Quantum Effect Devices™

RM7000™ Family

User Manual
Revision 2.0

Document Changes

Listed below are the major changes to the material.

1. Changed Section 5, “Memory Management Unit” include a description of the options for either a 48 even/odd entry JTLB or a 64 even/odd entry JTLB.
2. Changed section 5.4.2.8 on page 45 for a modification to include the CLK status bit for the clock multiplier state.
3. Changed Table 10.2: on page 113 to include the mode bit description for the JTLB size options.
4. Changed section 7.7, “Cache Maintenance” on page 92 to reflect the additional use of the “hint” field of the PREF instruction.
5. Changed Table 17.3:, “PerfControl Register Fields,” on page 244 for the new types of events supported.
6. Added section 10.4.1, “Processor Revision Identifier (PRId) Register (Set 0, Register 15)” on page 115 to this chapter
7. Added section 10.4.2, “Config Register (Set 0, Register 16)” on page 115 to this chapter
8. Added section 10.4.3, “Status Register (Set 0, Register 12)” on page 117 to this chapter
9. Added section 10.4.4, “Info Register (Set 0, Register 7)” on page 120 to describe the new Info register.
10. Added section 13.2.6, “Info Register (Set 0, Register 7)” on page 187 to describe the new Info register.
11. Added Info Register row to Table A.1: on page 271.
12. Added Appendix F on page 291 to describe enhanced features for revision 2 silicon.
13. Added Table 18.5: on page 269 to describe the new **PREF** instruction features.

Table of Contents

List of Figures

List of Tables

Section 1 Introduction

The QED RM7000 is a highly integrated superscalar microprocessor capable of issuing two instructions per clock cycle. It has two 64-bit integer units and a fully-pipelined 64-bit floating-point unit. The RM7000 contains separate 16 Kbyte, 4-way set associative primary instruction and data caches, as well as a 256 Kbyte 4-way set associative unified secondary cache. The RM7000 implements a non-blocking caching scheme, supports cache locking on a per-line basis for both the primary and secondary caches, and provides an R5000-compatible dedicated tertiary cache interface.

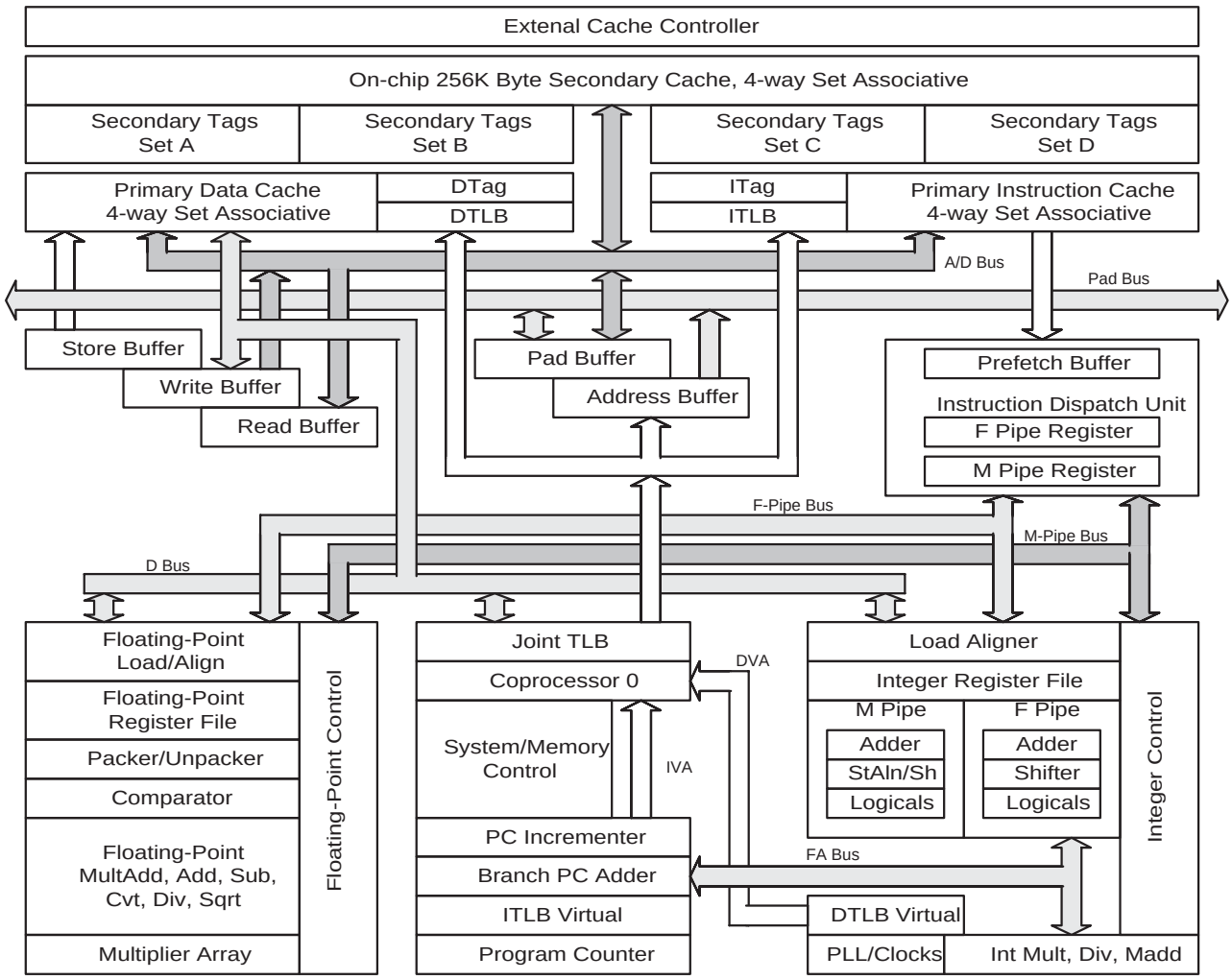


Figure 1.1 RM7000 Block Diagram

Figure 1.1 shows a block diagram of the RM7000 processor. The outlined blocks are abbreviated as follows:

- DCache - Primary Data Cache
- ICache - Primary Instruction Cache
- SCache - Secondary Cache
- BIU - Bus Interface Unit
- SSD - Superscaler Dispatch Unit
- FPU - Floating-Point Unit

- MMU - Memory Management Unit
- IU - Integer Unit

A non-pendant bus mode allows for multiple outstanding bus transactions that can be completed out-of-order. An enhanced interrupt handling mechanism allows for the addition of four interrupt pins as well as three new interrupt control and priority registers.

A second set of thirty-two CP0 registers (CP0_Set 1) has been added to accommodate the enhanced interrupt handling mechanism and additional error-reporting registers. Three new interrupt registers reside in this second set of CP0 registers, which are accessed through the **CFC0** and **CTC0** instructions.

Some of the features of the RM7000 are listed below and are explained further in the following subsections.

- Dual-Issue Superscalar Dispatch Unit with Multiple Instruction Pipelines
- Large On-chip Secondary Cache
- Non-Blocking Caches
- Cache Locking on a Per-Line Basis
- Dedicated Tertiary Cache Interface
- Enhanced Interrupt Handling Mechanism
- Data Prefetch Support
- Out-of-Order Memory Cycle Completion
- Extended CP0 Addressing
- Test and Debug Support

1.1 Superscalar Dispatch with Multiple Instruction Pipelines

The superscalar dispatch unit can issue up to two instructions per clock cycle. Instructions are divided into four classes: integer, load/store, branch, and floating-point. The superscalar dispatch unit contains two logical pipelines called the M-pipe (memory) and the F-pipe (function). The F-pipe handles branch operations and simple ALU operations such as add and shift, as well as integer multiply and divide. The M-pipe can handle both simple ALU operations as well as memory load/store operations. Refer to Section 3, “Superscalar Issue Mechanism”, for more information on the superscalar issue mechanism.

1.2 On-Chip Secondary Cache

In addition to separate 16 Kbyte primary instruction and data caches, the RM7000 contains a 256 Kbyte unified secondary cache. Integration of the secondary cache allows the RM7000 enhanced flexibility in cache organization and management policies and decreases the primary cache miss latency time compared to an off-chip secondary cache implementation.

The secondary cache data bus is 64-bits wide and matches that of the primary caches and system bus interface. The secondary cache tag RAM contains a 20-bit physical address, a 3-bit cache state field, and two parity bits.

The secondary cache is only accessed after a primary cache miss. A new secondary cache write protocol allows both the secondary and tertiary caches to be bypassed.

Refer to Section 7, “Cache Organization and Operation”, for more information on cache organization and operation.

1.3 Non-Blocking Caches

In a typical blocking cache architecture, the processor executes out of the primary caches until a cache miss occurs, at which time the processor stalls until the miss is resolved.

The primary and secondary caches of the RM7000 are non-blocking. A non-blocking cache allows for subsequent cache accesses even though a cache miss has occurred. The RM7000 supports two outstanding cache misses. The processor only

stalls if it either encounters a third cache access which might miss before either one of the previous cache misses has completed, or if a subsequent instruction required data from either one of the instructions that caused the cache miss.

Refer to Section 7, “Cache Organization and Operation”, for more information on cache organization and operation.

1.4 Cache Locking

The primary and secondary caches support cache locking. This mechanism allows the user to lock critical code or data segments in the cache on a per-line basis by setting the appropriate cache lock enable bits in the CP0 *ECC* register. Two of the four sets can be locked within each cache.

Refer to Section 7, “Cache Organization and Operation”, for more information on cache organization and operation.

1.5 Tertiary Cache Interface

The RM7000 contains an R5000-compatible tertiary cache interface with dedicated control signals. The address and data interface for the tertiary cache uses the **SysAD** bus. The tertiary cache is direct mapped and implements a block write-through scheme. The tertiary cache supports 512 Kbyte, 1 Mbyte, 2 Mbyte, 4 Mbyte, and 8 Mbyte sizes.

Refer to Section 7, “Cache Organization and Operation”, for more information on cache organization and operation.

1.6 Non-Pendant System Bus

The new RM7000 non-pendant bus mode allows for the reduction of external memory latency by supporting two outstanding bus transactions simultaneously. Non-pendant bus mode works best when the memory array is banked and pipelined, allowing for these multiple memory references to be serviced simultaneously. If the array is neither banked nor pipelined, there is no benefit to using the RM7000 in non-pendant bus mode.

The RM7000 supports the out-of-order completion of external memory transactions. The external memory controller determines whether the two outstanding cycles are completed out-of-order and asserts a dedicated RM7000 input signal to indicate the order of response.

Refer to Section 8, “Processor Bus Interface”, for more information on the RM7000 bus interface.

1.7 Enhanced Interrupt Handling

The RM7000 provides extended interrupt support over previous generations of MIPS processors. The number of physical interrupts has been expanded from six to ten. In addition, each interrupt can be separately prioritized and vectored via three new interrupt registers added to CP0. Interrupt vector spacing is controlled through the *Interrupt Control* register. Six different spacings are supported. Two other registers are used to support interrupt priority. The priorities can be set in any manner relative to each other, and each interrupt can be assigned a priority level between one and sixteen.

The interrupt registers are accessed through the **CFC0** and **CTC0** instructions. Refer to Section 14, “Interrupts”, for more information on interrupts.

1.8 Data Prefetch Support

The RM7000 processor supports data prefetching via the MIPS **PREF** and **PREFX** instructions. Prefetching of data is a technique whereby the processor can request a block of data prior to the time it is actually needed. These instructions are used by the compiler or assembly language programmer when it is known that an upcoming data reference is likely to miss in the cache.

Execution of the **PREF** instruction for integer operations, or the **PREFX** instruction for floating-point operations, at the appropriate place in the code stream causes the RM7000 to initiate a memory transaction without stalling the processor. The requested block of data is then fetched and placed in the data cache.

The **PREF** and **PREFX** instructions behave as special load instructions that cannot generate an address error nor a TLB exception. Refer to the MIPS IV Instruction Set Manual as well as section 7.8.6, “Using the Prefetch Instructions” on page 94, for more information on these instructions.

1.9 Extended CP0 Addressing

The MIPS architecture provides enough address space for 32 coprocessor 0 (CP0) registers. To accommodate the enhanced interrupt handling mechanism on the RM7000, 32 additional registers have been added. The original 32 registers are defined as Set 0, and the second 32 registers are defined as Set 1. Of the second set of 32 registers, five are used for the *Interrupt Control* (*INTCTL*), *Interrupt Priority Level Lo* (*IPLLo*), *Interrupt Priority Level Hi* (*IPLHi*), *Data Error Address 1* (*DErrAddr1*), and *Data Error Address 2* (*DErrAddr2*) registers. The remaining registers are reserved for future use. The three new interrupt control and two new error registers are accessed using the **CTC0** and **CFC0** instructions. Refer to Section 12, “System Interface Protocol”, for more information.

1.10 Test and Debug Support

To enhance hardware and software control over the processor during debug, the RM7000 includes a pair of *Watch* registers. Each register can be separately enabled to watch for a load address, a store address, or an instruction address. Address comparison can be done on either a single address or range of addresses. An address match to either *Watch* register results in an exception.

The *Watch* registers can also be used for additional address protection. Address ranges smaller than one page can be protected against modification by store instructions.

In addition to the *Watch* registers, the RM7000 contains a performance counter implemented using two new CP0 registers. One register is a 32-bit writable up counter, and the other is a 32-bit register containing a five-bit field that can select one of twenty-two event types that can be counted by the counter. When bit 31 of counter is set, the performance counter generates an interrupt.

An alternative use for the performance counter is as either a second timer or a watchdog timer. A watchdog interrupt can be used as an aid when the system of software “hangs”. Typically software periodically updates the count so that interrupts do not occur. When a “hang” event occurs the interrupt triggers, thereby resolving the “hang” condition. Refer to Section 17, “Debug and Test”, for more information.

Section 2 Processor Pipeline

The RM7000 processor incorporates a five-stage instruction pipeline for both the M-pipe (memory) and the F-pipe (function). The RM7000 can execute two instructions per clock depending on the instruction type. Refer to Section 3, “Superscalar Issue Mechanism”, for a listing of the instruction issue rules. A separate seven stage pipeline handles floating-point operations. The floating point pipeline is completely transparent to the user. The RM7000 performs virtual to physical address translation in parallel with the cache access.

Each stage takes one PCycle (one cycle of PClock, which runs at a multiple of the frequency of **SysClock**). Thus, the execution of each instruction takes at least five PCycles. An instruction can take longer to complete depending on the location of the requested data. For example, if the data is not in the cache, the data must be retrieved from main memory.

Once the pipeline has been filled, ten instructions can be executed simultaneously.

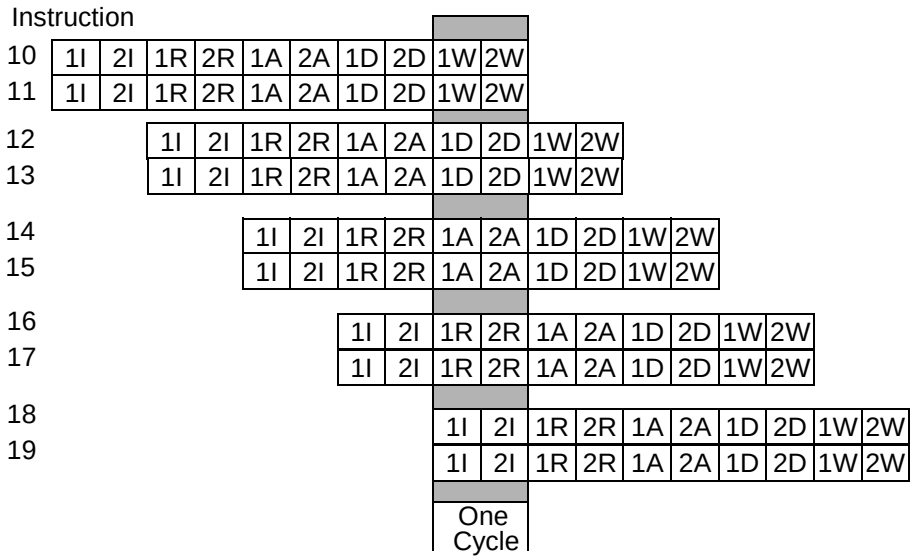


Figure 2.1 Instruction Pipeline Stages

Figure 2.1 shows the five stages of the instruction pipeline. As shown in the gray area, up to ten instructions can be executing simultaneously. Factors such as the out-of-order completion of loads and stores, and long-latency floating point operations can affect the number of instructions in the pipeline at any given time. Instruction dependencies, resource conflicts, and branches can result in some of the instruction slots being occupied by **NOP**s.

2.1 Instruction Pipeline Stages

- 1I - Instruction Fetch, Phase One
- 2I - Instruction Fetch, Phase Two
- 1R - Register Fetch, Phase One
- 2R - Register Fetch, Phase Two
- 1A - Execution, Phase One
- 2A - Execution, Phase Two
- 1D - Data Fetch, Phase One

- 2D - Data Fetch, Phase Two
- 1W - Write Back, Phase One
- 2W - Write Back, Phase Two

2.1.1 1I - Instruction Fetch, Phase One

During the 1I phase, the following occurs:

- Branch logic selects an instruction address and the instruction cache fetch begins.
- The instruction translation lookaside buffer (ITLB) begins the virtual-to-physical address translation.

2.1.2 2I - Instruction Fetch, Phase Two

The instruction cache fetch and the virtual-to-physical address translation continues.

2.1.3 1R - Register Fetch, Phase One

During the 1R phase, the following occurs:

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

2.1.4 2R - Register Fetch, Phase Two

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

- The instruction is decoded.
- Any required operands are fetched from the register file.
- Determine whether the instruction is issued or delayed depending on interlock conditions.

2.1.5 1A - Execution - Phase One

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

- Calculate branch address (if applicable).
- Any result from the A or D stages is bypassed.
- The ALU starts an integer operation.
- The ALU calculates the data virtual address for load and store instructions.
- The ALU determines whether the branch condition is true.

2.1.6 2A - Execution - Phase Two

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

- The integer operation begun in the 1A phase completes.
- Data cache address decode.
- Store data is shifted to the specified byte positions.
- The DTLB begins the data virtual-to-physical address translation.

2.1.7 1D - Data Fetch, Phase One

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

- The DTLB data address translation completes.

- The JTLB virtual-to-physical address translation begins.
- Data cache access begins.

2.1.8 2D - Data Fetch - Phase Two

- The data cache access completes. Data is shifted down and extended.
- The JTLB address translation completes.
- The data cache tag is checked against the PFN from the DTLB or JTLB for any data cache access.

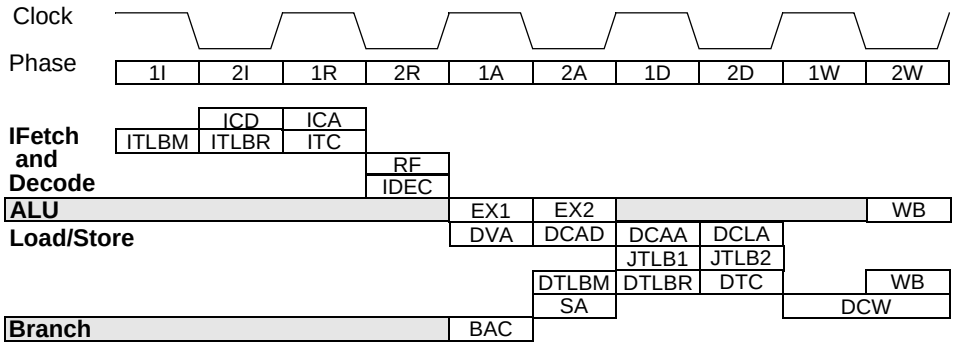
2.1.9 1W - Write Back, Phase One

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

2.1.10 2W - Write Back, Phase Two

For register-to-register and load instructions, the result is written back to the register file.

For register-to-register instructions, the instruction result is written back to the register file during the WB stage. Branch instructions perform no operation during this stage.



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 fetch
IDEC	Instruction decode	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 2.2 CPU Pipeline Activities

Figure 2.2 shows the activities occurring during each ALU pipeline stage for load, store, and branch instructions.

2.2 Branch Delay

The CPU pipeline has a one-cycle branch delay and a one-cycle load delay. 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 phase. Figure 2.3 illustrates the branch delay for one pipeline.

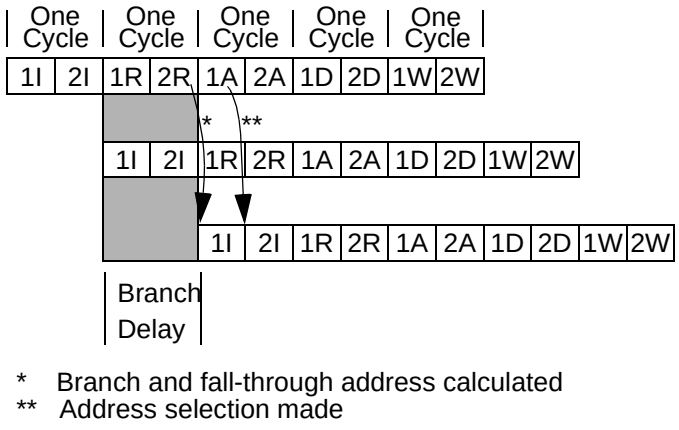


Figure 2.3 CPU Pipeline Branch Delay

2.3 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 phase of the subsequent instruction following the load delay slot. Figure 2.4 shows the load delay of two pipeline stages.

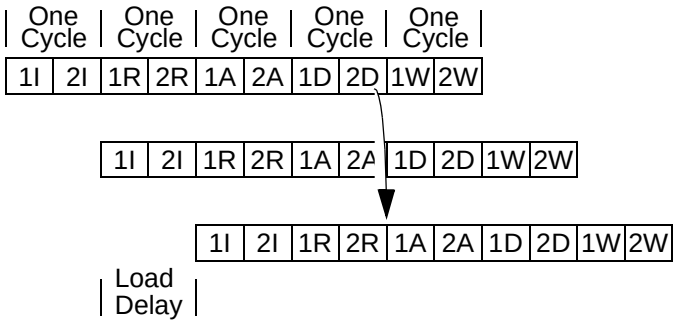


Figure 2.4 CPU Pipeline Load Delay

2.4 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 requires one part of the pipeline to advance while another part of the pipeline is held static.

At each cycle, exception and interlock conditions are checked for all active instructions.

Because each exception or interlock condition corresponds to a particular pipeline stage, a condition can be traced back to the particular instruction in the exception/interlock stage. For instance, a Reserved Instruction (RI) exception is raised in the execution (A) stage.

Table 2.1: Relationship of Pipeline Stage to Interlock Condition

State	Pipeline Stage				
	I	R	A	D	W
Stall			MULS	DCM	
			LL	CPE	
			DTLBM	CacheOP	
				ExtReq	
				Excpt	
Slip	ITM	ICM			
		Interlock			
		MDBusy			
		FCBusy			
		Sync			
Exceptions	ITLB	IBE	RI	DBE	
		IPErr	CUn	Reset	
		IWatch	BP	DPErr	
			SC	OVF	
			DTLB	FPE	
			TLBMod	DWatch	
			NMI	Trap	
			Intr		

Table 2.2: Pipeline Exceptions

Exception	Description
ITLB	Instruction Translation or Address Exception
Intr	External Interrupt
IBE	IBus Error
RI	Reserved Instruction
BP	Breakpoint
SC	System Call
CUn	Coprocessor Unusable
IPErr	Instruction Parity Error
OVF	Integer Overflow
FPE	FP Exception
DTLB	Data Translation or Address Exception
TLBMod	TLB Modified
DBE	Data Bus Error
DPErr	Data Parity Error
NMI	Non-maskable Interrupt
Trap	Trap
IWatch	Instruction Watch
DWatch	Data Watch
Reset	Reset

Table 2.3: Pipeline Interlocks

Interlock	Description
ITM	Instruction TLB Miss
ICM	Instruction Cache Miss
CPE	Coprocessor Possible Exception
DCM	Data Cache Miss
Interlock	register src-dest & dest-dest Interlocks
MDBusy	Multiply/Divide Busy
MULS	Multiplier stall
Sync	Memory Synchronization
LL	Load Linked Instruction
CacheOP	Cache Instruction
DTLBM	Data TLB Miss
WBFull	Write Buffer Full
ExtReq	External Interface Request
Excpt	Exception Synchronization
FCBusy	FP Busy

2.4.1 Exception Conditions

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

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

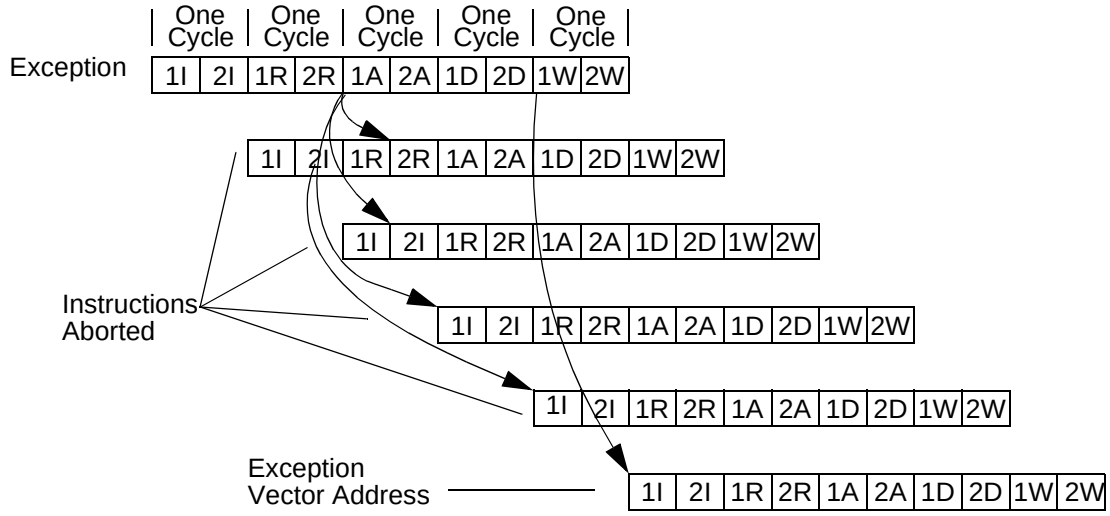


Figure 2.5 Exception Detection Mechanism

For most exceptions, 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 *Exception Program Counter* (*EPC*) is such that execution can be restarted for most exceptions. Most exceptions are guaranteed to be taken in order. Figure 2.5 illustrates the exception detection mechanism for a *Reserved Instruction* (RI) exception.

The non-blocking nature of the caches requires that parity error and bus error exceptions be allowed to occur after other subsequent instructions have completed after the load/store instruction which caused the error. For these two exception types, the value of the *EPC* is not guaranteed to hold an address where execution can be restarted. For this reason, these exceptions are classified as imprecise exceptions. The operating system must search the instruction stream near the *EPC* value to decide at which location can execution be restarted. Only parity errors and bus errors can be imprecise exceptions on this implementation. All other exception types are precise.

2.4.2 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 2.6 shows a data cache miss stall.

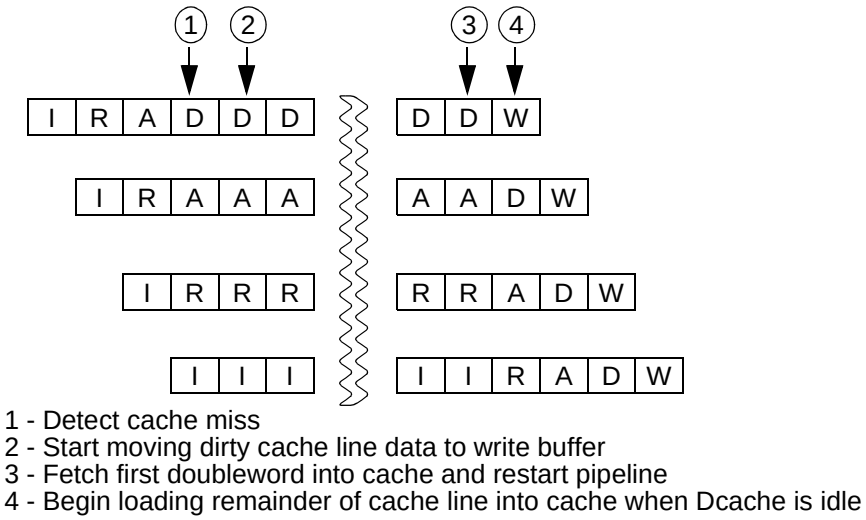


Figure 2.6 Servicing a Data Cache Miss

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 2.6 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 operations is completed.

2.4.3 Slip Conditions

During the 2R and 1A pipeline phases, 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, then the instruction is issued. Otherwise, the instruction “slips”. Slipped instructions 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. **NOP**s are inserted into the bubbles which are created in the pipeline. Branch likely instructions, **ERET**, and exceptions do not cause slips. Figure 2.7 shows how instructions can slip during an instruction cache miss.

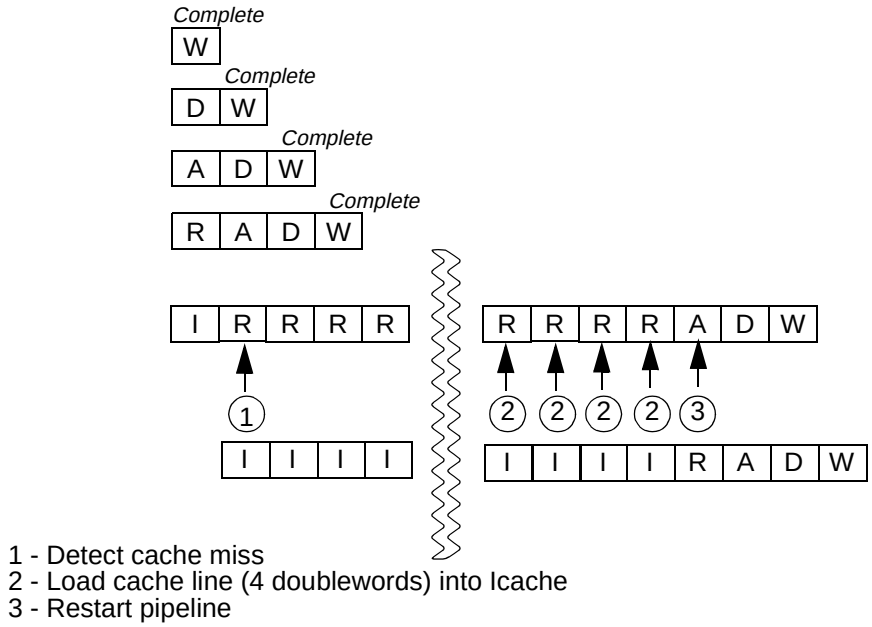


Figure 2.7 Slips During an Instruction Cache Miss

Instruction cache misses and slips are detected in the R-stage of the pipeline and never require a writeback operation as writes are not allowed to the instruction cache. The data cache early restart mechanism, 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 at the instruction that caused the cache miss.

2.4.4 Write Buffer

The RM7000 processor contains a write buffer which improves the performance of write operations to external memory. All writes to external memory, whether cache miss write-backs 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.

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 significantly increases performance by decoupling the **SysAD** bus transfers from the instruction execution stream. If the write buffer is full, additional stores are stalled until there is room for them in the write buffer.

Section 3 Superscalar Issue Mechanism

The RM7000 superscalar dispatch unit implements two separate pipelines and can issue up to two instructions per clock cycle. The Function (F) pipe handles integer, branch and floating point operations such as add, subtract, multiply, and divide. The Memory (M) pipe handles integer operations as well as load/store operations, and some floating point moves.

Table 3.1: lists the different types of F-and M-pipe operations. Each pipe can issue one instruction class per clock cycle.

In the following tables, the Branch group includes all branch and jump opcodes.

- The Load/Store type includes both integer and floating point loads as well all prefetch opcodes.
- The Integer **MUL**/**DIV** type includes the **MULT**, **DMULT**, **MULTU**, **DMULTU**, **DIV**, **DDIV**, **DIVU**, **DDIVU**, **MAD**, **MADU**, and **MUL** opcodes.
- The FP Conditional Move type includes the **MOVF**, **MOVF.FMT**, **MOVN**, **MOVN.FMT**, **MOVT**, **MOVT.FMT**, and **MOVZ.FMT** opcodes.
- The FP Unconditional Move type includes the **MOV.FMT** opcode.
- The FPR ↔ GPR type moves include the **MFC1**, **MTC1**, **DMTC1**, and **DMFC1** opcodes.

Instructions that deal with the control registers of the processor do not dual-issue with other instructions. This alleviates timing hazards when modifying these control registers. These instructions include **ERET**, **TLBR**, **TLBP**, **TLBWI**, **TLBWR**, **DMTC0**, **MTC0**, **DMFC0**, **MFC0**, **CTC0**, **CFC0**, **WAIT**, **CTC1**,and **CFC1**.

Table 3.1: Instruction Pipeline Operations

F-Pipe Operations	M-Pipe Operations
Integer Operation	Integer Operation
Floating Point Operation	Load/Store
Branch	FP Conditional Move
Integer Mul/Div	FPR ↔ GPR Move
FP Unconditional Move	

The various operations shown in Table 3.1: yield the following combinations that can be issued together:

Table 3.2: Dual Issue Combinations.

Operation Combinations
Integer Operation + Integer Operation
Integer Operation + Load/Store
Integer Operation + Floating Point Conditional Move
Integer Operation + FPR ↔ GPR Move
Floating Point Operation + Integer Operation
Floating Point Operation + Load/Store
Floating Point Operation + Floating Point Conditional Move
Floating Point Operation + FPR ↔ GPR Move
Branch Operation + Integer Operation
Branch Operation + Load/Store
Branch Operation + FP Conditional Move
Branch Operation + FPR ↔ GPR Move
Integer Mul/Div + Integer Operation
Integer Mul/Div + Load/Store
Integer Mul/Div + FP Conditional Move
Integer Mul/Div + FPR ↔ GPR Move
FP Unconditional Move + Integer Operation
FP Unconditional Move + Load/Store
FP Unconditional Move + FP Conditional Move
FP Unconditional Move + FPR ↔ GPR Move

The RM7000 does not have even/odd word alignment restrictions. For example, a floating point + load/store instruction pair can dual-issue just as well as a load/store + floating point instruction pair.

The RM7000 can also dual-issue instructions that are not instruction-fetch (doubleword) aligned. For example, instructions I and I + 1 are placed into the 2-deep instruction dispatch buffer. Instruction ‘I’ single-issues and instruction ‘I + 1’ is left in the buffer. Instruction ‘I + 2’ is then placed into the instruction dispatch buffer and instruction ‘I + 1’ is shifted into the location used by the oldest instruction within the dispatch buffer. In the next cycle instructions ‘I + 1’ and ‘I + 2’ can dual-issue together even though they were not fetched from the instruction cache at the same time. The dispatch unit dispatches the instructions to the F- and M-pipes based on the instruction issue rules in Table 3.1:.

Figure 3.1 shows a simplified diagram of how the instructions are routed to each pipeline.

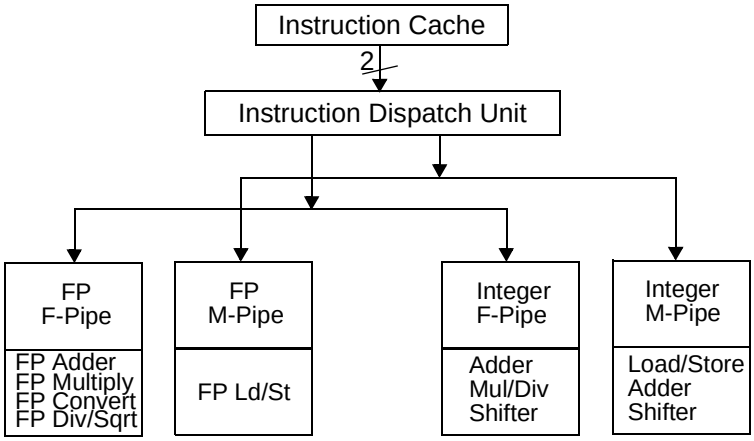


Figure 3.1 Instruction Issue Block Diagram

Figure 3.2 shows a simplified flow of instructions through the dual issue pipeline. Refer to Section 2, “Processor Pipeline”, for a definition of pipeline stages.

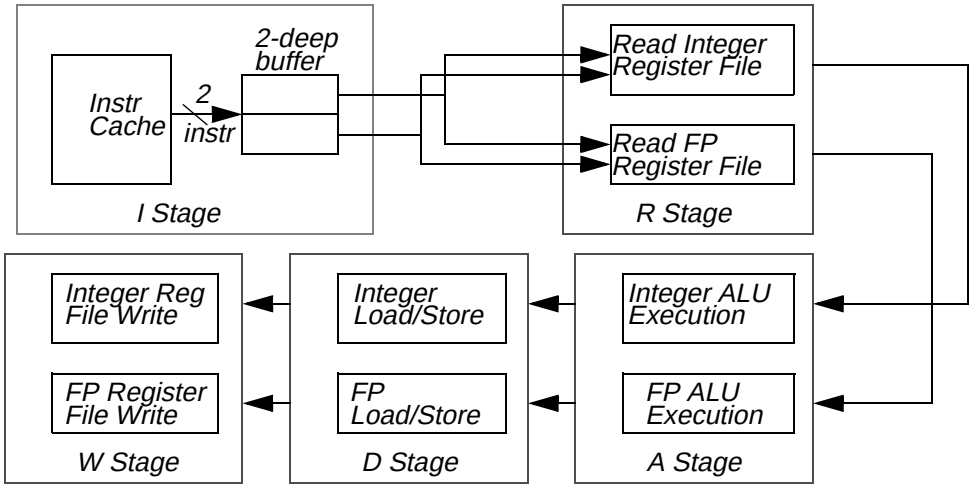


Figure 3.2 Dual Issue Mechanism

Section 4 Integer Unit

This section describes the Integer Unit (IU) of the RM7000 processor, including the programming model, instruction set and formats, and the pipeline.

The IU implements the MIPS IV instruction set and is fully backward compatible with previous generation processors running the MIPS III instruction set.

In addition to running the MIPS IV instruction set, the RM7000 also implements two implementation specific instructions which are executed by the IU; integer-multiply-accumulate (**MAD**) and a three-operand integer-multiply (**MUL**).

4.1 Overview

The RM7000 IU contains two complete integer arithmetic logic units (ALU) and implements a dual-pipeline architecture. The F-pipe handles integer, floating point, and branch operations as well as integer multiply/divide operations. The M-pipe handles integer, logical operations as well as load/store operations.

In addition to the dual-pipeline architecture, the RM7000 contains thirty-two 64-bit general purpose registers, *Hi* and *Lo* registers for the integer multiply and divide operations, and a program counter.

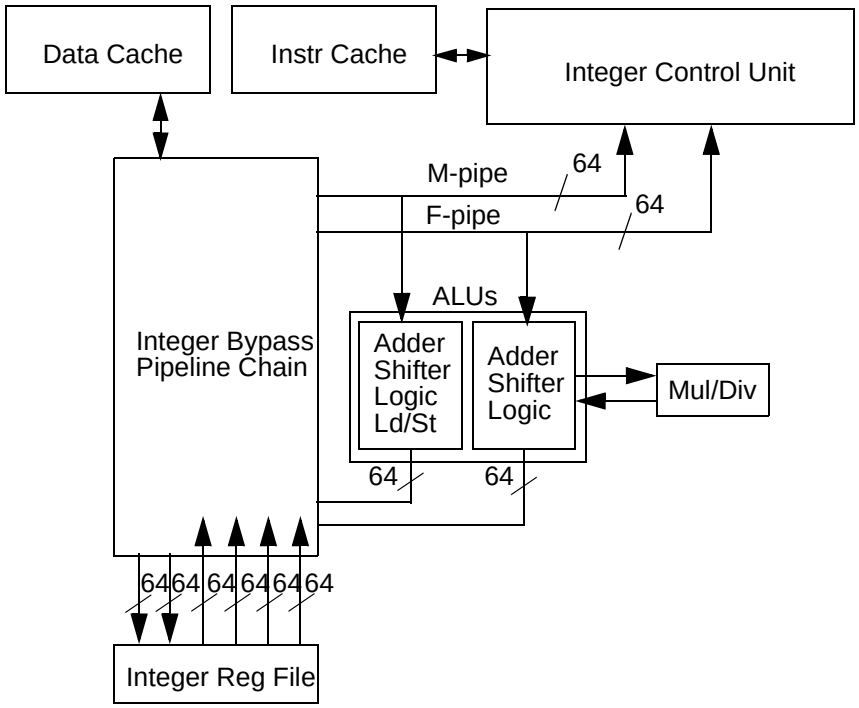


Figure 4.1 IU Functional Block Diagram

Figure 4.1 shows a block diagram of the integer unit.

4.2 Integer Unit Features

This section briefly describes the features of the RM7000 integer unit.

- **Multiple ALUs.** The IU contains two integer ALUs, each containing a shifter, an adder, and a logic unit. One ALU is dedicated to the F-pipe, while the other ALU is dedicated to the M-pipe.
- **Load and Store Instruction Set.** The IU uses a load-and-store-oriented instruction set, with single-cycle load and store operations.
- **Dedicated Multiply/Divide Unit.** The RM7000 contains a single integer multiply/divide unit optimized for high-speed multiply/divide and multiply-accumulate operations. This unit is connected to the F-pipe ALU.
- **Implementation Specific Multiply Operations.** The RM7000 supports a number of implementation specific multiply/multiply-add instructions that are not part of the MIPS IV instruction set. These instructions perform multiple operations in one instruction in order to maximize throughput.

4.3 Integer Register File

The IU contains thirty-two 64-bit general purpose registers. Register 0 (*r0*) is hard-wired to zero. These registers are for temporary storage during scalar operations and address calculations. In order to service the two integer execution units, the integer register file contains four read ports and two write ports. Each of these ports is defined by the instruction being executed (r1, r3, r10, etc.). Each port is fully bypassed both within and between the two execution units to minimize pipeline latency.

4.4 Pipeline Bypass

The Integer-Pipeline Bypass Chain allows the result of an operation to be passed to the next instruction without having to wait for the result to be written to the register file. One instruction can fetch the operands produced by the previous instruction without waiting for that instruction to reach the W pipeline stage.

4.5 Arithmetic Logic Units

The RM7000 contains two integer arithmetic logic units (ALU) that support F-and M-pipe operations. Both ALUs contain an adder, a shifter, and support for logic operations. The M-pipe ALU contains a load/store unit used for memory operations. The F-pipe contains a multiply/divide unit. Each ALU operates independently of the other, and each is optimized to perform all operations in a single processor cycle. Table 4.1: shows the basic ALU functions.

Table 4.1: ALU Functions

Unit	F-Pipe Operations	M-Pipe Operations
Adder	add, sub	add, sub, Data address add
Logic	logic, moves, zero shifts (nop)	logic, moves, zero shifts (nop)
Shifter	non zero shift	non zero shift, store align

4.6 Integer Multiply/Divide Operations

The F-pipe ALU contains a dedicated multiply/divide (**MUL**/**DIV**) unit that performs high-speed multiply and multiply-accumulate operations. The **MUL**/**DIV** unit implements all basic MIPS IV integer multiply instructions as well as two implementation-specific instructions; **MUL** and **MAD**. The **MUL** instruction allows the result of the multiply operation to go directly to the integer register file as opposed to the *Lo* register, as is the case with all basic MIPS IV multiply instructions. The **MUL** instruction eliminates the need to execute a Move-From-Lo (**MFLO**) instruction normally required to move the data from the *Lo* register to the register file. The portion of the multiply operation that would normally go to the *Hi* register is discarded.

The Multiply-Add (**MAD**) instruction multiplies two operands and adds the resulting product to the current contents of the *Hi* and *Lo* registers. The multiply accumulate operation is the core primitive in almost all signal processing algorithms, eliminating the need for a separate digital signal processor (DSP).

Section 5 Memory Management Unit

The RM7000 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 section describes the processor virtual and physical address spaces, the virtual-to-physical address translation, the operation of the TLB in making these translations, the cache memories, and those System Control Coprocessor (CP0) registers that provide the software interface to the TLB.

5.1 Translation Lookaside Buffers (TLB)

Mapped virtual addresses are translated into physical addresses using on-chip Translation Lookaside Buffers (TLB).[1] The primary TLB is the Joint TLB (JTLB). In addition, the RM7000 processor contains separate Instruction and a Data TLBs to avoid contention for the JTLB.

5.1.1 Joint TLB

For fast virtual-to-physical address translation, the RM7000 uses a large, fully associative TLB that maps virtual pages to their corresponding physical addresses. As indicated by its name, the Joint TLB, or JTLB is used for both instruction and data translations. The JTLB is organized as pairs of even/odd entries, and maps a virtual address and address space identifier into the large, 64 GByte physical address space. By default, the JTLB is configured as 48 pairs of even/odd entries to allow the mapping of 96 pages. A boot time option (mode bit 24) allows the JTLB to be configured as 64 pairs of even/odd entries to allow mapping of 128 pages of virtual memory.

Two mechanisms are provided to assist in controlling the amount of mapped space and the replacement characteristics of various memory regions. First, the page size can be configured, on a per-entry basis, to use page sizes in the range of 4KB to 16MB (in multiples of 4). A CP0 register, *PageMask*, is loaded with the desired page size of a mapping, and that size is stored into the TLB along with the virtual address when a new entry is written. Thus, operating systems can create 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 RM7000 provides a random replacement algorithm to select a TLB entry to be written with a new mapping; however, the processor also provides a mechanism whereby a system specific number of mappings can be locked into the TLB, thereby avoiding random replacement. This mechanism allows the operating system to guarantee that certain pages are always mapped for performance reasons and for deadlock avoidance. This mechanism also facilitates the design of real-time systems by allowing deterministic access to critical software.

The JTLB also contains information that controls 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 without write-allocate, or non-coherent writethrough with write-allocate.

The non-coherent protocols are used for both code and data on the RM7000 with data using write-back or write-through depending on the application.

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 0x0 0000 0000 || VA[28:0].

5.1.2 Instruction TLB

The RM7000 use a 4-entry instruction TLB, or ITLB, to minimize contention for the joint TLB, eliminate the timing critical path of translating through a large associative array, and save power. Each ITLB entry maps a 4KB page. The ITLB improves performance by allowing instruction address translation to occur in parallel with data address translation. When a miss occurs on an instruction address translation by the ITLB, the least-recently used ITLB entry is filled from the joint TLB. The operation of the ITLB is completely transparent to the user.

5.1.3 Data TLB

The RM7000 use a 4-entry data TLB, or DTLB, to minimize contention for the joint TLB, eliminate the timing critical path of translating through a large associative array, and save power. Each DTLB entry maps a 4KB page. The DTLB improves performance by allowing data address translation to occur in parallel with instruction address translation. When a miss occurs on a data address translation by the DTLB, the DTLB is filled from the JTLB. The DTLB refill is pseudo-LRU: the least recently used entry of the least recently used pair of entries is filled. The operation of the DTLB is completely transparent to the user.

5.1.4 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 no match occurs (TLB miss), an exception is taken and software refills the TLB from the page table resident in memory. Software can write over a selected TLB entry or use a hardware mechanism to write into a random entry.

5.1.5 Multiple Matches

The RM7000 processor do not provide any detection or shutdown mechanism for multiple matches in the TLB. Unlike earlier MIPS designs, multiple matches do not physically damage the TLB. Therefore, multiple match detection is not needed. The result of this condition is undefined, and software is expected to never allow this to occur.

5.2 Processor Modes

The RM7000 has three processor operating modes, an instruction set mode, and an addressing mode. All are described in this section.

5.2.1 Processor Operating Modes

The three operating modes are listed in order of decreasing system privilege:

- **Kernel Mode** (highest system privilege): can access and change any register. The innermost core of the operating system runs in kernel mode.
- **Supervisor Mode:** has fewer privileges and is used for less critical sections of the operating system.
- **User Mode** (lowest system privilege): prevents users from interfering with one another.

User mode is the processor's base operating mode. The processor is forced into Kernel mode when it is handling an error (***ERL*** bit is set) or an exception (***EXL*** bit is set).

The processor's operating mode is set by the *Status* register's ***KSU*** field, together with the ***ERL***, ***EXL***, ***KX***, ***SX***, ***UX,*** and ***XX*** bits. Table 5.1: lists the *Status* register settings for the three operating modes, as well as error and exception level settings; the blanks in the table indicate *don't cares*.

Table 5.1: Processor Modes

XX 31	KX 7	SX 6	UX 5	KSU 2	ERL 2	EXL 1	IE 0	Description	ISA III	ISA IV	Addressing Mode 32-Bit/64-Bit
			0	10	0	0		User mode	0	0	32
0			1	10	0	0			1	0	64
1			1	10	0	0			1	1	64
		0		01	0	0		Supervisor mode	1	1	32
		1		01	0	0			1	1	64
	0			00	0	0		Kernel mode	1	1	32
	1			00	0	0			1	1	64
	0				0	1		Exception level	1	1	32
	1				0	1			1	1	64
	0				1			Error level	1	1	32
	1				1				1	1	64
					0	0	1	Interrupts are enabled			

5.2.2 Instruction Set Mode

The processor's *instruction set mode* determines which instruction set is enabled. By default, the processor implements the MIPS IV Instruction Set Architecture (ISA). For compatibility with earlier machines, however, it can be limited to the MIPS III ISA or the MIPS I/II ISAs.

5.2.3 Addressing Modes

The processor's *addressing mode* determines whether it generates 32-bit or 64-bit virtual memory addresses.

Refer to Table 5.1: for the following addressing mode encodings:

- In *Kernel* mode the ***KX*** bit enables 64-bit addressing; all instructions are always valid.
- In *Supervisor* mode, the ***SX*** bit enables 64-bit addressing; all instructions are always valid.
- In *User* mode, the ***UX*** bit enables 64-bit addressing; the ***XX*** bit enables the new MIPS IV instructions.

5.2.4 Endian Modes

The RM7000 processor can operate either in big-endian mode or little-endian mode. The choice of the endianness is chosen at Power-up initialization time through the logical OR of the endianness serial mode-bit (bit 8) and the input value of the **BigEndian** pin. This choice is known as the base endianness and is used for kernel mode, supervisor mode and user mode operation.

It is possible for specific user-mode processes to operate in the reverse of the base-endianness. This is accomplished by setting the ***RE*** bit in the CP0 *Status* register while in kernel mode before the execution of such user mode processes.

5.3 Address Spaces

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

5.3.1 Virtual Address Space

The processor has three address spaces*:* kernel, supervisor, and user. Each space can be independently configured to be a 32-bit or 64-bit space by the ***KX***, ***SX***, and ***UX*** bits in the *Status* register.

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

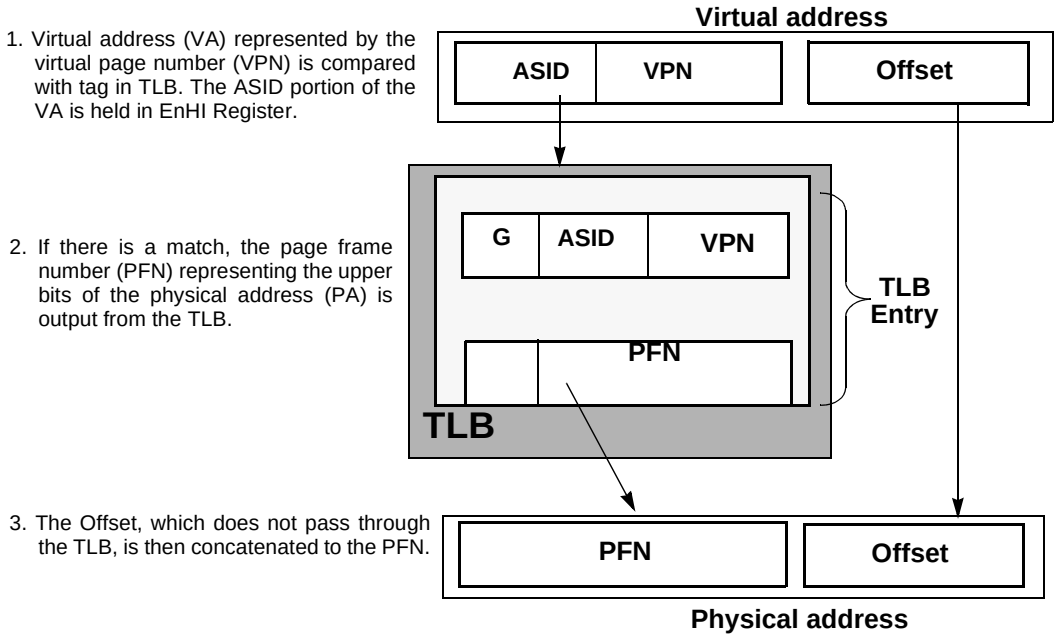


Figure 5.1 Overview of a Virtual-to-Physical Address Translation

Figure 5.1 shows the translation of a virtual address into a physical address. As shown , 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 each TLB entry.

5.3.2 Physical Address Space

Using a 36-bit address, the processor physical address space encompasses 64 gigabytes.

5.3.3 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.

The next sections describe the 32-bit and 64-bit address translations.

5.3.4 Virtual Address Translation

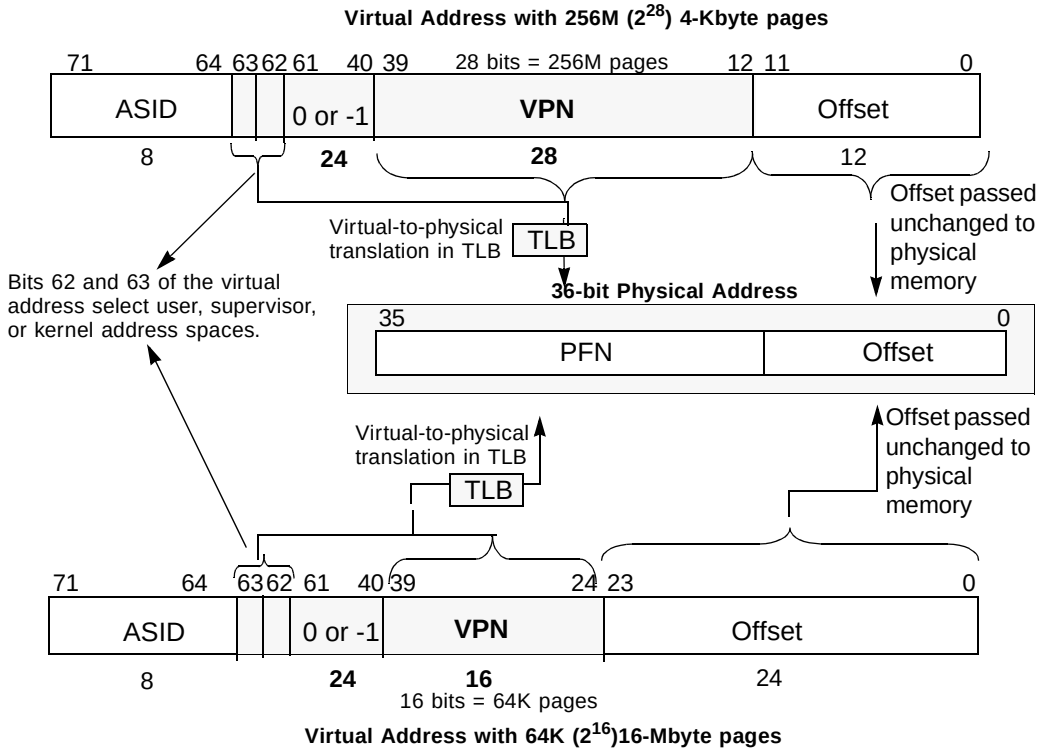


Figure 5.2 64-bit Mode Virtual Address Translation

Figure 5.2 shows the virtual-to-physical-address translation. 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 5.2 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 5.2 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.

5.3.5 Address Spaces

The processor has three address spaces.

- User address space
- Supervisor address space
- Kernel address space

Each space can be independently configured as either 32- or 64-bit.

5.3.6 User Address Space

In User address space, a single, uniform virtual address space—labelled User segment (*useg*), is available; its size is:

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

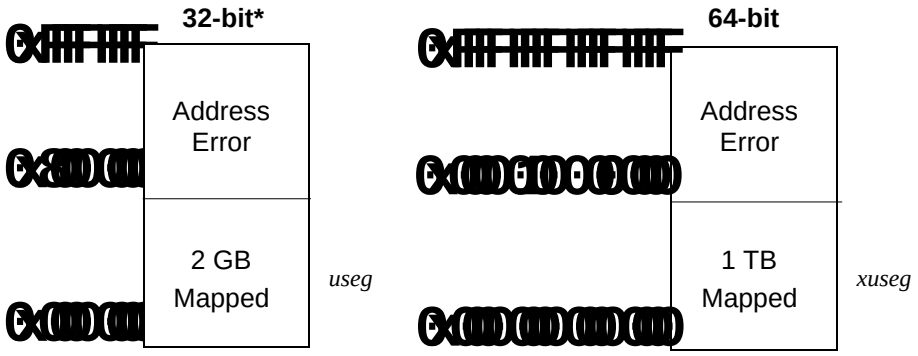


Figure 5.3 User Virtual Address Space as viewed from User Mode

Figure 5.3 shows the range of User virtual address space. User space can be accessed from user, supervisor, and kernel modes.

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

The ***UX*** bit in the *Status* register selects between 32- or 64-bit User address spaces as follows:

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

Table 5.2: lists the characteristics of the two user address spaces, *useg* and *xuseg*.

Table 5.2: 32-bit and 64-bit User Address Space Segments

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

5.3.6.1 32-bit User Space (useg)

In 32-bit User space, when ***UX*** = 0 in the *Status* register, all valid 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.

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. TLB misses on addresses in 32-bit User space (*useg*) use the TLB refill vector.

5.3.6.2 64-bit User Space (xuseg)

In 64-bit User space, when ***UX*** =1 in the *Status* register, addressing is extended to 64-bits. When ***UX***=1, 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. TLB misses on addresses in 64-bit User (*xuseg*) space use the XTLB refill vector.

5.3.7 Supervisor Space

Supervisor address space 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 Supervisor address space provides code and data addresses for supervisor mode.

Supervisor space can be accessed from supervisor mode and kernel mode.

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

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

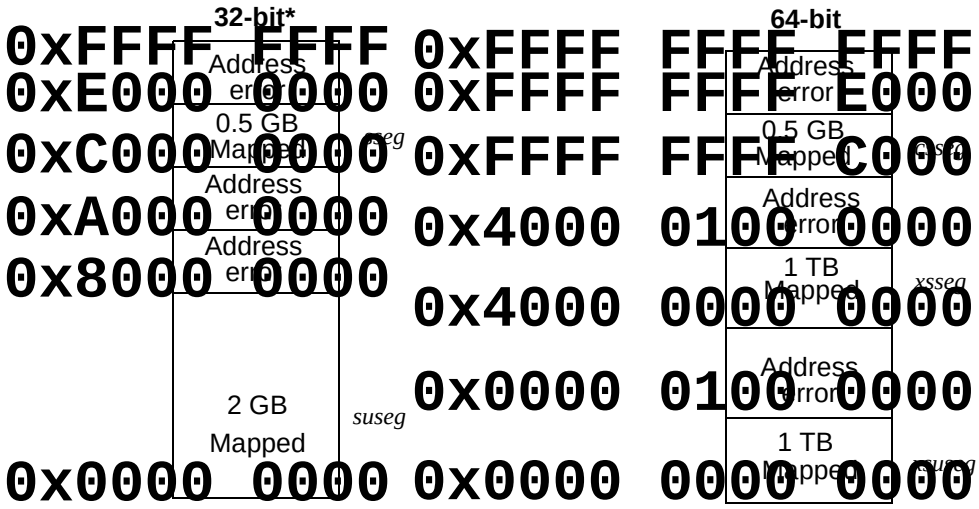


Figure 5.4 User and Supervisor Address Spaces; viewed from Supervisor mode

The ***SX*** bit in the *Status* register select between 32- or 64-bit Supervisor space addressing:

- when ***SX*** = 0, 32-bit supervisor space is selected and TLB misses on supervisor space addresses are handled by the 32-bit TLB refill exception handler
- when ***SX*** = 1, 64-bit supervisor space is selected and TLB misses on supervisor space addresses are handled by the 64-bit XTLB refill exception handler. Figure 5.4 shows Supervisor address mapping. Table 5.3: lists the characteristics of the supervisor space segments; descriptions of the address spaces follow.

Table 5.3: Supervisor Mode Addressing

A(63:62)	SX	UX	Segment Name	Address Range	Segment Size
00_2	X	0	suseg	0x0000 0000 0000 0000 through 0x0000 0000 7FFF FFFF	2 Gbytes (2^{31} bytes)
00_2	X	1	xsuseg	0x0000 0000 0000 0000 through 0x0000 00FF FFFF FFFF	1 Tbyte (2^{40} bytes)
01_2	1	X	xsseg	0x4000 0000 0000 0000 through 0x4000 00FF FFFF FFFF	1 Tbyte (2^{40} bytes)

A(63:62)	SX	UX	Segment Name	Address Range	Segment Size
11_2	X	X	sseg or csseg	0xFFFF FFFF C000 0000 through 0xFFFF FFFF DFFF FFFF	512 Mbytes (2^{29} bytes)

5.3.7.1 32-bit Supervisor, User Space (suseg)

In Supervisor Mode when accessing User space and ***UX*** = 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.

5.3.7.2 32-bit Supervisor, Supervisor Space (sseg)

In Supervisor space, 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.

5.3.7.3 64-bit Supervisor, User Space (xsuseg)

In Supervisor Mode when accessing User space and ***UX*** = 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.

5.3.7.4 64-bit Supervisor, Current Supervisor Space (xsseg)

In Supervisor space, 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.

This mapped space begins at virtual address 0x4000 0000 0000 0000 and runs through 0x4000 00FF FFFF FFFF.

5.3.7.5 64-bit Supervisor, Separate Supervisor Space (csseg)

In Supervisor space, 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.

5.3.8 Kernel Space

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

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

The ***KX*** bit in the *Status* register selects between 32- or 64-bit Kernel space 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 there until an Exception Return (**ERET**) instruction is executed or the ***EXL*** bit is cleared. The **ERET** instruction restores the processor to the address space existing prior to the exception.

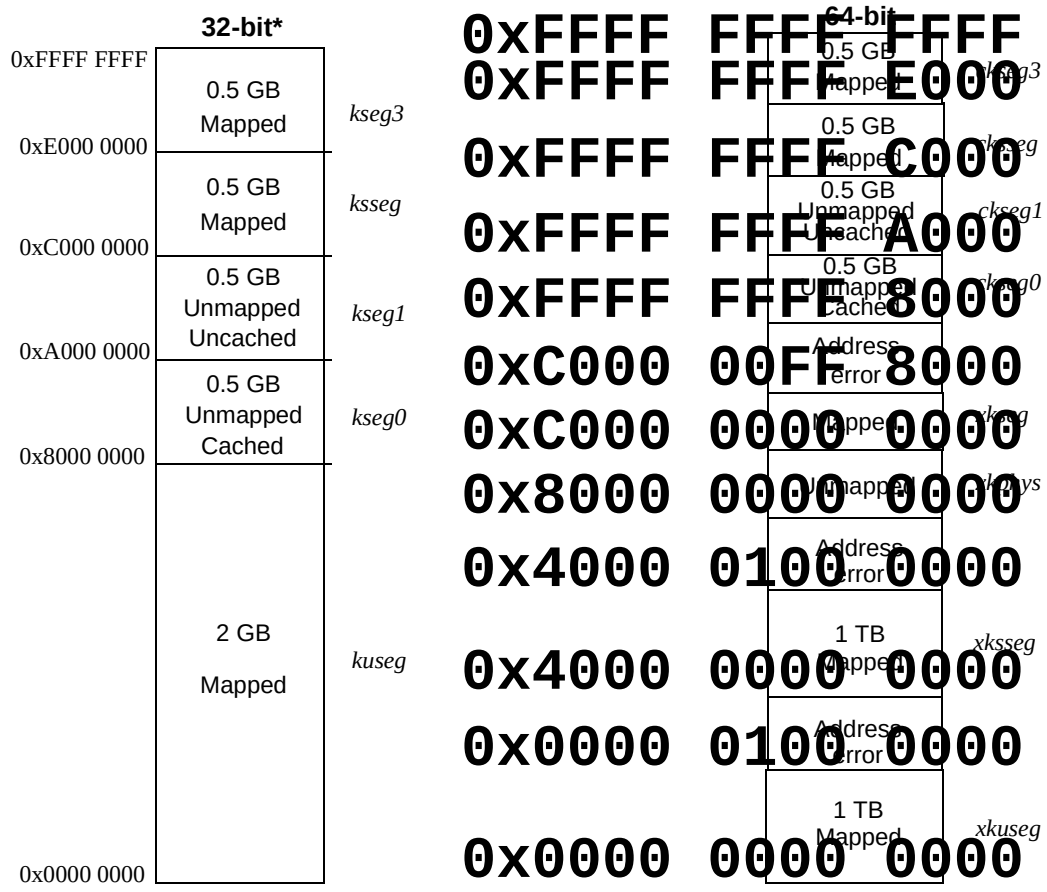


Figure 5.5 User, Supervisor, and Kernel Address Spaces viewed from Kernel mode

Kernel virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 5.5. Table 5.4: lists the characteristics of the kernel mode segments.

Table 5.4: Kernel Mode Addressing

A(63:62)	KX	SX	UX	Segment Name	Address Range	Segment Size
00_2	X	X	0	kuseg	0x0000 0000 0000 0000 through 0x0000 0000 7FFF FFFF	2 Gbytes (2^{31} bytes)
00_2	X	X	1	xkuseg	0x0000 0000 0000 0000 through 0x0000 00FF FFFF FFFF	1 Tbyte (2^{40} bytes)
01_2	X	1	X	xksseg	0x4000 0000 0000 0000 through 0x4000 00FF FFFF FFFF	1 Tbyte (2^{40} bytes)

A(63:62)	KX	SX	UX	Segment Name	Address Range	Segment Size
10_2	1	X	X	xkphys	0x8000 0000 0000 0000 through 0x8000 000F FFFF FFFF etc.	8× 64 Gbytes (2^{36} bytes)
11_2	1	X	X	xkseg	0xC000 0000 0000 0000 through 0xC000 00FF 7FFF FFFF	(2^{40}–2^{31}) bytes
11_2	X	X	X	kseg0	0xFFFF FFFF 8000 0000 through 0xFFFF FFFF 9FFF FFFF	512 Mbytes (2^{29} bytes)

A(63:62)	KX	SX	UX	Segment Name	Address Range	Segment Size
11_2	X	X	X	kseg1	0xFFFF FFFF A000 0000 through 0xFFFF FFFF BFFF FFFF	512 Mbytes (2^{29} bytes)
11_2	X	X	X	ksseg	0xFFFF FFFF C000 0000 through 0xFFFF FFFF DFFF FFFF	512 Mbytes (2^{29} bytes)
11_2	X	X	X	kseg3	0xFFFF FFFF E000 0000 through 0xFFFF FFFF FFFF FFFF	512 Mbytes (2^{29} bytes)

5.3.8.1 32-bit Kernel, User Space (kuseg)

In Kernel mode when accessing User space and ***UX*** = 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.

5.3.8.2 32-bit Kernel, Kernel Space 0 (kseg0)

In Kernel space, 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.

5.3.8.3 32-bit Kernel, 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. All memory accesses to this space are blocking.

5.3.8.4 32-bit Kernel, Supervisor Space (ksseg)

In Kernel mode when accessing Supervisor space and ***SX*** = 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.

5.3.8.5 32-bit Kernel, Kernel Space 3 (kseg3)

In Kernel space, 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.

5.3.8.6 64-bit Kernel, User Space (xkuseg)

In Kernel mode when accessing User space and ***UX*** = 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.

5.3.8.7 64-bit Kernel, Current Supervisor Space (xksseg)

In Kernel mode when accessing Supervisor space and ***SX*** = 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.

5.3.8.8 64-bit Kernel, Physical Spaces (xkphys)

In Kernel space, 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 zero 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 5.5:.

Table 5.5: Cacheability and Coherency Attributes

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, Blocking	0x9000 0000 0000 0000
3	Cacheable, noncoherent	0x9800 0000 0000 0000
4-5	Reserved	0xA000 0000 0000 0000
6	Uncached, non-blocking	0xB000 0000 0000 0000
7	Bypass	0xB800 0000 0000 0000

5.3.8.9 64-bit Kernel, Kernel Space (xkseg)

In Kernel space, 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.

5.3.8.10 64-bit Kernel, Compatibility Spaces

In Kernel space, 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 two bytes of address 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, blocking 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 *kssе*g. This space is independent of the setting of ***KX*** and requires that ***SX***=1.
- *ckseg3*. This 64-bit virtual address space is kernel virtual space, compatible with the 32-bit address model *kseg3*.

5.4 System Control Coprocessor

The System Control Coprocessor (CP0) supports memory management, address translation, exception handling, and other privileged operations. CP0 contains the registers shown in Figure 5.6 plus a 64-entry TLB. The sections that follow describe how the processor uses the memory management-related registers.

Each CP0 register has a unique identifier referred to as the *register number*. For instance, the *PageMask* register is register number 5.

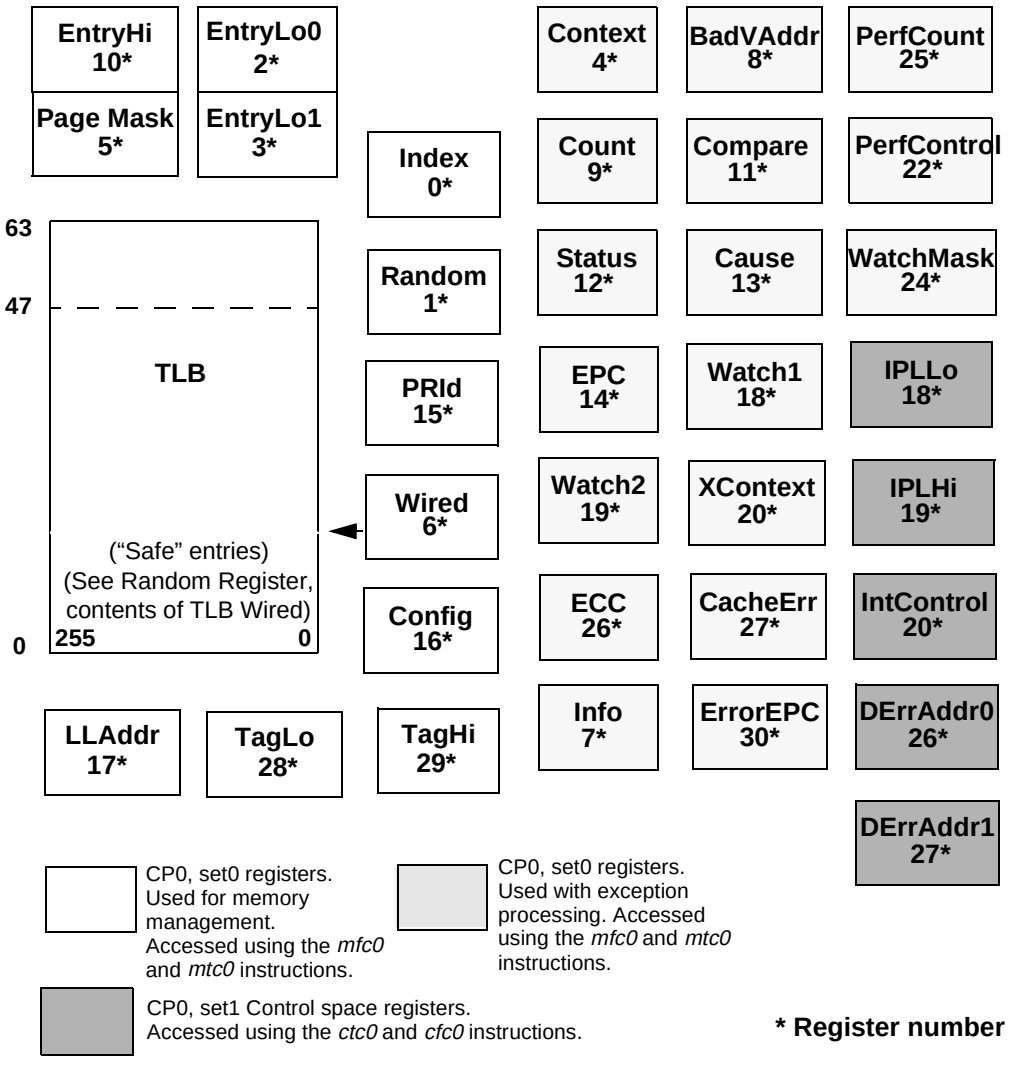


Figure 5.6 CP0 Registers and the TLB

5.4.1 Format of a TLB Entry

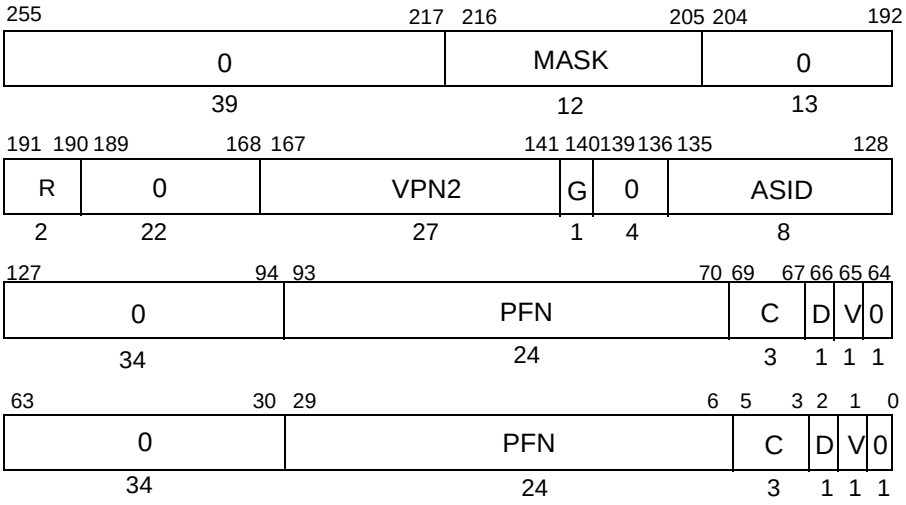
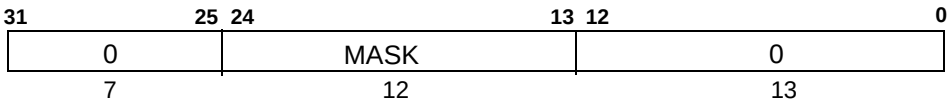


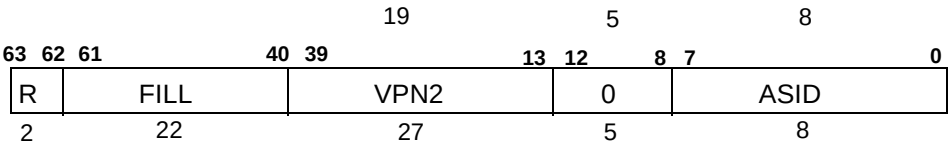
Figure 5.7 Format of a TLB Entry

Figure 5.7 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.



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

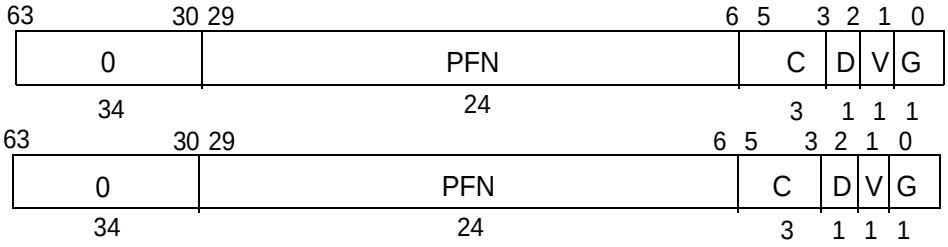
Figure 5.8 PageMask Register Format



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. zero on read; ignored on write.
0..... Reserved. Must be written as zeroes, and returns zeroes when read.

Figure 5.9 EntryHi Register Format

The format of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers are nearly the same as the TLB entry. The one exception is the ***Global*** field (***G*** bit), which is used in the TLB, but is reserved in the *EntryHi* register. Figure 5.8, Figure 5.9, and Figure 5.10 describe the formats for the *PageMask*, *EntryHi*, *EntryLo0/EntryLo1* registers.



PFNPage frame number; the upper bits of the physical address.

C Specifies the TLB page coherency attribute; see Table 5.6:.

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 5.10 EntryLo0 and EntryLo1 Register Formats

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 5.6: shows the coherency attributes selected by the ***C*** bits.

Table 5.6: TLB Page Coherency (C) Bit Values

***C*(5:3) Value**	**Page Coherency Attribute**
0	Cacheable, noncoherent, write-through, no write allocate
1	Cacheable, noncoherent, write-through, write allocate
2	Uncached, blocking
3	Cacheable noncoherent (writeback)
4	Reserved
5	Reserved
6	Uncached, non-blocking
7	Bypass

5.4.2 CP0 Registers

Table 5.7: lists the CP0 registers used by the MMU. The following sections describe each register.

Table 5.7: CP0 Set 0 Memory Management Registers

Reg. No.	Register Name
0	Index
1	Random
2	EntryLo0
3	EntryLo1
5	PageMask
6	Wired
10	EntryHi
15	PrID
16	Config
17	LLAdr
28	TagLo
29	TagHi

5.4.2.1 Index Register (Set 0, 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.

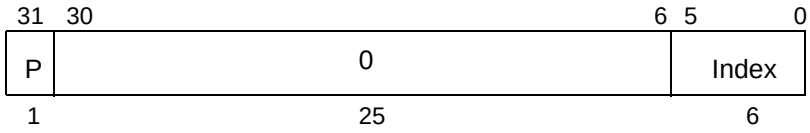


Figure 5.11 Index Register

Figure 5.11 shows the format of the *Index* register; Table 5.8: describes the *Index* register fields.

Table 5.8: Index Register Field Descriptions

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.

5.4.2.2 Random Register (Set 0, Register 1)

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

- A lower bound is set by the number of TLB entries reserved for exclusive use by the operating system (the contents of the *Wired* register).
- An upper bound is set by the total number of TLB entries available (47 maximum when the JTLB is configured as 48 entries, 63 maximum when the JTLB is configured as 64 entries).

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.

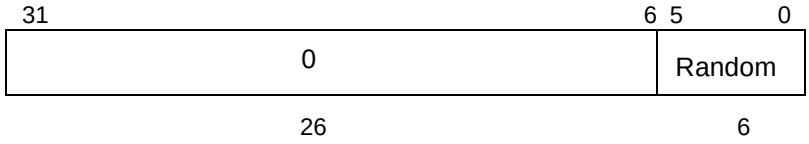


Figure 5.12 Random Register

Figure 5.12 shows the format of the *Random* register. Table 5.9: describes the *Random* register fields.

