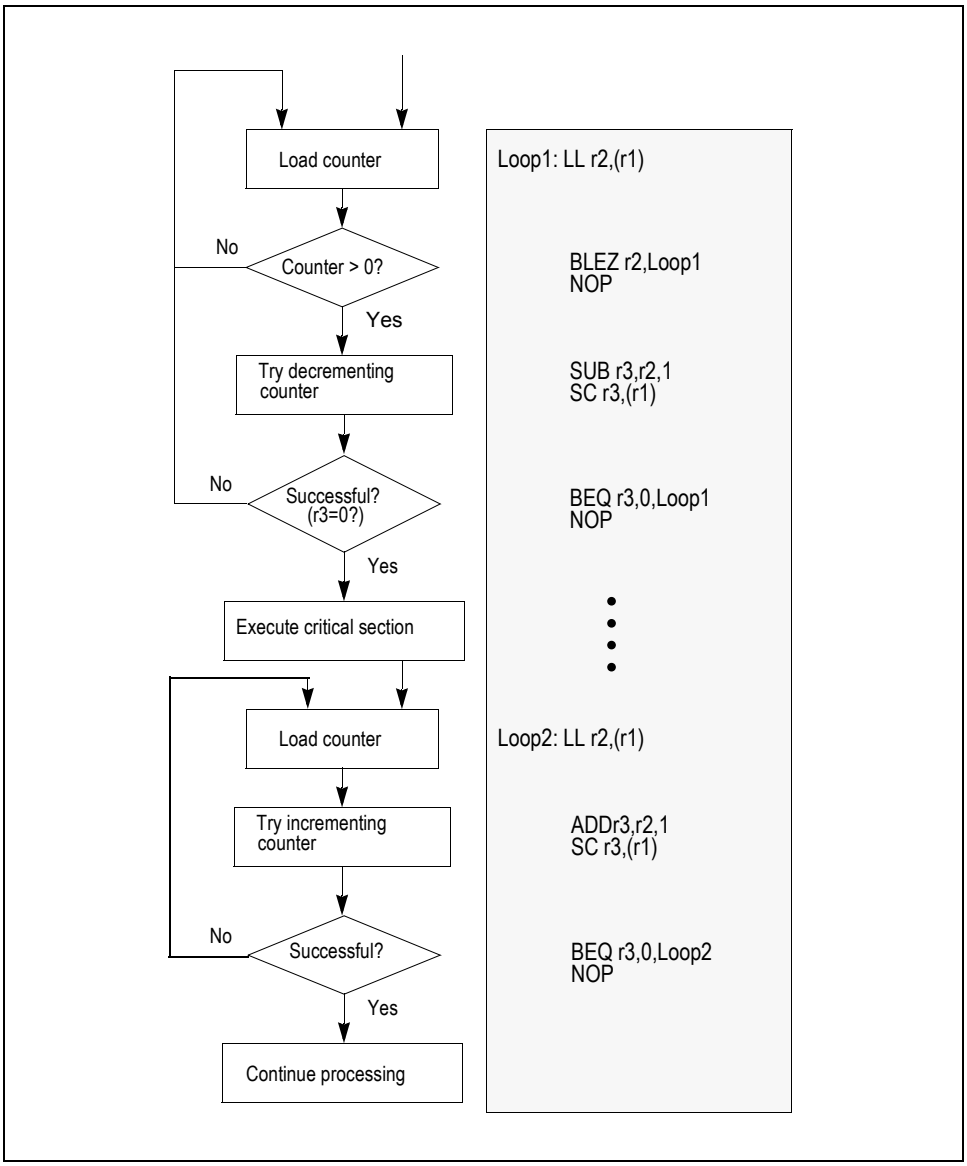


Figure 9.9 Counter Using LL and SC

Processor Signal Descriptions Introduction

Notes

This chapter describes the signals used by and in conjunction with the RC64574 and R64575 processors. Signals include the System interface, the Clock/Control interface, the Interrupt interface, the Initialization interface, the handshake signals and JTAG Interface.

Pin Description Table

The following table contains a list of system interface pins available on the RC64574/575. Pin names ending with an asterisk (*) are active when low.

Pin Name	Type	Description
System Interface		
ExtRqst*	I	**External request** An external agent asserts ExtRqst* to request use of the System interface. The processor grants the request by asserting Release*.
Release*	O	**Release interface** In response to the assertion of ExtRqst* or a CPU read request, the processor asserts Release* and signals to the requesting device that the system interface is available.
RdRdy*	I	**Read Ready** The external agent asserts RdRdy* to indicate that it can accept a processor read request.
WrRdy*	I	**Write Ready** An external agent asserts WrRdy* when it can now accept a processor write request.
ValidIn*	I	**Valid Input** Signals that an external agent is now driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.
ValidOut*	O	**Valid Output** Signals that the processor is now driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.
SysAD(63:0)	I/O	**System address/data bus** A 64-bit address and data bus for communication between the processor and an external agent. In 64 bit interface mode, during address phases only, SysAd(35:0) contains valid address information. The remaining SysAD(63:36) pins are not used. The whole 64-bit SysAD(63:0) may be used during the data transfer phase. For all double-word accesses (read or write), the low-order 3 bits (SysAD[2:0]) will always be output as zero during the address phase. In 32-bit interface mode and in the RC64574, 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(7:0)	I/O	**System address/data check bus** An 8-bit bus containing parity check bits for the SysAD bus during data bus cycles. In 32-bit mode and in the RC64574, SysADC(7:4) is not used. The SysADC(3:0) contains check bits for SysAD(31:0).
SysCmd(8:0)	I/O	**System command/data identifier bus** A 9-bit bus for command and data identifier transmission between the processor and an external agent.
SysCmdP	I/O	**System Command Parity** A single, even-parity bit for the SysCmd bus. This signal is always driven low.

Table 10.1 Pin Descriptions (Part 1 of 3)

Notes

Pin Name	Type	Description
Clock/Control Interface		
SysClock	I	**SystemClock** The system 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).
$V_{CC}P$	I	Quiet VCC for PLL Quiet VCC for the internal phase locked loop.
$V_{SS}P$	I	**Quiet V_{SS} for PLL** Quiet VSS for the internal phase locked loop.
Interrupt Interface		
Int*(5:0)	I	**Interrupt** Six general processor interrupts, bit-wise ORed with bits 5:0 of the interrupt register.
NMI*	I	**Non-maskable interrupt** Non-maskable interrupt, ORed with bit 6 of the interrupt register.
Initialization Interface		
$V_{CC}Ok$	I	**V_{cc} is OK** When asserted, this signal indicates to the processor that the power supply has been above the Vcc minimum for more than 100 milliseconds and will remain stable. The assertion of VCCOk initiates the initialization sequence.
ColdReset*	I	**Cold reset** This signal must be asserted for a power on reset or a cold reset. ColdReset must be de-asserted synchronously with SysClock.
Reset*	I	**Reset** 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 de-asserted synchronously with SysClock.
ModeClock	O	**Boot-mode clock** Serial boot-mode data clock output at the system clock frequency divided by two hundred fifty-six.
ModeIn	I	**Boot-mode data in** Serial boot-mode data input.
JTAG Interface		
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. An external pullup register is required.
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. An external pullup register is required.

Table 10.1 Pin Descriptions (Part 2 of 3)

Notes

Pin Name	Type	Description
TRST*	I	**JTAG Reset** 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 tristate the TDO pin. An external pullup register is required.
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 un connected.
JR_V_{cc}	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 10.1 Pin Descriptions (Part 3 of 3)

Logic Diagram — RC64574/RC64575

Figure 10.1 illustrates the direction and functional groupings for the processor signals.

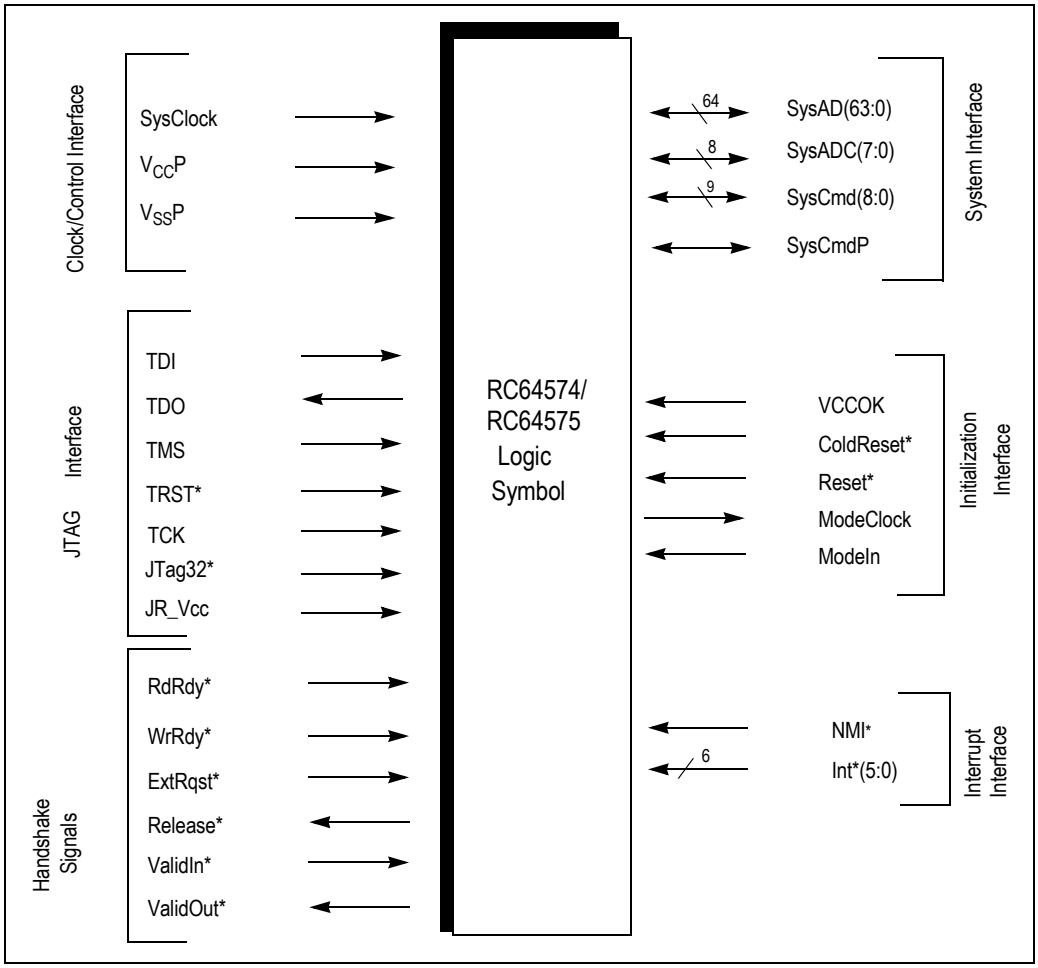


Figure 10.1 Logic Symbol for RC64574/RC64575

Notes

RC64574 Socket Compatibility to RC64474 & RC4640

The RC64574/575 is 100% pin compatible with the RC64474/475 with the supply voltage being the only difference. RC64474/475 requires a 3.3V supply, while RC64574/575 requires a 2.5V supply.

To ensure socket compatibility between the RC64574/RC64474 and the RC4640 devices, several pin changes are required, as shown in Table 10.2 and Table 10.3. **Note:** The RC64574/575 are 2.5V parts and as such all Vcc must be at the correct voltage for a given part.

Pin	RC4640	RC64574/ RC64474	Compatible to RV4640?	Comments
1	N.C	JTAG32*	Yes.	Pin has an internal pull-down, to enable 32-bit scan. Can also be left a N.C.
48	V_{ss}	TDO	Yes.	Can be driven with V_{ss}, if JTAG is not needed. Is tristated when TRst* is low.
49	V_{ss}	TMS	Yes.	Can be driven with V_{ss} if JTAG is not needed.
50	V_{ss}	TCK	Yes.	Can be driven with V_{ss} if JTAG is not needed.
51	V_{ss}	TRst*	Yes.	Can be driven with V_{ss} if JTAG is not needed.
52	V_{ss}	TDI	Yes.	Can be driven with V_{ss} if JTAG is not needed.
71	N.C.	JR_V_{cc}	Yes.	Can be left N.C. in RC64574, if JTAG is not need. If JTAG is needed, it must be driven to V_{cc}.

Table 10.2 RC64574 Socket Compatibility to RC64474 and R4640

RC64575 Socket Compatibility to RC64475 & RC4650

Pin	RV4650 32-bit	RC64575 32-bit RC64475 32-bit	RV4650 64-bit	RC64575 64-bit RC64475 64-bit	Compatible to RV4650?	Comments
53	N.C.	JTAG32*	No Connect	JTAG32*	Yes	In 32-bit, this pin can be left unconnected because of internal pull-down. In 64-bit, this assumes that JTAG will not be used. If using JTAG, this pin must be at V_{cc}.
150	N.C.	JR_V_{cc}	No Connect	JR_V_{cc}	Yes	In RC64475, can be left a N.C, if JTAG is not need. If JTAG is needed, it must be driven to V_{cc}.
180	N.C.	TDI	No Connect	TDO	Yes	If JTAG is not needed, can be left a N.C.
181	N.C.	TRsT*	No Connect	TRsT*	Yes	If JTAG is not needed, can be left a N.C.
182	N.C.	TCK	No Connect	TCK	Yes	If JTAG is not needed, can be left a N.C.
183	N.C.	TMS	No Connect	TMS	Yes	If JTAG is not needed, can be left a N.C.
184	N.C.	TDO	No Connect	TDIO	Yes	If JTAG is not needed, can be left a N.C.

Table 10.3 RC64575 Socket Compatibility to RC64475 & RC4650

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 RC64575 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 RC64574 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 11.1 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

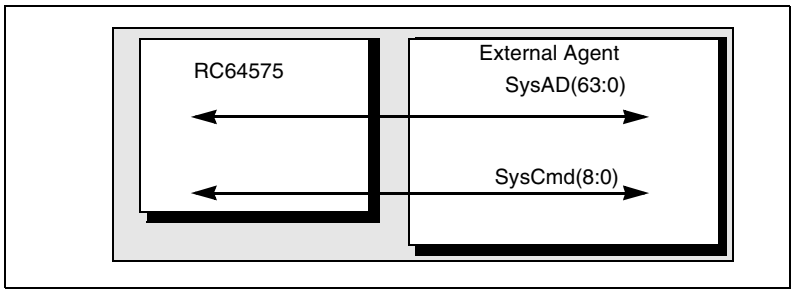


Figure 11.1 System Interface Buses

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 11.2, **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 11.2). **RdRdy*** does not need to be asserted during the issue cycle.

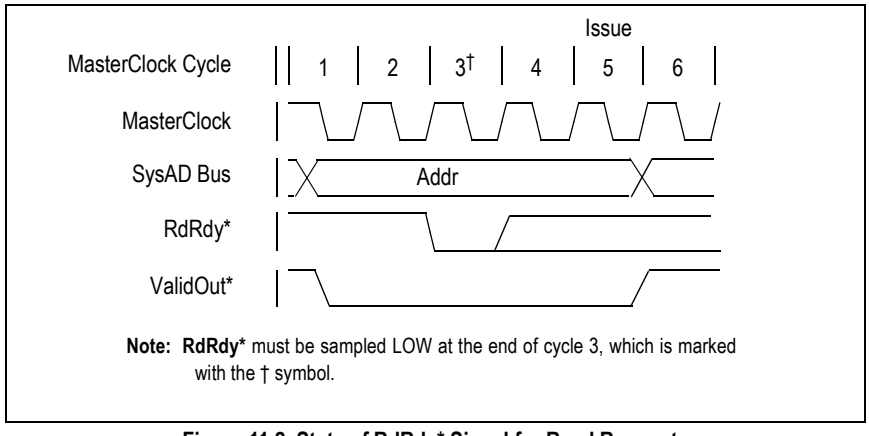


Figure 11.2 State of RdRdy* Signal for Read Requests

As shown in Figure 11.3, **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 11.3). **WrRdy*** does not need to be asserted during the issue cycle.

