Quantum Effect Devices™

RM5200™ Family

User Manual
Revision 3.2

Document Changes

This page lists the major changes to the RM5200 Family User Manual.

1. Section 6 “Cache Organization and Operation” has been updated to include external cache support.

QUANTUM EFFECT DEVICES, INC., 3255-3 SCOTT BLVD., SUITE 200, SANTA CLARA, CA 95054
PHONE 408.565.0300 FAX 408.565.0335 WEB www.qedinc.com

Table of Contents

List of Figures

List of Tables

Section 1 Introduction

The RM5200 Family consists of six microprocessors implementing the same high performance 64-bit processing core, based on the R5000. The family is targeted at high-end embedded applications over a wide variety of price and performance points.

All members of the RM5200 Family are highly integrated superscalar microprocessors, implementing a superset of the MIPS IV Instruction Set Architecture (ISA). Each has a high performance 64-bit integer unit, a high throughput, fully pipelined 64-bit floating-point unit, an operating system friendly memory management unit with a 48-entry fully associative TLB, a 2-way set associative instruction cache and data cache, and an efficient system interface.

Additional features of this family include an integer multiply-add instruction for DSP type operations; cache-locking on a per-set basis; the use of an additional dedicated interrupt exception vector for faster interrupt handling; and a fully functional IEEE 1149.1 JTAG boundary scan interface.

The block diagram of the RM5200 Family is shown in Figure 1.1. Differences between family members are in cache size, SysAD width, secondary cache interface, power requirements, and available speed options. The basic differences between each processor in the RM5200 Family are outlined in Table 1.1:.