Table 5.9: Random Register Field Descriptions

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

5.4.2.3 EntryLo0 (Set 0, Register 2) and EntryLo1 (Set 0, Register 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 5.10 on page 41 shows the format of these registers.

5.4.2.4 PageMask Register (Set 0, 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 5.10:, the operation of the TLB is undefined. Figure 5.8 on page 40 shows the format of the *PageMask* register.

Table 5.10: Mask Field Values for Page Sizes

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

5.4.2.5 Wired Register (Set 0, Register 6)

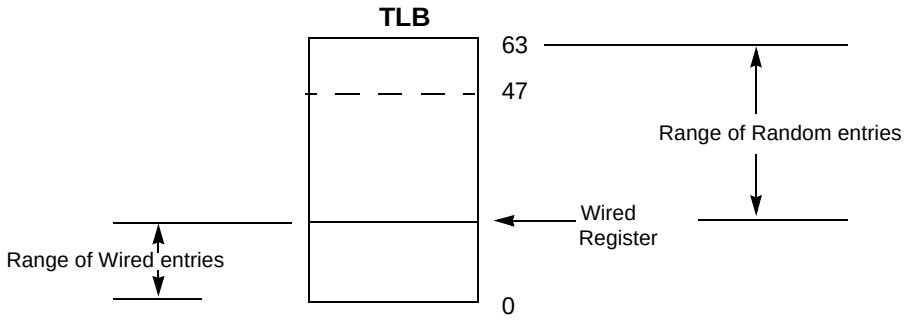


Figure 5.13 Wired Register Boundary

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 5.13. Wired entries are fixed, nonreplaceable entries, which cannot be overwritten by a TLB write random (**TLBWR**) operation. Random entries can be overwritten.

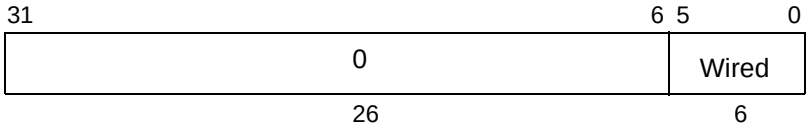


Figure 5.14 Wired Register

The *Wired* register is set to zero on a system reset. Writing this register also sets the *Random* register to the value of its upper bound (see *Random* register, above). Figure 5.14 shows the format of the *Wired* register; Table 5.11: describes the register fields.

Table 5.11: Wired Register Field Descriptions

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

5.4.2.6 EntryHi Register (Set 0, Register 10)

The *EntryHi* register holds the high-order bits of a TLB entry for TLB read and write operations. Figure 5.9 on page 40 shows the format of this register.

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.

5.4.2.7 Processor Revision Identifier (PRId) Register (Set 0, Register 15)

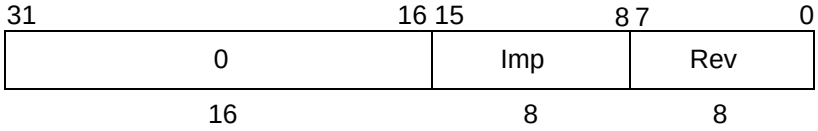


Figure 5.15 Processor Revision Identifier Register Format

The 32-bit, read-only *Processor Revision Identifier* (*PRId*) register contains information identifying the implementation and revision level of the CPU and CP0. Figure 5.15 shows the format of the *PRId* register; Table 5.12: describes the *PRId* register fields.

Table 5.12: PRId Register Fields

Field	Description
Imp	Implementation number = 0x27
Rev	Revision number
0	Reserved. Must be written as zeroes, and returns zeroes when read.

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 RM7000 processor is 0x27. The content of the high-order halfword (bits 31:16) of the register is 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.

5.4.2.8 Config Register (Set 0, Register 16)

The *Config* register specifies various configuration options which can be selected.

Some configuration options, as defined by *Config* bits 31:13,11:3 are read from the initialization mode stream by the hardware during reset and are included in the *Config* register as read-only status bits for the software to access. Other configuration options are read/write (*Config* register bits 12, 3:0) and controlled by software; on reset these fields are undefined.

Certain configurations have restrictions. The *Config* register should be initialized by software before caches are used. Caches should be written back to memory before line sizes are changed, and caches should be reinitialized after any change is made.

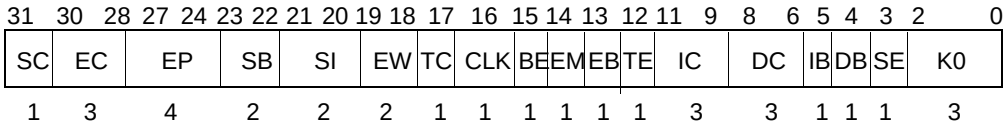


Figure 5.16 Config Register Format

Figure 5.16 shows the format of the *Config* register; Table 5.13: describes the *Config* register fields.

Table 5.13: Config Register Fields

Bit(s)	Field	Description
31	SC	Secondary Cache Present: Value loaded from Mode bit[25]. 0: Secondary Cache present 1: Secondary Cache not present.
30:28	EC	System clock ratio: Value loaded from Mode bits[7:5] 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 or 2.5 4: processor clock frequency divided by 6 5: processor clock frequency divided by 7 or 3.5 6: processor clock frequency divided by 8 7: processor clock frequency divided by 9 or 4.5
27:24	EP	Transmit data pattern (pattern for write-back data): Value loaded from Mode bits [4:1] 0: DDDDDoubleword every cycle 1: DDxDDx2 Doublewords every 3 cycles 2: DDxxDDxx2 Doublewords every 4 cycles 3: DxDxDxDx2 Doublewords every 4 cycles 4: DDxxxDDxxx2 Doublewords every 5 cycles 5: DDxxxxDDxxxx2 Doublewords every 6 cycles 6: DxxDxxDxxDxx2 Doublewords every 6 cycles 7: DDxxxxxxDDxxxxxx2 Doublewords every 8 cycles 8: DxxxDxxxDxxxDxxx2 Doublewords every 8 cycles
23-22	SB	Secondary Cache block size. (fixed at 8 words). 0: 4 words 1: 8 words 2: 16 words 3: 32 words
21:20	SI	System Configuration Identifier: Value loaded from Mode bits[17:16]. System designer is free to choose encoding as the CPU is not affected by these bits.
19:18	EW	SysAD bus width. Set to 64-bit by default. 00: 64-bit 01, 10, 11:Reserved
17	TC	Tertiary Cache Present: Value loaded from Mode bit 12. 0: Tertiary Cache present 1: Tertiary Cache not present.
16	CLK	Identifies the value of mode bit 20, set during the boot time initialization process. A value of 1 indicates that the half-integer clock multiplier is enabled.
15	BE	Big Endian Mode: Value loaded from the OR function of Mode bit 8 and the BigEndian pin. 0: Little Endian 1: Big Endian

Bit(s)	Field	Description
14	EM	ECC mode enable. Fixed to parity by default. 0: ECC mode 1: Parity mode
13	EB	Block ordering. Fixed to sub-block by default. 0: Sequential 1: Sub-block
12	TE	Tertiary Cache Enable: Setting this bit enables the external tertiary cache controller. (software writable)
11:9	IC	Primary Instruction cache Size (I-cache size = 2^{12+IC} bytes). Fixed to 16 Kbytes.
8:6	DC	Primary Data cache Size (D-cache size = 2^{12+DC} bytes). Fixed to 16 Kbytes.
5	IB	Primary Instruction cache line size. Fixed to 32 bytes. 0: 16 bytes 1: 32 bytes
4	DB	Primary Data cache line size. Fixed to 32 bytes. 0: 16 bytes 1: 32 bytes
3	SE	Secondary Cache Enable: Setting this bit enables the secondary cache. (software writable)
2:0	K0	*kseg0* coherency algorithm (software writable)

5.4.2.9 Load Linked Address (LLAddr) Register (Set 0, 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.

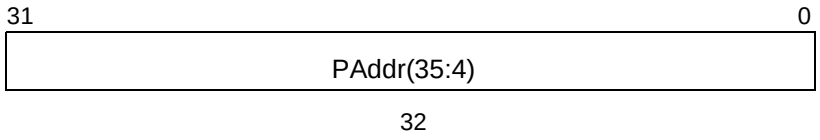


Figure 5.17 LLAddr Register Format

Figure 5.17 shows the format of the *LLAddr* register; ***PAddr*** represents bits of the physical address, PA(35:4).

5.4.2.10 TagLo Register (Set 0, Register 28)

The *TagLo* register is a 32-bit read/write registers that state bits and control information for the cache access The *TagLo* register is written by the **CACHE** and **MTC0** instructions.

The ***P*** field of the *TagLo* register is ignored on Index Store Tag operations. Parity is computed by the store operation.

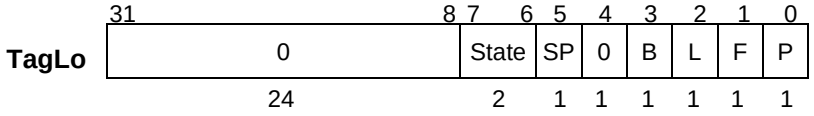


Figure 5.18 TagLo Register Formats (Primary Data Cache)

Note: Figure 5.18 shows the format of the TagLo register when accessing the primary data cache. Table 5.14: lists the corresponding field definitions of the TagLo register (primary data cache). Figure 5.19 and Table 5.15: shows the format when accessing the primary instruction cache. Figure 5.20 and Table 5.16: shows the format when accessing the secondary cache. Figure 5.21 and Table 5.17: shows the format when accessing the tertiary cache.

Table 5.14: TagLo Register Fields (Primary Data Cache)

Bit(s)	Field	Description
7:6	State	Primary Data Cache State: Specifies the primary data cache state. 00: Invalid 01: Pending 10: reserved 11: Data Valid
5	SP	Parity: Even parity bit for State field
3	B	Bypass. This bit is set when the secondary and the tertiary cache bypass coherency is used.
2	L	Lock: This bit is set when the access is to a locked cache set.
1	F	Fill bit used for FIFO set selection refill algorithm.
0	P	Even parity for the PTag, B, L, and F bits.

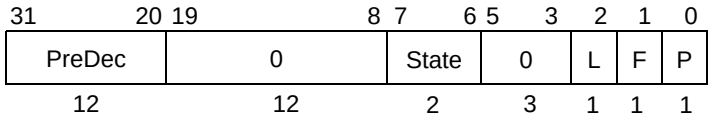


Figure 5.19 TagLo Register Format (Primary Instruction Cache)

Table 5.15: TagLo Register Fields (Primary Instruction Cache)

Bit(s)	Field	Description
31:20	PreDec	Instruction type predecode bits: Used by hardware to aid in dispatching instructions to the correct pipeline. Software should **never** modify these bits. If these bits are modified, chip behavior becomes undefined and unpredictable.
7:6	State	Primary Instruction Cache State: Specifies the primary instruction cache state. 00: Invalid 01: Pending 10: Instruction Valid 11: reserved
2	L	Lock: This bit is set when the access is to a locked cache set.
1	F	Fill bit used for FIFO set selection refill algorithm
0	P	Even parity for the PreDec, PTag, State, L, and F bits.

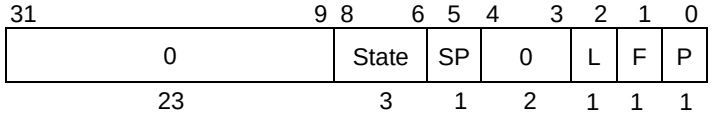


Figure 5.20 TagLo Register Format (Secondary Cache)

Table 5.16: TagLo Register Fields (Secondary Cache)

Bit(s)	Field	Description
8:6	State	Specifies the secondary cache state. 000: Invalid 001: Reserved 010: Reserved 011: Reserved 100: Clean Exclusive 101: Dirty Exclusive 110: Pending 111: Dirty Exclusive DCache
5	SP	Parity: Even parity bit for State field
2	L	Lock: This bit is set when the access is to a locked cache set.
1	F	Fill bit used for FIFO set selection refill algorithm.
0	P	Even parity for the STag, L, and F bits.

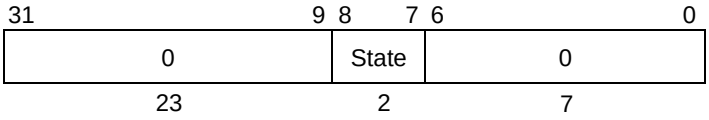


Figure 5.21 TagLo Register Format (Tertiary Cache)

Table 5.17: TagLo Register Fields (Tertiary Cache)

Bit(s)	Field	Description
8:7	State	Specifies the tertiary cache state 00: Invalid 01: Reserved 10: Valid 11: Reserved

5.4.2.11 TagHi Register (Set 0 Register 29)

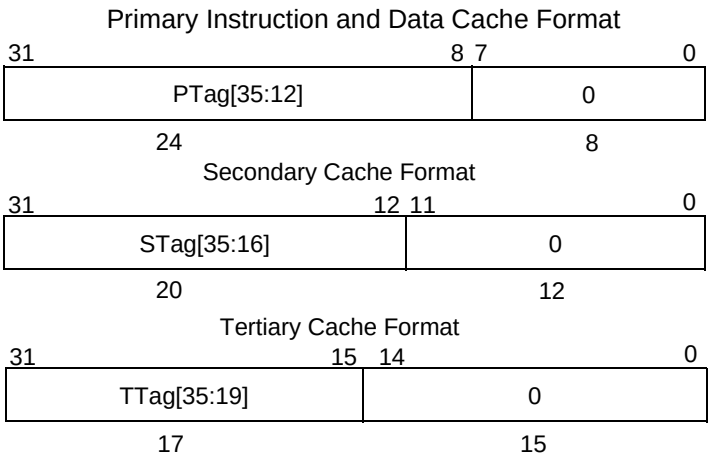


Figure 5.22 TagHi Register Formats

The *TagHi* register is a 32-bit read/write register that holds either the primary cache tag, secondary cache tag, or tertiary cache tag during cache initialization, cache diagnostics, or cache error processing. The *tag* registers are written by the **CACHE** and **MTC0** instructions. Figure 5.22 shows the *TagHi* register formats for the primary caches, secondary cache, and tertiary cache. The ***PTag*** field of each register stores the physical address.

5.4.3 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.

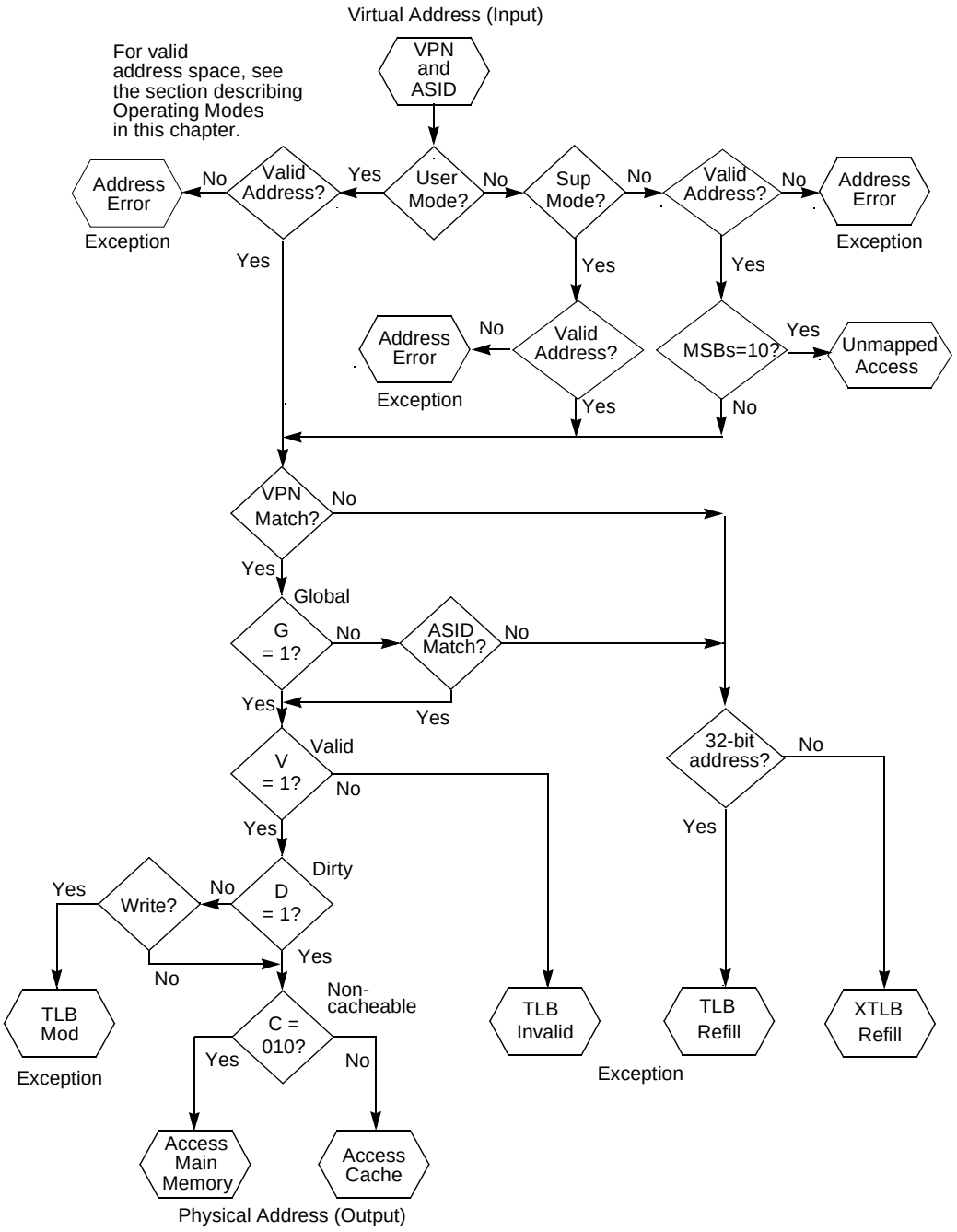


Figure 5.23 TLB Address Translation

Figure 5.23 illustrates the TLB address translation process.

5.4.4 TLB Exceptions

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

5.4.5 TLB Instructions

Table 5.18: lists the instructions that the CPU provides for working with the TLB.

Table 5.18: TLB Instructions

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

5.5 Code Examples

The first example is how to set up one TLB entry to map a pair of 4KB pages. A real time kernel might do something similar. Such simple kernels are only using the MMU for memory protection so the static mapping is sufficient. In statically mapped systems, all TLB exceptions are considered error conditions (access violations).

```
mtc0    r0,C0_WIRED          # make all entries available
                             # to random replacement
li      r2, (vpn2<<13)|(asid & 0xff);
mtc0    r2, C0_ENHI          # set the virtual address

li      r2, (epfn<<6)|(coherency<<3)|(Dirty<<2)|Valid<<1|Global)
mtc0    r2, C0_ENLO0         # set the physical address for the even
                             # page

li      r2, (opfn<<6)|(coherency<<3)|(Dirty<<2)|Valid<<1|Global)
mtc0    r2, C0_ENLO1         # set the physical address for the odd
                             # page

li      r2, 0                # set the page size to 4KB
mtc0    r2,C0_PAGEMASK

li      r2, index_of_some_entry    # needed for tlbwi only
mtc0    r2, C0_INDEX         # needed for tlbwi only

nop
nop
nop
nop
tlbwr                        # or tlbwi
```

True virtual memory operating systems (like UNIX) use the MMU for both memory protection and swapping pages between main memory and a long term storage device. This mechanism allows programs to address more memory than is physically allocated on the system. This on-demand paging mechanism requires dynamic mapping of pages. The dynamic mapping is implemented through the different types of MMU exceptions. The TLB Refill exception is the most common exception within such systems. The following is an example of a possible TLB Refill exception handler.

```
refill_exception:
        mfc0   k0,C0_CONTEXT
        sra    k0,k0,1             # index into the page table
        lw     k1,0(k0)            # read page table
        lw     k0,4(k0)
        sll    k1,k1,6
        srl    k1,k1,6
        mtc0   k1,C0_TLBLO0
        sll    k0,k0,6
        srl    k0,k0,6
        mtc0   k0,C0_TLBLO1
        nop
        nop
        nop
        nop
        tlbwr                      # write a random entry
        eret
```

This exception handler is kept very simple and short as it is executed often enough to affect system performance. This is the reason that the TLB Refill exception is allocated its own exception vector. This code assumes that the required mapping has been already set up in the main page table held in main memory. If this is not true then a second exception, a TLB Invalid exception, will be taken after the **ERET** instruction. The TLB Invalid exception happens much less frequently, which is fortunate as it has to calculate the desired mapping, possibly reading portions of the page table from long term storage. The TLB Mod exception is used to implement read-only pages and to mark which pages have been modified for process cleanup code.

To further protect different processes and users from each other, true virtual memory operating systems execute user programs in user mode. Below is an example of how to enter user mode from kernel mode.

```
        mtc0   r10, C0_EPC         # assume r10 holds desired usermode address
        mfc0   r1, C0_SR           # get current value of Status register
        and    r1,r1, ~(SR_KSU || SR_ERL)       # clear KSU and ERL field
        or     r1, r1, (KSU_USERMODE || SR_EXL) # set usermode and EXL bit
        mtc0   r1, C0_SR
        nop
        nop
        nop
        nop
        eret                       # jump to user mode
```

Section 6 Floating-Point Unit

This section describes the floating-point unit (FPU) of the RM7000 processor, including the programming model, instruction set and formats, and the pipeline.

The FPU, with associated system software, fully conforms to the requirements of ANSI/IEEE Standard 754–1985, *IEEE Standard for Binary Floating-Point Arithmetic*. In addition, the MIPS architecture fully supports the recommendations of the standard and precise exceptions.

6.1 Overview

The FPU operates as a coprocessor for the CPU (assigned coprocessor label *CP1*) and extends the CPU instruction set to perform arithmetic operations on floating-point values.

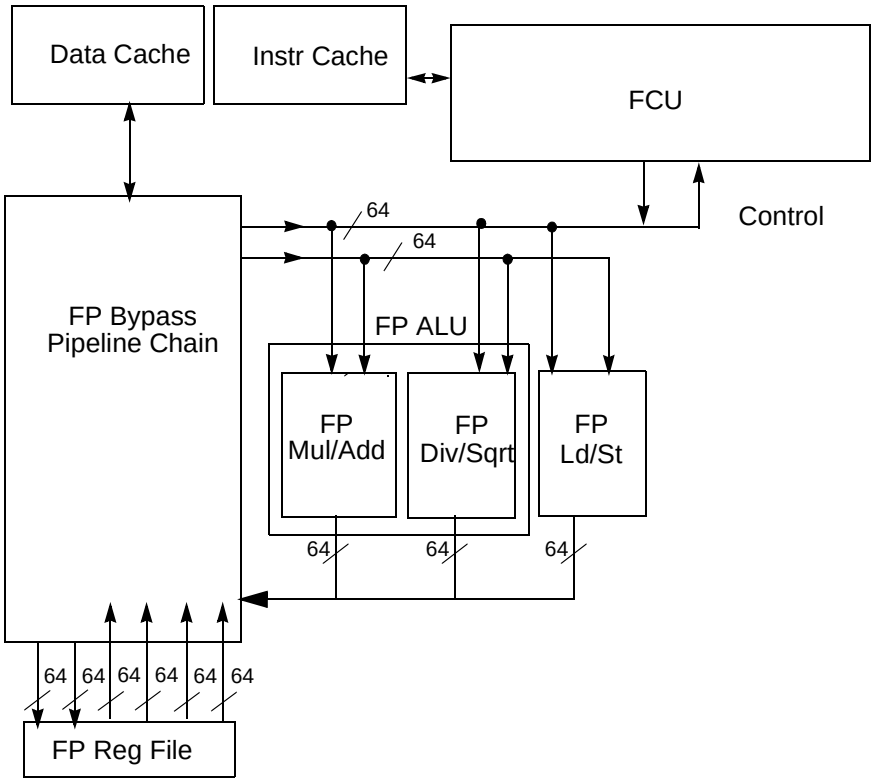


Figure 6.1 FPU Functional Block Diagram

Figure 6.1 illustrates the functional organization of the FPU.

6.2 FPU Features

This section briefly describes the operating model, the load/store instruction set, and the coprocessor interface in the FPU. A more detailed description is given in the sections that follow.

- **Full 64-bit Operation**. When the ***FR*** bit in the CPU *Status* register equals 0, the FPU is in 32-bit mode and contains thirty-two 32-bit registers that hold single- or, when used in pairs, double-precision values. When the ***FR*** bit in the CPU *Status* register equals 1, the FPU is in 64-bit mode and the registers are expanded to 64 bits wide. Each register can hold single- or double-precision values. The FPU also includes a 32-bit *Control/Status* register that provides access to all IEEE-Standard exception handling capabilities.
- **Load and Store Instruction Set**. Like the CPU, the FPU uses a load/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.

6.3 Exceptional Floating Point Input Values

The RM7000 processor contains an enhanced mechanism for handling exceptional floating point input values. This mechanism helps to minimize the number of conditions under which an FP exception is generated, thus improving overall floating point performance. The RM7000 passes Quiet-Not-a-Number (QNaN) outputs from any QNaN inputs. This is unlike previous generations of MIPS processors which always take an *FP Unimplemented* exception when encountering a QNaN input.

The RM7000 generates integer QNaN values when converting a floating point QNaN, SNaN, or INF value. In addition, the RM7000 generates integer QNaN values when converting an input value that overflows the integer format and the *Invalid* exception is disabled.

The RM7000 generates the correct output for the various combinations of infinity and denormal inputs as shown below. Previous MIPS implementations would take an *FP Unimplemented* exception if any input was a denormal.

1. infinity * denormal = infinity
17. infinity + denormal = infinity
18. denormal/infinity = zero
19. infinity/denormal = infinity

When the ***FS*** bit in the *Floating-Point Control* (*FCR31*) register is set, the RM7000 treats denormal inputs as zero. Denormal outputs are flushed to zero and in most cases do not cause an exception due to denormals. This is different from previous generations of MIPS processors, which only flushed denormal outputs to zero when this bit was set.

Refer to section 6.11, “Special Operand Handling”, for more information on any of the special operand values.

6.4 FPU Programming Model

This section describes the set of FPU registers and their data organization. The FPU registers include *Floating-Point General Purpose* registers *(FGR*s) and two control registers: *Control/Status* and *Implementation/Revision*.

6.5 Floating-Point General Registers (FGR)

The FPU has a set of *Floating-Point General Purpose* 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 6.2.
- 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 6.2.

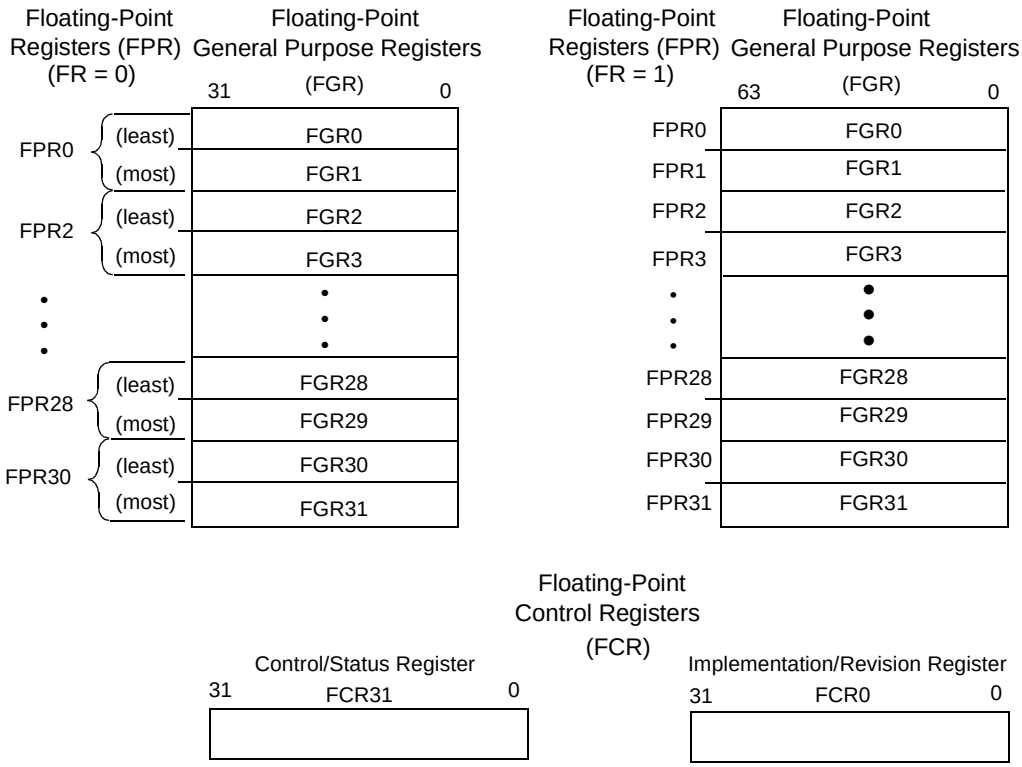


Figure 6.2 FPU Registers

6.6 Floating-Point Registers (FPR)

The FPU provides:

- 16 *Floating-Point* registers (*FPR*s) when the ***FR*** bit in the *Status* register equals 0, or
- 32 *Floating-Point* registers (*FPR*s) when the ***FR*** bit in the *Status* register equals 1.

These 64-bit registers hold floating-point values during floating-point operations and are physically formed from the *General Purpose* registers (*FGRs*). When the ***FR*** bit in the *Status* register equals 1, the *FPR* references a single 64-bit *FGR.*

The *FPR*s hold values in either single- or double-precision floating-point format. If the ***FR*** bit equals 0, only even numbers (the *least* register, as shown in Figure 6.2) can be used to address *FPR*s. When the ***FR*** bit is set to a 1, all *FPR* register numbers are valid.

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

6.7 Floating-Point Control Registers (FCR)

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

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

Table 6.1: lists the assignments of the *FCR*s.

Table 6.1: Floating-Point Control Register Assignments

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

6.7.1 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.

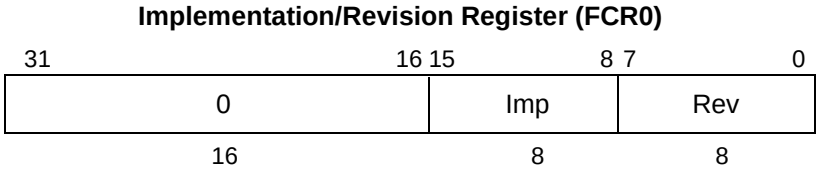


Figure 6.3 Implementation/Revision Register (FCR0)

Figure 6.3 shows the layout of the register; Table 6.2: describes the *Implementation and Revision* register (*FCR0*) fields.

Table 6.2: FCR0 Fields

Field	Description
Imp	Implementation number (0x27)
Rev	Revision number in the form of *y.x*
0	Reserved. Must be written as zeroes, and returns zeroes when read.

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, QED 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.

6.7.2 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.

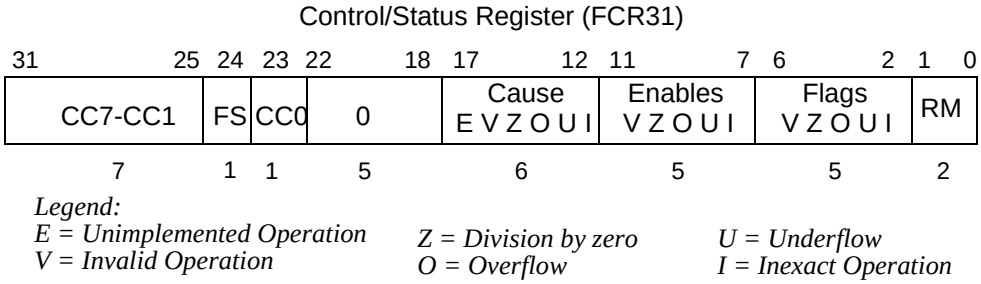


Figure 6.4 FP Control/Status Register Bit Assignments

Figure 6.4 shows the format of the *Control/Status register*, and Table 6.3: describes the *Control/Status* register fields. Figure 6.5 shows the *Control/Status* register ***Cause***, ***Flag***, and ***Enable*** fields.

Table 6.3: Control/Status Register Fields

Field	Description
CC7-CC1	Condition bits 7-1
FS	When set, denormalized inputs and results are flushed to 0 instead of causing an unimplemented operation exception. On the RM7000, even if the FS bit is set, if a madd, msub, nmadd or nmsub instruction encounters a denormalized result during the multiply portion of the calculation, an unimplemented operation exception is always taken.
CC0	Condition bit 0. 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.

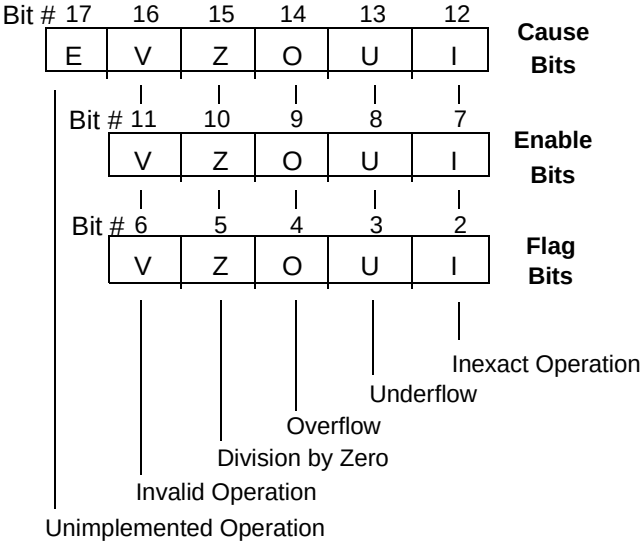


Figure 6.5 Control/Status Register Cause, Flag, and Enable Fields

6.7.2.1 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.

6.7.2.2 IEEE Standard 754

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

6.7.2.3 Control/Status Register FS Bit

When the ***FS*** bit is set, denormalized results are flushed to 0 instead of causing an unimplemented operation exception. Both the underflow and inexact exceptions should be disabled by the user for this behavior. Denormalized inputs are flushed to zero. This new behavior is specific to the RM7000 processor.

Only the arithmetic operations are affected by the ***FS*** bit.

6.7.2.4 Control/Status Register Condition Bits

When a floating-point Compare operation takes place, the result is stored at one of the eight Conditions bits (bits 31-25,23) to save or restore the state of the condition. The condition bit is set if the condition is true; the bit is cleared if the condition is false. The Condition bits are affected only by *Compare* and *Move Control To FPU* instructions.

6.7.2.5 Control/Status Register Cause, Flag, and Enable Fields

Figure 6.5 illustrates the ***Cause***, ***Flag***, and ***Enable*** fields of the *Control/Status* register.

6.7.2.5.1 Cause Bits

Bits 17:12 in the *Control/Status* register contain ***Cause*** bits, as shown in Figure 6.5, which reflect the results of the most recently executed instruction. The ***Cause*** bits are a logical extension of the *Cause* register; they identify the exceptions raised by the last floating-point operation and raise an interrupt or exception if the corresponding enable bit is set. If more than one exception occurs on a single instruction, each appropriate bit is set.

The ***Cause*** bits are written by each floating-point operation (but not by load, store, or move operations). The ***Unimplemented Operation*** (***E***) bit is set to a 1 if software emulation is required, otherwise it remains zero. The other bits are set or cleared 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.

6.7.2.5.2 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 those ***Cause*** bits whose ***Enable*** bits are cleared, 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.

6.7.2.5.3 Flag Bits

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 Control To Coprocessor 1* (**CTC1**) instruction.

When a floating-point exception is taken, the flag bits are not set by the hardware; floating-point exception software is responsible for setting these bits before invoking a user handler.

6.7.2.6 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 6.4:, these bits specify the rounding mode that the FPU uses for all floating-point operations.

Table 6.4: Rounding Mode Bit Decoding

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.

6.8 Floating-Point Formats

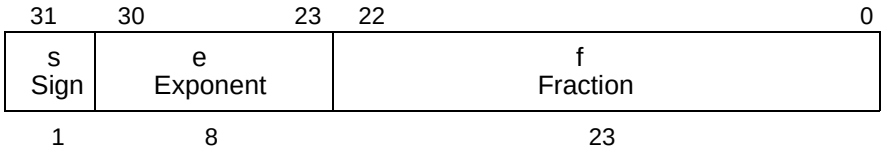


Figure 6.6 Single-Precision Floating-Point Format

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 6.6.

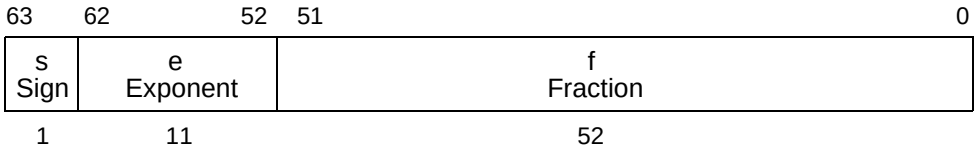


Figure 6.7 Double-Precision Floating-Point Format

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 6.7.

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 $\pm^{\infty}$ and NaNs [Not a Number])

For single- and double-precision formats, each representable nonzero numerical value has just one encoding.

For single- and double-precision formats, the value of a number, *v*, is determined by the equations shown in Table 6.5:.

Table 6.5: Calculating Values in Single and Double-Precision Formats

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

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 6.6: defines the values for the format parameters; minimum and maximum floating-point values are given in Table 6.7:.

Table 6.6: Floating-Point Format Parameter Values

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 6.7: Minimum and Maximum Floating-Point Values

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

6.9 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 6.8 illustrates binary fixed-point formats; Table 6.8: lists the binary fixed-point format fields.

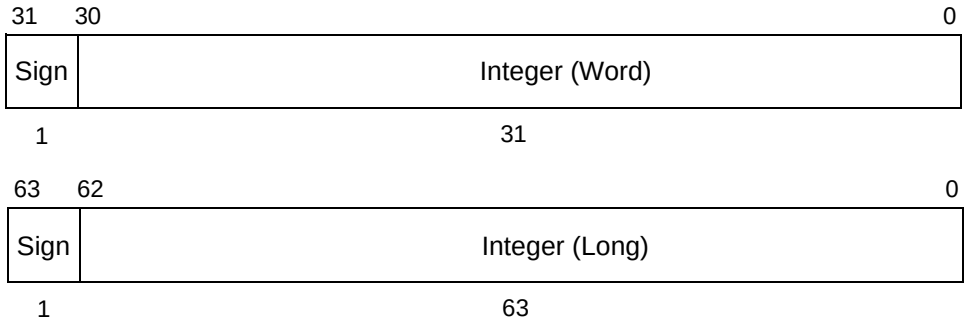


Figure 6.8 Binary Fixed-Point Formats

Field assignments of the binary fixed-point format are described in Table 6.8:.

Table 6.8: Binary Fixed-Point Format Fields

Field	Description
sign	sign bit
integer	integer value

The integer *Quiet Not a Number* (QNaN) value is represented as the largest positive integer. For the word format the QNaN value is 0x7FFF FFFF. For the long format the QNaN value is 0x 7FFF FFFF FFFF FFFF.

6.10 Floating-Point Instruction Set Overview

All FPU instructions are 32 bits long and are 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 6.9: through Table 6.12:, 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.

Table 6.9: FPU Instruction Summary: Load, Move and Store Instructions

OpCode	Description
CFC1	Move Control Word From FPU
CTC1	Move Control Word To FPU
DMFC1	Doubleword Move From FPU
DMTC1	Doubleword Move To FPU
LDC1	Load Doubleword to FPU
LDXC1	Load Doubleword Indexed to FPU
LWC1	Load Word to FPU
LWXC1	Load Word Indexed to FPU
MFC1	Move Word From FPU
MTC1	Move Word To FPU
PREFX	Prefetch Indexed - Register + Register
SDC1	Store Doubleword From FPU
SDXC1	Store Doubleword Indexed From FPU
SWC1	Store Word from FPU
SWXC1	Store Word Indexed from FPU
MOVF	GPR Conditional Move on FP False
MOVF.fmt	FP Conditional Move on FP False
MOVN.fmt	FP Conditional Move on GPR non-zero
MOVT	GPR Conditional Move on FP True
MOVT.fmt	FP Conditional Move on FP True
MOVZ.fmt	FP Conditional Move on GPR zero

Table 6.10: FPU Instruction Summary: Conversion Instructions

OpCode	Description
CEIL.W.fmt	Floating-point Ceiling to 32-bit Fixed Point
CEIL.L.fmt	Floating-point Ceiling to 64-bit Fixed Point
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
FLOOR.W.fmt	Floating-point Floor to 32-bit Fixed Point
FLOOR.L.fmt	Floating-point Floor 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

Table 6.11: FPU Instruction Summary: Computational Instructions

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
MADD.fmt	Floating-point Multiply-Add
MSUB.fmt	Floating-point Multiply-Subtract
NMADD.fmt	Floating-point Negated Multiply-Add
NMSUB.fmt	Floating-Point Negated Multiply-Subtract
RECIP.fmt	Floating-point Reciprocal
RSQRT.fmt	Floating-point Reciprocal Square Root

Table 6.12: FPU Instruction Summary: Compare and Branch 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

6.10.1 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 6.9:.

6.10.1.1 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**/**LWXC1**) or Store Word From Coprocessor 1 (**SWC1**/**SWXC1**) instructions, which reference a single 32-bit word of the FPU general registers
- Load Doubleword (**LDC1**/**LDXC1**) or Store Doubleword (**SDC1**/**SDXC1**) 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.

6.10.1.2 Transfers Between FPU and CPU

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

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

Like the floating-point load and store operations, these operations perform no format conversions and never cause floating-point exceptions. The instruction following a **MTC1** or **DMTC1** instruction can use the contents of the modified register. There is no delay slot for a **MTC1** or **DMTC1** instruction.

6.10.1.3 Load Delay and Hardware Interlocks

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

6.10.1.4 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.

6.10.1.5 Endianness

Endianness refers to the order that bytes are stored in memory. In big-endian systems the byte reached with the lowest byte address in a half-word, word, or double-word datum, will be the left-most byte. In a little-endian system the byte reached with the lowest byte address in a half-word, word, or double-word datum, will be the right most byte.

Regardless of byte-numbering order (endianness) of the data, the address specifies the byte in the addressed field with the lowest byte address. The other byte or bytes in the datum can be reached with positive offsets from this base address.

6.10.2 Floating-Point Conversion Instructions

Conversion instructions perform conversions between the various data formats such as single- or double-precision, fixed- or floating-point formats.

6.10.3 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
- 4-Operand Register-Type instructions, which perform floating-point multiply-add operations.

For a detailed description of each instruction, refer to the MIPS IV instruction set manual.

6.10.3.1 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 MIPS IV instruction set manual.

6.10.3.2 Floating-Point Compare Instructions

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 6.13: lists the mnemonics for the compare instruction conditions.

Table 6.13: Mnemonics and Definitions of Compare Instruction Conditions

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

6.11 Special Operand Handling

In most situations the RM7000 can pass QNaN input operands to the output without taking an unimplemented FP exception. It can also recognize situations where denormals and infinity operands would produce exact results.

Table 6.15: shows how the RM7000 behaves with different combinations of these floating-point special operands. Table 6.14: defines the abbreviations used in Table 6.15:.

Table 6.14: Operand Abbreviations

Abbreviation	Definition
SNaN	Signalling Not a Number
SNaN-B	Not a SNaN
QNaN	Quiet Not a Number
NaN	Not a Number
NaN-B	Not a NaN
Inf	Infinity
Inf-B	Not Infinity, Not NaN
Max	Maximum representable number
Min	Minimum representable number
Norm	Normal representable FP number
DeNorm	DeNormal FP number
Zero	Signed zero
Zero-B	Non-zero normal number
*	Any value

Table 6.15: Floating-Point Special Operands

Operation	OpA	OpB	OpC	Exception Enabled Result	Exception Disabled Result	FS Mode Result	Remarks
*	SNaN	*	*	Invalid	Gen QNaN		SNaN has the highest priority. QNaN has the next highest priority.
*	*	SNaN	*	Invalid	Gen QNAN		
*	QNaN	SNaN-B	SNaN-B	Pass QNaN	Pass QNaN		
*	SNaN-B	QNaN	SNaN-B	Pass QNaN	Pass QNaN		
Add-Subtract (OpA ± OpB)							
Add Sub	Inf	Norm		OpA	OpA		Treat denorm as zero in FS mode.
Add Sub	Inf	DeNorm		OpA	OpA		
Add Sub	DeNorm	DeNorm		Unimple-mented	Unimple-mented	± 0	
Add Sub	DeNorm	Norm		Unimple-mented	Unimple-mented	OpB	
Add Sub	Norm	DeNorm		Unimple-mented	Unimple-mented	OpA	
Add	+Inf	-Inf		Invalid	Gen QNaN		
Add	-Inf	+Inf		Invalid	Gen QNaN		
Sub	+Inf	+Inf		Invalid	Gen QNaN		
Sub	-Inf	-Inf		Invalid	Gen QNaN		

Operation	OpA	OpB	OpC	Exception Enabled Result	Exception Disabled Result	FS Mode Result	Remarks
Multiply-Accumulate (OpA * OpB) ± OpC							
MAdd MSub	Zero-B	Zero-B	NaN	Unimple-mented	Unimple-mented	Unimple-mented	Cannot determine flags
MAdd MSub	QNaN	*	SNaN	Invalid	Gen QNaN		Sets flags invalid
MAdd MSub	Inf	*	SNaN	Invalid	Gen QNaN		Invalid flag set
MAdd MSub	Zero	*	SNaN	Invalid	Gen QNaN		Invalid flag set
MAdd MSub	DeNorm	DeNorm	SNaN	Unimple-mented	Unimple-mented	Gen QNaN	Sets invalid in FS mode
MAdd MSub	DeNorm	Zero-B	SNaN	Unimple-mented	Unimple-mented	Gen QNaN	Invalid, sets V
MAdd MSub	Inf	Zero	*	Invalid	Gen QNaN		Invalid flag set
MAdd MSub	Inf	Inf	QNaN	Pass QNaN	Pass QNaN		
MAdd MSub	Inf	DeNorm	QNaN	Pass QNaN	Pass QNaN	Gen QNaN	Invalid flag set in FS mode
MAdd MSub	Inf	Zero-B	QNaN	Pass QNaN	Pass QNaN		No flags
MAdd MSub	Inf-B	Zero	QNaN	Pass QNaN	Pass QNaN		
MAdd MSub	Zero	Denorm	QNaN	Pass QNaN	Pass QNaN		
MAdd MSub	DeNorm	DeNorm	QNaN	Unimple-mented	Unimple-mented	Pass QNaN	
MAdd MSub	DeNorm	Zero-B	QNaN	Unimple-mented	Unimple-mented	Pass QNaN	Associate with pass-ing NaN
MAdd MSub	Inf	Zero-B	Inf-B	± Inf	± Inf		
MAdd MSub	Inf	Inf	Inf-B	± Inf	± Inf		
MAdd MSub	Inf	DeNorm	Inf-B	± Inf	± Inf	Gen QNaN	Invalid flag set in FS mode
MAdd MSub	Inf-B	Inf-B	DeNorm	Unimple-mented	Unimple-mented	OpA * OpB	
MAdd MSub	Zero	Inf-B	Norm	± OpC	± OpC	± OpC	
MAdd MSub	DeNorm	Inf-B	Norm	Unimple-mented	Unimple-mented	± OpC	In FS mode
MAdd MSub	Zero	Inf-B	Inf	± Inf	± Inf	± Inf	
MAdd MSub	DeNorm	Inf-B	Inf	Unimple-mented	Unimple-mented	± Inf	In FS mode
MAdd	+Inf	+NaN-B	-Inf	Invalid	Gen QNaN		Sets invalid flag
MAdd	-Inf	-NaN-B	-Inf	Invalid	Gen QNaN		Sets invalid flag
MAdd	-Inf	+NaN-B	+Inf	Invalid	Gen QNaN		Sets invalid flag
MAdd	+Inf	-NaN-B	+Inf	Invalid	Gen QNaN		Sets invalid flag

Operation	OpA	OpB	OpC	Exception Enabled Result	Exception Disabled Result	FS Mode Result	Remarks
MSub	+Inf	+NaN-B	+Inf	Invalid	Gen QNaN		Sets invalid flag
MSub	-Inf	-NaN-B	+Inf	Invalid	Gen QNaN		Sets invalid flag
MSub	-Inf	+NaN-B	-Inf	Invalid	Gen QNaN		Sets invalid flag
MSub	+Inf	-NaN-B	-Inf	Invalid	Gen QNaN		Sets invalid flag
Divide - OpA / OpB Reciprocal - 1 / OpA Square Root - sqrt(OpA) Reciprocal Square Root - 1 / sqrt(OpA)							
Div	Inf	Inf		Invalid	Gen QNaN		Sets invalid flag
Div	Inf	Inf-B		± Inf	± Inf	± Inf	
Div	Zero	Zero		Invalid	Gen QNaN		Sets invalid flag
Div	Zero	DeNorm		± 0	± 0	Gen QNaN	Sets V cause in FS mode
Div	Zero	Zero-B		± 0	± 0		
Div	Zero	Inf		± 0	± 0		
Div	DeNorm	Zero		Div/Zero	± Inf	Gen QNaN	Sets V cause in FS mode
Div	DeNorm	DeNorm		Unimple-mented	Unimple-mented	Gen QNaN	Sets V cause in FS mode
Div	DeNorm	Inf		± 0	± 0	± 0	
Div	Zero-B	Zero		Div/Zero	± Inf		Sets Z
Div	Zero-B	DeNorm		Unimple-mented	Unimple-mented	± Inf	Sets Z cause in FS mode
Recip	Zero			Div/Zero	± Inf		
Recip	DeNorm			Unimple-mented	Unimple-mented	± Inf	Sets Z cause in FS mode
Sqrt	DeNorm			Unimple-mented	Unimple-mented	± 0	
Sqrt	-Norm			Invalid	Gen QNaN		
Rsqrt	Zero			Div/Zero	± Inf		Sets Z
Rsqrt	DeNorm			Unimple-mented	Unimple-mented	± Inf	Sets Z cause in FS mode
Multiply - OpA * OpB							
Mul	Inf	Zero		Invalid	Gen QNaN		
Mul	Inf	DeNorm		± Inf	± Inf	Gen QNaN	sets V cause in FS mode
Mul	Inf	Zero-B		± Inf	± Inf		
Mul	DeNorm	DeNorm		Unimple-mented	Unimple-mented	± 0	Treat denorm as 0
Mul	DeNorm	Norm		Unimple-mented	Unimple-mented	± 0	in FS mode
Mul	Zero	Inf-B		± 0	± 0	± 0	

Operation	OpA	OpB	OpC	Exception Enabled Result	Exception Disabled Result	FS Mode Result	Remarks
Convert - S ← D							
cvt.s.d	Zero			Zero	Zero		
cvt.s.d	DeNorm			Unimple-mented	Unimple-mented	Zero	
cvt.s.d	SNaN			Invalid	Gen QNaN (single)		
cvt.s.d	QNaN			Unimple-mented	Unimple-mented		
cvt.s.d	Inf			Inf (single)	Inf (single)		
Convert - D ← S							
cvt.d.s	Zero			Zero	Zero		
cvt.d.s	DeNorm			Unimple-mented	Unimple-mented	Zero	
cvt.d.s	SNaN			Invalid	Gen QNaN (double)		
cvt.d.s	QNaN			Unimple-mented	Unimple-mented		
cvt.d.s	Inf			Inf (double)	Inf (double)		
Convert - W ← S							
cvt.w.s	Zero			Zero	Zero		
cvt.w.s	DeNorm			Unimple-mented	Unimple-mented	Zero	
cvt.w.s	SNaN			Invalid	Gen QNaN (word)		
cvt.w.s	QNaN			Invalid	Gen QNaN (word)		
cvt.w.s	Inf			Invalid	Gen QNaN (word)		
cvt.w.s	Norm			Invalid	Gen QNaN (word)		On overflow (in > 2^{31} -1) or (in < -2^{31})
Convert - L ← S							
cvt.l.s	Zero			Zero	Zero		
cvt.l.s	DeNorm			Unimple-mented	Unimple-mented	Zero	
cvt.l.s	SNaN			Invalid	Gen QNaN (long)		
cvt.l.s	QNaN			Invalid	Gen QNaN (long)		
cvt.l.s	Inf			Invalid	Gen QNaN (long)		
cvt.l.s	Norm			Unimple-mented	Unimple-mented		$2^{52} \leq in \leq 2^{63}$ $-2^{52} \geq in \geq -2^{63}$
cvt.l.s	Norm			Invalid	Gen QNaN (long)		On overflow (in > 2^{63}) or (in < -2^{63})

Operation	OpA	OpB	OpC	Exception Enabled Result	Exception Disabled Result	FS Mode Result	Remarks
Convert - W ← D							
cvt.w.d	Zero			Zero	Zero		
cvt.w.d	DeNorm			Unimple-mented	Unimple-mented	Zero	
cvt.w.d	SNaN			Invalid	Gen QNaN (word)		
cvt.w.d	QNaN			Invalid	Gen QNaN (word)		
cvt.w.d	Inf			Invalid	Gen QNaN (word)		
cvt.w.d	Norm			Invalid	Gen QNaN (word)		On overflow (in > 2^{31}-1) or (in < -2^{31})
Convert - L ← D							
cvt.l.d	Zero			Zero	Zero		
cvt.l.d	DeNorm			Unimple-mented	Unimple-mented	Zero	
cvt.l.d	SNaN			Invalid	Gen QNaN (long)		
cvt.l.d	QNaN			Invalid	Gen QNaN (long)		
cvt.l.d	Inf			Invalid	Gen QNaN (long)		
cvt.l.d	Norm			Unimple-mented	Unimple-mented		$2^{52} \leq in \leq 2^{63}$ $-2^{52} \geq in \geq -2^{63}$
cvt.l.d	Norm			Invalid	Gen QNaN (long)		On overflow (in > 2^{63}) or (in < -2^{63})
Absolute							
abs	Zero			Zero	Zero		Sign = 0
abs	DeNorm			DeNorm	DeNorm	Zero	Sign = 0
abs	Inf			Inf	Inf		Sign = 0
abs	SNaN			Invalid	Gen QNaN		
abs	QNaN			Pass QNaN	Pass QNaN		
Negate							
neg	Zero			Zero	Zero		Appropriately signed
neg	DeNorm			DeNorm	DeNorm	Zero	Appropriately signed
neg	Inf			Inf	Inf		Appropriately signed
neg	SNaN			Invalid	Gen QNaN		
neg	QNaN			Pass QNaN	Pass QNaN		

6.12 FPU Instruction Pipeline Overview

The FPU provides two execution units that parallel the integer ALU.

The FP Load/Store unit executes the memory transactions, and **CFC1**/**CTC1**/**MFC1**/**MTC1**/***conditional move*** instructions for the FPU. The FP Load/Store unit uses a 5-stage pipeline similar to the integer ALU. In this pipeline, the FP register updates occur in the W stage.

The second execution unit is called the FP ALU. The FP ALU is comprised of the FP Multiply/Add unit and the Divide/Sqrt unit.The Divide/Sqrt unit executes divide, square root, reciprocal, and reciprocal square root instructions. The FP Multiply/Add unit is used for all other FP ALU operations. The FP ALU uses a seven stage pipeline. The first five stages uses the same names as the integer pipeline. The X (eXtended) and E (End) stages are appended after the W stage. In this longer pipeline, the FP register updates occur in the E stage.

The Multiply/Add unit has an internal 4-cycle pipeline. An instruction's operands and operation enter this internal pipeline at the A stage of the FP ALU. Once an instruction's operation enters this internal pipeline, it always runs to completion without waiting for the stalls or slips that might be affecting the rest of the FP ALU. The completed result always leaves this internal pipeline in either 4, 5 or 6 processor cycles. An instruction that uses the Multiply/Add unit can occupy the A stage of the FPU ALU for 1, 2, or 3 running (non-stalled) processor cycles. When a result is completed at the end of this internal pipeline, it is passed to the seven stage pipeline in one of the A, D, W, X or E stages

The FP Divide/Sqrt unit is not pipelined. Divide/Sqrt operands enter this unit in the A stage of the FP ALU. An instruction that uses this unit will only occupy the A stage for one running (non-stalled) cycle. Results from this unit can enter the seven stage pipeline out of code order.

The FPU control logic schedules when the completed results from the different units can enter the seven stage pipeline to update the registers. Once an instruction has entered the W stage, it is guaranteed to not cause an exception and will always update it's destination register. To reduce the effects of this longer pipeline, bypass logic is used to forward an instruction's results as soon as possible to subsequent instructions.

6.12.1 Instruction Execution

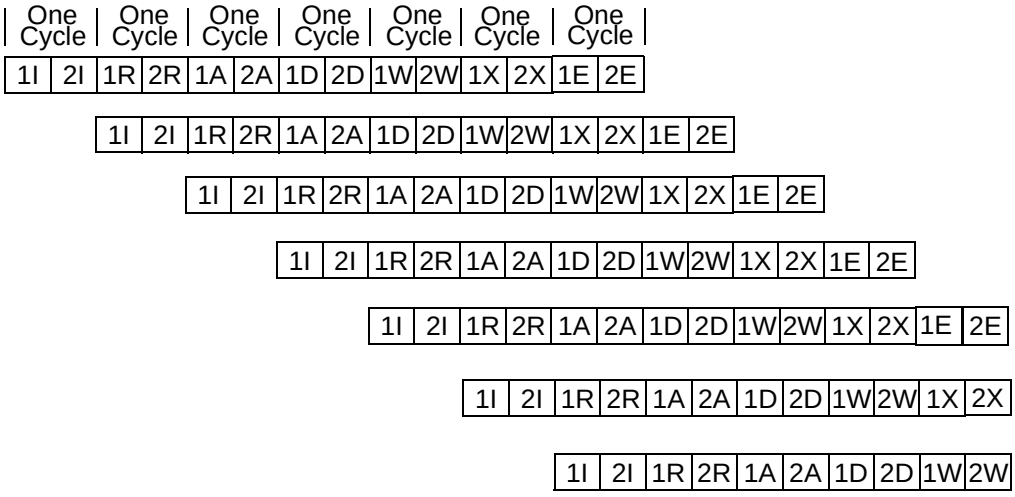


Figure 6.9 FPU Instruction Pipeline

Figure 6.9 illustrates the 7-instruction overlap in the FPU Multiply/Add unit. The figure assumes that one instruction is completed every PCycle.

6.12.2 Instruction Execution Cycle Time

Unlike the CPU, which executes almost all instructions in a single cycle, more time may be required to execute FPU instructions.

Appendix C, "Cycle Time and Latency Tables", gives the minimum latency, in processor pipeline cycles, of each floating-point operation for the currently implemented configurations. These latency calculations assume the result of the operation is immediately used in a succeeding operation.

6.12.3 Instruction Scheduling Constraints

Note: The FPU control logic is kept from issuing instructions to the FPU ALU units (Multiply/Add and Divide/Sqrt) by the limitations in their micro-architectures - a pipeline stage is already busy, the desired unit is already busy, operand registers are not ready, etc. An FPU ALU instruction can be issued at the same time as any other non-FP ALU instructions. This includes all integer instructions as well as floating-point loads and stores.

Section 7 Cache Organization and Operation

The RM7000 contains three separate caches:

- Primary Instruction Cache This 16 Kbyte, 4-way set associative cache contains only instruction information.
- Primary Data Cache: This 16 Kbyte, 4-way set associative cache contains only data information.
- Secondary Cache: This 256 Kbyte, 4-way set associative cache contains both instruction and data information.

7.1 Caches Overview

The primary caches each require one cycle to access. Each primary cache has its own 64-bit read data path and 128-bit write data path, allowing both caches to be accessed simultaneously. The primary caches provide the integer and floating-point units with an aggregate bandwidth of over 5 GBytes per second.

The secondary cache also has a 64-bit data path and is accessed only on a primary cache miss. The secondary cache cannot be accessed in parallel with either of the primary caches and has a three-cycle miss penalty on a primary cache miss.

During a primary instruction or data cache refill, the secondary cache provides 64 bits of data every cycle following the initial 3-cycle latency. This results in a aggregate bandwidth of 2.5 GBytes per second.

In addition to the three on-chip caches, the RM7000 provides a dedicated tertiary cache interface and supports tertiary cache sizes of 512 Kbytes, 2 Mbytes, and 8 Mbytes. The tertiary cache is only accessed after a secondary cache miss and hence cannot be accessed in parallel with the secondary cache. Both the secondary and tertiary caches can be disabled by setting the appropriate bits in the CP0 *Config* register. The secondary and tertiary caches are only capable of block writes and are never modified on a partial write. All of the RM7000 caches are virtually indexed and physically tagged, eliminating the potential for virtual aliasing.

Having multiple cache hierarchies on-chip means that special consideration must be given during a primary cache flush operation. In previous designs with no on-chip secondary cache, flushing of the primary caches caused the data to be moved to off-chip secondary cache or main memory. Using the same code sequence in the RM7000 still moves data to the secondary cache. But since the secondary cache is on-chip, it must also be flushed in order to move the data off-chip. To manage this multiple cache hierarchy, the RM7000 uses a combination of secondary cache operations (cacheops) and the **SYNC** instruction to assure that store data intended for main memory actually appears at the signal pins instead of being written to the secondary cache. Execution of the **SYNC** instruction guarantees that all previous load or store instructions are completed before any of the instructions following the **SYNC** instruction can be executed.

7.1.1 Non-Blocking Caches

The RM7000 implements a non-blocking architecture for each of the three on-chip caches. Non-blocking caches improve overall performance by allowing the cache to continue operating even though a cache miss has occurred.

In a typical blocking-cache implementation, the processor executes out of the cache until a miss occurs, at which time the processor stalls until the miss is resolved. The processor initiates a memory cycle, fetches the requested data, places it in the cache, and resumes execution. This operation can take many cycles depending on the design of the memory system.

In a non-blocking implementation, the caches do not stall on a miss. The processor continues to operate out of the primary caches until one of the following events occurs:

1. Two cache misses are outstanding and a third load/store instruction appears on the instruction bus.
2. A subsequent instruction requires data from either of the instructions that caused the cache misses.

The RM7000 supports two outstanding cache misses for both the primary and secondary caches. When a primary cache miss occurs, the processor checks the secondary cache to determine if the requested data is present. If the data is not present a tertiary cache/main memory access is initiated. In this case, even though there was a primary and subsequent secondary cache miss, they are seen by the processor as one miss since both accesses were for the same address location.

During this time the processor continues executing out of the primary cache. If a second primary cache miss occurs a second secondary cache access is generated. Even though two cache misses are outstanding, the processor continues to execute out of the primary cache. If a third primary cache miss occurs prior to the time either of the two aforementioned misses have been resolved, the processor stalls until either one is completed.

The non-blocking caches in the RM7000 allow for more efficient use of techniques such as loop unrolling and software pipelining. To take maximum advantage of the caches, code should be scheduled to move loads as early as possible, away from instructions that may actually use the data.

To facilitate systems that have I/O devices which depend on in-order loads and stores, the default setting for the RM7000 is to force uncached references to be blocking. These uncached references can be changed to non-blocking by using the new uncached, non-blocking cache coherency attribute (code 6). Refer to section 7.6, “Cache Coherency” on page 88, for more information.

7.1.2 Cache Locking

The RM7000 processor supports cache locking of the primary and secondary caches on a per-line basis. Cache locking allows critical code or data segments to be locked into the caches. In the primary data and secondary caches, the locked contents can be updated on a store hit, but cannot be selected for replacement on a miss. Each of the three caches can be locked separately. However, only two of the four sets of each cache can be locked.

The RM7000 allows a maximum of 128 Kbytes of data or code to be locked in the secondary cache, a maximum of 8 Kbytes of code to be locked in the instruction cache, and a maximum of 8 Kbytes of data to be locked in the data cache.

Primary cache locking is accomplished by setting the appropriate cache lock enable bits and specifying which set to lock in the *ECC* register, then bringing the desired data/code into the caches by using either a Load instruction for data, or a **FILL_ICACHE** CACHE operation for instructions while the cache lock enable bit is set. Locking in the secondary cache is accomplished by setting a separate secondary cache lock enable bit in the *ECC* register, then executing either a load instruction for data, or a **FILL_ICACHE** instruction for instructions while the secondary cache lock enable bit is set. It is recommended that the instruction stream that loads the caches for locking be executed from an uncached address segment in order to not pollute the icache. Table 7.1: shows how the *ECC* register bits control cache locking and set selection.

Table 7.1: Cache Locking Control

Cache	Lock Enable	Set Select	How to activate
Primary Instruction	ECC[27]	ECC[28]=0 → A ECC[28]=1 → B	CACHE Fill_I
Primary Data	ECC[26]	ECC[28]=0 → A ECC[28]=1 → B	Load/Store
Secondary	ECC[25]	ECC[28]=0 → A ECC[28]=1 → B	CACHE Fill_I or Load/Store

Only sets A and B of a cache can be locked. ***ECC[28]*** determines the set to be locked as shown in Table 7.1:, Set A can be locked by clearing the ***ECC[28]*** bit and performing a load operation. Set B can then be locked by setting the ***ECC[28]*** bit and performing another load operation. This procedure allows both sets to be locked together. With the desired data/code in the caches, setting the lock enable bit inhibits cache updates. The lock enable bits should be cleared to allow future memory transactions to fill the caches normally.

A locked cache line can be unlocked by either clearing the lock bit in the tag RAM using the **INDEX_STORE_TAG CACHE** instruction, or by invalidating the cache line using one of the invalidate **CACHE** instructions. Invalidation of a cache line causes that line to be unlocked, even if the corresponding lock bit has not been cleared. Once the processor invalidates the line

it becomes a candidate for a fill operation. When the fill cycle occurs the lock bit is cleared. Refer to section 7.8, “Code Examples” on page 93, for an example code sequence of a cache locking operation.

A new bypass coherency attribute (code 7) can be used to bypass the secondary and tertiary caches. However, this attribute can also be used to lock the contents of the secondary cache. The secondary cache is first preloaded with data using one of the other coherency attributes. The bypass or uncached coherency attribute is then used for all subsequent instruction and data accesses to implicitly lock the secondary cache. Using this method causes the secondary cache to behave as a read-only memory and ensures that data is never overwritten by a cache line fill or writeback.

7.1.3 Cache Line Ownership

The processor is the owner of and is responsible for the contents of a cache line when that line is in the dirty exclusive state. There is only one owner for each cache line.

The ownership of a cache line is set and maintained through the following protocol:

- The processor assumes ownership of a primary cache line if the state of that line is in the *DataValid* or *InstrValid* state.
- The processor assumes ownership of a secondary cache line if that line is in the *dirty-exclusive* or *dirty-exclusive-Dcache* state.
- If the processor owns a cache line that is in a writeback page and that line is replaced during the execution of a *Writeback* or W*riteback Invalidate* operation, it writes the line back to memory.
- Main 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.

7.1.4 Write Buffers

The RM7000 incorporates five secondary cache write buffers and one primary cache write buffer. The primary cache write buffer is located between the data cache and the secondary cache and provides temporary storage for either one data cache writeback, or four separate uncached stores. Four secondary cache write buffers are used for temporary storage of cacheline writebacks out of the secondary cache to the system interface. The fifth secondary cache write buffer provides temporary storage for a maximum of four uncached stores on their way to the system interface.

For uncached and write-through store operations, the write buffers significantly increase performance by decoupling the **SysAD** bus transfers from the instruction execution stream.

7.1.5 Orphaned Cache Lines

The integrated secondary cache of the RM7000 allows for flexibility in cache organization and management policies that are not practical with an external secondary cache. To obtain maximum cache performance, the RM7000 does not require that the primary caches are always a subset of the secondary cache. This can result in certain primary cache lines becoming ‘orphans’. The following is an example of how a primary cache line can become an ‘orphan’. This example assumes that the data cache is using a writeback protocol.

1. Data A resides in both the data cache and the secondary cache.
2. Data A in the data cache is modified by a store operation.
3. Data B is fetched from memory by a load operation and overwrites data A in the secondary cache but goes to a different location in the data cache. The data cache now contains data A and data B. But the secondary cache contains only data B. At this time the data cache is no longer a proper subset of the secondary cache. Data A is now an ‘orphan’. Orphaned lines in the primary data cache are written back directly to tertiary cache/main memory instead of the secondary cache.

Parent lines in the secondary cache are not written back if both the parent line in the secondary and the child line in the data cache are both dirty. In this case the data cache has the newest dirty data, thus the secondary cache line does not have to be written back.

7.1.6 Replacement Algorithm

Each of the three on-chip caches uses the same cyclic replacement algorithm. The algorithm attempts to perform a round-robin replacement for sets 0, 1, 2, and 3. Each of the four cache lines, one per set at a particular cache index has a tag at the corresponding index in the tag RAM. Each tag RAM contains a corresponding fill (***F***) bit. The algorithm uses the state of the ***F*** bits to determine which set to replace.

7.1.7 Cache Attributes

Table 7.2: shows the attributes for the three on-chip caches as well as the attribute requirements for the tertiary cache.

Table 7.2: RM7000 Cache Attributes

Attribute	Instruction	Data	Secondary	Tertiary
Size	16KB	16KB	256KB	512KB to 8MB, in powers of two
Associativity	4-way	4-way	4-way	Direct Mapped
Replacement Algorithm	Cyclic	Cyclic	Cyclic	Direct
Line size	32 byte	32 byte	32 byte	32 byte
Index	$vAddr_{11..0}$	$vAddr_{11..0}$	$pAddr_{15..0}$	$pAddr_{22..0}$
Tag	$pAddr_{35..12}$	$pAddr_{35..12}$	$pAddr_{35..16}$	$pAddr_{35..19}$
Write policy	n.a.	write-back, write through	Block writeback, Bypass	Block writethrough, Bypass
Read policy	n.a.	Non-blocking (2 outstanding)	Non-blocking for data, blocking for instruction (2 outstanding)	Non-blocking for data, blocking for instruction (2 outstanding)
Read Order	Critical word first	Critical word first	Critical word first	Critical word first
Write Order	Sequential	Sequential	Sequential	Sequential
Miss Restart Following:	Complete Line	First double (if waiting for data)	n.a.	n.a.
Parity	Word	Byte	DoubleWord	Doubleword
Cache locking	Per line	Per line	Per line	none

7.2 Primary Instruction Cache

The primary instruction cache is 16 Kbytes in size and implements a 4-way set associative architecture. Line size is 32-bytes, or eight instructions. The 64-bit read path allows the RM7000 to fetch two instructions per clock cycle which are passed to the superscalar dispatch unit.

7.2.1 Instruction Cache Organization

The instruction cache is organized as shown in Figure 7.1. The cache is 4-way set associative and contains 128 indexed locations. Each time the cache is indexed, the tag and data portion of each set are accessed. Each of the four tag addresses are compared against the translated portion of the virtual address to determine which set contains the correct data.

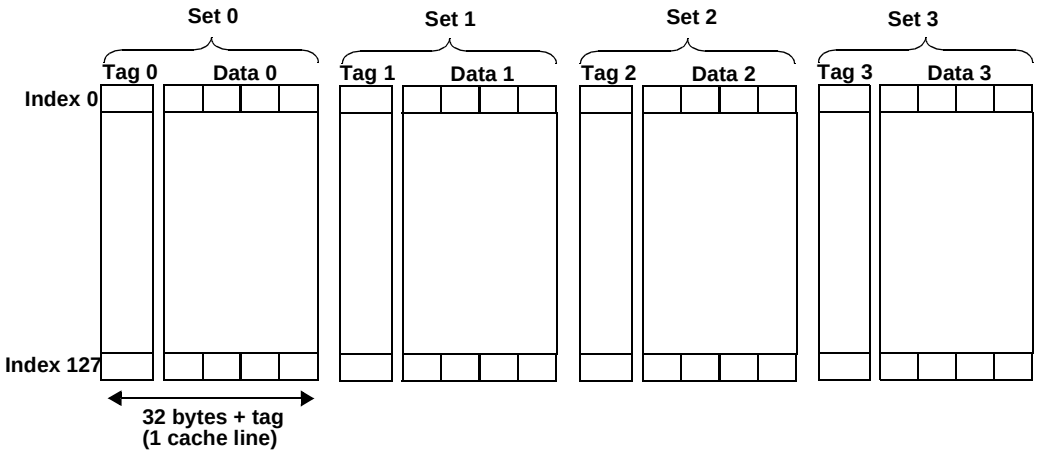
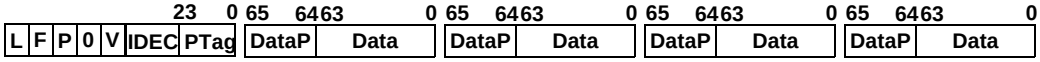


Figure 7.1 Instruction Cache Organization

When the instruction cache is indexed, each of the four sets shown in Figure 7.1 returns a single cache line. Each cache line consists of 32 bytes of data protected by word parity, a 24-bit physical tag address, and three tag control bits. Figure 7.2 shows the instruction cache line format.



IDEC	Instruction Predecode bits
L	Lock bit. Used for locking the instruction cache. 1=lock, 0=unlock
F	FIFO replacement bit. Complemented on refill
P	Even parity for the PTag and V fields
V	Tag valid bit
PTag	24-bit physical address tag (bits 35:12 of the physical address)
DataP	Even parity; 1 parity bit per word of data
Data	Cache Data

Figure 7.2 Instruction Cache Line Format

7.2.2 Accessing the Instruction Cache

The RM7000 implements a 4-way set associative cache that is virtually indexed and physically tagged. Even though the instruction cache is physically indexed, the access is done in parallel with the virtual-to-physical address translation because only the upper bits of the address are translated. The lower bits are used directly for indexing the cache and do not go through translation. Figure 7.3 shows how the virtual address is divided on an instruction cache access.

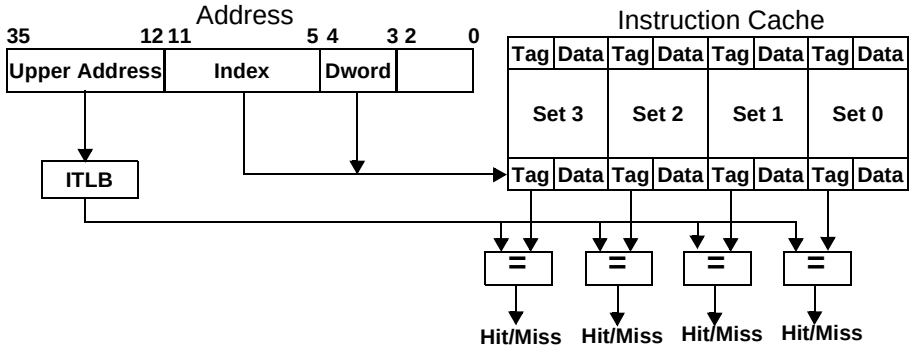


Figure 7.3 Accessing the Instruction Cache

The lower 12 bits of address are used for indexing the instruction cache as shown in Figure 7.3. Bits 11:5 are used for indexing one of the 128 locations. Within each set there are four 64-bit doublewords of data. Bits 4:3 are used to index one of these four doublewords. The tag for each cache line is accessed using address bits 11:5.

When the cache is indexed, the four blocks of data and corresponding physical address tags are fetched from the cache at the same time the upper address is being translated. The translated address from the instruction translation lookaside buffer (ITLB) is compared with each of the four address tags. If any of the four tags yield a valid compare, the data from that set is used. This is called a '*primary cache hit*'. If there is no match between the translated address and any of the four address tags, the cycle is aborted and a secondary cache access is initiated. This is called a *'primary cache miss'*.

Locking a cache block prevents its contents from being overwritten by a subsequent cache miss. This mechanism allows the programmer to lock critical code into the cache and thereby guarantee deterministic behavior for a locked code sequence. Only valid cache lines can be locked. If a cache line within set 0 or 1 is invalid while either set is locked, that cache line can be changed by subsequent instruction fetches.Refer to section 7.8, "Code Examples" on page 93, for an example code sequence of a cache locking operation.

7.3 Primary Data Cache

The primary data cache is 16 Kbytes in size and implements a 4-way set associative architecture. Line size is 32-bytes, or eight words. The data cache contains a 64-bit read path and a 128-bit write path. The data cache can be used in either write-through or writeback mode. These modes are selected on a per-page basis.

The data cache is virtually indexed and physically tagged, eliminating the potential for virtual aliasing. The data cache is non-blocking, meaning that a miss in the data cache does not stall the pipeline.

The normal write policy is writeback, where a store operation to the data cache does not cause the secondary cache or main memory to be updated. The writeback protocol increases overall system performance by reducing bus traffic. Data is written to the slower memories only when a data cache line is replaced.

The data cache also supports the write-through protocol on a per-page basis. In write-through mode all writes to the primary data cache are simultaneously written to main memory. The secondary and tertiary caches are never accessed in write-through mode. Software controls which cache policy to use through programming the TLB coherency bits.

7.3.1 Data Cache Organization

The data cache is organized as shown in Figure 7.4. The cache is 4-way set associative and contains 128 indexed locations. Each time the cache is indexed, the tag and data portion of each set are accessed. Each of the four tag addresses are compared against the translated portion of the virtual address to determine which set contains the correct data.

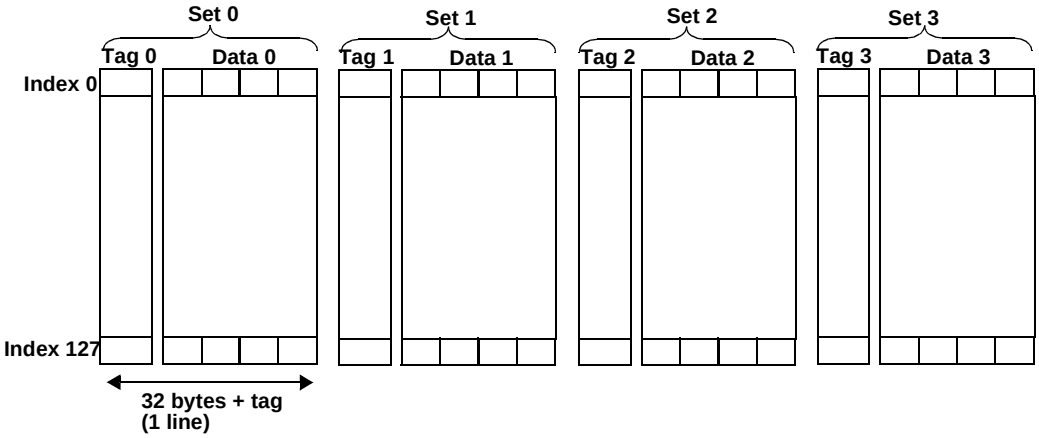
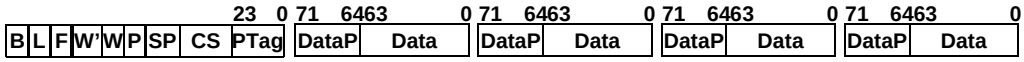


Figure 7.4 Data Cache Organization

When the data cache is indexed, each of the four sets shown in Figure 7.4 return a single cache line. Each cache line consists of 32 bytes of data protected by byte parity, a 24-bit physical tag address, and six tag control bits. Figure 7.5 shows the data cache line format.