Notes

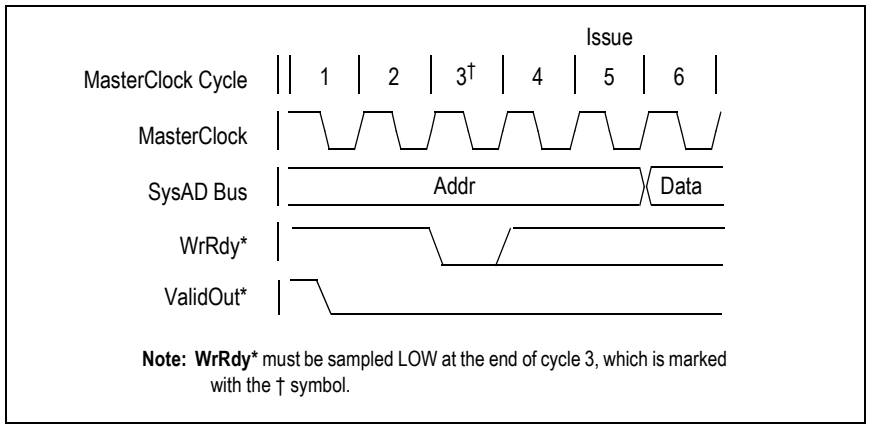


Figure 11.3 State of WrRdy* Signal for Write Requests

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 RC64574/RC64575 processors use **ValidOut*** and the external agent uses **ValidIn*** to indicate valid command and data on the **SysCmd** and **SysAD** buses.*

System Interface Protocols

Figure 11.4 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.

[1] **MasterClock** is the input clock to the processor.

Notes

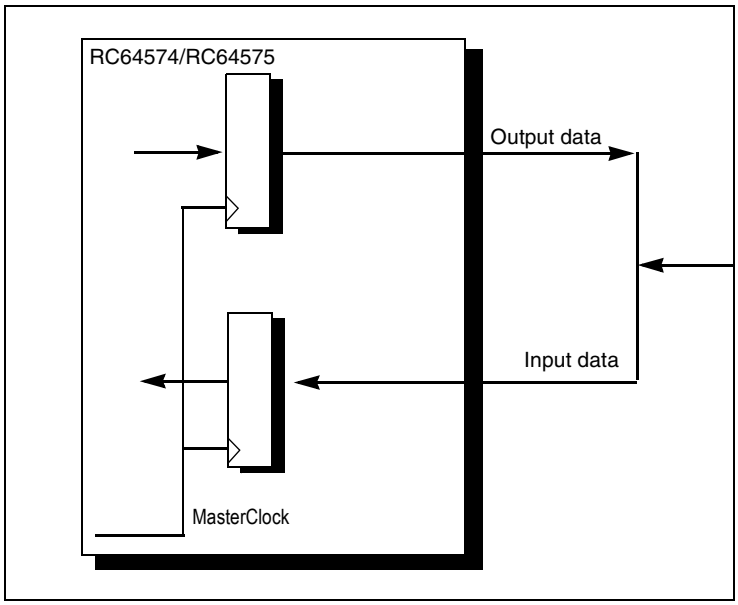


Figure 11.4 System Interface Register-to-Register Operation

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.

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.

Notes

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 11.5.

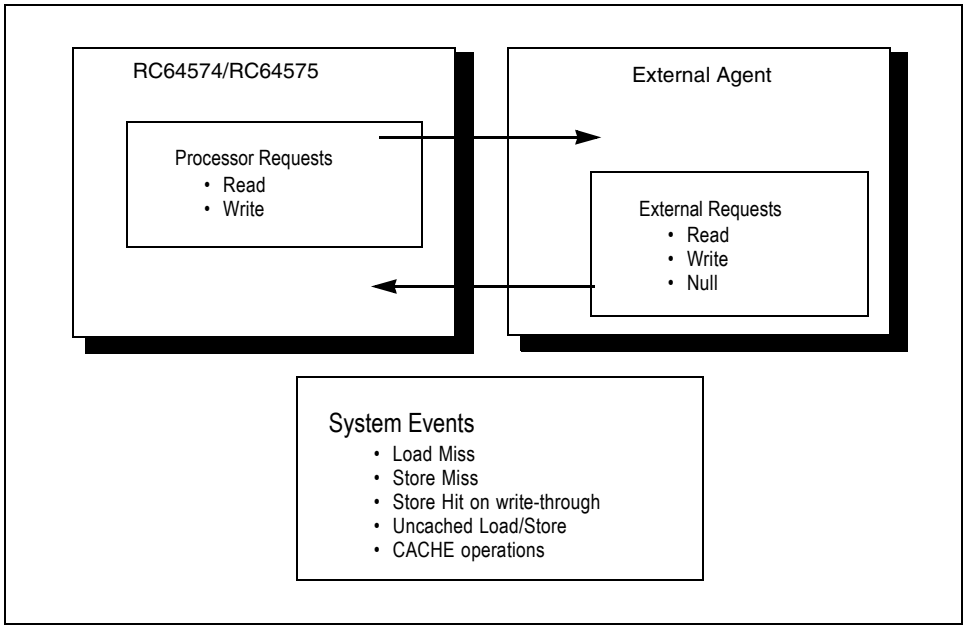


Figure 11.5 Requests and System Events

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 11.6.*

After the processor has issued a write request in either of the two new write modes, write reissue and pipelined writes, the processor can immediately issue a subsequent write, provided the **WrRdy*** requirement is met. Chapter 14 discusses this issue in more detail.

Notes

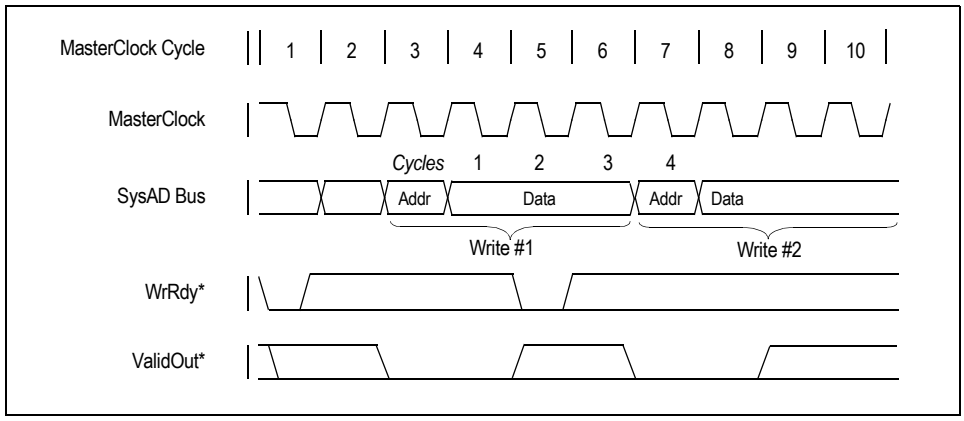


Figure 11.6 Back-to-Back Write Cycle Timing (RISCore4000 family)

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 11.7, processor requests include only reads and writes.

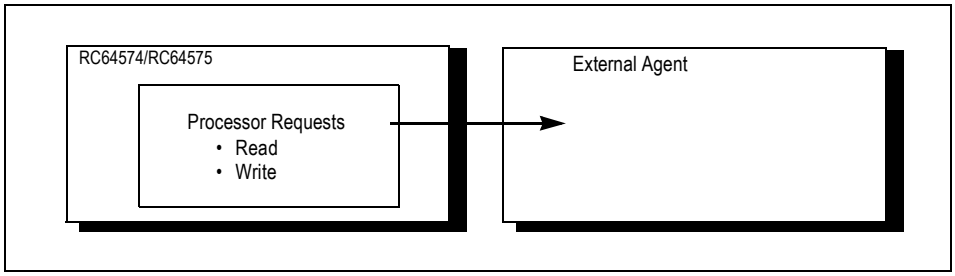


Figure 11.7 Processor Requests

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 11.8.

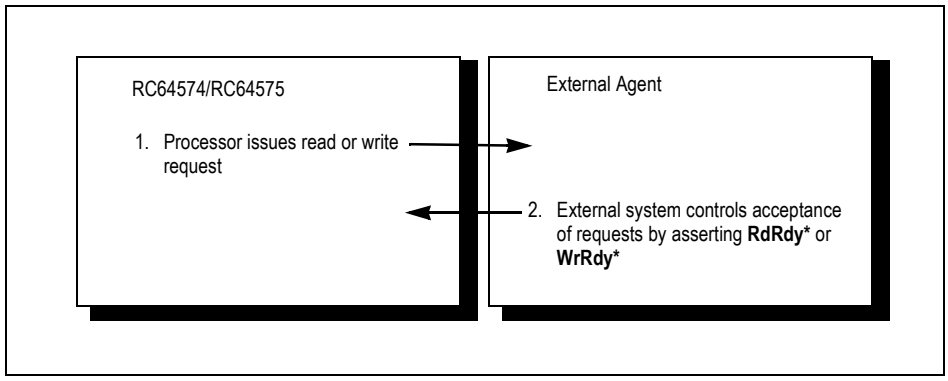


Figure 11.8 Processor Request

Notes

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 modes to enhance the throughput of non-block writes have been added to the RC64574/RC64575 devices. These modes allow for 2 cycle throughput on back-to-back non-block writes. The actual protocol is discussed in Chapter 14, "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 14, "The Write Interface").

External Requests

External requests include read, write and null requests, as shown in Figure 11.9. This section also includes a description of read response, a special case of an external request.

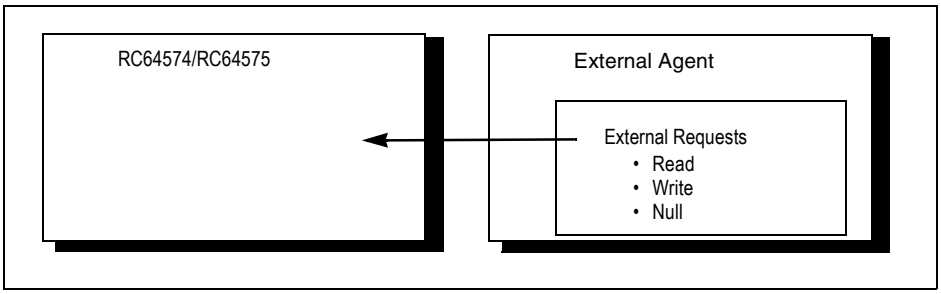


Figure 11.9 External Requests

***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 11.10. 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.

Notes

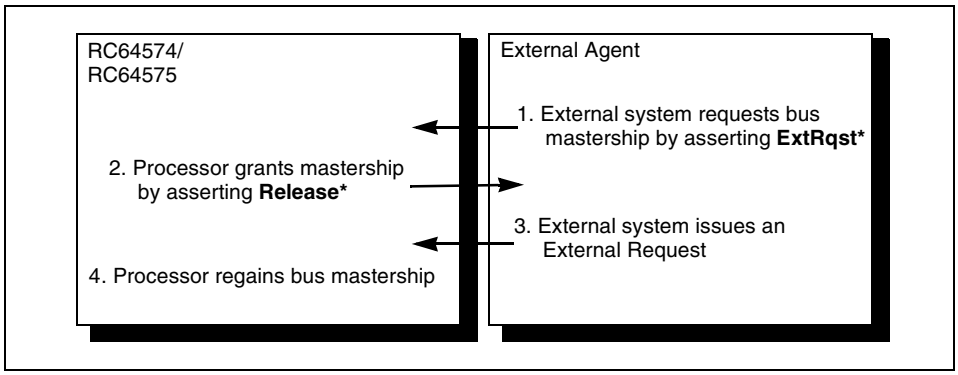


Figure 11.10 External Requests

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.*

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 RC64574/RC64575 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.

Notes

System Interface Endianness

The endianness of the system interface is programmed at boot time through the boot-time mode control interface (see "Initialization and Reset Interface" in Chapter 12 for specifics), and remains fixed until the next time the processor boot-time mode bits are read (reset). 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 11.1 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.

Category	Minimum PCycles	Maximum PCycles
1	4	6
2	4	24
3	0	0

Table 11.1 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.

Notes

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 Clocking, Reset and Initialization Interface

Notes

Introduction

This chapter describes the clock signals (“clocks”) used in the RC64574 and RC64575 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 12.1 and Figure 12.2 illustrate these terms.

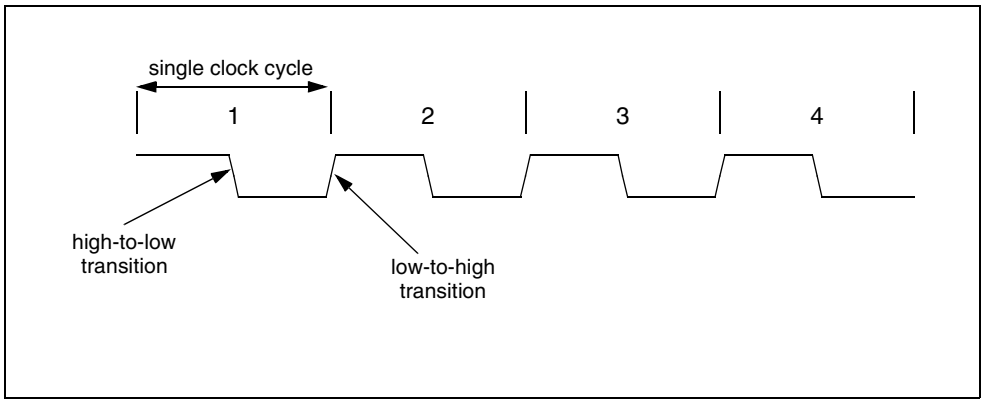


Figure 12.1 Signal Transitions

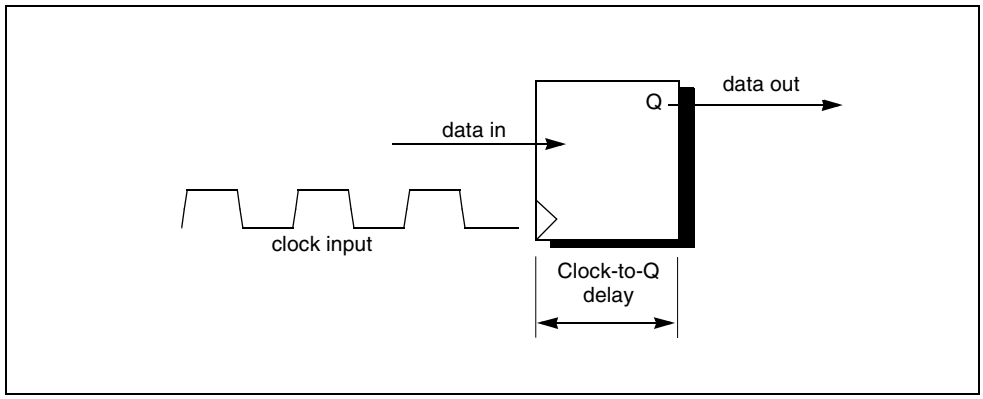


Figure 12.2 Clock-to-Q Delay

System Clocks

The RC64574/RC64575 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.

Notes

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.

Figure 12.3 shows the clocks data setup, output and hold timing[1] for the basic system clocks.

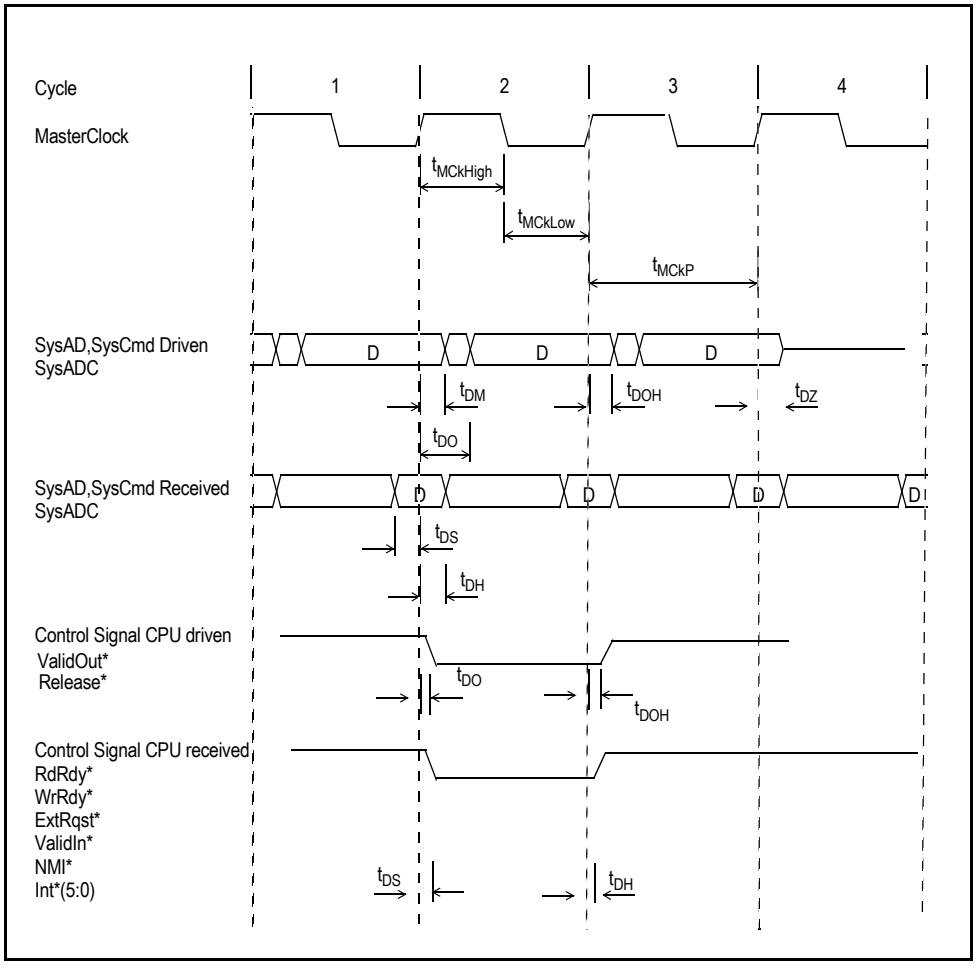


Figure 12.3 RC64574/RC64575 System Clocks Data Setup, Output, and HoldTiming

System Timing Parameters

As shown in Figure 12.3, 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**.