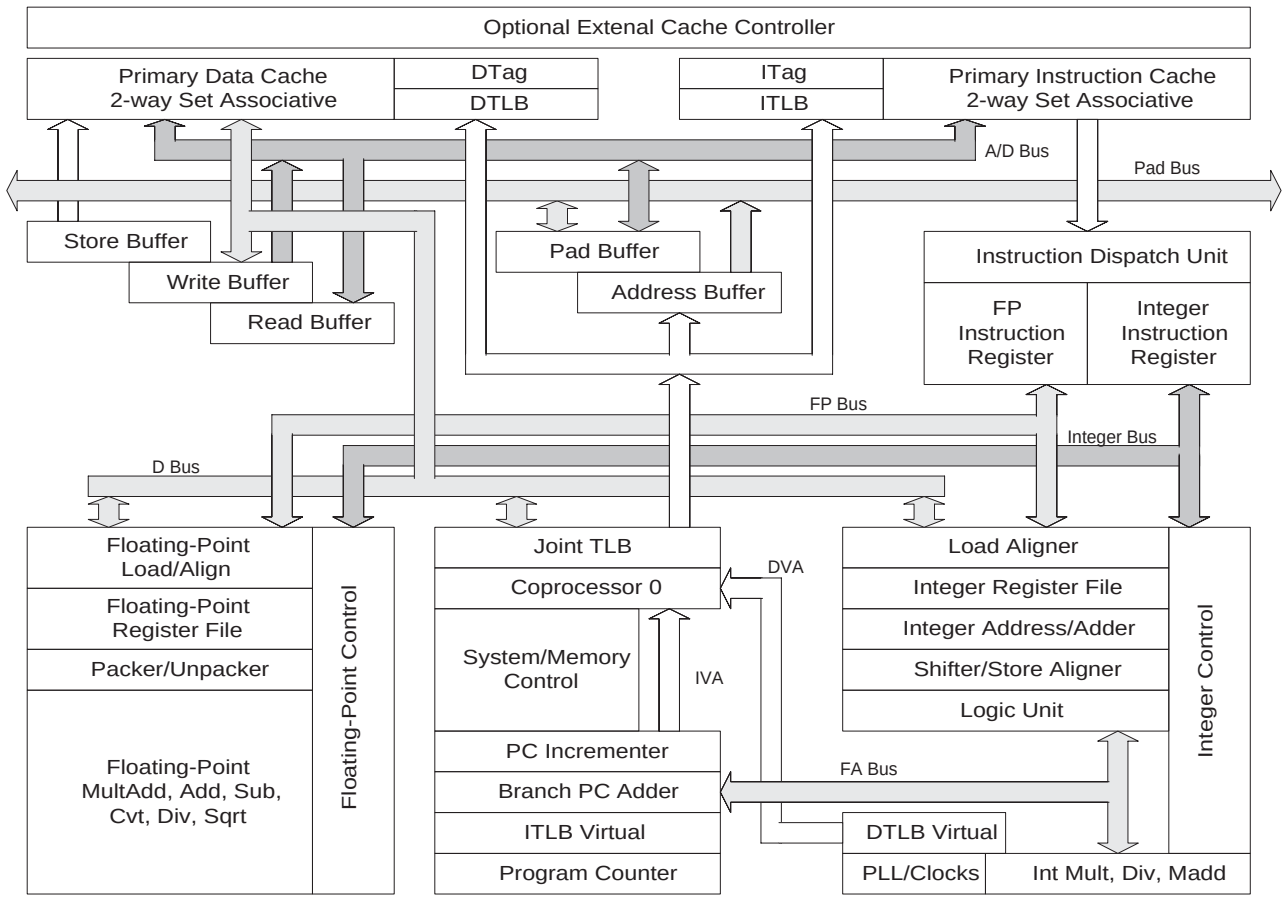


Figure 1.1 RM5200 Family Block Diagram

Table 1.1: Differences in the RM5200 Family Products

	RM5230	RM5260	RM5270	RM5231	RM5261	RM5271
I/D Cache Size	16/16KB	16/16KB	16/16KB	32/32KB	32/32KB	32/32KB
SysAD Bus Width	32-bit	64-bit	64-bit	32-bit	64-bit	64-bit
Secondary Cache	No	No	Yes	No	No	Yes
VccInt/VccIO	3.3V/3.3V	3.3V/3.3V	3.3V/3.3V	2.5V/3.3V	2.5V/3.3V	2.5V/3.3V
5V Tolerant IO	Yes	Yes	Yes	No	No	No
Max CPU Freq.	175Mhz	200Mhz	200Mhz	250Mhz	266Mhz	266Mhz
Package	128 PQuad	208 PQuad	304 SBGA	128 PQuad	208 PQuad	304 SBGA

1.1 MIPS IV Instruction Set Architecture (ISA)

The RM5200 Family executes the MIPS IV ISA, a superset of the MIPS III ISA. Full backward compatibility is provided to MIPS I, II, and III ISAs. A specific MIPS ISA can be selected to assure complete code compatibility with older implementations. This feature enables a wide variety of applications to take advantage of the MIPS Open Architecture philosophy. 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).

Additional instructions that have been added include the set of compound multiply-add instructions. These take advantage of the fact that many floating-point computations use the chained multiply-add paradigm.

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.

1.2 SuperScaler Dispatch

The RM5200 processors are capable of issuing two instructions simultaneously through an efficient asymmetric superscalar dispatch unit. Through this mechanism an integer instruction and a floating-point computation may be performed in parallel. The integer instruction may be one of any available including: alu, branch, load/store, and floating-point load/store. The floating-point computation may be a floating-point add, subtract, combined multiply-add, converts, etc.

1.3 CPU Registers

All RM5200 processors have a user visible state consisting of 32 general purpose registers, 2 special purpose registers for integer multiplication and division, a program counter, and no condition code bits.

1.4 Processor Pipeline

The RM5200 processors use a 5-stage pipeline for integer operations, loads, stores, and other non-floating-point operations. Once the pipeline has been filled, five instructions can be executed simultaneously. In addition to this standard pipeline there is an extended 7-stage pipeline for floating-point operations.

Each stage of the pipeline takes one PCycle (one internal clock cycle) which operates at a multiple of the input clock frequency (SysClock). For the RM52x0 processors this is an integer multiple (2, 3, 4, 5, 6, 7, 8). The RM52x1 processors have the addition of a 9x multiplier and half-clock cycle multiples (2.5, 3.5, and 4.5) to allow more flexibility in maximizing CPU speed to System Bus (**SysAD**) speed.

1.5 Memory Management Unit

A full-featured Memory Management Unit (MMU) is included on each processor. It uses on-chip translation lookaside buffers (TLB) to translate virtual addresses into physical addresses. Three modes of execution privilege are available to system software to provide a secure environment for user processes and provide configurability of address space depending on user applications. These execution modes are: User mode, Supervisor mode, and Kernel mode.