B	Bypass bit. Used for bypassing the secondary and tertiary caches on a dirty writeback, 1=bypass, 0=do not bypass
L	Lock bit. Used for locking the data cache. 1=lock, 0=unlock
F	FIFO replacement bit. Complemented on refill
W'	Even parity for the Writeback bit
W	Writeback bit (set if cache line has been written)
P	Even parity for the PTag and CS fields
SP	Even parity for the CS field
CS	Primary cache state 00 = Invalid 01 = Pending 11 = Data Valid
PTag	24-bit physical address tag (bits 35:12 of the physical address)
DataP	Even parity; 1 parity bit per byte of data
Data	Cache Data

Figure 7.5 Data Cache Line Format

7.3.2 Accessing the Data Cache

The RM7000 implements a 4-way set associative data cache that is virtually indexed and physically tagged. Even though the data cache is physically indexed, the access is done in parallel with the virtual-to-physical address translation because only the upper bits of the address are translated. The lower bits are used directly for indexing the cache and do not go through translation. Figure 7.6 shows how the virtual address is divided on a data cache access.

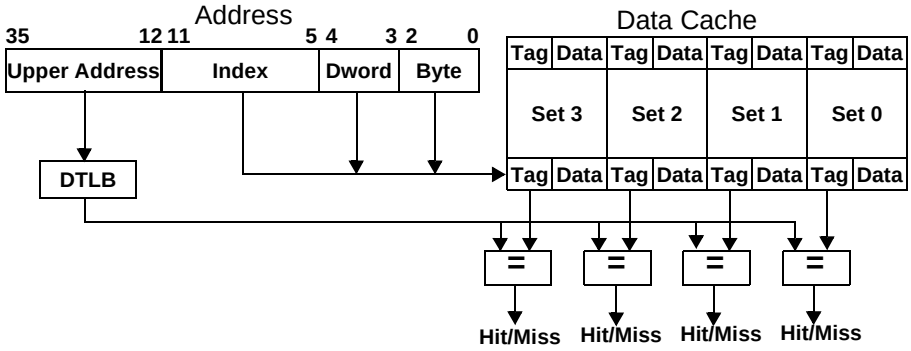


Figure 7.6 Accessing the Data Cache

The lower 12 bits of address are used for indexing the data cache as shown in Figure 7.6. Bits 11:5 are used for indexing one of the 128 locations. Within each set there are four 64-bit doublewords of data. Bits 4:3 are used to index one of these four doublewords. Bits 2:0 are used to index one of the eight bytes within each doubleword. The tag for each cache line is accessed using address bits 11:5.

Locking a cache block prevents its contents from being overwritten by a subsequent cache miss. This mechanism allows the programmer to lock critical data in the cache. Only valid cache lines can be locked. If a cache line within set 0 or 1 is invalid while either set is locked, that cache line can be changed by subsequent data fetches.Refer to section 7.8, “Code Examples” on page 93, for an example code sequence of a cache locking operation.

7.3.3 Data Cache Policies

The primary data cache supports the following policies:

- Uncached, blocking
- Writeback, non-blocking
- Write-through with Write Allocate, non-blocking
- Write-through without Write Allocate, non-blocking
- Writeback with Secondary and Tertiary Bypass
- Uncached, non-blocking

Refer to section 7.6, “Cache Coherency” on page 88, for more information on each of these cache policies.

Figure 7.7 shows the hierarchy of cache accesses during a store operation. In this diagram, WB refers to a writeback protocol, and WT refers to a write-through protocol.

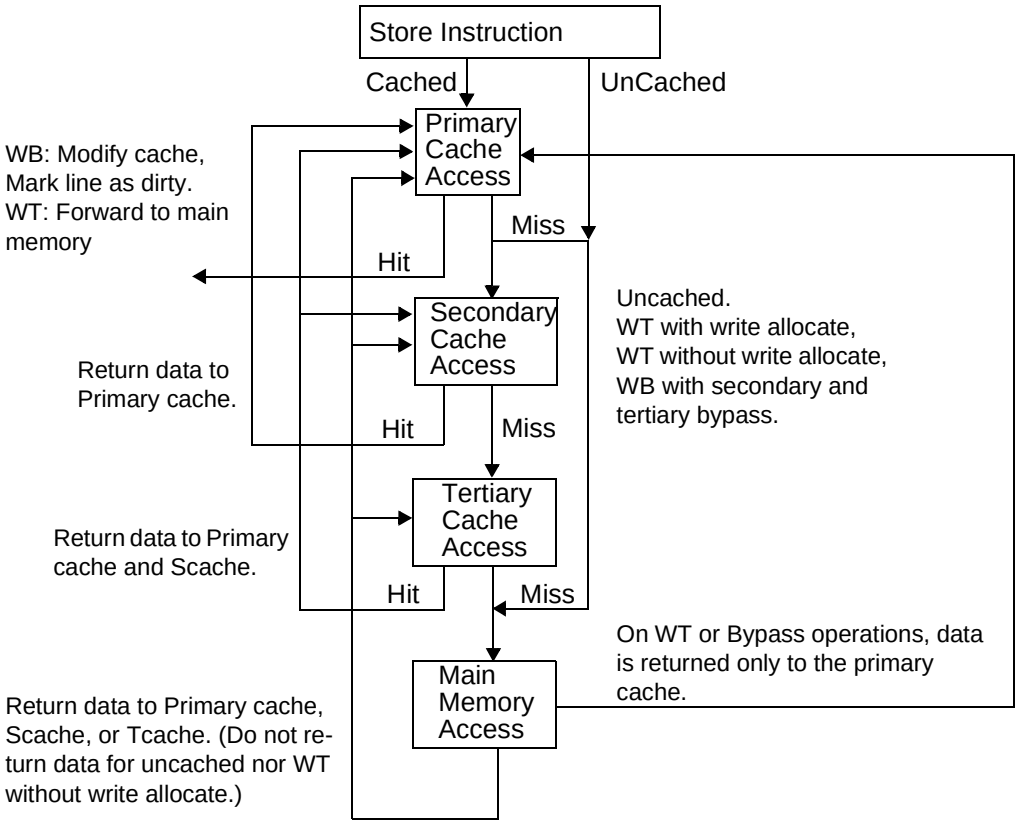


Figure 7.7 RM7000 Cache Refill Hierarchy

7.3.4 Data Cache Store Buffer

The data cache contains a 64-bit store buffer that holds a single data entry. When the RM7000 executes a store instruction, this buffer is written with the store data while the tag comparison is performed. If the tag matches, data is written into the data cache in the next cycle that the cache is not being accessed (the next non-load cycle). The store buffer allows the RM7000 to execute a store every processor cycle and to perform back-to-back stores without incurring a penalty. In the event of a store immediately followed by a load to the same address, a combined merge and cache write followed by cache read occurs such that no penalty is incurred.

In the RM7000 the ***W*** (***write-back***) bit, not the cache state, indicates whether or not the primary cache contains modified data that must be written back to memory. Note that there is no hardware support for cache coherency. Thus the only cache states used are *DataValid* and *Invalid*.

7.3.5 Primary Cache States

The RM7000 supports the following primary data cache states. These states are visible in bits 7:6 of the *TagLo* Register after an **INDEX_LOAD_TAG CACHE** instruction has been executed. The cache state encoding is as follows:

- **Invalid (00):** A cache line that does not contain valid information is marked invalid and cannot be read. A cache line in any other state than invalid is assumed to contain valid information.
- **Pending (01):** In the pending state, the cache line is valid and has been allocated due to a previous cache miss, but the data has not yet arrived. The pending state is a transient one. When the data arrives, the state machine transitions to a valid state. Software should never write this state to a primary cache tag.

- **InstrValid (10):** An InstrValid cache line contains valid information that has been filled from either a clean secondary cache line, tertiary cache line, or main memory. The instruction cache uses InstrValid as its valid state. This state is not used by the data cache.
- **DataValid (11):** A DataValid cache line contains valid information that has been filled from either a a secondary cache line, tertiary cache line or main memory. The cache line may or may not be consistent with memory and is owned by the processor. The ***W*** bit is set to indicate that the data is inconsistent with main memory. This state is not used by the instruction cache.

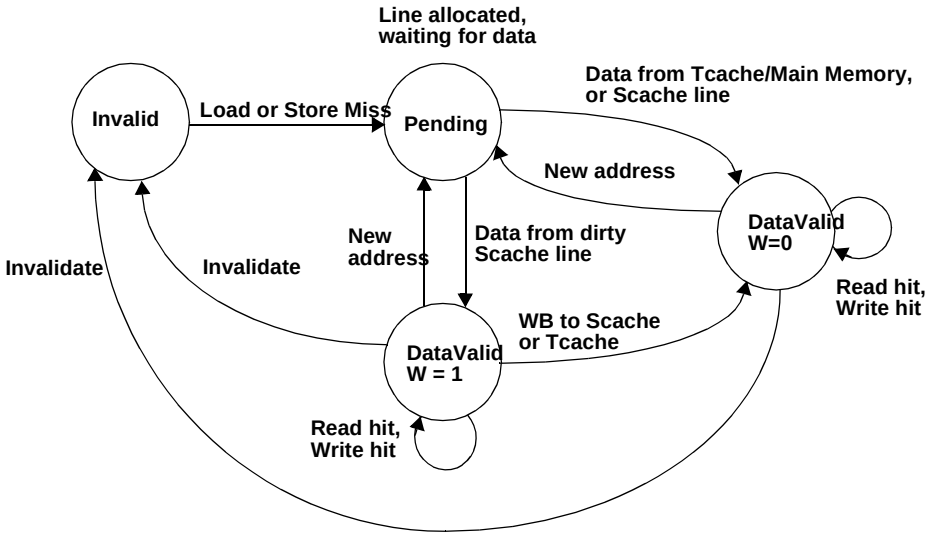


Figure 7.8 Primary Data Cache State Transitions

Figure 7.8 shows a cache state transition diagram for the primary data cache. Figure 7.9 shows a cache state transition diagram for the primary instruction cache. These diagrams use the following abbreviations:

- Dcache = primary data cache
- Scache = secondary cache
- Tcache = tertiary cache
- WB = writeback

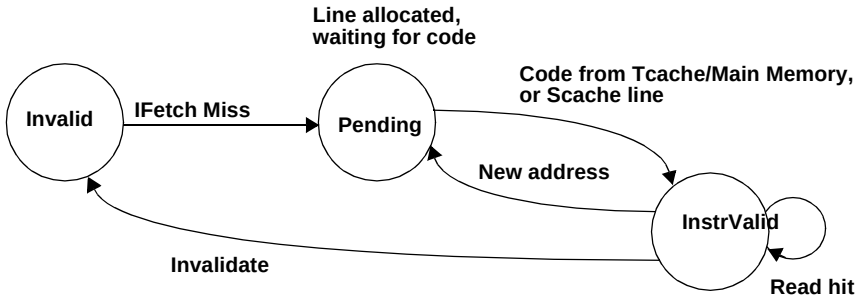


Figure 7.9 Primary Instruction Cache State Transitions

7.4 Secondary Cache

The RM7000 implements a 256 Kbyte, 4-way set associative on-chip secondary cache. Line size is 32 bytes. The secondary cache is implemented with a 64-bit wide read/write data path which matches that of the primary caches and the system interface. Integration of the secondary cache greatly reduces the overall cache miss latency and reduces the overall power consump-

tion since secondary cache data no longer must pass through I/O buffers to be accessed by the processor. Integration of the secondary cache also increases the flexibility in cache organization and management policies.

The RM7000 does not require that the primary caches be always a subset of the secondary cache. A secondary cache line can be updated from the tertiary cache or main memory without also updating the same line in the primary cache. This technique allows for maximum cache performance but can result in certain primary cache lines becoming *orphans*. Refer to section 7.1.5, “Orphaned Cache Lines” on page 77, for more information.

The secondary cache supports only block read and block write transactions. Write-through stores are never stored in the secondary cache. The secondary cache can also be bypassed using a new bypass cache coherency attribute. This attribute bypasses both the secondary and tertiary caches. When this attribute is selected, the secondary and tertiary caches are not filled on a load miss, and are not written on dirty writebacks from the primary caches.

Secondary cache data accesses are non-blocking, meaning that a cache miss on a data access does not cause the processor to stall until the miss is resolved. The secondary cache supports two outstanding data misses. However, instruction accesses are blocking, meaning that a cache miss on an instruction access causes the processor to stall until the miss is resolved.

The secondary cache supports cache locking on a per line basis, allowing the user to assure that critical data or code segments are not overwritten on a cache miss. Only two sets can be locked at any given time. Refer to section 7.8, “Code Examples” on page 93, for an example code sequence of a cache locking operation.

The secondary cache can be disabled by software by clearing ***Config.SE*** bit. Software can read ***Config.SC*** bit to check for the existence of the secondary cache before initializing/flushing the cache. This allows software to be ready for future RM7000 derivatives that might not implement that cache.

7.4.1 Secondary Cache Organization

The secondary cache is similar to the instruction and data caches, except that there are 2048 indexed locations. Each set contains 64 Kbytes (32 bytes/line x 2048 lines). The four sets provide for a total of 256 Kbytes. Each time the cache is indexed, the tag and data portion of each set are accessed. Each of the four tag addresses is compared against the physical address to determine which set contains the correct data.

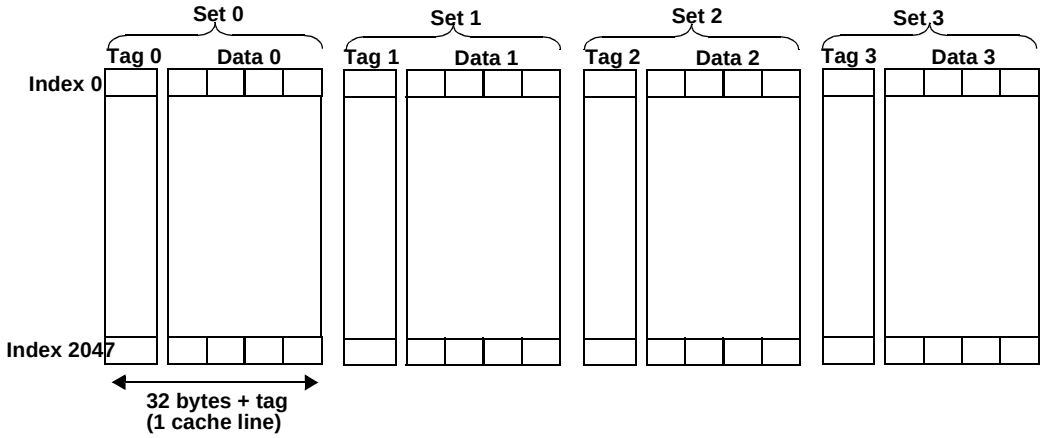
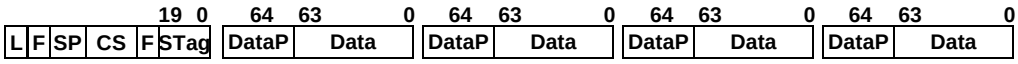


Figure 7.10 Secondary Cache Organization

Figure 7.10 shows the organization of the secondary cache. When the secondary cache is indexed, each of the four sets contains a single cache line. Each cache line consists of 32 bytes of data supported by doubleword parity, a 24-bit physical tag address, and three tag control bits. Figure 7.11 shows the secondary cache line format.



L	Lock bit. Used for locking the instruction cache. 1=lock, 0=unlock
F	FIFO replacement bit. Complemented on refill
SP	State parity. Provides even parity for the CS field
CS	Secondary cache state 000 = Invalid 110 = Pending 100 = Clean Exclusive 101 = Dirty Exclusive 111 = Dirty Exclusive Dcache
P	Tag parity. Provides even parity for the L, F, and STag fields
STag	20-bit secondary cache tag (bits 35:16 of the physical address)
DataP	Even parity; 1 parity bit per doubleword of data
Data	Cache Data

Figure 7.11 Secondary Cache Line Format

7.4.2 Accessing the Secondary Cache

The 256 Kbyte secondary cache is only accessed on a primary cache miss. Once the processor has determined that the requested address does not match the corresponding primary cache tag, a secondary cache access is initiated. Like the primary caches, the secondary cache is also physically indexed and physically tagged.

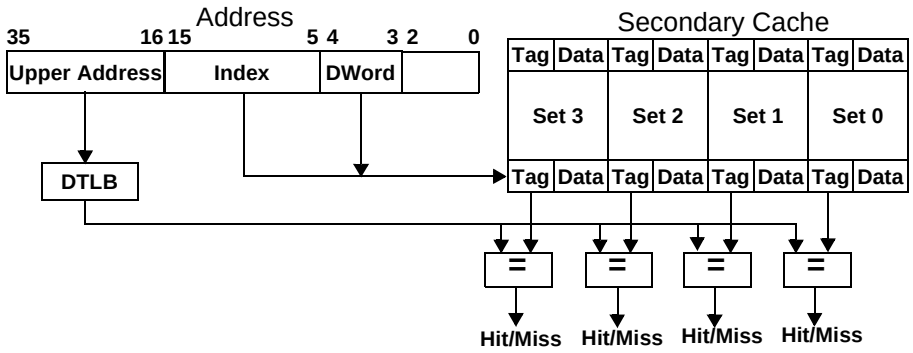


Figure 7.12 Accessing the Secondary Cache

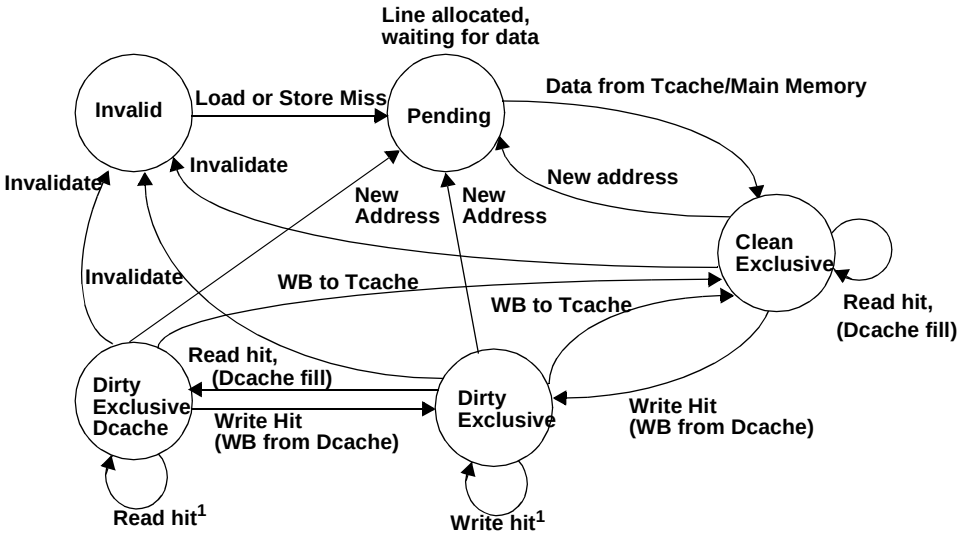
The lower 16 bits of address are used for indexing the data cache as shown in Figure 7.12. Bits 15:5 are used for indexing one of the 2048 sets. Within each indexed entry of each set there are four 64-bit doublewords of data. The tag for each cache line is accessed using address bits 15:5.

7.4.3 Secondary Cache States

The RM7000 supports the following secondary cache states. These states are visible in bits 8:6 of the *TagLo* Register after an **INDEX_LOAD_TAG_SC CACHE** instruction is executed.The state encoding are as follows:

- **Invalid (000):** A cache line that does not contain valid information is marked invalid and cannot be used. A cache line in any other state than invalid is assumed to contain valid information.
- **Pending (110):** In the pending state, the cache line is valid and has been allocated due to a previous cache miss, but the data has not yet arrived. The pending state is a transient one. When the data arrives, the cache line transitions to a valid state. Software should never write this state to a secondary cache tag.
- **Clean Exclusive (100):** A clean exclusive cache line contains valid information. The cache line is valid and is consistent with main memory.
- **Dirty Exclusive (101):** A dirty exclusive cache line contains valid information that has been modified by the processor and is not consistent with main memory. In this state the secondary cache contains the most current data.

- **Dirty Exclusive Dcache (111):** In this state the primary data cache contains the most current data. The secondary cache contains an older version of the data. A cache line in this state contains valid information that has been modified by the processor and is not consistent with main memory.



1. This condition should not occur as the cache access should hit in the primary data cache.

Figure 7.13 Secondary Cache State Transitions

Figure 7.13 shows a cache state transition diagram for the secondary cache. This diagram uses the same abbreviations listed for Figure 7.8 on page 83 in section 7.3.5, “Primary Cache States”.

7.5 Tertiary Cache

The RM7000 provides support for an external tertiary cache. The tertiary cache interface is identical to the MIPS R5000 secondary cache interface. The RM7000 supports tertiary cache sizes of 512 Kbytes to 8 Mbytes, in powers of two. The tertiary cache is direct-mapped, implements a block write-through protocol, is non-blocking, and supports two outstanding cache misses for data accesses.

The RM7000 uses the **SysAD** bus for address and data communication with the tertiary cache and provides a dedicated control interface which includes the following groups of signals:

- 16-bit tertiary cache line index
- 2-bit tertiary cache word index
- Eight enable signals for the tertiary cache tag and data RAMs
- Tertiary cache valid signal
- Tertiary cache tag match signal
- Tertiary cache block clear signal

For a complete listing of tertiary pins and pin descriptions, refer to Section 12, “System Interface Protocol”, of this manual. For specific timing information refer to the QED RM7000 Datasheet.

In addition to block read and block write operations, the tertiary cache also performs tag invalidate, tag probe, and block clear operations. The tag invalidate operation consists of writing to the tag RAM and invalidating the line in question by clearing the appropriate valid bit. Invalidates are the only tertiary cache operations that can occur back-to-back.

The tertiary cache tag RAM probe is similar to a tag RAM read operation. The RM7000 asserts the tertiary cache data and chip enables in the same clock that the address and line index busses are driven. The RM7000 then tri-states the **SysAD** bus. One clock later the RM7000 drives the tag RAM output enable and the tag RAM drives the tag information onto the **SysAD** bus.

The tertiary cache block clear operation allows for the invalidation of an entire cache line in one operation. This operation is accomplished by clearing an entire column of tag RAM valid bits. However, in order to support this operation the tag RAM must support a flash clear of the valid bit column.

The tertiary cache can be disabled by software by clearing the ***Config.TE*** bit. Software can read the ***Config.TC*** bit to check for the existence of the tertiary cache before initializing/flushing the cache.

7.5.1 Tertiary Cache Organization

The tertiary cache is a direct-mapped cache that contains instruction and data information. The RM7000 supports tertiary cache sizes of 512 Kbytes, 1Mbytes, 2 Mbytes, 4Mbytes and 8 Mbytes. Each indexed location in the cache contains four 64-bit doublewords protected by byte parity. Each time the cache is indexed, the tag and data portion of each set are accessed. The tag address is compared against the translated portion of the virtual address to determine if the data resides in the cache.

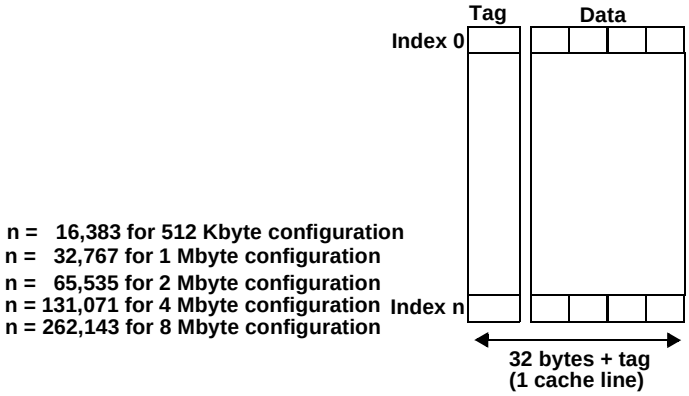
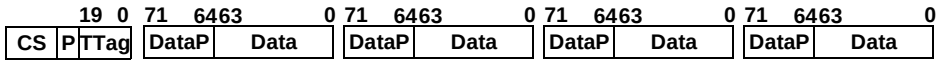


Figure 7.14 Tertiary Cache Organization

Figure 7.14 shows the organization of the tertiary cache. When the tertiary cache is indexed, each location contains a single cache line. Each cache line consists of 32 bytes of data protected by byte parity, a 17-bit physical tag address, and a tag valid bit. Figure 7.15 shows the tertiary cache line format.



CS	Tertiary cache state 000 = Invalid 100 = Valid All other values reserved
P	Tag parity. Provides even parity for the TTag and CS fields
TTag	17-bit tertiary cache tag (bits 35:19 of the physical address)
DataP	Even byte parity; 8 parity bit per doubleword of data
Data	Cache Data

Figure 7.15 Tertiary Cache line Format

7.5.2 Accessing the Tertiary Cache

The tertiary cache is only accessed on a secondary cache miss. Once the processor has determined that the requested address does not match the corresponding secondary cache tag, a tertiary cache access is initiated. Like the secondary cache, the tertiary cache is also physically indexed and physically tagged. The size of the index field varies depending on the size of the tertiary cache implemented. All five tertiary cache sizes are shown in Figure 7.16.

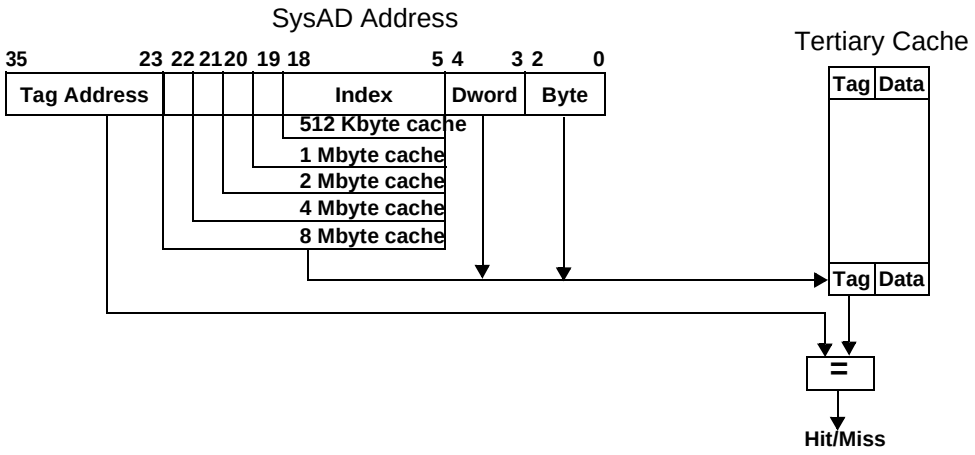


Figure 7.16 Accessing the Tertiary Cache

The lower bits of address are used for indexing the data cache as shown in Figure 7.16. Bits 18:5 are used for indexing a 512 Kbyte configuration. Bits 19:5 are used for indexing a 1 Mbyte configuration.Bits 20:5 are used for indexing a 2 Mbyte configuration. Bits 21:5 are used for indexing a 4 Mbyte configuration. Bits 22:5 are used for indexing an 8 Mbyte configuration. Within each indexed entry there are four 64-bit doublewords of data. Bits 4:3 are used to index one of these four doublewords. Bits 2:0 are used to index one of the eight bytes within each doubleword.

The tertiary cache is accessed simultaneously with main memory to minimize the overall main memory latency. If the data is found in the tertiary cache, the main memory access is simply aborted. However, if the data is not found in the tertiary cache, the main memory access has already begun and data can be retrieved as quickly as possible.

7.5.3 Tertiary Cache States

The RM7000 supports two tertiary cache states. These states are visible in bits 8:7 of the *TagLo* Register (tertiary cache format) after an **INDEX_LOAD_TAG_ED CACHE** instruction. The encoding is as follows:

- **Invalid (00):** A cache line that does not contain valid information is marked invalid and cannot be used. A cache line in any other state than invalid is assumed to contain valid information.
- **Valid (10):** A valid cache line contains valid information. A valid cache state is defined as a line that is in any state other than invalid.

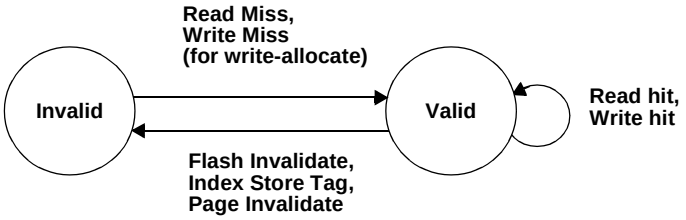


Figure 7.17 Tertiary Cache State Transitions

Figure 7.17 shows a cache state transition diagram for the tertiary cache

7.6 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 RM7000 does not provide any hardware cache coherency. Cache coherency must be handled by software.

7.6.1 Cache Coherency Attributes

Cache coherency attributes are necessary to ensure the consistency of data throughout the system. Bits in the translation look-aside buffer (TLB) control coherency on a per-page basis. Specifically, the TLB contains 3 bits per entry that provide the coherency attribute types shown in Table 7.3:.

The non-blocking coherencies implement a weakly ordered memory model. This model allows the following behaviors:

- The processor does not have to stall if a processor load request has not completed. Subsequent processor load or store operations may be started before the first has completed.
- Memory transactions can occur on the external pin bus out of program order. Program order of memory transactions can be enforced through the use of the **SYNC** instruction.
- Memory transactions can occur on the external pin bus even though the instructions which caused the memory transactions are later nullified in the pipeline due to an exception.

Such behaviors aid in achieving higher levels of processor throughput. However, some peripheral devices require a strongly ordered memory model (memory transactions occur in program order and only valid instructions can cause memory transactions). For this reason, it is strongly advised that such devices be referenced using the Uncached, Blocking coherency (Coherency Code 2). Processor read requests using this coherency stall the processor until the transaction completes. Processor write requests using this coherency are given the highest priority for accessing the external pin bus. These two properties ensure that processor load and store instructions using this coherency are completed in program order. For this reason, kseg1 and the uncached section of xkphys use Coherency Code 2.

Table 7.3: RM7000 Cache Coherency Attributes

Attribute Type	Coherency Code
Writethrough without write-allocate (non-blocking)	0
Writethrough with write-allocate (non-blocking)	1
Uncached, Blocking	2
Writeback (non-blocking)	3
Uncached, Non-Blocking	6
Bypass (non-blocking)	7

The following subsections describe each of the coherency attributes listed in Table 7.3:.

7.6.2 Write-through without Write-allocate (Code 0)

Cache lines with a *Write-through without Write-allocate* attribute type can reside only in the primary data cache. A load miss causes the processor to issue a block read request to a location within the cached page. This mode allows the primary data cache to be filled on a load miss but not on a store miss. A primary cache store hit causes data to be written to both the primary data cache and main memory simultaneously. The secondary and tertiary caches may be modified only during a line fill and then only due to the writeback of a displaced cache line. Partial (non-blocking) stores are never written to the secondary or tertiary caches.

On a primary cache load miss, data is fetched from main memory with a block read request and is written only to the primary data cache. Neither the secondary nor tertiary caches are accessed.

On a primary cache store miss, data is sent to main memory. The primary cache is not filled. The secondary and tertiary caches are not accessed.

This coherency follows the weakly ordered memory model described in section 7.6.1, “Cache Coherency Attributes”.

7.6.3 Write-through with Write-allocate (Code 1)

Lines with a *Write-through with Write-allocate* attribute type can reside only in the primary data cache; a load or store miss causes the processor to issue a block read request to a location within the cached page. A primary cache store hit causes data to be written to both the primary data cache and main memory simultaneously. The secondary and tertiary caches are modified only during a line fill and then only due to the writeback of a displaced cache line. Partial (non-blocking) stores are never written to the secondary or tertiary caches.

On a primary cache load miss, data is fetched from main memory with a block read request and is written only to the primary data cache. Neither the secondary nor tertiary caches are accessed.

On a primary cache store miss, data is sent to main memory. The primary cache line will be filled from main memory. This is the only difference between the *Write-Through without Write-Allocate* and the *Write-Through with Write-Allocate* coherency attributes.The secondary and tertiary caches are not accessed.

This coherency follows the weakly ordered memory model described in section 7.6.1, “Cache Coherency Attributes”.

7.6.4 Uncached, Blocking (Coherency Code 2)

Lines within an *Uncached* page are never in a cache. When a page 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 all caches) for any load or store to a location within that page. No caches are accessed when this coherency attribute is active.

Processor read requests using this coherency stall the processor until the transaction completes while processor write requests using this coherency are given the highest priority for accessing the external pin bus. These two properties ensure that processor load and store instructions using this coherency are completed in program order (strongly ordered memory model).

7.6.5 Writeback (Coherency Code 3)

Lines with the *Writeback* attribute can reside in a cache. On a data cache store hit, only the data cache is modified. The secondary cache, tertiary cache, and main memory are only modified if the cache line of a dirty block is needed for a newer access.

This mode allows the primary data cache to be filled on either a load miss or a store miss. A primary cache store hit causes data to be written to the primary data cache only. The secondary and tertiary caches are modified only during block writebacks and line fills. Partial (non-blocking) stores are never written to the secondary or tertiary caches. Main memory is modified only for block writebacks.

On a primary cache load or store miss, the RM7000 checks the secondary cache for the requested address. If there is a secondary cache hit the data is filled from the secondary cache. If a secondary cache miss occurs, data is fetched from either the tertiary cache or main memory and written to the secondary and primary caches.

On a secondary cache load or store miss, the RM7000 checks the tertiary cache for the requested address. For maximum performance, the RM7000 accesses the tertiary cache and main memory simultaneously with a block read request. If the address is present on the tertiary cache, data is fetched and written to both the secondary and primary caches and the pending memory access is aborted. If the address is not present in the tertiary cache, the main memory access has already begun and data is fetched as soon as it becomes available.

On a tertiary cache load or store miss, data is fetched from main memory and written to the tertiary, secondary, and primary caches.

This coherency follows the weakly ordered memory model described in section 7.6.1, “Cache Coherency Attributes”.

7.6.6 Uncached, Non-Blocking (Coherency Code 6)

Memory references that use the Uncached, Non-Blocking coherency do not affect the caches. Loads do not stall the pipeline until the requested data is required by a subsequent instruction. Uncached, Non-Blocking loads follow the out-of-order completion protocol and are allowed to complete out of the original instruction order. The Uncached, Non-Blocking coherency is invoked by using coherency code 6 when writing a JTLB entry. This coherency is useful for access to I/O devices that do not

require the strict instruction ordering of loads. For those devices that are sensitive to instruction order, the **SYNC** instruction can be used. These memory transactions can be strongly-ordered by bracketing load and store instructions with **SYNC** instructions.

Processor read requests using this coherency do not stall the processor. Processor write requests using this coherency are given the highest priority for accessing the external pin bus.This coherency follows the weakly ordered memory model described in section 7.6.1, “Cache Coherency Attributes”.

7.6.7 Bypass (Coherency Code 7)

The Bypass coherency allows bypassing of the secondary and tertiary caches. Memory references that use this coherency do not affect either the secondary or tertiary caches. The bypass coherency is invoked by using coherency code 7 when writing a JTLB entry.

On primary cache load misses, data is read directly from main memory with a block read request and written into the primary cache. On a store hit, only the primary data cache is modified. Main memory is modified only for block writebacks. The secondary and tertiary caches are not modified. On a store miss, data is read directly from main memory with a block read request.

This coherency follows the weakly ordered memory model described in section 7.6.1, “Cache Coherency Attributes”.

Table 7.4: shows a sequential list of events and the behavior of each cache coherency attribute. This table assumes the secondary and tertiary caches are enabled and includes the following abbreviations:

— Primary Data Cache - *Dcache*
— Primary Instruction Cache - *Icache*
— Secondary Cache - *Scache*
— Tertiary Cache - *Tcache*
— Writeback - *WB*
— Write-Allocate - *W/A*
— Write-Through - *W/T*
— Main Memory - *MM*
— Non-Blocking - *NB*

Table 7.4: Cache Events and Coherency Behavior

Event	Cache Coherency Attribute					
	0	1	2	3	6	7
	W/T without W/A	W/T with W/A	Uncached, Blocking	WB	Uncached, NB	Bypass
Dcache lookup	Yes	Yes	No	Yes	No	Yes
Fill Dcache on load miss	Yes	Yes	No	Yes	No	Yes
Fill Dcache on store miss	No	Yes	No	Yes	No	Yes
Store data modifies?	(1)	(1)	MM only	(2)	MM only	(3)
Load miss, lookup Scache. If Scache miss, fill Scache	No	No	No	Yes	No	No
Store miss, lookup Scache. If Scache miss, fill Scache	No	No	No	Yes	No	No
Dcache WB, line written back to the Scache	No WB’s	No WB’s	No WB’s	Yes (4)	No WB’s	No
Dcache WB, written back to main memory	No WB’s	No WB’s	No WB’s	Yes (4)	No WB’s	Yes
Load miss, lookup Tcache, if Tcache miss, fill Tcache	No	No	No	Yes	No	No
Store miss, lookup Tcache, If Tcache miss, fill Tcache	No	No	No	Yes	No	No

Event	Cache Coherency Attribute					
	0	1	2	3	6	7
	W/T without W/A	W/T with W/A	Uncached, Blocking	WB	Uncached, NB	Bypass
Scache WB written back to the Tcache/Main Memory	No WB's	No WB's	No WB's	Yes (5)	No WB's	No
Weakly Ordered Memory Model (load and stores can occur out of program order)	Yes	Yes	No	Yes	Yes	Yes

Note 1: Primary data cache and main memory are modified simultaneously. Secondary and tertiary cache may be modified only during line fills by primary cache displacements. Partial stores do not modify the secondary or tertiary caches.

Note 2: Only the primary data cache is modified directly. Main memory is modified only on block writebacks. The secondary and tertiary cache are modified only for block writebacks and line fills. Partial stores do not modify the secondary or tertiary caches.

Note 3: Only the primary data cache is modified directly. Main memory is modified only on block writebacks. The secondary and tertiary caches are never modified in this mode.

Note 4: Data is written back to the secondary cache only if the secondary cache contains the cache line. Otherwise data is written back directly to the tertiary cache and main memory.

Note 5: Data is written back from the secondary cache to main memory only if the secondary cache contains the most current data. If the secondary cache does not contain the most current data no writeback occurs.

7.7 Cache Maintenance

With multiple levels of on-chip memory, care must be taken to ensure that modified data has reached external memory before a process task switch.To flush all on-chip write buffers, software should use the **SYNC** instruction. This instruction will stall the processor until all pending store operations have reached the external pin bus and all pending load operations have completed by writing their destination registers. Previous implementations of the MIPS architecture such as the R5000 and R4700 did not implement this instruction. Refer to the MIPS IV Instruction Set Manual for a description of this instruction.The RM7000 implementation of the **SYNC** instruction affects all load/store operations.

Since data cache misses do not stall the RM7000, it becomes useful to speculatively preload data before the data is actually used. The RM7000 implements the **PREF** and **PREFX** data prefetch instructions for this purpose. Execution of the **PREF** instruction for integer operations, or the **PREFX** instruction for floating point operations, at the appropriate place in the code stream causes the RM7000 to initiate a memory transaction without stalling the processor. The requested block of data is then fetched and placed in the data cache. This allows techniques such as software pipelining and loop unrolling to be utilized. The **PREF** and **PREFX** instructions behave as special load instructions that cannot generate an address error nor a TLB exception. Previous implementations of the MIPS IV architecture such as the R5000 and RM52x0 did not implement these instructions. Refer to the MIPS IV Instruction Set Manual for a description of these instructions.

The RM7000 implementation of these instructions uses the hint field to specify the desired prefetch action for he data load prefetch. If the "hint" field contains any value other than 30, then the fetched data will be loaded. If a value of 30 is in the "hint" field, then the following operations can take place:

- If there is a primary Dcache hit, no action is taken.
- If there is a primary Dcache miss, then a cache is prepared for a store operation with no memory transaction taken (if the cacheline was valid and dirty, then it is written back):
  - a cacheline is allocated for the requested address,
  - the cacheline state is set to valid,
  - the W bit is set,
  - the data field is filled with zeroes.

The **CACHE** instruction is used when performing maintenance of the caches. Unlike previous generation MIPS processors, the RM7000 may not stall until the **CACHE** operation is finished. Refer to section 18.3, "CACHE (cache management)", for more information on the **CACHE** instruction.

The RM7000 contains three “Hit” type cache operations; *Hit_Invalidate, Hit_Writeback_Invalidate,* and *Hit_Writeback*. The RM7000 treats the “Hit” type **CACHE** operation much like a load instruction and allows the instruction to be pipelined. If there is no cache hit, the “Hit” type can be executed without any pipeline stall. If there is a cache hit, but the cache line is clean, the only latency incurred is that required for invalidating the tag RAM.

Since the secondary cache “Hit” cache operations flush the primary data cache at the same time, it is not necessary to use separate primary cache operation if the secondary cache is being written back or invalidated.

The RM7000 allows the *Hit_Writeback and Hit_Writeback_Invalidate* types of primary and secondary **CACHE** operations to be executed while in user mode. In operating systems that run user mode processes, this feature allows the cache to be flushed without the overhead of a system call. User-mode **CACHE** operations are enabled by setting bit 27 of the CP0 *Status* register. This bit is cleared on power-up and during a hard reset.

The RM7000 conforms to the MIPS R4000 behavior for the “Hit” type of secondary **CACHE** operations. A secondary cache “Hit” type CACHE operation also checks the primary data cache for any matching blocks and responds as follows:

1. If a secondary cache *Hit_Writeback or Hit_Writeback_Invalidate* cache operation is requested, the newest data is written back to system memory, even if the newest data is found in the primary data cache.
2. If a secondary cache *Hit_Invalidate or Hit_Writeback_Invalidate* **CACHE** operation is requested, any matching block in the primary data cache is also invalidated.

The RM7000 differs from the R4000 in it’s behavior for the “Index” type of secondary **CACHE** instruction. The “Index” type of secondary **CACHE** operation only operates on the secondary cache and does not probe the primary caches. In addition, secondary cache **CACHE** operations do not affect the primary instruction cache.

7.8 Code Examples

Since the RM7000 has 4-way set associative caches, four **CACHE** instructions are required to deal with one cache index location.

7.8.1 Data Cache Invalidation

The following code example can be used for dcache line invalidation.

```
        cache  mtco   r0,C0_TAGLO
        nop                                     # assume r1 holds address
                                                # which points to cache index
        cache  Index_Store_Tag_D,0x0(r1)        # invalidate one set
        cache  Index_Store_Tag_D,0x1000(r1)     # invalidate 2nd set
        cache  Index_Store_Tag_D,0x2000(r1)     # invalidate 3rd set
        cache  Index_Store_Tag_D,0x3000(r1)     # invalidate 4th set
```

7.8.2 Data Cache and Secondary Cache Flush

The following code can be used to flush a portion of the dcache and secondary cache, perhaps after a DMA transfer.

```
        li     r1,Your_Start_Address            # beginning address,
                                                # should be aligned to a cacheline
        li     r2,Your_End_Address              # last address,
                                                # should be aligned to a cacheline
A:      cache  Hit_Writeback_Invalidate_SD,0x0(r1)# writeback & invalidate
                                                # the desired address
        bne    r1,r2,A
        addi   r1,0x20                          # do next cacheline
```

7.8.3 Data Cache Locking

The following code example can be used for locking the data cache.

```
li     r1,LOCK_DCACHE | LOCK_SETO       #setup set 0 for locking
nop
nop
lw     r3,0(r10)                        #lock this data
nop
nop
mtc0   r0,C0_ECC                        #next load not locked
```

7.8.4 Instruction Cache Locking

The following code example can be used for locking the instruction cache. When locking the primary instruction cache, the RM7000 should be executing code uncached. Executing code from the instruction cache while attempting to lock it can result in unpredictable behavior.

```
li     r1,LOCK_ICACHE | LOCK_SETO       #setup set 0 for locking
mtc0   r1,C0_ECC
nop
nop
cache  Fill_I,0(r10)                    #lock this code
nop
nop
mtc0   r0,C0_ECC                        #next instr fetch not locked
```

7.8.5 Synchronizing the Internal Caches

The following code is an example of how the **SYNC** instruction might be used to ensure that external memory is synchronized with the internal caches.

```
                          # assume in some exception handler
sw     r1,0(r2)           # modify some external control register
                          # that is mapped in cached space.
sync                      # make sure store reaches memory first
eret                      # before return to user program
```

7.8.6 Using the Prefetch Instructions

The following code is an example of using **PREF** instructions to speculatively chase pointers in a linked-list. The first **PREF** instruction is meant to hide memory latency for the third **LW** instruction while the second **PREF** instruction is meant to hide memory latency for the first **LW** instruction.

```
A:      lw      r1,NEXT(r2)         # get address of entry I in linked list
        pref    DATA(r1)            # speculatively fetch data from entry I
        beq     r1,r0,B             # is entry I-1 end of list and entry I
                                    # does not exist?
        or      r2,r1,r1            # setup for entry I
        lw      r1,VALID(r2)        # is entry I valid?
        beq     r1,r0,A             # if not valid, proceed to next entry
        pref    NEXT(r2)            # speculatively fetch address of entry I+1
        lw      r3,DATA(r2)         # get data from entry I
        div     r10,r3,r10          # do something with data
        sw      r10,OUTPUT(r2)      # save output
        beq     r0,r0,A             # do it again
        nop
```

Section 8 Processor Bus Interface

The RM7000 processor provides an efficient system interface allowing communication and the transfer of data and addresses between the processor and the system. The RM7000 multiplies the input **SysClock** by 2, 2.5, 3, 3.5, 4, 4.5, 5, 6, 7, 8, or 9 to produce the pipeline clock. The system interface consists of a 64-bit Address/Data (**SysAD**) bus and a 9-bit command bus. In addition, there are 10 handshake signals. The system interface has a simple timing specification for transferring data between the processor and memory.

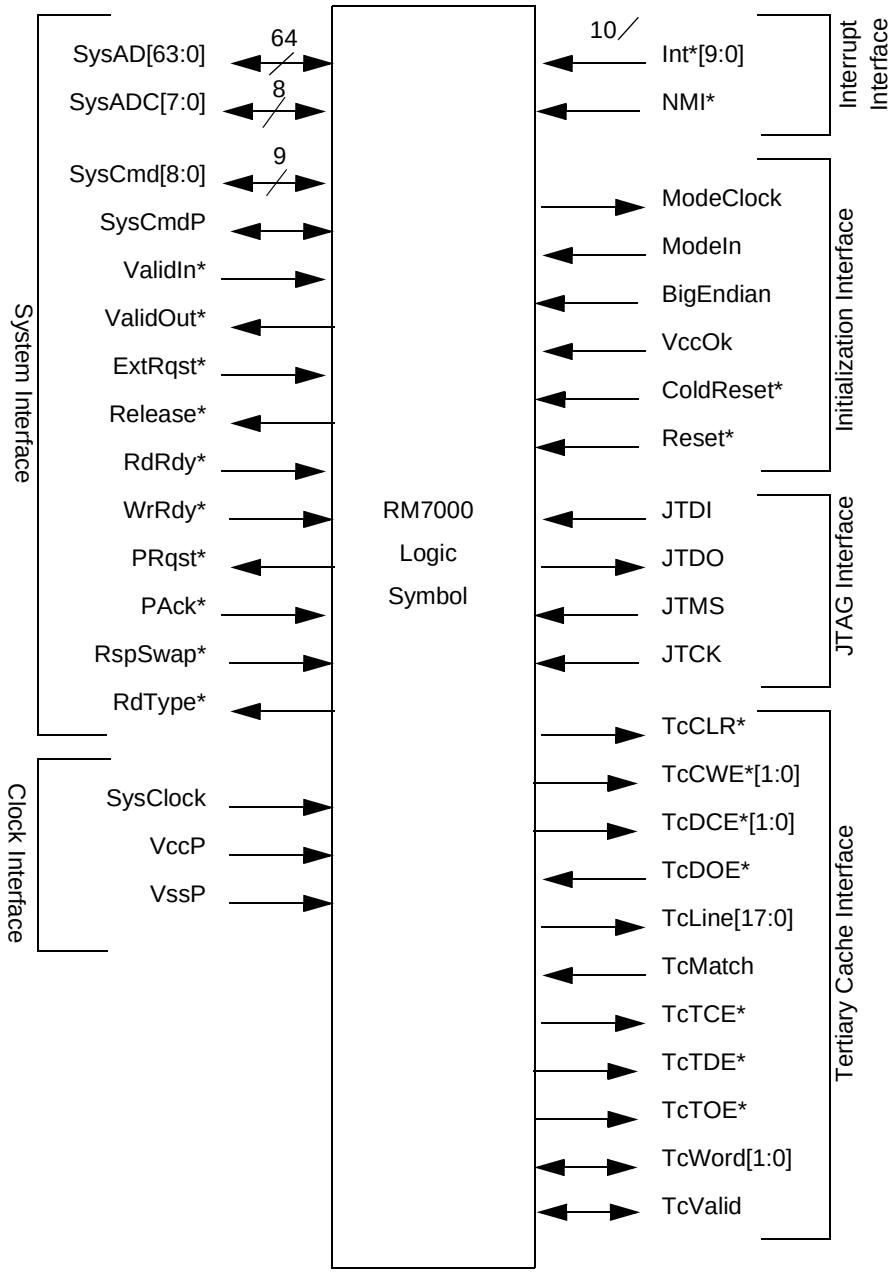


Figure 8.1 RM7000 Processor Signal Groups

The interrupt interface has been expanded and includes 10 external maskable interrupt signals and one non-maskable interrupt.

The tertiary cache interface on the RM7000 processor is identical to the secondary cache interface on the R5000 processor.

Figure 8.1 shows the signal groupings for the RM7000. These include the system interface, clock interface, interrupt interface, tertiary cache interface, initialization interface, and Joint Test Action Group (JTAG) interface.

Active-low signals have a trailing asterisk (*). The arrows indicate the signal type, input, output, or bidirectional.

8.1 Non-Pendant Bus Mode

The RM7000 supports two outstanding read transactions that can be completed out-of-order. Three new system interface pins, **PRqst***, **PAck***, and **RspSwap***, are used to support two outstanding read transactions and out-of-order completion.

In previous generations the processor would drive read address and control information onto the bus, and assert **Release***, allowing data to be driven back to the processor by the memory system. The processor could not regain control of the bus until after the last data for the read request was returned by the memory system. In non-pendant bus mode the RM7000 still asserts **Release*** to relinquish control of the bus. However, if a second read transaction can be issued before the first one is completed, the RM7000 asserts **PRqst*** to request return of control of the **SysAD** bus. The memory controller responds by asserting **PAck***, indicating that bus mastership has been returned to the processor. Once this occurs, the RM7000 processor can initiate another memory transaction. The processor then drives the new address and control information for the second memory transaction and asserts **Release***.

The external memory controller determines in what order the data is returned from memory. For example, if the first transaction takes longer to complete than the second transaction, the memory controller can return data from the second transaction first. The assertion of **RspSwap*** for one cycle by the memory controller two or more cycles prior to data being driven out of the memory system indicates to the processor that the pending transactions are being completed out-of-order. If **RspSwap*** is not asserted for one cycle, two or more cycle prior to data being driven indicates to the RM7000 processor that the pending transactions are being completed in order.

8.2 System Interface Signals

System interface signals provide the connection between the RM7000 processor and the other components in the system. As shown in Figure 8.1, the system interface consists of a 64-bit multiplexed **SysAD** address/data bus, a 9-bit command bus, an 8-bit parity check bus, and control signals.

8.2.1 System Address/Data Bus

The multiplexed System Address Data (**SysAD**) bus is used to transfer addresses and data between the RM7000 and the rest of the system. The **SysAD** bus is protected with an 8-bit parity check bus (**SysADC**).

The flexible system interface allows for connection to memory and I/O systems of varying frequencies. The data rate and the bus frequency at which the RM7000 transmits data to the system interface are programmable via boot time mode control bits. In addition to the standard clock divide ratios, the RM7000 provides three half-integer multipliers (2.5, 3.5 and 4.5) to provide greater granularity between the pipeline and system clock frequencies. In addition, the rate at which the processor receives data is fully controlled by the external device. Refer to Section 10, “Initialization Interface”, for more information on the boot time mode stream bits.

8.2.2 System Command Bus

The 9-bit System Command (**SysCmd**) interface indicates whether the **SysAD** bus carries address or data. If **SysAD** carries an address, then the **SysCmd** bus indicates the type of transaction (read or write). If **SysAD** carries data, the **SysCmd** bus provides information about the data transaction such as; last data word transmitted, or the data contains an error. The **SysCmd** bus is bidirectional to support both processor requests and external requests to the RM7000. Processor requests are initiated by the RM7000 and responded to by an external device. External requests are issued by an external device and require the RM7000 to respond.

The RM7000 supports one-to-eight-byte transfers, as well as block transfers on the **SysAD** bus. In the case of a sub-double word transfer, the three low-order address bits give the byte address of the transfer, and the **SysCmd** bus indicates the number of bytes being transferred.

8.2.3 Handshake Signals

There are ten handshake signals on the system interface. Two of these, **RdRdy*** and **WrRdy***, are used by an external device to indicate to the RM7000 whether it can accept a new read or write transaction. The RM7000 samples these signals before deasserting the address on read and write requests.

ExtRqst* and **Release*** are used to transfer control of the **SysAD** and **SysCmd** buses from the processor to an external device. The external device asserts **ExtRqst*** to request control of the bus. The RM7000 responds by asserting **Release*** to relinquish control of the bus.

ValidOut* and **ValidIn*** are used by the RM7000 and the external device respectively to indicate that there is a valid command or data on the **SysAD** and **SysCmd** buses. The RM7000 asserts **ValidOut*** when it is driving these buses with a valid command or data, and the external device drives **ValidIn*** when it has control of the buses and is driving a valid command or data.

PRqst* and **PAck*** are used by the RM7000 to regain control of the bus on demand when the processor wishes to issue a second non-pendant read request. The handshake mechanism is similar to that of the **ExtRqst*** and **Release*** signals. Once the processor has relinquished control of the bus by asserting **Release***, it may regain control of the bus by asserting **PRqst***. The external master responds by asserting **PAck***, thereby returning control of the bus to the RM7000.

The **RspSwap*** signal is an input to the RM7000 and is asserted for one cycle by the memory controller to indicate that a read response is being returned out of order when there are multiple outstanding read requests. If the signal is not asserted, the RM7000 assumes that the cycles are being completed in order.

The **RdType** signal is driven by the processor and indicates whether the read cycle is an instruction read or a data read.

Table 8.1: shows a listing of the system interface signals

Table 8.1: System Interface Signals

Name	Definition	Direction	Description
SysAD(63:0)	System Address/Data bus	Input/Output	Address and data bus for communication between the processor and an external agent.
SysADC(7:0)	System Address/Data Check bus	Input/Output	Bus containing parity for the SysAD bus during data cycles.
SysCmd(8:0)	System Command/data identifier bus	Input/Output	A 9-bit bus for command and data identifier transmission between the processor and an external agent.
SysCmdP	System Command/data identifier bus parity	Input/Output	Unused on input and zero on output. This signal is defined to maintain R4000 compatibility.
RdRdy*	Read Ready	Input	An external agent asserts RdRdy* to indicate that it can accept a processor read request.
WrRdy*	Write Ready	Input	An external agent asserts WrRdy* when it can accept a processor write request.
ExtRqst*	External Request	Input	An external agent asserts ExtRqst* to request control of the System interface. The processor grants the request by asserting Release*.
Release*	Release interface	Output	In response to the assertion of ExtRqst*, the processor asserts Release*, signalling to an external agent that it has control of the System interface.
ValidIn*	Valid Input	Input	The external agent asserts ValidIn* when it is driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.

Name	Definition	Direction	Description
ValidOut*	Valid Output	Output	The processor asserts ValidOut* when it is driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.
PRqst*	Processor Request	Output	This signal is asserted by the RM7000 to request control of the system interface.
PAck*	Processor Acknowledge	Input	Assertion of this signal by the external agent indicates to the RM7000 that it has been granted control of the system interface.
RspSwap*	Response Swap	Input	RspSwap* is asserted by the external agent to signal the processor that a memory reference will be completed out of order. In order for the processor to have adequate time to switch the address to the tertiary cache, this signal must be asserted a minimum of two cycles prior to the data itself being presented. Note that this signal works as a toggle. For each cycle that it is held asserted, the order of return is reversed. By default, any time the processor issues a second read, it is assumed that the reads will be returned in order. No action is required if the reads are returned in order.
RdType	Read Type	Output	During the address cycle of a read request, this pin is asserted high for instruction reads and deasserted low for data reads.

8.3 Clock Interface Signals

Table 8.2: lists the Clock interface signals.

Table 8.2: Clock Interface Signals

Name	Definition	Direction	Description
SysClock	System Clock	Input	Master clock input used as the system interface reference clock. Pipeline operation frequency is derived by multiplying this clock up by the factor selected during boot initialization.
VccP	Quiet Vcc for PLL	Input	Quiet Vcc for the internal phase locked loop.
VssP	Quiet Vss for PLL	Input	Quiet Vss for the internal phase locked loop.

8.4 Interrupt Interface Signals

The Interrupt interface signals make up the interface used by external agents to interrupt the RM7000 processor. The RM7000 supports an enhanced interrupt handling mechanism that includes support for ten external interrupts. Table 8.3: lists the Interrupt interface signals.

Table 8.3: Interrupt Interface Signals

Name	Definition	Direction	Description
Int*(9:0)	Interrupt	Input	Ten general processor interrupts. These signals are bit-wise OR'ed with bits 9:0 of the interrupt register.
NMI*	Non-Maskable Interrupt	Input	Non-Maskable interrupt, OR'ed with bit 10 of the interrupt register.

8.5 Initialization Interface Signals

The Initialization interface signals allows an external agent to initialize the processors operating parameters. The fundamental operating modes for the processor are initialized by the boot-time mode control interface. The boot-time mode control serial interface operates at a frequency equivalent to **SysClock** divided by 256 (SysClock/256). This low frequency allows the initialization information to be kept in a low cost EPROM. This informational could also be generated by a system interface ASIC.

Immediately after the **VccOk** signal is asserted, the processor reads a serial bit stream of 256 bits to initialize all the fundamental operation modes. **ModeClock** runs continuously from the assertion of **VccOk**.

Table 8.4: lists the Initialization interface signals.

Table 8.4: Initialization Interface Signals

Name	Definition	Direction	Description
BigEndian	Endian Mode Select	Input	Allows the system to change the processor addressing mode without rewriting the mode ROM. If endianness is to be specified via the BigEndian pin, program mode ROM bit 8 to zero. If endianness is to be specified by the mode ROM, the BigEndian pin is grounded.
ColdReset*	Cold Reset	Input	This signal must be asserted for a power on reset or a cold reset. ColdReset* must be deasserted synchronously with SysClock.
ModeClock	Boot Mode Clock	Output	Serial boot-mode data clock output; runs at the system clock frequency divided by 256: (SysClock/256).
ModeIn	Boot Mode DataIn	Input	Serial boot-mode data input.
Reset*	Reset	Input	This signal must be asserted for any reset sequence. It may be asserted synchronously or asynchronously for a cold reset, or synchronously to initiate a warm reset. Reset* must be deasserted synchronously with SysClock.
VccOk	Vcc is OK	Input	When asserted, this signal indicates to the processor that the +2.5 volt power supply has been above 2.0 volts and a stable MasterClock has been provided for more than 100 milliseconds. The assertion of VccOk initiates the reading of the boot-time mode control serial stream.

8.6 Tertiary Cache Interface

The RM7000 tertiary cache interface is identical to the MIPS R5000 secondary cache interface and provides all of the control signals necessary for interfacing to external data RAMs and tag RAMs. The tertiary cache is accessed only on a secondary cache miss. A tertiary cache access is initiated simultaneously with a main memory access, allowing some of the memory access time to be hidden in the event of a tertiary cache miss. If there is a tertiary cache “hit” the main memory access is aborted. However, if there is a tertiary cache “miss”, the main memory access is already in progress and the data can be retrieved in the shortest possible time. External logic asserts **TcMatch** to indicate that the requested data resides in the tertiary cache. The **TcValid** signal is driven by the RM7000 on a write operation to indicate whether the cache line is valid or invalid. On tag read operations the tag RAM drives **TcValid** to indicates the state of the line being accessed.

In addition to the standard data RAM and tag RAM read, write, output, and chip enables, the tertiary cache interface supports a block clear function. The RM7000 asserts **TcCLR*** to invalidate all entries in the tertiary cache tag RAM. This is accomplished by clearing all of the valid bits for each tag RAM entry. Note that this signal is valid only for tag RAMs that support the block clear function, which is not required in order for the cache to operate.

Table 8.5: shows a listing of the tertiary cache interface signals.

Table 8.5: Tertiary Cache Interface Signals

Name	Definition	Direction	Description
TcCLR*	Tertiary Cache block Clear	Output	For tag RAMs that support the block clear function, the assertion of TcCLR* by the RM7000 causes all valid bits to be cleared in the tag RAMs. Many RAM's may not support a block clear. It is not required for the cache to operate.
TcCWE*(1:0)	Tertiary Cache Write Enable	Output	Assertion of these signals by the RM7000 causes write data to be written into the cache. Two identical signals are provided to balance the capacitive load relative to the remaining cache interface signals.
TcDCE*(1:0)	Tertiary Cache Data RAM Chip Enables	Output	Assertion of these signals by the RM7000 enables the Data RAM's to do a read or write operation. Two identical signals are provided to balance the capacitive load relative to the remaining cache interface signals.
TcDOE*	Tertiary Cache Data Output Enable	Input	Assertion of this signal by external logic causes the data RAM's to drive data onto their I/O pins. This signal is monitored by the processor to recognize the last data cycle one cycle before it is on the SysAD bus.
TcLine (17:0)	Tertiary Cache Line Index	Output	Tertiary cache line index.
TcMatch	Tertiary Cache Tag Match	Input	Assertion of this signal by the tag RAM indicates that a match occurred between the value on its data inputs and the contents of its RAM at the value on its address inputs.
TcTCE*	Tertiary Cache Tag Chip Enable	Output	Assertion of this signal by the RM7000 causes either a read or a write of the Tag RAM depending on the state of the Tag RAM's write enable signal. This signal is monitored by the external agent and indicates that a tertiary cache access is starting.
TcTDE*	Tertiary Cache Tag Data Enable	Output	Assertion of this signal by the RM7000 causes the value on the data inputs of the Tag RAM to be latched into the RAM. If a refill of the RAM is necessary, this latched value is written into the Tag RAM array. Latching the tag allows a shared address/data bus to be used without incurring a penalty during the refill sequence.
TcTOE*	Tertiary Cache Tag Output Enable	Output	This signal is asserted by the RM7000 and causes the Tag RAM to drive data onto its I/O pins.
TcWord (1:0)	Tertiary Cache double Word index	Input/Output	This signal is asserted by the RM7000 on a cache hit and by the external agent on a cache miss refill.
TcValid	Tertiary Cache Valid	Input/Output	This signal is driven by the RM7000 during a write operation and indicates that the cache line is valid. If this signal is not asserted during a write, the corresponding cache line is invalid. On tag read operations the Tag RAM drives this signal to indicate the state of the cache line.

8.7 JTAG Interface Signals

The RM7000 interface supports JTAG boundary scan in conformance with the IEEE 1149.1 specification. The JTAG interface is especially helpful for checking the integrity of the processor pin connections. Table 8.6: lists the JTAG interface signals.

Table 8.6: JTAG Interface Signals

Name	Definition	Direction	Description
JTDI	JTAG Data In	Input	JTAG serial scan data input.
JTCK	JTAG Clock input	Input	JTAG serial scan clock
JTDO	JTAG Data Out	Output	JTAG serial scan data output.
JTMS	JTAG Mode Select	Input	JTAG command, indicates the incoming serial data is command data.

Section 9 Clock Interface

The RM7000 incorporates a single master input clock, **SysClock**, upon which all internal and external clocks are based. All output timings are relative to this clock.

The internal pipeline clock, PClock, is derived by multiplying **SysClock** by the multiplication factor selected by bits [20, 7:5] of the boot mode settings during power-up. PClock is phase-aligned to **SysClock**. All internal registers and latches use PClock.

9.1 PClock Divisors

The RM7000 incorporates four new clock divisor ratios. In addition to the standard clock ratios (2, 3, 4, 5, 6, 7, 8, and 9), the RM7000 supports ratios of 2.5, 3.5 and 4.5. These non-integral clock divisors help compensate for increasing internal clock frequencies by giving the designer of wider range of bus frequencies to choose from.

9.2 Data Alignment to SysClock

Processor input data must be stable for a maximum of T_{DS} ns before the rising edge of **SysClock** and must remain stable a minimum of T_{DH} ns after the rising edge of **SysClock**. An input timing diagram is shown in Figure 9.2.

Processor output data changes a minimum of T_{DOmin} ns and becomes stable a maximum of T_{DOmax} ns after the rising edge of **SysClock**. 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. An output timing diagram is shown in Figure 9.3.

9.3 Phase-Locked Loop (PLL)

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

Clocks generated using PLL circuits contain some inherent inaccuracy, or jitter; a clock aligned with **SysClock** by the PLL can lead or trail **SysClock** by as much as the related maximum jitter T_{jo} allowed by the individual vendor. The $T_{jitterIn}$ parameter must be added to the T_{ds}, T_{dh}, and T_{do} parameters, and subtracted from the T_{dm} parameters to get the total input and output timing parameters.

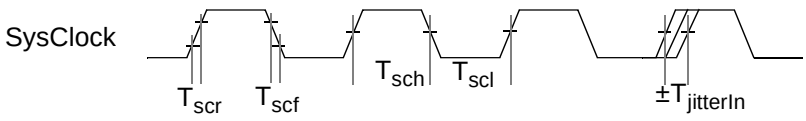


Figure 9.1 SysClock Timing

Figure 9.1 shows the **SysClock** timing parameters. The **SysClock** input must meet the maximum rise time (T_{scr}), maximum fall time (T_{scf}), minimum T_{sch} time, minimum T_{scl} time, and $T_{jitterIn}$ parameters for proper operation of the PLL. Refer to the RM7000 datasheet for these values.

9.4 Input and Output Timing Diagrams

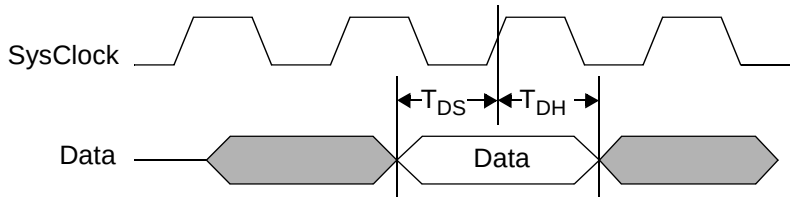


Figure 9.2 Input Timing

Figure 9.2 shows the input timing parameters.

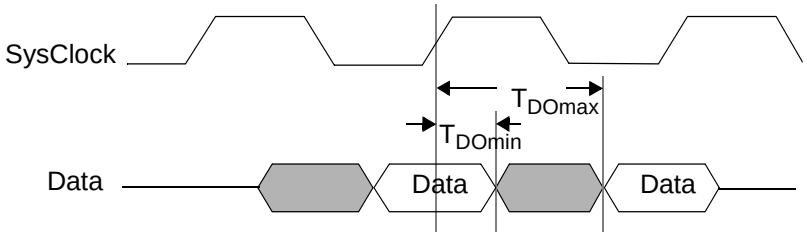


Figure 9.3 Output Timing

Figure 9.3 shows the output timing parameters measured at the midpoint of the rising clock edge.

9.5 Boot Mode Clock

The RM7000 drives the **ModeClock** output which is used as the clock for the serial boot serial data stream. This signal is derived by dividing the **SysClock** frequency internally by 256. The boot mode clock commences at **VccOk** being asserted and continues running after the boot sequence has completed.

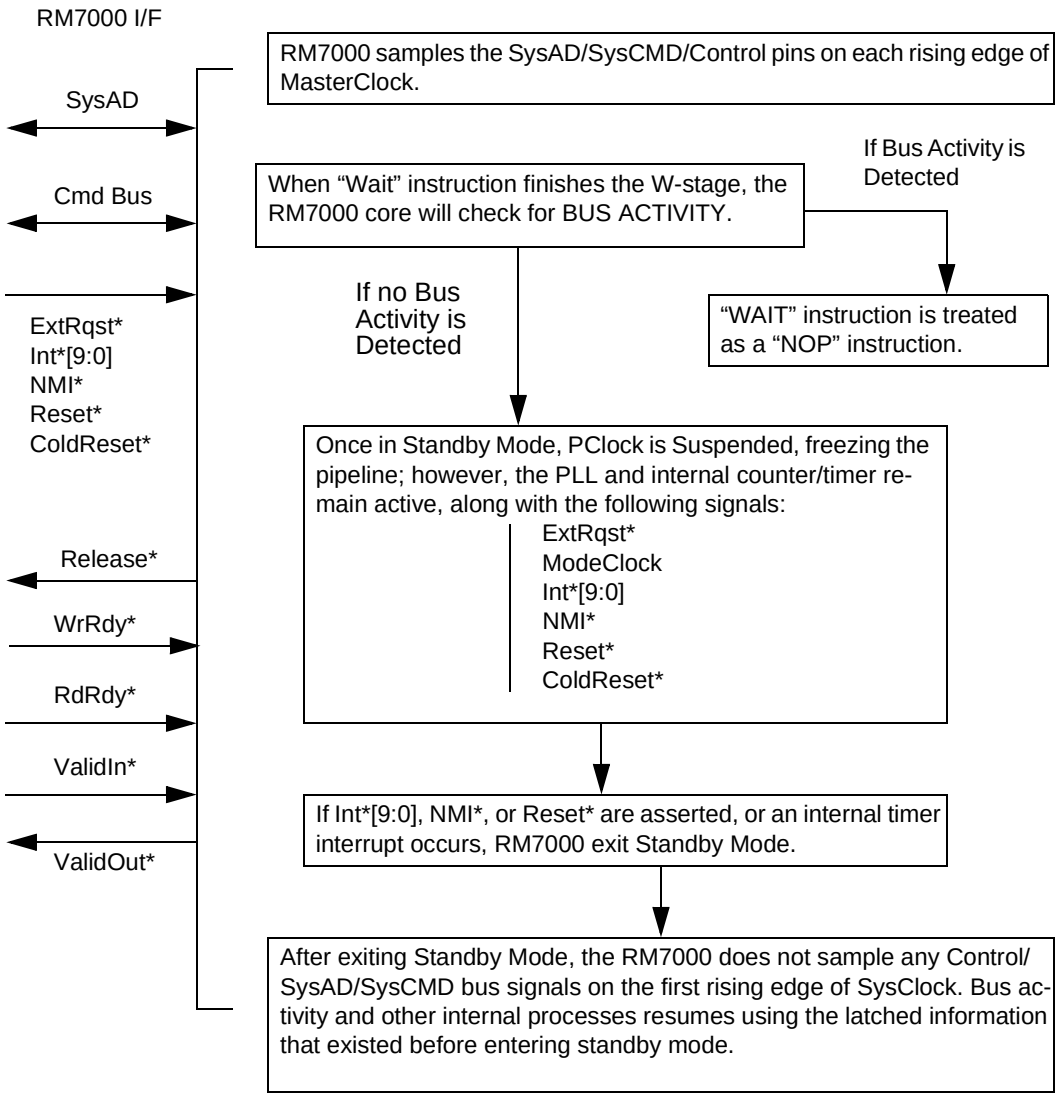
9.6 Standby Mode

The RM7000 provides a Standby mode for reducing the amount of power consumed by the internal core. In Standby mode the processor bus clocks are stopped and the RM7000 is placed in an idle state. During this time the PLL, internal timer/counter, and the boot ROM clock remain active and continue to operate, along with the following “wake-up” input signals:

- **Int*[9:0]**
- **NMI***
- **ExtReq***
- **Reset***
- **ColdReset***

Execution of a **WAIT** instruction places the RM7000 in Standby mode. Once the processor enters the Standby mode, any interrupt, including an internally generated timer interrupt, causes the processor to exit Standby mode and resume normal execution.

To enter Standby mode the **SysAD** bus must be idle when the **WAIT** instruction finishes the W stage of the pipeline. If the bus is not idle the **WAIT** instruction is treated as a **NOP**. The **WAIT** instruction is typically inserted in the idle loop of the operating system or real time executive. Figure 9.4 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[9:0], Reset*, ColdReset* and ExtRqst*, must remain unchanged.*

Figure 9.4 Standby Mode Operation

9.7 PLL Analog Power Filtering

The Phase Lock Loop circuit requires several passive components for proper operation. The recommended configuration is shown in Figure 9.5. The smaller value capacitor should be placed as close as possible to the device.

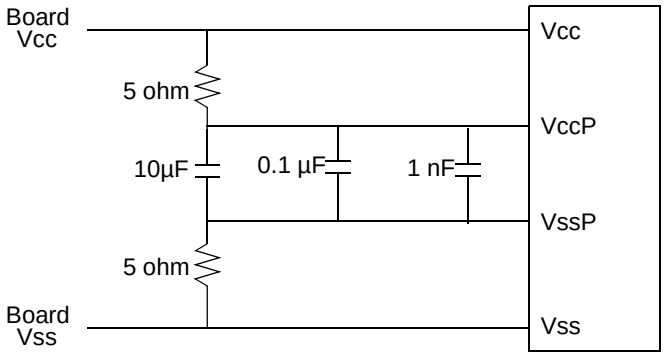


Figure 9.5 PLL Filter Circuit

Section 10 Initialization Interface

The RM7000 is initialized based on the following three types of reset conditions.

- **Power-on reset**: This type of reset occurs when the system is first turned on. A power-on reset sequence causes the RM7000 to initialize the internal state machines and begin reading from the boot ROM.
- **Cold reset**: A cold reset condition restarts all clocks, but the power supply remains stable. A cold reset completely reinitializes the internal state machines of the processor without saving any state information.
- **Warm reset**: A warm reset condition restarts the processor, but does not affect the clocks or the processors internal state machines.

The Initialization interface is a serial interface that operates at the frequency of the **SysClock** divided by 256: (**SysClock**/256). This low-frequency operation allows the initialization information to be stored in a low-cost ROM device.

There are also several CP0 register that can be used to set, or read, the configuration state of the processor. These registers include the *Config Register*, the *Info Register*, and the *Processor ID (PRID) Register*.

10.1 Processor Reset Signals

The RM7000 processor incorporates three types of reset input signals; **VccOk, ColdReset***, and **Reset***.

VccOk: The **VccOk** signals is asserted (high) by system logic and indicates to the processor that the following voltage parameters have been met. Each of these voltages must be at 90% of the target before **VccOk** is asserted.

- The +3.3 volt power supply (**VccIO**) has been above 3.0 volts for more than 100 milliseconds.
- The +2.5 volt power supply (**VccP**) has been above 2.25 volts for more than 100 milliseconds.
- The +2.5 volt power supply (**VccInt**) has been above 2.25 volts for more than 100 milliseconds.

The assertion of **VccOk** initiates the reading of the boot-time mode control serial stream. Refer to section 10.2, “Initialization Sequence” on page 113 for more information.

ColdReset*: The **ColdReset*** signal must be asserted (low) for either a power-on reset or a cold reset. **ColdReset*** must be deasserted synchronously with **SysClock**.

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 deasserted synchronously with **SysClock**.

ModeIn: Serial boot mode data in.

ModeClock: Serial boot mode data clock. This clock operates at the **SysClock** frequency divided by 256 (**SysClock**/256).

Table 10.1: RM7000 Processor Signal Summary

Description	Name	I/O	Asserted State	Tri-State	Reset State
System address/data bus	SysAD[63:0]	I/O	High	Yes	(1)
System address/data check bus	SysADC[7:0]	I/O	High	Yes	(1)
System command/data identifier bus	SysCmd[8:0]	I/O	High	Yes	(1)
System command/data identifier bus parity	SysCmdP	I/O	High	Yes	(1)
Valid Input	ValidIn*	I	Low	No	N/A
Valid Output	ValidOut*	O	Low	Yes	(2)
External request	ExtRqst*	I	Low	No	N/A
Release interface	Release*	O	Low	Yes	(2)
Read ready	RdRdy*	I	Low	No	N/A
Processor Request	PRqst*	O	Low	Yes	(2)
Processor Acknowledge	PAck*	I	Low	No	N/A
Read Type	RdType	O	High	Yes	(2)
Response Swap	RspSwp*	I	Low	No	N/A
Write ready	WrRdy*	I	Low	No	N/A
Interrupts	Int*[9:0]	I	Low	No	N/A
Non-Maskable Interrupt	NMI*	I	Low	No	N/A
Boot mode data in	ModeIn	I	High	No	N/A
Boot Mode clock	ModeClock	O	High	No	(4)
Master Clock	MasterClock	I	High	No	N/A
Vcc is within specified range	VccOK	I	High	No	N/A
Cold Reset	ColdReset*	I	Low	No	N/A
Reset	Reset*	I	Low	No	N/A
Big Endian	Bigendian	I	High	No	N/A
JTAG Data In	JTDI	I	High	No	N/A
JTAG Clock	JTCK	I	High	No	N/A
JTAG Data Output	JTDO	O	High	Yes	High
JTAG Command	JTMS	I	High	No	N/A
Tertiary Cache Flash Clear	TcCLR*	O	Low	Yes	(2)
Tertiary Cache Chip Write Enable	TcCWE(1:0)*	O	Low	Yes	(2)
Tertiary Cache Data Chip Enable	TcDCE(1:0)*	O	Low	Yes	(2)
Tertiary Cache Data Output Enable	TcDOE*	I	Low	No	N/A
Tertiary Cache TAG Match	TcMatch	I	High	No	N/A
Tertiary Cache TAG Chip Enable	TcTCE*	O	Low	Yes	(2)
Tertiary Cache Device Enable	TcTDE*	O	Low	Yes	(2)
Tertiary Cache TAG Output Enable	TcTOE*	O	Low	Yes	(2)
Tertiary Cache Line index	TcLine[17:0]	O	High	Yes	(1)
Tertiary Cache Word index	TcWord[1:0]	I/O	High	Yes	(1)
Tertiary Cache TAG Valid	TcValid	I/O	High	Yes	(2)

Key to Reset State Column:

Note 1: All I/O pins (SysAD[63:0], SysADC[7:0], etc) remain tri-stated until the Reset* signal deasserts.