1. Values for AC & DC parameters are listed in the RC64574/RC64575 *RISController Embedded 64-bit Microprocessor* data sheet.

Notes

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.

PLL Components and Operation

The storage capacitor required for the Phase Locked Loop circuit is contained in the RC64574/RC64575 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 Figure 12.4.

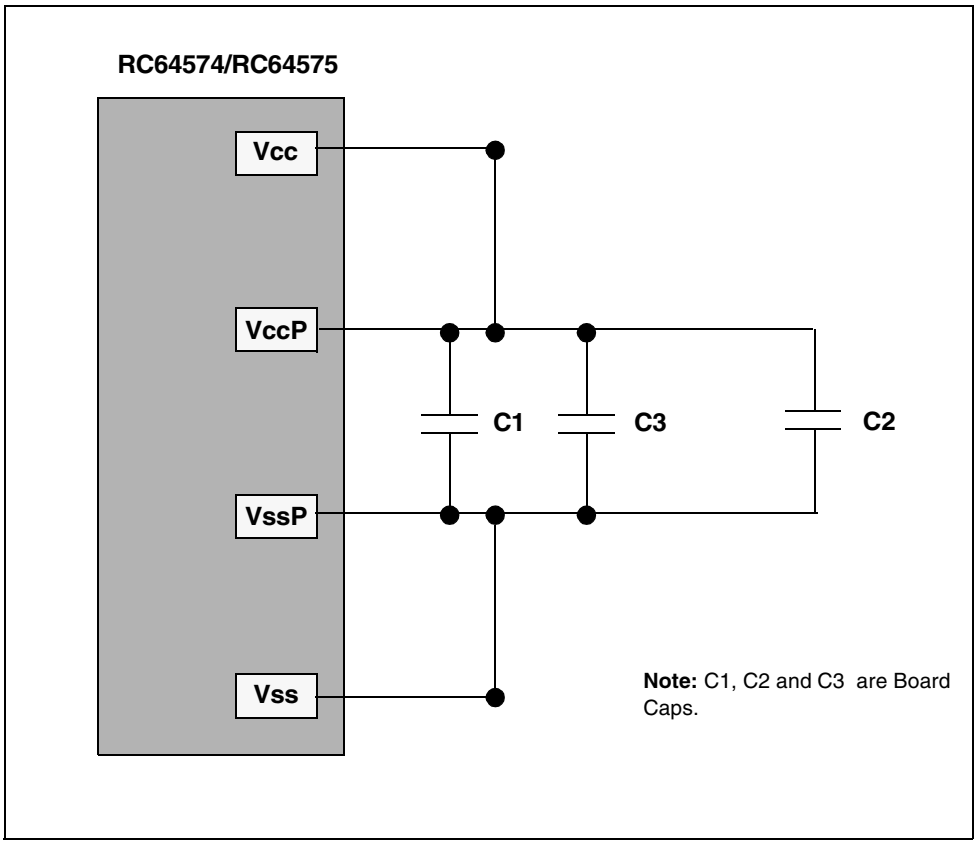


Figure 12.4 PLL Passive Components

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.

Notes

Connecting to an External Agent

MasterClock is used to drive both the processor and the external agent. The RC64574/RC64575 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)*

Figure 12.5 shows a block-level diagram of a system using the R4650 processor

.

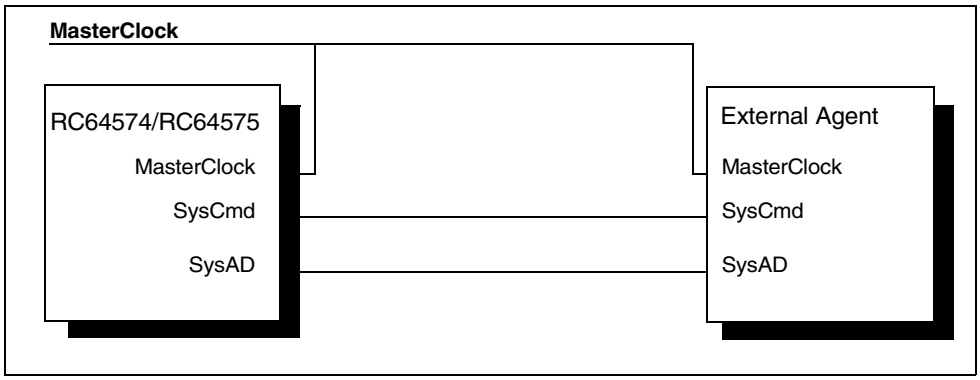


Figure 12.5 RC64574/RC64575 Processor System

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 12.6, Figure 12.7, and Figure 12.8. The RC64574/RC64575 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 12-6.*

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

- ***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 12.1 lists the processor signals and their possible states.

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

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 12.1 RC64474/RC64475 Processor Signal Summary

Notes

Power-On Reset

Figures 12.6, 12.7 and 12.8 illustrate the power-on, cold, and warm resets respectively. 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 10). 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

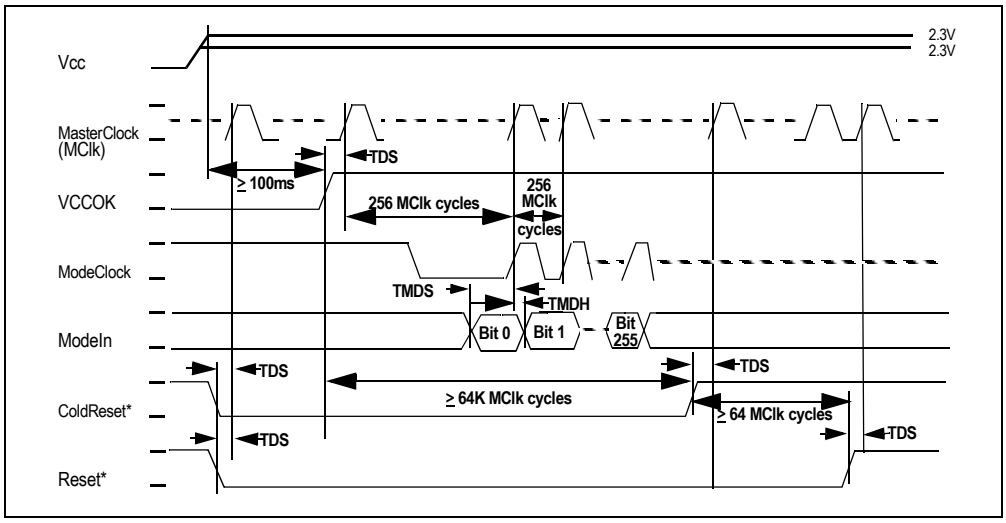


Figure 12.6 Power-on Reset

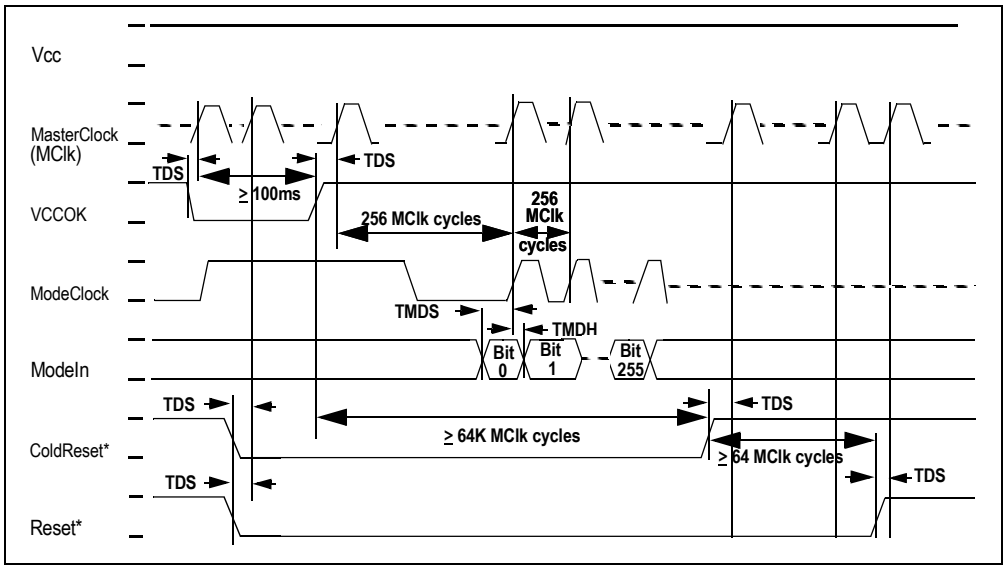


Figure 12.7 Cold Reset

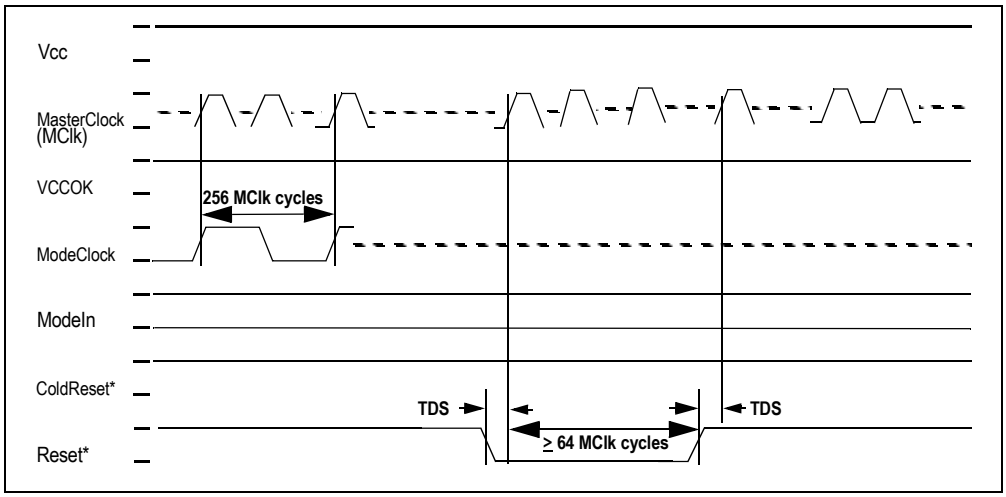


Figure 12.8 Warm Reset

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 12.2 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 27 to 256 are reserved bits.*
- *Zeros must be scanned in for all reserved bits.*

Serial Bit	Description	Value & Mode Setting
0	Reserved	Must be set to 0.
1:4	Transmit-data-pattern. Bit 4 is MSB	64-bit bus width: 0: DDDD 1: DDxDDx 2: DDxxDDxx 3: DxDxDxDx 4: DDxxxDDxxx 5: DDxxxxDDxxxx 6: DxxDxxDxxDxx 7: DDxxxxxxDDxxxxxx 8: DxxxDxxxDxxxDxxx 9-15: Reserved. Must not be selected. 32-bit bus width: 0: WWWWWWWW 1: WWxWWxWWxWWx 2: WWxxWWxxWWxxWWxx 3: WxWxWxWxWxWxWxWx 4: WWxxxWWxxxWWxxxWWxxx 5: WWxxxxWWxxxxWWxxxxWWxxxx 6: WxxWxxWxxWxxWxxWxxWxxWxx 7: WWxxxxxxWWxxxxxxWWxxxxxxWWxxxxxx 8: WxxxWxxxWxxxWxxxWxxxWxxxWxxxWxxx 9-15: Reserved. Must not be selected.
5:7	PClock-to-MasterClock-Ratio. Bit 7 is MSB	0: 2 1: 3 2: 4 3: 5 4: 6 5: 7 6: 8 7: Reserved
8	Endianness	0: Little endian 1: Big endian

Table 12.2 Boot-Time Mode Stream (Part 1 of 2)

Notes

Serial Bit	Description	Value & Mode Setting
9:10	Non-block write Mode. Bit 10 is MSB	00: R4000 compatible 01: Reserved 10: Pipelined-Write-Mode 11: Write-Reissue-Mode
11	TimerIntEn	Timer interrupt settings: 0: Enable Timer Interrupt on Int(5) 1: Disable Timer Interrupt on Int(5)
12	System Interface Bus Width.	Interface bus width control settings: 0: 64-bit system interface 1: 32-bit system interface
13:14	Drv_Out Bit 14 is MSB	Slew rate control of the output drivers: 10: 100% strength (fastest) 11: 83% strength 00: 67% strength 01: 50% strength (slowest)
15:17	Write address to write data delay.	From 0 to 7 MasterClock cycles: 0: AD... 1: AxD... 2: AxxD... 3: AxxxD... 4: AxxxxD... 5: AxxxxxD... 6: AxxxxxxD... 7: AxxxxxxxD...
18	Reserved	User must select ‘0’
19	Extend Multiplication Repeat Rate.	Initial setting of the “Fast Multiply” bit. 0: Enable Fast Multiply 1: Do not Enable Fast Multiply **Note:** For pipeline speeds >250MHz, this bit must be set to ‘1’.
20:24	Reserved	User must select ‘0’
25:26	System configuration identifier.	Software visible in processorConfig[21:20] 0: Config[21:20] = Mode Bit [25:26]
27:256	Reserved	User must select ‘0’

Table 12.2 Boot-Time Mode Stream (Part 2 of 2)

Notes

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.

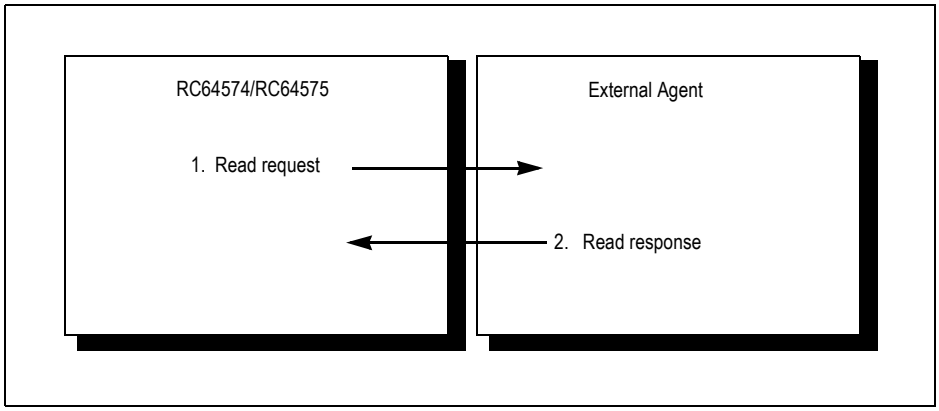


Figure 13.1 Read Response

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: NCRProcessor noncoherent block read request NCR/WProcessor noncoherent block read request followed by processor block write request NCWProcessor 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 transfers 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 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

.