A Joint TLB (JTLB) is used for both instruction and data translations. It is fully associative, mapping 96 virtual pages to their corresponding physical addresses. It is organized as 48 pairs of even-odd entries, and maps a virtual address and address space identifier into the large, 64GB physical address space. 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). The user has a choice of using the hardware supported random replacement algorithm or implementing their own algorithm in software. In addition, there is a mechanism to allow a number of mappings to be locked into the JTLB. There is also information contained in the JTLB that controls the cache coherency protocol for each page.

In addition to the JTLB there are a Data TLB (DTLB) and Instruction TLB (ITLB). The 2-entry ITLB maps a 4KB page per entry. The 4-entry DTLB also maps a 4KB per entry. They improve performance by allowing instruction address translation and data address translation to occur in parallel. This minimizes contention for the JTLB, eliminates the timing critical path of translating through a large associative array, and saves power. Both are filled from the JTLB when a miss occurs. This operation is transparent to user.

1.6 System Control Processor and Registers

The integrated system control processor (CP0) in the MIPS architecture is responsible for the virtual memory subsystem, the exception control system, and the diagnostics capability of the processor. The contents of CP0 are implementation dependent and not part of any MIPS ISA. Associated with the CP0 are the on-chip system control registers. These provide the path through which the virtual memory system's page mapping is examined and modified, exceptions are handled, and operating modes are controlled. Additional registers in this group are used to implement a real-time cycle counting facility, to aid in cache diagnostic testing, and to assist in data error detection.

1.7 Integer Unit

The integer unit is fully compatible with all MIPS ISA in current use. It includes 32 general purpose 64-bit registers, a load/store architecture with single-cycle ALU operations (add, sub, logical, shift), and an autonomous multiply/divide unit. The autonomous multiply/divide unit optimizes high-speed multiply and multiply-accumulate operations, maximizing throughput while still using an area efficient implementation. Additional resources include the *HI*/*LO* result registers for the two-operand integer multiply/divide operations, and the program counter. It includes 2 implementation specific instructions that are useful in the embedded market; integer multiply-accumulate (eliminating the need for a separate DSP engine) and a 3-operand integer multiply.

1.8 Floating-Point Unit

The integrated Floating-Point Unit (FPU), with associated system software, fully conforms to the requirements of ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic. An integrated floating-point coprocessor (CP1) is a tightly coupled co-execution unit decoding and executing instructions in cooperation with the integer unit. Associated with the FPU is the floating-point register file, made up of thirty-two 64-bit registers. (These registers may be used as general purpose registers if floating-point operations are not required for a specific application.) Full 64-bit operation is achieved by taking advantage of the 64-bit wide data cache. Thus a floating-point load or store doubleword instruction may be issued every cycle.

This unit maintains fully precise floating-point exceptions while allowing both overlapped and pipelined operations. The FPU operation set includes floating-point add, subtract, multiply, divide, square root, reciprocal, reciprocal square root, conditional moves, conversion between fixed-point and floating-point format, conversion between floating-point formats, and floating-point compare.

1.9 Cache Memory

Cache memory is fully integrated on all RM5200 Family processors. Separate data and instruction caches, with single processor cycle accesses, keep the high-performance pipeline full and operating efficiently. Each cache (ICache and DCache) is two-way set associative. Cache locking of set A of each cache can be programmed by setting a bit in the Status Register, allowing critical code to be protected from being overwritten, thereby guaranteeing deterministic behavior for the locked code sequence.

Cache size differs between members of the RM5200 Family (refer to Table 1.1:). An RM52x0 processor has 16KB ICache and 16KB DCache. An RM52x1 processor has 32KB ICache and 32KB DCache.

1.10 System Interface

RM5200 processors use a high-performance multiplexed address/data system interface that is backward compatible with previous MIPS processors. The system interface consists of an Address/Data bus (**SysAD**) with check bits, a System Command bus (**SysCMD**), handshake signals, and interrupt inputs. The RM523x processors have a 32-bit **SysAD** with 4 check bits. All other processors utilize a 64-bit **SysAD** with 8 check bits. For all processors the **SysCMD** bus is 9-bits. All other system signal pins have identical functionality for all processors with the exception of the secondary cache interface. The secondary cache capability is only available with the RM527x processors (see Table 1.1:).