Note 2: All output only pins (ValidOut*, Release*, etc), except the clocks, are tri-stated until the ColdReset* signal deasserts.

Note 3: All clocks, except ModeClock, are tri-stated until VccOK asserts.

Note 4: ModeClock is always driven

Note 5: NA - Not Applicable to input pins.

10.1.1 Power-on Reset

The sequence for a power-on reset is listed below. Figure 10.1 shows the power-on system reset timing diagram.

1. Power-on reset applies a stable **VccIO** of at least +3.0 volts from the +3.3 volt power supply to the processor, a stable **VccP** of at least +2.25 volts from the +2.5 volt power supply, and a stable **VccInt** of at least +2.25 volts from the +2.25 volt power supply. It also supplies a stable, continuous system clock at the processor operational frequency.
2. After at least 100 ms of stable **VccIO**, **VccP**, **VccInt**, and **SysClock**, the system asserts the **VccOK** signal, which initializes the processor operating parameters. After the mode bits have been read in, the processor allows its internal phase locked loops to lock, stabilizing the processor internal clock (**PClock).**
3. **ColdReset*** is asserted for at least 64K (2^{16}) **SysClock** 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 **SysClock**. **ColdReset*** must be asserted when **VccOK** asserts. The behavior of the processor is undefined if **VccOK** asserts while **ColdReset*** is deasserted.
4. After **ColdReset*** is deasserted synchronously, **Reset*** is deasserted to allow the processor to begin running. Note that **Reset*** must be held asserted for at least 64 **SysClock** cycles after the deassertion of **ColdReset***. **Reset*** must be deasserted synchronously with **SysClock**.

Note: **ColdReset*** *must be asserted when* **VccOK** *asserts. The behavior of the processor is undefined if* **VccOK** *asserts while* **ColdReset*** *is deasserted.*

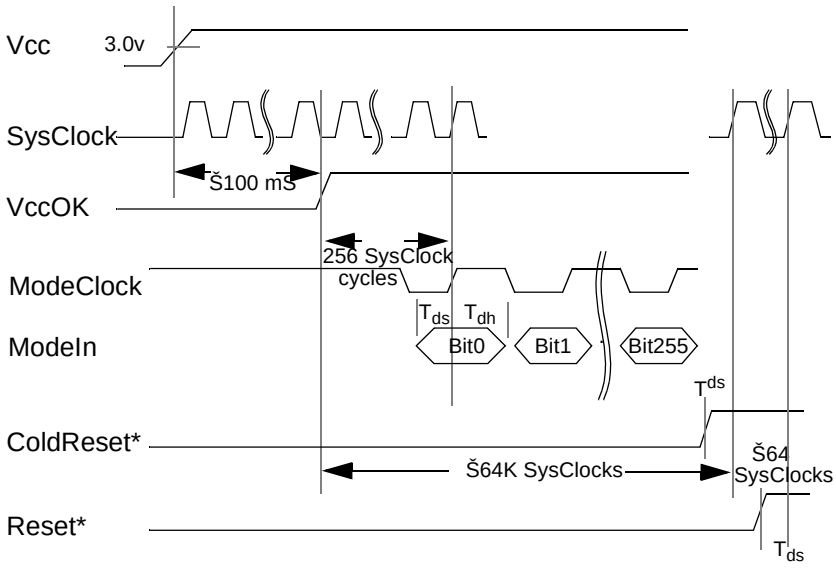


Figure 10.1 Power-On Reset Timing Diagram

10.1.2 Cold Reset

A cold reset can begin anytime after the processor has read the initialization data stream, causing the processor to take a 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, the system deasserts **VccOK** for a minimum of at least 64 **SysClock** cycles before reassertion.

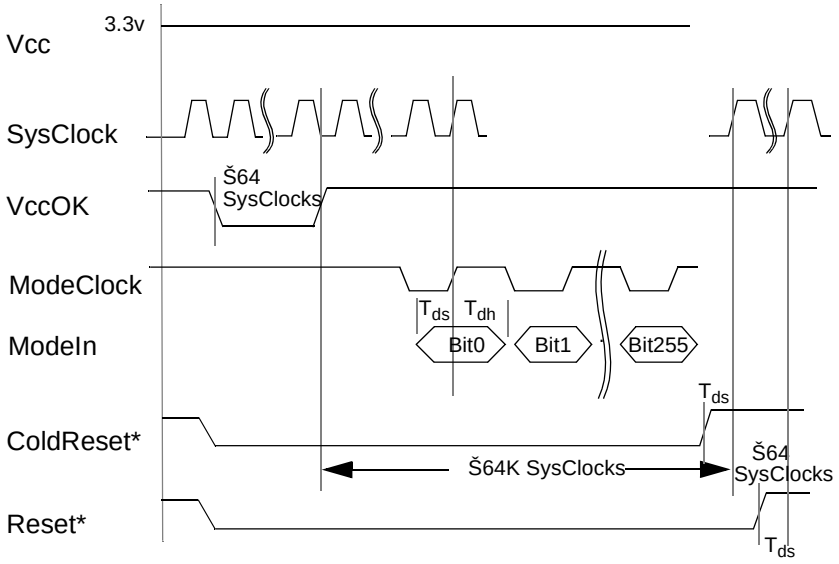


Figure 10.2 Cold Reset Timing Diagram

Figure 10.2 shows the cold reset timing diagram.

10.1.3 Warm Reset

To execute a warm reset, the system asserts the **Reset*** input synchronously with **SysClock**. **Reset*** must remain asserted for at least 64 **SysClock** cycles before being deasserted synchronously with **SysClock**. 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 take a Soft Reset exception.

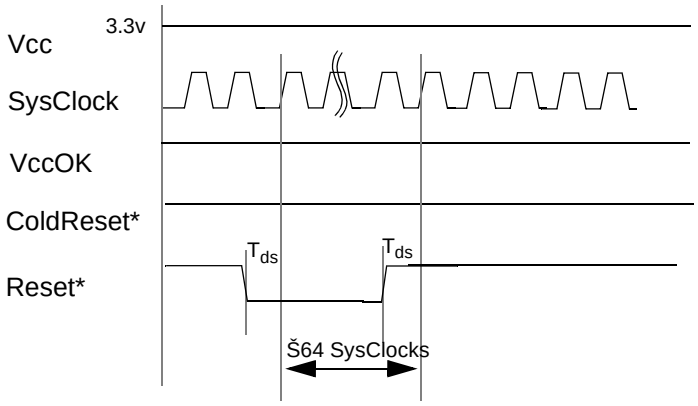


Figure 10.3 Warm Reset Timing Diagram

Figure 10.3 shows the warm reset timing diagram.

10.1.4 Processor Reset State

After a power-on reset or cold 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.

10.2 Initialization Sequence

The boot-mode initialization sequence begins immediately after the system asserts **VccOk**. As the processor reads the serial stream of 256 bits through the **ModeIn** pin, the boot-mode are used to initialize all fundamental processor modes.

The initialization sequence is listed below.

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 256 **SysClock** cycles after **VccOK** is asserted.
3. Each bit of the initialization stream is presented at the **ModeIn** pin after each rising edge of **ModeClock**. The processor samples 256 initialization bits from the **ModeIn** input.

10.3 Boot-Mode Settings

The following rules apply to the boot-mode settings:

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

Table 10.2: shows the boot mode settings.

Table 10.2: Boot Mode Settings

Bit	Value	RM7000 Mode Setting
0	0	Reserved: must be zero
4:1	64-bit Write-back data rate	
	0	DDDD
	1	DDxDDx
	2	DDxxDDxx
	3	DxDxDxDx
	4	DDxxxDDxxx
	5	DDxxxxDDxxxx
	6	DxxDxxDxxDxx
	7	DDxxxxxxDDxxxxxx
	8	DxxxDxxxDxxxDxxx
	9:15	Reserved
7:5	SysCkRatio: **SysClock** to **Pclock** Multiplier. See mode bit 20 below. Integer multipliers enabled / half multipliers enabled	
	0	Multiply by 2
	1	Multiply by 3
	2	Multiply by 4
	3	Multiply by 5 / Multply by 2.5
	4	Multiply by 6
	5	Multiply by 7 / Multiply by 3.5
	6	Multiply by 8
	7	Multiply by 9 / Multiply by 4.5

Bit	Value	RM7000 Mode Setting
8	EndBit: Specifies byte ordering. Logically ORed with the **BigEndian** signal.	
	0	Little-Endian ordering
	1	Big Endian ordering
10:9	Determines how non-block writes are handled.	
	0	R4000 compatible
	1	Reserved
	2	Pipelined non-block writes
	3	Non-block write-reissue
11	TmrIntEn: Disables Timer Interrupt on Int*[5]	
	0	Timer Interrupt Enabled, Int*[5} disabled
	1	Timer Interrupt Disabled, Int*[5] enabled
12	Tertiary Cache Interface	
	0	Disable tertiary cache
	1	Enable tertiary cache
14:13	DrvOut: Output driver slew rate control	
	10	100% (fastest)
	11	83%
	00	67%
	01	50% (slowest)
15	Tertiary cache Pipelined Burst RAM type:	
	0	Dual-Cycle Deselect
	1	Single-Cycle Deselect
17:16	XX	Tertiary Cache Size:System designer is free to choose encoding as the CPU does not use these bits. If the tertiary cache is not implemented, these bits can be used for processor identification in a multiple-CPU system.
19:18	0	Reserved: Must be zero
20	SysClock multipliers:	
	0	Integer multipliers (2, 3, 4, 5, 6, 7, 8, 9)
	1	Half integer multipliers (2.5, 3.5,4.5)
23:21	0	Reserved: Must be zero
24	Joint TLB Size:	
	0	48 Pairs of even/odd entries allowing 96 pages to be mapped.
	1	64 Pairs of even/odd entries allowing 128 pages to be mapped.
25	Enable integrated secondary cache:	
	0	Disable cache
	1	Enable cache
26	Enable two outstanding reads with out-of-order return:	
	0	Disable
	1	Enable
255:27	0	Reserved: Must be zero

10.4 CP0 Registers used in initialization and configuration

While most of the CP0 registers are used for memory management and interrupts and exception processing, there are four registers that can provide the use information on and control over the configuration of the RM7000. These three registers are the *Processor ID (PRID)* register, the *Configuration* register, the *Status* register, and the *Info* register.

10.4.1 Processor Revision Identifier (PRId) Register (Set 0, Register 15)

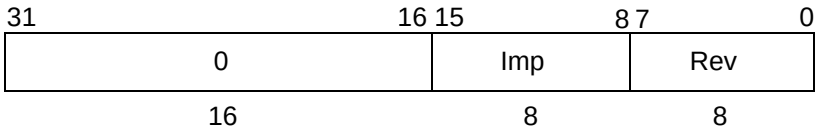


Figure 10.4 Processor Revision Identifier Register Format

The 32-bit, read-only *Processor Revision Identifier* (*PRId*) register contains information identifying the implementation and revision level of the CPU and CP0. Figure 10.4 shows the format of the *PRId* register; Table 10.3: describes the *PRId* register fields.

Table 10.3: PRId Register Fields

Field	Description
Imp	Implementation number = 0x27
Rev	Revision number
0	Reserved. Must be written as zeroes, and returns zeroes when read.

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 RM7000 processor is 0x27. The content of the high-order halfword (bits 31:16) of the register is 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.

10.4.2 Config Register (Set 0, Register 16)

The *Config* register specifies various configuration options which can be selected.

Some configuration options, as defined by *Config* bits 31:13,11:3 are read from the initialization mode stream by the hardware during reset and are included in the *Config* register as read-only status bits for the software to access. Other configuration options are read/write (*Config* register bits 12, 3:0) and controlled by software; on reset these fields are undefined.

Certain configurations have restrictions. The *Config* register should be initialized by software before caches are used. Caches should be written back to memory before line sizes are changed, and caches should be reinitialized after any change is made.

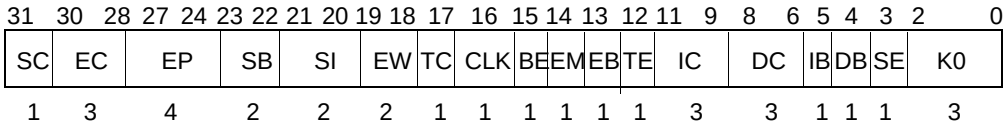


Figure 10.5 Config Register Format

Figure 10.5 shows the format of the *Config* register; Table 10.4: describes the *Config* register fields.

Table 10.4: Config Register Fields

Bit(s)	Field	Description
31	SC	Secondary Cache Present: Value loaded from Mode bit[25]. 0: Secondary Cache present 1: Secondary Cache not present.
30:28	EC	System clock ratio: Value loaded from Mode bits[7:5] 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 or 2.5 4: processor clock frequency divided by 6 5: processor clock frequency divided by 7 or 3.5 6: processor clock frequency divided by 8 7: processor clock frequency divided by 9 or 4.5
27:24	EP	Transmit data pattern (pattern for write-back data): Value loaded from Mode bits [4:1] 0: DDDD Doubleword every cycle 1: DDxDDx 2 Doublewords every 3 cycles 2: DDxxDDxx 2 Doublewords every 4 cycles 3: DxDxDxDx 2 Doublewords every 4 cycles 4: DDxxxDDxxx 2 Doublewords every 5 cycles 5: DDxxxxDDxxxx 2 Doublewords every 6 cycles 6: DxxDxxDxxDxx 2 Doublewords every 6 cycles 7: DDxxxxxxDDxxxxxx2 Doublewords every 8 cycles 8: DxxxDxxxDxxxDxxx2 Doublewords every 8 cycles
23-22	SB	Secondary Cache block size. (fixed at 8 words). 0: 4 words 1: 8 words 2: 16 words 3: 32 words
21:20	SI	System Configuration Identifier: Value loaded from Mode bits[17:16]. System designer is free to choose encoding as the CPU is not affected by these bits.
19:18	EW	SysAD bus width. Set to 64-bit by default. 00: 64-bit 01,10,11: Reserved
17	TC	Tertiary Cache Present: Value loaded from Mode bit 12. 0: Tertiary Cache present 1: Tertiary Cache not present.
16	CLK	Identifies the value of mode bit 20, set during the boot time initialization process. A value of 1 indicates that the half-integer clock multiplier is enabled.

Bit(s)	Field	Description
15	BE	Big Endian Mode: Value loaded from the OR function of Mode bit 8 and the BigEndian pin. 0: Little Endian 1: Big Endian
14	EM	ECC mode enable. Fixed to parity by default. 0: ECC mode 1: Parity mode
13	EB	Block ordering. Fixed to sub-block by default. 0: Sequential 1: Sub-block
12	TE	Tertiary Cache Enable: Setting this bit enables the external tertiary cache controller. (software writable)
11:9	IC	Primary Instruction cache Size (I-cache size = 2^{12+IC} bytes). Fixed to 16 Kbytes.
8:6	DC	Primary Data cache Size (D-cache size = 2^{12+DC} bytes). Fixed to 16 Kbytes.
5	IB	Primary Instruction cache line size. Fixed to 32 bytes. 0: 16 bytes 1: 32 bytes
4	DB	Primary Data cache line size. Fixed to 32 bytes. 0: 16 bytes 1: 32 bytes
3	SE	Secondary Cache Enable: Setting this bit enables the secondary cache. (software writable)
2:0	K0	*kseg0* coherency algorithm (software writable)

10.4.3 Status Register (Set 0, 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. The following list describes the more important *Status* register fields.

- 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]**. (***IM7*** enables the Timer Interrrupt on ***IP7*** unless mode bit 11 is set to enable **Int*5** on ***IP7***.) **Int*[9:6]** are enabled in the *IntControl* register. Please refer to Section 14, “Interrupts”, to use **Int*[9:6]** and to enable the prioritized, vectorized interrupt capability.
- The 3-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.

10.4.3.1 Status Register Format

Figure 10.6 shows the format of the *Status* register. Table 10.5: describes the *Status* register fields. Figure 10.7 and Table 10.6: provide additional information on the ***Diagnostic Status*** (***DS***) field. All bits in the ***DS*** field except ***TS*** are readable and writable.

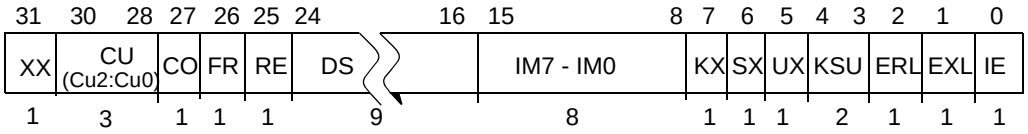


Figure 10.6 Status Register

Table 10.5: Status Register Fields

Bit(s)	Field	Description
31	XX	Enables execution of MIPS IV instructions in user-mode 1: MIPS IV instructions usable 0: MIPS IV instructions unusable
30:28	CU	Controls the usability of each of the three coprocessor unit numbers. CP0 is always usable when in Kernel mode, regardless of the setting of the CU_0 bit. 1: usable 0: unusable
27	CO	Setting this bit enables use of the Hit_Writeback and Hit-Writeback_Invalidate CACHE instruction while in user mode. Clearing this bit disables use of these instructions while in user mode.
26	FR	Enables additional floating-point registers 0: 16 registers 1: 32 registers
25	RE	*Reverse-Endian* bit, valid in User mode.
24:16	DS	*Diagnostic Status* field (see Figure 10.7).
15:8	IM[7:0]	*Interrupt Mask* (Lower): controls the enabling of eight of the external, internal, and software interrupts. An interrupt is taken if interrupts are enabled, and the corresponding bits are set in both the *Interrupt Mask* field of the *Status* register and the *Interrupt Pending* field of the *Cause* register. IM[7:2] correspond to Int*5 through Int*0. IM7 also enables the Timer Interrrupt if mode bit 11 is 0. IM[1:0] correspond to SWINT1 and SWINT0. (Int*[9:6] are enabled in the IntControl register). 0: disabled 1: enabled
7	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
6	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
5	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
4:3	KSU	Mode bits 10: User 01: Supervisor 00: Kernel

Bit(s)	Field	Description
2	ERL	Error Level; set by the processor when Reset, Soft Reset, NMI, or Cache Error exception are taken. 0: normal 1: error When ERL is set: Interrupts are disabled. The ERET instruction uses the return address held in ErrorEPC instead of EPC. Kuseg and xkuseg are treated as unmapped and uncached regions.This allows main memory to be accessed in the presence of cache errors.
1	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 When EXL is set: Interrupts are disabled. TLB refill exceptions will use the general exception vector instead of the TLB refill vector. EPC is not updated if another exception is taken.
0	IE	Interrupt Enable 0: disable interrupts 1: enables interrupts

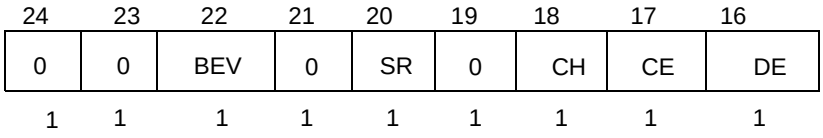


Figure 10.7 Status Register Diagnostic Status Field

Table 10.6: Status Register Diagnostic Status Bits

Bit(s)	Field	Description
22	BEV	Controls the location of TLB refill and general exception vectors. 0: normal 1: bootstrap
20	SR	1: Indicates that a Soft Reset or NMI has occurred.
18	CH	Hit (tag match and valid state) or miss indication for last CACHE Hit Invalidate, Hit Write Back Invalidate, or Hit Write Back for the secondary cache. 0: miss 1: hit
17	CE	Contents of the ECC register set or modify the check bits of the caches when CE = 1; see description of the *ECC* register.
16	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.

10.4.4 Info Register (Set 0, Register 7)

The *Info* register specifies various configuration options which have been selected at boot-time as well as the particulars of the on-chip cache confiration. It also contains a global interrupt enable/disable bit..

Some configuration options, as defined by *Info register* bits 28:23 are read from the initialization mode stream by the hardware during reset and are included in the *Info* register as read-only status bits for the software to access. The only option writable by softe is the global interrupt enable.disable bit (bit 0, GIE).

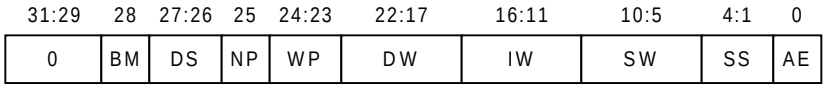


Figure 10.8 Info Register Format

Figure 10.5 shows the format of the *Info* register; Table 10.4: describes the *Info* register fields.

Table 10.7: Info Register Fields

Bits	Field	Description
28	BM	Burst Mode: Read from modebit 15 0 : pipelined Scache RAMS (dual cycle deselect) 1 : burst mode Scache RAMS (single cycle deselect)
27..26	DS	Drive Strength: Drive strength of pad output drivers. Read from modebits 14:13 00 : 67% drive strength 01 : 50% drive strength 10 : 100% drive strength 11 : 83% drive strength
25	NP	Non-pendant bus mode: Read from modebit 26 0 : overlapping reads disabled 1 : overlapping reads enabled
24..23	WP	Write Protocol: Read from modebits 10:9 00 : R4000 compatible 01 : Reserved 10 : Pipelined writes 11 : Write re-issue
22..17	DW	Primary Data Cache Sets: 000100 : 4 sets all other patterns : reserved
16..11	IW	Primary Instruction Cache Sets: 000100 : 4 sets all other patterns : reserved
10..5	SW	Secondary Cache Sets: 000100 : 4 sets all other patterns : reserved
4..1	SS	Secondary Cache Size: 0010 : 256 Kbytes all other patterns : reserved
0	AE	Atomic Enable: Atomic Interrupt Enable/Disable. This bit allows atomic enable/disable of interrupts without a read-modify-write sequence to the SR.IE bit. Writing a one to this bit globally enables interrupts Writing a zero to this bit globally disables interrupts This bit returns a zero on a read.

Section 11 System Interface Transactions

There are two broad categories of transactions: *processor requests* and *external requests*. The processor issues a *processor requests* through the System interface to access an external resource. An external agent requesting access to a processor internal resource generates an *external request*. This section describes these requests along with the tertiary cache transactions.

11.1 Terms Used

The following terms are used in the next sections of this document and are clarified here for reference:

- An *external agent* is any logic device connected to the processor, over the System interface, that accepts processor 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 perform a processor or external request.
- *Syntax* refers to the precise definition of bit patterns on encoded buses, such as the command bus.

11.2 Processor Requests

The RM7000 issues a *processor request* through the System interface, to access an external resource. 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.

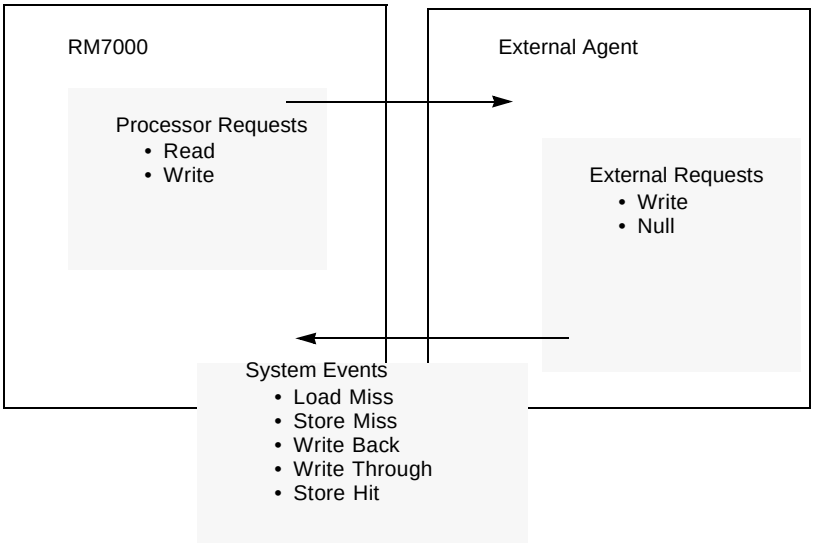


Figure 11.1 Requests and System Events

An external agent requesting access to a processor internal resource generates an *external request* through the System interface. System events and processor and external request cycles are shown in Figure 11.1.

11.2.1 Rules for Processor Requests

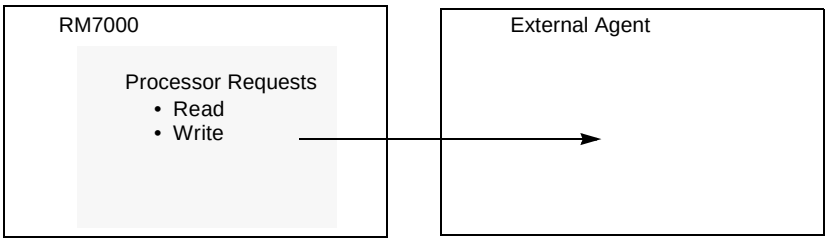


Figure 11.2 Processor Requests to External Agent

As shown in Figure 11.2, a processor request may be either a read or a write request.

A read request asks for a block, doubleword, partial doubleword, word, or partial word of data either from main memory or from another system resource.

A 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.

By default, the RM7000 uses a pendant system interface that is compatible with the MIPS R5000. This means the processor is only allowed to have one request pending at any time and that request must be serviced by an external device before the RM7000 issues another request. The RM7000 can issue read and write requests to an external device, whereas an external device can issue null and write requests to the RM7000.

The RM7000 also supports a non-pendant system interface that is enabled by the mode stream during initialization. This enhanced mode allows the CPU to place a second read request on the system interface before a previous read request has been serviced. In addition, the RM7000 allows the external device to complete the second read request before the first. The order of request completion is communicated via the *RspSwp** signal.

For processor reads the RM7000 asserts ValidOut* and simultaneously drives the address and read command on the SysAD and SysCmd buses. If the system interface has RdRdy* asserted, then the processor tristates its drivers and releases the system interface to slave state by asserting Release* for one cycle. The external device can then begin sending data to the RM7000.

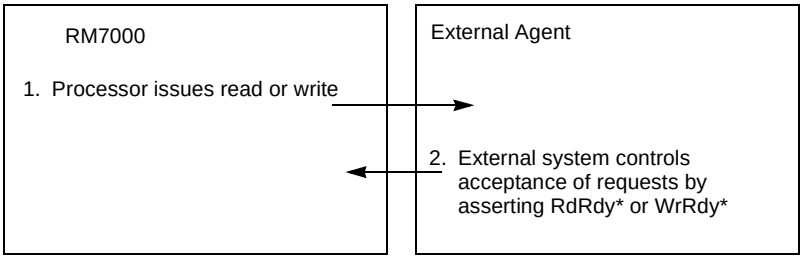


Figure 11.3 Processor Request Flow Control

The input signals **RdRdy*** and **WrRdy*** 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. Processor request flow control is shown in Figure 11.3.

11.2.2 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 issues if a read address cycle is driven on the **SysAD** bus and the signal **RdRdy*** was asserted two cycles earlier.

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 two or more cycles before the issue cycle.

11.2.3 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 processor will not issue a write request if a processor read request has been issued and the requested data has not yet been received.

In the original R4000- compatibility write protocol, a processor write request issues if a write address cycle is driven out on the **SysAD** bus and the signal **WrRdy*** was asserted two cycles earlier.

Like the R4600, R4700, and R5000, the RM7000 processor support two enhancements to the original R4000 write protocol: *Write Reissue* and *Pipelined Writes*. In *Write Reissue* protocol, a write rate of one write every two bus cycles can be achieved. In *Write-Reissue* protocol, a write issues if **WrRdy*** is asserted two cycles previously and is also asserted during the issue cycle. If it is not still asserted then the last write will reissue. Pipelined writes have the same two bus cycle write repeat rate, but can issue one additional write following the deassertion of **WrRdy***. These modes are selected by mode bits 9 and 10 at power-up.

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 two or more cycles.

11.3 External Requests

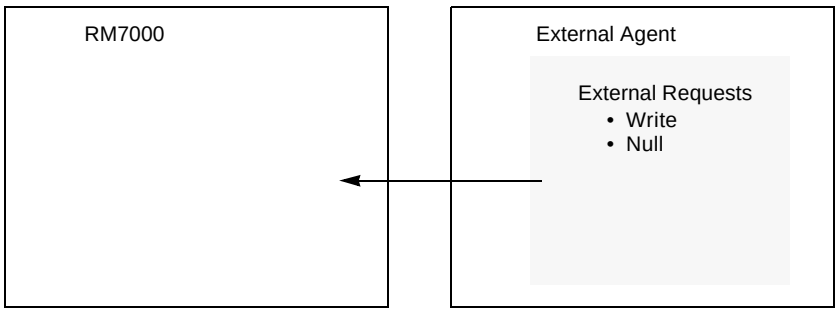


Figure 11.4 External Requests to Processor

External requests include write and null requests, as shown in Figure 11.4. This section also includes a description of a read response, a special case of an external request.

A *Write request* is executed by an external device when it wishes to update the processors internal interrupt register.

A *Null request* is executed when the external agent wishes the processor to reassert ownership of the system interface (the external agent wants the system interface to transition from slave state to master state). Typically, a null request is executed after an external device has acquired control of the processor interface via the **ExtRqst*** signal and has completed a transaction on the system bus that does not involve the processor. Typically this transaction would be a DMA read or write from the I/O system.

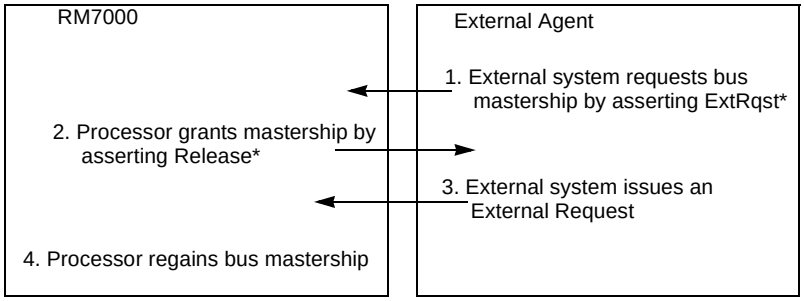


Figure 11.5 External Request Arbitration

The processor controls the flow of external requests through the arbitration signals **ExtRqst*** and **Release*** as shown in Figure 11.5. 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. If **Release*** is asserted as part of an uncompelled change to slave state during a processor read request, and the tertiary cache is enabled, the tertiary cache access must be resolved and be a miss. Otherwise the system interface returns to master state.

Mastership of the System interface is always returned to the processor by the external agent after an external request is issued. The processor will 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 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*** for one cycle. The processor signals that it is ready to accept an external request based on the criteria listed below.

- The processor completes any request in progress.
- While waiting for the assertion of **RdRdy*** to issue a processor read request, the processor will accept an external request if that 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 will accept an external request provided that request is delivered to the processor one or more cycles before **WrRdy*** is asserted.
- If the processor is 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.

The RM7000 processor implements two additional arbitration signals, **PRqst*** and **PAck***, to support multiple outstanding read requests. If the processor encounters a primary or secondary cache miss while a read request is already pending, it can request immediate control of the SysAD bus in order to issue a second tertiary cache/main memory read. The processor asserts **PRqst***, indicating to the external agent that it requires control of the bus. The external agent relinquishes control of the bus by asserting **PAck*** for one cycle. A second read request is issued and the processor asserts **Release*** to return mastership of the bus to the external agent.

11.3.1 External Write Request

When an external agent issues a write request, the internal interrupt register 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 Interrupt register.

11.3.2 Read Response

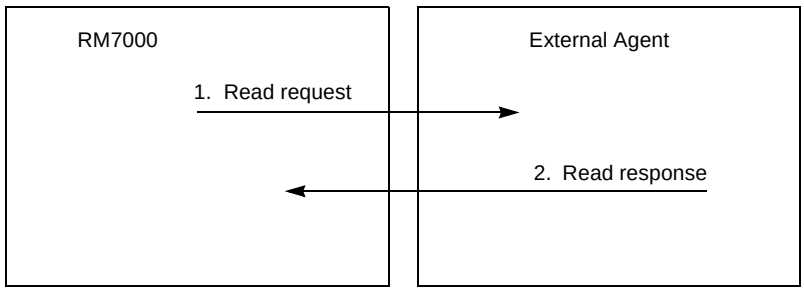


Figure 11.6 External Agent Read Response to Processor

A *read response* returns data in response to a processor read request, as shown in Figure 11.6. 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, a read response is handled separately from all other external requests, hence it is simply called a *read response*.

The data identifier associated with the response data can signal that the returned data is erroneous, causing the processor to take a bus error.

11.4 Tertiary Cache Transactions

The tertiary cache is an external third level cache for the processor controlled jointly by the processor and the external agent. Figure 11.7 illustrates a processor request to the tertiary cache and external agent.

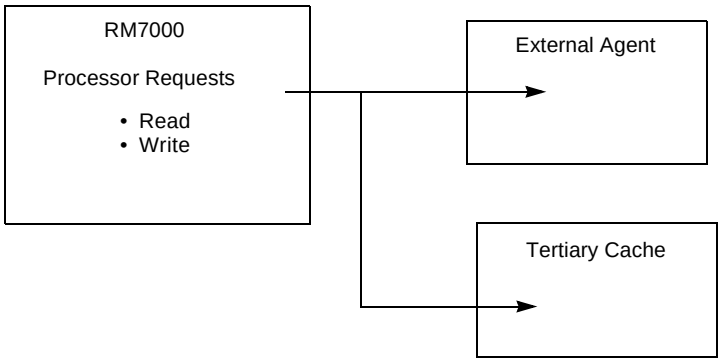


Figure 11.7 Processor Requests to Tertiary Cache and External Agent

11.4.1 Tertiary Cache Probe, Invalidate, and Clear

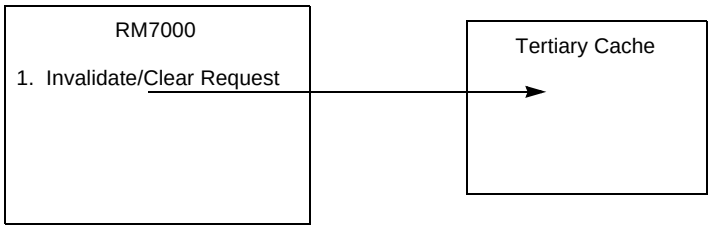


Figure 11.8 Tertiary Cache Invalidate and Clear

For invalidate, clear, and probe operations, the tertiary cache is controlled by the processor. The external agent is not involved in these operations. Tertiary cache invalidate, clear, and probe operations are not flow-controlled and proceed at the maximum data rate. Figure 11.8 and Figure 11.9 show the tertiary cache invalidate and tag probe operations.

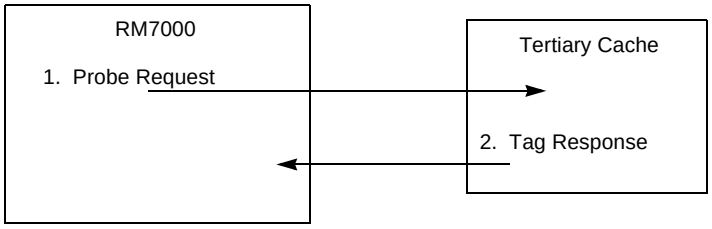


Figure 11.9 Tertiary Cache Tag Probe

11.4.2 Tertiary Cache Write

For a tertiary cache write-through, the processor issues a block write operation that is directed to both the tertiary cache and the external agent. Issuance of a tertiary cache write is controlled by the normal **WrRdy*** flow control mechanism. Tertiary cache write data transfers proceed at the data transfer rate specified in the Mode ROM for block writes. Figure 11.10 illustrates a tertiary cache write operation.

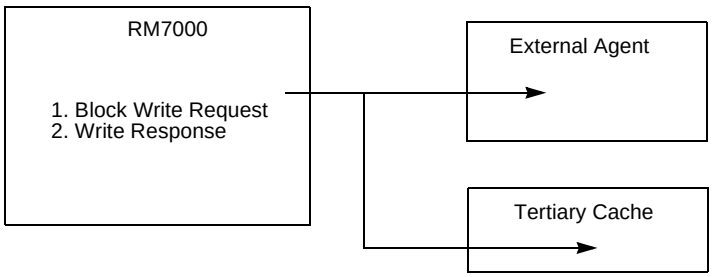


Figure 11.10 Tertiary Cache Write-Through

11.4.3 Tertiary Cache Read

For tertiary cache reads, the processor issues a speculative block read to both the tertiary cache and the external agent.

- If the block is present in the tertiary cache, the tertiary cache provides the read response and the block read to the external agent is aborted.
- If the block is not present in the tertiary cache, the tertiary cache read is aborted and the external agent provides the read response to both the tertiary cache and the processor.

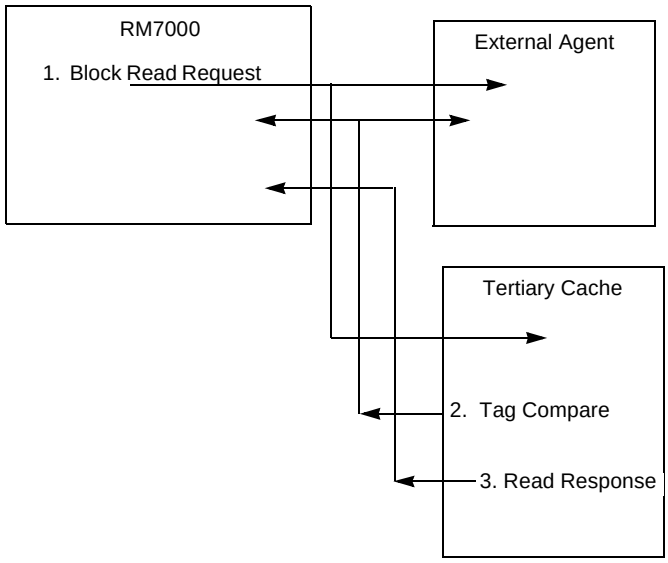


Figure 11.11 Tertiary Cache Read Hit

Figure 11.11 and Figure 11.12 show a tertiary cache read hit and miss respectively.

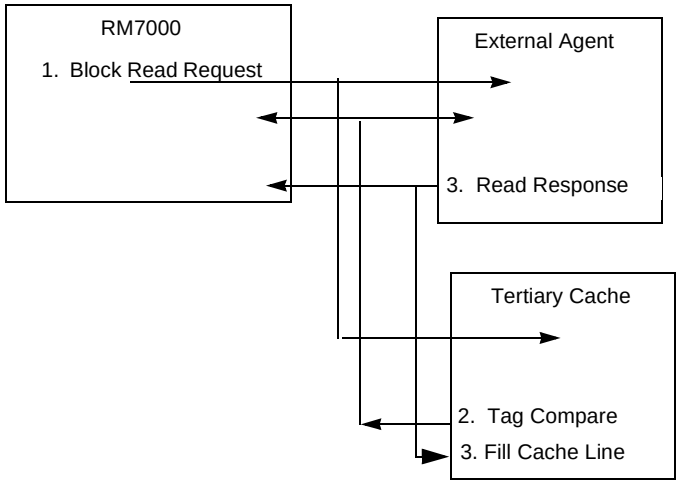


Figure 11.12 Tertiary Cache Read Miss

Issuance of the tertiary cache read is controlled by the normal **RdRdy*** flow control mechanism. Tertiary cache read responses always proceed at the maximum data transfer rate. External agent read responses to the tertiary cache proceed at the data transfer rate generated by the external agent.

11.5 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
- uncached instruction fetch

- load linked/store conditional

11.5.1 Load Miss

Before the processor can proceed after a load miss in the on-chip caches, 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 dirty cache line, the current cache line must be written back before the new line can be loaded in the primary cache.

The processor examines the coherency attribute in the TLB entry for the page that contains the requested cache line, and issues a noncoherent read request.

Table 11.1: shows the actions taken on a load miss on all on-chip caches.

Table 11.1: Load Miss to On-chip Caches

Page Attribute	State of Data Cache Line Being Replaced	
	Clean/Invalid	Dirty (W=1)
Noncoherent	NCBR	NCBR/W

NCBR Processor noncoherent block read request

NCBR/W Processor noncoherent block read request followed by processor block write request

The processor takes the following steps:

1. The processor issues a noncoherent block 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. If the tertiary cache is enabled and the page coherency attribute is write-back, the response data is also written into the tertiary cache.
3. Since the RM7000 supports two outstanding cache misses, the processor pipeline continues to run. The first doubleword of the data cache miss is received and used to resolve the miss. The remaining three doublewords are placed in the cache after all three doublewords have been received when the data cache is otherwise idle.

If the current cache line must be written back, the processor issues a block write request to save the dirty cache line in memory. If the tertiary cache is enabled and the page attribute is write-back, the write back data is also written into the tertiary cache.

11.5.2 Store Miss

When a processor store misses in the on-chip caches, the processor will request, from the external agent, the cache line that contains the target location of the store for pages that are being maintained with the ‘bypass’, ‘write-back’ or ‘write-through with write-allocate’ coherency attribute.

The processor then executes one of the following requests:

- If the coherency attribute is noncoherent write-back, or ‘write-through with write-allocate’, a noncoherent block read request is issued.
- If the coherency attribute is ‘write-through without write-allocate’, a non-block write request is issued.

Table 11.2: shows the actions taken on a store miss to all on-chip caches.

Table 11.2: Store Miss to On-chip Caches

Page Attribute	State of Data Cache Line Being Replaced	
	Clean/Invalid	Dirty (W=1)
Noncoherent-write-back or write-through with write-allocate	NCBR	NCBR/W
Write-through with no-write-allocate	NCW	NA

NCBR Processor noncoherent block read request

NCBR/W Processor noncoherent block read request followed by processor block write request

NCW Processor noncoherent write request

If the coherency attribute is write-back, or ‘write-through with write-allocate’, the processor issues a noncoherent block read request for the cache line that contains the desired data. The pipeline continues to run while the external agent is retrieving the data. If the current cache line must be written back, the processor issues a write request for the current cache line.

If the page coherency attribute is write-through, the processor issues a non-block write request.

For a ‘write-through without write-allocate’ store miss, the processor issues a non-block write request only.

11.5.3 Store Hit

The action on the system bus is determined by whether the line is write-back or write-through. For lines with a write-back policy, a store hit does not cause any processor request on the bus. For lines with a write-through policy, the store generates a processor non-block write request for the store data.

11.5.4 Uncached Loads or Stores

When the processor performs an uncached load, it issues a noncoherent doubleword, partial doubleword, word, or partial word read request. On an uncached store the processor issues a doubleword, partial doubleword, word, or partial word write request. All uncached writes by the processor are buffered from the system interface by a 4-entry write buffer. The uncached write requests are sent to the system bus before any waiting processor read requests. Uncached write requests are given the highest priority to the system bus in order to decrease the chances of any memory read-write ordering problems. Once the processor has started an uncached write request on the system bus, it will issue all pending write requests to the system bus before resuming any processor read requests. Uncached loads and stores do not affect the tertiary cache.

11.5.5 Uncached Instruction Fetch

Normally the Boot ROM and I/O devices are located in an uncached address region.

The RM7000 accesses Boot ROM in this space with two 32-bit instruction fetches as well. In little endian mode, the first instruction (even word) is from **SysAD[31:0]** with **SysAD[63:32]** being ignored. The second instruction (odd word) is from **SysAD[63:32]** with **SysAD[31:0]** being ignored.

In big endian mode, the first instruction (even word) is from **SysAD[63:32]** with **SysAD[31:0]** being ignored while the second instruction (odd word) is from **SysAD[31:0]** with **SysAD[63:32]** being ignored.

After both instructions have been fetched they are issued to the pipeline.

By having the RM7000 ignore 32-bits of the **SysAD** bus during an uncached instruction fetch the system designer is able to put each instruction onto both halves of the **SysAD** bus. This makes for a simpler Boot ROM interface.

11.5.6 Load Linked/Store Conditional Operation

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.

Section 12 System Interface Protocol

The following sections contain a cycle-by-cycle description of the system interface protocol.

12.1 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 of addresses and data from the processor is indicates by the assertion of the **ValidOut*** signal. Validity of addresses and data from the external agent is indicated by the assertion of the **ValidIn*** signal. Validity of data from the tertiary cache is determined by the state of the pipelined **TcDCE*** and **TcCWE*** signals from the processor and the **TcDOE*** signal from the external agent.

The **SysCmd** bus identifies the contents of the **SysAD** bus during any cycle in which it is valid from the processor or the external agent. 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 **SysCmd8** = 0. The remainder of the **SysCmd** bus, **SysCmd(7:0)**, contains the encoded system interface command.
- During data cycles **SysCmd8** = 1, the remainder of the **SysCmd** bus, **SysCmd(7:0)**, contains an encoded data identifier. There is no **SysCmd** value associated with a tertiary cache read response.

12.2 Issue Cycles

There are two types of processor request 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 for a processor write request.

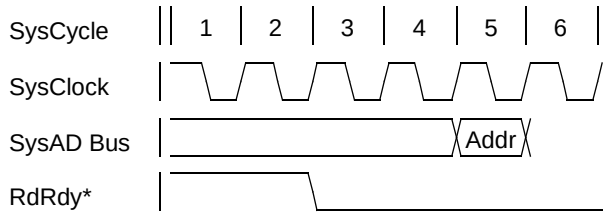


Figure 12.1 State of RdRdy* Signal for Read Requests

As shown in Figure 12.1, **RdRdy*** must be asserted two cycles prior to an address cycle for a processor read request in order to define that address cycle as the issue cycle.

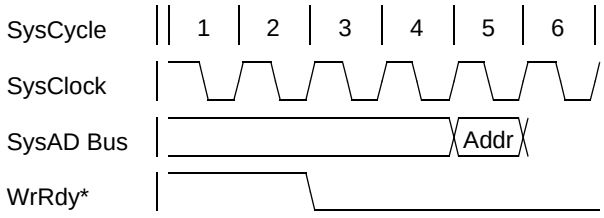


Figure 12.2 State of WrRdy* Signal for Write Requests

As shown in Figure 12.2, **WrRdy*** must be asserted two cycles prior to an address cycle for a processor write request in order to define that address cycle as the issue cycle.

The processor will repeat the address cycle for a request until the conditions for the issue cycle is 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 what action the processor takes. The processor can 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.

12.3 Handshake Signals

The RM7000 processor manages the flow of requests through the following ten control signals:

- **RdRdy*** and **WrRdy*** are used by the external agent to indicate when it can accept a new read (**RdRdy***) or write (**WrRdy***) transaction.
- **ExtRqst*** and **Release*** are used to transfer control of the **SysAD** and **SysCmd** buses. **ExtRqst*** is used by an external agent to indicate a need to acquire mastership of the interface. **Release*** is asserted by the processor when it transfers the mastership of the system interface to the external agent. For tertiary cache reads, assertion of **Release*** to the external agent is speculative, and is aborted if there is a hit in the tertiary cache.
- The RM7000 processor asserts **ValidOut*** and the external agent asserts **ValidIn*** to indicate valid command/data on the **SysCmd**/**SysAD** buses.
- The RM7000 processor supports two outstanding read transactions and allows data to be returned out-of-order from the original issue sequence. If a read is in progress and the processor needs to issue a second read request, it can request immediate control of the bus by asserting the **PRqst*** signal. The external agent responds by asserting **PAck*** (for one cycle) to indicate that it is relinquishing control of the bus. The processor asserts **ValidOut*** and drives the address for the second read request onto the bus, then asserts **Release*** to return control of the bus to the external agent.
- When two read transactions are outstanding, data for these reads can be returned out-of-order. For example, the processor issues read A, then some time later issues read B while read A is in progress. If the data from read B becomes available before the data from read A, the external agent can return the data for read B and assert the **RspSwap*** signal (for one cycle) to inform the RM7000 that data is being returned out-of-order. The assertion of **RspSwap*** to inform the processor that read data is being returned out-of-order must occur a minimum of two cycles before the first read data cycle.
- During the address cycle of a read request, **RdType** indicates whether the read request is an instruction read or a data read.

12.4 System Interface Operation

Processor inputs are fed directly to input registers that latch these input signals with the rising edge of **SysClock**. This allows the system interface to run at the highest possible clock frequency.

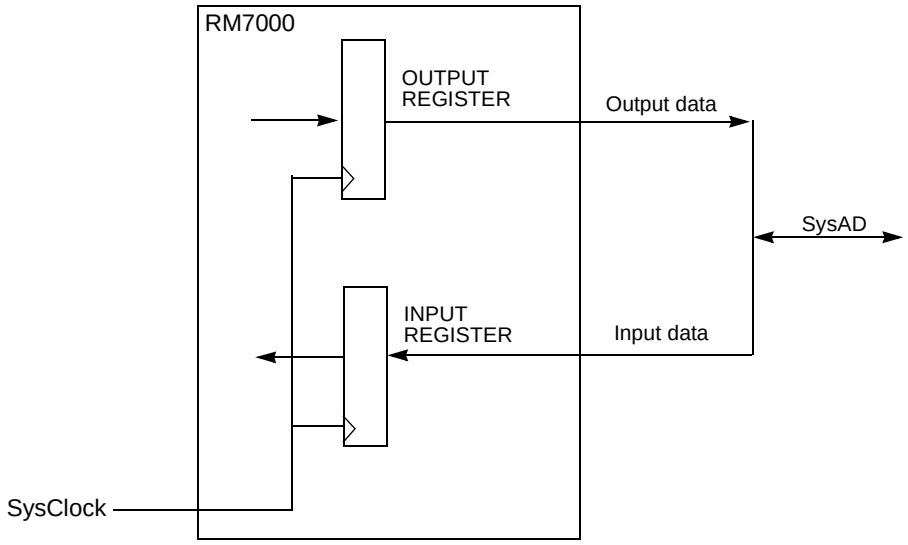


Figure 12.3 System Interface Register-to-Register Operation

Figure 12.3 shows how 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 **SysClock.**

12.4.1 Master and Slave States

When the RM7000 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*. When the tertiary cache is driving the **SysAD** and **SysADC** buses, the system interface is in tcache state.

In master state, the processor asserts the signal **ValidOut*** whenever the **SysAD** and **SysCmd** buses are valid.

In slave state, the external agent asserts the signal **ValidIn*** whenever the **SysAD** and **SysCmd** buses are valid. The tertiary cache drives the **SysAD** and **SysADC** buses in response to the **TcDCE***, **TcCWE***, and **TcDOE*** signals.

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

- The external agent requests and is granted mastership of the system interface (external arbitration).
- The processor issues a read request.

12.4.2 External Arbitration

The system interface must be in slave state for the external agent to issue an external request through the system interface. 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 accept an external request, it releases the system interface from master to slave state by asserting **Release*** for one cycle.
3. The system interface returns to master state as soon as the issue of the external request is complete.

12.4.3 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 when a read request issues and an uncompelled change to slave state then occurs. This transition to slave state allows the external agent to return read response data without arbitrating for bus ownership.

If the tertiary cache is enabled and a tertiary cache hit occurs, the bus is returned to master state after response data is returned to the processor from the tertiary cache.

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. If the external agent issues some other type of external request while there is a pending read request, the processor performs another uncompelled change to slave state by asserting **Release*** for one cycle.

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 an external request without arbitrating for the system interface; that is, without asserting **ExtRqst***.

Table 12.1: lists the abbreviations and definitions for each of the buses that are used in the timing diagrams that follow.

Table 12.1: System Interface Requests

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 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

12.5 Processor Request Protocols

Processor request protocols described in this section include:

- read
- write

In the timing diagrams, the two closely spaced, wavy vertical lines, such as those shown in Figure 12.4, indicate one or more identical cycles which are not illustrated due to space constraints.

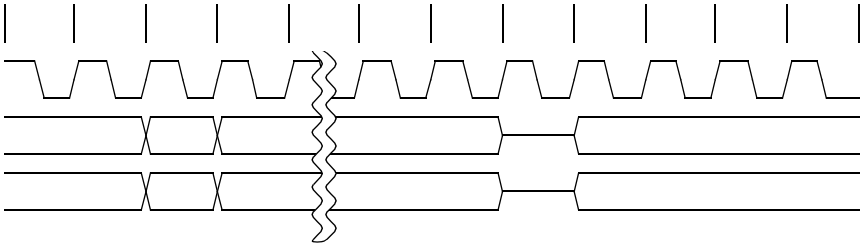


Figure 12.4 Symbol for Illustrated Cycles

12.5.1 Processor Read Request Protocol

The following sequence describes the protocol for doubleword, partial doubleword, word and partial word processor read requests. The tertiary cache block read request protocol is described later in this section.

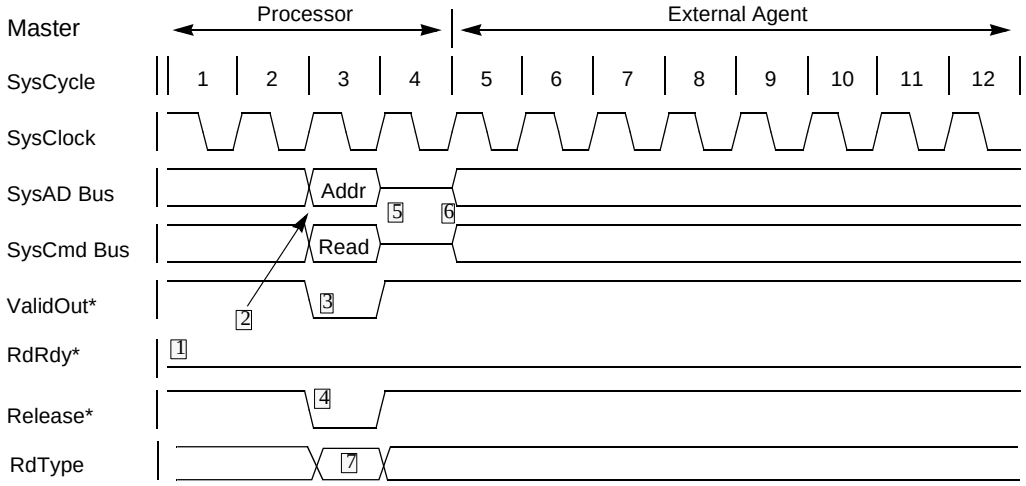


Figure 12.5 Processor Read Request Protocol

Figure 12.5 illustrates a processor read request, coupled with an uncompelled change to slave state, that occurs as the read request is issued.

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. The physical address is driven onto **SysAD[35:0]**, and virtual address bits [13:12] are driven onto **SysAD[57:56]**. All other bits are driven to zero.
3. At the same time, the processor asserts **ValidOut*** for one cycle, indicating valid data is present on the **SysCmd** and the **SysAD** buses.
4. The processor makes an uncompelled change to slave state during the issue cycle of the read request. The external agent need not assert the signal **ExtRqst*** for the purposes of returning a read response, but it must note and respond to the uncompelled change to slave state. An external request other than the read response may be performed before the read response begins once the system interface is in the slave state.
5. The processor stops driving the **SysCmd** and the **SysAD** buses one cycle after the assertion of **Release***.
6. The external agent begins driving the **SysCmd** and the **SysAD** buses two cycles after the assertion of **Release***.
7. RdType will be high if the address cycle is for an instruction read. RdType will be low if the address cycle is for a data read. RdType is NOT valid for all other cycles.

Once in slave state 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.

Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

When a read request is issued, the processor asserts **Release*** (for one cycle) to perform an uncompelled change to slave state. Once in the slave state the processor will always accept either a read response or an external request or allow an independent transmission on the **SysAD** bus. When an external request is accepted while a read is pending the processor will always perform an uncompelled changed to slave state following the external request. An uncompelled change to slave state is signaled by **Release*** being asserted for one cycle.

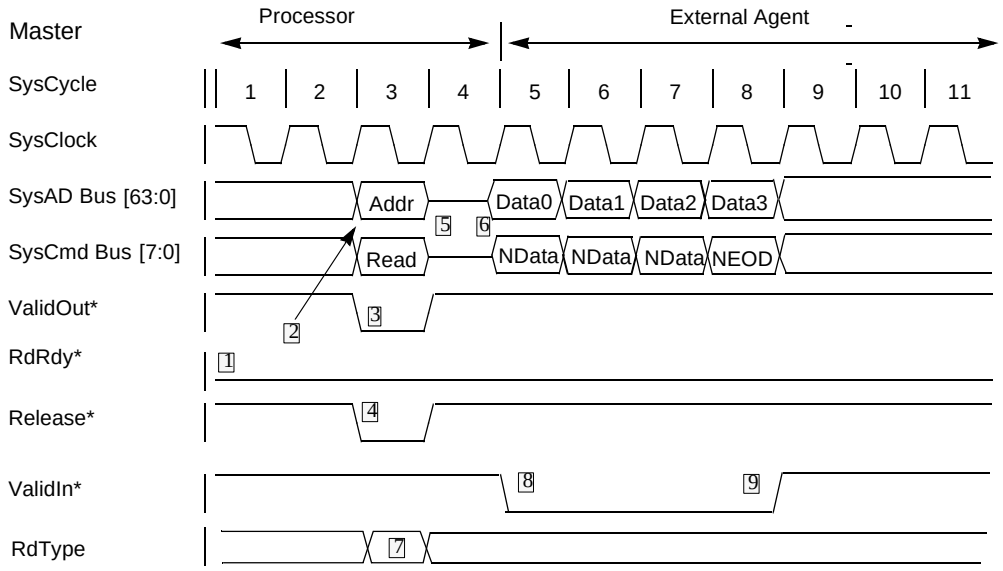


Figure 12.6 Processor Non-coherent, Non-Tertiary Cache Block Read Request

Figure 12.6 shows a Processor Block Read transaction on the 64-bit bus interface with zero wait states and no terrtiary cache access. After the bus has been granted to the external agent by the uncompelled change to slave state:

8. The external agent notifies the processor that valid data is on the bus by asserting **ValidIn*** and placing a noncoherent data identifier on the **SysCmd** bus.
9. The external agent notifies the processor that the transaction has been completed by placing a noncoherent-end-of-data identifier on the SysCmd pins and then deasserting **ValidIn***.

The external agent controls the data rate by controlling when **ValidIn*** is asserted. The processor only accepts data on the **SysAD** bus when **ValidIn*** is asserted and the **SysCmd** bus contains a data identifier.

12.5.2 Processor Write Request Protocol

Processor write requests are issued using one of three protocols.

- Doubleword, partial doubleword, word, or partial word writes use one of three non-block write request protocols. The protocol used is selected by mode bits 9 and 10 at power-up.
- Cache block writes use the block write request protocol.
- Tertiary cache block write protocol.

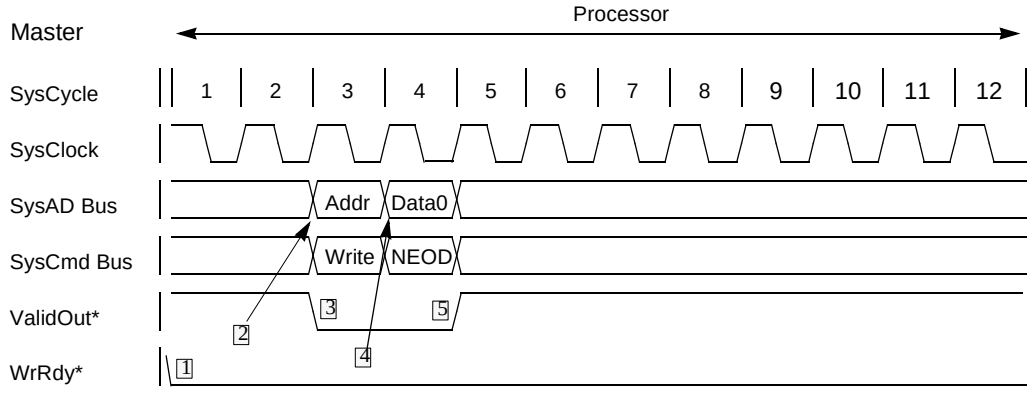


Figure 12.7 Processor Noncoherent Non-Block Write Request

Processor write requests are issued with the system interface in master state, as described below. Figure 12.7 shows a processor noncoherent non-block write request while Figure 12.8 shows a processor noncoherent block write request with no tertiary cache access. The following numbered steps are the same for both Figure 12.7 and Figure 12.8.

1. **WrRdy*** is asserted low, indicating the external agent is ready to accept a write request.
2. A processor non-block, or block, write request is issued by driving a write command on the **SysCmd** bus and a write address on the **SysAD** bus. The physical address is driven onto **SysAD[35:0]**, and virtual address bits [13:12] are driven onto **SysAD[57:56]**. All other bits are driven to zero.
3. The processor asserts **ValidOut***.
4. The processor drives a data identifier on the **SysCmd** bus and data on the **SysAD** bus.
5. The data identifier associated with the last data cycle will contain a last data cycle indication. At the end of the last data cycle, **ValidOut*** is deasserted.

Note: Timings for the **SysADC** *and* **SysCmdP** *buses are the same as those of the* **SysAD** *and* **SysCmd** *buses, respectively.*

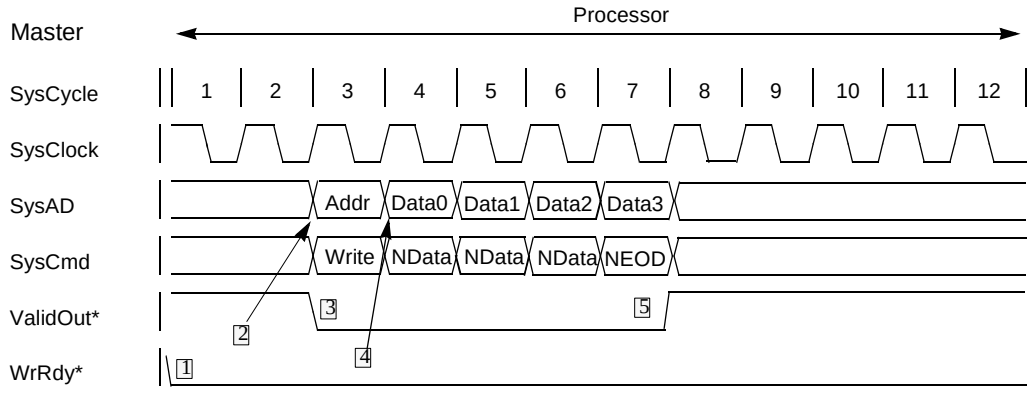


Figure 12.8 Processor Non-Coherent, Non-Tertiary Block Write Request

12.5.3 Processor Request Flow Control

The external agent uses **RdRdy*** to control the flow of processor read requests. Both block and non-block processor read requests are controlled in the same manner. Processor read requests are delayed by **RdRdy*** only at the Address cycle. The **RdRdy*** signal is not used during data cycles to delay a bus transaction.

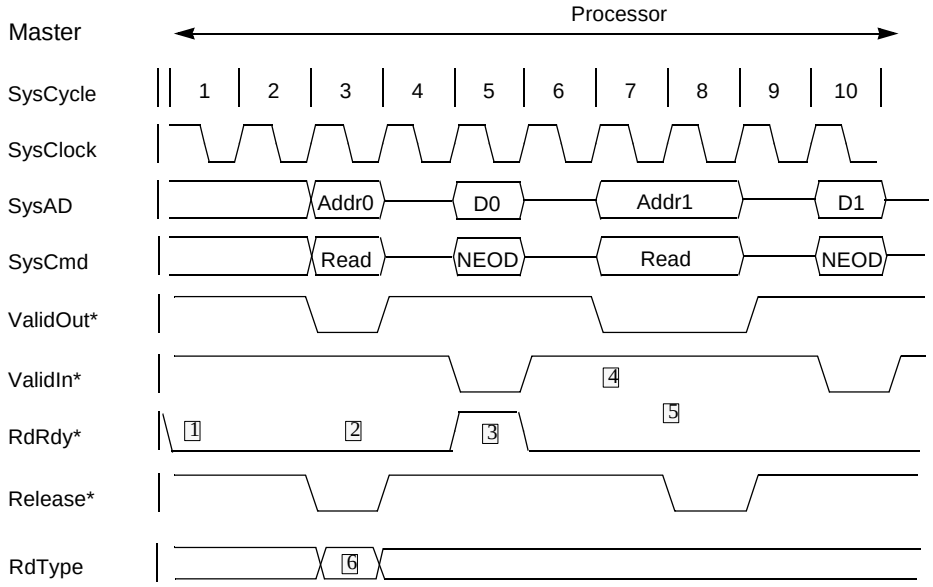


Figure 12.9 Processor Read Request Flow Control

Figure 12.9 illustrates this flow control, as described in the steps below.

1. The processor samples the **RdRdy*** signal to determine if the external agent is capable of accepting a read request.
2. Read request is issued to the external agent.
3. The external agent deasserts **RdRdy***, indicating it cannot accept additional read requests.
4. The read request issue is stalled because **RdRdy*** was negated two cycles earlier.
5. Read request is again issued to the external agent.
6. RdType will be high if the address cycle is for an instruction read. RdType will be low if the address cycle is for a data read. RdType is NOT valid for all other cycles.

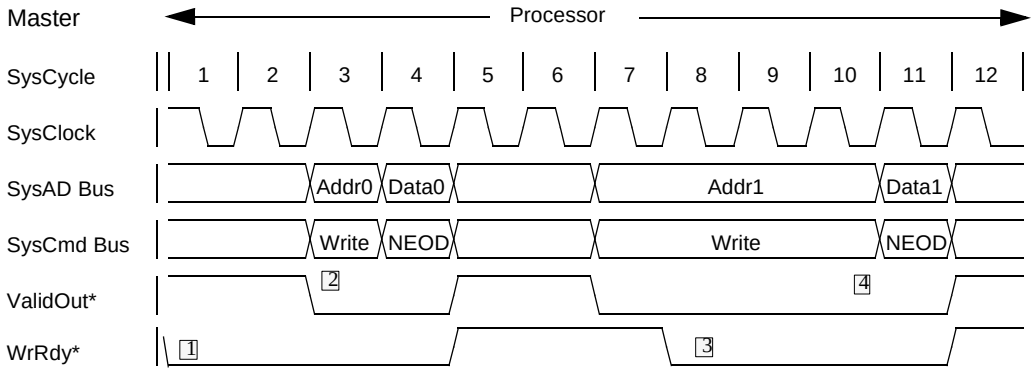


Figure 12.10 Processor Write Request Flow Control

Figure 12.10 illustrates two processor write requests in which the issue of the second is delayed for the assertion of **WrRdy***. Processor write requests are delayed by **WrRdy*** only at the address cycle. The **WrRdy*** signal is not used during data cycles to delay a bus transaction.

1. **WrRdy*** is asserted, indicating the external agent is ready to accept a write request.
2. The processor asserts **ValidOut***, a write command on the **SysCmd** bus, and a write address on the **SysAD** bus.
3. The second write request is delayed until the **WrRdy*** signal is again asserted.
4. A write request does not issue an address cycle is accomplished for the write request for which the signal **WrRdy*** was asserted two cycles earlier.

Note: Timings for the **SysADC** *and* **SysCmdP** *buses are the same as those of the* **SysAD** *and* **SysCmd** *buses, respectively.*

The processor write protocol requires that **WrRdy*** be asserted two cycles prior to the issue cycle of a write request. An external agent that negates **WrRdy*** immediately upon receiving a write that fills its buffer suspends any subsequent writes for four system cycles in R4000 non-block write-compatible mode. The processor always inserts at least two unused cycles after a write address/data pair in order to give the external agent time to suspend the next write.

Both block and non-block processor write requests are flow controlled by **WrRdy*** in the same manner.

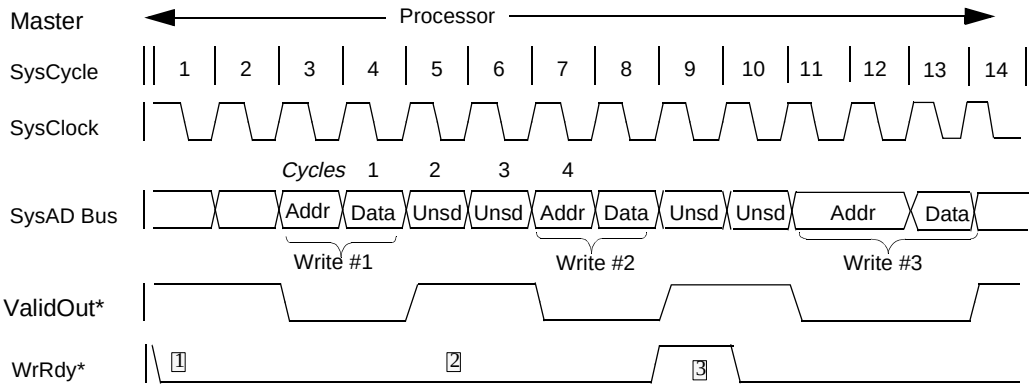


Figure 12.11 R4000-Compatible Back-to-Back Write Cycle Timing

Figure 12.11 shows back-to-back non-block write requests in R4000-compatible mode. This non-block write mode is selected with mode bits 9 and 10 set to zeros.

1. **WrRdy*** is asserted, indicating the processor can issue a write request.
2. **WrRdy*** remains asserted, indicating the external agent can accept another write request.
3. **WrRdy*** deasserts, indicating the external agent cannot accept another write request, stalling the issue of the next write request.

An address/data pair every four system cycles is not sufficiently high performance for all applications. For this reason, the RM7000 processor provides two write protocol modes that modify the R4000 back-to-back write protocol to allow an address/data pair every two cycles. These two protocols are as follows:

— *Write Reissue* allows **WrRdy*** to be negated during the address cycle and forces the write request to be re-issued.
— *Pipelined Writes* leave the sample point of **WrRdy*** unchanged and require that the external agent accept one more write than dictated by the R4000 compatible write protocol.

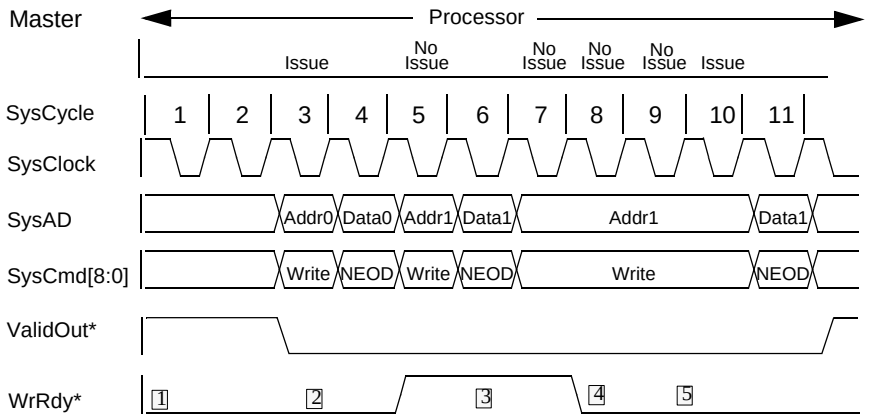


Figure 12.12 Write Reissue

The write re-issue protocol is shown in Figure 12.12. Writes are issued when **WrRdy*** is asserted both two cycles prior to an address cycle and during that address cycle.

If a processor block write transaction is delayed by **WrRdy*** being deasserted during its address cycle, then only the first data cycle will be presented on the bus before the address is presented on the bus again. This write protocol is selected when mode bits 9 and 10 are set to ones.

1. **WrRdy*** is asserted, indicating the external agent can accept a write request.
2. **WrRdy*** remains asserted as the write is issued, and the external agent is ready to accept another write request.
3. **WrRdy*** deasserts during the following address cycle. This write request is aborted and the address re-presented.
4. **WrRdy*** is asserted, indicating the external agent can accept a write request.
5. **WrRdy*** remains asserted for two or more cycles. The previously aborted write request is issued.

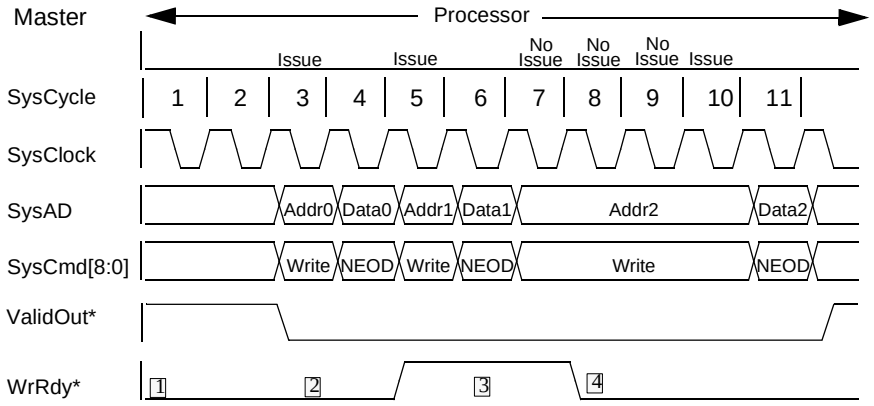


Figure 12.13 Pipelined Writes

The pipelined write protocol is shown in Figure 12.13. Writes are issued when **WrRdy*** is asserted two cycles before the address cycle and the external agent is required to accept one more write after **WrRdy*** is deasserted. This write protocol is selected when mode bit 9 is set to a zero and mode bit 10 is set to a one.

1. **WrRdy*** is asserted, indicating the external agent can accept a write request.
2. **WrRdy*** remains asserted as the write is issued, and the external agent is able to accept another write request.
3. **WrRdy*** is deasserted, indicating the external agent cannot accept another write request; it does, however, accept this write.
4. **WrRdy*** is asserted, indicating the external agent can accept a write request.

12.6 External Request Protocols

External requests can only be issued with the system interface in slave state. An external agent asserts **ExtRqst*** to arbitrate for the system interface, then waits for the processor to release the system interface to slave state by asserting **Release*** for one cycle 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—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.

This section describes the following external request protocols:

- null
- write
- read response

12.6.1 External Arbitration Protocol

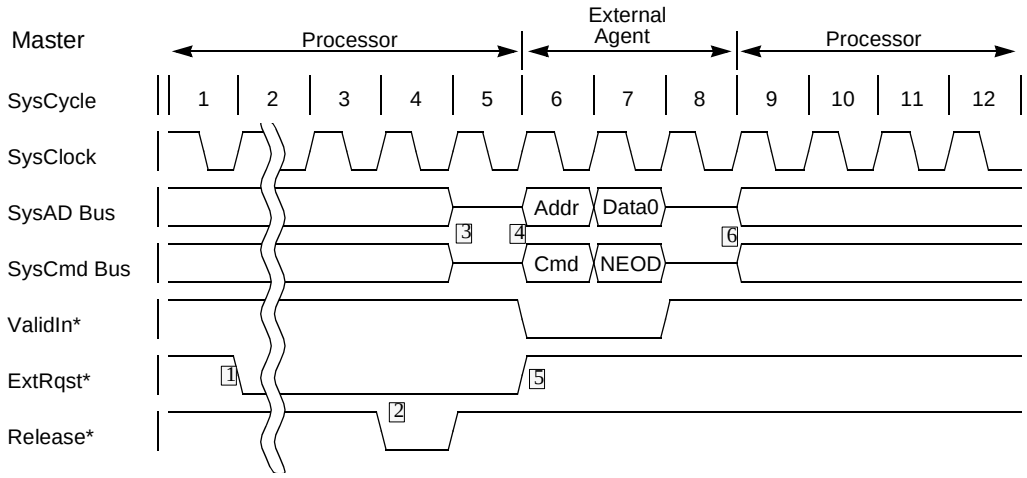


Figure 12.14 Arbitration Protocol for External Requests

System interface arbitration uses the signals **ExtRqst*** and **Release*** as described above. Figure 12.14 shows 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*** to request mastership of the **SysAD** bus.
2. The processor waits until it is ready to handle an external request, whereupon it asserts **Release*** for one cycle.
3. The processor tristates the **SysAD** and **SysCmd** buses.
4. The external agent must wait at least two cycles after the assertion of **Release*** and then drives the **SysAD** and **SysCmd** buses.
5. The external agent **ExtRqst*** two cycles after the assertion of **Release***, unless the external agent wishes to perform an additional external request.
6. The external agent must tristate the **SysAD** and **SysCmd** buses at the completion of an external request.

The processor can issue a processor request one cycle after the external agent tristates the bus.

Note: Timings for the **SysADC** *and* **SysCmdP** *buses are the same as those of the* **SysAD** *and* **SysCmd** *buses, respectively.*

12.6.2 External Null Request Protocol

The processor supports a system interface external null request, which 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.

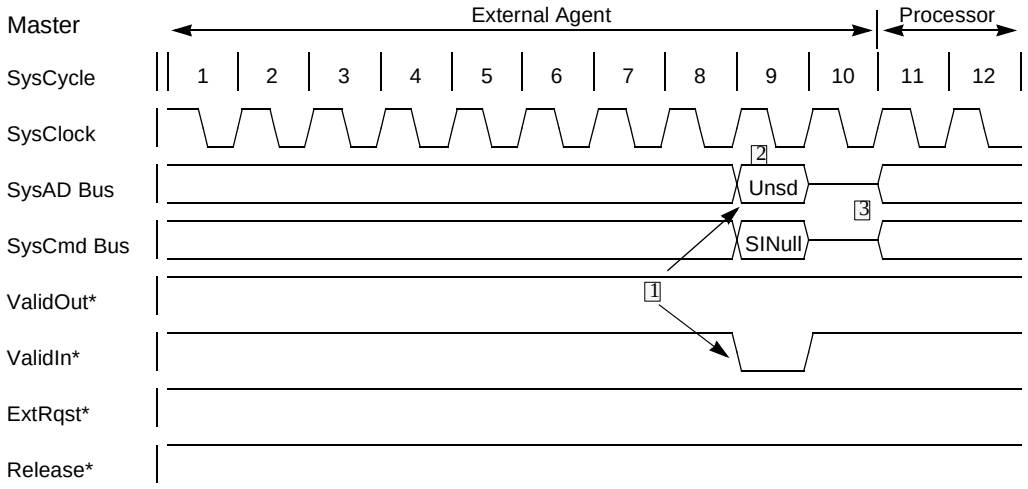


Figure 12.15 System Interface Release External Null Request

Figure 12.15 shows a timing diagram of an external null request, which consists of the following steps:

1. The external agent drives a system interface release external null request command on the **SysCmd** bus, and asserts **ValidIn*** for one cycle to return system interface mastership to the processor.
2. The **SysAD** bus is unused (does not contain valid data) during the address cycle associated with an external null request.
3. 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.

12.6.3 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 12.16 shows a timing diagram of an external write request, which consists of the following steps:

1. The external agent asserts **ExtRqst*** to arbitrate for mastership of the system interface.
2. The processor releases the system interface to slave state by asserting **Release*** for one cycle.
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 noncoherent last data cycle indication.
6. After the data cycle is issued, the write request is complete and the external agent tristates the **SysCmd** and **SysAD** buses, 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.