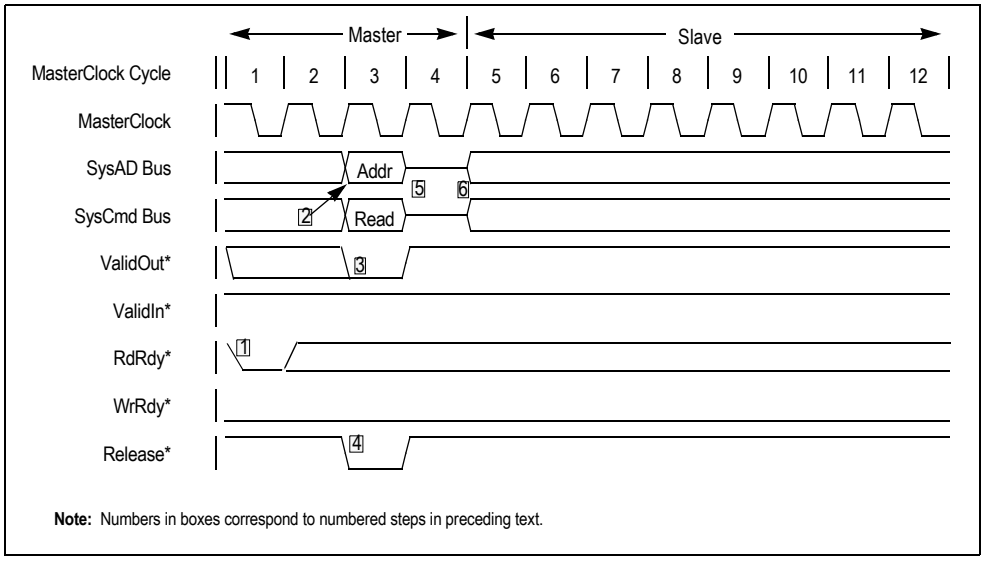


Figure 13.2 Processor Read Request Protocol

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 RC64574/RC64575 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: 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 RC64574/RC64575 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: 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.

Notes

Example of Data Single Read with Zero Wait-state

Table 13.5 shows a data 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	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 Figure 13.3. For a larger “multiply-by” it will take longer to get the response data.

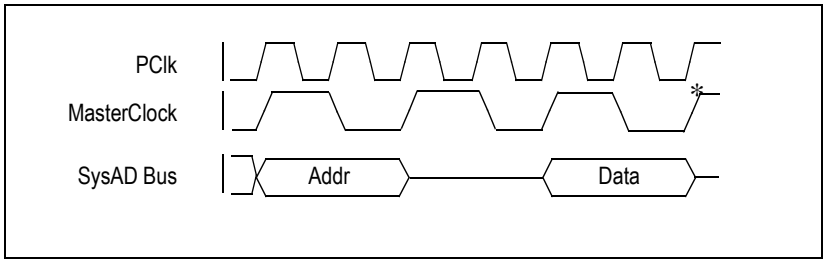


Figure 13.3 Uncached Read—External Cycles

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.*

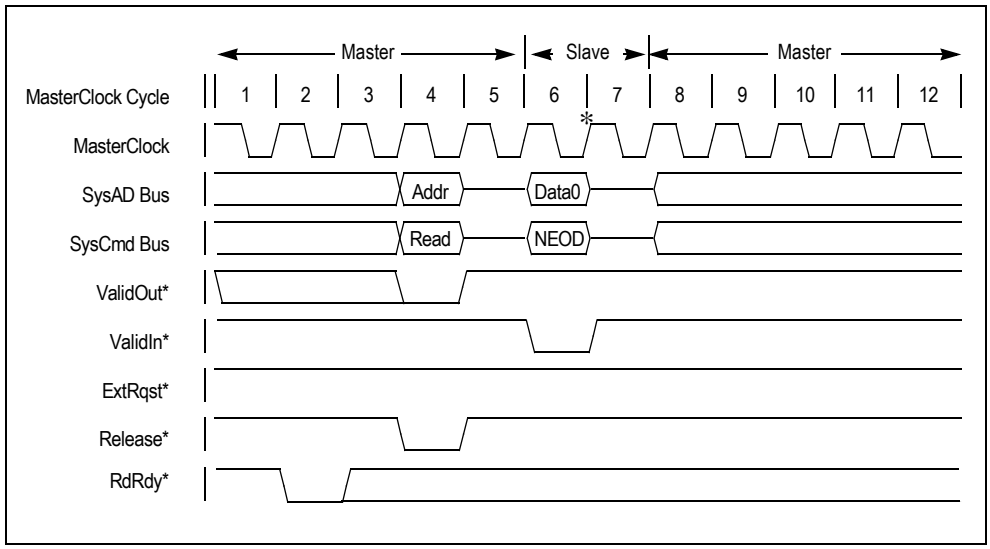


Figure 13.4 Processor Read Cycle

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 RC64574/RC64575,) 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.

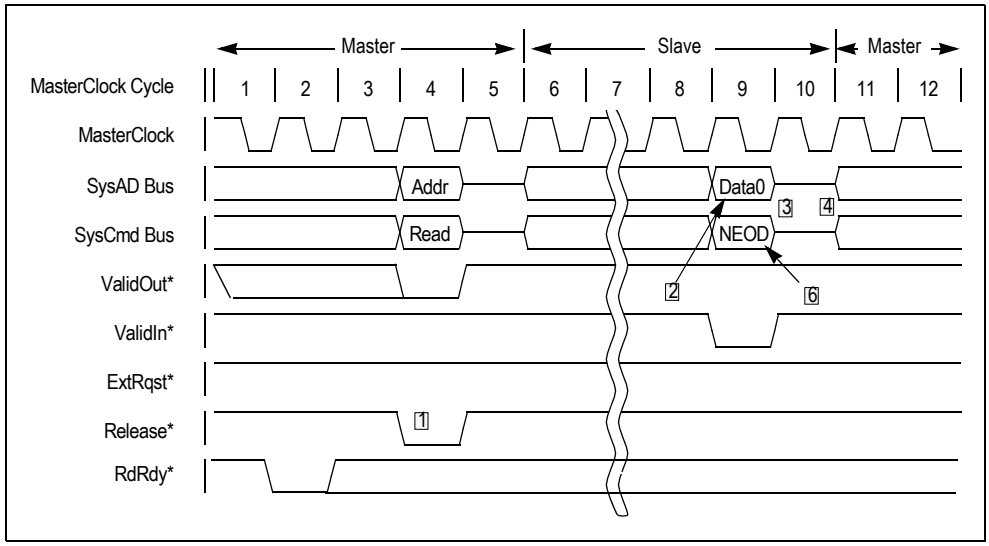


Figure 13.5 Processor Word Read Request Followed by a Word Read Response (64-bit bus interface)

Notes

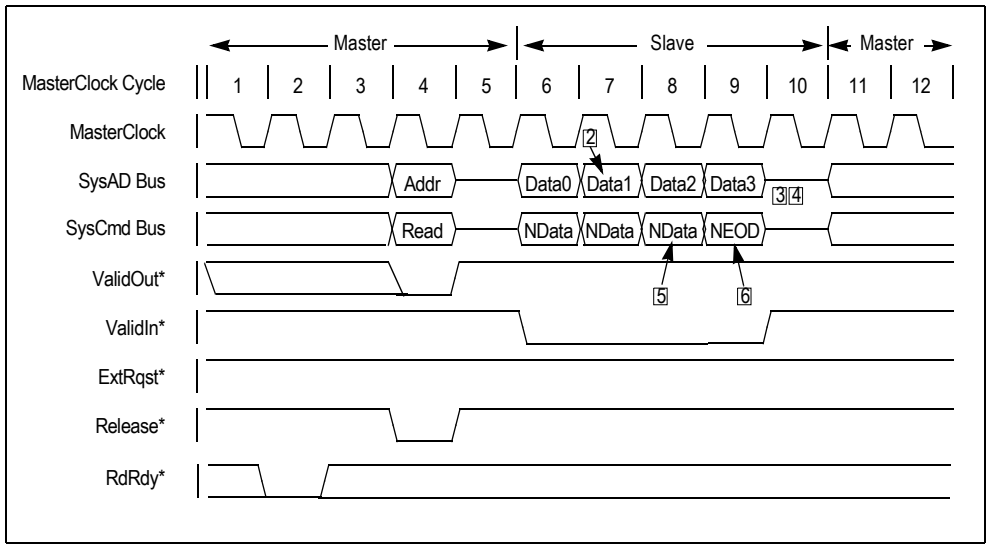


Figure 13.6 Block Read Response with Zero Wait-State (64-bit bus interface)

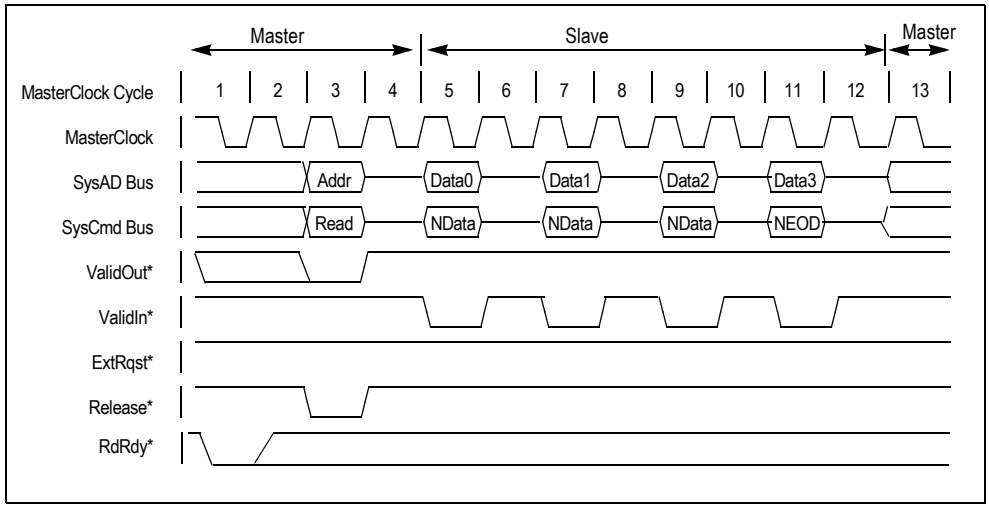


Figure 13.7 Block Read Transaction with One Wait-State (64-bit bus interface)

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-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.

Notes

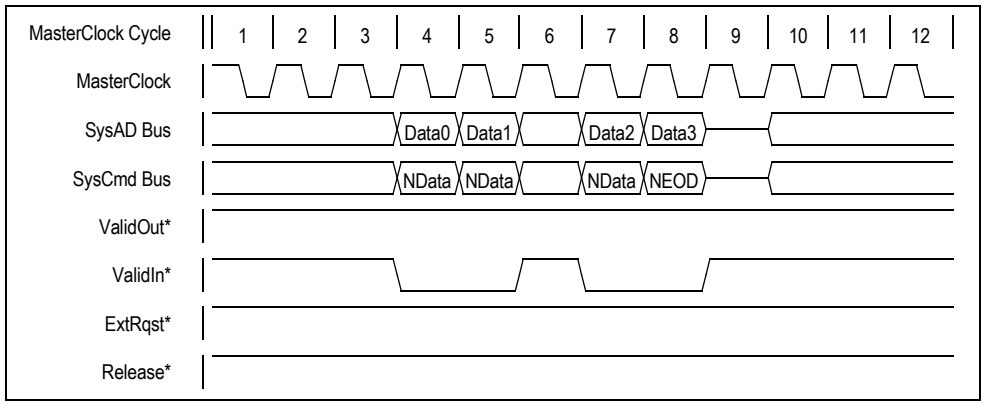


Figure 13.8 Read Response, Reduced Data Rate, System Interface in Slave State (64-bit bus interface)

64-bit and 32-bit Bus Modes

The bus interface of the RC64575 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 RC64575 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 RC64575 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 RC64575 is double word (64-bit) aligned.

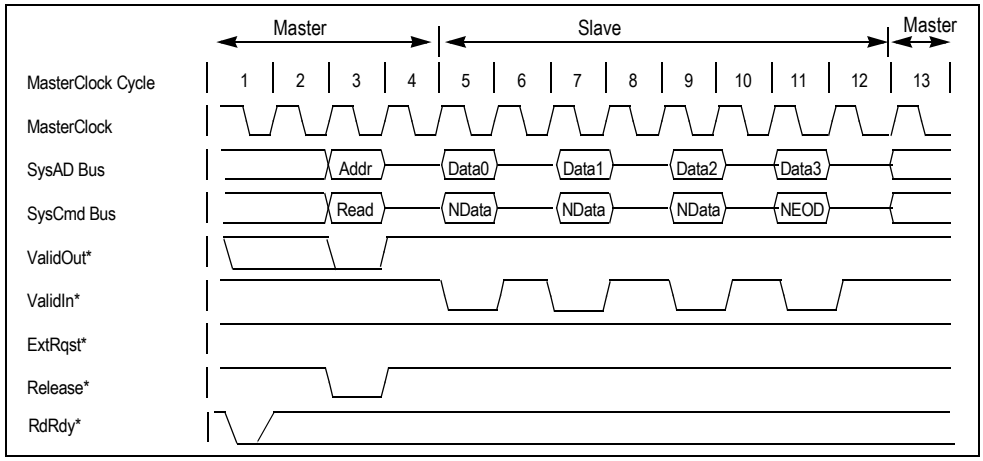


Figure 13.9 Block Read Transaction with One Wait-State

Notes

64-bit Bus Mode Single (Uncached) Read Operation

In 64-bit bus mode, the RC64575 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.

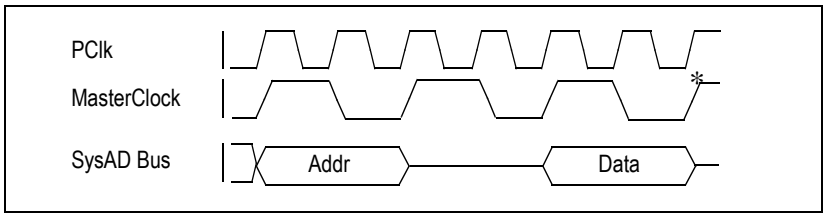


Figure 13.10 64-Bit Uncached Read—External Cycles

32-bit Bus Mode

In 32-bit bus mode, the RC64574/RC64575 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 RC64574/RC64575 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 RC64574/RC64575 in either 64-bit or 32-bit bus mode.

32-bit Bus Mode Block Read Operation

In 32-bit bus mode, the RC64574 or RC64575 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 RC64574 or RC64575.

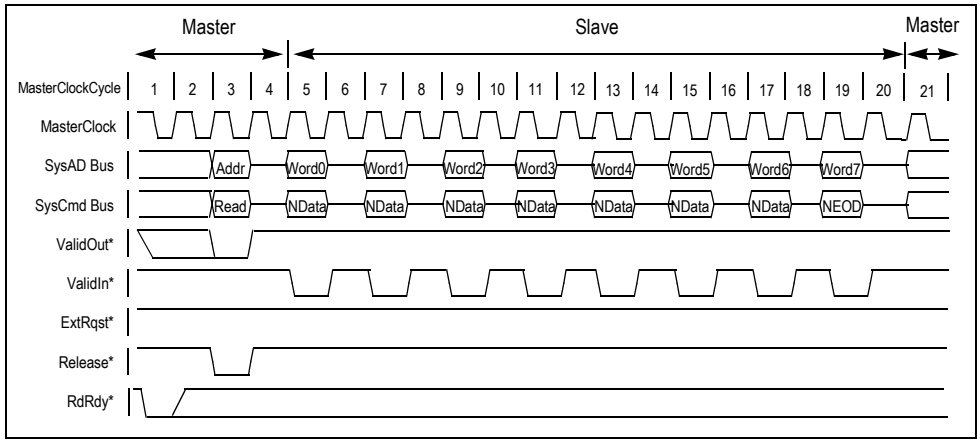


Figure 13.11 Block Read Transaction with One Wait-State

Notes

The RC64574/RC64575 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 RC64575 or the RC64574 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.

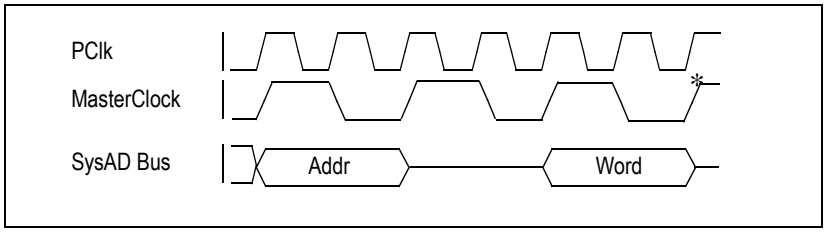


Figure 13.12 32-bit Bus Mode Uncached Read for Single Word

Figure 13.13 illustrates the timing diagram for an uncached read operation of a double word value.