The **SysAD** bus is used to transfer addresses and data between the processor and the rest of the system. It is configurable to allow easy interfacing to memory and I/O systems of varying frequencies. Data rate and bus/CPU frequency are programmable via boot time serial mode bits.

The **SysCMD** bus indicates whether the **SysAD** bus carries an address or data, what kind of transaction is to take place, and information about data transactions. It is bidirectional to support both processor requests and external device requests.

The six handshake signals are used to communicate with an external device. Requests from an external device can take one of two forms; Write requests and Null requests. Write requests are used to update one of the processors writable resources such as the internal interrupt register. A null request is executed when the external device wishes the processor to reassert ownership of the processor external interface.

1.11 Secondary Cache

Only the RM527x processors incorporate a secondary cache interface (refer to Table 1.1:). For these processors the secondary cache is direct mapped and block write-through with byte parity protection for data. The secondary cache supports 512K, 1MByte, and 2MByte cache sizes (assuming nK x 18 RAM organization), with synchronous SRAM.

The secondary interface uses the SysAD bus for data and tags while providing a separate bus for addresses, and a handful of secondary cache specific control signals.

1.12 Standby Mode

Standby mode provides a means to reduce the amount of power consumed by the internal core when the CPU would otherwise not be performing any useful operations. The **WAIT** instruction is used to enter the standby mode. When in standby mode, the processors Phase Lock Loop (PLL), and external interrupt pins continue to operate in their normal fashion. The rest of the chip is powered down in standby mode.

1.13 JTAG Interface

All RM5200 processors have a fully functional IEEE 1149.1 JTAG boundary scan capability through the JTAG interface. This interface is helpful for checking the integrity of the processor's pin connections.

1.14 Power Requirements

Table 1.1 indicates different power requirements on Vcc Internal and Vcc I/O for RM52x0 processors and RM52x1 processors. This requirement is a result of differing process technologies. The RM52x0 processors use a 0.35 micron process technology while the RM52x1 processors use a 0.25 micron process technology. These two technologies dictate the Vcc requirements.

The 0.35 micron technology allows use of 3.3V for both internal and I/O Vcc. It further allows these processors to incorporate interfaces that are 5V tolerant on the I/O signals. The 0.25 micron technology dictates that the internal Vcc be 2.5V, and a separate 3.3V supply be used for the I/O, with no 5V tolerance.

Section 2 Processor Pipeline

The RM5200 processors have a 5-stage instruction pipeline. In addition to this standard pipeline, the RM5200 uses an extended 7-stage pipeline for floating-point operations. Like the R5000, the RM5200 does virtual to physical translation in parallel with 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—for example, if the required data is not in the cache, the data must be retrieved from main memory.

Once the pipeline has been filled, five instructions can be executed simultaneously. Figure 2.1 shows the five stages of the instruction pipeline.

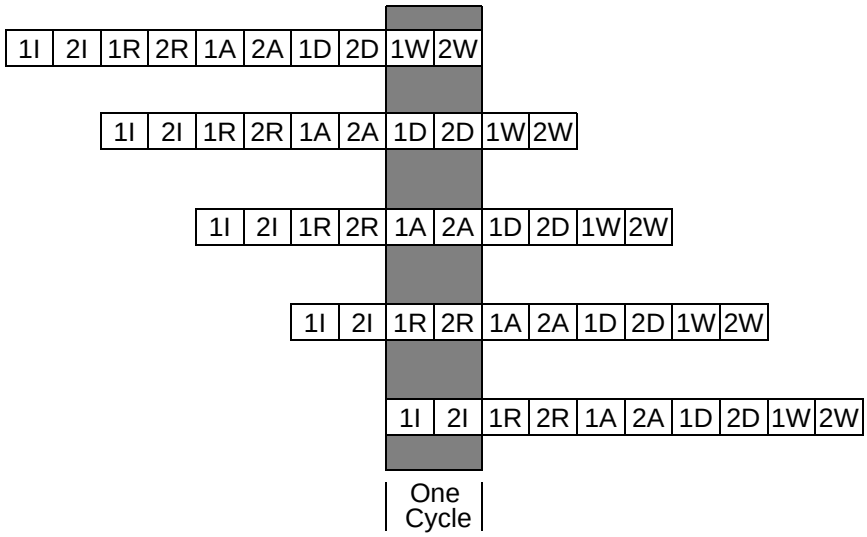


Figure 2.1 Instruction Pipeline Stages

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 decoder decodes the instruction.
- Any required operands are fetched from the register file.
- Determine whether 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 are 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.