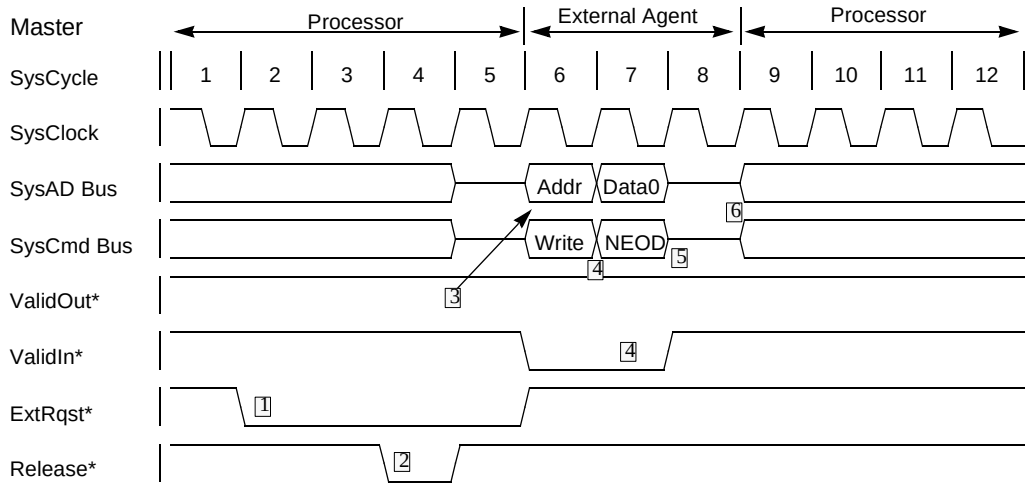


Figure 12.16 External Write Request; System Interface Initially in Master

External write requests are only allowed to write a 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.

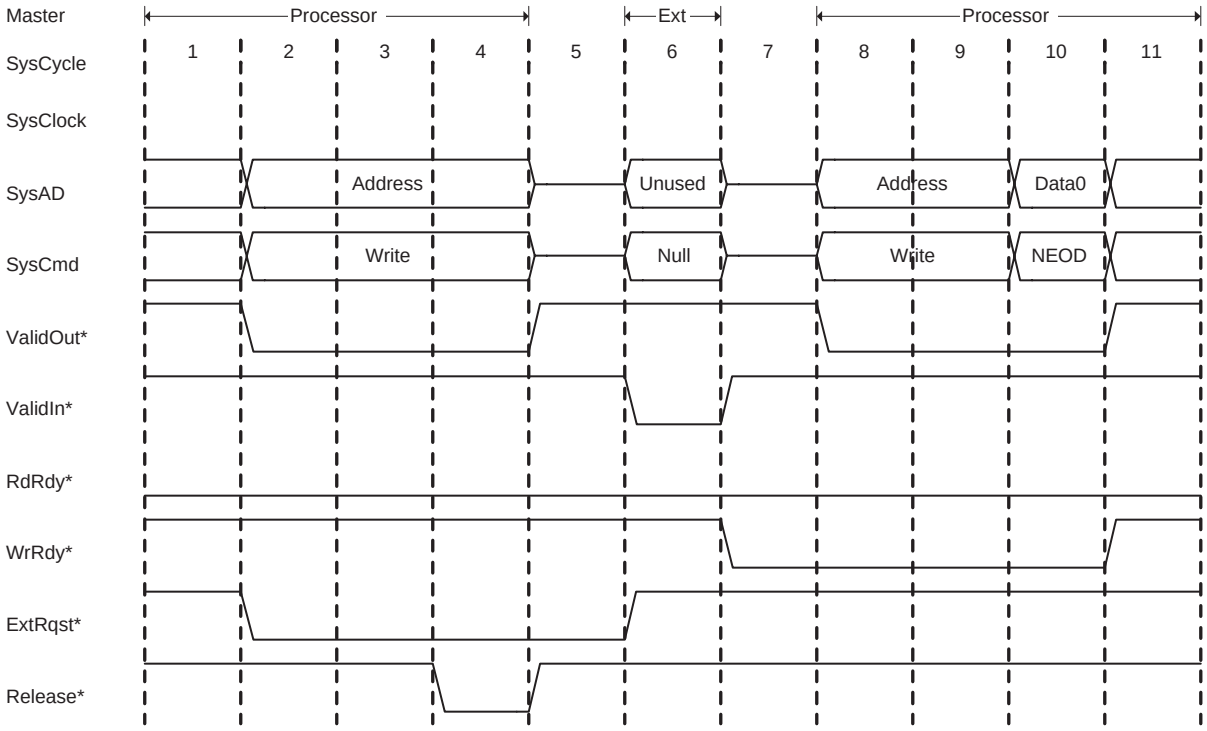


Figure 12.17 Write Request Waiting for WrRdy* Abandoned Temporaily for an External Null Request

Figure 12.17 shows the timing for for a write request when it had been adandoned for an external null request.

12.6.4 Read Response Protocol

An external agent must return data to the processor in response to a processor read request using the read response protocol. A read response 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 either 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 tristates the **SysCmd** and **SysAD** buses.
4. The system interface returns to master state.
5. The processor always performs an uncompelled change to slave state after issuing a read request.
6. The data identifier for data cycles must indicate the fact that this data is *response data*.
7. 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, the response data does not need to identify the initial cache state. The cache state is automatically 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, the external agent must return a block of data the correct size regardless of the fact that the data may be in error.

The processor only checks the error bit for the first doubleword of a block. The remaining error bits for a block are ignored.

Read response data must only be delivered to the processor when a processor read request is pending. The behavior of the processor is undefined if a read response is presented to it when there is no processor read pending.

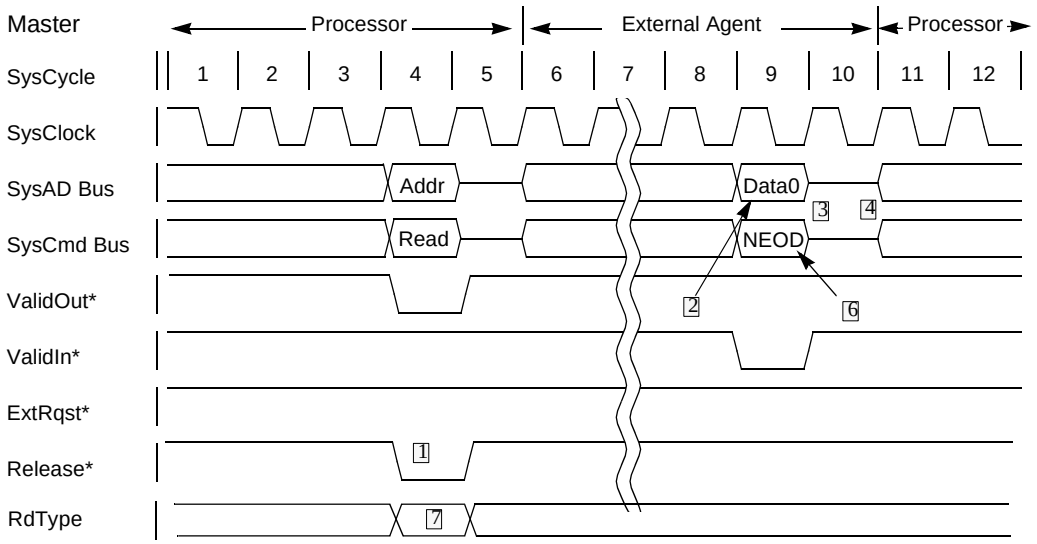


Figure 12.18 Processor Word Read Request, followed by a Word Read Response

Figure 12.18 illustrates a processor word read request followed by a word read response.

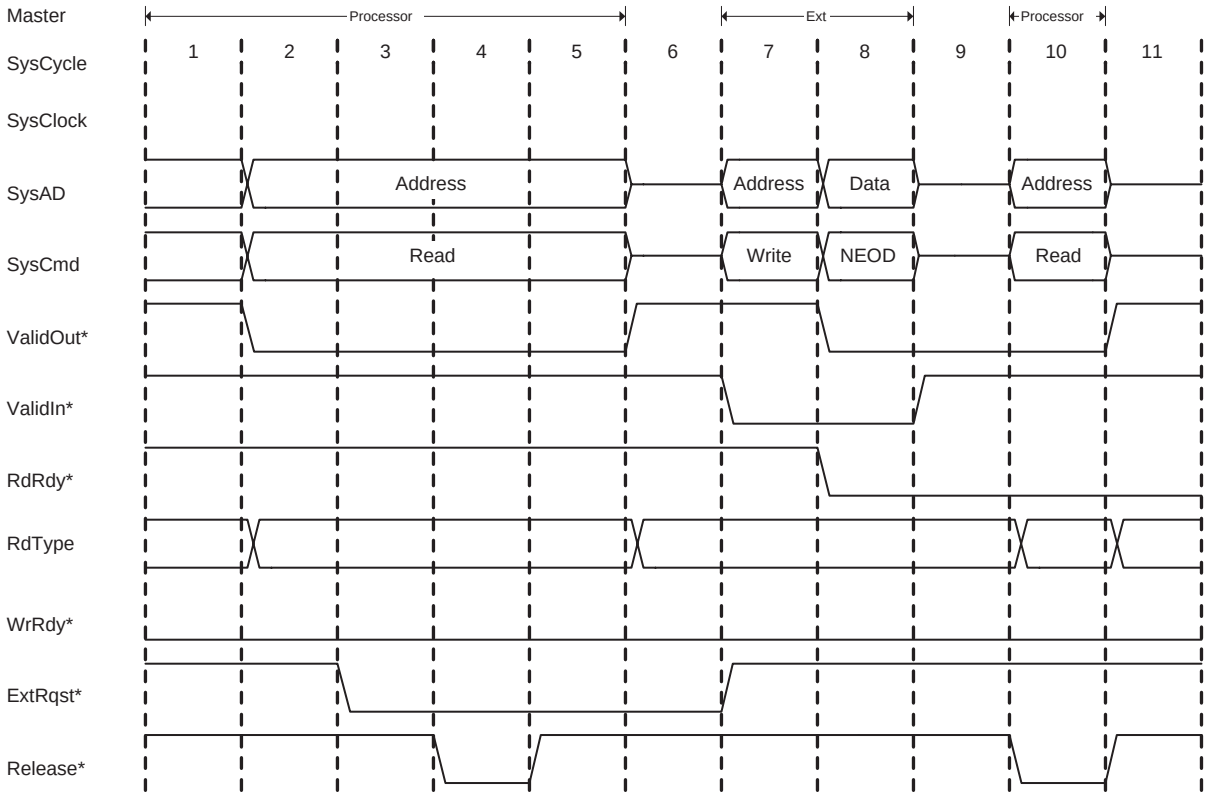


Figure 12.19 Read Request Wating for RdRdy* Abandoned Temporarily for an External Write Request.

Figure 12.19 illustrates a read response that has been interrupted by an external write operation.

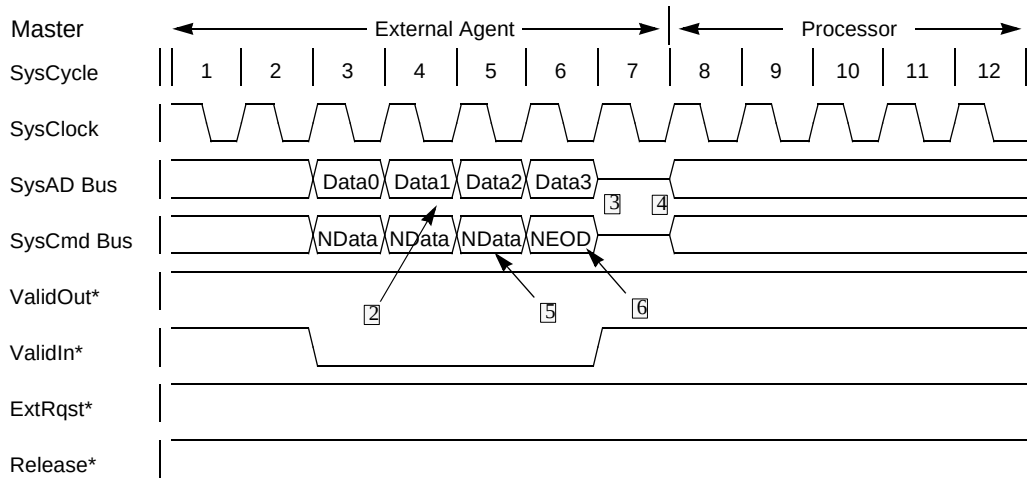


Figure 12.20 Block Read Response, System Interface already in Slave State

Figure 12.20 illustrates a read response for a processor block read with the system interface already in slave state.

Note: Timings for the **SysADC** *and* **SysCmdP** *buses are the same as those of the* **SysAD** *and* **SysCmd** *buses, respectively.*

12.7 Tertiary Cache Read Protocol

There are three scenarios that can occur on a tertiary cache read access.

1. Tertiary cache read hit
2. Tertiary cache read miss
3. Tertiary cache read miss with bus error

Each of these scenarios is discussed in the following sections.

12.7.1 Tertiary Cache Read Hit

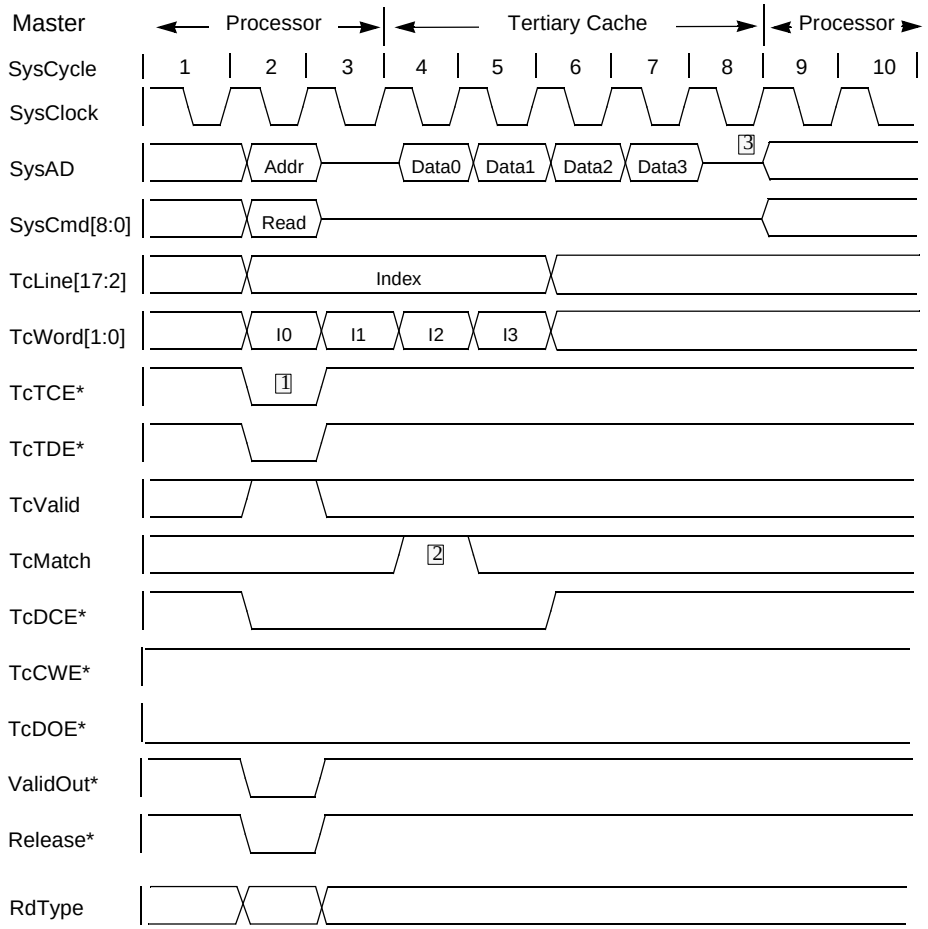


Figure 12.21 Tertiary Cache Read Hit

Figure 12.21 shows the tertiary cache read hit protocol. At the time the read request is issued, it is not known where the requested data resides. Block read requests are issued to both the tertiary cache and main memory simultaneously. If the data is found in the tertiary cache, it is fetched and the main memory access is aborted. However, if the data is not found in the tertiary cache, the main memory access is allowed to continue. When a block read request is speculatively issued to both the tertiary cache and the external agent controlling main memory, but completed by the tertiary cache (tertiary cache hit), the following sequence of steps occur:

1. The processor issues a block read request and also asserts the **TcTCE***, **TcTDE***, and **TcDCE*** tertiary cache control signals. In addition the processor drives the cache index onto **TcLine[15:0]** and the sub-block order doubleword index onto **TcWord[1:0]**. Assertion of **TcTCE***, along with **ValidOut*** and **SysCmd**, indicates to the external agent that this is a tertiary cache read request. In addition, the assertion of **TcTCE*** initiates a tag RAM probe. The assertion of **TcTDE*** loads the tag portion of the **SysAD** bus into the tag RAM. The **TcValid** signal is asserted to probe for a valid cache tag. The assertion of **TcDCE*** initiates a speculative read of the tertiary cache data RAMs.
2. The **TcMatch** signal from the tag RAM is sampled by both the processor and the external agent. Assertion of **TcMatch** indicates a tertiary cache tag hit, causing the external agent to abort the memory read. This transitions the bus from Master state to Tcache state. The data RAMs now drive the **SysAD** bus and supply the first of a 4 doubleword burst in response to the 4-cycle **TcDCE*** burst. The **SysCmd** bus is not driven during the tertiary cache read.

3. Mastership of the **SysAD** bus is returned to the processor.

12.7.2 Tertiary Cache Read Miss

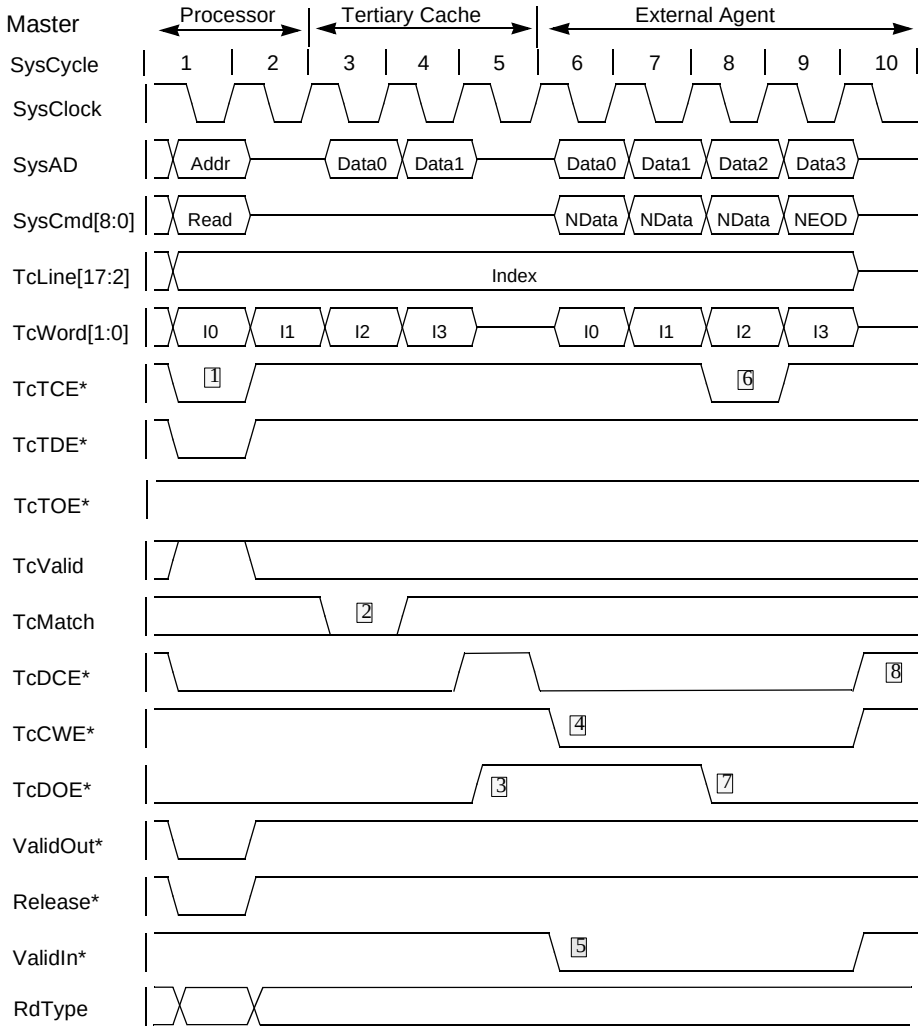


Figure 12.22 Tertiary Cache Read Miss

Figure 12.22 shows the tertiary cache read miss protocol when a block read request is speculatively issued to both the tertiary cache and the external agent, but is completed by the external agent with a response to both the tertiary cache and the processor.

1. The processor issues a block read request and also asserts the **TcTCE***, **TcTDE***, **TcDCE***, and **TcValid** signals and drives the cache index onto **TcLine[15:0]** and **TcWord[1:0]**.
2. The **TcMatch** signal from the tag RAM is sampled by the processor and external agent. Since the signal is deasserted, indicating a tertiary cache miss, the **SysAD** data from the tertiary cache is invalid.
3. The external agent deasserts **TcDOE*** to tri-state the data RAM outputs, indicating that it will be supplying the read response. The processor tri-states its **TcWord[1:0]** outputs to allow the external agent to drive them during the read response.
4. The processor asserts **TcCWE*** to prepare the data RAMs for a write of the response data.
5. The external agent supplies the first doubleword of the read response and asserts **ValidIn***. The data is both written into the tertiary cache and accepted by the processor. **SysCmd** indicates that data is not erroneous. Note that this response may be delayed additional cycles.
6. The processor asserts **TcTCE*** to write the tag value stored in the tag RAM data input register two cycles after **ValidIn*** is asserted.

7. The external agent asserts **TcDOE*** to indicate that it will supply the last doubleword of the read response in the next cycle.
8. The processor deasserts **TcDCE*** two cycles after the assertion of **TcDOE*** in order to complete the tertiary cache line fill.

12.7.3 Tertiary Cache Read Miss with Bus Error

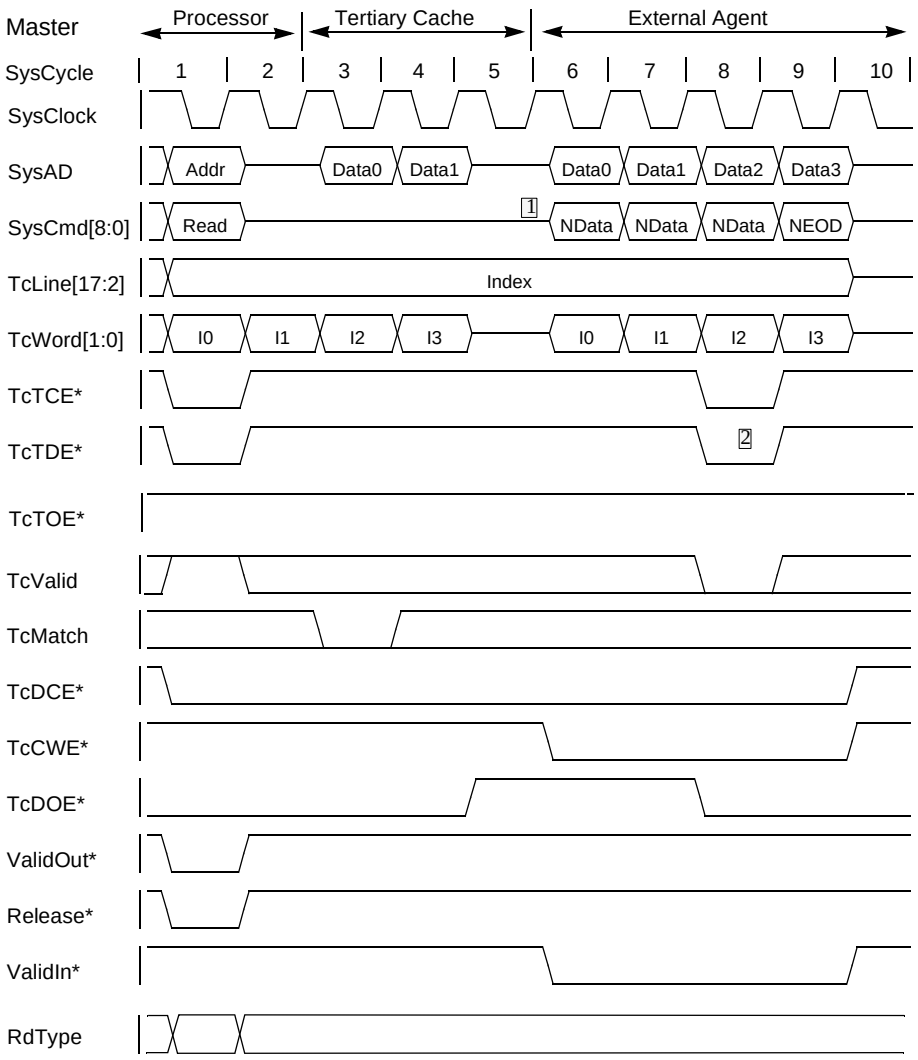


Figure 12.23 Tertiary Cache Read Miss with Bus Error

Figure 12.23 shows the tertiary cache read miss with bus error protocol. This protocol is the same as the tertiary cache read miss except:

1. The external agent supplies the first doubleword of the read response data with the data error bit set (**SysCmd[5]**=1). Note that the data error bit of **SysCmd** is only checked during the first doubleword of a read response.
2. The processor asserts **TcTCE*** and **TcTDE*** to write the new tag value into the tertiary cache tag RAM with **TcValid** deasserted to invalidate the line.

12.7.4 Tertiary Cache Read Waiting for RdRdy*

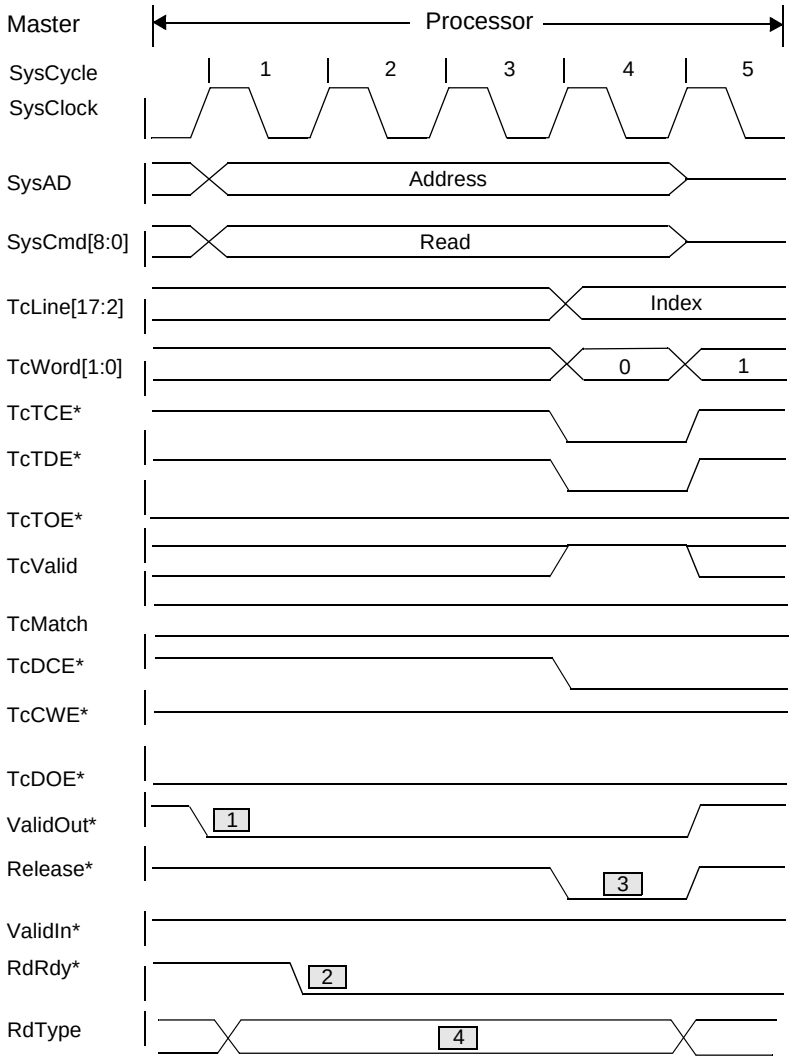


Figure 12.24 Tertiary Cache Read Waiting for RdRdy*

Figure 12.24 shows a tertiary cache read stalled waiting for **RdRdy*** to assert. The number sequence below corresponds to the numbers in Figure 12.24.

1. **ValidOut*** is asserted along with address and command.
2. **RdRdy*** is asserted, clearing the read stall condition.
3. Two cycles later the tertiary cache read is issued.
4. RdType will be high if the address cycle is for an instruction read. RdType will be low if the address cycle is for a data read. RdType is NOT valid for all other cycles.

12.8 Tertiary Cache Write

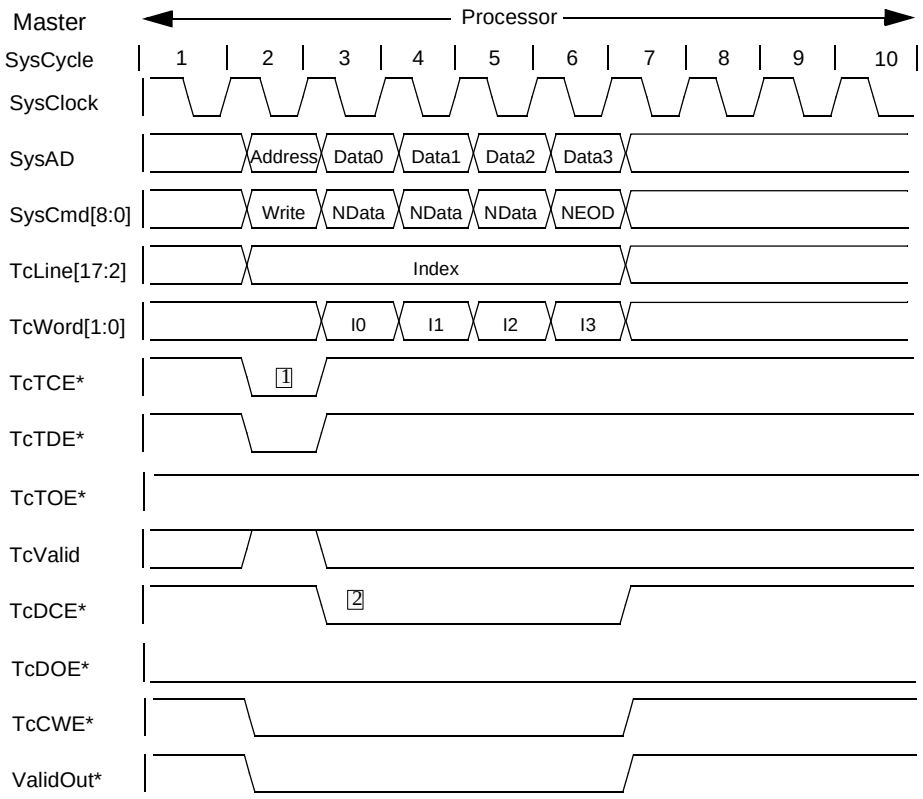


Figure 12.25 Tertiary Cache Write Operation

Figure 12.25 shows the tertiary cache write protocol. For the external agent, this protocol is the same as a non-tertiary cache block write to the external agent, but the data is also written into the tertiary cache.

1. The processor issues a block write and also asserts **TcTCE***, **TcTDE***, and **TcCWE*** in order to write the tag portion of the address on **SysAD** into the tertiary cache tag RAM. The processor asserts **TcValid** to set the tertiary cache tag to valid.
2. The processor asserts **TcDCE*** to write the block into the tertiary cache data RAMs.

12.9 Tertiary Cache Write Stalls

There are three types of tertiary cache write stalls where a write is stalled by the deassertion of **WrRdy*** or waiting for **WrRdy*** to assert.

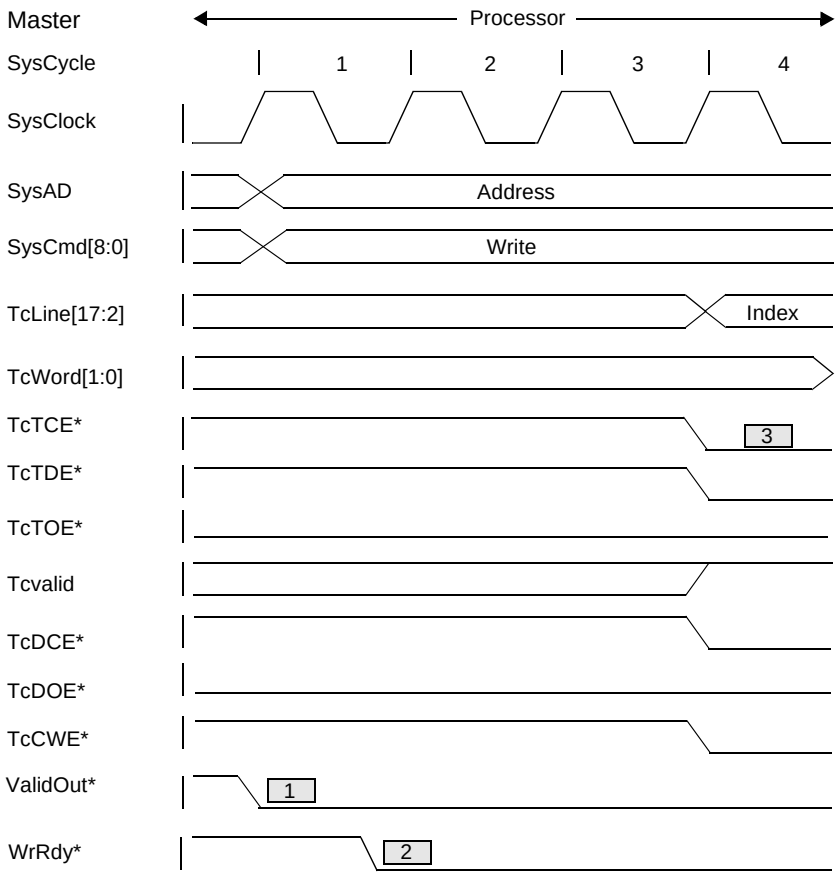


Figure 12.26 Tertiary Cache Write Stall - R4000-Compatible Mode

Figure 12.26 shows a tertiary cache write stall in *R4000-Compatible* write mode.

1. **ValidOut*** asserts together with address and command,
2. **WrRdy*** asserts, ending the stall condition,
3. Two cycles later the tertiary cache write issues.

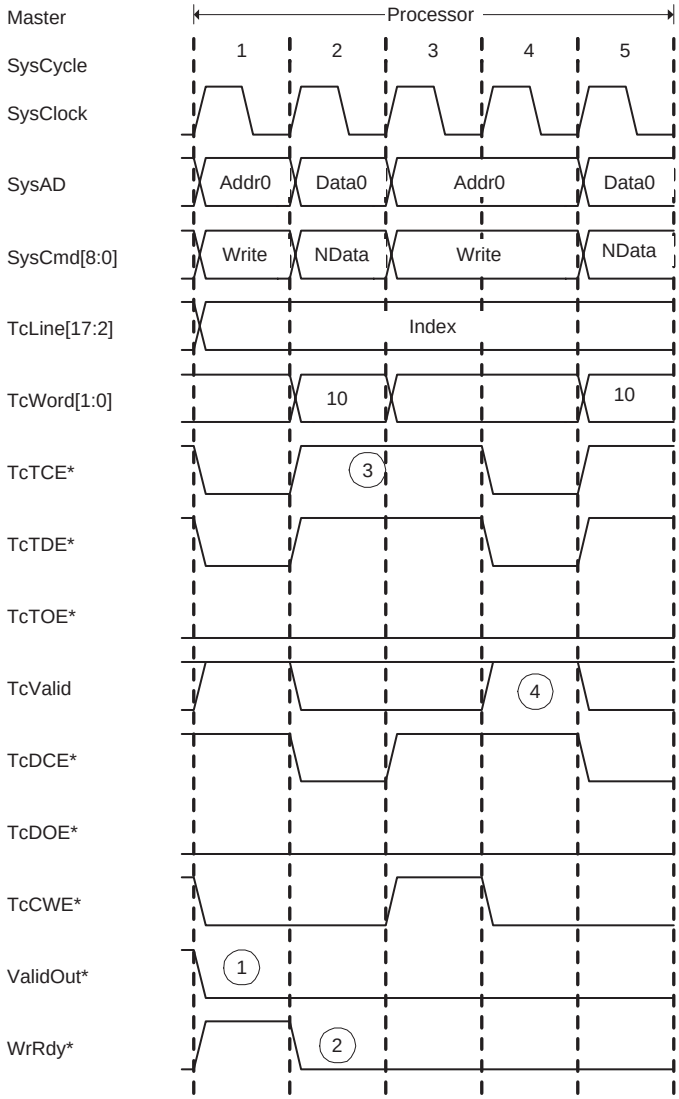


Figure 12.27 Tertiary Cache Write Stall - Write Reissue Mode

Figure 12.27 shows a tertiary cache write stall in *Write-Reissue* mode. In *Pipelined-Write* mode, deassertion of **WrRDY*** during a tertiary cache write operation does not affect that operation.

1. **ValidOut*** asserts along with address and command. **WrRdy*** deasserts, indicating another write cannot be accepted.
2. **WrRdy*** asserts, indicating another write can now be accepted.
3. The write is aborted and the tertiary cache write signals are deasserted.
4. Two cycles after **WrRdy*** asserts, the tertiary cache write reissues.

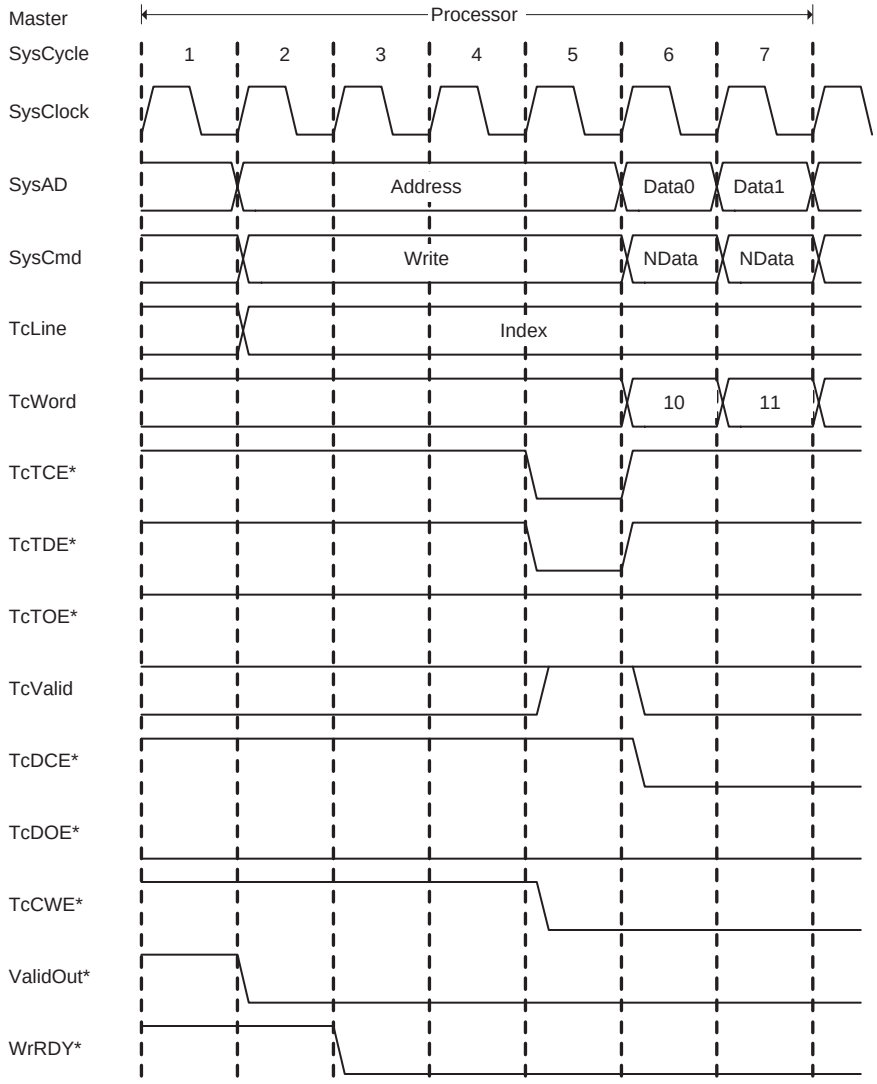


Figure 12.28 Tertiary Cache Write Waiting for WrRdy*

Figure 12.28 shows a tertiary cache write while waiting for **WrRdy***.

12.10 Tertiary Cache Line Invalidate

The RM7000 processor has the ability to invalidate either a single line or 128 consecutive lines (address aligned) of the tertiary cache. The invalidate operation consists of writing to the Tag RAM and invalidating the line in question. The **TcTCE***, **TcTDE***, and **TcCWE*** signals are driven active in the same clock as the **SysAD** and **TcLine** buses with **TcValid** deasserted. Invalidates are the only cache operations which may occur back-to-back. Note that **ValidOut*** is not asserted during tertiary cache invalidate operations as the external agent does not participate in tertiary cache invalidates.

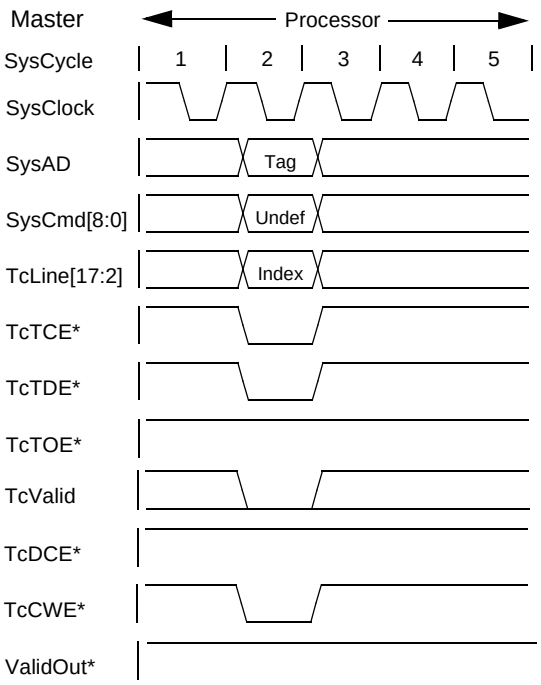


Figure 12.29 Tertiary Cache Line Invalidate

Figure 12.29 shows the tertiary cache invalidate protocol. The repeat rate for cache line invalidate instructions is two cycles. The repeat rate for cache page invalidate is one cycle per line for 128 consecutive cycles.

12.11 Tertiary Cache Probe Protocol

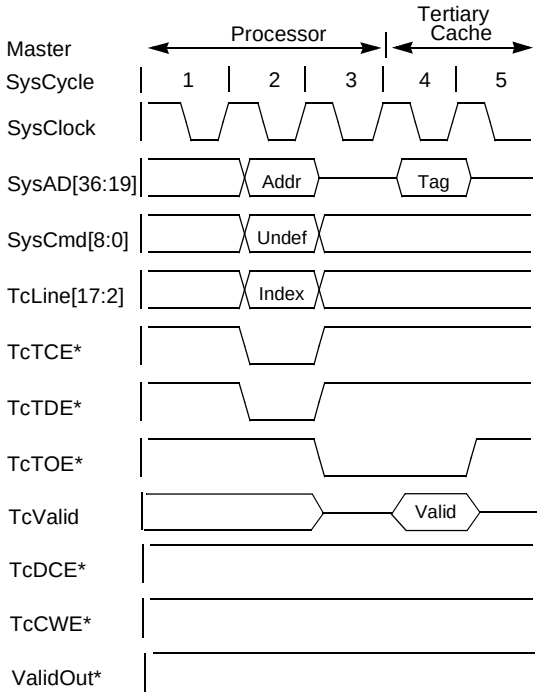


Figure 12.30 Tertiary Cache Probe (Tag RAM Read)

Figure 12.30 shows the tertiary cache probe protocol. The tertiary cache probe operation consists of a Tag RAM read operation. The **TcTCE*** and **TcTDE*** signals are asserted in the same clock as system address and the tertiary cache line index. The

processor then tri-states the **SysAD** bus. **TcTOE*** is asserted one clock later and the tag information is driven onto the **SysAD** bus. **ValidOut*** is not asserted during a tertiary cache probe operation as the external agent does not participate in tertiary cache probes. The Tag RAM bits are driven onto **SysAD [35:19]** and **TcValid**, which are the only **SysAD** signals valid during a probe operation.

12.12 Tertiary Cache Flash Clear Protocol

In addition to the line invalidate operation, the RM7000 processor also has the ability to invalidate the entire tertiary cache in one operation. This operation allows the processor to clear the entire column of Tag RAM valid bits. In order to execute this operation the Tag RAM must support a flash clear of the valid bit column.

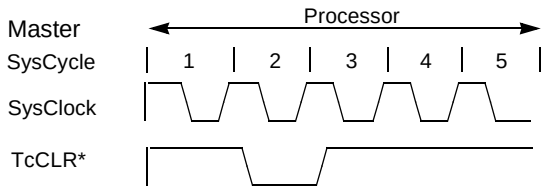


Figure 12.31 Tertiary Cache Flash Clear

As with the line invalidate operation, **ValidOut*** is not asserted during the flash clear operation as the external agent does not participate in flash clear operations. In addition, the **TcTCE***, **TcTDE***, and **TcCWE*** signals are not asserted. The assertion of **TcCLR*** is all that is necessary for the Tag RAM to perform the requested operation. Figure 12.31 illustrates the tertiary cache flash clear protocol.

12.13 Non-Pendant Bus Transactions

The external protocol of the RM7000 has been extended to allow two outstanding reads to be pending at the same time, and the responses to those reads returned either in-order or out-of-order. For the most part, the external interface protocol is the same whether or not the extended protocol is enabled. The principal differences are listed below and detailed logic waveforms for the affected cases are illustrated in the following subsections:

1. The principal difference is the location of the tertiary cache tag write. In R5000-compatible mode, the tag is not written until near the end of the return data transfer from the external agent. In the extended protocol this write occurs as soon as it is known that a tertiary miss has occurred. This placement allows the processor to perform another tertiary cache lookup before the data from the first miss has arrived.
2. Beyond the slight differences in the already existing signals as noted above, the extended protocol relies on two new signals, **PRqst*** and **PAck***, to effect an interlocked interface mastership transfer from the external agent back to the processor when the processor desires to issue a second read prior to the completion of the currently outstanding read. **PRqst*** is only asserted to place another processor read request on the bus. It is never asserted for processor write requests. A third signal, **RspSwap***, is used by the external agent to notify the processor that it will be returning read data out-of-order. In other words, the data associated with the second address will be returned before the data associated with the first address.
3. Although strictly speaking, the handling of **TcDOE*** is no different from the non-extended protocol, it is useful to make some general statements about the purpose of **TcDOE*** and how it can most easily be handled in the extended protocol. **TcDOE*** should be held normally asserted by the system ASIC and must be deasserted during the tristate cycle to immediately disable the tertiary cache data RAM output drivers. The reassertion of **TcDOE*** has no effect on the RAM outputs since write enable, **TcCWE***, is asserted at the time **TcDOE*** is reasserted. Rather, the reassertion edge is used to signal the processor that the last datum is coming on the following cycle, and this information is used to cause the deassertion of **TcCWE*** and **TcDCE***. **TcDOE*** must be reasserted on the return to master state on the cycle following a **PAck*** assertion cycle. It must then remain asserted until a transition back to slave state.
4. **PRqst*** will only be asserted by the processor when the system interface is in slave state. **PRqst*** can be asserted only if there is one and only one read request pending on the bus. The pin will remain asserted until two bus cycles after **PAck*** is asserted or until two cycles after the first data cycle occurs for the pending read request.
5. **PAck*** may be asserted by the external agent only if the following conditions are all true:

a. The system interface is in slave state
b. **PRqst*** is asserted
c. **RdRdy*** is asserted
d. **ExtRqst*** is not asserted
e. **ValidIn*** is not asserted.

Once all these conditions are true, **PAck*** may be asserted for one cycle.

6. Once a pulse on the **PAck*** pin has returned the bus to master state, the processor will issue another read address cycle. At this time, **RdRdy*** must not be de-asserted until two cycles after the address cycle. Also, **ExtRqst*** must not be asserted until bus cycles after the address cycle.
7. The processor will not issue a second processor read request after an assertion of **ExtRqst*** has put the bus in master state. The processor will only issue a second read request through the **PRqst*** - **PAck*** handshake sequence.
8. The bus will remain in slave state after an EOD data cycle if there is another read request pending on the bus. After an external request has completed, the processor will do an uncompelled change to slave state if there is a pending read request.
9. Processor write requests will never be issued by the processor if there is a processor read request pending on the bus.

This extended non-pendant behavior is enabled by a bit in the boot initialization stream. If this bit is clear, then the processor will operate in R5000 compatibility (pendant) mode. In R5000 compatibility mode, the processor will never assert **PRqst***, so the system can tie both **PAck*** and **RspSwap*** pins to VCC. If the non-pendant mode is enabled and the external agent is not capable of returning data out-of-order, then the **RspSwap*** pin can be tied to Vcc.

12.13.1 Non-Pendant Tertiary Cache Read Hit

For reads which hit in the tertiary cache, there is no difference between the R5000-compatible protocol and the extended protocol.

12.13.2 Non-Pendant Tertiary Cache Read Miss, One-Read

Figure 12.32 shows the extended protocol for a block read request which misses in the tertiary cache when there are no other processor actions prior to the servicing of the request by the external agent.

1. All actions of the processor up until time 1 are the same as in the R5000-compatible protocol. At 1, the processor asserts **TcTCE*** and **TcCWE*** in order to effect the tag write, and deasserts **TcDCE*** for a single cycle in order to prevent the data array from being written prior to the external agent having taken control of **TcWord[1:0]**.

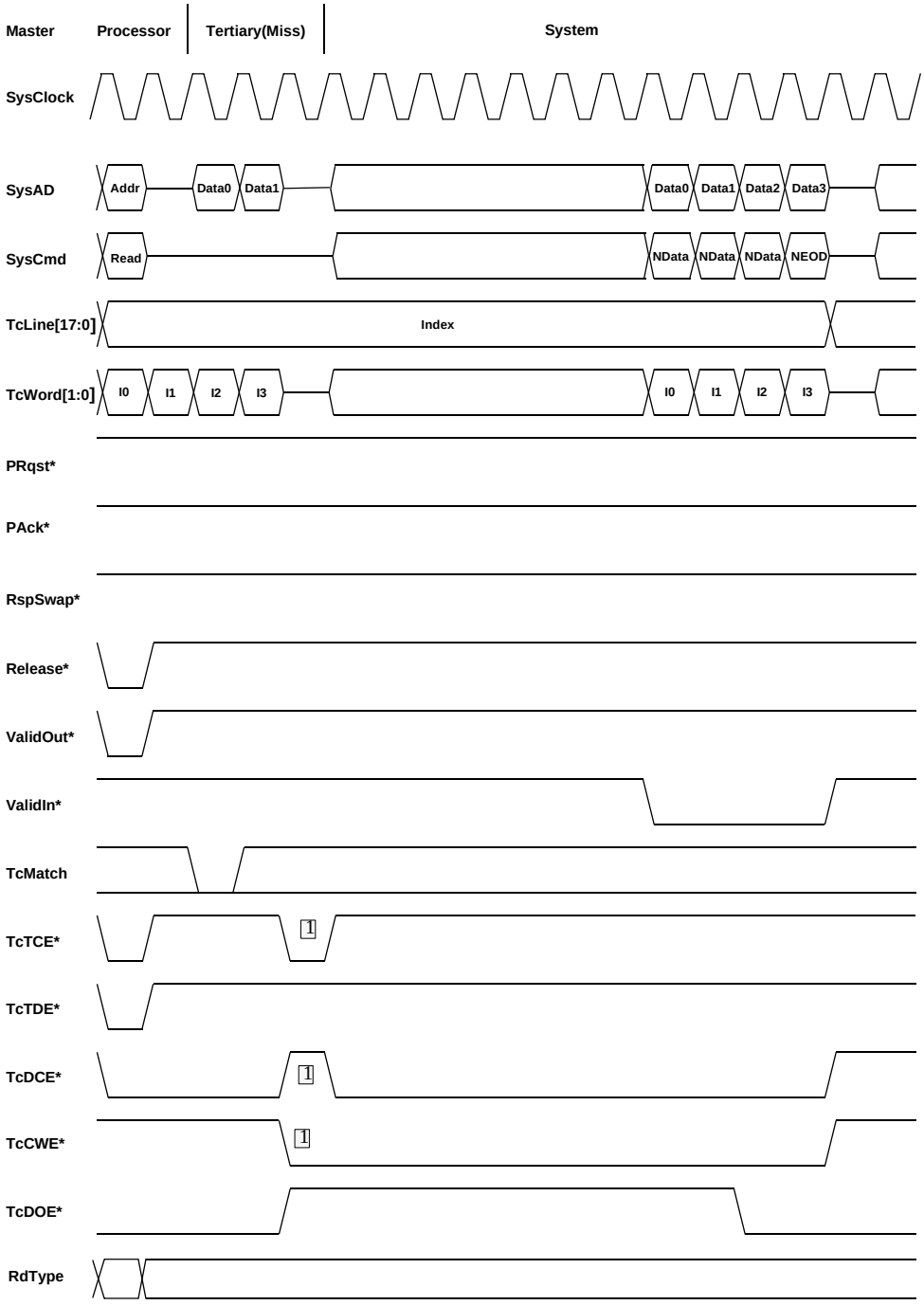


Figure 12.32 Tertiary Cache Read Miss, One Read

12.13.3 Non-Pendant Two Reads, In-Order Return

Figure 12.33 shows the tertiary cache read miss protocol in which the following events occur.

- A block read request misses in the tertiary cache.
- A second block read request also misses in the tertiary cache.
- The first request is serviced by the external agent.
- The processor transitions into a state equivalent to the simple one-read case of Figure 12.32.

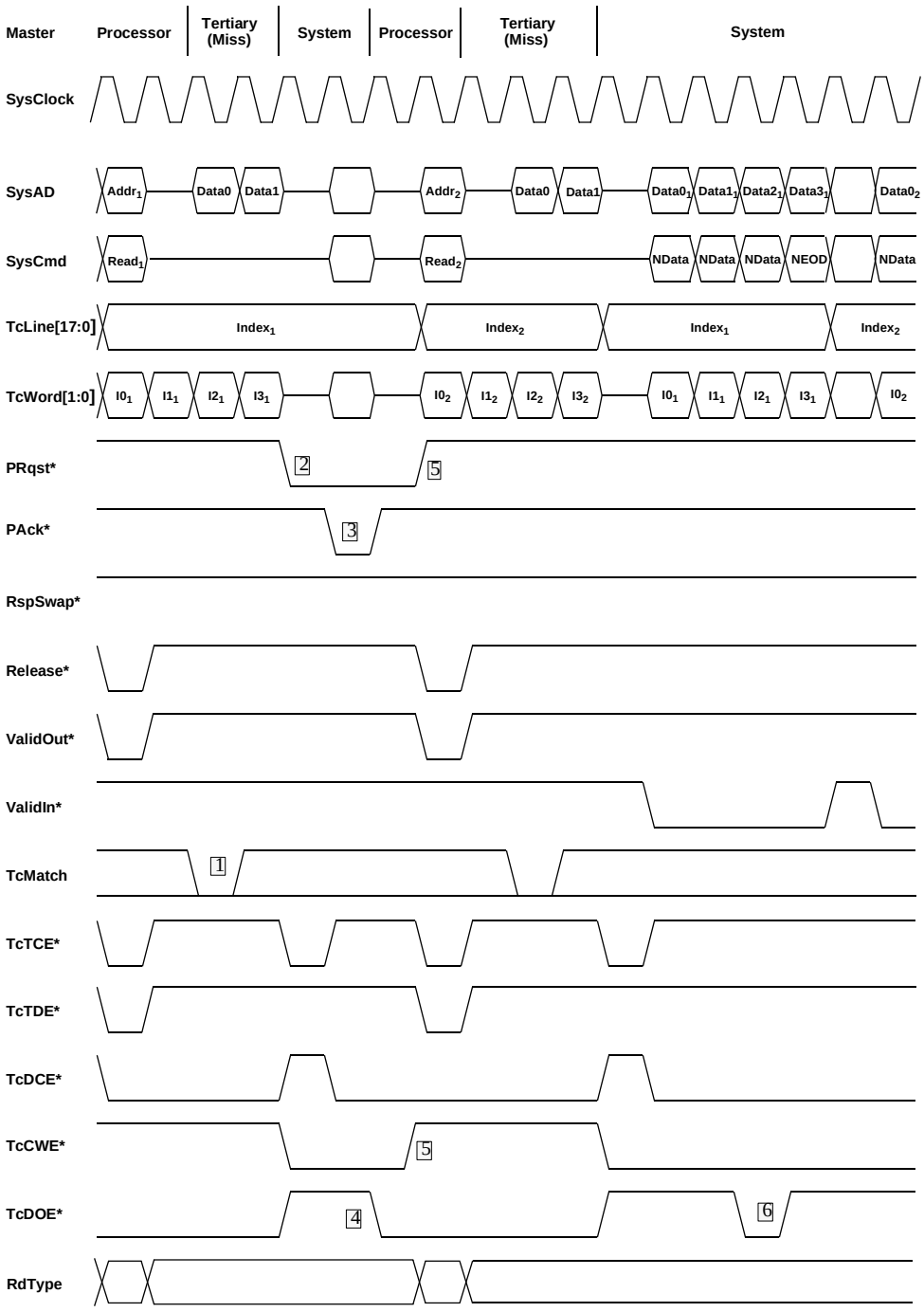


Figure 12.33 Two Outstanding Reads, In-Order Return

The signal transitions in Figure 12.33 are as follows:

1. The **TcMatch** signal from the tag RAM is sampled by the processor and external agent indicating a tertiary cache miss.
2. The processor asserts **PRqst*** to request ownership of the bus.
3. The external agent asserts **PAck*** returning mastership of the bus to the processor.
4. Simultaneous with the deassertion of **PAck***, the external agent reasserts **TcDOE*** to re-enable the Tag RAM output drivers in preparation for a new tertiary cache access.
5. The processor deasserts **PRqst***, completing the handshake and simultaneously deasserts **TcCWE*** in preparation for a Tag read. **Release***, **ValidOut***, etc. are asserted just as for the issue of any block read.
6. After asserting **TcDOE*** to signal the final data word, the external agent must again deassert **TcDOE*** in preparation for the memory data returning from the second read.

12.13.4 Non-Pendant Two Reads, Out-of-Order Return

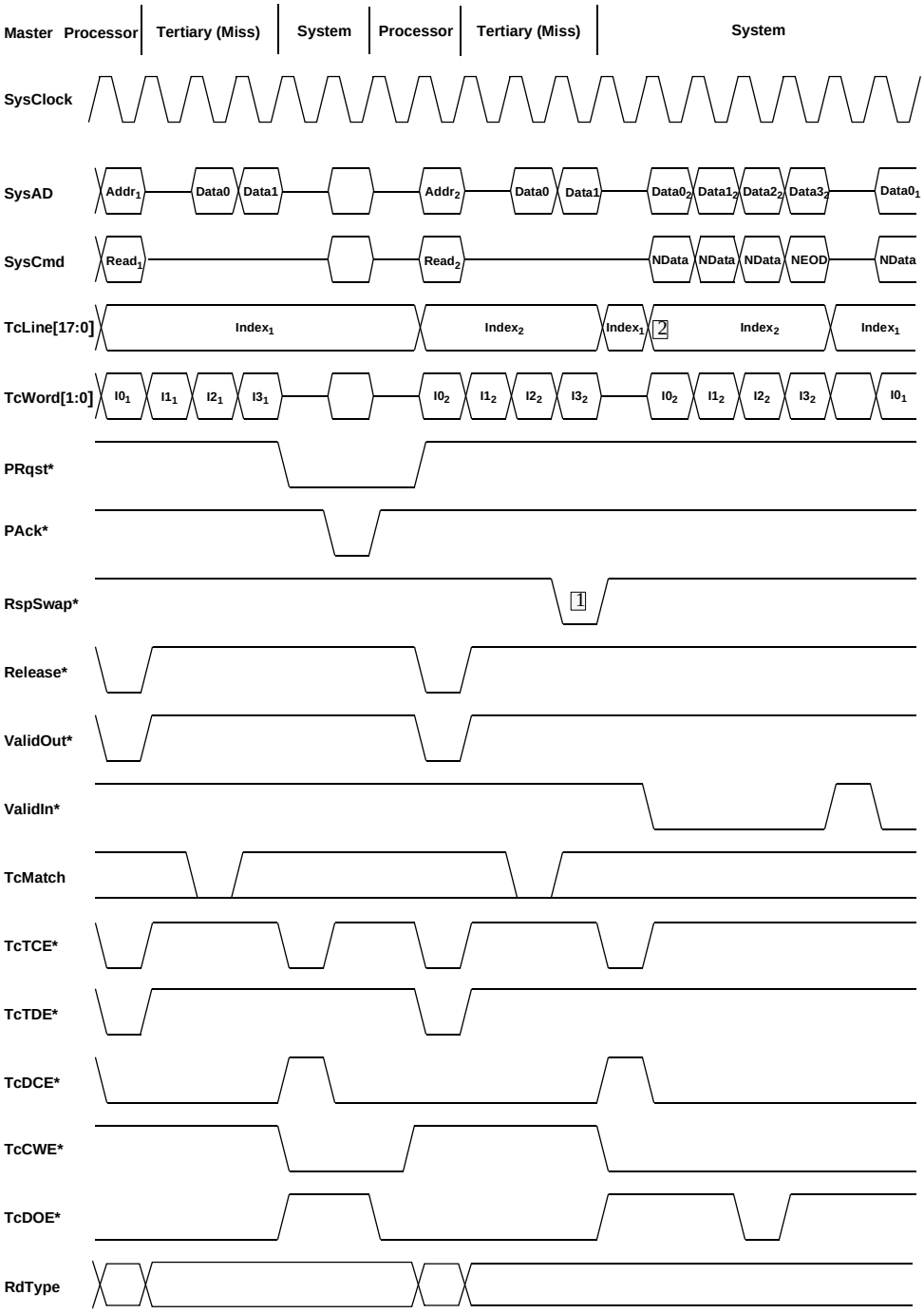


Figure 12.34 Two Outstanding Reads, Out-of-Order Return

Figure 12.34 shows a case where the data is returned by the external agent in the opposite order from which it was requested.

1. A minimum of two cycles prior to returning the data, the external agent must assert **RspSwap*** (for one cycle) so that the processor can switch the Tag RAM address. This pin can be asserted as soon as the bus reaches slave state (if there is no tertiary cache) or as soon as the cycle after the tertiary cache lookup has gotten a miss for the second request and deas-serted **TcMatch**. **RspSwap*** must be asserted a minimum of two cycles ahead of when the first data cycle appears on the bus, so the processor has time to switch to the correct cache index when writing the data into the caches. The **RspSwap*** pin must be asserted for only one cycle as every cycle in which it's asserted will toggle the processors view of the data between in-order return and out-or-oder return.

2. The processor switches the tertiary cache address lines to the index to the address of the second read request in preparation for writing the data into the tags. A similar process occurs for the on-chip caches.

12.14 Non-Pendant Read Hit Under Outstanding Read

Figure 12.35 shows a tertiary cache read hit occurring while a previous read is pending. Steps where operation is distinct from the earlier cases are listed below.

1. The deassertion of **TcDOE*,** as well as the turning on of the external agent's **SysAD** and **TcWord** drivers, occurs in response to the completion of the tertiary cache read.
2. Since the tag has already been written, the processor waits one cycle before asserting **TcCWE***. There is no activity on **TcDCE***.
3. As usual, the external agent reasserts **TcDOE*** during the cycle prior to the last data cycle and the processor deasserts **TcCWE*** and **TcDCE*** in response**.**

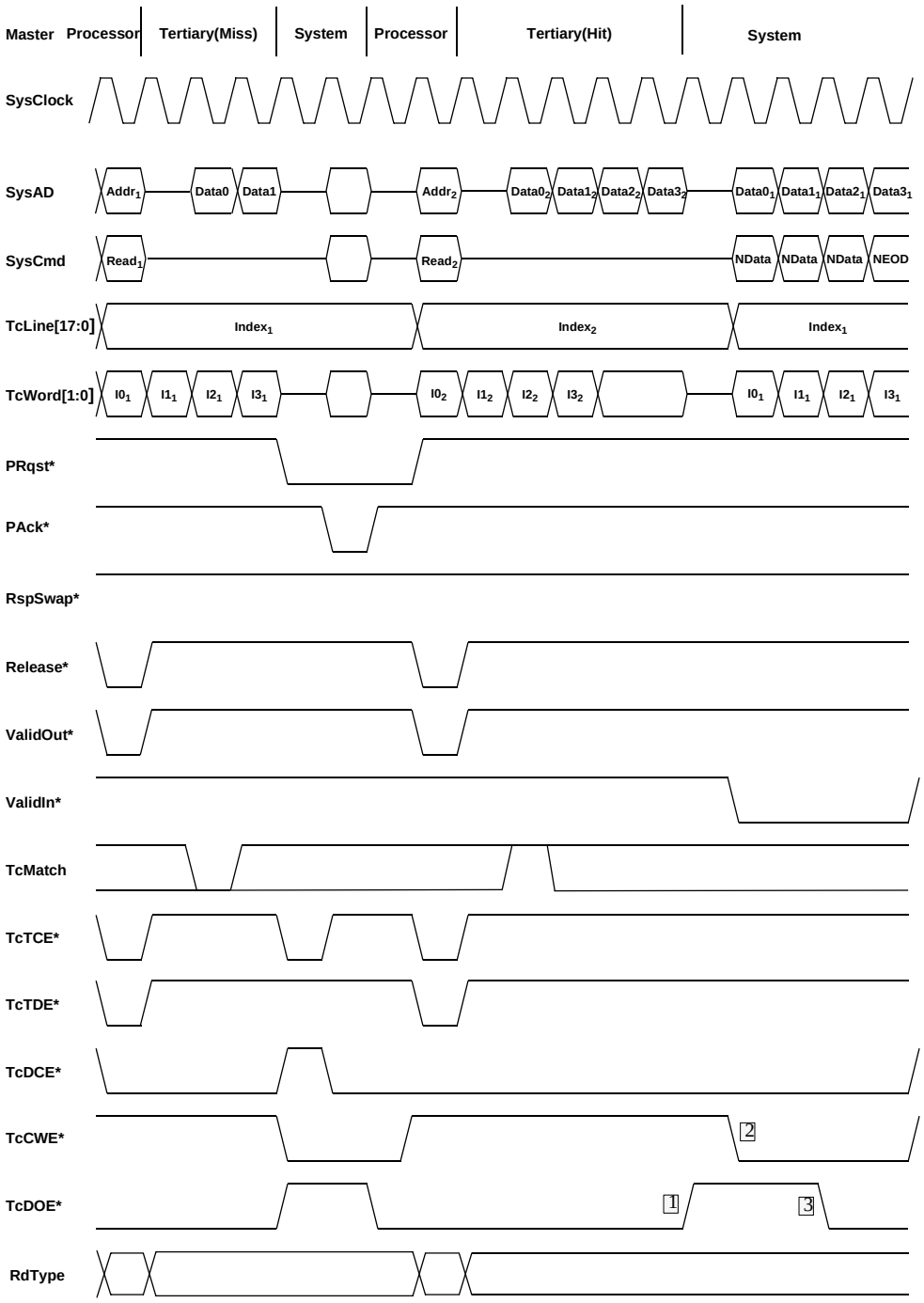


Figure 12.35 Read Hit Underneath Outstanding Read

12.15 SysADC[7:0] Protocol

The following rules apply to the data identifiers on **SysADC[7:0]** during a block read response.

- The erroneous data bit (**SysCmd5**) is checked on only the first doubleword of the transfer. If data is erroneous (**SysCmd5**=1), the primary cache lines are invalidated and a bus error exception is generated.
- A parity error on the first doubleword is detected as it issued and causes a cache parity error exception. The cache line is marked invalid. Parity errors in subsequent doubles are detected if and when they are referenced.
- On the following three doublewords: The erroneous data bit is ignored. Parity for each of the three doublewords is written into the cache, but is not checked until the data is referenced.

- Any read that fills the tertiary cache must receive correct parity for all 4 doublewords (**SysCmd[4]**=0) for data going to the tertiary cache.
- For a tertiary cache read hit: The erroneous data bit is ignored. Parity checking is implicitly ON, such that the tertiary cache must implement the **SysADC** bits.
- If a memory error occurs during a block read operation, the **SysADC** bits should be forced to bad parity for all bytes affected by the memory error during the read response. Since the processor performs an early-restart on data cache line fills, setting the **SysCmd[5]** bit on any doubleword other than the first doubleword will not cause a bus error. Forcing bad parity causes a cache error to be generated if any of the remaining three doublewords of data are referenced.

12.16 Data Rate Control

The system interface supports a maximum data rate of one doubleword per cycle. 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.

The processor only accepts data as valid when **ValidIn*** is asserted and the **SysCmd** bus contains a data identifier. The processor continues to accept data until it receives the data doubleword tagged as the last one.

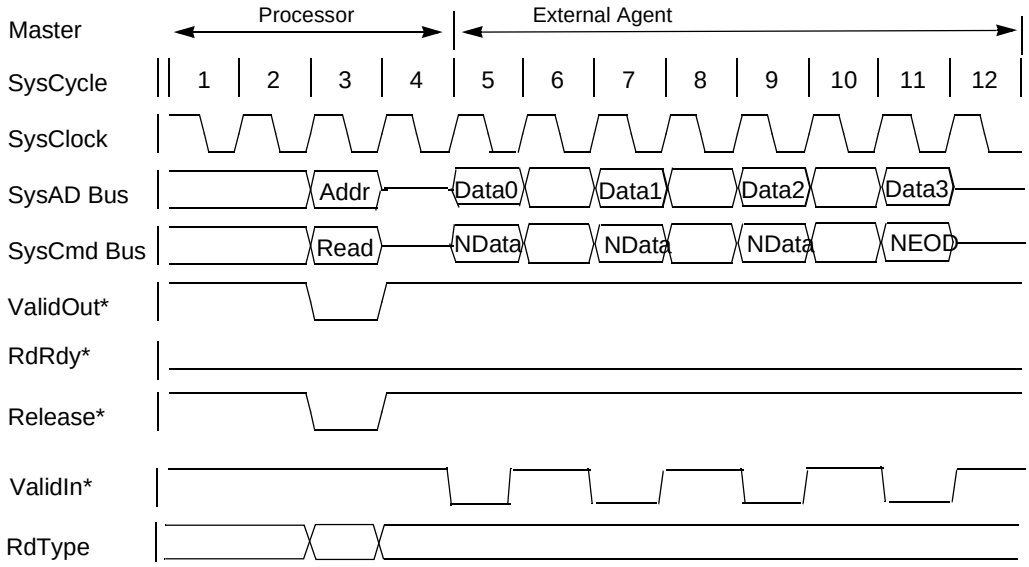


Figure 12.36 External Agent controlling Processor Read Data Flow by deasserting ValidIn*

Figure 12.36 shows how the External Agent can introduce wait states by deasserting the **ValidIn*** pin.

For *Processor Write Requests*, the data flow is controlled by the processor write data transfer pattern. This pattern is set during power-up initialization through the serial mode-bit stream. Modebits[4:1] are used to control the processor write data flow pattern.

12.17 Write Data Transfer Patterns

A data pattern is a sequence of letters indicating the *double-word-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 12.2: lists the maximum processor data rate for each of the possible block write modes that may be specified at boot time for a 64-bit interface.

Table 12.2: Transmit Data Rates and Patterns

Maximum Data Rate	Data Pattern
1 Double/1 Cycle	DDDD
2 Doubles/3 Cycles	DDxDD
1 Double/2 Cycles	DDxxDDxx
1 Double/2 Cycles	DxDxDxDx
2 Doubles/5 Cycles	DDxxxDDxxx
1 Double/3 Cycles	DDxxxxDDxxxx
1 Double/3 Cycles	DxxDxxDxxDxx
1 Double/4 Cycles	DDxxxxxxDDxxxxxx
1 Double/4 Cycles	DxxxDxxxDxxxDxxx

In Table 12.2:, data patterns are specified using the letters **D** and **x**; **D** indicates a double-word-data cycle and **x** indicates an unused cycle.

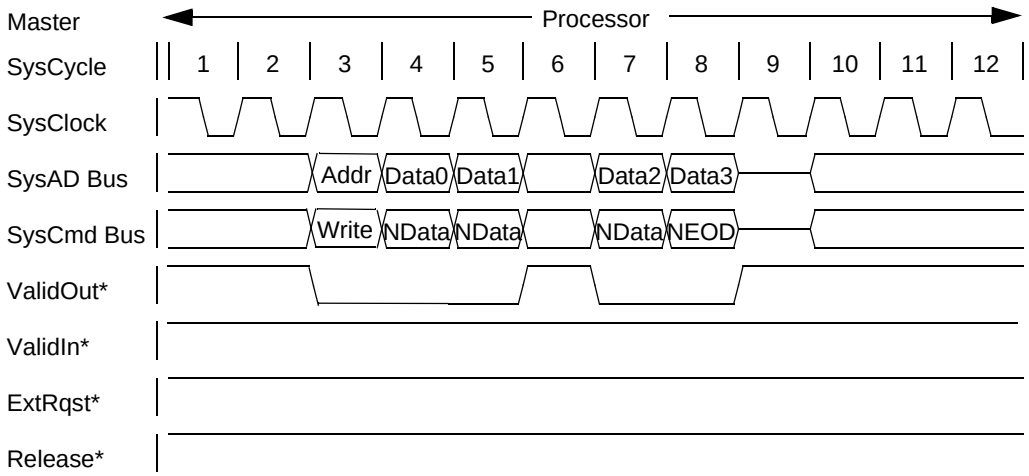


Figure 12.37 Write with Reduced Data Rate DDx

Figure 12.37 shows a write in which data is provided to the external agent at a rate of two doublewords every three cycles using the data pattern **DDx**.

12.18 Independent Transmissions on the SysAD Bus

In most applications, the **SysAD** bus is a point-to-point connection, running from the processor to a bidirectional registered transceiver residing in an external agent. For these applications, the **SysAD** bus has only two possible drivers, the processor or the external agent.

Certain applications may require connection of additional drivers and receivers to the **SysAD** bus, allowing transmissions over the **SysAD** bus that the processor is not involved in. These are called *independent transmissions*. To effect an independent transmission, the external agent must control mastership of the **SysAD** bus by using arbitration handshake signals and external null requests.

An independent transmission on the **SysAD** bus follows this procedure:

1. The external agent asserts **ExtRqst*** to requests mastership of the **SysAD** bus.
2. The processor does a compelled change to slave state and signals this by asserting **Release*** for one cycle.

3. The external agent then allows the independent transmission to take place on the **SysAD** bus, making sure that **ValidIn*** is not asserted while the transmission is occurring.
4. When the independent transmission is complete, the external agent issues a *system interface release external null request* with **ValidIn*** asserted to return the system interface to master state.

12.19 System Interface Endianness

The endianness of the system interface is programmed at boot time through the boot-time mode control interface and the **BigEndian** pin. The **BigEndian** pin allows the system to change the processor addressing mode without changing the mode ROM. If endianness is specified via the **BigEndian** pin, mode ROM bit 8 is cleared (set to zero). If endianness is specified by the mode ROM, the **BigEndian** pin should be grounded. 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.

12.20 System Interface Cycle Counts

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 transactions.

12.21 Release Latency

Release latency is defined as the number of cycles the processor may 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:

1. Category 1: when the external request signal is asserted two cycles before the last cycle of a processor request.
2. 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.
3. Category 3: when the processor makes an uncompelled change to slave state.

Table 12.3: summarizes the minimum and maximum release latencies for requests that fall into categories 1, 2, and 3. Note that the maximum and minimum cycle count values are subject to change.

Table 12.3: Release Latency for External Requests

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

12.22 System Interface Commands/Data Identifiers

System interface commands specify the nature and attributes of any system interface request; this specification is made during the address cycle for the request. System interface data identifiers specify the attributes of data transmitted during a system interface data cycle.

The following sections describe the syntax, that is, the bitwise encoding of system interface commands and data identifiers.

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.

12.22.1 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.

12.22.2 System Interface Command Syntax

This section describes the **SysCmd** bus encoding for system interface commands. Figure 12.38 shows a common encoding used for all system interface commands.

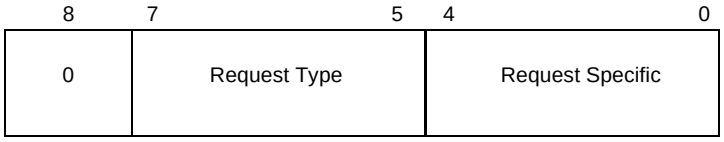


Figure 12.38 System Interface Command Syntax Bit Definition

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 12.4: shows the types of requests encoded by the **SysCmd(7:5)** bits.

SysCmd(4:0) are specific to each type of request and are defined in each of the following sections.

Table 12.4: Encoding of SysCmd(7:5) for System Interface Commands

SysCmd(7:5)	Command
0	Read Request
1	Reserved
2	Write Request
3	Null Request
4-7	Reserved

12.22.2.1 Read Requests

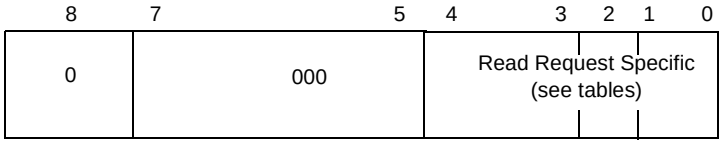


Figure 12.39 Read Request SysCmd Bus Bit Definition

Figure 12.39 shows the format of a **SysCmd** read request.

Tables 12.5 through 12.7 list the encodings of **SysCmd(4:0)** for read requests.