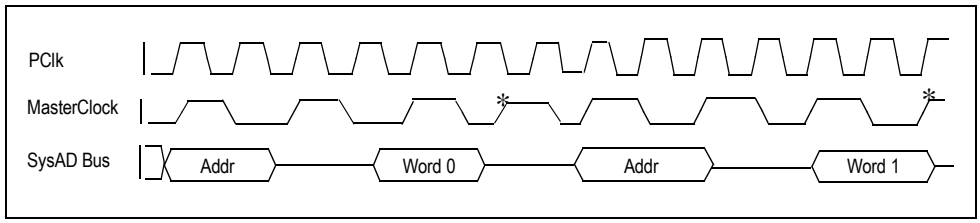


Figure 13.13 32-bit Bus Mode Uncached Read for Double Word

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.

Notes

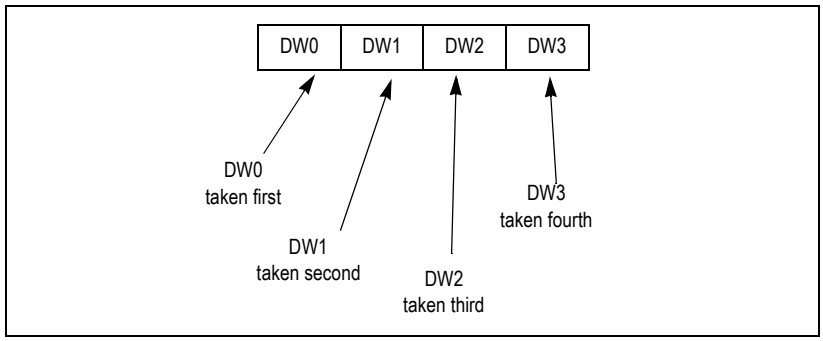


Figure 13.14 Retrieving a Data Block in Sequential Order

Subblock Ordering Example

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).

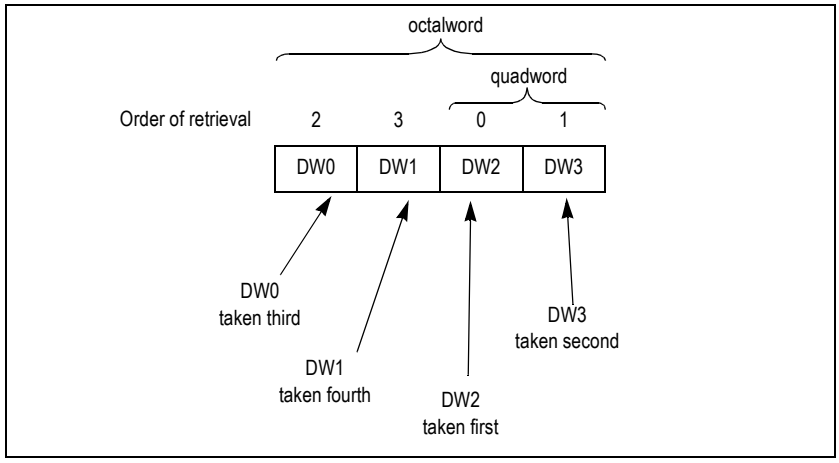


Figure 13.15 Retrieving Data in a Subblock Order

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.

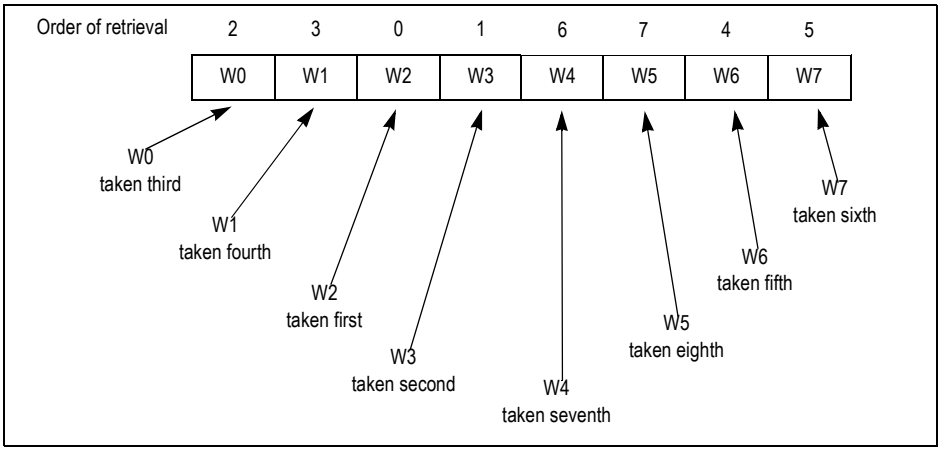


Figure 13.16 Retrieving Data in a Subblock Order

Notes

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).

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.

Notes

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

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 and 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 RC64574/RC64575 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.

Notes

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.

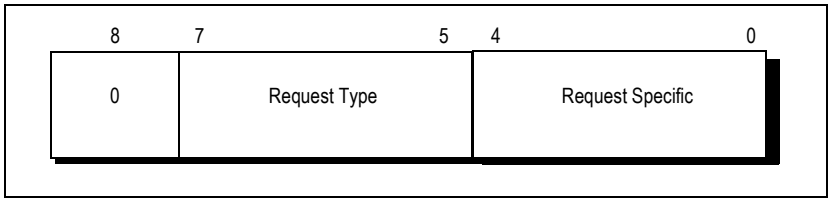


Figure 13.17 System Interface Command Syntax Bit Definition

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.

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.

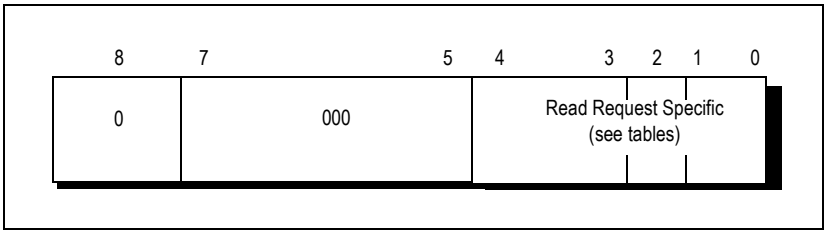


Figure 13.18 Read Request SysCmd Bus Bit Definition

Tables 13.12, 13.13, and 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

Notes

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

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.

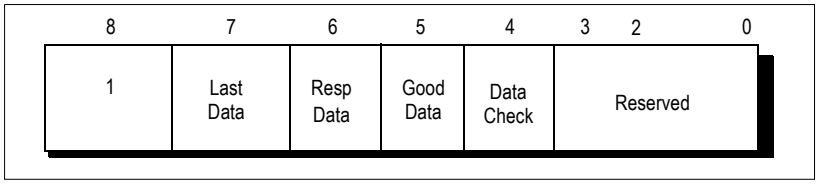


Figure 13.19 Data Identifier SysCmd Bus Bit Definition

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*

Notes

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.

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)

Notes

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.

# Bytes	Address	SysAD Byte Lanes Used (Big Endian)							
SysCmd (2:0)	Mod 8	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
1 (000)	0	•							
	1		•						
	2			•					
	3				•				
	4					•			
	5						•		
	6							•	
	7								•
2 (001)	0	•	•						
	2			•	•				
	4					•	•		
	6							•	•
3 (010)	0	•	•	•					
	1		•	•	•				
	4					•	•	•	
	5						•	•	•
4 (011)	0	•	•	•	•				
	4					•	•	•	•
5 (100)	0	•	•	•	•	•			
	3				•	•	•	•	•
6 (101)	0	•	•	•	•	•	•		
	2			•	•	•	•	•	•
7 (110)	0	•	•	•	•	•	•	•	
	1		•	•	•	•	•	•	•
8 (111)	0	•	•	•	•	•	•	•	•
		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

Notes

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 13.18 shows the byte lanes used for partial word transfers for both little and big endian in 32-bit bus mode.

# Bytes	Address	SysAD Byte Lanes Used (Big Endian)			
SysCmd(2:0)	Mod 4	31:24	23:16	15:8	7:0
1 (000)	0	•			
	1		•		
	2			•	
	3				•
2 (001)	0	•	•		
	2			•	•
3 (010)	0	•	•	•	
	1		•	•	•
4 (011)	0	•	•	•	•
		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

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).

> **Note:** The bus interface of the RC64574 and RC64575 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 RC64574/RC46575 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 RC64574/RC64575 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

Notes

width interface. The selection of the write protocol is static, which means that it should be selected once 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.

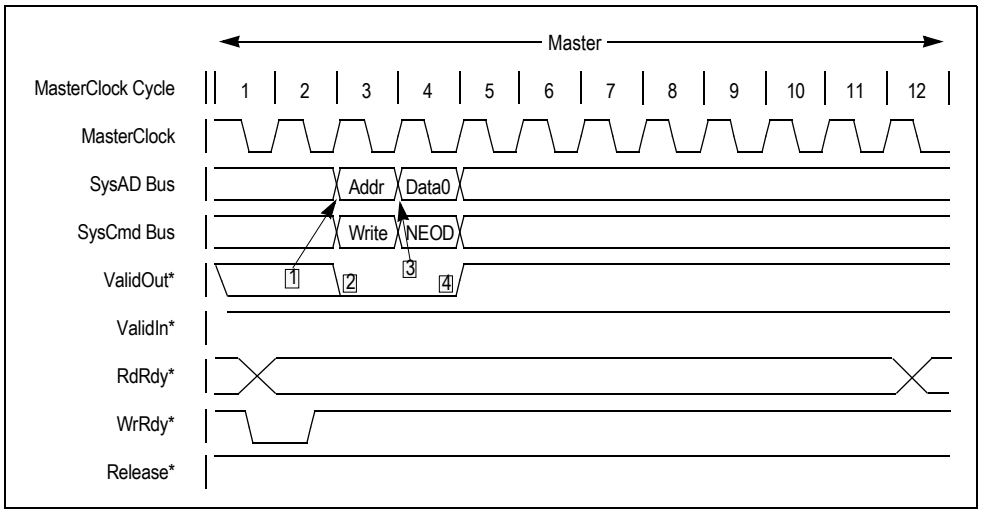


Figure 14.1 Processor Noncoherent Word Write Request Protocol

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

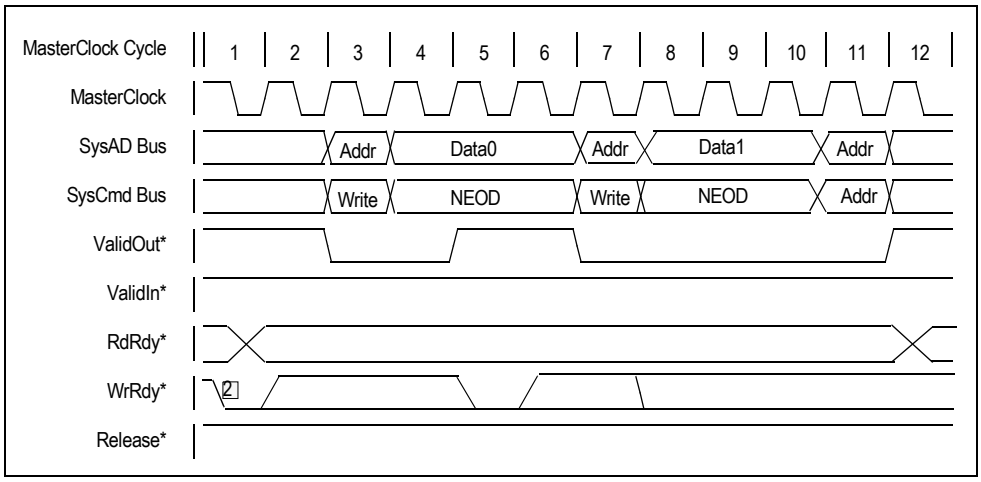


Figure 14.2 R4000 Compatible Write Mode

The RC64574/RC64575 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 RC64574/RC64575 provide two 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 Figure 14.3 because **WrRdy*** is sampled LOW at *3 and at *4.*

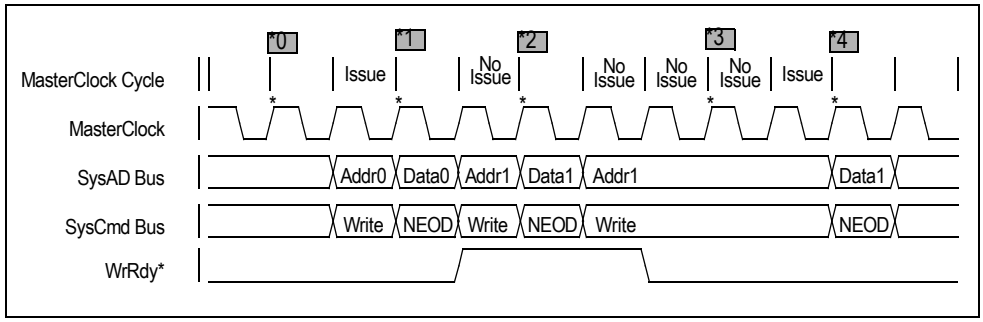


Figure 14.3 Write Reissue

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.*

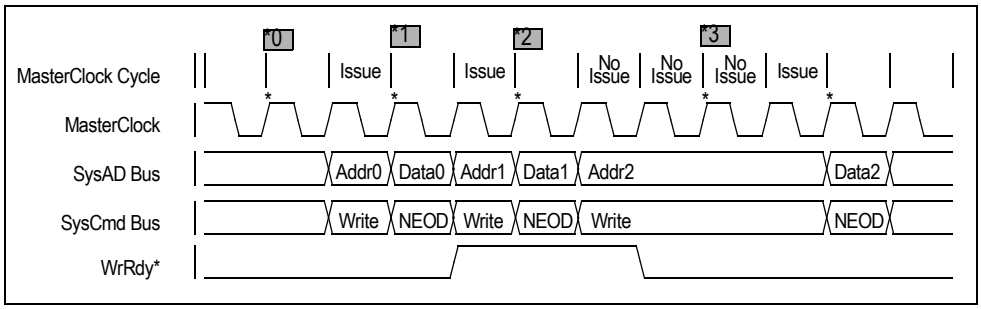


Figure 14.4 Pipelined Writes

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.

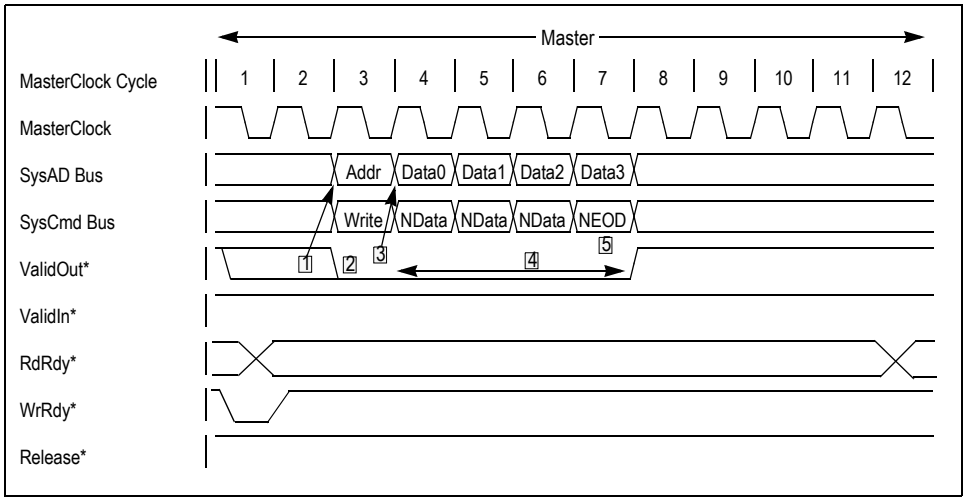


Figure 14.5 Processor Noncoherent Block Write Request Protocol

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	WxxWxxWxxWxxWxxWxxWxx
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 and 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.