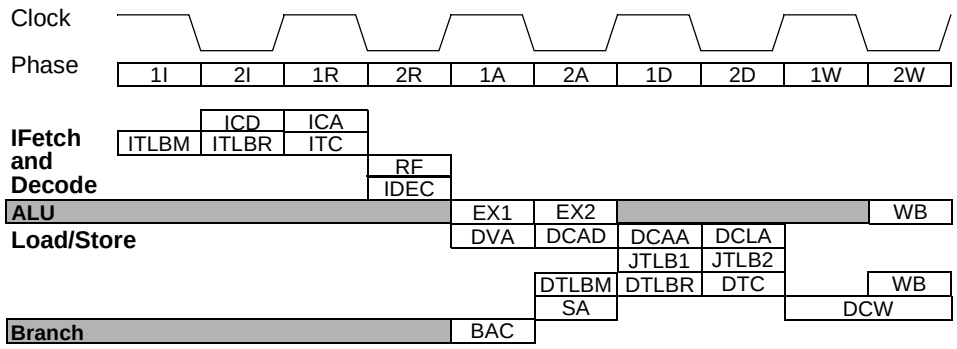
2.1.10 2W - Write Back, Phase Two

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

2.1.11 WB - Write Back

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.

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



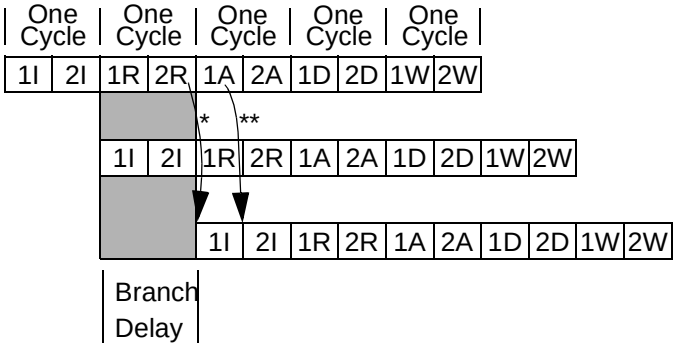
ICD	Instruction cache address decode	ICA	Instruction cache array access
ITLBM	Instruction address translation match	ITLBR	Instruction address translation read
ITC	Instruction tag check	RF	Register operand fetch
IDEC	Instruction address translation stage 2	EX1	Execute operation - phase 1
EX2	Execute operation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

Figure 2.2 CPU Pipeline Activities

2.2 Branch Delay

The CPU pipeline has a branch delay of one cycle and a load delay of one cycle. The one-cycle branch delay is a result of the branch comparison logic operating during the 1A pipeline stage of the branch. This allows the branch target address calculated in the previous stage to be used for the instruction access in the following 1I phase.

Figure 2.3 illustrates the branch delay.



* Branch and fall-through address calculated
** Address selection made

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.

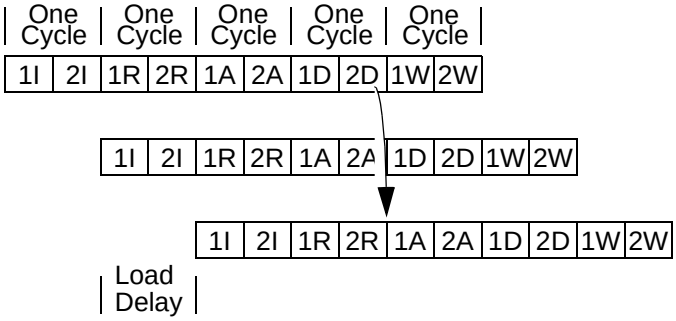


Figure 2.4 CPU Pipeline Load Delay

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 require 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
Stalls			MULS	DCM	
				CPE	
Slips	ITM	ICM			
		Interlock			
		MDBusy			
		FCBusy			
Exceptions	ITLB	IBE	RI	DBE	
		IPErr	CUn	Reset	
			BP	DPErr	
			SC	OVF	
			DTLB	FPE	
			TLBMod		
			NMI		
			Intr		