Table 12.5: Encoding of SysCmd(4:3) for Read Requests

SysCmd(4:3)	Read Attributes
0-1	Reserved
2	Noncoherent block read
3	Doubleword, partial doubleword, word, or partial word

Table 12.6: Encoding of SysCmd(2:0) for Block Read Request

SysCmd(2)	Reserved (Not Used)

SysCmd(1:0)	SysCmd(1:0)
0	Reserved
1	8 words
2-3	Reserved

Table 12.7: Encoding of SysCmd(2:0) Read Request Data Size

SysCmd(2:0)	Read Data Size
0	1 byte valid (Byte)
1	2 bytes valid (Halfword)
2	3 bytes valid (Tribyte)
3	4 bytes valid (Word)
4	5 bytes valid (Quintibyte)
5	6 bytes valid (Sextibyte)
6	7 bytes valid (Septibyte)
7	8 bytes valid (Doubleword)

12.22.2.2 Write Requests

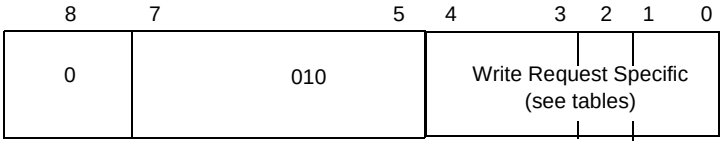


Figure 12.40 Write Request SysCmd Bus Bit Definition

Figure 12.40 shows the format of a **SysCmd** write request.

Table 12.8: lists the write attributes encoded in bits **SysCmd(4:3)**. Table 12.9: lists the block write replacement attributes encoded in bits **SysCmd(2:0)**. Table 12.10: lists the write request bit encodings in **SysCmd(2:0)**.

Table 12.8: Encoding of SysCmd(4:3) for Write Requests

SysCmd(4:3)	Write Attributes
0	Reserved
1	Reserved
2	Block write
3	Doubleword, partial doubleword, word, or partial word

Table 12.9: Encoding of SysCmd(2:0) for Block Write Requests

SysCmd(2)	Reserved

SysCmd(1:0)	Write Block Size
0	Reserved
1	8 words
2-3	Reserved

Table 12.10:Encoding of SysCmd(2:0) for Write Request Data Size

SysCmd(2:0)	Write Data Size
0	1 byte valid (Byte)
1	2 bytes valid (Halfword)
2	3 bytes valid (Tribyte)
3	4 bytes valid (Word)
4	5 bytes valid (Quintibyte)
5	6 bytes valid (Sextibyte)
6	7 bytes valid (Septibyte)
7	8 bytes valid (Doubleword)

12.22.2.3 Null Requests

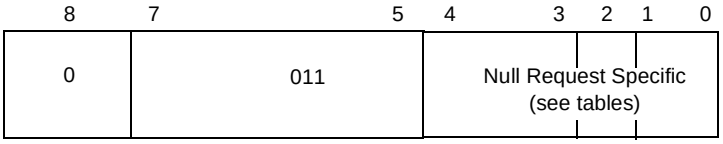


Figure 12.41 Null Request SysCmd Bus Bit Definition

Figure 12.41 shows the format of a **SysCmd** null request. System interface release external null requests use the null request command. Table 12.11: lists the encodings of **SysCmd(4:3)** for external null requests. **SysCmd(2:0)** are reserved for null requests.

Table 12.11:External Null Request Encoding of SysCmd(4:3)

SysCmd(4:3)	Null Attributes
0	System Interface release
1-3	Reserved

12.22.3 System Interface Command Identifier Summary

Below is a summary of the command identifiers that can appear on the system interface bus. Pins that are not used by a specific encoding are denoted by X.

Table 12.12:System Command Bus Identifiers

SysCmd[8:0] Encoding									Command Mnemonic	Command Description
8	7	6	5	4	3	2	1	0		
0	0	0	0	1	1	0	0	0	RdByte	Read a single byte
0	0	0	0	1	1	0	0	1	RdHalf	Read 2 bytes
0	0	0	0	1	1	0	1	0	RdTriByte	Read 3 bytes
0	0	0	0	1	1	0	1	1	RdWord	Read 4 bytes (Word)
0	0	0	0	1	1	1	0	0	RdQuintiByte	Read 5 bytes
0	0	0	0	1	1	1	0	1	RdSextiByte	Read 6 bytes
0	0	0	0	1	1	1	1	0	RdSeptiByte	Read 7 bytes
0	0	0	0	1	1	1	1	1	RdDoubleWord	Read 8 bytes (doubleword)
0	0	0	0	1	0	X	0	1	RdBlock	Read 32 bytes (cache line)
0	0	1	0	1	1	0	0	0	WrByte	Write a single byte
0	0	1	0	1	1	0	0	1	WrHalf	Write 2 bytes
0	0	1	0	1	1	0	1	0	WrTriByte	Write 3 bytes
0	0	1	0	1	1	0	1	1	WrWord	Write 4 bytes (word)
0	0	1	0	1	1	1	0	0	WrQuintiByte	Write 5 bytes
0	0	1	0	1	1	1	0	1	WrSextiByte	Write 6 bytes
0	0	1	0	1	1	1	1	0	WrSeptiByte	Write 7 bytes
0	0	1	0	1	1	1	1	1	WrDoubleWord	Write 8 bytes (doubleword)
0	0	1	0	1	0	X	0	1	WrBlock	Write 32 bytes (cache line)
0	0	1	1	0	0	X	X	X	NullReq	Null Request Command

12.22.4 System Interface Data Identifier Syntax

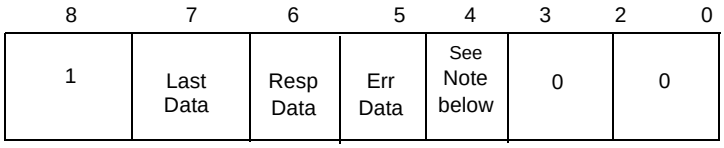


Figure 12.42 Data Identifier SysCmd Bus Bit Definition

This section defines the encoding of the **SysCmd** bus for system interface data identifiers. Figure 12.42 shows a common encoding used for all system interface data identifiers. **SysCmd(8)** must be set to 1 for all system interface data identifiers.

Note: **SysCmd(4)** *is reserved for a processor data identifier. For an external data identifier,* **SysCmd(4)** *indicates whether or not to check the data and check bits for an error.*

12.22.4.1 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.

12.22.4.2 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. In the case of a block response, the entire line must be delivered to the processor no matter how minimal the error. Note that the processor only checks **SysCmd(5)** during the first doubleword of a block read response.

SysCmd(4) indicates to the processor whether to check the data and check bits for this data element, for both coherent and noncoherent external data identifiers. **SysCmd(4)** is reserved for processor data identifiers.

SysCmd(3) is reserved for all data identifiers.

SysCmd(2:0) are reserved for non-coherent and external data identifiers.

Table 12.13: lists the encodings of **SysCmd(7:3)** for processor data identifiers. Table 12.14: lists the encodings of **SysCmd(7:3)** for external data identifiers.

Table 12.13:Processor Data Identifier Encoding of SysCmd(7:3)

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 12.14:External Data Identifier Encoding of SysCmd(7:3)

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)	Last Data Element Indication
0	Check the data and check bits
1	Do not check the data and check bits

SysCmd(3)	Reserved

12.23 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. Virtual address bits **VA[13:12]** can be aliased by the on-chip cache, so they appear on **SysAD[57:56]**. The remaining bits of the **SysAD** bus are unused during address cycles.

12.23.1 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 the address are 0.
- Doubleword requests set the low-order 3 bits of the address to 0.
- Word requests set the low-order 2 bits of the address to 0.
- Halfword requests set the low-order bit of the address to 0.
- Byte, tribyte, quintibyte, sextibyte, and septibyte requests use the full byte address.

12.23.2 Subblock Ordering

The order in which data is returned in response to a processor block read request is called subblock ordering. In subblock ordering, the processor delivers the address of the requested doubleword within the block. An external agent must return the block of data using subblock ordering, starting with the addressed doubleword.

For block write requests, 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 the block.

During data cycles, 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 12.15: lists the byte lanes used for partial word transfers for both big and little endian.

Table 12.15:Partial Word Transfer Byte Lane Usage

# Bytes SysCmd[2:0]	Address Mod 8	SysAD byte lanes used (Big Endian							
		63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
1 (000)	0	X							
	1		X						
	2			X					
	3				X				
	4					X			
	5						X		
	6							X	
	7								X
2 (001)	0	X	X						
	2			X	X				
	4					X	X		
	6							X	X
3 (010)	0	X	X	X					
	1		X	X	X				
	4					X	X	X	
	5						X	X	X
4 (011)	0	X	X	X	X				
	4					X	X	X	X
5 (100)	0	X	X	X	X	X			
	3				X	X	X	X	X
6 (101)	0	X	X	X	X	X	X		
	2			X	X	X	X	X	X
		7:0	15:8	23:16	31:24	39:32	47:40	55:48	63:56
		SysAD byte lanes used (Little Endian)							

# Bytes SysCmd[2:0]	Address Mod 8	SysAD byte lanes used (Big Endian							
		63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
7 (110)	0	X	X	X	X	X	X	X	
	1		X	X	X	X	X	X	X
8 (111)	0	X	X	X	X	X	X	X	X
		7:0	15:8	23:16	31:24	39:32	47:40	55:48	63:56
		SysAD byte lanes used (Little Endian)							

12.23.3 Processor Internal Address Map

External writes provide access to processor internal resources that may be of interest to an external agent. The processor decodes bits **SysAD(6:4)** of the address associated with an external write request to determine which processor internal resource is the target. The *Interrupt* register is the only processor internal resource available for an external write 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.

12.24 Tertiary Cache Mode Configuration

The existence of a tertiary cache is signaled to the processor by setting ROM mode serial bit [12] to a one, and is reflected in CP0 *Config* register ***TC*** bit [17]. When the ***TC*** bit = 0, the tertiary cache is present.

The tertiary cache may be disabled using the CP0 *Config* register ***TE*** bit [12]. When the ***TE*** bit = 1 (set) the tertiary cache is enabled. The ***TE*** bit is cleared at reset. When the tertiary cache is enabled by setting the ***TE*** bit, the state of the cache is undefined and software must explicitly invalidate the entire tertiary cache before using it.

If no tertiary cache is present, or the tertiary cache is disabled, the processor drives all tertiary cache signals to their inactive state. If no tertiary cache is present the **TcMatch** and **TcDOE*** signals become don't care inputs and must be connected to valid logic levels. If the tertiary cache is present and enabled, then the **SysADC** signals must carry valid parity during block read responses.

The doublewords transferred on **SysAD** during tertiary cache block read transactions are in sub-block order. The doublewords transferred on **SysAD** during tertiary cache block write transactions are in sequential order.

The tertiary cache RAM type is specified by the processor ROM mode serial bit [15]. Bit [15] = 1 indicates single cycle deselect timing; bit [15] = 0 indicates dual cycle deselect timing. The difference between these two settings is in the timing of the signal **TcDCE*** on a tertiary cache read hit. On a tertiary cache read hit **TcDCE*** is asserted at cycle T1. If bit [15] = 0, **TcDCE*** is deasserted at cycle T5. If bit [15] = 1, **TcDCE*** is deasserted at cycle T6, one cycle later. This allows for the use of commodity SRAMs in the system design.

The size of the tertiary cache is set by the system design. The size can be from 512KB to 8MB in powers of two. It's size may be indicated by processor mode ROM serial bits [17:16]. The system designer is free to choose the encoding.The state of these mode bits appear as CP0 *Config* register bits [21:20].

12.25 Tertiary Cache Synchronous SRAMs

The RM7000 supports the use of pipelined synchronous SRAMs for the data RAMs and tag RAMs. All synchronous SRAMs are pipelined and allow an access every cycle. The following sub-sections discuss the internal layout and timing diagrams for the basic data and tag RAMs.

12.25.1 Tertiary Cache Data RAMs

The data RAMs are pipelined synchronous SRAMs (SSRAMs) with registered inputs and outputs. The chip enable and write enable signals are pipelined. The output enable signal is asynchronous. Figure 12.43 shows a block diagram of the data RAM.

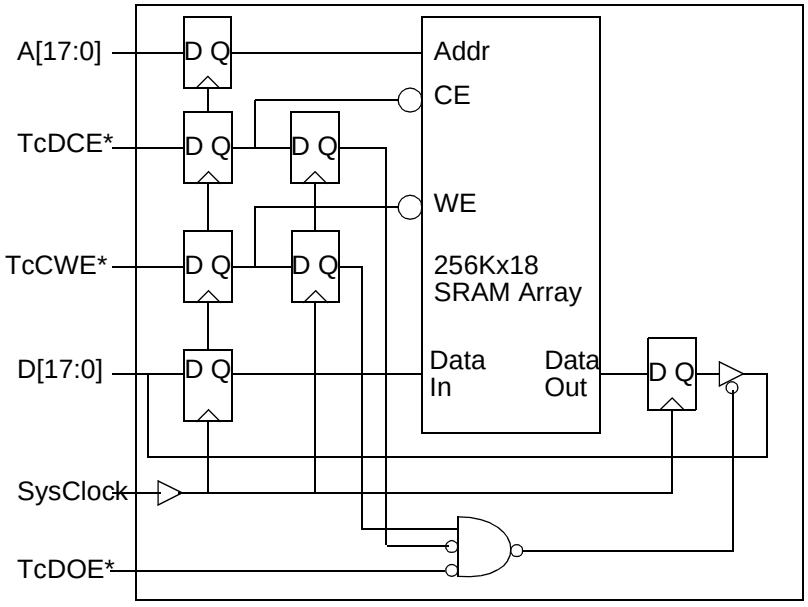


Figure 12.43 Data RAM Block Diagram

As illustrated in the above block diagrams, the RAMs synchronously enable their outputs two cycles after a read operation is issued, and synchronously disable their outputs two cycles after the end of a read operation.

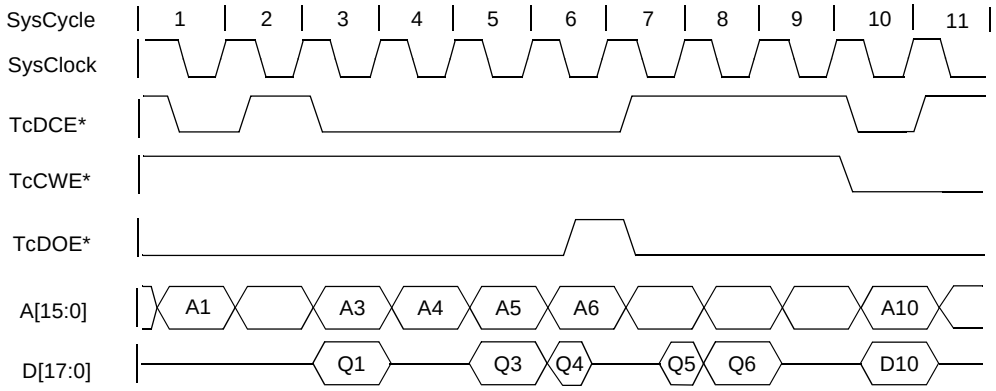


Figure 12.44 Data RAM Read Cycles Followed By a Write Cycle

Figure 12.44 shows a timing diagram of a data RAM read followed by write cycle, while Figure 12.45 shows a diagram of a write followed by a read cycle.

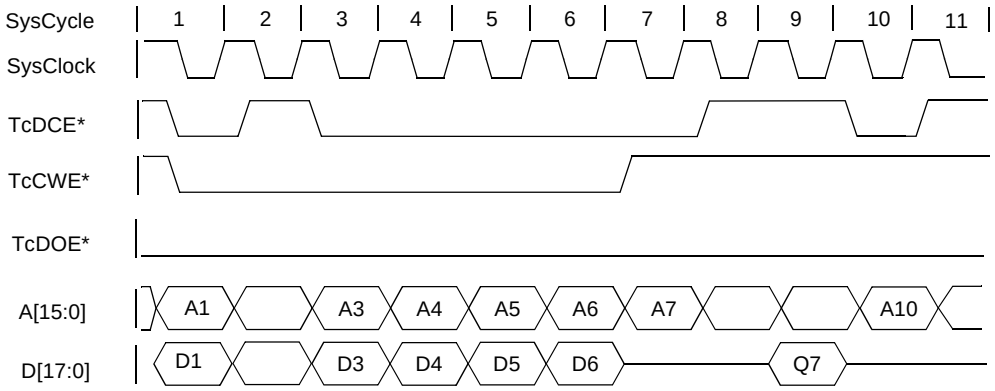


Figure 12.45 Data RAM Write Cycles Followed By a Read Cycle

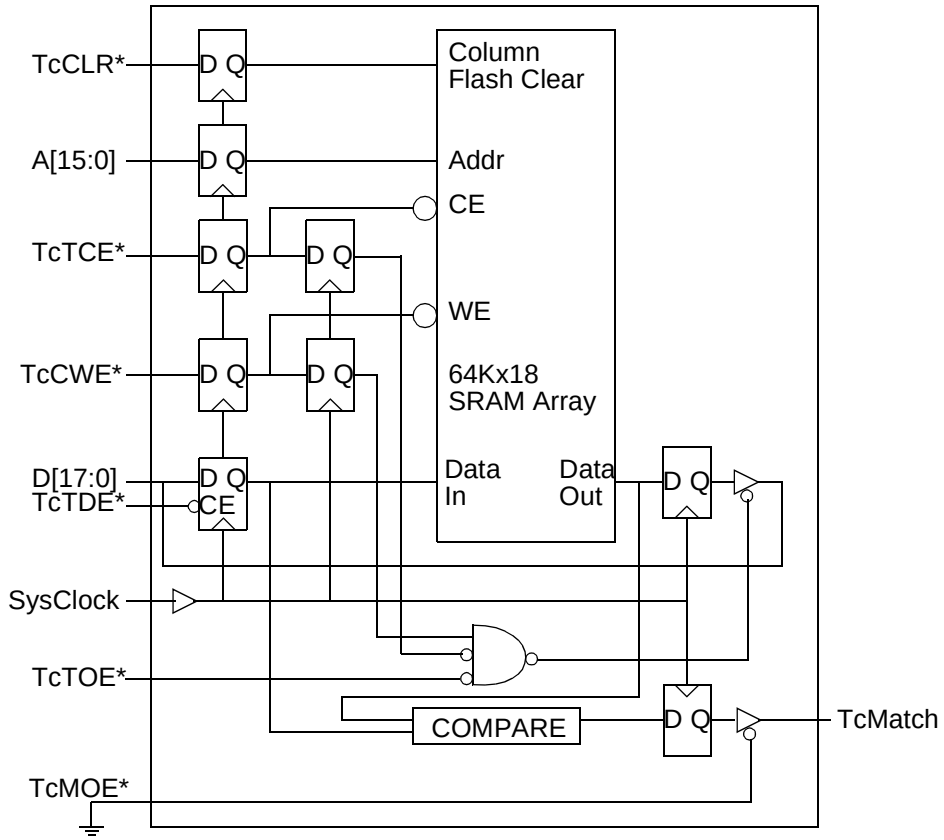


Figure 12.46 Tag RAM Block Diagram

The tag RAM has the same architecture as the data RAM with the addition of a load enable signal for the data input register and a registered comparator output of the data input register and the RAM array. The tag RAM may optionally support a flash clear of the valid bit column. Figure 12.46 shows a block diagram of the tag RAM.

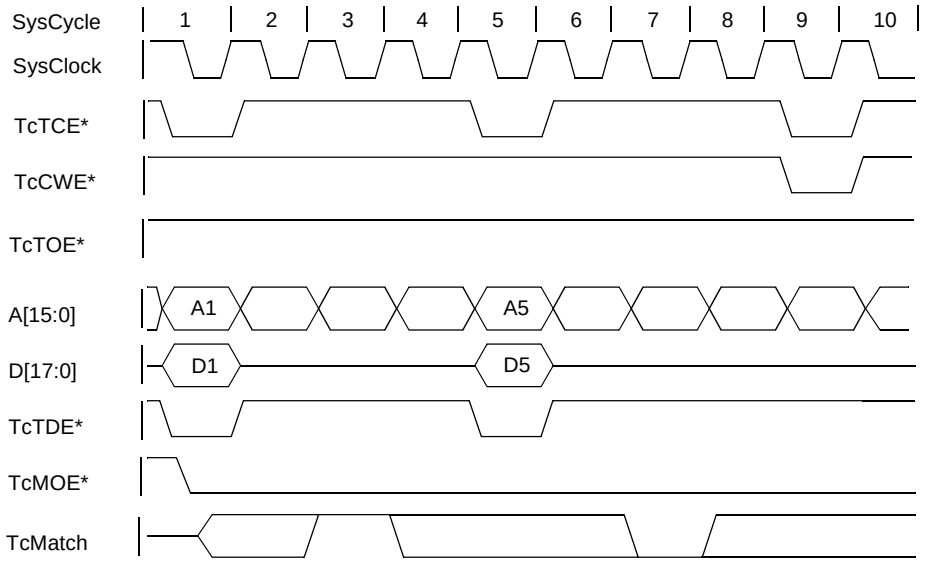


Figure 12.47 Tag RAM Hit and Miss Read-Followed-By-Write Cycles

Figure 12.47 shows a timing diagram of hit and miss read followed by write cycles. Figure 12.48 shows a timing diagram of basic read and write cycles.

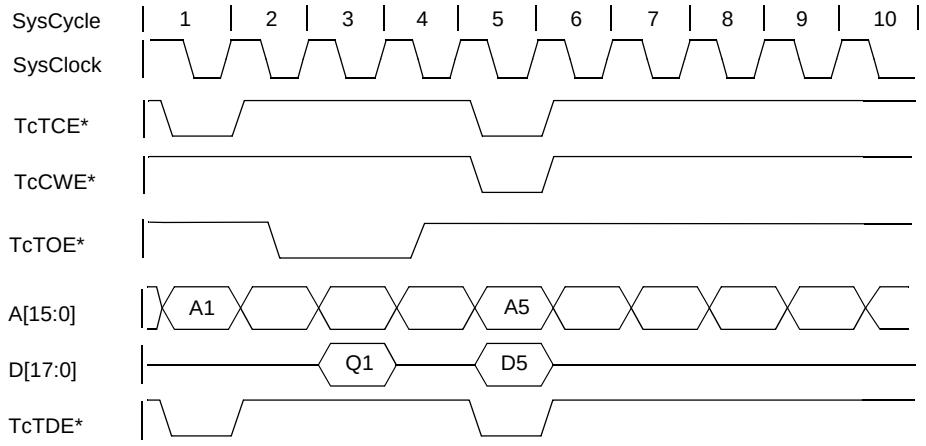


Figure 12.48 Tag RAM Read and Write Cycles

12.26 Error Checking

The RM7000 supports parity for both internal cache and bus interface transactions. Bus errors are specific to system interface transactions. Parity errors use the cache error exception vector located at address 0x100. Bus errors use the general exception vector located at address 0x180.

12.27 Parity Errors

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 are shown in the example below:

Data(3:0)	Odd Parity Bit	Even Parity Bit
0010	0	1

The example above 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)**.

The example below shows odd and even parity bits for various data values:

Data(3:0)	Odd Parity Bit	Even Parity Bit
0110	1	0
0000	1	0
1111	1	0
1101	0	1

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 as it has an even number of 1s. However, it is impossible to tell *which* bit is in error.

The RM7000 supports both precise and imprecise parity errors. A parity error is considered precise when the exact address of the instruction that caused the error has been captured by the CPU in the *Error_EPC* register. An imprecise parity error means that the CPU was not able to capture the exact address of the instruction that caused the error as the CPU has already executed other instructions past the one which caused the error.

Parity errors are checked at the output of the primary data and instruction cache arrays for cache ‘hits’. These parity checkers check load data and instructions. Data array parity is not checked on primary and secondary cache writebacks.

Parity is also checked at the output of the instruction, data, and secondary cache tags. If a primary data cache parity error occurs on a writeback, the bad data is written to the secondary cache or tertiary cache/main memory depending on the type of cache attributes.

All parity errors take the cache error exception vector (0x100).

12.27.1 Precise Parity Errors

All instruction-related parity errors are precise, whereas data-related parity errors are imprecise. Precise parity errors occur when a parity error is detected on an:

- Instruction cache hit
- Uncached instruction fetch

Instruction cache hit parity errors update the index field of the *CacheErr* register. For the precise parity errors the *CacheErr* register ***Index*** field does contain the correct value.

12.27.2 Imprecise Parity Errors

All data-related parity errors are imprecise, meaning that the CPU was not able to capture the exact address of the instruction that caused the error, rather the error is associated to a range of addresses. Imprecise parity errors take the cache error exception vector (0x100). These errors occur when the processor detects a parity error on:

- Any non-blocking primary data cache miss (prefetch, load, store) from either the secondary cache or the system interface.
- A data cache hit.
- An uncached data fetch.
- A secondary cache tag error on a primary cache writeback.

For the imprecise parity errors the *CacheErr* register ***Index*** field does not contain the correct value.

The RM7000 includes two new registers in CP0 register set 1 for reporting the address locations on imprecise parity errors. These new registers are called *DErrAddr0* and *DErrAddr1*. Figure 12.49 shows the formats for these two registers.

For imprecise parity errors, the bad physical address is contained in one of the *DErrAddr* address registers. The data physical address that generated the exception is indicated by the ***E0*** and ***E1*** bits of the *CacheErr* register. Note that these registers are set for bus errors as well as parity errors.

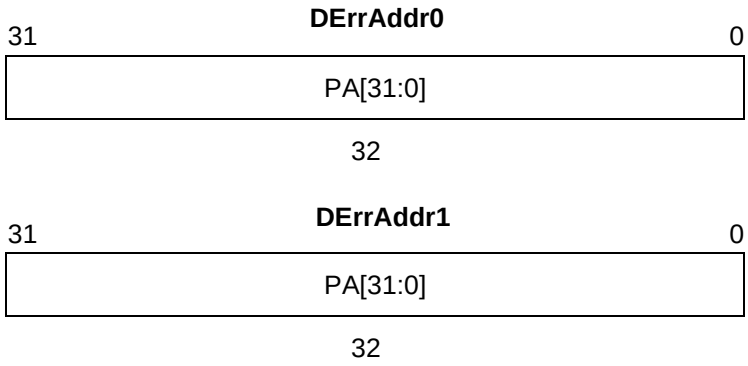


Figure 12.49 Imprecise Parity Error Registers

In Figure 12.49, PA[31:0] indicates the address that caused the imprecise parity error or bus error.

12.28 Bus Errors

The RM7000 supports both precise and imprecise bus errors. A bus error is considered precise when the exact address of the instruction that caused the error has been captured by the CPU in the *EPC* register. An imprecise bus error means that the CPU was not able to capture the exact address of the instruction that caused the error as the CPU has already executed other instructions past the one which caused the error. All instruction-related bus errors are precise, whereas data-related bus errors can be either precise or imprecise. Bus errors are generated only on the system interface and must be reported on the first doubleword received on the system interface. A bus error reported after the first doubleword will be ignored.

Bus errors always take the general exception vector (0x180).

12.28.1 Precise Bus Errors

All instruction-related bus errors are precise. Precise bus errors occur when the system reports an error on an:

- Instruction cache miss
- Uncached instruction fetch
- Blocking data fetch

For precise bus errors the *CacheErr* register index field does store the correct value.

12.28.2 Imprecise Bus Errors

Data-related bus errors can be either precise or imprecise. An imprecise bus error means that the CPU was not able to capture the exact address of the instruction that caused the error as the CPU has already executed other instructions past the one which caused the error. Imprecise bus errors take the general exception vector located at address 0x180. These errors occur when the processor detects a bus error on:

- Any non-blocking primary data cache load miss.
- An uncached non-blocking access

An imprecise bus error is denoted when the ***EC*** bit in the *CacheErr* register is set.

For the imprecise buserrors the *CacheErr* register index field does not store the correct value.

The RM7000 includes two new registers in CP0 register set 1 for reporting the address locations on imprecise bus errors. These new registers are called *DErrAddr0* and *DErrAddr1*. Figure 12.49 shows the formats for these two registers.

For imprecise bus errors, the bad physical address is contained in one of the *DErrAddr* address registers. The data physical address that generated the exception is indicated by the ***E0*** and ***E1*** bits of the *CacheErr* register. Note that these registers are set for bus errors as well as parity errors.

12.28.2.1 System Interface

The processor generates correct check bits for doubleword, partial doubleword, word, or partial-word data transmitted to the System. Interface. For block data, 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 **SysCmd[4]** bit in the data identifier, it is possible to prevent the processor from checking read response data from the system interface.

The processor does not check addresses received from the system interface and does not generate check bits for addresses transmitted to the system interface.

The processor takes a cache error exception when it detects an error based on data check bits. Software is responsible for error handling.

12.28.2.2 System Interface Command Bus

In the RM7000 processor, the system interface command bus has a single parity bit, **SysCmdP**, that provides even parity over the 9 bits of this bus. The **SysCmdP** parity bit is not implemented. It is not generated when the system interface is in master state and is not checked when the system interface is in slave state. This signal is defined to maintain R4000 compatibility and is not functional in the RM7000 processor.

12.28.2.3 Summary of System Interface Error Checking Operations

Error checking operations at the System Interface performed by the processor are summarized in tables 12.16 and 12.17.

Table 12.16:Error Checking and Generation Summary for Internal Transactions on the SysAD Bus Interface

Bus	Uncached Load	Uncached Store	Cache Load from System Interface	Cache Write to System Interface	Cache Instruction
SysAD Address and Address Check bits; Transmit	Check bits Not Generate (always driven low)	Check bits Not Generated (always driven low)	Check bits Not Generated (always driven low)	Check bits Not Generated (always driven low)	Check bits Not Generated (always driven low)
SysCmd Command and Command Check bits; Transmit	Check bit Not Generated (always driven low)	Check bit Not Generated (always driven low)	Check bits Not Generated (always driven low)	Check bit Not Generated (always driven low)	Check bit Not Generated (always driven low)
SysCmd Data Identifier, and Identifier Check Bits; Transmit	Not Checked	Check bit Not Generated (always driven low)	Not Checked	Check bit Not Generated (always driven low)	Not Checked
SysAD Data and Data Check Bits; Transmit	NA	Data From register. Check bits generated	NA	Data and Check bits from cache, Not Checked.	Data and Check bits from cache, Not Checked
SysCmd Data Identifier, and Identifier Check Bits; Receive	Not Checked	NA	Not Checked	NA	Not Checked
SysAD Data and Data Check Bits; Receive	Checked, Trap on error	NA	Checked on requested doubleword, Trap on error	NA	Checked on requested doubleword, Trap on error

Table 12.17:Error Checking and Generation Summary for External Transactions on the System Interface.

Bus	External Write
SysAD Address, and Address Check bits; Receive	Not Checked
SysCmd Command and Command Check bits; Receive	Not Checked
SysCmd Data Identifiers and Identifier Check Bits; Receive	Not Checked
SysAD Data and Data Check Bits; Receive	Not Checked

Section 13 Exception Processing

This section describes the exception processing mechanism in the RM7000 processor, including the format and use of each CPU exception register.

13.1 Overview of Exception Processing

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 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.

The registers described later in the section assist in this exception processing by retaining address, cause and status information.

13.2 Exception Processing Registers

This section describes the CP0 registers that are used in exception processing. Table 13.1: lists these registers, along with their number—each register has a unique identification number that is referred to as its *register number.* For instance, the *ECC* register is register number 26. The remaining CP0 registers are used in memory management.

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 13.1: are used in exception processing, and are described in the sections that follow. Set 0 registers are accessed with **MFC0** and **MTC0** instructions.

Table 13.1: CP0 Set 0 Exception Processing Registers

Reg. No.	Register Name
4	Context
8	BadVAddr (Bad Virtual Address)
9	Count
11	Compare register
12	Status
13	Cause
14	EPC (Exception Program Counter)
18	Watch1
19	Watch2
20	XContext
22	PerfControl (Performance counter control)
24	WatchMask
25	PerfCount (Performance counter)
26	ECC (Error Checking and Correction)
27	CacheErr (Cache Error and Status)
30	ErrorPC (Error Exception Program Counter

Table 13.2: lists the CP0 Set 1 exception processing registers. Set 1 registers are accessed with **CFC0** and **CTC0** instructions. The interrupt registers are described in Section 14, “Interrupts”, of this manual.

Table 13.2: CP0 Set 1 Exception Processing Registers

Reg. No.	Register Name
18	IPLLo (Interrupt Priority Level Lo)
19	IPLHi (Interrupt Priority Level Hi)
20	IntControl (Interrupt Control)
26	DErrAddr0 (Data Error Address 0)
27	DErrAddr1 (Data Error Address 1)

The CPU’s general purpose 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. The 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.

13.2.1 Context Register (Set 0, 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 operating system 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 13.1 shows the format of the *Context* register; Table 13.3: describes the *Context* register fields.

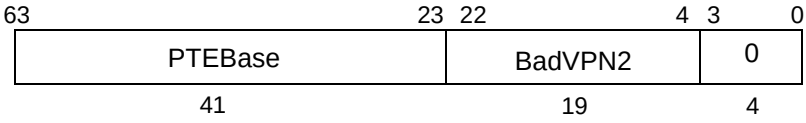


Figure 13.1 Context Register Format

Table 13.3: Context Register Fields

Bit(s)	Field	Description
63:23	PTEBase	This field is a read/write field for use by the operating system. It is normally written with a value that allows the operating system to use the *Context* register as a pointer into the current PTE array in memory.
22:4	BadVPN2	This field is written by hardware on a miss. It contains the virtual page number (VPN) of the most recent virtual address that did not have a valid translation.

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.

13.2.2 Bad Virtual Address Register (BadVAddr, Set 0, Register 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, or Address Error.

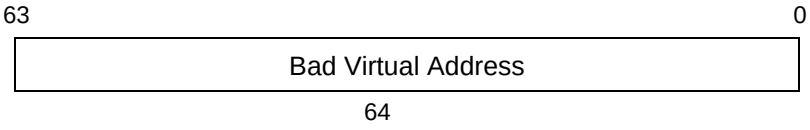


Figure 13.2 BadVAddr Register Format

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

Figure 13.2 shows the format of the *BadVAddr* register.

13.2.3 Count Register (Set 0, Register 9)

The *Count* register is a read/write register which acts as a timer, incrementing at a constant rate, at the CPU pipeline frequency, whether or not an instruction is executed, retired, or any forward progress is made through the pipeline.

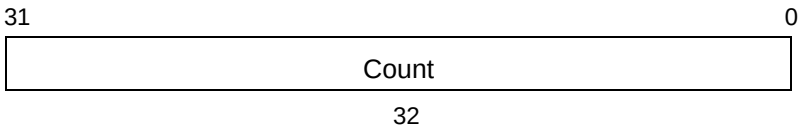


Figure 13.3 Count Register Format

Figure 13.3 shows the format of the *Count* register.

13.2.4 Compare Register (Set 0, Register 11)

The *Compare* register maintains a stable value that does not change on its own.

When the value of the *Count* register equals the value of the *Compare* register, an interrupt bit in the *Cause* register can be set. If mode bit 11 is 0, then the timer interrupt is selected to be stored in ***IP7*** and masked by ***IM7***. If mode bit 11 is 1, then **Int*5** is selected to be stored in ***IP7*** and masked by ***IM7***. If ***TE***=1 when the *Count* register equals the *Compare* register, then ***IP12*** is set. If ***TE***=0, then ***IP12*** is not set. This causes an interrupt as soon as the interrupt is enabled. To clear the timer interrupt, a value must be written to the *Compare* register.

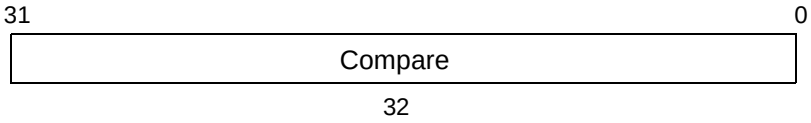


Figure 13.4 Compare Register Format

The *Compare* register is a read/write register. In normal use however, the *Compare* register is only written. Figure 13.4 shows the format of the *Compare* register.

13.2.5 Status Register (Set 0, 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. The following list describes the more important *Status* register fields.

- 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]**. (***IM7*** enables the Timer Interrrupt on ***IP7*** unless mode bit 11 is set to enable **Int*5** on ***IP7***.) **Int*[9:6]** are enabled in the *IntControl* register. Please refer to Section 14, “Interrupts”, to use **Int*[9:6]** and to enable the prioritized, vectorized interrupt capability.
- The 3-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.

13.2.5.1 Status Register Format

Figure 13.5 shows the format of the *Status* register. Table 13.4: describes the *Status* register fields. Figure 13.6 and Table 13.5: provide additional information on the ***Diagnostic Status*** (***DS***) field. All bits in the ***DS*** field except ***TS*** are readable and writable.

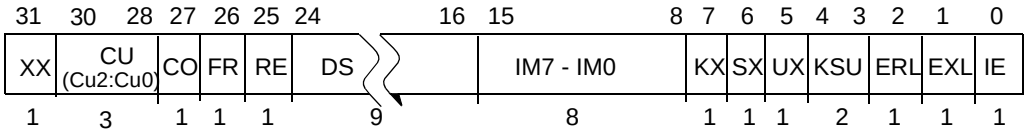


Figure 13.5 Status Register

Table 13.4: Status Register Fields

Bit(s)	Field	Description
31	XX	Enables execution of MIPS IV instructions in user-mode 1: MIPS IV instructions usable 0: MIPS IV instructions unusable
30:28	CU	Controls the usability of each of the three coprocessor unit numbers. CP0 is always usable when in Kernel mode, regardless of the setting of the CU_0 bit. 1: usable 0: unusable
27	CO	Setting this bit enables use of the Hit_Writeback and Hit-Writeback_Invalidate CACHE instruction while in user mode. Clearing this bit disables use of these instructions while in user mode.
26	FR	Enables additional floating-point registers 0:16 registers 1:32 registers
25	RE	*Reverse-Endian* bit, valid in User mode.
24:16	DS	*Diagnostic Status* field (see Figure 13.6).
15:8	IM[7:0]	*Interrupt Mask* (Lower): controls the enabling of eight of the external, internal, and software interrupts. An interrupt is taken if interrupts are enabled, and the corresponding bits are set in both the *Interrupt Mask* field of the *Status* register and the *Interrupt Pending* field of the *Cause* register. IM[7:2] correspond to Int*5 through Int*0. IM7 also enables the Timer Interrrupt if mode bit 11 is 0. IM[1:0] correspond to SWINT1 and SWINT0. (Int*[9:6] are enabled in the IntControl register). 0: disabled 1: enabled
7	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
6	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
5	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
4:3	KSU	Mode bits 10: User 01: Supervisor 00: Kernel
2	ERL	Error Level; set by the processor when Reset, Soft Reset, NMI, or Cache Error exception are taken. 0: normal 1: error When ERL is set: — Interrupts are disabled. — The ERET instruction uses the return address held in ErrorEPC instead of EPC. — Kuseg and xkuseg are treated as unmapped and uncached regions.This allows main memory to be accessed in the presence of cache errors.

Bit(s)	Field	Description
1	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 When EXL is set: — Interrupts are disabled. — TLB refill exceptions will use the general exception vector instead of the TLB refill vector. — EPC is not updated if another exception is taken.
0	IE	Interrupt Enable 0: disable interrupts 1: enables interrupts

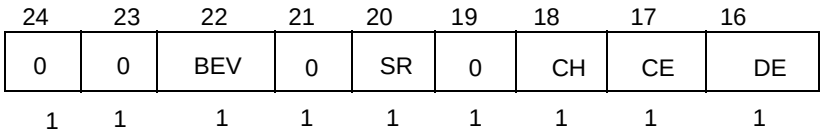


Figure 13.6 Status Register Diagnostic Status Field

Table 13.5: Status Register Diagnostic Status Bits

Bit(s)	Field	Description
22	BEV	Controls the location of TLB refill and general exception vectors. 0: normal 1: bootstrap
20	SR	1: Indicates that a Soft Reset or NMI has occurred.
18	CH	Hit (tag match and valid state) or miss indication for last CACHE Hit Invalidate, Hit Write Back Invalidate, or Hit Write Back for the secondary cache. 0: miss 1: hit
17	CE	Contents of the ECC register set or modify the check bits of the caches when CE = 1; see description of the *ECC* register.
16	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.

13.2.5.2 Status Register Modes and Access States

Fields of the *Status* register set the modes and access states described in the sections that follow.

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

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

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

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

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

32- and 64-bit 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 MIPS III & MIPS IV 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 space is enabled when ***KX*** = 1. 64-bit operations are always valid in Kernel mode.
- 64-bit addressing and operations are enabled for Supervisor space when ***SX*** = 1.
- 64-bit addressing and operations are enabled for User space when ***UX*** = 1.

Kernel Address Space Accesses: Access to the kernel address space is allowed when the processor is in Kernel mode.

Supervisor Address Space Accesses: Access to the supervisor address space is allowed when the processor is in Kernel or Supervisor mode, as described above in the section titled “Operating Modes.”

User Address Space Accesses: Access to the user address space is allowed in any of the three operating modes.

13.2.5.3 Status Register Reset

The contents of the *Status* register are undefined at reset, except for the following bits in the ***Diagnostic Status*** field:

- ***ERL*** and ***BEV*** = 1

The ***SR*** bit distinguishes between the Reset exception and the Soft Reset exception (caused either by **Reset*** or **Nonmaskable Interrupt [NMI]**).

13.2.6 Info Register (Set 0, Register 7)

The *Info* register specifies various configuration options which have been selected at boot-time as well as the particulars of the on-chip cache confiration. It also contains a global interrupt enable/disable bit..

Some configuration options, as defined by *Info register* bits 28:23 are read from the initialization mode stream by the hardware during reset and are included in the *Info* register as read-only status bits for the software to access. The only option writable by softe is the global interrupt enable.disable bit (bit 0, GIE).

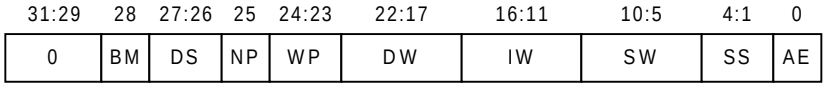


Figure 13.7 Info Register Format

Figure 13.7 shows the format of the *Info* register; Table 13.6: describes the *Info* register fields.

Table 13.6: Info Register Fields

Bits	Field	Description
28	BM	Burst Mode: Read from modebit 15 0 : pipelined Scache RAMS (dual cycle deselect) 1 : burst mode Scache RAMS (single cycle deselect)
27..26	DS	Drive Strength: Drive strength of pad output drivers. Read from modebits 14:13 00 : 67% drive strength 01 : 50% drive strength 10 : 100% drive strength 11 : 83% drive strength

Bits	Field	Description
25	NP	Non-pendant bus mode: Read from modebit 26 0 : overlapping reads disabled 1 : overlapping reads enabled
24..23	WP	Write Protocol: Read from modebits 10:9 00 : R4000 compatible 01 : Reserved 10 : Pipelined writes 11 : Write re-issue
22..17	DW	Primary Data Cache Sets: 000100 : 4 sets all other patterns : reserved
16..11	IW	Primary Instruction Cache Sets: 000100 : 4 sets all other patterns : reserved
10..5	SW	Secondary Cache Sets: 000100 : 4 sets all other patterns : reserved
4..1	SS	Secondary Cache Size: 0010 : 256 Kbytes all other patterns : reserved
0	AE	Atomic Enable: Atomic Interrupt Enable/Disable. This bit allows atomic enable/disable of interrupts without a read-modify-write sequence to the SR.IE bit. Writing a one to this bit globally enables interrupts Writing a zero to this bit globally disables interrupts This bit returns a zero on a read.

13.2.7 Cause Register (Set 0, Register 13)

The 32-bit read/write *Cause* register describes the cause of the most recent exception. The interrupt pending has been expanded from 8 to 16 bits to accommodate the enhanced interrupt mechanism of the RM7000. The expansion of the interrupt pending field caused the ***IV*** bit to be moved from bit 23 to bit 24. The ***W1*** and ***W2*** bits enable the two new *Watch* registers used during device debug and test.

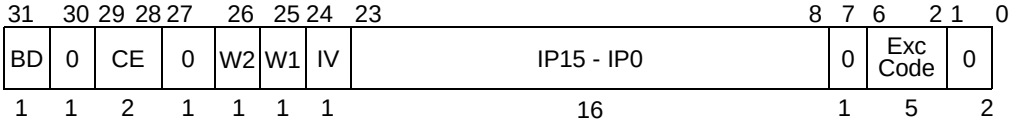


Figure 13.8 Cause Register Format

Figure 13.8 shows the fields of this register. Table 13.7: describes the *Cause* register fields. A 5-bit exception code (***ExcCode***) indicates the cause of the most recent exception, as listed in Table 13.8:.

All bits in the *Cause* register, with the exception of the ***IP(1:0)*** and ***IV*** bits, are read-only. Setting ***IP1*** or ***IP0*** will cause a software interrupt. Setting ***IV*** will enable the RM7000 interrupt extensions and set the interrupt base address vector offset to 0x200. The ***IV*** bit is cleared on Reset.

Table 13.7: Cause Register Field

Bit(s)	Field	Description
31	BD	Indicates whether the last exception taken occurred in a branch delay slot. 1: delay slot 0: normal
29:28	CE	This 2-bit field contains the coprocessor unit number and referenced when a Coprocessor Unusable exception is taken.
27	W2	If set this bit indicates that the Watch2 register caused a Watch exception.
26	W1	If set this bit indicates that the Watch1 register caused a Watch exception.
24	IV	Enables the new dedicated interrupt vectors offset from the base exception vector. 1: interrupts use new exception vector base offset (0x200) and enables priority encoder and programmable interrupt exception offsets. 0: interrupts use common exception vector base offset (0x180).
23:8	IP[15:0]	Indicates an interrupt is pending. Each bit in this 16 bit field corresponds to interrupts 15:0 and indicates the interrupt pending status. 1: interrupt pending 0: no interrupt
6:2	ExcCode	Exception code field (see Table 13.9:)
	0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 13.8: Interrupt Pending Field

Bit(s)	Field	Description
23:22	IP[15:14]	Reserved
21	IP13	Performance Counter
20	IP12	Timer (if the ***TE*** bit is set)
19:10	IP[11:2]	Int*9 through Int*0. IP7 is capable of storing either the state of the Timer interrupt (if mode bit 11 is 0) or the state if Int*5.
9:8	IP[1:0]	SWINT1 & SWINT0

Table 13.9: Cause Register ExcCode Field

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

Exception Code Value	Mnemonic	Description
14	----	Reserved
15	FPE	Floating-Point exception
16	IWE	Instruction Watch Exception
17-22	----	Reserved
23	DWE	Data Watch Exception
24-31	----	Reserved

13.2.8 Exception Program Counter (EPC) Register (Set 0, Register 14)

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

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

- the virtual address of the instruction that was the direct cause of the exception, or
- the virtual address of the immediately preceding branch or jump instruction (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.

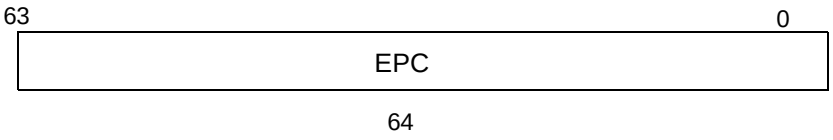


Figure 13.9 EPC Register Format

Figure 13.9 shows the format of the *EPC* register.

13.2.9 Watch1 Register (Set 0, Register 18)

The *Watch1* register works in conjunction with the *WatchMask* register and is used for debug and test. The register can be programmed to watch for a load address, a store address, an instruction address, or a range of address. Each time there is an address match, a watch exception is generated. If the contents of this register causes a watch exception, bit 25 (W1) in the *Cause* register is set. Refer to Section 17, “Debug and Test”, for more information.

13.2.10 Watch2 Register (Set 0, Register 19)

The *Watch2* register works in conjunction with the *WatchMask* register and is for debug and test. The register can be programmed to watch for a load address, a store address, an instruction address, or a range of address. Each time there is an address match, a watch exception is generated. If the contents of this register causes a watch exception, bit 26 (***W2***) in the *Cause* register is set. Refer to Section 17, “Debug and Test”, for more information.

13.2.11 XContext Register (Set 0, 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 used 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 13.10 shows the format of the *XContext* register; Table 13.10: describes the *XContext* register fields.

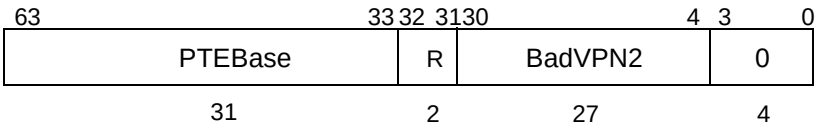


Figure 13.10 XContext Register Format

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.

Table 13.10:XContext Register Fields

Bit(s)	Field	Description
63:33	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.
32:31	R	The *Region* field contains bits 63:62 of the virtual address. 00: user 01: supervisor 11: kernel.
30:4	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.

13.2.12 PerfControl Register (Set 0, Register 22)

The *Performance Control* register is used during debug and test. This register controls the performance counters and whether counting is done in User mode or Kernel mode. Refer to Section 17, “Debug and Test”, for more information

13.2.13 WatchMask Register (Set 0, Register 24)

The *WatchMask* register is used during debug and test to mask some of the physical address bits, allowing either or both of the *Watch* registers to compare against a range of address as opposed to a specific address. The granularity of the address range is limited to the power of 2. Refer to Section 17, “Debug and Test”, for more information.

13.2.14 PerfCount Register (Set 0, Register 25)

The *Performance Counter* register is used during debug and test. This register is a 32-bit writable counter controlled by the *PerfControl* register. Refer to Section 17, “Debug and Test”, for more information.

13.2.15 Error Checking and Correcting (ECC) Register (Set 0, Register 26)

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

The *ECC* register is loaded by the **INDEX LOAD TAG** CACHE operation. Content of the *ECC* register 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 CACHE operation Fill.

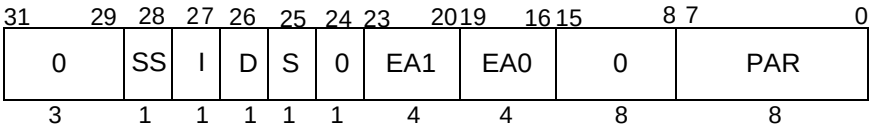


Figure 13.11 ECC Register Format

Figure 13.11 shows the format of the *ECC* register; Table 13.11: describes the register fields.

Table 13.11:ECC Register Fields

Bit(s)	Field	Description
28	SS	Set Select: Setting this bit causes the RM7000 to use set 1 of the cache during locking. Clearing this bit causes the processor to use set 0 during locking.
27	I	Setting this bit enables instruction cache locking.
26	D	Setting this bit enables data cache locking.
25	S	Setting this bit enables secondary cache locking.
23:20	EA1	Holds the data physical address [35:32] that caused the last parity error or last bus error if the E1 and EC bits within the CP0 CacheErr register are set. (see Figure 13.14)
19:16	EA0	Holds the data physical address [35:32] that caused the last parity error or last bus error if the E0 and EC bits within the CP0 CacheErr register are set. (Figure 13.15)
7:0	PAR	An 8-bit field specifying the parity bits read from or written to a primary cache. The following are the PAR field values for Index_Store_Tag_D and Index_Load_Tag_D cache operations: — PAR0 Even parity for least significant byte of requested doubleword — PAR1 Even parity for 2nd least significant byte — PAR2 Even parity for 3rd least significant byte — PAR3 Even parity for 4th least significant byte — PAR4 Even parity for 4th least significant byte — PAR5 Even parity for 3rd most significant byte — PAR6 Even parity for 2nd most significant byte — PAR7 Even parity for most significant byte of requested doubleword The following are the PAR field values for Index_Store_Tag_I and Index_Load_Tag_I cache operations: — PAR0 Even parity for least significant word of requested doubleword — PAR1 Even parity for most significant word of requested doubleword The following is the PAR field values for Index_Store_Tag_SD and Index_Load_Tag_SD cache operations: — PAR0 Even parity for the requested doubleword
	0	Reserved. Must be written as zeroes, and returns zeroes when read.

13.2.16 Cache Error (CacheErr) Register (Set 0, Register 27)

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

The *CacheErr* register holds cache index and status bits that indicate the source and nature of the error; it is loaded when a Cache Error exception is asserted.

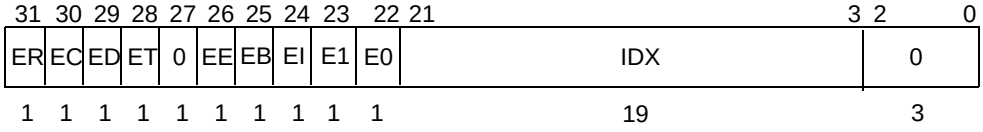


Figure 13.12 CacheErr Register Format

Figure 13.12 shows the format of the *CacheErr* register and Table 13.12: describes the *CacheErr* register fields.

Table 13.12:CacheErr Register Fields

Bit(s)	Field	Description
31	ER	Type of reference 0: instruction 1: data
30	EC	Cache level of the error 0: primary. Error is precise and IDX field is valid. 1: secondary or system interface. E1 or E2 is set to indicate a valid DErrAddr register. The IDX field is invalid.
29	ED	Indicates if a data field error occurred 0: no error 1: error
28	ET	Indicates if a tag field error occurred 0: no error 1: error
26	EE	Indicates if a system interface error has occurred. 0: no error 1: error on first doubleword of bus transaction.
25	EB	Set if a data error occurred in addition to the instruction error (indicated by the remainder of the bits). If so, this requires flushing the data cache after fixing the instruction error.
24	EI	This bit is set on a secondary data cache ECC error while refilling the primary cache on a store miss. The ECC handler must first do an Index Store Tag to invalidate the incorrect data from the primary data cache.
23	E1	If set, this bit indicates that the *DErrAddr1* register is valid and contains the data physical address that caused the last parity error or bus error.
22	E0	If set, this bit indicates that the *DErrAddr0* register is valid and contains the data physical address that caused the last parity error or bus error.
21:3	IDX	Physical address [21:3] of the reference that encountered the error.
	0	Reserved. Must be written as zeroes, and returns zeroes when read.

13.2.17 Error Exception Program Counter Register (Set 0, Register 30)

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

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

- 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.

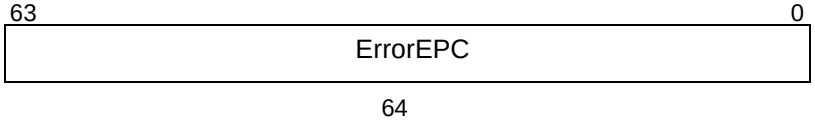


Figure 13.13 ErrorEPC Register Format

Figure 13.13 shows the format of the *ErrorEPC* register.

13.2.18 Data Error Address Register 0 (DErrAddr0, Set 1, Register 26)

If the ***E0*** and ***EC*** bits within the CP0 *CacheErr* register are set, then this register holds the data physical address that caused the last parity or last bus error. This read-only register is accessed with the **CFC0** instruction. Data physical address bits [35:32] are held on the ***EA0*** field of the CP0 *ECC* register.

Please refer to Section 12, “System Interface Protocol”, for more details on parity errors and bus errors.

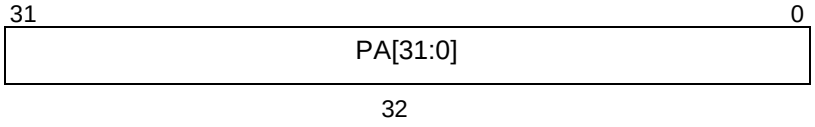


Figure 13.14 DErrAddr0 Register Format

13.2.19 Data Error Address Register 1 (DErrAddr1, Set 1, Register 27)

If the ***E1*** and ***EC*** bits within the CP0 *CacheErr* register are set, then this register holds the data physical address that caused the last parity error or imprecise bus error. This read-only register is accessed with the **CFC0** instruction. Data physical address bits [35:32] are held on the ***EA0*** field of the CP0 *ECC* register.

Please refer to Section 12, “System Interface Protocol”, for more details on parity errors and bus errors.

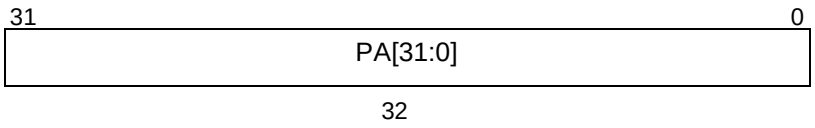


Figure 13.15 DErrAddr1 Register Format

13.3 Processor Exceptions

This section describes the processor exceptions and the cause of each exception, its processing by the hardware, and servicing by a handler (software). The types of exceptions are described in the following subsections.

13.3.1 Exception Types

This section gives sample exception handler operations for the following exception types:

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

When the ***EXL*** and ***ERL*** bits in the *Status* register are 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 ***ERL*** bit is a 1, the processor is also 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.

When the processor takes an cache parity error, the ***ERL*** bit is set to 1.

Returning from an exception, also resets the ***EXL*** bit to 0.

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

13.3.1.1 Reset Exception Process

```
T:Random ← TLBENTRIES–1
Wired ← 0
Config ¨{SC, EC, EP, TS, TC,BE} fields read from serial mode stream
ErrorEPC ¨ PC
Status.BEV ¨ 1, Status.SR ¨ 0, Status.ERL ¨ 1, Status.CO ¨ 0
PC ¨ 0xFFFF FFFF BFC0 0000
Count ← 0
Compare ← 0
Context ← 0
XContext ← 0
Cause ← 0
BadVaddr ← 0
TagLo ← 0
TagHi ← 0
ECC ← 0
CacheErr ← 0
IntCtl ← 0
IPLLo  ← 0
IPLHi  ← 0
PerfCount ← 0
PerfControl ← 0
Watch1 ← 0
Watch2 ← 0
WatchMask ← 0
DErrAddr0 ← 0
DErrAddr1 ← 0
```

Figure 13.16 Reset Exception Processing

Figure 13.16 shows the Reset exception process.

13.3.1.2 Cache Error Exception Process

```
T:ErrorEPC ¨ PC
CacheErr ¨ {ER, EC, ED, ET, EE, EB, EI, E1, E0, IDX} fields set for current error
Status.ERL ¨ 1
if Status.BEV = 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 13.17 Cache Error Exception Processing

Figure 13.17 shows the Cache Error exception process.

13.3.1.3 Soft Reset and NMI Exception Process

```
T:ErrorEPC ← PC
Status.BEV ← 1, Status.SR ←1, Status.ERL← 1
PC ← 0xFFFF FFFF BFC0 0000
```

Figure 13.18 Soft Reset and NMI Exception Processing

Figure 13.18 shows the Soft Reset and NMI exception process.

13.3.1.4 General Exception Process