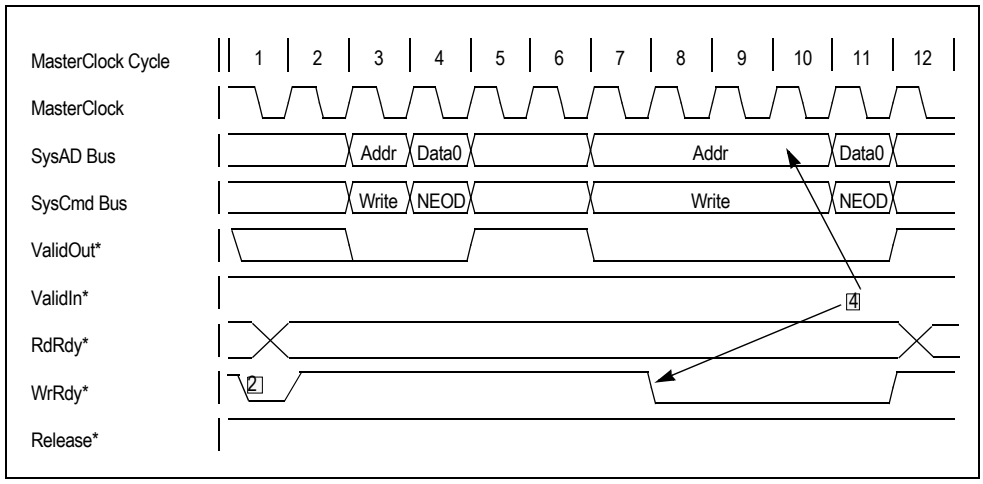


Figure 14.6 Two Processor Write Requests, Second Write Delayed for the Assertion of WrRdy*

64-bit and 32-bit Bus Modes

The bus interface of the RC64575 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 RC64575 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 RC64575 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 RC64575 is double word (64-bit) aligned.

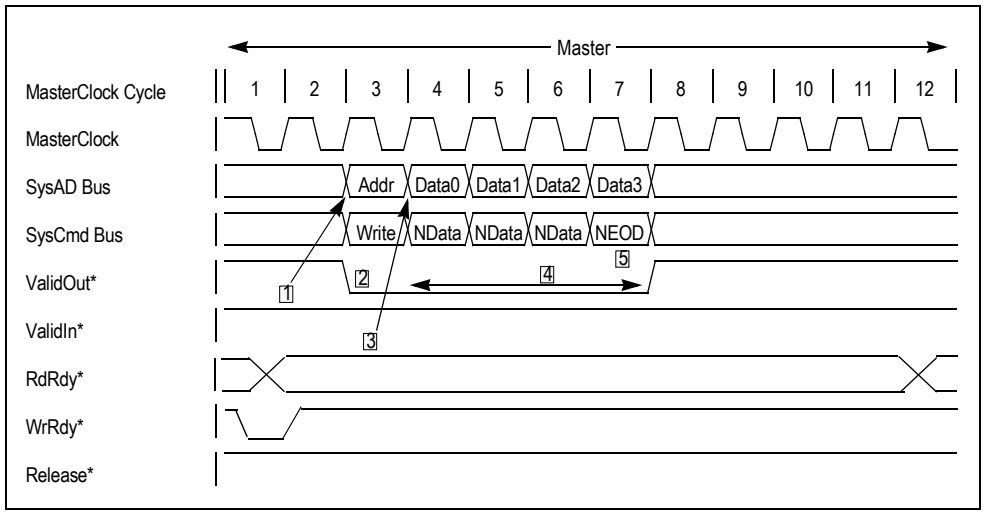


Figure 14.7 Processor Noncoherent Block Write Request Protocol

64-bit Bus Mode Single (Uncached) Write Operation

In 64-bit bus mode, the RC64575 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.

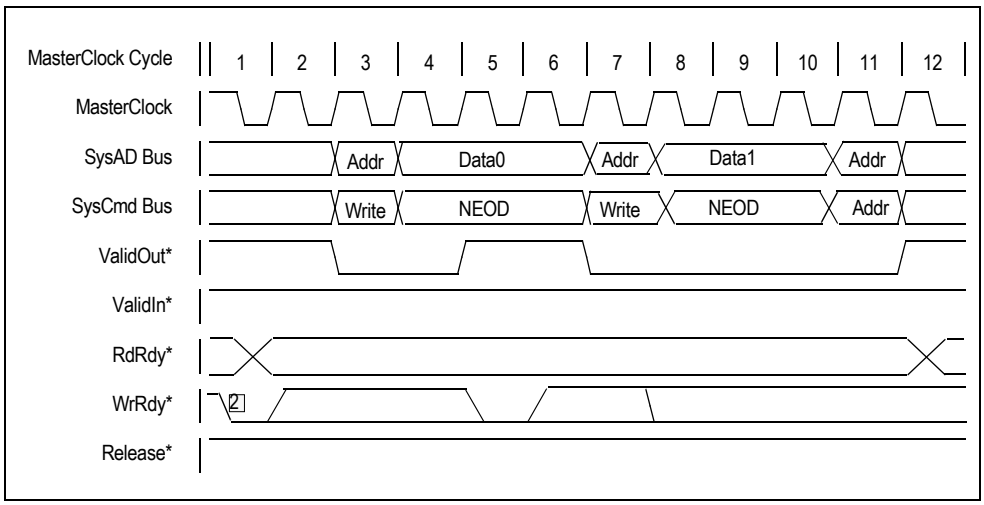


Figure 14.8 R4000 Family Compatible Write Mode

Notes

The RC64574/RC64575 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.*

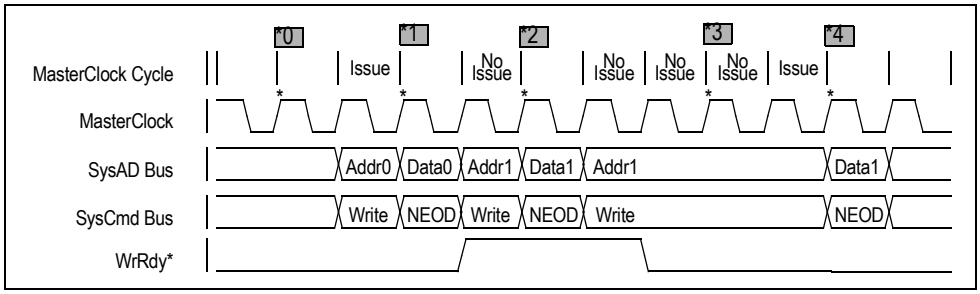


Figure 14.9 Write Reissue

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.*

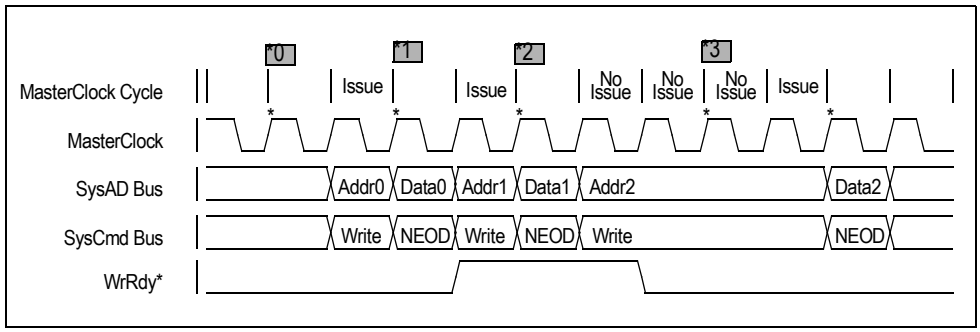


Figure 14.10 Pipelined Writes

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 RC64574 or RC64575 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 RC64574 or RC64575 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 RC64574/RC64575.

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.

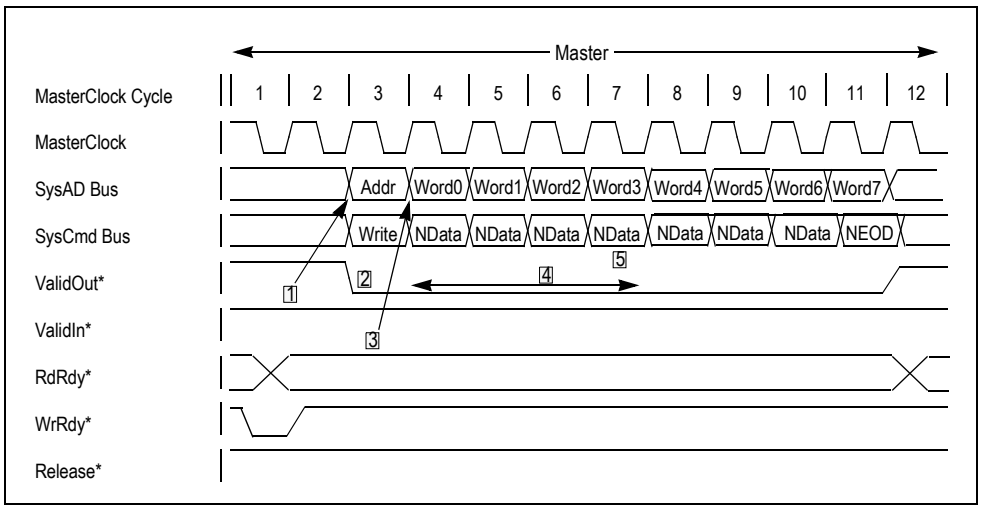


Figure 14.11 Processor Noncoherent Block Write Request Protocol

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 RC64574 or RC64575 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

Notes

requests. The first request will transfer 4 bytes and the second will transfer up to 4 bytes. 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, 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.

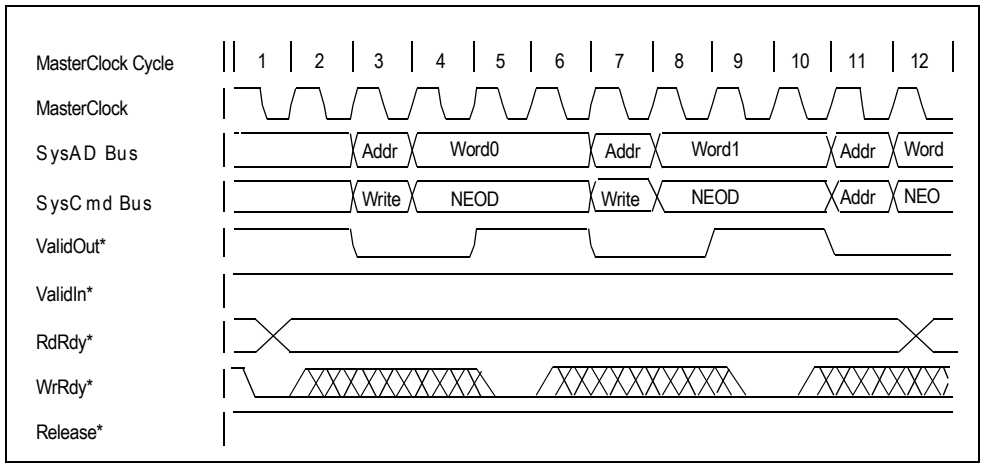


Figure 14.12 R4000 Family Compatible Write Protocol

The RC64574/RC64575 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.*

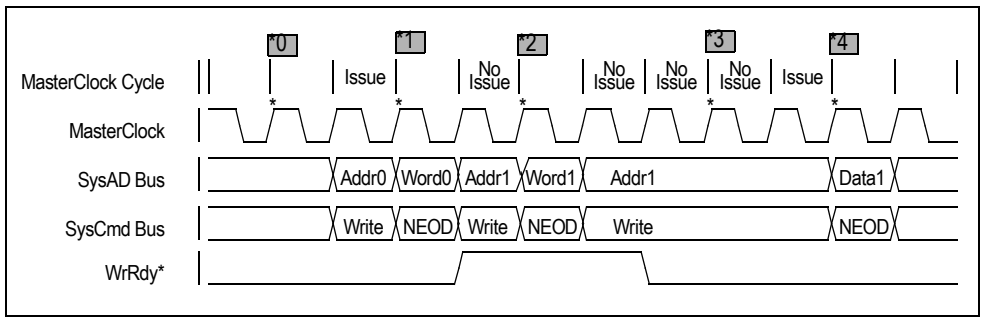


Figure 14.13 Write Reissue

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.*

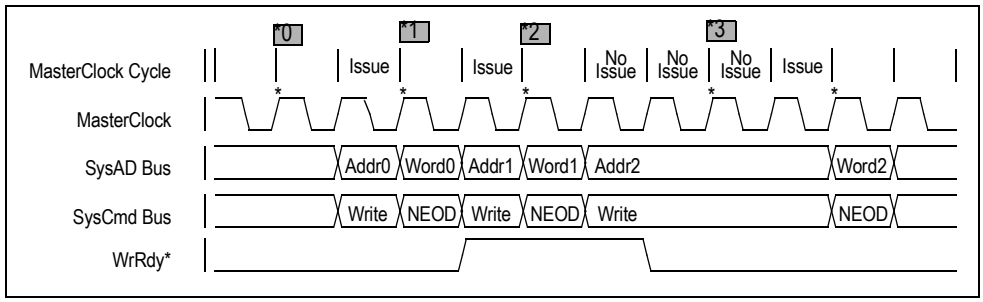


Figure 14.14 Pipelined Writes

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

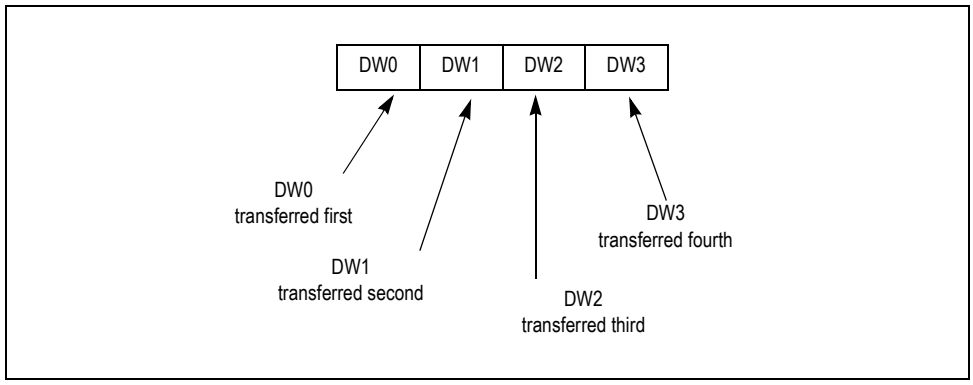


Figure 14.15 Transferring a Data Block in Sequential Order

Figure 14.16 shows a sequential order in which Word0 (W0) is transferred first and Word 7 (W7) is transferred last.

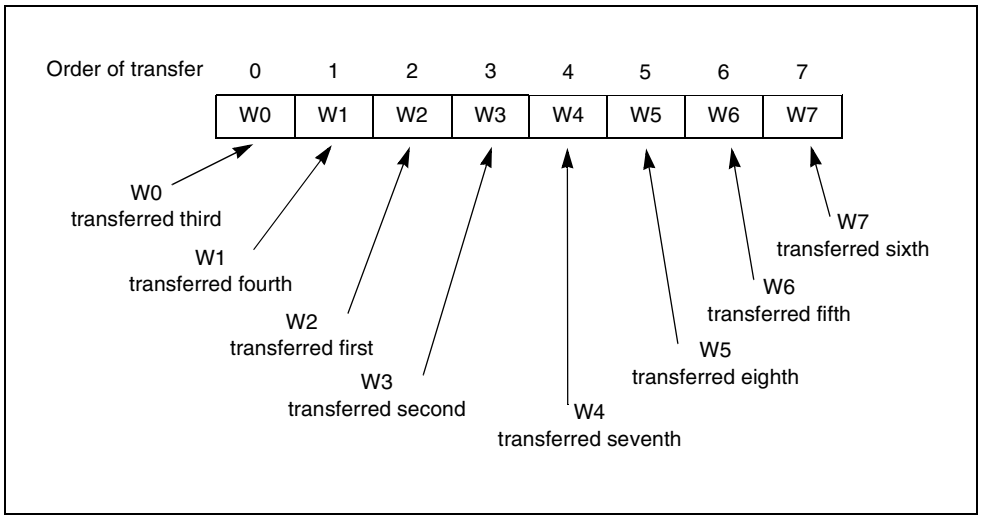


Figure 14.16 Transferring Data in a Subblock Order

Table 14.4 shows the byte lanes used for 64-bit bus mode partial word transfers for both little and big endian.

Notes

# Bytes	Address	SysAD byte lanes used (big endian)							
SysCmd(2:0)	Mod 8	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
1 (000)	0	•							
	1		•						
	2			•					
	3				•				
	4					•			
	5						•		
	6							•	
	7								•
2 (001)	0	•	•						
	2			•	•				
	4					•	•		
	6							•	•
3 (010)	0	•	•	•					
	1		•	•	•				
	4					•	•	•	
	5						•	•	•
4 (011)	0	•	•	•	•				
	4					•	•	•	•
5 (100)	0	•	•	•	•				
	3				•	•	•	•	•
6 (101)	0	•	•	•	•	•	•		
	2			•	•	•	•	•	•
7 (110)	0	•	•	•	•	•	•	•	
	1		•	•	•	•	•	•	•
8 (111)	0	•	•	•	•	•	•	•	•
		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	Address	SysAD Byte Lanes Used (Big Endian)			
SysCmd(2:0)	Mod 4	31:24	23:16	15:8	7:0
1 (000)	0	•			
	1		•		
	2			•	
	3				•
2 (001)	0	•	•		
	2			•	•
3 (010)	0	•	•	•	
	1		•	•	•
4 (011)	0	•	•	•	•
		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 and 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 RC64574/RC64575 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.

Notes

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 14.17 shows a common encoding used for all system interface commands.

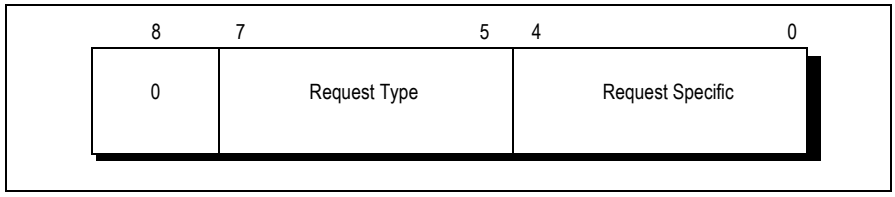


Figure 14.17 System Interface Command Syntax Bit Definition

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.

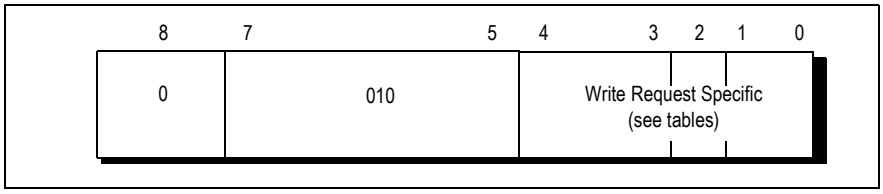


Figure 14.18 Write Request SysCmd Bus Bit Definition

Table 14.7 lists the write attributes encoded in bits **SysCmd(4:3)**.

Notes

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)

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.

Notes

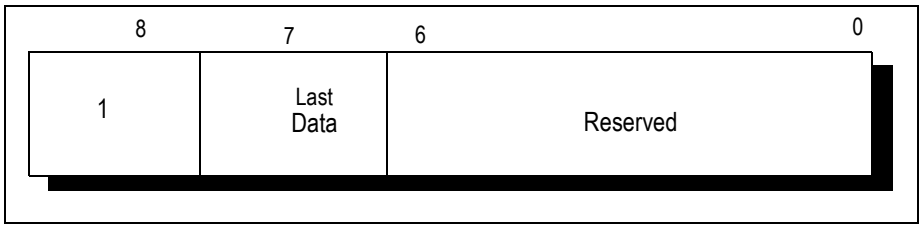


Figure 14.19 Data Identifier SysCmd Bus Bit Definition

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)

Notes

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.

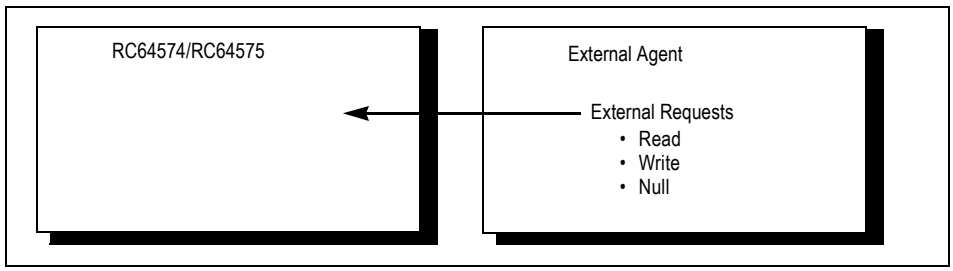


Figure 15.1 External Requests

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.

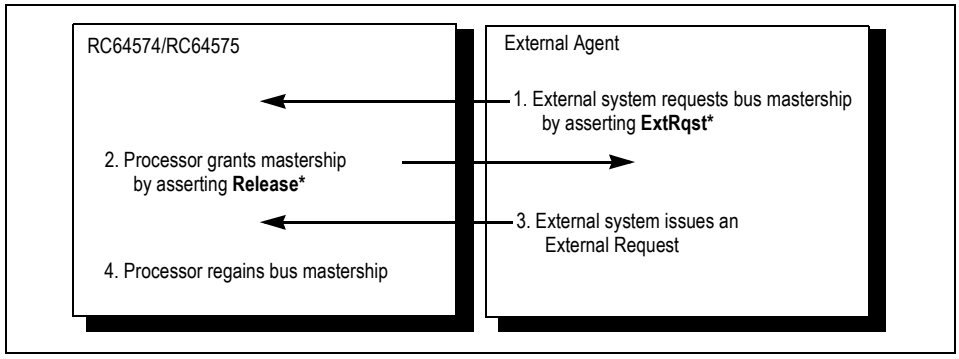


Figure 15.2 Processor Control of External Request, through Arbitration Signals

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***.

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.*

Notes

- *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 RC64574/RC64575 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.

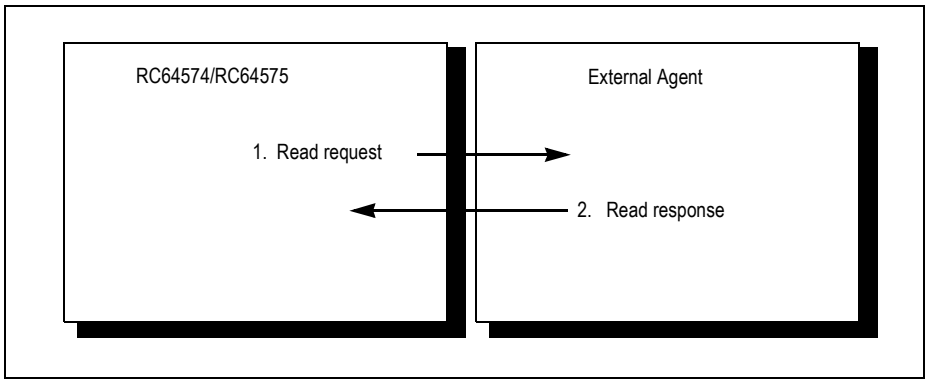


Figure 15.3 Read Response

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.

Notes

6. The external agent sets the **SysAD** and the **SysCmd** buses to tri-state at the completion of an external request.

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.

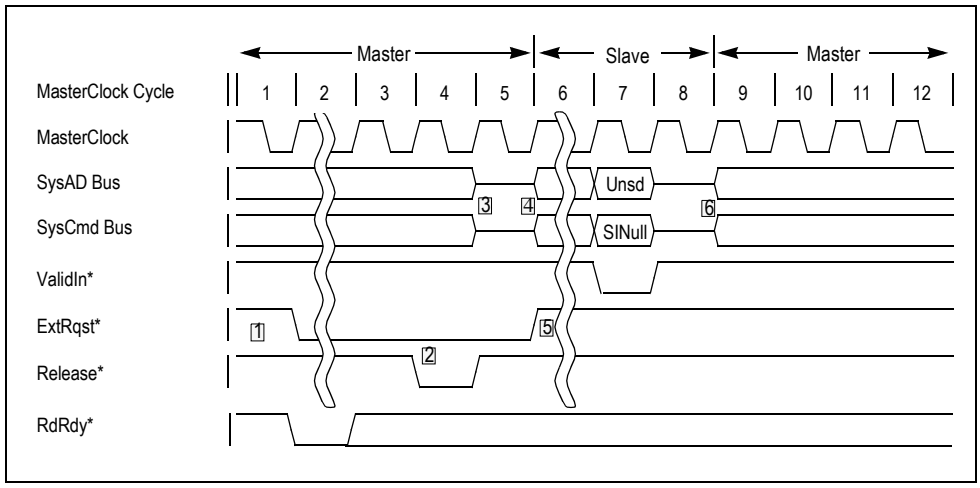


Figure 15.4 Arbitration Protocol for External Requests

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

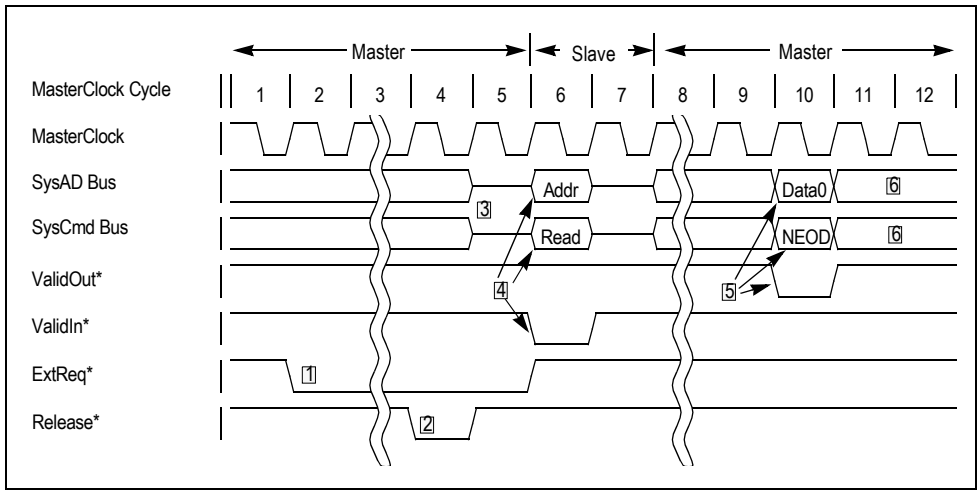


Figure 15.5 External Read Request, System Interface in Master State

External Null Request Protocol

The RC64574/RC64575 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.

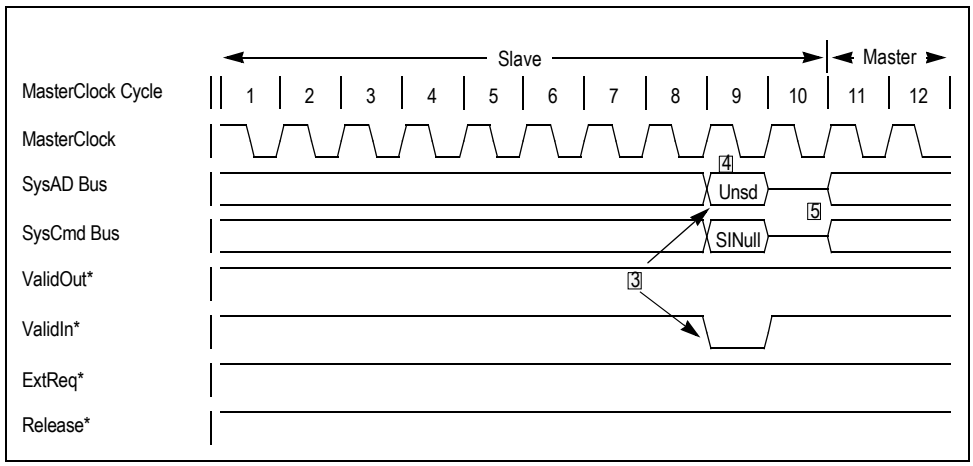


Figure 15.6 System Interface Release External Null Request

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 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.

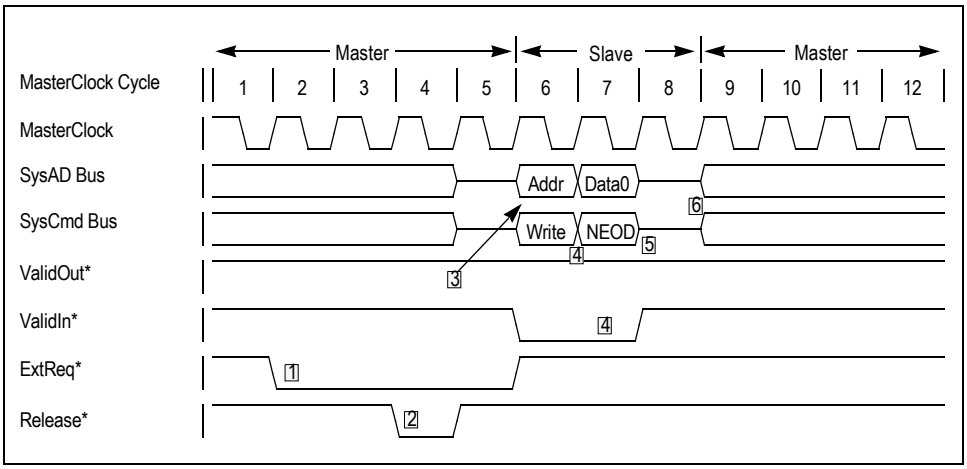


Figure 15.7 External Write Request, with System Interface Initially in Master State

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 13, "The Read Interface."

Interface Commands and 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 RC64574/RC64575 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.

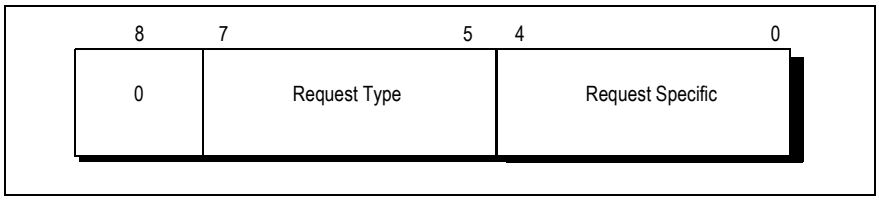


Figure 15.8 System Interface Command Syntax Bit Definition

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.

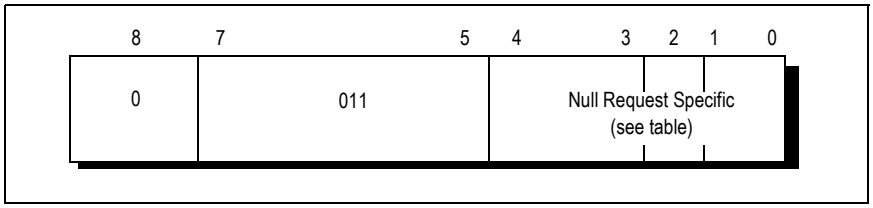


Figure 15.9 Null Request SysCmd Bus Bit Definition

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.

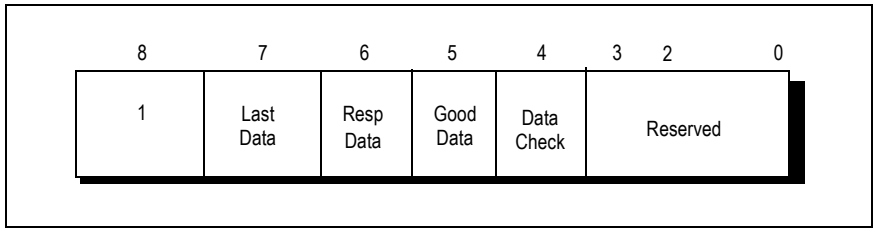


Figure 15.10 Data Identifier SysCmd Bus Bit Definition

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.*

Notes

- *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.*

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 RC64574/RC64575 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 16, “Processor Interrupts.”

Processor Interrupts

Notes

Introduction

The RC64574/575 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 6. CPU exception processing is also described in Chapter 6. Floating-point exception processing is described in Chapter 8.

The six CPU hardware interrupts can either be caused by external write requests to the RC64574/575 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 RC64574/575 or by a dedicated pin in the RC64574/575. 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.