```
T:Cause ¨ {BD,CE, IP, ExcCode} fields set for current exception
if Status.EXL = 0 then /* not in nested exception */
EPC ¨ PC
endif
Status.EXL ¨ 1
if (Cause.ExcCode=Interrupt && Cause.IV) then
vector = 0x200 + (IPL x SPACING)
else
vector = 0x180
if Status.BEV = 1 then /* What is the BEV bit setting */
PC ¨ 0xFFFF FFFF BFC0 0200 + vector /*access to uncached space*/
else
PC ¨ 0xFFFF FFFF 8000 0000 + vector /*access to cached space*/
endif
```

Figure 13.19 General Exception Processing

Figure 13.19 shows the process used for exceptions other than Reset, Soft Reset, NMI, and Cache Error.

13.3.1.5 Exception Vector Locations

The exception vector locations are dependent on the value of the ***BEV*** bit within the *Status* register. The ***BEV*** bit is set when the chip is powered up and typically the bootup prom within the system uses the ***BEV***=1 vector locations. Once the prom code starts the full operating system, typically the ***BEV*** bit will be cleared by software and the full operating system uses the ***BEV***=0 vector locations.

The processor can be programmed so that each interrupt priority level can have it's own dedicated exception vector. Upon power-up, the processor is reset so that the interrupts share the common exception vector with other exceptions (R4xxx compatibility mode). Please refer to Section 14, "Interrupts", to enable the prioritized, vectorized interrupt capability of the RM7000.

Table 13.13: shows the 64-bit-mode vector address for all exceptions depending upon the state of the ***BEV*** bit; 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 13.13:Vector Locations

Exception	BEV=0	BEV=1
Reset, Soft Reset, NMI	0xFFFF FFFF BFC0 0000	0xFFFF FFFF BFC0 0000
TLB refill, EXL=0	0xFFFF FFFF 8000 0000	0xFFFF FFFF BFC0 0200
XTLB refill, EXL=0 (X=64-bit TLB)	0xFFFF FFFF 8000 0080	0xFFFF FFFF BFC0 0280
Cache Error	0xFFFF FFFF A000 0100[a]	0xFFFF FFFF BFC0 0300
Interrupt, IV=0	0xFFFF FFFF 8000 0180	0xFFFF FFFF BFC0 0380
Interrupt, IV=1	0xFFFF FFFF 8000 0200 + (IPL x SPACING)[b]	0xFFFF FFFF BFC0 0400 + (IPL x SPACING)[c]
Others	0xFFFF FFFF 8000 0180	0xFFFF FFFF BFC0 0380

a. The Cache Error exception vector uses an uncached address to avoid using the caches in the presence of cache errors.
b. Refer to Section 14, “Interrupts”, to enable the vectorized, prioritized interrupt capability of the RM7000.
c. Refer to Section 14, “Interrupts”, to enable the vectorized, prioritized interrupt facility of the RM7000.

13.3.1.6 TLB Refill Vector Selection

In all present implementations of the MIPS III ISA, there are two TLB refill exception vectors:

- one for references to 32-bit address space (TLB Refill)
- one for references to 64-bit address space (XTLB Refill)

The TLB refill vector selection is based on the address space of the address (*user, supervisor,* or *kernel*) that caused the TLB miss, and the value of the corresponding extended addressing bit in the *Status* register (***UX***, ***SX***, or ***KX***). The current operating mode of the processor is not important except that it plays a part in specifying in which address space an address resides. The *Context* and *XContext* registers are entirely separate page-table-pointer registers that point to and refill from two separate page tables. For all TLB exceptions (Refill, Invalid, TLBL or TLBS), the ***BadVPN2*** fields of both registers are loaded as they were in the R4000.

In contrast to the RM7000, the R4000 processor selects the vector based on the current operating mode of the processor (*user, supervisor,* or *kernel*) and the value of the corresponding extended addressing bit in the *Status* register (***UX***, ***SX*** or ***KX***). In addition, the *Context* and *XContext* registers are not implemented as entirely separate registers; the ***PTEbase*** fields are shared. A miss to a particular address goes through either TLB Refill or XTLB Refill, depending on the source of the reference. There can be only a single page table unless the refill handlers execute address-deciphering and page table selection in software.

*Note: Refills for the 0.5 Gbyte supervisor mapped region, sseg/ksseg, are controlled by the value of **KX** rather than **SX**. This simplifies control of the processor when supervisor mode is not being used.*

Table 13.14: lists the TLB refill vector locations, based on the address that caused the TLB miss and its corresponding mode bit.

Table 13.14:TLB Refill Vectors

Space	Address Range	Regions	Exception Vector
Kernel	0xFFFF FFFF E000 0000 to 0xFFFF FFFF FFFF FFFF	kseg3	Refill (KX=0) or XRefill (KX=1)
Supervisor	0xFFFF FFFF C000 0000 to 0xFFFF FFFF DFFF FFFF	sseg, ksseg	Refill (KX=0) or XRefill (KX=1)
Kernel	0xC000 0000 0000 0000 to 0xC000 0FFE FFFF FFFF	xkseg	XRefill (KX=1)

Space	Address Range	Regions	Exception Vector
Supervisor	0x4000 0000 0000 0000 to 0x4000 0FFF FFFF FFFF	xsseg, xksseg	XRefill (SX=1)
User	0x0000 0000 8000 0000 to 0x0000 0FFF FFFF FFFF	xsuseg, xuseg, xkuseg	XRefill (UX=1)
User	0x0000 0000 0000 0000 to 0x0000 0000 7FFF FFFF	useg, xuseg, suseg, xsuseg, kuseg, xkuseg	Refill (UX=0) or XRefill (UX=1)

13.3.1.7 Priority of Exceptions

Table 13.15: 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.

Table 13.15:Exception Priority Order

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
Watch — 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
Watch — Data access
Virtual Coherency — Data access
Bus error — Data access
Interrupt *(lowest priority)*

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

13.3.1.8 Reset Exception

13.3.1.8.1 Cause

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

13.3.1.8.2 Processing

The CPU provides a special interrupt vector for this exception:

- location 0xFFFF FFFF BFC0 0000

The Reset vector resides in unmapped and uncached CPU address space, so the hardware need not initialize the TLB or the cache to process this exception. It also means the processor can fetch and execute instructions while the caches and virtual memory are in an undefined state.

The contents of all registers in the CPU are undefined when this exception occurs, except for the following register fields:

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

13.3.1.8.3 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.

13.3.1.9 Soft Reset Exception

13.3.1.9.1 Cause

The Soft Reset exception occurs in response to assertion of the **Reset*** input. Execution begins at the Reset vector when the **Reset*** signal is negated.

The Soft Reset exception is not maskable.

13.3.1.9.2 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.

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.

Because the Soft Reset can abort cache and bus operations, the cache and memory states are undefined when the Soft Reset exception occurs.

13.3.1.9.3 Servicing

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

13.3.1.10 Non Maskable Interrupt (NMI) Exception

13.3.1.10.1 Cause

The Non-Maskable Interrupt exception occurs in response to one of the following conditions:

- A low level on the **NMI*** pin.
- An external request write to any address with **SYSAD[6:4]** = 0 and bits 15 & 31 set in the data word

The NMI interrupt is not maskable and occurs regardless of the settings of the ***EXL***, ***ERL***, and ***IE*** bits in the *Status* Register.

13.3.1.10.2 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.

13.3.1.10.3 Servicing

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

13.3.1.11 Address Error Exception

13.3.1.11.1 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.

13.3.1.11.2 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.

13.3.1.11.3 Servicing

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

13.3.1.12 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 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.

13.3.1.12.1 TLB Refill Exception

13.3.1.12.1.1 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.

13.3.1.12.1.2 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* registers are undefined. The *EPC* register contains the address of the instruction that caused the exception, unless this instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the ***BD*** bit of the *Cause* register is set.

13.3.1.12.1.3 Servicing

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

13.3.1.12.2 TLB Invalid Exception

13.3.1.12.2.1 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.

13.3.1.12.2.2 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.

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.

13.3.1.12.2.3 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.

13.3.1.12.3 TLB Modified Exception

13.3.1.12.3.1 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.

13.3.1.12.3.2 Processing

The common exception vector is used for this exception, and the ***Mod*** code in the *Cause* register is set.

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers contain the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The contents of the *EntryLo* registers are undefined.

The *EPC* register contains the address of the instruction that caused the exception unless that instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the ***BD*** bit of the *Cause* register is set.

13.3.1.12.3.3 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.

13.3.1.13 Cache Error Exception

13.3.1.13.1 Cause

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

The RM7000 allows both precise and imprecise cache errors. Instruction parity errors are always precise while data parity errors are always imprecise.

13.3.1.13.2 Processing

Precise cache errors records the error source instruction address in the *ErrorEPC* register while imprecise cache errors place an instruction address after the source of the error in the *ErrorEPC* register.

The processor sets the ***ERL*** bit in the *Status* register. Processing then transfers to a special vector in uncached space;

— If ***BEV*** = 0, the vector is 0xFFFF FFFF A000 0100.
— If ***BEV*** = 1, the vector is 0xFFFF FFFF BFC0 0300.

13.3.1.13.3 Servicing

All errors should be logged. In general, parity errors can not be corrected. Only when the requested data has not modified can the system recover by invalidating the cache line and reloading from system memory. The ***CacheError.EC*** bit is first checked to determine where to look for the error address.

If the requested data has not been modified and the ***CacheError.EC*** bit is clear, the system uses the **CACHE** instruction to invalidate the cache block. The ***CacheError.Idx*** field is used to construct the operand address for the **CACHE** instruction. Then overwrite the old data through a cache miss, and resumes execution with an **ERET** instruction. Other errors are not correctable and are likely to be fatal to the current process.

If the requested data has not been modified and the ***CacheError.EC*** bit is set, the system uses the **CACHE** instruction to invalidate the cache block. Either *DErrAddr0* register or *DErrAddr1* register (as noted by the ***E0*** and ***E1*** bits in the *CacheError* register) is used to construct the operand address for the **CACHE** instruction. Then overwrite the old data through a cache miss. Since the *ErrorEPC* register does not hold exception source location, the operating system must parse the instruction stream to determine the exception source location. The virtual address of the load and store reference can then be obtained by interpreting the instruction. The physical address of the interpreted instruction can be obtained by using the TLBP instruction and reading the *EntryLo* registers to compute the physical page number.This exception source location address should be placed in the *ErrorEPC* register. Only then can execution resume with an **ERET** instruction. Other errors are not correctable and are likely to be fatal to the current process.

13.3.1.14 Bus Error Exception

13.3.1.14.1 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, or an uncached reference. A Bus Error exception resulting from a processor write transaction must be reported using the general interrupt mechanism.

The RM7000 allows both precise and imprecise bus errors. Imprecise bus errors occur when a non-blocking load has encountered an error off chip. Instruction bus errors are always precise as are data bus errors that occur for blocking loads.

13.3.1.14.2 Processing

Precise bus errors records the error source instruction address in the *EPC* register while imprecise bus errors place an instruction address after the source of the error in the *EPC* register. Precise bus errors will clear the ***CacheError.EC*** bit while imprecise bus errors will set the ***CacheError.EC*** bit.

The common exception 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.

For precise bus errors, 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.

For imprecise bus errors, either the *DErrAddr0* register or the *DErrAddr1* register (as noted by the ***E0*** and ***E1*** bits in the *CacheError* register) holds the physical address that caused the error.

13.3.1.14.3 Servicing

For precise errors, 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 instruction virtual address is contained in the *EPC* register (or +4 the contents of the *EPC* register if the ***BD*** bit of the *Cause* register is set).

- 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* registers to compute the physical page number.

For imprecise errors, either the *DErrAddr0* register or the *DErrAddr1* register are used to obtain the physical address of the fault.

The process executing at the time of this exception is handed a bus error signal, which is usually fatal.

13.3.1.15 Integer Overflow Exception

13.3.1.15.1 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.

13.3.1.15.2 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.

13.3.1.15.3 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.

13.3.1.16 Trap Exception

13.3.1.16.1 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.

13.3.1.16.2 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.

13.3.1.16.3 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.

13.3.1.17 System Call Exception

13.3.1.17.1 Cause

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

13.3.1.17.2 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.

13.3.1.17.3 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.

13.3.1.18 Breakpoint Exception

13.3.1.18.1 Cause

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

13.3.1.18.2 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.

13.3.1.18.3 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.

13.3.1.19 Reserved Instruction Exception

13.3.1.19.1 Cause

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

- an attempt is made to execute an instruction with an undefined major opcode (bits 31:26).
- an attempt is made to execute a **SPECIAL** instruction with an undefined minor opcode (bits 5:0).
- an attempt is made to execute a **REGIMM** instruction with an undefined minor opcode (bits 20:16).
- an attempt is made to execute MIPS III operations in 32-bit mode when in User or Supervisor modes.
- an attempt is made to execute MIPS IV opcodes in user-mode with ***SR[31]*** = 0.

The 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.

13.3.1.19.2 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.

13.3.1.19.3 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.

13.3.1.20 Coprocessor Unusable Exception

13.3.1.20.1 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.

13.3.1.20.2 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.

13.3.1.20.3 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.

13.3.1.21 Floating-Point Exception

13.3.1.21.1 Cause

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

13.3.1.21.2 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.

13.3.1.21.3 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.

13.3.1.22 Interrupt Exception

13.3.1.22.1 Cause

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

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

13.3.1.22.2 Processing

The common exception vector is used for this exception if ***Cause[IV]***=0, and the ***Int*** code in the *Cause* register is set. If ***Cause[IV]***=1, then one of the Interrupt exception vectors is used.

Please refer to Section 14, “Interrupts”, for discussion on the RM7000 prioritized and vectorized interrupt facility.

13.3.1.22.3 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. If the interrupt is created by an external request write, then it must be cleared by a second external request write.

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** (**ERET**) instruction is executed. The **SYNC** instruction can be used for this purpose. Otherwise the interrupt may be serviced again even though there is no actual interrupt pending.

13.3.1.23 Instruction Watch Exception

13.3.1.23.1 Cause

The Instruction Watch exception (IWE) occurs when an instruction fetch is attempted on the physical addresses specified by the *Watch1*, *Watch2*, and *WatchMask* registers. The *IWatch* exception equates to *Cause* register exception code 16.

13.3.1.23.2 Processing

The common exception vector is used for this exception and the ***IWE*** code is set in the *Cause* register. Either the ***W1*** bit or the ***W2*** bit in the *Cause* register is set to indicate which *Watch* register caused the exception.

13.3.1.23.3 Servicing

The instruction watch exception is a debugging aid. Typically the exception handler transfers control to a debugger, allowing the user to examine the situation. To continue, the instruction watch exception must be disabled to execute the faulting instruction.

13.3.1.24 Data Watch Exception

13.3.1.24.1 Cause

The Data Watch Exception (DWE) occurs when a load or store instruction attempts to reference the load or store physical addresses specified by the *Watch1*, *Watch2*, and *WatchMask* registers. The D*Watch* exception equates to *Cause* register exception code 23.

In previous implementations (R4640), if a load/store operation caused a watch exception, the related processor read/write request would never appear on the system bus. Due to the nature of the non-blocking caches, on the RM7000 it is possible for a processor read request to appear on the system bus even though the related load/store instruction has taken a DWE exception. Though the bus transaction might complete, neither the load destination register nor the store destination in the cache are ever modified if a DWE exception is taken. From the software point of view, the load/store instruction never executed, but from the bus point of view, the read request might have completed. For uncached, blocking loads, the bus request never appears on the bus if the DWE exception is taken.

13.3.1.24.2 Processing

The common exception vector is used for this exception and the DWE code is set in the *Cause* register. Either the ***W1*** bit or the ***W2*** bit in the *Cause* register is set to indicate which *Watch* register caused the exception.

13.3.1.24.3 Servicing

The data watch exception is a debugging aid. Typically the exception handler transfers control to a debugger, allowing the user to examine the situation. To continue, the data watch exception must be disabled to execute the faulting instruction.

13.3.1.25 Exception Handling and Servicing Flowcharts

The remainder of this section contains flowcharts 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).

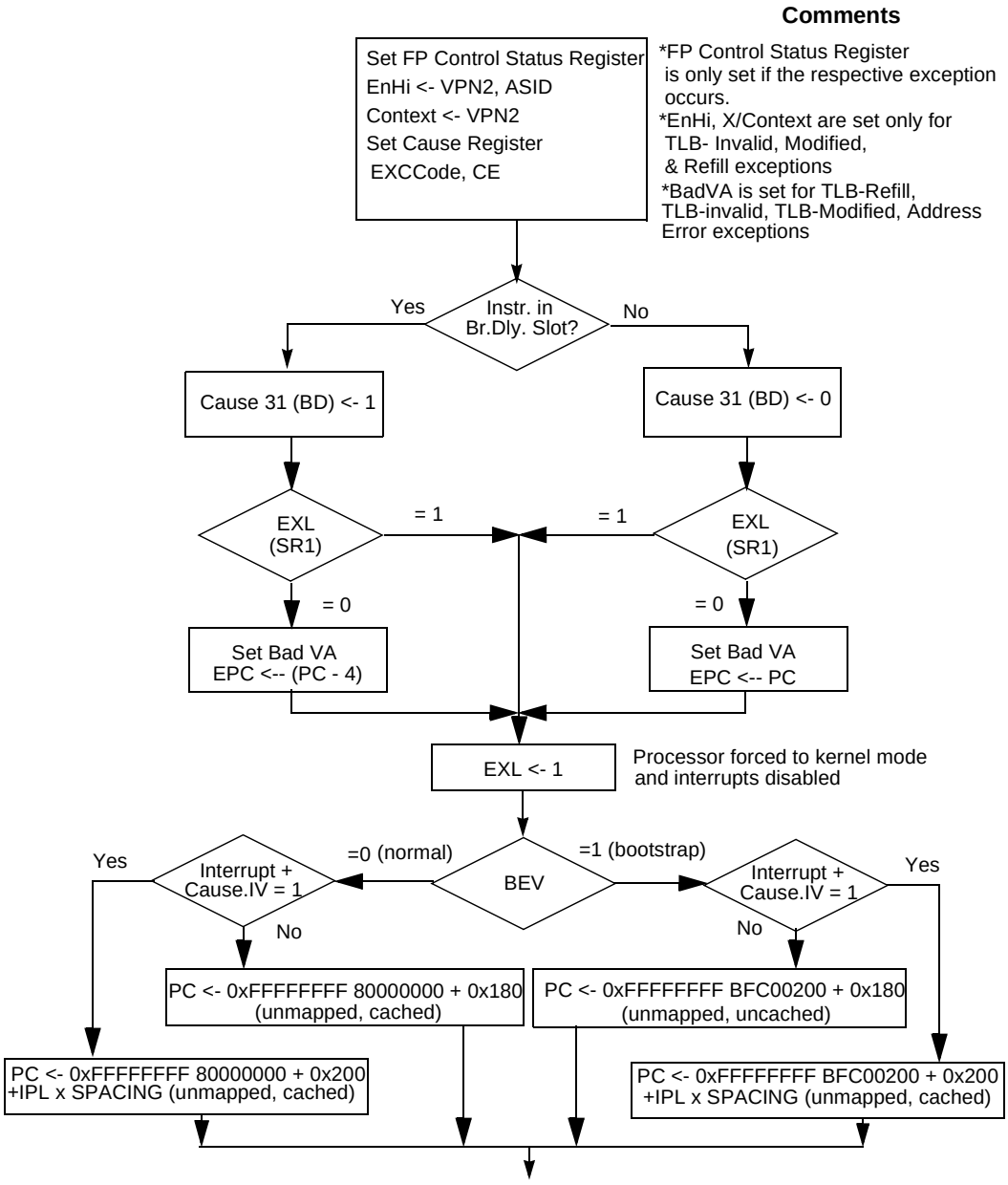


Figure 13.20 General Exception Handler (HW)

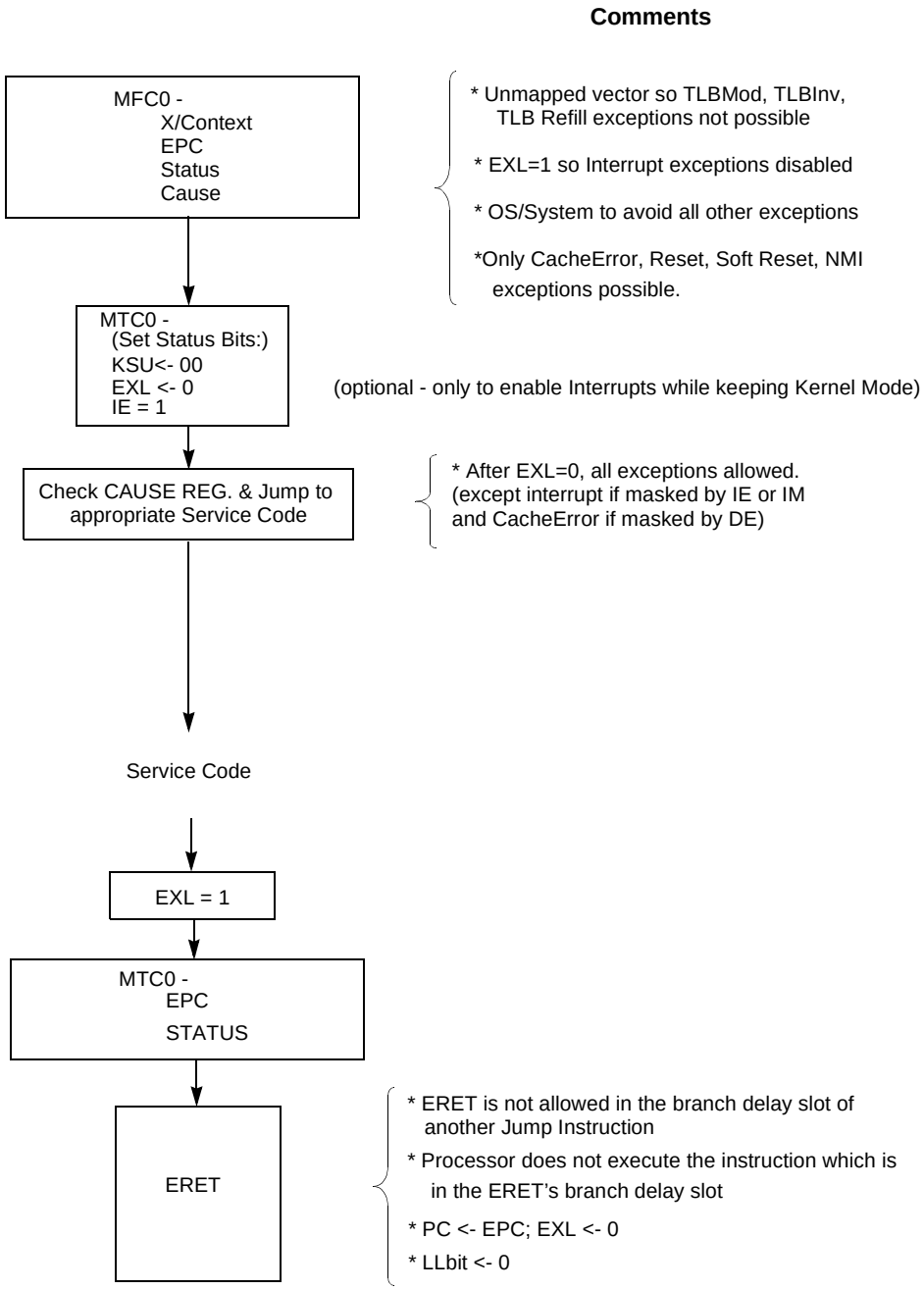


Figure 13.21 General Exception Servicing Guidelines (SW)

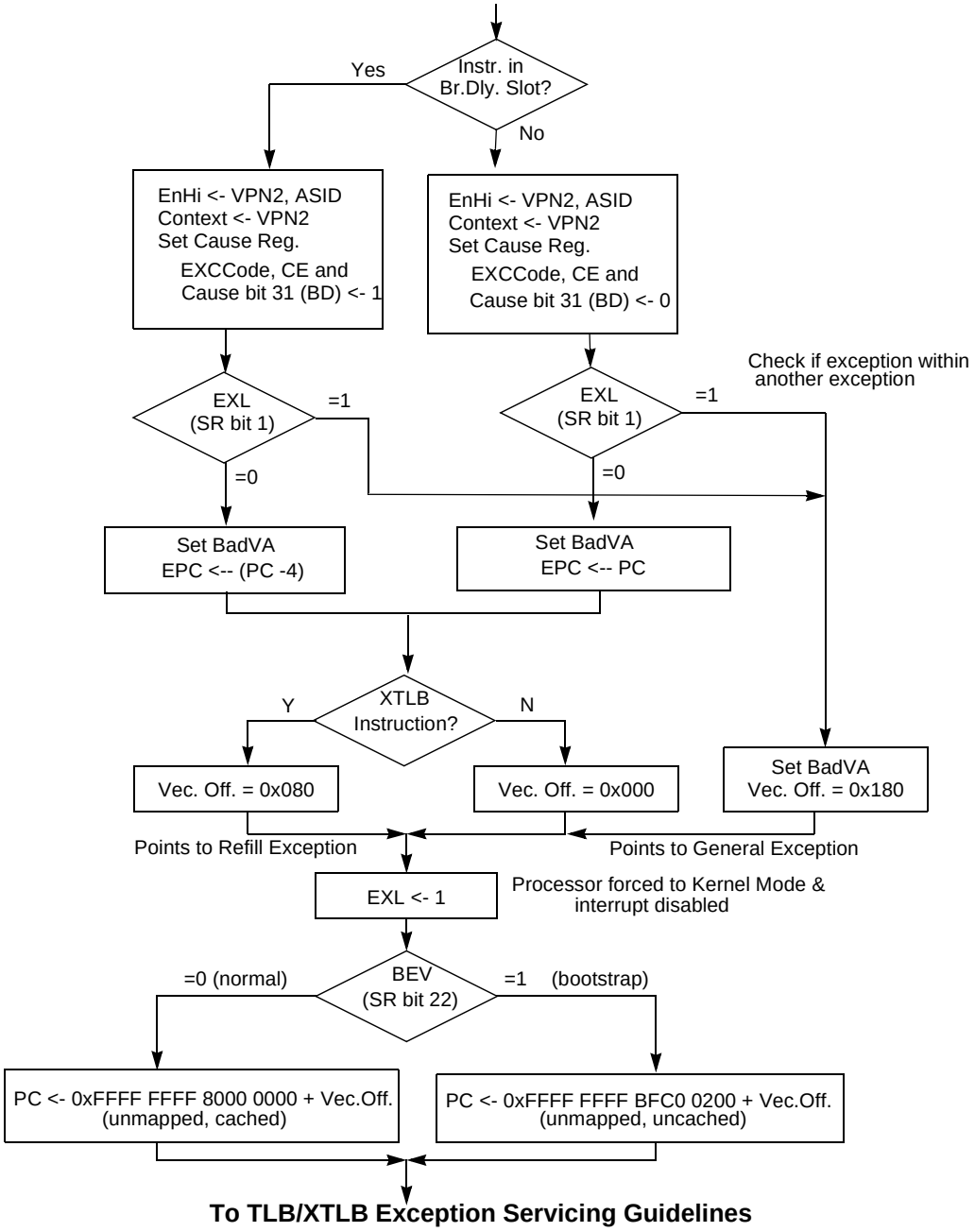


Figure 13.22 TLB/XTLB Miss Exception Handler (HW)

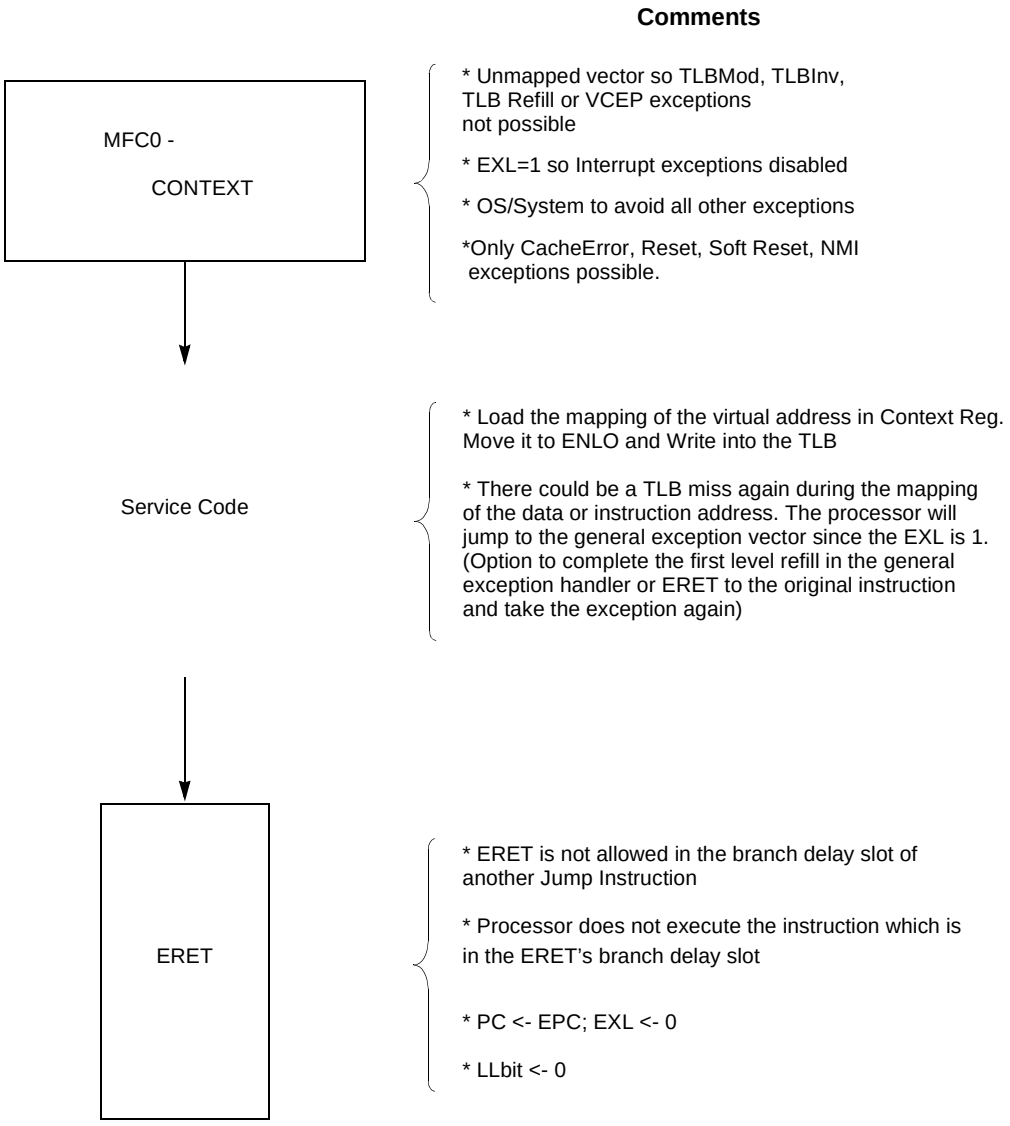


Figure 13.23 TLB/XTLB Exception Servicing Guidelines (SW)

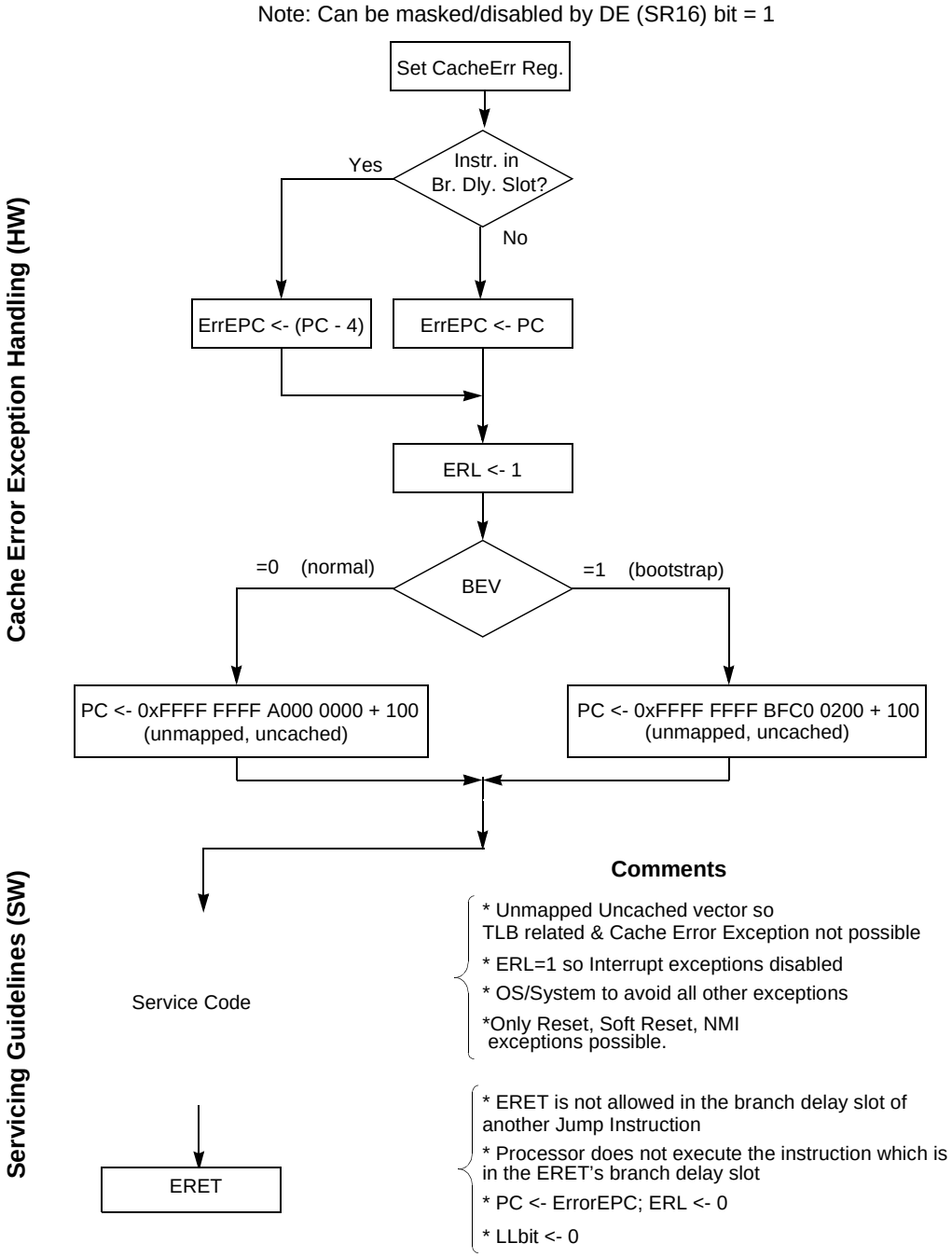


Figure 13.24 Cache Error Exception Handling (HW) and Servicing Guidelines

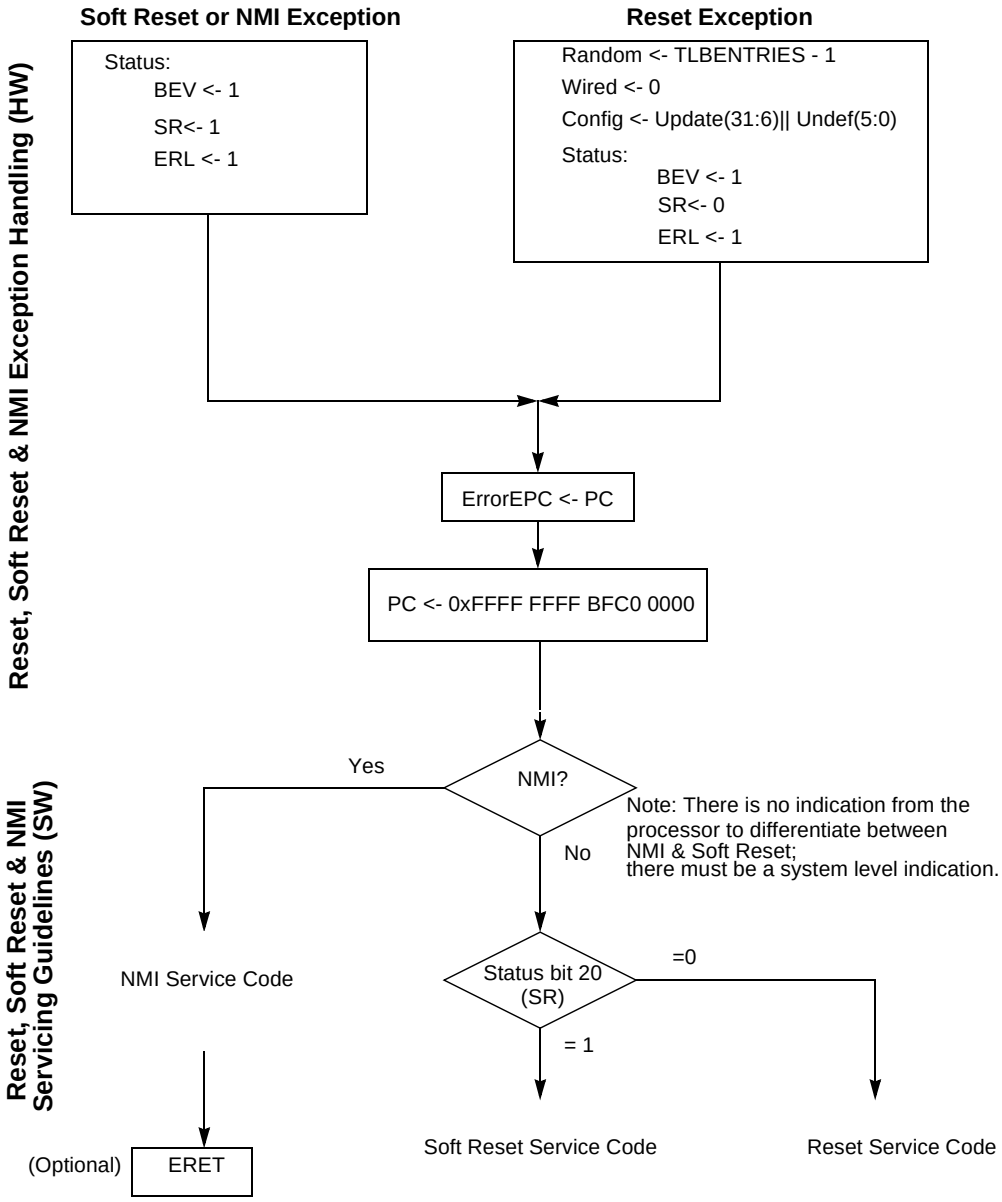


Figure 13.25 Reset, Soft Reset & NMI Exception Handling

Section 14 Interrupts

The RM7000 provides an enhanced interrupt handling mechanism quite different from that of previous generation MIPS processors. In addition to the six standard external interrupts, the RM7000 incorporates four additional external interrupt pins for a total of ten. Internal interrupts have also been added to support the performance counter used during debug and test, and software performance profiling.

The enhanced interrupt mechanism includes programmable interrupt priorities and separate vector offsets. Each interrupt source can be programmed with its own priority level and each priority level can have a different exception vector.

Interrupts no longer must be sorted totally by software. To streamline the interrupt handling process, the RM7000 provides hardware interrupt support through the use of a hardware priority encoder and three new CP0 registers. Two of the registers store the interrupt priority level for each interrupt. A third register stores interrupt mask and configuration information. If multiple interrupts are active at the same time, the highest priority interrupt is handled first.

In previous generation MIPS processors, most of the exceptions, including interrupts, shared the general exception vector located at offset 0x180. The processor first had to read the *Cause* register to determine the type of exception, then read the branch table to obtain the correct branch target address. Only after obtaining the branch target address could exception processing begin. The RM7000 uses multiple exception vectors solely for interrupts, thus eliminating having to read the *Cause* register and branch table before handling the interrupt. The hardware priority encoder on the RM7000 determines which exception vector is used.

14.1 Interrupt Sources

The RM7000 provides ten external interrupt signals, **Int*[9:0]**. These signals are mapped to bits [11:2] of the *Cause* register ***Interrupt Pending*** (***IP***) field, which has been expanded from 8 to 16 bits. In R5000-mode, all interrupts use the general exception vector [0x180] and are not prioritized by hardware. The ***IV*** bit in the CP0 *Cause* register enables the new vector offsets for the interrupts.

External interrupts are requested in one of two ways. External logic can either assert one of the **Int*[9:0]** pins, or issue an external request write command to the interrupt register.

The timer interrupt is requested when the contents of the *Count* register match the contents of the *Compare* register. On previous MIPS implementations, if the ***smMdTimIntDis2P*** boot mode bit (bit 11) is clear then the timer interrupt is enabled and the **Int5*** pin can not be used as an external interrupt source. On these chips, if this mode bit is set, then the timer interrupt is disabled and **Int5*** pin can be used as an external interrupt source. As an enhancement, the RM7000 allows the Timer interrupt to be mapped to bit [12] of the ***Interrupt Pending*** field instead of sharing bit [7] with the external interrupt **Int5***. This allows the timer interrupt to coexist with an external interrupt source on **Int5***. This enhancement is enabled by setting the ***TE*** bit within the *IntControl* register.

The Performance Counter interrupt is generated when bit [31] of that counter is set while incrementing the count of the selected event type. The Performance Counter interrupt is mapped to bit[13] of the ***Interrupt Pending*** field.

The RM7000 supports two software interrupts mapped to bits [1:0] of the *Cause* register ***IP*** field. Software interrupt 0 (*SWINT0*) is requested by setting ***IP0***, while software interrupt 1 (*SWINT1*) is requested by setting ***IP1***.

All interrupts can be disabled by clearing bit 0 of the *Status* register, which is the global ***Interrupt Enable*** bit (***IE*** bit). Interrupts are also disabled when another exception is being handled (either the ***EXL*** or ***ERL*** bit within the *Status* register is set).

14.2 Extended CP0 Addressing

The MIPS architecture provided enough address space for 32 coprocessor 0 registers. To accommodate the enhanced interrupt handling mechanism on the RM7000, 32 additional registers have been defined. The original 32 registers are designated as Set 0, while the second 32 registers are designated as Set 1. Of the second set of 32 registers, five are implemented. Three are used for interrupt priority and control, and two for imprecise parity error support. The remaining registers are reserved for future use. The new registers are accessed using the **CFC0** and **CTC0** instructions.

14.3 Interrupt Registers

The enhanced interrupt mechanism of the RM7000 required modifications to the *Cause* register, as well as the inclusion of three new interrupt registers. *Interrupt Control* [*IC*], *Interrupt Pending Level Lo* [*IPLLo*], and *Interrupt Pending Level Hi* [*IPLHi*]. These registers reside in CP0_Set 1 and are accessed using the **CFC0** and **CTC0** instructions.

14.3.1 Cause Register

The *Cause* register has been modified from the R5000 definition to accommodate the new interrupt handling mechanism. The ***Interrupt Pending*** **[*IP*]** field has been expanded from 8 to 16 bits. The ***Enable Priority Encoder*** (***IV)*** bit has been moved to accommodate the expanded ***Interrupt Pending*** (***IP***) field. Refer to Section 13, “Exception Processing”, for more information on the *Cause* register format and bit descriptions.

14.3.2 Status Register

The CP0 *Status* register contains an eight bit ***Interrupt Mask*** (***IMl***) field used for masking external interrupts **Int*[5:0]** and the two software interrupts. A global ***Interrupt Enable*** [***IE***] bit is used to enable or disable all interrupts. The four new interrupts, **Int*[9:6]**, are masked using the *Interrupt Control* (*IC*) register discussed below in section 14.3.4.

14.3.3 Info Register

The CP0 *Info* register contains an ***Atomic Interrupt Enable*** [***AE***] bit is used to easily enable or disable all interrupts.

14.3.4 Interrupt Control Register (IntControl, Set 1, Register 20)

This read/write register is accessed with **CFC0** and **CTC0** instructions.

The new *Interrupt Control* (*IC*) register contains an eight bit interrupt mask field, a five bit spacing field for interrupt vector spacing, and a ***Timer Exclusive*** bit for remapping of the timer interrupt. All bits in the *IC* register are cleared on power-up and on a hard reset, which sets all interrupts to the same priority (0). Figure 14.1 shows the format of the *Interrupt Control* register.

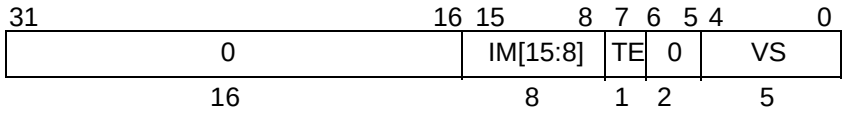


Figure 14.1 Interrupt Control Register

Table 14.1: Interrupt Control Register Fields

Field	Description
IM[15:8]	Interrupt Mask (Upper): The IM[15:8] field contains the interrupt mask for the Performance Counter, the timer, and Int*[9:6]. If a mask bit is set, the corresponding interrupt source is allowed to generate an interrupt. The mask bits are defined as follows: — IM[15:14] - Reserved — IM13 - Performance Counter Interrupt — IM12 - alternative Timer Interrupt [if TE = 1] — IM11 - Int*9 — IM10 - Int*8 — IM9 - Int*7 — IM8 - Int*6
TE	Timer Enable: The TE bit is used to enable the Timer Interrupt to be stored on IP12. Cleared is the default.
VS	Vector Spacing: This 5-bit spacing field is used to set the spacing between Interrupt vectors. Six different spacings are supported. Refer to section 14.4 for more information on interrupt vector spacing.

14.3.5 Interrupt Priority Level Lo Register (IPLLo, Set 1, Register 18)

This read/write register is accessed with **CFC0** and **CTC0** instructions.

The 32-bit *Interrupt Priority Level Lo* register (*IPLLo*) contains the priority level for interrupt sources 7:0. The register is divided into eight 4-bit fields, allowing each interrupt to be programmed with a priority level between 0 and 15. The interrupt priorities can be set in any order, including setting all interrupts to the same priority level. Level 0 is the highest priority level. Level 15 is the lowest. All bits in the *IPLLo* register are cleared on power-up and on a hard reset, which sets all interrupts to the same priority (0).

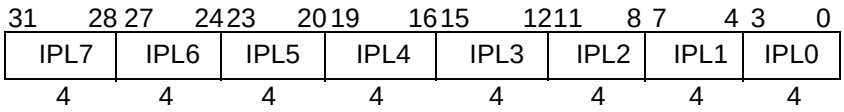


Figure 14.2 Interrupt Priority Level Lo Register

Figure 14.2 shows the format of the *IPPLo* register.

Table 14.2: Interrupt Priority Level Lo Register Fields

Field	Description
IPL7	Interrupt priority level for Int*5/Timer Interrupt.
IPL6	Interrupt priority level for external interrupt Int*4.
IPL5	Interrupt priority level for external interrupt Int*3.
IPL4	Interrupt priority level for external interrupt Int*2.
IPL3	Interrupt priority level for external interrupt Int*1.
IPL2	Interrupt priority level for external interrupt Int*0.
IPL1	Interrupt priority level for software interrupt 1.
IPL0	Interrupt priority level for software interrupt 0.

14.3.6 Interrupt Priority Level Hi Register (IPLHi, Set 1, Register 19)

This read/write register is accessed with **CFC0** and **CTC0** instructions.

The 32-bit *Interrupt Priority Level Hi* register (*IPLHi*) contains the priority level for the *Performance Counter*, the Timer, and external interrupt pins **Int*[9:6]**. The register is divided into eight 4-bit fields, allowing each interrupt to be programmed with a priority level between 0 and 15. Bits 31:24, which correspond to interrupts 15 and 14, are reserved and should not be used. The interrupt priorities can be set in any order, including setting all interrupts to the same priority level. Priority 0 is the highest level and priority 15 is the lowest. All bits in the *IPLHi* register are cleared on power-up and on a hard reset, which sets all interrupts to the same priority (0).

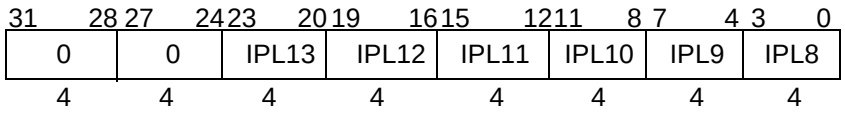


Figure 14.3 Interrupt Priority Level Hi Register

Figure 14.3 shows the format of the *IPLHi* register.

Table 14.3: Interrupt Priority Level Lo Register Fields

Field	Description
IPL13	Interrupt priority level for the performance counter interrupt.
IPL12	Interrupt priority level for the timer interrupt. This priority level is used when the TE bit in the IntControl register is set.
IPL11	Interrupt priority level for external interrupt Int*9.
IPL10	Interrupt priority level for external interrupt Int*8.
IPL9	Interrupt priority level for external interrupt Int*7.
IPL8	Interrupt priority level for external interrupt Int*6.

14.4 Interrupt Vector Spacing

The interrupt vector spacing field (VS) in the *Interrupt Control* register allows the user to choose how much memory is allocated between interrupt vectors. A small amount of vector spacing can be used to accommodate memory-constrained designs, allowing the user to set up the vectors as jumps to the actual interrupt routines. For designs where interrupt latency is important, a larger amount of vector spacing can be allocated, allowing the entire routine to be located at that vector, thereby minimizing the number of branches or jumps.

The interrupt vector base offset is 0x200. This means that ***IPL0*** always starts at base offset 0x200 and increments based on the value in the ***VS*** field. Table 14.4: shows the possible vector spacings for the ***VS*** field.

If all interrupt sources are programmed to use priority level 0 and a spacing of 0, then the interrupt behavior is identical to that of the R4640/4650 and R5000 processors. This is the power-on default.

Table 14.4: Interrupt Vector Spacing

IC[4:0] (VS Field)	Address Increment Between Vectors	Field Size
0x00	0x000	0
0x01	0x020	32
0x02	0x040	64
0x04	0x080	128
0x08	0x100	256
0x10	0x200	512
All others	Reserved	N/A

There are sixteen possible vector offsets for each vector spacing. If all of these vectors are not used, the interrupt service routine can be customized, allowing some interrupt service routines to be larger than others for a single space setting. For example, an interrupt service routine can overrun the minimum space allocated for a particular interrupt as long as the next sequential space is not used by any other interrupt. If subsequent interrupt vectors are not utilized, an interrupt service routine may extend into the un-used vector address space. Figure 14.4 shows an example of multiple sizes of interrupt service routines. This example assumes the following:

- A vector spacing of 0x100 (***SPACING*** = 0x08).
- The ***SR.BEV*** bit is cleared, indicating an exception vector base address of 0x80000000.
- The ***Cause.IV*** bit is set, indicating an interrupt vector offset of (0x200 + (Interrupt Priority Level x Address Increment Size)).

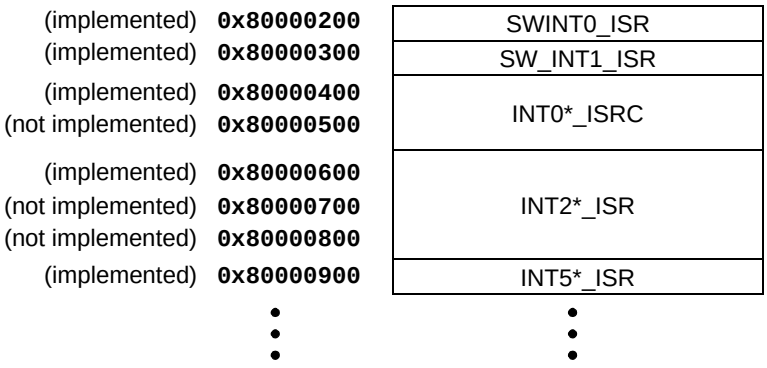
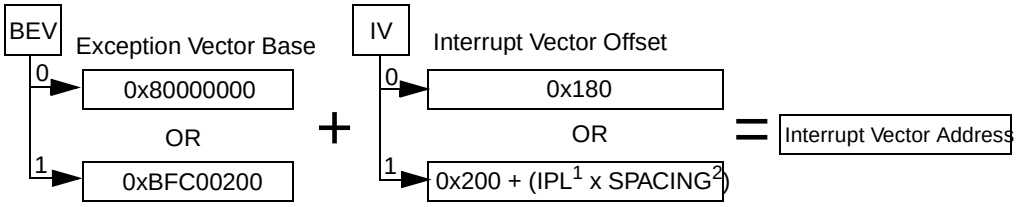


Figure 14.4 Customized Vector Spacing Example

The resulting interrupt vector address is listed along side each entry in Figure 14.4.

If the ***IV*** bit in the CP0 *Cause* register is set, the active interrupt priority, combined with the setting in the ***SPACING*** field, generates a vector offset that is added to the interrupt base address of 0x200 that in turn generates the interrupt vector offset. This offset is then added to the vector base to produce the final interrupt vector address. If the ***IV*** is cleared, the normal exception vector is used.

The ***BEV*** bit in the CP0 *Status* register determines the exception vector base address. If the bit is set, the exception base address is 0xBFC00200. If the bit is cleared the address is 0x80000000. Figure 14.5 shows a diagram of how the final interrupt vector address is calculated.



1] Interrupt Priority Level (see Table 13.5)
2] Address Increment Size derived from the SPACING field of the Interrupt Control register.

Figure 14.5 Interrupt Vector Address Calculation

Table 14.5: lists the various vector spacings and the corresponding location of the vector. This table is valid when the *Cause* register ***IV*** bit is set.

Table 14.5: Interrupt Vector Offsets

Interrupt Priority Level	Spacing Field [IC[4:0]], IV bit = 1					
	0x00	0x01	0x02	0x04	0x08	0x10
0	0x200	0x200	0x200	0x200	0x200	0x200
1	0x200	0x220	0x240	0x280	0x300	0x400
2	0x200	0x240	0x280	0x300	0x400	0x600
3	0x200	0x260	0x2C0	0x380	0x500	0x800
4	0x200	0x280	0x300	0x400	0x600	0xA00
5	0x200	0x2A0	0x340	0x480	0x700	0xC00
6	0x200	0x2C0	0x380	0x500	0x800	0xE00
7	0x200	0x2E0	0x3C0	0x580	0x900	0x1000
8	0x200	0x300	0x400	0x600	0xA00	0x1200
9	0x200	0x320	0x440	0x680	0xB00	0x1400
10	0x200	x0340	0x480	0x700	0xC00	0x1600
11	0x200	0x360	0x4C0	0x780	0xD00	0x1800
12	0x200	0x380	0x500	0x800	0xE00	0x1A00
13	0x200	0x3A0	0x540	0x880	0xF00	0x1C00
14	0x200	0x3C0	0x580	0x900	0x1000	0x1E00
15	0x200	0x3E0	x05C0	0x980	0x1100	0x2000

Interrupt Priority Level	Spacing Field [IC[4:0]], IV bit =01
	All
All	0x180

Table 14.6: Interrupt List

Interrupt Source Level	Source		Interrupt Control Bit Fields			External Request	
	Name	Generated By	Mask (IM) Bit	Cause Register (IP) Bit	Priority Register Bit(s)	Set/Clear	Enable
	Reset	ColdReset*					
	NMI	NMI*				SYSAD15	SYSAD31
	Soft Reset	Reset*					

Interrupt Source Level	Source		Interrupt Control Bit Fields			External Request	
	Name	Generated By	Mask (IM) Bit	Cause Register (IP) Bit	Priority Register Bit(s)	Set/Clear	Enable
0	SWINT0	SW Setting Cause.IP0	SR8	8	IPLL[3:0]		
1	SWINT1	SW Setting Cause.IP1	SR9	9	IPLL[7:4]		
2	INT0	Int0*	SR10	10	IPLL[11:8]	SYSAD0	SYSAD16
3	INT1	Int1*	SR11	11	IPLL[15:12]	SYSAD1	SYSAD17
4	INT2	Int2*	SR12	12	IPLL[19:16]	SYSAD2	SYSAD18
5	INT3	Int3*	SR13	13	IPLL[23:20]	SYSAD3	SYSAD19
6	INT4	Int4*	SR14	14	IPLL[27:24]	SYSAD4	SYSAD20
7	INT5	Int5*	SR15	15	IPLL[31:28]	SYSAD5	SYSAD21
8	INT6	Int6*	IC8	16	IPLH[3:0]	SYSAD6	SYSAD22
9	INT7	Int7*	IC9	17	IPLH[7:4]	SYSAD7	SYSAD24
10	INT8	Int8*	IC10	18	IPLH[11:8]	SYSAD8	SYSAD24
11	INT9	Int9*	IC11	19	IPLH[15:12]	SYSAD9	SYSAD25
12	ALT.Timer	Compare Register	IC12	20	IPLH[19:16]		
13	PerCnter	Performance Counter	IC13	21	IPLH[23:20]		
14	Reserved		IC14	22	IPLH[27:24]		
15	Reserved		IC15	23	IPLH[31:28]		

14.5 Debug and Test Support

The RM7000 interrupt mechanism supports the use of the *Performance Counter* for code debug and test. The *Performance Counter* is mapped to ***INT13***. The RM7000 allows the Timer to be mapped to ***INT12*** instead of sharing ***INT5*** as in previous generations. This allows ***INT5*** to be used solely as an external interrupt source.

To allow for maximum flexibility during debug, the *Performance Counter* and Timer have corresponding interrupt mask bits ***IM[13:12]*** in the *Interrupt Control* register and ***Interrupt Pending*** bits (***IP[13:12]***) in the *Cause* register.

Refer to Section 17, “Debug and Test”, for more information on the *Watch* registers and the *Performance Counter*s.

14.6 Non-maskable Interrupt [NMI]

The non-maskable interrupt is caused either by an external request writing to the RM7000 or by a asserting the **NMI*** pin. This pin is latched into an internal register by the rising edge of **SysClock.**

If an external request write is used to generate a non-maskable interrupt, **SysAD15** and **SysAD31** must be set during the data cycle. Previous generations of MIPS processors used **SysAD6** and **SysAD22** for this purpose.

14.7 Asserting Interrupts Through External Request Writes

Writes via the external request mechanism of the system interface to the CPU are directed to various internal resources, based on an internal address map of the processor. An external request write with **SysAD[6:4]** = 0 during the address cycle will write to an architecturally transparent register called the *Interrupt* register. This register is available for external request writes, but not for external request reads.

During the external request's write data cycle, **SysAD[25:16]** and **SysAD31** selects the enables for the eleven individual *Interrupt* register bits. **SysAD[9:0]** and **SysAD15** are the values to be written into these selected bits. This allows any subset of the *Interrupt* register to be set or cleared with a single external request write. Figure 14.6 shows the mechanics of an external request write to the *Interrupt* register.

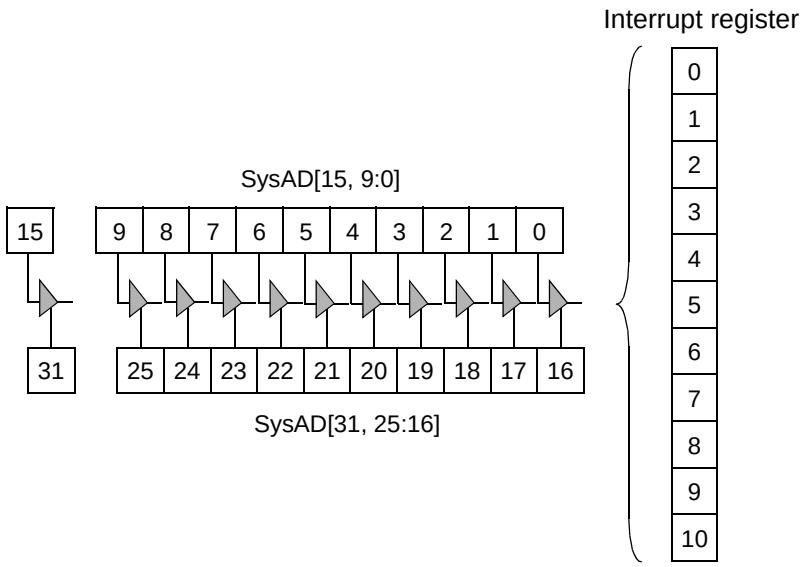
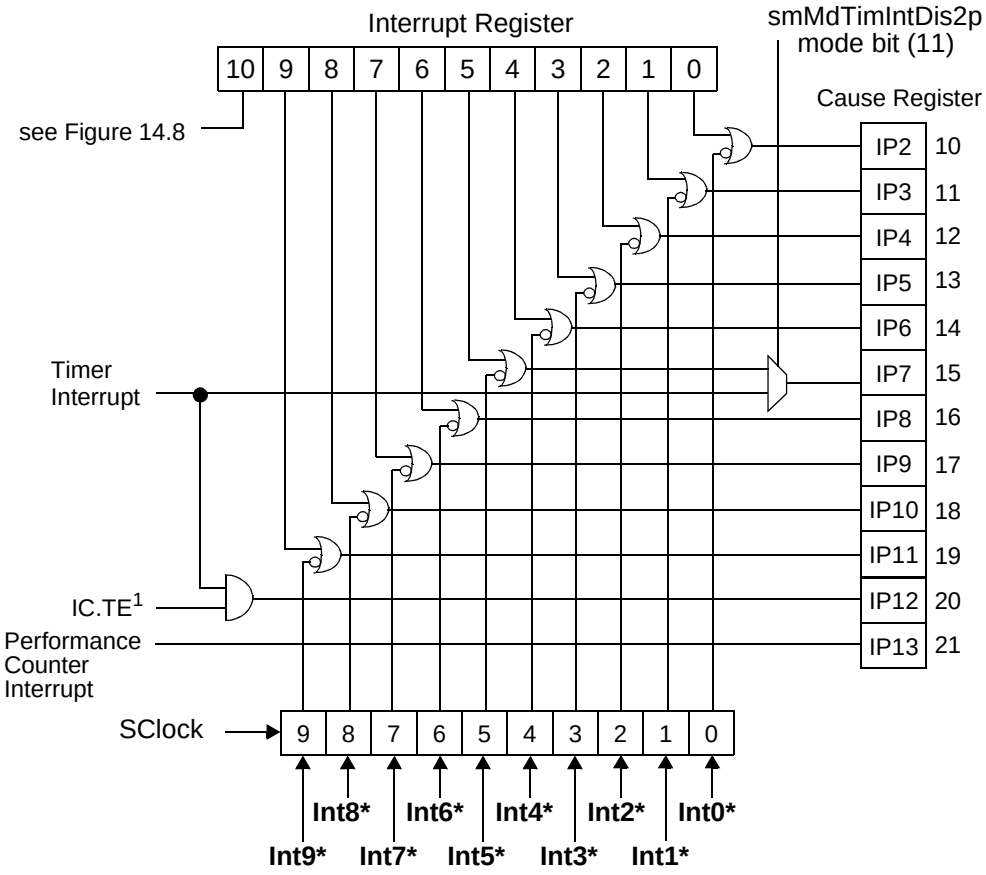


Figure 14.6 Interrupt Register Bits and Enables

Figure 14.7 shows how the RM7000 interrupt register can be read through the *Cause* register.

- Bits 9:0 of the *Interrupt* register are bit-wise OR'ed with the current value of interrupt pins **Int*[9:0].** The result is directly readable as bits 19:10 of the *Cause* register.
- Bit 10 of the *Interrupt* register is bit-wise OR'ed with the latched value of the **NMI*** signal. The resulting signal is used to generate a nonmaskable interrupt.



1) TE bit in the *Interrupt Control* register. Used to remap the timer interrupt.

Figure 14.7 RM7000 Interrupt Signals

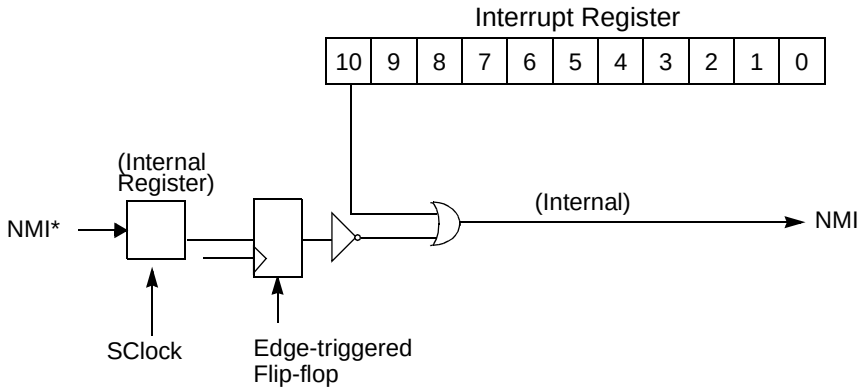


Figure 14.8 RM7000 Nonmaskable Interrupt Signal

Figure 14.8 shows the internal derivation of the **NMI*** signal for the RM7000 processor.

The **NMI*** pin is latched by the rising edge of **SysClock.** Bit 10 of the *Interrupt* register is then OR'ed with the inverted value of **NMI*** to form the nonmaskable interrupt. Either the falling edge of the latched **NMI*** pin or an external request write to any address with **SysAD[6:4]**=0 and a data of 0x8000 8000 will causes the actual interrupt.

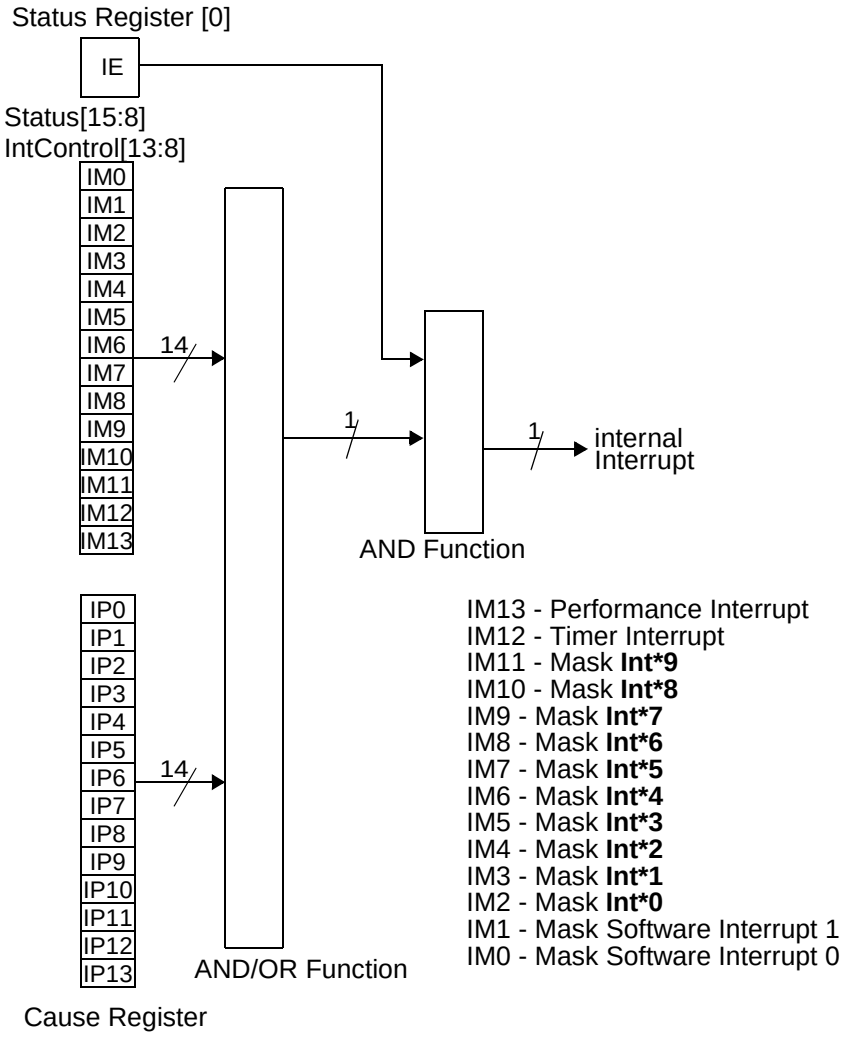


Figure 14.9 Masking the RM7000 Interrupt

Figure 14.9 shows the masking of the internal interrupt enable signal.

- *Cause* register bits 21:8 (***IP13-IP0***) are AND-ORed with *Status* register ***Interrupt Mask*** bits 15:8 (***IM7-IM0***) and *Interrupt Control* register ***Interrupt Mask*** bits 13:8 (***IM13-IM8***) 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 internal interrupt signal.

Section 15 Floating-Point Exceptions

This section describes FPU floating-point exceptions, including FPU exception types, exception trap processing, exception flags, saving and restoring state when handling an exception, and trap handlers for IEEE Standard 754 exceptions.

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.

15.1 Exception Types

The *Floating-Point Control/Status* register described in Section 6, “Floating-Point Unit”, 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, ***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.

Figure 15.1 illustrates the *Control/Status* register bits that support exceptions.

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.

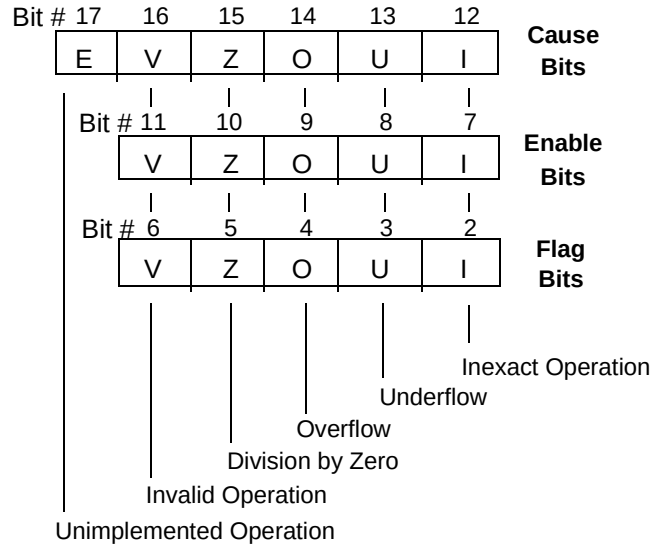


Figure 15.1 Control/Status Register Exception/Flag/Trap/Enable Bits

15.2 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.

15.3 Flags

A ***Flag*** bit is provided for each IEEE exception. This ***Flag*** bit is set on the assertion of its corresponding exception, with no corresponding exception trap signaled.

The ***Flag*** bit is reset by writing a new value into the *Status* register; flags can be saved and restored by software either individually or as a group.

When no exception trap is signaled, 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 15.1: lists the default action taken by the FPU for each of the IEEE exceptions.

Table 15.1: Default FPU Exception Actions

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 + ∞

Table 15.2: lists the exception-causing situations and contrasts the behavior of the FPU with the requirements of the IEEE Standard 754.

Table 15.2: FPU Exception-Causing Conditions

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	V	V	Source out of integer range
Signaling NaN source	V	V	V	
Invalid operation	V	V	V	0/0, etc.
Exponent underflow	U	E	See Table 6.1:	Normalized exponent < E_{min}
Denormalized or QNaN	None	See Table 6.1:	See Table 6.1:	Denormalized is (exponent = E_{min}-1 and mantissa <> 0)

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

15.4 FPU Exceptions

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

15.4.1 Inexact Exception (I)

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

- the rounded result of an operation is not exact, or
- the rounded result of an operation overflows, or

- 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.

Trap Enabled Results: If Inexact exception traps are enabled, the result register is not modified and the source registers are preserved.

Trap Disabled Results: The rounded or overflowed result is delivered to the destination register if no other software trap occurs.

15.4.2 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: $(+ \infty) + (- \infty)$ or $(- \infty) - (- \infty)$
- 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 the operand 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).

Trap Enabled Results: The original operand values are undisturbed.

Trap Disabled Results: A quiet NaN is delivered to the destination register if no other software trap occurs.

15.4.3 Division-by-Zero Exception (Z)

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

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: The result, when no trap occurs, is a correctly signed infinity.

15.4.4 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.)

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: The result, when no trap occurs, is determined by the rounding mode and the sign of the intermediate result.

15.4.5 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.

Trap Enabled Results: 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 Results: 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 15.1:).

15.4.6 Unimplemented Instruction Exception (E)

Any attempt to execute an instruction with an operation code or format code that has been reserved for future definition sets the ***Unimplemented*** bit in the ***Cause*** field in the FPU *Control/Status* register and traps. The operand and destination registers remain undisturbed and the instruction is emulated in software. Any of the IEEE Standard 754 exceptions can arise from the emulated operation, and these exceptions in turn are simulated.

The Unimplemented Instruction exception can also be signaled when unusual operands or result conditions are detected that the implemented hardware cannot handle properly. These include:

- Denormalized operand, except for Compare instruction and when the ***FS*** bit is set.
- 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)

Note: 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.

On the RM7000 additional causes of the unimplemented exception include:

- If the multiply portion of the **MADD**, **MSUB**, **NMADD**, or **NMSUB** instruction would produce an overflow, underflow or denormal output, regardless of the value of ***FCR31.FS***.
- The **CVT.L.FMT** instruction with an output that would be greater than 2^{52}-1 or less than -2^{52}
- Attempting to execute a MIPS IV floating-point instruction if the MIPS IV instruction set has not been enabled

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. It is up to the user to supply this software. Most compilers simply leave the three floating point exceptions disabled and have it default to whatever value is generated in the hardware.

Trap Enabled Results: The original operand values are undisturbed.

Trap Disabled Results: This trap cannot be disabled.

15.5 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.

15.6 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 both bits.

Section 16 Processor Synchronization

In a multiprocessor system, it is essential that two or more processors working on a common task 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 here: *test-and-set*, and *counter*. Even though the RM7000 does not support symmetric multi-processing (SMP), these are useful for multi-master and heterogeneous multi-processing.

Synchronization is also essential when multiple threads/tasks are sharing variables.

16.1 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 16.1 illustrates a test-and-set synchronization procedure that uses a semaphore. When the semaphore is set to 0 the shared data is unlocked. When the semaphore is set to 1 the shared data is locked.

The processor begins by loading the semaphore and checking to see if it is unlocked (set to 0) in steps 1 and 2. If the semaphore is not 0, the processor loops back to step 1. If the semaphore is 0, indicating the shared data is not locked, the processor next tries to lock out any other access to the shared data (step 3). If not successful, the processor loops back to step 1, and reloads the semaphore.

If the processor is successful at setting the semaphore (step 4), it executes the critical section of code (step 5) and gains access to the shared data, completes its task, unlocks the semaphore (step 6), and continues processing.

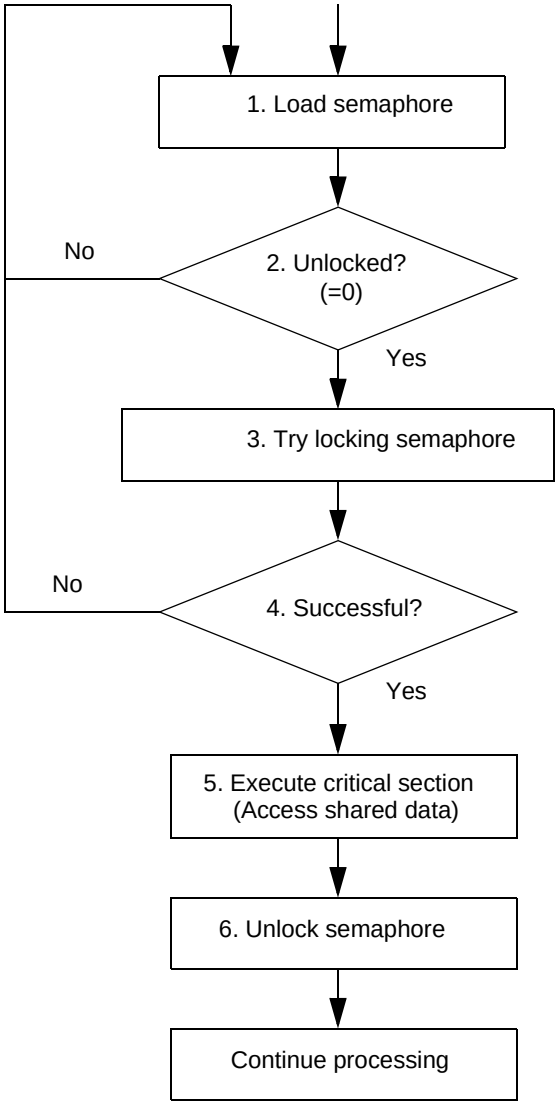


Figure 16.1 Synchronization with Test-and-Set

16.2 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 16.2 illustrates the counter synchronization procedure.

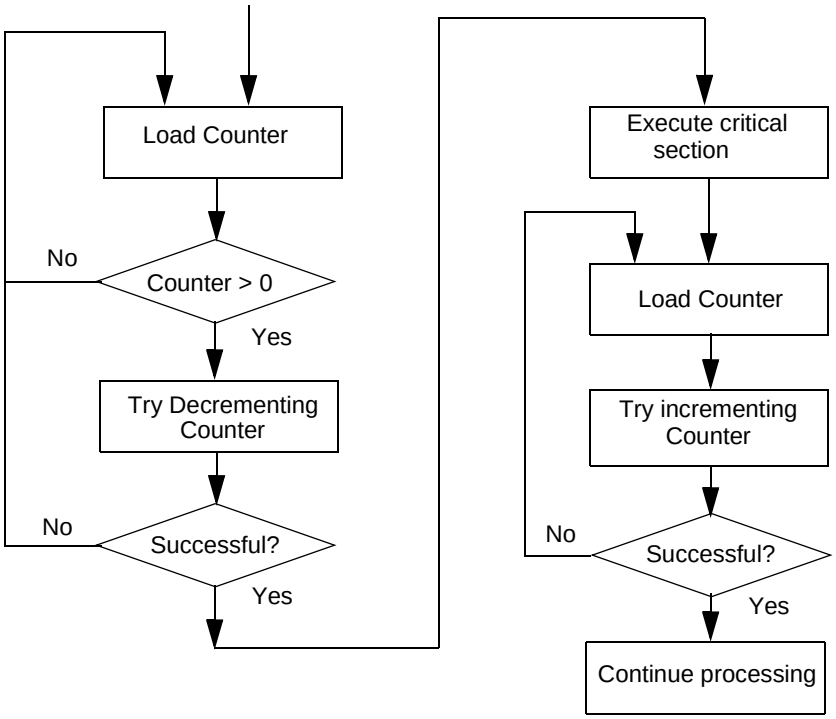


Figure 16.2 Synchronization Using a Counter

16.3 Load Linked and Store Conditional

The RM7000 Load Linked (**LL**) and Store Conditional (**SC**) instructions provide additional 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, 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:

- Provide a mechanism for generating all of the common synchronization primitives including test-and-set, counters, sequencers, etc., with no additional overhead.
- During an **LL** or **SC** operation, bus traffic is generated only if the state of the cache line changes, Locked words stay in the cache until some other processor takes ownership of that cache line.

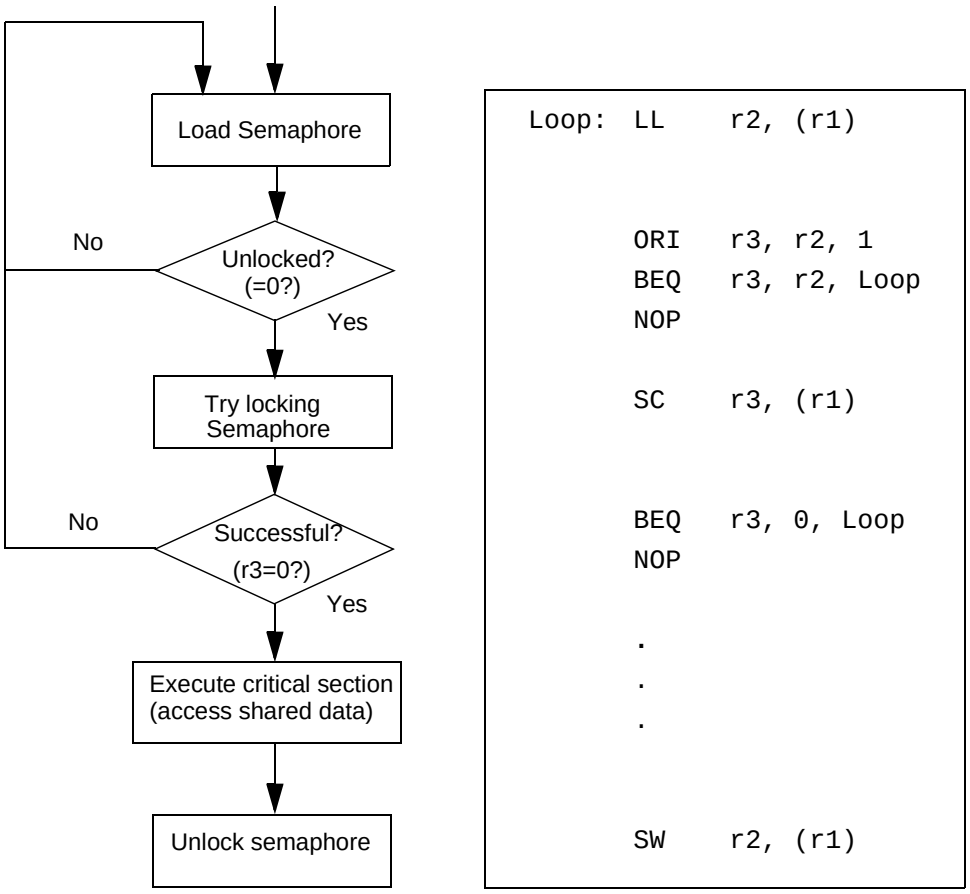


Figure 16.3 Test-and-Set using LL and SC

Figure 16.3 shows how to implement test-and-set using **LL** and **SC** instructions.

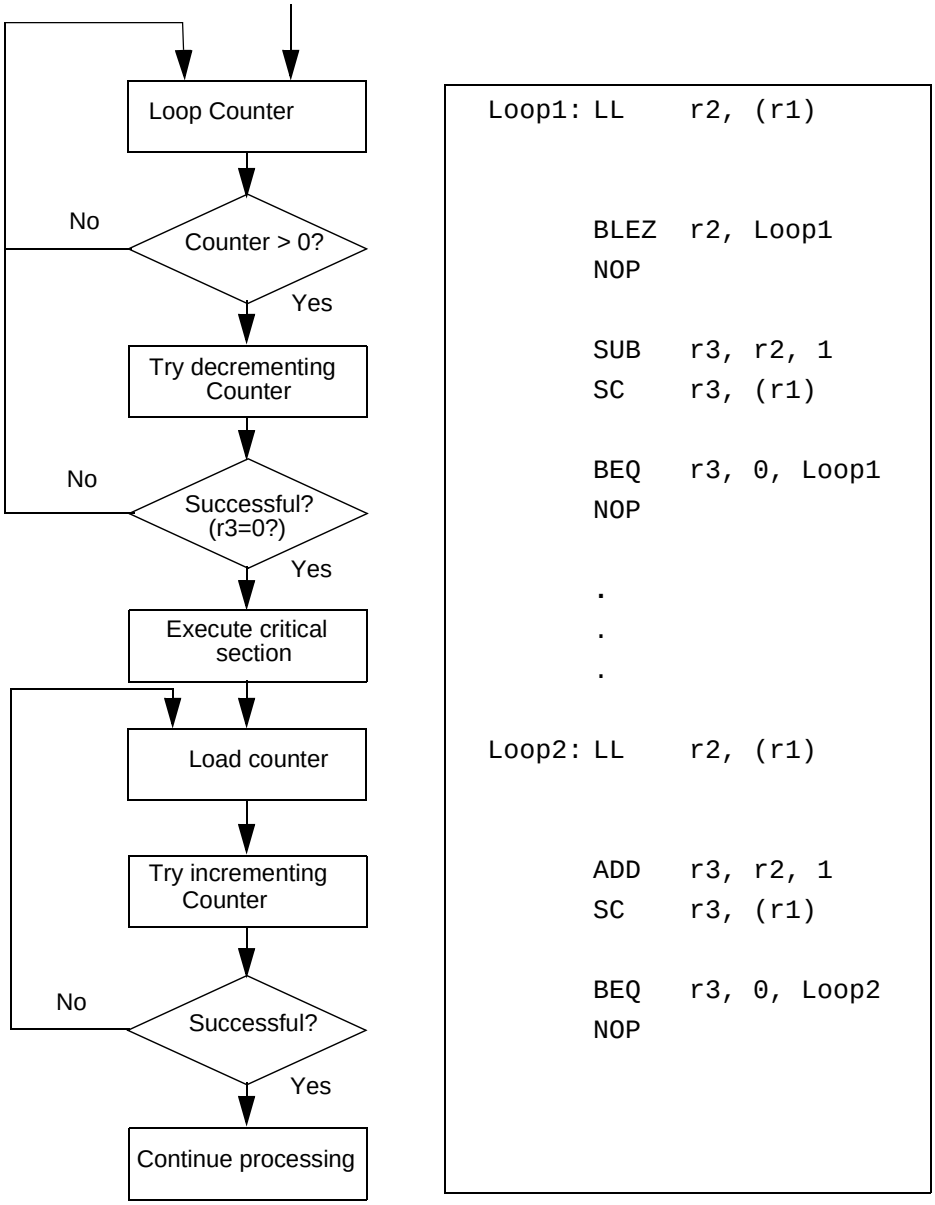


Figure 16.4 Counter Using LL and SC

Figure 16.4 shows synchronization using a counter implemented with **LL** and **SC** instructions.

Section 17 Debug and Test

The RM7000 incorporates five new CP0 registers used during device debug and test. Two 64-bit 'Watch' registers (*Watch1* and *Watch2)* can be used to watch for a specific load, store, or instruction address. A third *WatchMask* register is used to mask either or both of the *Watch* registers.

In addition to the three *Watch* registers, the RM7000 contains performance counter (*PerfCount*) and performance counter control (*PerfControl*) registers to aid in tracing performance bottlenecks. The *PerfCount* register can be preloaded with a certain value and causes an interrupt when bit 31 of the register is set. The *PerfControl* register selects one of 24 event types that can be monitored in either kernel mode or user mode.

17.1 Watch Exceptions

The RM7000 uses three CP0 registers to implement watch exceptions. A watch exception occurs when a memory access is attempted on an address that is being monitored. If the address being accessed equals the address being monitored by either the *Watch1* or *Watch2* registers, the RM7000 generates a Watch exception. All address comparisons are done on physical addresses.

Each *Watch* register can be enabled separately and can be programmed to watch for a load address, a store address, an instruction address. For each of these modes, either *Watch* register can be programmed to generate an address match on a range of addresses instead of a specific address. If either *Watch* register is enabled to monitor a load or store address and a match occurs, a watch exception is generated using *Cause* register exception code 23 as defined for the MIPS R4000 processor. If either *Watch* register is enabled to monitor an instruction address and a match occurs, the *IWatch* exception is taken using *Cause* register exception code 16. This exception is lower priority than instruction stream bus errors, but higher priority than any data exceptions.

In previous implementations (R4640), if a load/store operation caused a watch exception, the related processor read/write request would never appear on the system bus. Due to the nature of the non-blocking caches, on the RM7000 it is possible for the processor read request to appear on the system bus eventhough the related load/store instruction has taken a DWE exception. Though the bus transaction might complete, neither the load destination register nor the store destination in the cache are ever modified if a DWE exception is taken. From the software point of view, the load/store instruction never executed, but from the bus point of view, the read request might have completed. For uncached, blocking loads, the bus request never appears when a DWE exception is taken.

Two new bits have been added to the CP0 *Cause* register to indicate which *Watch* register caused the exception. If the *Watch1* register causes an exception, *Cause* register bit 25 is set. If the *Watch2* register causes an exception, *Cause* register bit 26 is set.

A third register, *WatchMask*, can be used to mask some of the physical address bits, allowing either or both of the *Watch* registers to compare against a range of addresses as opposed to a specific address. The granularity of the address range is limited to the power of 2 bits.

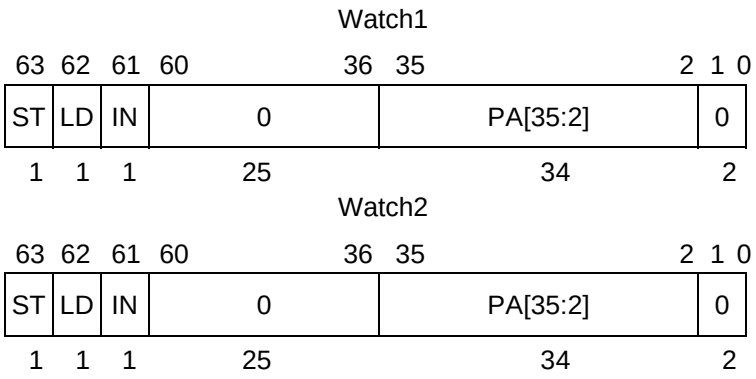


Figure 17.1 Watch Registers

Figure 17.1 shows the format of the *Watch1* and *Watch2* registers.

Table 17.1: Watch Register Fields

Field	Description
ST	Setting the ST bit enables watch exceptions on a store address match. Clearing the bit disables store address watch exceptions.
LD	Setting the LD bit enables watch exceptions on a load address match. Clearing the bit disables load address watch exceptions.
IN	Setting the IN bit enables watch exceptions on an instruction address match. Clearing the bit disables instruction address watch exceptions.
PA[35:2]	This 34-bit field contains physical address bits [35:2].

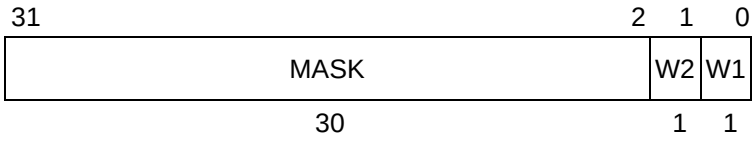


Figure 17.2 WATCHMASK Register

Figure 17.2 shows the format of the *WatchMask* register. Bits 31:2 of the *WatchMask* register are used to mask physical address bits 31:2 of the physical address. Physical address bits 35:32 of the *Watch* registers cannot be masked.

Table 17.2: WATCHMASK Register Fields

Field	Description
MASK	This 30-bit MASK field corresponds to PA[31:2] in the WATCH registers. If any of the MASK bits are set, the corresponding PA[31:2] bits of the address are ignored. PA[35:32] of the Watch registers cannot be masked.
W2	Setting this bit applies the watch mask on the WATCH_2 register.
W1	Setting this bit applies the watch mask on the WATCH_1 register.

17.2 Performance Counters

The RM7000 contains two new CP0 registers to aid in code profiling and determining performance bottlenecks, thereby improving the level of control during debug and testing of the device. When the RM7000 generates a performance counter interrupt, the return address can be saved to build different kinds of execution profiles. The number of performance counter interrupts can be used to calculate the hit rates and cycle counts for the different events. Performance counter interrupts can only occur when the ***IE*** bit in the CP0 *Status* register is set, and the ***IM(13)*** bit of the CP0 *Interrupt Control* (*IC*) register is set.

The performance counter function is implemented using two registers *PerfCount* and *PerfControl*. The *PerfCount* register is a 32-bit writable counter that causes an interrupt when bit 31 is set by the counter. This interrupt is reported by the ***IP(13)*** bit in the *Cause* register. The *PerfControl* register is a 32-bit register containing a 5-bit field that selects one of 24 event types. Two other bits allow for performance counting in either kernel mode and/or user mode. Note that only one event can be counted at a time. Once the interrupt handler software has serviced the performance counter interrupt, the handler should clear bit 31 of the *PerfCount* register before re-enabling interrupts to eliminate the same interrupt from occurring immediately after counting starts.

The *PerfCount* register is located at CP0 Set 0 register 25. The *PerfControl* register is located at CP0 Set 0 register 22.

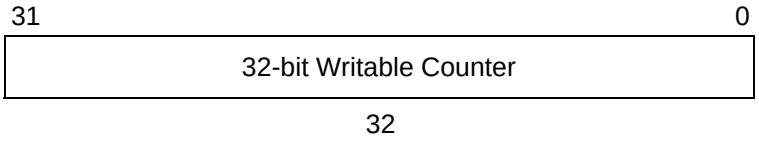


Figure 17.3 PerfCount Register

Figure 17.3 shows the format of the *PerfCount* register.

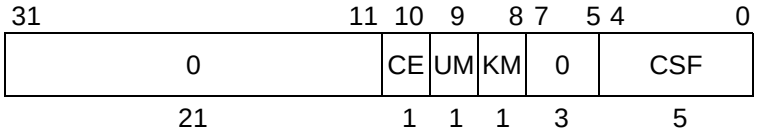


Figure 17.4 PerfControl Register

Figure 17.4 shows the format of the *PerfControl* register.

Table 17.3: PerfControl Register Fields

Field	Description
CE	Count Enable: Setting the CE bit enables incrementing of the PerfCount register. Clearing the bit disables the counter.
UM	User Mode: Setting this bit enables counting in User mode. Clearing the bit disables counting in User mode.
KM	Kernel Mode: Setting this bit enables counting in Kernel mode. Clearing the bit disables counting in Kernel mode.
CSF	Counter Source Field: Selects what type of event to count. 00: Clock cycles 01: Total instructions issued 02: Floating-point instructions issued 03: Integer instructions issued 04: Load instructions issued 05: Store instructions issued 06: Dual issued pairs 07: Branch prefetches 08: External Cache Misses 09: Stall cycles 0A: Secondary cache misses 0B: Instruction cache misses 0C: Data cache misses 0D: Data TLB misses 0E: Instruction TLB misses 0F: Joint TLB instruction misses 10: Joint TLB data misses 11: Branches taken 12: Branches issued 13: Secondary cache writebacks 14: Primary cache writebacks 15: Dcache miss stall cycles (cycles where both cache miss tokens taken and a third try is requested) 16: Cache misses 17: FP possible exception cycles 18: Slip Cycles due to multiplier busy 19: Coprocessor 0 slip cycles 1A: Slip cycles doe to pending non-blocking loads 1B: Write buffer full stall cycles 1C: Cache instruction stall cycles 1D: Multiplier stall cycles 1E: Stall cycles due to pending non-blocking loads - stall start of exception

Note 1: Slip cycles are only counted when the machine is in the run state. (Sicne stalls cause slips, the slip cycle count will only reflect non-stalled slip causes.)

Note 2: Slip count = (total cycle count) - (dual-issued instruction pair counts)

Note 3: FP and Integer Pipe resource slip cycles are equal to the difference between “Total Slip Cycles” and the sum of the reported slip cycles (including slips due to stall).

17.3 Debugging with on-chip caches

The presence of caches makes debugging more difficult as the number of external bus transactions are greatly reduced. One possible solution is to make specific pages uncached during debug. Pages within mapped address segments can be programmed to use the uncached coherency by modifying their ENLO0/ENLO1 C field to use coherency code 2. Addresses within kernel space (kseg0) can be programmed to use the uncache coherency by modifying ***CONFIG[2:0]***.

17.4 Debugging aids on the System Interface

During address cycles of processor requests, the processor outputs some information that might be useful during debugging.

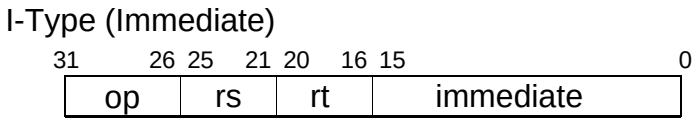
- The **RdType** pin denotes whether a processor read request was either for an instruction fetch or for a data fetch.
- In addition to the physical address bits appearing on **SysAD[35:0]**, the processor also places virtual address bits[13:12] on **SysAD[57:56]**. These bits might help in determining the virtual address of the processor request.

Section 18 Instruction Set Summary

Some of the RM7000 processor instructions are implementation-specific and are therefore not part of the MIPS IV instruction set. These are coprocessor instructions that perform operation in their respective coprocessors. Coprocessor loads and stores are I-type. Coprocessor computational instructions have coprocessor-dependent formats.

18.1 I-Type Instructions

All RM70000 implementation-specific instructions are classified as Immediate (I-type). Figure 18.1 shows the I-type instruction format.



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

Figure 18.1 CPU Instruction Formats

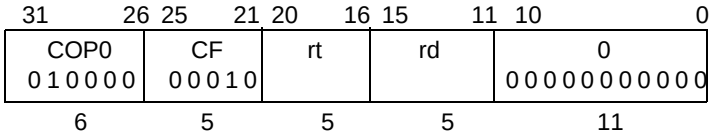
18.2 Implementation Specific CP0 Instructions

The RM7000 processor incorporates the implementation-specific coprocessor 0 (CP0) instructions shown in Table 18.1:. CP0 instructions are those that are not architecturally visible (not used by user-mode programs) and are used by the kernel.

Table 18.1: RM7000 Implementation Specific Instructions

CP0 Instruction	Definition
CACHE	Cache Management
CFC0	Move Control from CP0
CTC0	Move Control to CP0
DMFC0	Doubleword Move from CP0
DMTC0	Doubleword Move to CP0
ERET	Return from Exception
MFC0	Move from CP0
MTC0	Move to CP0
TLBP	Probe for TLB Entry
TLBR	Read Indexed TLB Entry
TLBWI	Write Indexed TLB Entry
TLBWR	Write Random TLB Entry
WAIT	Enter Standby Mode

18.2.1 CFC0 (Move Control From CP0)



18.2.1.1 Format:

```
CFC0    rt, rd
```

18.2.1.2 Description:

The contents of CP0 set 1 register *rd* are loaded into general register *rt*.

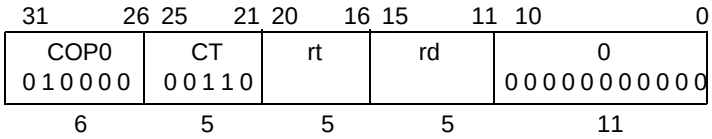
18.2.1.3 Operation:

```
32,64   T:      data ← CCR[0,rd]
        T+1:    GPR[rt] ← data
```

18.2.1.4 Exceptions:

Coprocessor unusable exception.

18.2.2 CTC0 (Move Control to Coprocessor)



18.2.2.1 Format:

```
CTC0    rt, rd
```

18.2.2.2 Description:

The contents of general purpose register *rt* are loaded into CP0 set 1 register *rd*.

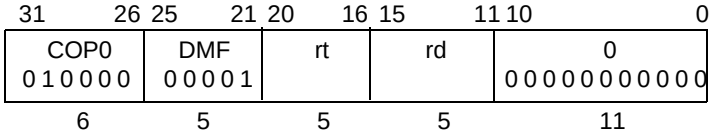
18.2.2.3 Operation:

```
32,64   T:      data ← GPR[rt]
        T+1:    CCR[0,rd] ← data
```

18.2.2.4 Exceptions:

— Coprocessor unusable exception.

18.2.3 DMFC0 (Doubleword Move From CP0)



18.2.3.1 Format:

```
DMFC0   rt, rd
```

18.2.3.2 Description:

The contents of CP0 set 0 register *rd* are loaded into general register *rt.*

This operation is defined in kernel mode regardless of the setting of the ***Status.KX*** bit. Execution of this instruction in supervisor mode with ***Status.SX*** = 0 or in user mode with ***UX*** = 0, causes a reserved instruction exception.

All 64-bits of the general register destination are written from the coprocessor register source. The operation of **DMFC0** on a 32-bit coprocessor 0 register is undefined.

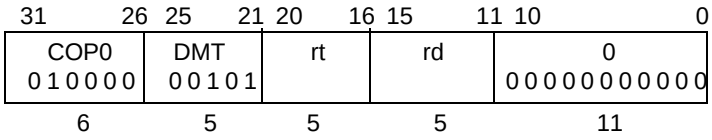
18.2.3.3 Operation:

```
T:      data ← CPR[0,rd]

T+1:    GPR[rt] ← data
```

18.2.3.4 Exceptions:

— Coprocessor unusable exception.
— Reserved instruction exception for supervisor mode with Status (***SX*** = 0) or user mode with Status (***UX*** = 0).

18.2.4 DMTC0 (Doubleword Move To CP0)



18.2.4.1 Format:

```
DMTC0   rt, rd
```

18.2.4.2 Description:

The contents of general register *rt* are loaded into CP0 set 0 register *rd*.

This operation is defined in kernel mode regardless of the setting of the ***Status.KX*** bit. Execution of this instruction in supervisor mode with ***Status.SX*** = 0 or in user mode with ***UX*** = 0, causes a reserved instruction exception.

All 64-bits of the coprocessor 0 register are written from the general register source. The operation of **DMTC0** on a 32-bit coprocessor 0 register is undefined.

Because the state of the virtual address translation system may be altered by this instruction, the operation of load instructions, store instructions, and TLB operations immediately prior to and after this instruction are undefined.

18.2.4.3 Operation:

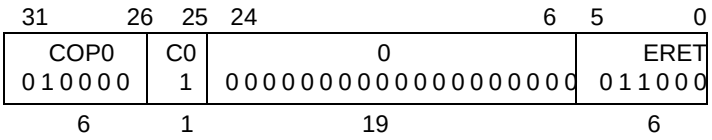
```
T:      data ← GPR[rt]

T+1:    CPR[0,rd] ← data
```

18.2.4.4 Exceptions:

Coprocessor unusable exception.

Reserved instruction exception for supervisor mode with ***Status.SX*** = 0 or user mode with ***Status.UX*** = 0.

18.2.5 ERET (Exception Return)



18.2.5.1 Format:

```
ERET
```

18.2.5.2 Description:

ERET is the instruction used for returning from an interrupt, exception, or error trap. Unlike a branch or jump instruction, **ERET** does not execute the next instruction.

An **ERET** instruction must not be placed in a branch delay slot.

If the processor is servicing an error trap (SR_2 = 1), then load the *PC* from the *ErrorEPC* and clear the ***ERL*** bit of the *Status* register (SR_2). Otherwise (SR_2 =0), load the *PC* from the *EPC*, and clear the ***EXL*** bit of the *Status* register (SR_1).

An **ERET** executed between a **LL** and **SC** also causes the **SC** to fail.

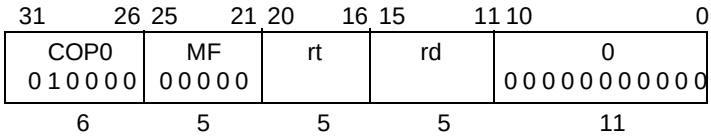
18.2.5.3 Operation:

```
T:    if SR2 = 1 then
          PC ← ErrorEPC
          SR ← SR31..3 || 0 || SR1..0
      else
          PC ← EPC
          SR ← SR31..2  || 0 || SR0
      endif
      LLbit ← 0
```

18.2.5.4 Exceptions:

Coprocessor unusable exception.

18.2.6 MFC0 (Move From CP0)



18.2.6.1 Format:

```
MFC0    rt, rd
```

18.2.6.2 Description:

The contents of CP0 set 0 register *rd* are loaded into general register *rt*.

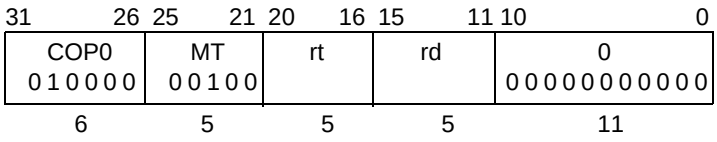
18.2.6.3 Operation:

```
32   T:     data ← CPR[0,rd]
     T+1:   GPR[rt] ← data
64   T:     data ← CPR[0,rd]
     T+1:   GPR[rt] ← (data31)32 || data31:0
```

18.2.6.4 Exceptions:

— Coprocessor unusable exception.

18.2.7 MTC0 (Move To CP0)



18.2.7.1 Format:

```
MTC0    rt, rd
```

18.2.7.2 Description:

The contents of general register *rt* are loaded into CP0 set 0 register *rd*.

Because the state of the virtual address translation system may be altered by this instruction, the operation of load instructions, store instructions, and TLB operations immediately prior to and after this instruction are undefined.

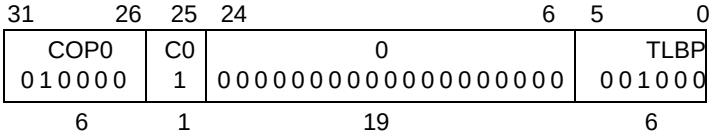
18.2.7.3 Operation:

```
32, 64   T:     data ← GPR[rt]
         T+1:   CPR[0,rd] ← data
```

18.2.7.4 Exceptions:

Coprocessor unusable exception.

18.2.8 TLBP (TLB Probe)



18.2.8.1 Format:

TLBP

18.2.8.2 Description:

The *Index* register is loaded with the address of the TLB entry whose contents match the contents of the *EntryHi* register. If no TLB entry matches, the high-order bit of the *Index* register is set.

The architecture does not specify the operation of memory references associated with the instruction immediately after a **TLBP** instruction, nor is the operation specified if more than one TLB entry matches.

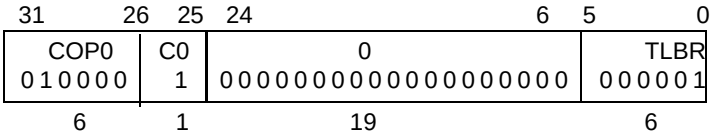
18.2.8.3 Operation:

```
T:   Index ← 1 || 0^31
      For i in 0..TLBEntries - 1
          if (TLB[i]167..141 and not (0^15 || TLB[i]216..205))
          = (EntryHi39..13 and not (0^15 || TLB[i]216..205)) and
          (TLB[i]140 or (TLB[i]135..128 = EntryHi7..0)) then
               Index ← 0^26 || i5..0
          endif
      endfor
```

18.2.8.4 Exceptions:

— Coprocessor unusable exception.

18.2.9 TLBR (Read Indexed TLB Entry)



18.2.9.1 Format:

TLBR

18.2.9.2 Description:

The *EntryHi* and *EntryLo* registers are loaded with the contents of the TLB entry pointed at by the contents of the TLB *Index* register. The operation is invalid (and the results are unspecified) if the contents of the TLB *Index* register are greater than the number of TLB entries in the processor.

The ***G*** bit (which controls ASID matching) read from the TLB is written into both of the *EntryLo0* and *EntryLo1* registers.

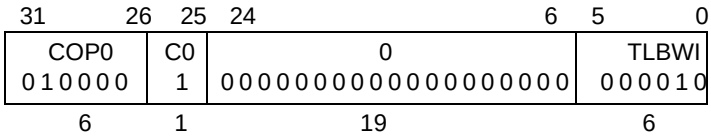
18.2.9.3 Operation:

```
T:   PageMask  ← TLB[Index5..0]255..192
     EntryHi   ← TLB[Index5..0]191..128 and not TLB[Index5..0]255..192
     EntryLo1  ← TLB[Index5..0]127..65  || TLB[Index5..0]140
     EntryLo0  ← TLB[Index5..0]63..1    || TLB[Index5..0]140
```

18.2.9.4 Exceptions:

Coprocessor unusable exception.

18.2.10 TLBWI (Write Indexed TLB Entry)



18.2.10.1 Format:

```
TLBWI
```

18.2.10.2 Description:

The TLB entry pointed to by the contents of the TLB *Index* register is loaded with the contents of the *EntryHi* and *EntryLo* registers.

The ***G*** bit of the selected TLB entry is written with the logical AND of the ***G*** bits in the *EntryLo0* and *EntryLo1* registers.

The operation is invalid (and the results are unspecified) if the contents of the TLB Index register are greater than the number of TLB entries in the processor.

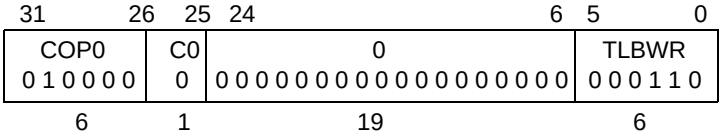
18.2.10.3 Operation:

```
T:   TLB[Index5..0]  ←
       EntryHi[39:25] || (EntryHi[24:13] and not PageMask) || EntryLo1 || EntryLo0
```

18.2.10.4 Exceptions:

— Coprocessor unusable exception.

18.2.11 TLBWR (Write Random TLB Entry)



18.2.11.1 Format:

```
TLBWR
```

18.2.11.2 Description:

The TLB entry pointed to by the contents of the TLB *Random* register is loaded with the contents of the *EntryHi* and *EntryLo* registers.

The ***G*** bit of the selected TLB entry is written with the logical AND of the ***G*** bits in the *EntryLo0* and *EntryLo1* registers.

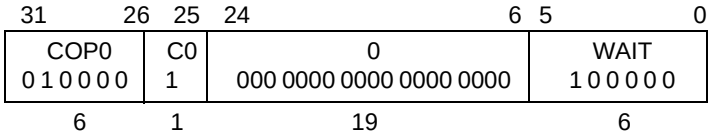
18.2.11.3 Operation:

```
T:   TLB[Random5..0] ←
        EntryHi[39:25] || (EntryHi[24:13] and not PageMask) || EntryLo1 || EntryLo0
```

18.2.11.4 Exceptions:

— Coprocessor unusable exception.

18.2.12 WAIT (Enter Standby Mode)



18.2.12.1 Format:

```
WAIT
```

18.2.12.2 Description:

The **WAIT** instruction is used to put the CPU into Standby Mode. In Standby Mode, most of the internal clocks are shut down which freezes the pipeline and reduces power consumption.

18.2.12.3 Operation:

```
T:      if SysAD bus is idle then
            Enter Standby Mode
        endif
```

18.2.12.4 Exceptions:

— Coprocessor unusable exception.

18.3 CACHE (cache management)

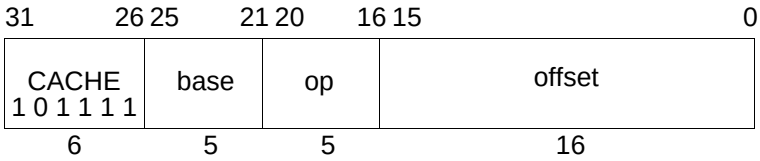


Figure 18.2 RM7000 CACHE Instruction Format

18.3.0.1 Format:

```
CACHE   op, offset(base)
```

18.3.0.2 Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The virtual address is translated to a physical address using the TLB, and the 5-bit sub-opcode specifies a cache operation for that address.

If the ***CU0*** (the lowest-order bit in the ***CU*** field) and CO bits in the *Status* register are cleared while in either user mode or supervisor mode, a *Coprocessor Unusable* exception is taken.

If the ***CU0*** bit is set and the ***CO*** bit is cleared while in either user mode or supervisor mode, a *Reserved Instruction* exception is taken.

If the ***CO*** bit is set while in user mode, then the *Hit_Writeback* and *Hit_Writeback_Invalidate* operations are available. All other operations cause a *Reserved Instruction* exception.

The operation of this instruction on any operation/cache combination not listed below, or on the secondary cache or tertiary cache when none is present, is undefined. The operation of this instruction on uncached addresses is also undefined.

The Index operation uses part of the virtual address to specify a cache block. The on-chip caches in the RM7000 are 4-way set associative, which implies that they are divided into quarter segments. Each quarter equates to a set.

For a primary cache of 16 Kbytes divided into four 4 Kbyte segments with 32 bytes per tag, vAddr(11:5) specifies the block. In addition, vAddr(13:12) specifies which cache set to operate on.

For a secondary cache of 256 Kbytes divided into four 64 Kbyte segments with 32 bytes per tag, vAddr(15:5) specifies the block. In addition, vAddr(17:16) specifies which cache set to operate on.

For a tertiary cache of $2^{CACHEBITS}$ bytes with 32 bytes per tag, $pAddr_{CACHEBITS..5}$ specifies the block. The tertiary cache is direct-mapped, so there is no need to specify which cache set to operate on.

Index Load Tag also uses $vAddr_{LINEBITS...4:3}$ to select the doubleword for reading parity. When the ***CE*** bit of the *Status* register is set, Hit WriteBack, Hit WriteBack Invalidate, Index WriteBack Invalidate, and Fill also use $vAddr_{LINEBITS ...4:3}$ to select the doubleword that has its parity modified. This operation is performed unconditionally.

The Hit operation accesses the specified cache as normal data references, and performs the specified operation if the cache block contains valid data with the specified physical address (a hit). If the cache block is invalid or contains a different address (a miss), no operation is performed.

Writebacks from the primary cache go to both the secondary cache and tertiary cache/main memory. If the secondary cache is disabled, the data goes to the tertiary cache/main memory. Data comes from the primary data cache, if present, and is modified (the ***W*** bit is set). The address to be written is specified by the cache tag and not the translated physical address.

Writebacks from the secondary cache go to the tertiary cache/main memory. A hit secondary writeback always writes to the most recent data, even if the newest data comes from the primary data cache if it is present and dirty (***W*** bit is set). Otherwise the data comes from the secondary cache. The address to be written is specified by the cache tag and not by the translated physical address.

TLB Refill and TLB Invalid exceptions can occur on any operation. For Index operations (where the physical address is used to index the cache but need not match the cache tag) unmapped addresses may be used to avoid TLB exceptions. This operation never causes TLB Modified or Virtual Coherency exceptions.

Bits 17...16 of the instruction specify the cache as follows:

Code	Name	Cache
0	I	Primary instruction
1	D	Primary data
2	T	Tertiary
3	S	Secondary

Bits (20...18) (this value is listed under the **Code** column) of the instruction specify the operation as follows:

Code	Caches	Name	Operation
0	I	Index Invalidate	Set the cache state of the cache block to Invalid.
0	D	Index Writeback Invalidate	Examine the cache state and Writeback bit (*W* bit) of the primary data cache block at the index specified by the virtual address. If the state is not Invalid and the *W* bit is set, write the block. The address to write is taken from the primary cache tag. Set the cache state of primary cache block to Invalid. The writeback goes to the next level of the memory hierarchy - secondary cache if there is a secondary cache hit and secondary cache is enabled or the tertiary cache/main memory if there is a secondary cache miss.
0	T	Flash Invalidate	Flash invalidate the entire tertiary cache in one operation for tag RAMs that support the block clear function. (Asserts the TcCLR* pin)
0	S	Index Writeback Invalidate	Examine the cache state of the secondary data cache at the index specified by the physical address. If the state is Dirty Exclusive, write back the block to tertiary cache/memory and set the cache state to Invalid. The address to write is taken from the secondary cache tag, which is not necessarily the physical address used to index the cache. The primary data cache is not affected.
1	All	Index Load Tag	Read the tag for the cache block at the specified index and place it into the *TagLo* and *TagHi* CP0 registers, ignoring any parity errors.

Code	Caches	Name	Operation
2	All	Index Store Tag	Write the tag for the cache block at the specified index from the *TagLo* and *TagHi* CP0 registers.
3	D	Create Dirty Exclusive	This operation is used to avoid loading data needlessly from secondary cache or memory when writing new contents into an entire cache block. If the cache block does not contain the specified address, and the block is dirty, write it back to the secondary cache (if present) and to memory. In all cases, set the cache block tag to the specified physical address, set the cache state to Dirty Exclusive.
4	I,D	Hit Invalidate	If the cache block contains the specified address, mark the cache block invalid.
4	S	Hit Invalidate	If the cache block contains the specified address, mark the cache block invalid and also invalidate any matching block, if present, in the primary data cache. The *STag* field of the secondary cache tag is used to determine the locations in the primaries to search. The *CH* bit in the *Status* register is set or cleared to indicate a hit or miss.
5	D	Hit Writeback Invalidate	If the cache block contains the specified address, write the data back if it is dirty, and mark the cache block invalid. The writeback goes to the next level of the memory hierarchy - secondary cache if there is a secondary cache hit and secondary cache is enabled or the tertiary cache/main memory if there is a secondary cache miss.
5	I	Fill	Fill the primary instruction cache block from secondary cache or memory.
5	S	Hit Writeback Invalidate	If the cache block contains the specified address, write back the data if it is dirty and mark the secondary cache block and any matching block in the primary data cache invalid. Modified data in the primary data cache is used during the write back operation. The *STag* field of the secondary cache tag is used to determine the locations in the primaries to check for matching primary blocks. The *CH* bit in the *Status* register is set or cleared to indicate a hit or miss.
5	T	Page Invalidate	The RM7000 processor performs a burst of 128 *Index_Store_Tag* operations to the tertiary cache at the page specified by the effective address, which must be page-aligned. To invalidate the page, the state bits in the *TagLo* register should be zero. Interrupts are deferred during this operation.
6	D	Hit Writeback	If the cache block contains the specified address, and the *W* bit is set, write back the data and clear the *W* bit. The writeback goes to the next level of the memory hierarchy - secondary cache if there is a secondary cache hit and secondary cache is enabled or the tertiary cache/main memory if there is a secondary cache miss.
6	I	Hit Writeback	If the cache block contains the specified address, data is written back unconditionally.
6	S	Hit Writeback	If the cache block contains the specified address, and the cache state is Dirty Exclusive, write back the data to memory and change the cache state to Clean Exclusive. The *CH* bit in the *Status* register is set or cleared to indicate a hit or miss. The writeback operation looks in the primary data cache for modified data. If the data in the primary cache is used, the W bit is cleared.

18.3.0.3 Operation:

```
32, 64T:vAddr ← ((offset15)48 || offset15...0) + GPR[base]
(pAddr, uncached) ← Address Translation (vAddr, DATA)
                    cache operation(op,vAddr,pAddr)

CacheOp (op, vAddr, pAddr)
```

18.3.0.4 Exceptions:

- Coprocessor unusable exception
- Reserved Instruction exception
- TLB invalid exception
- TLB refill exception
- Bus Error exception
- Address error exception

18.4 Implementation Specific Integer Instructions

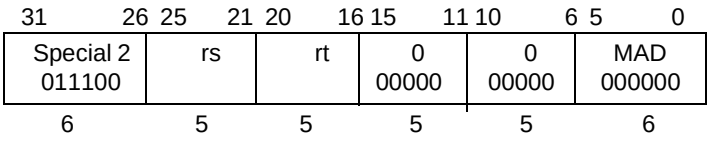
The instruction listed in Table 18.2: are implementation-specific instruction used by the RM7000 integer unit.

Table 18.2: RM7000 Integer Unit Instructions

Integer Instruction	Definition
MAD	Multiply/Add
MADU	Multiply/Add Unsigned
MUL	Multiply

The **MUL** instruction allows the RM7000 to return the multiply result directly to the register file, thereby eliminating the need for a separate instruction to read the *HI/LO* registers. The **MAD** and **MADU** instructions implement an atomic multiply-accumulate operation. These instructions multiply two numbers and add the product to the content of the *HI/LO* registers.

18.4.1 MAD (Multiply/Add)



18.4.1.1 Format:

```
MAD     rs, rt
```

18.4.1.2 Description:

The RM7000 adds a **MAD** instruction (Multiply-accumulate, with the *HI* and *LO* registers as the accumulator) to the base MIPS III ISA. The **MAD** instruction is defined as:

HI,LO ← HI,LO + rs * rt

The Lower 32-bits of the accumulator are stored in the lower 32-bits of the *LO* register, while the upper 32-bits of the result are stored in the lower 32-bits of the *HI* register. This is done to allow this instruction to operate compatibly in 32-bit processors.

The actual repeat rate and latency of this operation are dependent on the size of the operands.

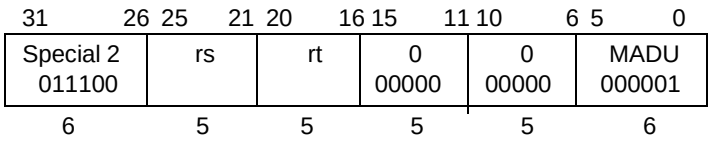
18.4.1.3 Operation:

$$\begin{aligned} T: \quad & temp \leftarrow (HI_{31..0} \,||\, LO_{31..0}) + ((rs_{31})^{32} \,||\, rs_{31..0}) \times ((rt_{31})^{32} \,||\, rt_{31..0}) \\ & HI \leftarrow (temp_{63})^{32} \,||\, temp_{63..32} \\ & LO \leftarrow (temp_{31})^{32} \,||\, temp_{31..0} \end{aligned}$$

18.4.1.4 Exceptions:

— None

18.4.2 MADU (Multiply/Add Unsigned)



18.4.2.1 Format:

```
MADU    rs, rt
```

18.4.2.2 Description:

The RM7000 adds a **MADU** instruction (Multiply-accumulate-unsigned, with the *HI* and *LO* registers as the accumulator) to the base MIPS III ISA. The **MADU** instruction is defined as:

HI,LO ← HI,LO + rs * rt

The Lower 32-bits of the accumulator are stored in the lower 32-bits of the *LO* register, while the upper 32-bits of the result are stored in the lower 32-bits of the *HI* register. This is done to allow this instruction to operate compatibly in 32-bit processors.

The actual repeat rate and latency of this operation are dependent on the size of the operands.

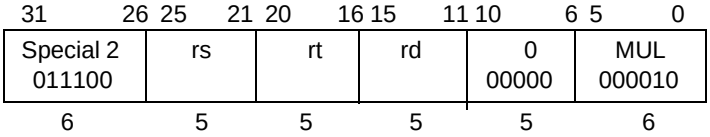
18.4.2.3 Operation:

$$\begin{aligned} T: \quad & temp \leftarrow (HI_{31..0} \,||\, LO_{31..0}) + ((0)^{32} \,||\, rs_{31..0}) \times ((0)^{32} \,||\, rt_{31..0}) \\ & HI \leftarrow (temp_{63})^{32} \,||\, temp_{63..32} \\ & LO \leftarrow (temp_{31})^{32} \,||\, temp_{31..0} \end{aligned}$$

18.4.2.4 Exceptions:

— None

18.4.3 MUL (Multiply)



18.4.3.1 Format:

```
MUL     rd, rs, rt
```

18.4.3.2 Description:

The RM7000 adds a true 3-operand (32 ← 32x32) multiply instruction to the base MIPS III ISA, whereby rd = rs * rt. This instruction eliminates the need to explicitly move the multiply result from the *LO* register back to a general register.

The execution time of this operation is operand size dependent.

The *HI* and *LO* registers are undefined after executing this instruction. For 16-bit operands, the latency of **MUL** is 4 cycles, with a repeat rate of 2 cycles. For 32 x 16-bit operands, this instruction stalls the pipeline for 1 cycle. For 32 x 32-bit operands, this instruction stalls the pipeline for 2 cycles.

18.4.3.3 Operation:

```
T:    Temp ← rs31..0 x rt31..0
      rd ← (temp31)32 || temp31..0
      HI ← undefined
      LO ← undefined
```

18.4.3.4 Exceptions:

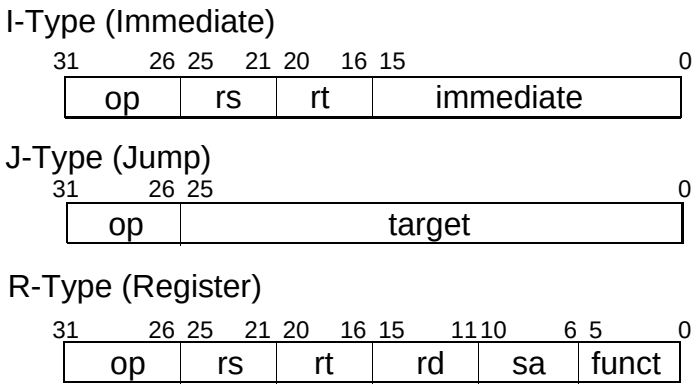
— None

18.5 MIPS IV Instruction Set

The RM7000 processor executes the MIPS IV instruction set, which is a superset of and backward compatible with the MIPS III instruction set. 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.

A summary of the MIPS IV instruction set additions is listed along with a brief explanation of each instruction. For more information on the MIPS IV instruction set, refer to the MIPS IV instruction set manual.

There are three types of instruction types as shown in Figure 18.1.



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 18.3 CPU Instruction Formats

In the MIPS architecture, coprocessor instructions are implementation-dependent.

18.5.1 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 integer load and store instructions directly support is *base register plus 16-bit signed immediate offset*. floating-point load and store instructions also support an indexed addressing, *register + register*, addressing mode.

18.5.1.1 Scheduling a Load Delay Slot

In the RM7000 processor, the instruction immediately following a load instruction can use the contents of the loaded register, however in such cases hardware interlocks insert additional real cycles. Consequently, scheduling load delay slots can be desirable, both for performance and R-Series processor compatibility. However, the scheduling of load delay slots is not absolutely required.

18.5.1.2 Defining Access Types

Access type indicates the size of a data item to be loaded or stored, set by the load or store instruction opcode.

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

In either endianness, bit 0 is always the least-significant bit located at the right-most bit position. Thus, bit designations are always little-endian.

In either endianness, the instruction encoding remains the same. Endianness only affects the word addressing of instructions and does not affect how instructions are encoded within a word. This is true when the instructions are stored in main memory and as well as when they are fetched into the processor. For example, the opcode will always reside in the 6 most significant (left-most) bits within each instruction word (bytes 0 and 1 of a word in big-endian mode; bytes 3 and 2 of a word in little-endian mode).

The processor uses these four data formats:

- a 64-bit doubleword
- a 32-bit word
- a 16-bit half-word
- a 8-bit byte

The processor uses byte addressing for half-word, word and doubleword accesses with the following alignment restrictions:

- 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 are an exception to these alignment restrictions:

LWL, **LWR**, **SWL**, **SWR**, **LDL**, **LDR**, **SDL**, **SDR**

These instructions are used in pairs to provide addressing of misaligned words or of misaligned doublewords.

The access type, together with the three low-order bits of the address, define the bytes accessed within the addressed double-word. Only the combinations shown in Table 18.3: are permissible, All other combinations cause address error exceptions.

Table 18.3: Byte Access within a Doubleword

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							

18.5.2 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

18.5.2.1 64-bit Operations

The RM7000 processor is a 64-bit architecture that supports 32-bit operands. These operands must be sign extended. Thirty-two 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. In addition, 32-bit data is stored sign-extended in a 64-bit register.

18.5.2.2 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.

The RM7000 includes the Multiply (**MUL**), Multiply-add (**MAD**) and Multiply-add Unsigned (**MADU**) instructions and has reduced the latencies and repeat rates of the existing multiply instructions to substantially improve the integer multiply performance. For example, all multiply instructions in the MIPS R5000 write their results to the *HI/LO* register pair. The new **MUL** instruction in the RM7000 returns the multiply result directly to the register file and eliminates the need for a separate instruction to read the *HI/LO* registers. The **MAD** and **MADU** instructions implement the atomic multiply-accumulate to the content of the *HI/LO* registers. The integer multiply latencies and repeat rates have been reduced by one cycle for the word multiply and by two cycles for the doubleword multiply.

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

One exception to this rule are the branch-likely instructions. These instructions execute their *delay slot* instructions only if the branch is taken.

18.5.2.3.1 Overview of Jump Instructions

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 64-bit byte address contained in one of the general purpose registers.

18.5.2.3.2 Overview of Branch Instructions

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 64 bits). All branches occur with a delay of one instruction.

If a conditional branch likely is not taken, the instruction in the delay slot is nullified.

18.5.2.4 Special Instructions

Special instructions allow the software to initiate traps; they are always R-type. Exception instructions are extensions to the MIPS ISA.

18.5.2.5 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.

18.5.3 MIPS IV Instruction Set Additions

The RM7000 microprocessor runs the MIPS IV instruction set, which is a superset of and backward compatible with the MIPS III instruction set. These new instructions enable the MIPS architecture to compete in markets that have traditionally been dominated by vector architectures.

A set of compound multiply-add instructions has been added, taking advantage of the fact that the majority of floating-point computations use the chained multiply-add paradigm. The intermediate multiply result is rounded before the addition is performed.

A *register + register* addressing mode for floating-point loads and stores has been added which eliminates the extra integer add required in many array accesses. However, issuing of a *register + register* load causes a one cycle stall in the pipeline, which makes it useful only for compatibility with other MIPS IV implementations. A *register + register* addressing mode for integer memory operations is not supported.

A set of four conditional move operators allows floating-point arithmetic ‘IF’ statements to be represented without branches. ‘THEN’ and ‘ELSE’ clauses are computed unconditionally and the results placed in a temporary register. Conditional move operators then transfer the temporary results to their true register. Conditional moves must be able to test both integer and floating-point conditions in order to supply the full range of IF statements. Integer tests are performed by comparing a general register against a zero value.

floating-point tests are performed by examining the floating-point condition codes. Since floating-point conditional moves test the floating-point condition code, the RM7000 provides 8 condition codes to give the compiler increased flexibility in scheduling the comparison and the conditional moves. Table 18.4: lists in alphabetical order the new instructions which comprise the MIPS IV instruction set.

Table 18.4: MIPS IV Instruction Set Additions and Extensions

Instruction	Definition
BC1F	Branch on FP Condition Code False
BC1T	Branch on FP Condition Code True
BC1FL	Branch on FP Condition Code False Likely
BC1TL	Branch on FP Condition Code True Likely
C.cond.fmt (cc)	Floating Point Compare
LDXC1	Load Double Word indexed to COP1
LWXC1	Load Word indexed to COP1
MADD.fmt	Floating Point Multiply-Add
MOVF	Move conditional on FP Condition Code False
MOVN	Move on Register Not Equal to Zero
MOVT	Move conditional on FP Condition Code True
MOVZ	Move on Register Equal to Zero
MOVF.fmt	FP Move conditional on Condition Code False
MOVN.fmt	FP Move on Register Not Equal to Zero
MOVT.fmt	FP Move conditional on Condition Code True
MOVZ.fmt	FP Move conditional on Register Equal to Zero
MSUB.fmt	Floating Point Multiply-Subtract
NMADD.fmt	Floating Point Negative Multiply-Add
NMSUB.fmt	Floating Point Negative Multiply-Subtract
PREFX	Prefetch Indexed --- Register + Register
PREF	Prefetch --- Register + Offset
RECIP.fmt	Reciprocal Approximation
RSQRT.fmt	Reciprocal Square Root Approximation
SDXC1	Store Double Word indexed to CP1
SWXC1	Store Word indexed to CP1

18.5.3.1 Summary of Instruction Set Additions

The following is a brief description of the additions to the MIPS III instruction set. These additions comprise the MIPS IV instruction set.

18.5.3.1.1 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 instructions are upward compatible extensions of the Branch on Floating-Point Coprocessor instructions of the MIPS instruction set. 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. 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 is 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.

18.5.3.1.2 Floating Point Compare

C.COND.FMT - Compare the contents of two FPU registers

The contents of the two FPU 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.

18.5.3.1.3 Indexed Floating Point Load

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

The two Index Floating Point Load instructions are exclusive to the MIPS IV instruction set and transfer floating-point data types from memory to the floating-point registers using the *register + register* addressing mode. There are no indexed loads to general registers. 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.

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.

18.5.3.1.4 Integer Conditional Moves

MOVT - Move conditional on condition code true
MOVF - Move conditional on condition code false
MOVN - Move conditional on register not equal to zero
MOVZ - Move conditional on register equal to zero

The four integer move instructions are exclusive to the MIPS IV instruction set and are used to test a condition code or a general register and then conditionally perform an integer move. The value of the floating-point condition code specified in the instruction 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 contents of the specified source register is copied into the specified destination register.

18.5.3.1.5 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 exclusive to the MIPS IV instruction set and 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 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.

18.5.3.1.6 Floating Point Multiply-Add

MADD.FMT - Floating Point Multiply-Add
MSUB.FMT - Floating Point Multiply-Subtract
NMADD.FMT - Floating Point Negative Multiply-Add
NMSUB.FMT - Floating Point Negative Multiply-Subtract

These four instructions are exclusive to the MIPS IV instruction set and accomplish two floating-point operations with one instruction. Each of these four instructions performs intermediate rounding.

18.5.3.1.7 Prefetch

PREF - Register + offset format
PREFX - Register + register format

The two prefetch instructions are exclusive to the MIPS IV instruction set and allow the compiler to issue instructions early so the corresponding data can be fetched and placed as close as possible to the CPU. Each instruction contains a 5-bit 'hint' field which gives the coherency status of the line being prefetched. The line can be either shared, exclusive clean, or exclusive dirty. The contents of the general register specified by the base is added either to the 16 bit sign-extended offset or to the contents of the general register specified by the index to form a virtual address. This address together with the 'hint' field is sent to the cache controller and a memory access is initiated.

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. The prefetch instruction never generates TLB-related exceptions. The **PREF** instruction is considered a standard processor instruction while the **PREFX** instruction is considered a standard Coprocessor 1 instruction.

Table 18.5: RM7000 Prefetch (PREF) Instructions

Hint Field	Name	Data use and desired prefetch action
31, 29..0	Load	Fetch data as if for a load
30	PrepareFor-Store	On primary dcache miss, allocate cacheline for requested address. Cacheline state is set to valid, the W bit is set and the data field is filled with all zeros. No memory transaction is done. If the victim cacheline was valid and dirty, it is written back. On primary dcache hit, no action is taken.

18.5.3.1.8 Reciprocal's

RECIP.FMT - Reciprocal
RSQRT.FMT - Reciprocal Square Root

The reciprocal instruction performs a reciprocal on a floating-point value. The reciprocal of the value in the floating-point source register is placed in a destination register.

The reciprocal square root instruction performs a reciprocal square root on a floating-point value. The reciprocal of the positive square root of a value in the floating-point source register is placed in a destination register.

The **RECIP** and **RSQRT** instructions comply with IEEE accuracy requirements. The rounding mode for the square-root intermediate value of the **RSQRT** instruction is RZ (round towards zero).

The **RECIP** instruction has the same latency as a **DIV** instruction, but a **RSQRT** is faster than a **SQRT** followed by a **RECIP**.

Refer to Appendix C, "Cycle Time and Latency Tables", for the various operation latency tables.

18.5.3.1.9 Indexed Floating Point Store

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

The two Index Floating Point Store instructions are exclusive to the MIPS IV instruction set and transfer floating-point data types from the floating-point registers to memory using register + register addressing mode. There are no indexed loads to general registers. 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.

Appendix A CP0 Register Map

The RM7000 includes two sets of CP0 registers. Each register set contains thirty-two 32-bit registers.

The register number is encoded in bits [15:11] of the instruction. Table A.1: lists those CP0 registers accessible through the **MTC0**, **DMTC0**, **MFC0**, and **DMFC0** instructions. Table A.2: lists the second set of CP0 registers. These registers are accessible through the **CFC0** and **CTC0** instructions.

Table A.1: CP0 Registers, Set 0

Register Number	Register Name	Register Description	Read/Write	Located in
0	Index	Six bit TLB index	r/w	Section 5
1	Random	Random TLB index	r	Section 5
2	EntryLo0	Even page PFN	r/w	Section 5
3	EntryLo1	Odd page PFN	r/w	Section 5
4	Context	Page table entry pointer	r/w	Section 13
5	PageMask	Sets TLB page size	r/w	Section 5
6	Wired	Locks TLB entries	r/w	Section 5
7	Info	CPU Information Register	r/w	Section 10
8	BadVAddr	Bad virtual address	[31:1] = r [0] = r/w	Section 13
9	Count	Pipeline cycle counter	r/w	Section 13
10	EntryHi	Upper bits of TLB entry	r/w	Section 5
11	Compare	Count register compare value	r/w	Section 13
12	Status	Execution mode information	r/w	Section 13
13	Cause	Exception cause	r/w	Section 13
14	EPC	Exception program counter	r/w	Section 13
15	PRId	Processor Revision ID	r	Section 5
16	Config	Processor Configuration	[31:13], [11:4] = r [12, 3:0] = r/w	Section 5
17	LLAddr	Load Linked instruction address	r/w	Section 5
18	Watch1	Breakpoint address for Watch exception	r/w	Section 17
19	Watch2	Breakpoint address for Watch exception	r/w	Section 17
20	XContext	Page table entry pointer for 64-bit addressing mode	r/w	Section 13
21	Reserved		---	---
22	PerfControl	Performance counter control	r/w	Section 17
23	Reserved		---	---
24	WatchMask	Mask Watch register address bits	r/w	Section 17
25	PerfCount	Performance counter	r/w	Section 17
26	ECC	Error checking and correction	r/w	Section 13
27	CacheErr	Parity error status	r	Section 13
28	TagLo	Tag state and control bits	r/w	Section 5

Register Number	Register Name	Register Description	Read/Write	Located in
29	TagHi	Tag physical address bits	r/w	Section 5
30	ErrorEPC	Parity error program counter	r/w	Section 5
31	Reserved		---	---

Table A.2: CP0 Registers, Set 1

Register Number	Register Name	Register Description	Read/Write	Located in
0 - 17	Reserved		---	---
18	IPLLo	Interrupt priority level for interrupts [7:0]	r/w	Section 14
19	IPLHi	Interrupt priority level for interrupts [15:8]	r/w	Section 14
20	IntCtl	Interrupt Control	r/w	Section 14
21-25	Reserved		---	---
26	DErrAddr0	Imprecise error address	r	Section 12
27	DErrAddr1	Imprecise error address	r	Section 12
28-31	Reserved		---	---

Appendix B Instruction Hazards

This appendix identifies the RM7000 instruction hazards. Certain combinations of instructions are not permitted because their results are unpredictable in combination with 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.

The integer multiplier/divide unit on the RM7000 fully interlocks all register writes to the *HI*/*LO* registers. This removes any code restrictions on when those registers can be written or read.

For the following code hazards, the behavior is undefined and unpredictable. The following is a list of instruction hazards.

- Any instruction that would modify the *PageMask, EntryHi, EntryLo0, EntryLo1,* or *Random* CP0 Registers should not be followed by a **TLBWR** instruction. There should be at least four integer instructions (2 CPU cycles) between the register modification and the **TLBWR** instruction.
- Any instruction that would modify *PageMask, EntryHi, EntryLo0, EntryLo1,* or *Index* CP0 Registers should not be followed by a **TLBWI** instruction. There should be at least four integer instructions (2 CPU cycles) between the register modification and the **TLBWI** instruction.
- Any instruction that would modify the *Index* CP0 Register or the contents of the JTLB should not be followed by a **TLBR** instruction. There should be at least four integer instructions (2 CPU cycles) between the register modification and the **TLBR** instruction.
- Any instruction that would modify the *PageMask* or *EntryHi* or CP0 Registers or the contents of the JTLB should not be followed by a **TLBP** instruction. There should be at least four integer instructions (2 CPU cycles) between the register modification and the **TLBP** instruction.
- Any instruction that would modify the *PageMask* or *EntryHi* CP0 Registers or the contents of the JTLB should not be followed by any instruction that uses the JTLB. There should be at least four integer instructions (2 CPU cycles) between the register modification and the use of the JTLB.
- Any instruction that would modify the *EPC, ErrorEPC,* or *Status* CP0 Registers should not be followed by an **ERET** instruction. There should be at least four integer instructions (2 CPU cycles) between the register modification and the **ERET** instruction.
- If any instruction modifies the ***K0*** field of the *Config* register, the kseg0 and ckseg0 address segments should not be used for 5 CPU cycles. The register modification and the use of these address segments should be separated by at least ten integer instructions.
- If the data cache is to be locked, it is recommended that the two instructions that follow the instruction that sets ***SR.DL*** should not cause a data cache miss. This miss might cause a cache refill into locked set that the user did not expect. The register modification and the next data cache miss should be separated by at least four integer instructions.
- If the instruction cache is to be locked, it is recommended that the four instructions that follow the instruction that sets ***SR.IL*** should not cause an instruction cache miss. This miss might cause a cache refill into the locked set of the instruction cache that the user did not expect. The register modification and next instruction cache miss should be separated by at least eight integer instructions.
- A branch or jump instruction is not allowed to be in the delay-slot of another branch/jump instruction. This sequence is illegal in the MIPS architecture.
- An **ERET** instruction is not allowed to be in the delay-slot of a branch/jump instruction. This sequence is illegal in the MIPS architecture.
- Any instruction that modifies a control register (**MTC0**, **CTC0**, **CTC1**) should not be placed in the delay-slot of a branch/jump instruction. The effect of the control register modification can occur after the branch target has been fetched and executed. This is a code hazard if the software expects the state change to occur before the branch target is fetched and executed.

- Modifying the TLB registers or entries is not allowed in mapped address space.

Appendix C Cycle Time and Latency Tables

This appendix lists the latency and repeat cycles for integer and floating point operations as well as the cycle counts for cache operations. Table C.1: shows the latency and repeat rates for integer multiply and divide operations. Table C.2: shows the latency and repeat rates for floating point operations.

Table C.1: Integer Multiply and Divide Operations

Opcode	Operand Size	Latency	Repeat Rate	Stall Cycles
mult/u, mad/u	16 bit	4	3	0
	32 bit	5	4	0
mul	16 bit	4	2	1
	32 bit	5	3	2
dmult, dmultu	any	9	8	0
div, divd	any	36	36	0
ddiv, ddivu	any	68	68	0

Table C.2: Floating Point Operations

Opcode	Latency	Repeat
fadd (sngl/dbl)	4	1
fsub (sngl/dbl)	4	1
fmult (sngl/dbl)	4/5	1/2
fmadd (sngl/dbl)	4/5	1/2
fmsub (sngl/dbl)	4/5	1/2
fdiv (sngl/dbl)	21/36	19/34
fsqrt (sngl/dbl)	21/36	19/34
frecip (sngl/dbl)	21/36	19/34
frsqrt (sngl/dbl)	38/68	36/66
fcvt.s.d	4	1
fcvt.s.w	6	3
fcvt.s.l	6	3
fcvt.d.s	4	1
fcvt.d.w	4	1
fcvt.d.l	4	1
fcvt.w.s	4	1
fcvt.w.d	4	1
fcvt.l.s	4	1
fcvt.l.d	4	1
fcmp (sngl/dbl)	1	1
fmov (sngl/dbl)	1	1
fmovc (sngl/dbl)	1	1
fabs (sngl/dbl)	1	1

Opcode	Latency	Repeat
fneg (sngl/dbl)	1	1
lwc1, lwxc1	2	1
ldc1, ldxc1	2	1
swc1, swxc1	2	1
sdc1, sdxc1	2	1
mtc1, dmtc1	2	1
mfc1, dmfc1	2	1

C.1 Cycle Counts for RM7000 Cache Misses

To describe processor sequences that include a memory access, the number of cycles must be calculated based on the system response to a memory access. Such sequences will be described with equations based on the following mnemonics:

- SYSDIV: The number of processor cycles per system cycle, ranges from 2 - 8.
- ML: Number of system cycles of memory latency defined as the number of cycles the SysAD bus is driven by the external agent before the first doubleword of data appears.
- DD: Number of system cycles required to return the block of data, defined to be the number of cycles beginning when the first doubleword of data appears on the SysAD bus and ending when the last double word of data appears on the SysAD bus inclusive.
- {0 to (SYSDIV - 1)}: In many equations this term is used. It has a value (number of cycles) between 0 and (SYSDIV - 1) depending on the alignment of the execution of the cache miss or cache op with the system clock.

C.1.1 Primary Data Cache Misses

1. All cycle counts are in processor cycles.
2. Data cache misses have lower priority than write backs, external requests, and instruction cache misses. If the writeback buffer contains unwritten data when a data cache miss occurs, the buffer is retired before the handling of the data cache miss is begun. Instruction cache misses are given priority over data cache misses. If an instruction cache miss occurs at the same time as a data cache miss, the instruction cache miss is handled first. External requests are completed before beginning the handling of a data cache miss.
3. In handling a data cache miss a writeback may be required which will fill a writeback buffer. Writebacks can affect subsequent cache misses since they can stall until the write back buffer is written back to memory.
4. All cycle counts are best case assuming no interference from the mechanisms described above.

C.1.2 Primary Instruction Cache Misses

1. All cycle counts are in processor cycles.
2. Instruction cache misses have lower priority than writebacks and external requests. If the writeback buffer contains unwritten data when an instruction cache miss occurs, the buffer is retired before the handling of the instruction cache miss is begun. External requests are completed before beginning the handling of an instruction cache miss.
3. All cycle counts are best case assuming no interference from the mechanisms described above.

C.2 Cycle Counts for Cache Instructions

1. All cycle counts are in processor cycles.
2. All cache ops have lower priority than cache misses, writebacks and external requests. If the write-back buffer contains unwritten data when a cache op is executed, the buffers is 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 is handled first. Cache ops are mutually exclusive with respect to data cache misses. External requests are completed before beginning a cache op.

3. For some 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 such 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.
4. Cache ops of the form xxxx_Writeback_xxxx may perform a writeback which will fill the writeback buffer. Writebacks can affect subsequent cache ops since they can stall until the buffer is written back to memory. Cache ops which fill the write back buffer are noted in the following tables.
5. All cycle counts are best case assuming no interference from the mechanisms described above.

Table C.3: Primary Data Cache Operations

Code	Name	Number of Cycles
0	Index_Writeback_Invalidate_D	10 Cycles
1	Index_Load_Tag_D	10 Cycles
2	Index_Store_Tag_D	7 Cycles
3	Create_Dirty_Exclusive_D	10 Cycles for a cache hit. 13 Cycles for a cache miss if the cache line is clean. 13 Cycles for a cache miss if the cache line is dirty. (Writeback)
4	Hit_Invalidate_D	1 Cycle for a cache miss. 3 Cycles for a cache hit.
5	Hit_Writeback_Invalidate_D	1 Cycle for a cache miss. 4 Cycles for a cache hit if the cache line is clean. 4 Cycles for a cache hit if the cache line is dirty.
6	Hit_Writeback_D	7 Cycles for a cache miss. 10 Cycles for a cache hit if the cache line is clean. 12 Cycles for a cache hit if the cache line is dirty.

Table C.4: Primary Instruction Cache Operations

Code	Name	Number of Cycles
0	Index_Invalidate_I	8 Cycles.
1	Index_Load_Tag_I	8 Cycles.
2	Index_Store_Tag_I	11 Cycles.
3	NA	
4	Hit_Invalidate_I	8 Cycles for a cache miss. 10 Cycles for a cache hit.
5	Fill_I	13 Cycles for secondary cache hit 2 + {0 to (SYSDIV-1)} + (6 x SYSDIV) + 2 Cycles for secondary cache miss, tertiary cache hit 10 + {0 to (SYSDIV -1)} + (2 x SYSDIV) + (ML x SYSDIV) + (DD x SYSDIV) + 2 Cycles for secondary cache miss, tertiary cache miss
6	Hit_Writeback_I	8 Cycles for a cache miss. 22 Cycles for a cache hit. (Writeback)

Table C.5: Secondary Cache Operations

Code	Name	Number of Cycles
0	Index_Writeback_Invalidate_S	13 Cycles for no primary cache writeback and no secondary cache writeback. 16 Cycles for no primary cache writeback and secondary cache writeback. 19 Cycles for primary cache writeback and no secondary cache writeback. 22 Cycles for both primary cache and secondary cache writebacks.
1	Index_Load_Tag_S	15 Cycles.
2	Index_Store_Tag_S	13 Cycles.
3	NA	
4	Hit_Invalidate_S	7 Cycles for a secondary cache miss. 9 Cycles for a secondary cache hit.
5	Hit_Writeback_Invalidate_S	7 Cycles for no primary cache writeback and secondary cache miss. 13 Cycles for primary cache writeback and secondary cache miss. 10 Cycles for no primary cache writeback, secondary cache hit and no secondary cache writeback. 13 Cycles for no primary cache writeback, secondary cache hit and secondary cache writeback. 16 Cycles for primary cache writeback, secondary cache hit and no secondary cache writeback. 19 Cycles for primary cache writeback, secondary cache hit and secondary cache writeback.
6	Hit_Writeback_S	9 Cycles for a primary cache miss and secondary cache miss 12 Cycles for a primary cache miss and secondary cache hit and no secondary cache writeback. 15 Cycles for a primary cache miss and secondary cache hit and secondary cache writeback. 10 Cycles for a primary cache hit and no primary cache writeback and secondary cache miss. 15 Cycles for a primary cache hit and primary cache writeback and secondary cache miss. 12 Cycles for a primary cache hit and no primary cache writeback, secondary cache hit and no secondary cache writeback. 15 Cycles for a primary cache hit and no primary cache writeback, secondary cache hit and secondary cache writeback. 18 Cycles for a primary cache hit and primary cache writeback, secondary cache hit and no secondary cache writeback. 21 Cycles for a primary cache hit and primary cache writeback, secondary cache hit and secondary cache writeback

Table C.6: Tertiary Cache Operations

Code	Name	Number of Cycles
0	Flash_Invalidate_T	13 + {0 to (SYSDIV-1)} + (1 x SYSDIV) + 3 Cycles
1	Index_Load_Tag_T	13 + {0 to (SYSDIV-1)} + (4 x SYSDIV) + 4 Cycles
2	Index_Store_Tag_T	13 + {0 to (SYSDIV-1)} + (1 x SYSDIV) + 3 Cycles
3	NA	
4	NA	
5	Page_Invalidate_T	13 + {0 to (SYSDIV-1)} + (128 x SYSDIV) + 3 Cycles
6	NA	

Appendix D Subblock Order

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 appendix describes these retrieval methods, with an emphasis on subblock ordering.

Sequential ordering retrieves the data elements of a block in serial, or sequential, order.

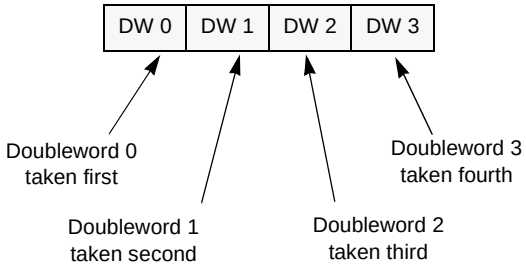


Figure D.1 Retrieving a Data Block in Sequential Order

Figure D.1 shows a sequential order in which doubleword 0 is taken first and doubleword 3 is taken last.

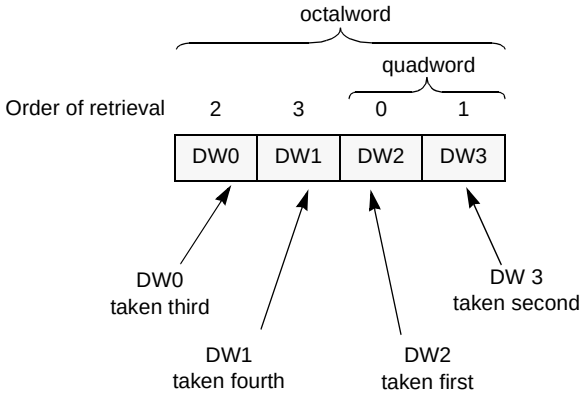


Figure D.2 Retrieving Data in a Subblock Order

Subblock ordering allows the system to define the order in which the data elements are retrieved. The smallest data element of a block transfer for the RM7000 is a doubleword, and Figure D.2 shows the retrieval of a block of data that consists of 4 doublewords, in which DW2 is taken first.

Using the subblock ordering shown in Figure D.2, the doubleword at the target address is retrieved first (DW2), followed by the remaining doubleword (DW3) in this quadword.

It may be easier way to understand subblock ordering by taking a look at the method used for generating the address of each doubleword 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, starting at doubleword zero (00_2).

Using this scheme, Table D.1: through Table D.3: 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, and 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 found by taking the XOR of address 10_2 with the binary count of DW2, 10_2. The result is 00_2, or DW0.

The remaining tables illustrate this method of subblock ordering, using various address permutations.

Table D.1: Subblock Ordering Sequence: Address 10_2

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 D.2: Subblock Ordering Sequence: Address 11_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 D.3: Subblock Ordering Sequence: Address 01_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

Appendix E JTAG Interface

The RM7000 processor implements JTAG boundary scan to facilitate board testing. The JTAG interface is fully compliant with the IEEE 1149.1 standard and supports the following mandatory boundary scan instructions: **BYPASS**, **EXTEST**, and **SAMPLE/PRELOAD**.

After power is applied to the part, internal power-on-reset logic resets the JTAG function and the function remains reset until **JTMS** is asserted and **JTCK** is running.

The normal reset sequence, which involves the assertion of **Reset***, **ColdReset*** and **VccOk**, does not initialize the JTAG function. This is in compliance with the IEEE 1149.1 standard which does not allow a device reset to initialize the JTAG logic.

The BSDL files for each processor type can be downloaded at:

http://www.qedinc.com.

E.1 Test Data Registers

The RM7000 processor test logic contains three registers; *Bypass*, *Boundary Scan*, and *Instruction*.

E.1.1 Bypass Register

The *Bypass* register is a one-bit shift register that provides a connection between the Test Data In (**JTDI**) and Test Data Out (**JTDO**) pins when no other test data registers are selected. The *Bypass* register allows for the rapid movement of test data to and from other board components without affecting the normal operation of these components. When using the *Bypass* register, data is transferred without inversion from **JTDI** to **JTDO**.

E.1.2 Boundary Scan Register

The *Boundary Scan* register is a single shift-register containing cells which connect to all of the input and output pins of the RM7000 processor. This register allows for the testing of board interconnections to detect defects such as opens circuits and short circuits. Figure E.1 shows the logical structure of the *Boundary Scan* register.

Input cells only capture data and do not affect processor operation. Data is transferred without inversion from **JTDI** to **JTDO** through the *Boundary Scan* register during scanning. The *Boundary Scan* register is operated using the **EXTEST** and **SAMPLE /PRELOAD** instructions.

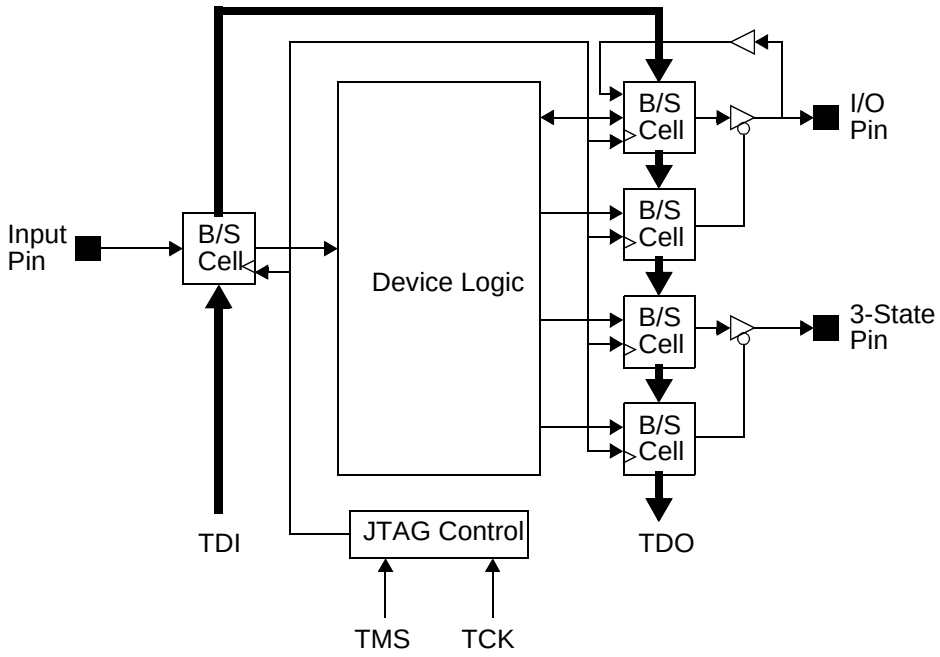


Figure E.1 JTAG Boundary Scan Cell Organization

E.1.3 Instruction Register

The *Instruction* register selects the boundary scan test to be performed and the test data register to be accessed. The *Instruction* register is three bits wide. The execution of the boundary scan instructions is controlled by the Test Access Port (TAP) controller. Refer to section E.3, “TAP Controller”, for more information on the TAP controller and its states.

Table E.1: shows the encoding of the instructions supported by the RM7000.

Table E.1: JTAG Instruction Register Encoding

Instruction Code	Instruction
000	EXTEST
001-101	Reserved
110	SAMPLE/PRELOAD
111	BYPASS

E.2 Boundary Scan Instructions

The RM7000 processor supports three instructions; **EXTEST**, **SAMPLE/PRELOAD**, and **BYPASS**. These instructions are encoded in the *Instruction* register as shown in section E.1.3, “Instruction Register”. The following subsections describe these instructions in more detail.

E.2.1 EXTEST

The **EXTEST** instruction selects the boundary scan cells to be connected between **JTDI** and **JTDO**. Execution of the **EXTEST** instruction causes the output boundary scan cells to drive the output pins of the RM7000 processor. Values scanned into the register become the output values. The input boundary scan cells sample the input pins of the RM7000 processor. Bidirectional pins can be either inputs or outputs depending on their control setting.

The **EXTEST** instruction uses the following TAP controller states. Refer to section E.3, “TAP Controller”, for more information on the TAP controller:

- The RM7000 outputs the preloaded data to the pins at the falling edge of **JTCK** in the *Update_IR* TAP controller state. The JTAG instruction register is updated with the **EXTEST** instruction.
- The **EXTEST** instruction selects the cells to be tested in the *Shift_DR* TAP controller state.
- Once the **EXTEST** instruction has been executed, the output pins can change state on the falling edge of **JTCK** in the *Update_DR* TAP controller state.

After execution of the **EXTEST** instruction, the RM7000 processor must be reset.

E.2.2 SAMPLE/PRELOAD

The **SAMPLE/PRELOAD** instruction is used to sample the state of the device pins. Execution of the **SAMPLE/PRELOAD** instruction causes the output boundary scan cells to sample the value driven by the RM7000 processor. Input boundary scan cells sample their corresponding input pins on the rising edge of **JTCK**. I/O pins can be driven by either the RM7000 processor or external logic. The values shifted to the input latches are not used by internal logic.

The **SAMPLE/PRELOAD** instruction uses the following TAP controller states:

- The **SAMPLE/PRELOAD** instruction selects the cells to be tested in the *Shift_DR* TAP controller state.
- The state of the pins to be tested are sampled on the rising edge of **JTCK** in the *Capture_DR* TAP controller state.
- The boundary scan cells are latched into the output latches on the falling edge of **JTCK** in the *Update_DR* TAP controller state.

This instruction can also be used to preload the boundary scan output cells with specific values. These preloaded values are then enabled to the output pins using the **EXTEST** instruction.

E.2.3 BYPASS

The **BYPASS** instruction is used to bypass a component that is connected in series with other components. This allows for rapid movement of data through the various components on the board by bypassing those that do not need to be tested. The **BYPASS** instruction is forced onto the instruction register output latches during the *Test_Logic_Reset* state. When the **BYPASS** instruction is executed, test data is passed from **JTDI** to **JTDO** via the single-bit *Bypass* register, effectively bypassing the RM7000 processor.

This instruction can be entered by holding **JTDI** at a HIGH logic level while completing an instruction scan cycle. This allows for easier access to a specific device on a multi-device board.

E.3 TAP Controller

The Test Access Port (TAP) controller is a synchronous state machine that controls the test logic sequence of operations. The TAP controller changes state on the rising edge of **JTCK**, and during power-up.

The value of the Test Mode Select (**JTMS**) signal on the rising edge of **JTCK** controls the state transitions of the TAP controller state machine. Figure E.2 shows a diagram of the TAP controller state machine.

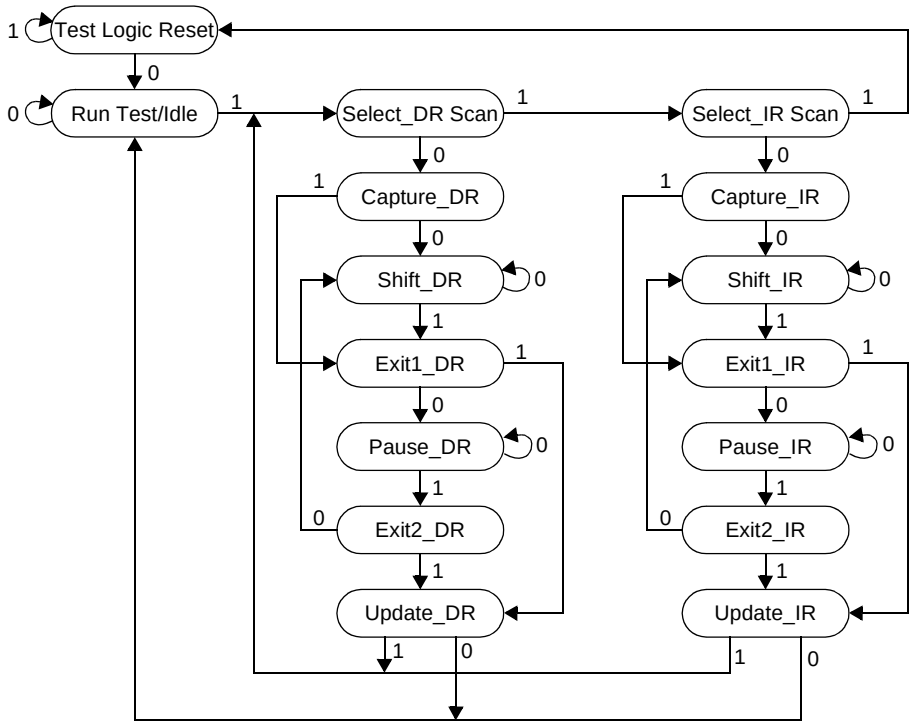


Figure E.2 TAP Controller State Diagram

E.3.1 Test-Logic-Reset State

In the *Test-Logic-Reset* state the boundary scan test logic is disabled, allowing normal operation of the RM7000 processor to continue. The test logic enters the *Test-Logic-Reset* state when the **JTMS** input is held HIGH for at least five rising edges of **JTCK**. The **BYPASS** instruction is forced into the instruction register output latches during this state. The controller remains in the *Test-Logic-Reset* state as long as **JTMS** is HIGH.

If the controller transitions from the *Test-Logic-Reset* state as a result of an erroneous low signal on **JTMS** (for one rising edge of **JTCK**), the controller returns to the *Test-Logic-Reset* state if **JTMS** becomes HIGH for three rising edges of **JTCK**. The operation of the test logic is such that, should the above condition occur, no disturbance is caused to the on-chip system logic. When the state machine transitions from the *Test-Logic-Reset* state to the *Run-Test/Idle* state, no action is taken because the current instruction has been set to select operation of the *Bypass* register. The test logic is also inactive in the *Select_DR* and *Select_IR* controller states.

E.3.2 Run-Test/Idle State

The controller enters the *Run-Test/Idle* state between scan operations. The controller remains in this state as long as **JTMS** is held LOW. The instruction register and all test data registers retain their previous state. The instruction cannot change when the TAP controller is in this state.

When **JTMS** is sampled HIGH at the rising edge of **JTCK**, the controller transitions to the *Select_DR* state.

E.3.3 Select_DR_Scan State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If **JTMS** is sampled LOW at the rising edge of **JTCK**, the controller transitions to the *Capture_DR* state. A HIGH on **JTMS** causes the controller to transition to the *Select_IR* state. The instruction cannot change while the TAP controller is in this state.

E.3.4 Select_IR_Scan State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If **JTMS** is sampled LOW at the rising edge of **JTCK**, the controller transitions to the *Capture_IR* state. A HIGH on **JTMS** causes the controller to transition to the *Test-Reset-Logic* state. The instruction cannot change while the TAP controller is in this state.

E.3.5 Capture_DR State

In this state the boundary scan register captures input pin data is the current instruction is either **EXTEST** or **SAMPLE/PRELOAD**. If **JTMS** is sampled LOW at the rising edge of **JTCK**, the controller transitions to the *Shift_DR* state. A HIGH on **JTMS** causes the controller to transition to the *Exit1_DR* state. The instruction cannot change while the TAP controller is in this state.

E.3.6 Shift_DR State

In this state the test data register connected between **JTDI** and **JTDO** as a result of the current instruction shifts data one stage toward its serial output on the rising edge of **JTCK**. If **JTMS** is sampled LOW at the rising edge of **JTCK**, the controller remains in the *Shift_DR* state. A HIGH on **JTMS** causes the controller to transition to the *Exit1_DR* state. The instruction cannot change while the TAP controller is in this state.

E.3.7 Exit1_DR State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If **JTMS** is sampled LOW at the rising edge of **JTCK**, the controller transitions to the *Pause_DR* state. A HIGH on **JTMS** causes the controller to transition to the *Update_DR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

E.3.8 Pause_DR State

The Pause_DR state allows the controller to temporarily halt the shifting of data through the test data register in the serial path between **JTDI** and **JTDO**. All test data registers selected by the current instruction retain their previous state. If **JTMS** is sampled LOW at the rising edge of **JTCK**, the controller remains in the *Pause_DR* state. A HIGH on **JTMS** causes the controller to transition to the Exit2*_DR* state. The instruction cannot change while the TAP controller is in this state.

E.3.9 Exit2_DR State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If **JTMS** is sampled LOW at the rising edge of **JTCK**, the controller transitions to the *Shift_DR* state to allow another serial shift of data. A HIGH on **JTMS** causes the controller to transition to the *Update_DR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

E.3.10 Update_DR State

The boundary scan register includes a latched parallel output to prevent changes at the parallel output while data is shifted in response to the **EXTEST** and **SAMPLE/PRELOAD** instructions. When the TAP controller is in this state and the *Boundary Scan* register is selected, data is latched into the parallel output of this register from the shift-register path on the falling edge of **JTCK**. Data held at this latched parallel output only changes during this state.

If **JTMS** is sampled LOW at the rising edge of **JTCK**, the controller transitions to the *Run-Test/Idle* state. A HIGH on **JTMS** causes the controller to transition to the *Select_DR_Scan* state. The instruction cannot change while the TAP controller is in this state and all shift register stages in the test data registers selected by the current instruction retain their previous state.

E.3.11 Capture_IR State

In this state the shift register contained in the *Instruction* register loads a fixed pattern (001) on the rising edge of **JTCK**. The test data registers selected by the current instruction retain their previous state.

If **JTMS** is sampled LOW at the rising edge of **JTCK**, the controller transitions to the *Shift_IR* state. A HIGH on **JTMS** causes the controller to transition to the *Exit1_IR* state. The instruction cannot change while the TAP controller is in this state.

E.3.12 Shift_IR State

In this state the test data register connected between **JTDI** and **JTDO** as a result of the current instruction shifts data one stage toward its serial output on the rising edge of **JTCK**. If **JTMS** is sampled LOW at the rising edge of **JTCK**, the controller remains in the *Shift_IR* state. A HIGH on **JTMS** causes the controller to transition to the *Exit1_IR* state. The instruction cannot change while the TAP controller is in this state.

E.3.13 Exit1_IR State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If **JTMS** is sampled LOW at the rising edge of **JTCK**, the controller transitions to the *Pause_IR* state. A HIGH on **JTMS** causes the controller to transition to the *Update_IR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state and the instruction register retains its previous state.

E.3.14 Pause_IR State

The Pause_DR state allows the controller to temporarily halt the shifting of data through the test data register in the serial path between **JTDI** and **JTDO**. All test data registers selected by the current instruction retain their previous state. If **JTMS** is sampled LOW at the rising edge of **JTCK**, the controller remains in the *Pause_IR* state. A HIGH on **JTMS** causes the controller to transition to the Exit2*_IR* state. The instruction cannot change while the TAP controller is in this state.

E.3.15 Exit2_IR State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If **JTMS** is sampled LOW at the rising edge of **JTCK**, the controller transitions to the *Shift_IR* state to allow another serial shift of data. A HIGH on **JTMS** causes the controller to transition to the *Update_IR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

E.3.16 Update_IR State

The instruction shifted into the *Instruction* register is latched onto the parallel output from the shift register path on the falling edge of **JTCK**. Once the instruction has been latched it becomes the new instruction.

If **JTMS** is sampled LOW at the rising edge of **JTCK**, the controller transitions to the *Run-Test/Idle* state. A HIGH on **JTMS** causes the controller to transition to the *Select_IR_Scan* state.

E.4 TAP Controller Initialization

In the RM7000, the TAP controller is initialized during power-on reset. The TAP controller can be initialized by driving the **JTMS** input HIGH for at least five **JTCK** periods. This places the TAP controller in the *Test_Logic_Reset* state. The RM7000 does not support the JTAG TRST* signal.

E.5 Boundary Scan Signals

The RM7000 supports those signals listed in Table E.2:. The Test-Reset (TRST*) input is not supported. The **JTMS** input must be used to initialize the TAP controller.

Table E.2: JTAG Boundary Scan Signals

Pin Name	Pin Type	Internal Weak Pull-up	IEEE 1149.1 Function
JTDI	Input	Yes	Serial scan input
JTDO	Output	No	Serial scan output
JTMS	Input	Yes	Test Mode select
JTCK	Input	Yes	JTAG Scan Clock

Appendix F Revision 2.0 Differences

The revision 2.0 release of the RM7000 contains several enhancements over the previous silicon revision of the RM7000. Some of these enhancements increase the operational capabilities while others affect the features concerning performance measurement. The changes for the new RM7000 include:

- JTLB increases from 48 to 64 entries
- PRID Register revision number change
- CPU Config Register change
- Prefetch (PREF) instruction update
- New CP0 Info Register
- Perf Control Register changes

Code running on the original RM7000 does not need to be modified to run on the new revision of the RM7000. Some of these features are transparent to the system designer while others require software and/or CPU initialization changes in order to utilize those capabilities.

F.1 JTLB Entry Increase

The number of entries in the JTLB has increased from 48 to 64. Given that the entries are pairs of even-odd units, this change allows up to 128 virtual pages to be mapped to the 64 Gbyte physical address space. The Index field in the Index Register (bits 5:0) has been modified and now accepts a range from 0-63 to index up to 64 JTLB entries. The Wired field (bits 5:0) in the Wired Register (CP0 Set 0, Register 6) has also been modified and now accepts a range from 0-63 to lock pages into the JTLB. The read-only Random Register (CP0 Set 0, Register 1) Random field (bits 5:0) also has a range from 0-63.The JTLB could be initialized in the following manner where the number of JTLB entries (NTLBENTRIES) was determined by reading the PRID Revision Field:

```
LEAF(tlb_init)
        mtc0   zero, CP0_TLBLO0          /* Invalid even page */
        mtc0   zero, CP0_TLBLO1          /* Invalid odd page */
        mtc0   zero, CP0_PAGEMASK               /* Use default 4Kbyte pages */
        li     t3, K1BASE                /* Use unmapped address */
        li     t4, NTLBENTRIES-1         /* 48 or 64 entries */
        nop
/* Initialize the TLB */
1:      mtc0   t3, CP0_TLBHI
        mtc0   t4, CP0_INDEX
        addu   t3, 8192                  /* Increment by 8Kbytes */
        tlbwi                            /* Write the entry */
        bnez   t4, 1b
        subu   t4, 1
        j      ra
END(tlb_init)
```

For binary code compatibility, mode bit 24 is used to enable access to the new JTLB entries. If mode bit 24 is set to the default value of zero then the RM7000 appears to have 48 JTLB entries. If mode bit 24 is set to one, then all 64 JTLB entries are available and need to be initialized.

F.2 PRID Register revision number change

The PRID Register (CP0 Set 0, Register 15) holds the Implementation and revision identification numbers for the RM7000. The Implementation field (bits 15:8) is unchanged with a value of 0x27. The Revision field (bits 7:0) changes from 0x10 to 0x20. The Revision field can be used by software to identify the added functionality of RM7000A.

```
#define RM7000       0x27               # RM7000/RM7000A Implementation Number
#define RM7000_REV          0x10        # RM7000 Revision Number
#define RM7000A_REV         0x20        # RM7000A Revision Number

        mfc0   v1, CP0_PRID             # Read PRID register
        and    v1, 0xff00               # Keep Implementation Field
        li     t1, (RM7000 << 8)
        beq    v1, t1, 6f
        nop
        …
        …
6:      mfc0   v1, CP0_PRID             # Read PRID register
        and    v1, 0x00ff               # Keep Revision Field
        li     t2, 48                   # NTLBENTRIES = 48
        li     t1, RM7000A_REV
        bne    v1, t1, 7f
        nop
        li     t2, 64                   # NTLBENTRIES = 64
7:
```

F.3 CPU Config Register change

The Config Register (CP0 Set 0, Register 16) bit 16 is now used to identify the value of mode bit 20. Mode bit 20 is set to zero to specify integral clock divisors (2, 3, 4, 5, 6, 7, 8, & 9). Mode bit 20 is set to one to enable half divisors (2.5, 3.5, 4.5).

F.4 Prefetch (PREF) instruction update

The Prefetch instruction implementation for the new revision of the RM7000 is shown in the table below. All values of the Hint Field would perform the Load action on the original RM7000.

Table F.1: RM7000 Prefetch (PREF) Instructions

Hint Field	Name	Data use and desired prefetch action
31, 29..0	Load	Fetch data as if for a load
30	PrepareFor-Store	On primary dcache miss, allocate cacheline for requested address. Cacheline state is set to valid, the W bit is set and the data field is filled with all zeros. No memory transaction is done. If the victim cacheline was valid and dirty, it is written back. On primary dcache hit, no action is taken.

F.5 New CP0 Info Register

The Info Register (CP0 Set 0, Register 7) was added to provide information regarding how the new RM7000 is configured by the mode bits as well as internal configurations. The Info Register has the following definition:

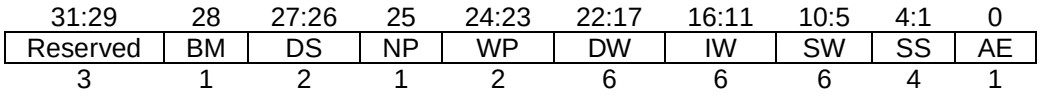


Figure F.1 CP0 Info Register (Set 0 Register 7)

Note: All bits in the CP0 Info Register are read-only except for the AE bit.

Table F.2: CP0 Info Register Fields

Bits	Field	Description
28	BM	Burst Mode: Read from modebit 15 0 : pipelined Scache RAMS (dual cycle deselect) 1 : burst mode Scache RAMS (single cycle deselect)
27..26	DS	Drive Strength: Drive strength of pad output drivers. Read from modebits 14:13 00 : 67% drive strength 01 : 50% drive strength 10 : 100% drive strength 11 : 83% drive strength
25	NP	Non-pendant bus mode: Read from modebit 26 0 : overlapping reads disabled 1 : overlapping reads enabled
24..23	WP	Write Protocol: Read from modebits 10:9 00 : R4000 compatible 01 : Reserved 10 : Pipelined writes 11 : Write re-issue
22..17	DW	Primary Data Cache Sets: 000100 : 4 sets all other patterns : reserved
16..11	IW	Primary Instruction Cache Sets: 000100 : 4 sets all other patterns : reserved
10..5	SW	Secondary Cache Sets: 000100 : 4 sets all other patterns : reserved
4..1	SS	Secondary Cache Size: 0010 : 256 Kbytes all other patterns : reserved
0	AE	Atomic Enable: Atomic Interrupt Enable/Disable. This bit allows atomic enable/disable of interrupts without a read-modify-write sequence to the SR.IE bit. Writing a one to this bit globally enables interrupts Writing a zero to this bit globally disables interrupts This bit returns a zero on a read.

F.6 Perf Control Register Changes

The following modifications/additions were made to the Perf Control Register:

Table F.3: Counter Source Field (CSF) Field Changes to Perf Control (CP0 Set 0, Register 22)

CSF BITS 4..0	RM7000 revision 1.x	RM7000 revision 2.x
0x08	Slip cycles	External cache misses
0x18	Not used	Slip cycles due to multiplier busy
0x19	Not used	Coprocessor 0 slip cycles
0x1a	Not used	Slip cycles due to pending non-blocking loads
0x1b	Not used	Write buffer full stall cycles
0x1c	Not used	Cache instruction stall cycles
0x1d	Not used	Multiplier stall cycles
0x1e	Not used	Stall cycles due to pending non-blocking loads – stall start of exception

Note 1: Counting of slip cycles has changed so that cycles are only counted when machine is in the Run state. Since stalls cause slips the new slip cycle counts will reflect only non-stalled slip causes.

Note 2: Total slip count is equal to total cycle count less dual-issued instruction pair counts

Note 3: FP and Integer Pipe resource slip cycles is equal to the difference between Total Slip cycles and the sum of the reported slip cycles (including slips due to stalls).

F.7 Determining silicon revision

The revision of the silicon contained within the package can easily be determined from the package markings. Each device contains a part number in one of the following forms:

1. RM7000-sssQ where sss refers to the speed grade of the device.
2. RM7000-sssQ-Znnn where sss refers to the speed grade of the device, Z refers to the silicon revision, and nnn is a customer specific ID.

In the first case shown above, all of the devices are 1.x silicon and do NOT contain any of the additional enhancements listed in this section. In the second case listed above, only devices with the letter “E” in the Z position of the device marking are revision 2.x silicon.

Index

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Disclaimers

When using this document, keep the following in mind:

3. This document may, wholly or partially, be subject to change without notice. Quantum Effect Devices, Inc. reserves the right to make changes to its products or specifications at any time without notice, in order to improve design or performance and to supply the best possible product.
4. All rights are reserved. No one is permitted to reproduce or duplicate, in any form, the whole or part of this document without QED's permission.
5. QED will not be held responsible for any damage to the user that may result from accidents or any other reasons during operation of the user's unit according to this document.
6. LIFE SUPPORT POLICY: QED's products are not designed, intended, or authorized for use as components intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which failure of the product could create a situation where personal injury or death may occur. Should a customer purchase or use the products for any such unintended or unauthorized application, the customer shall indemnify and hold QED and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that QED was negligent regarding the design or manufacture of the part.
7. QED does not assume any responsibility for use of any circuitry described other than the circuitry embodied in a QED product. The company makes no representations that the circuitry described herein is free from patent infringement or other rights of third parties, which may result from its use. No license is granted by implication or otherwise under any patent, patent rights, or other rights, of QED.
8. The QED logo and RISCMark are trademarks of Quantum Effect Devices, Inc.
9. MIPS is a registered trademark of MIPS Technologies, Inc. All other trademarks are the respective property of the trademark holder.

www.qedinc.com

Corporate and West Coast Sales
Quantum Effect Devices, Inc.
3255-3 Scott Blvd. Suite 200
Santa Clara, CA 95054
(408) 565-0300
(408) 565-0335 (fax)
sales@qedinc.com

East Coast Sales Office
Quantum Effect Devices, Inc.
7511 Mourning Dove Road, Suite 104
Raleigh, NC 27615
(919) 376-5415
(919) 376-5416 (fax)

European Sales Office
Quantum Effect Devices, Inc.
Concept One
55 Riverholme Drive
West Ewell, Surrey KT19 7TG
United Kingdom
[44] 181 393 5903
[44] 181 393 5903 (fax)