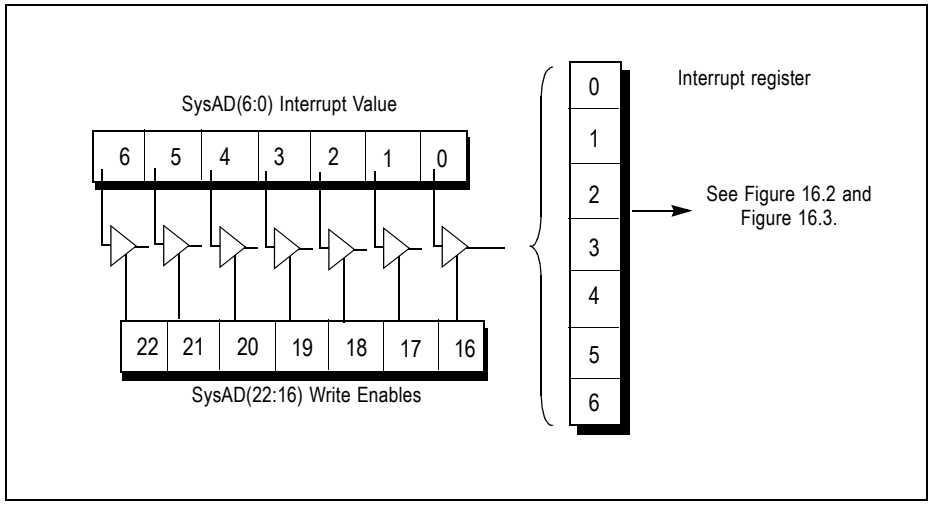


Figure 16.1 Interrupt Register Bits and Enables

Figure 16.2 shows how the RC64574/RC64575 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 RC64574/RC64575 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

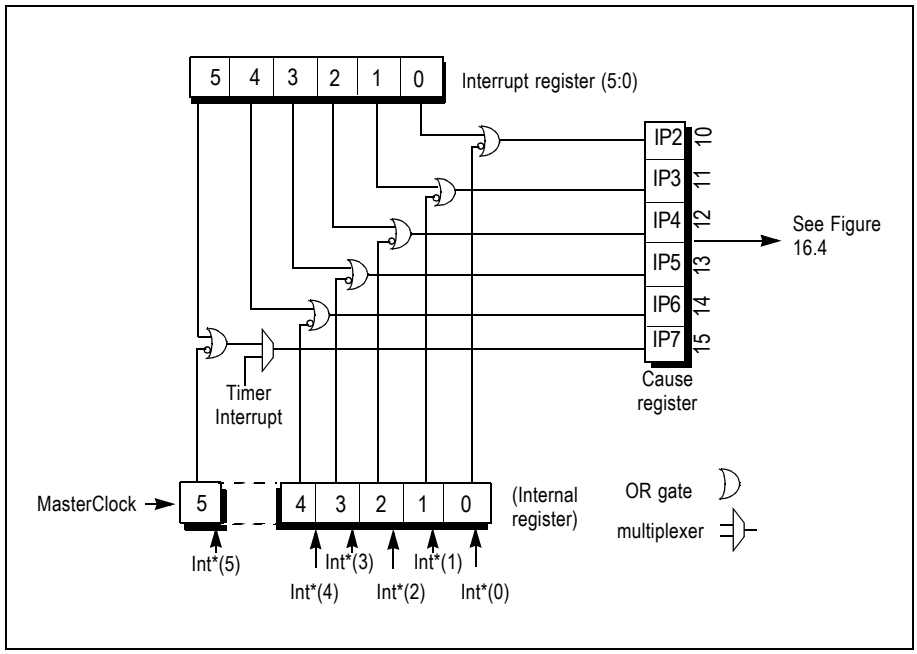


Figure 16.2 RC64574/RC64575 Interrupt Signals

Figure 16.3 shows the internal derivation of the nonmaskable (**NMI)** signal, for the RC64574/RC64575 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.

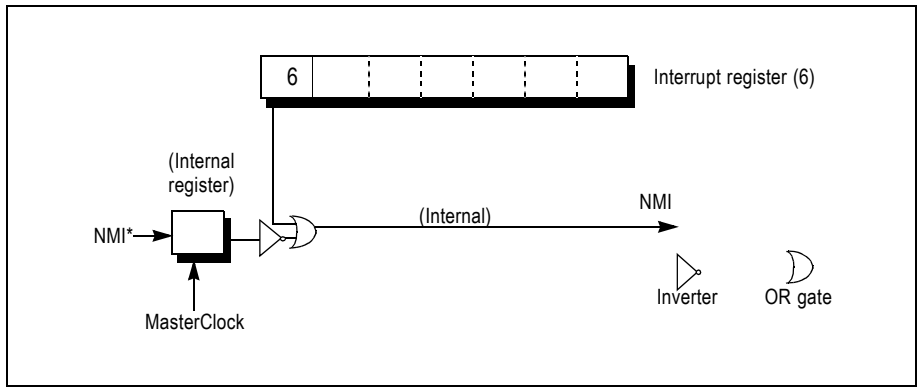


Figure 16.3 RC64574/RC64575 Nonmaskable Interrupt Signal

Figure 16.4 shows the masking of the RC64574/RC64575 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 RC64574/RC64575 interrupt signal.*

Notes

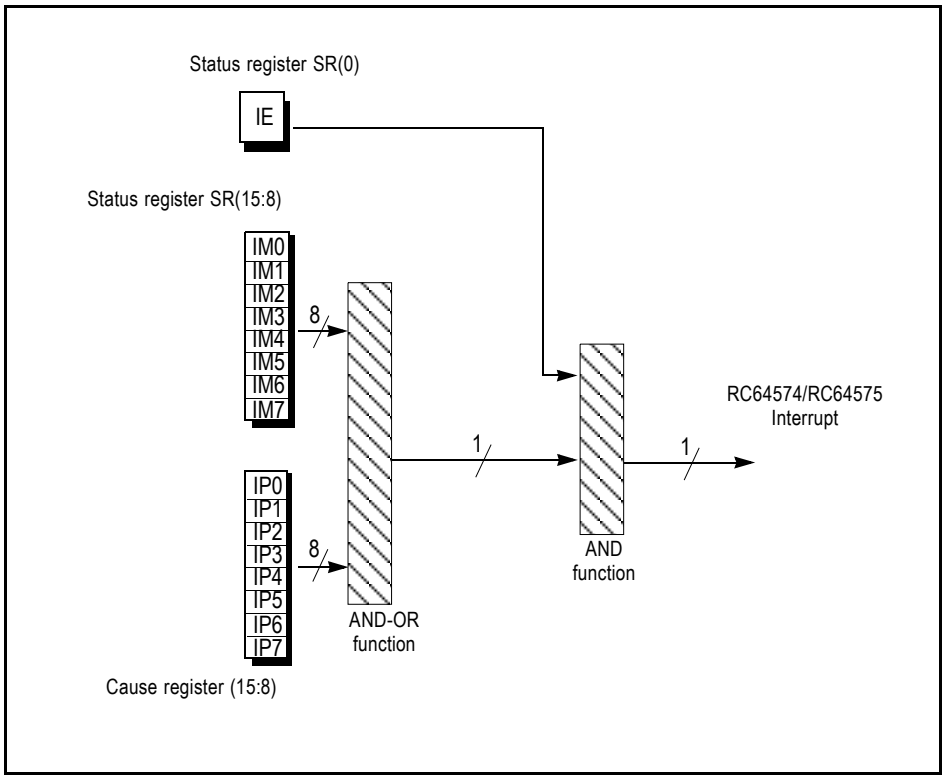


Figure 16.4 Masking of the RC64574/RC64575 Interrupts

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 RC64574 and RC64575 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 RC64574/RC64575 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 RC64574/RC64575 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 RC64574/RC64575 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 Interface 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 RC64574/RC64575, the following six additional pins—TDI, TDO, TMS, TCK, TRST* and JTAG32—have been added to support the implementation of this interface. Table 18.1 gives a description of the JTAG interface pins.

Note: The JTAG boundary scan should only be used when the RC64574/RC64575 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 unconnected.
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

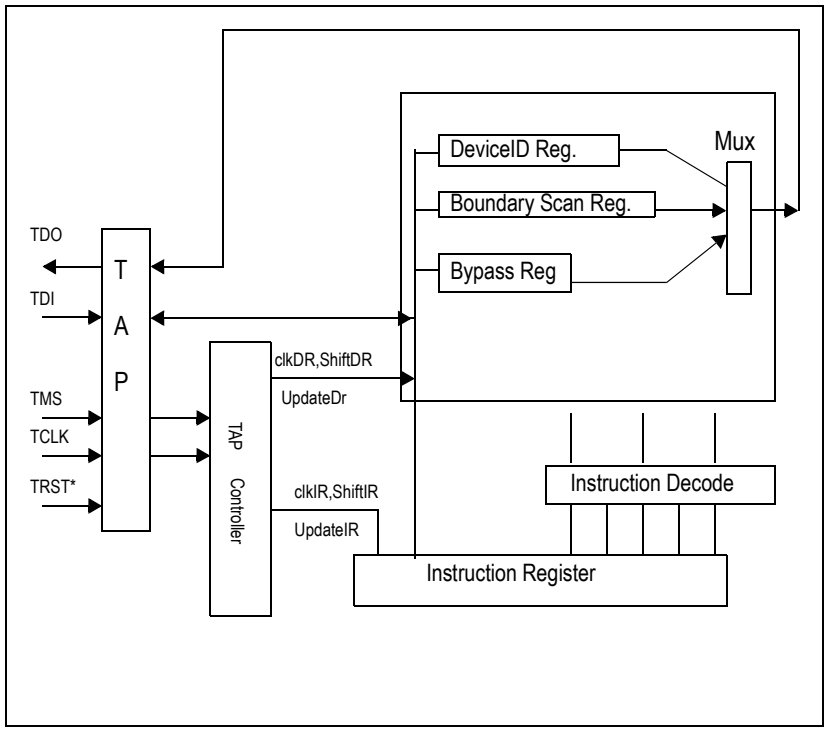


Figure 18.1 Standard Boundary Scan Architecture

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.

Notes

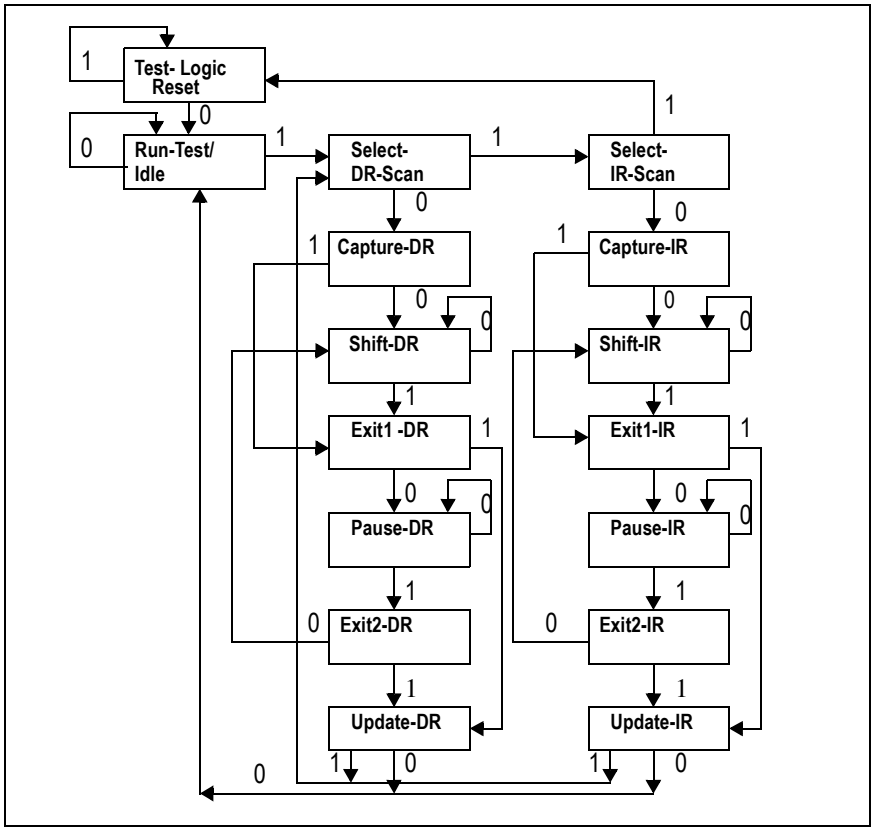


Figure 18.2 TAP Controller State Diagram

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

Notes

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.
- *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 shown in Table 18.2:

Notes

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.

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, 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.

Notes

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 RC64574/RC64575. 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. Refer to Figure 18.3.

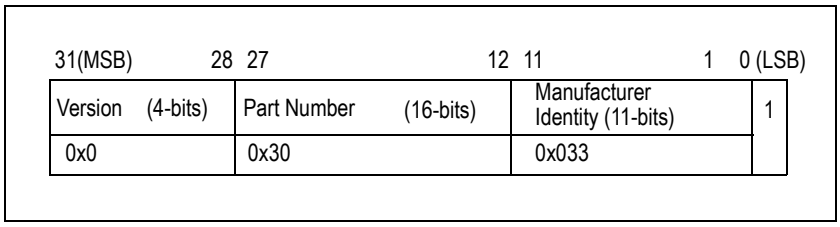


Figure 18.3 Device Identification Register Format

Cache Operations' Timing

Notes

Introduction

This appendix lists cycle operation counts and caveats for RC64574/RC64575 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.

Notes

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.

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.

Standby Mode Operation

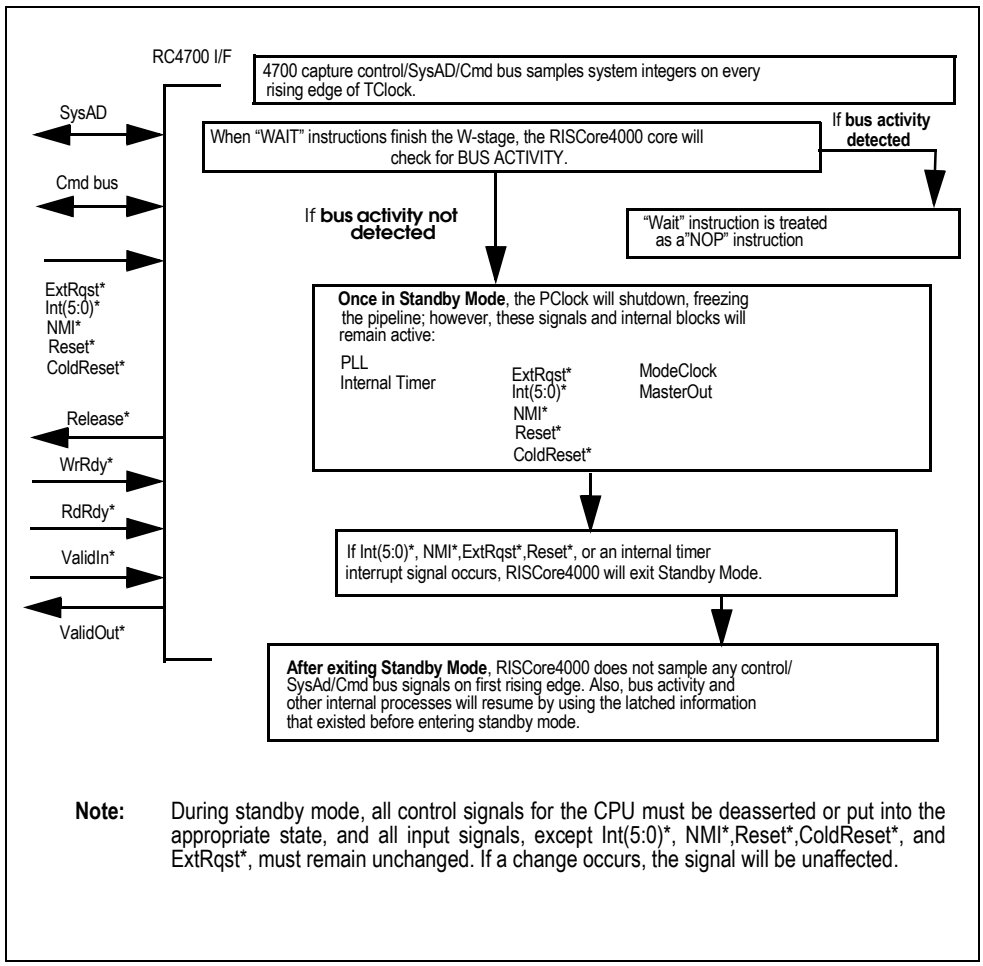
Notes

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 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 RC64574/575 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		DESTINATION	
	Name	Stage	Name	Stage
MTC0	gpr rt	2(A)	cpr rd	4(W)[1]
MFC0	cpr rd	2(A)	gpr rt	4(W)[1]
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)[2]
TLBP	PageMask, EntryHi	2(A)	Index	4(W)
ERET	EPC or ErrorEPC, Status.ERL	2(A)	Status.EXL, Status.ERL	4(W)[3]
			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)[3]		
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)

Table C.1 Coprocessor 0 Hazards

1. There must be at least one instruction between a MTC0 and a MFC0.

2. TLBW_ instructions will cause a one cycle slip.

3. Instructions 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.

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

S

T

U

V

W

X

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.