Table 2.2: Pipeline Exception

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
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
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;

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

This implementation allows all instructions which occurred before the exception to complete, and all instructions which occurred after the instruction to be aborted. Hence the value of the EPC is such that execution can be restarted. In addition, all exceptions are guaranteed to be taken in order. Figure 2.5 illustrates the exception detection mechanism for a Reserved Instruction (RI) exception.

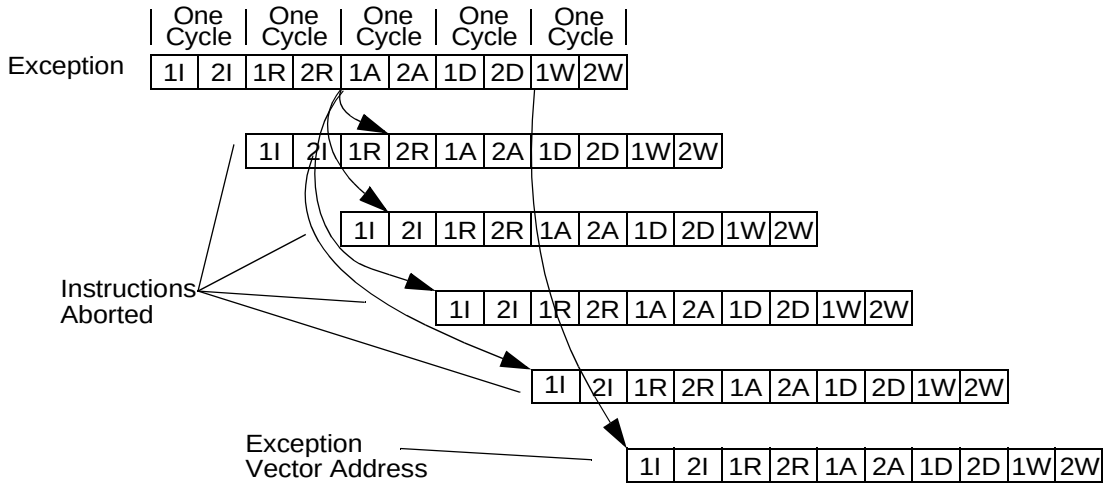


Figure 2.5 Exception Detection Mechanism

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.

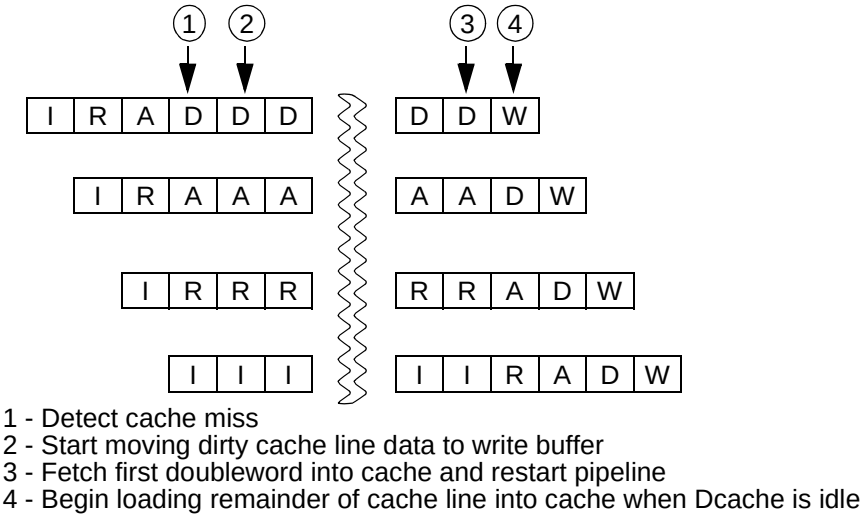


Figure 2.6 Servicing a Data Cache Miss

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 Figures 2.6 and 2.7 indicates the memory access. Once the memory is accessed and the first doubleword of data is returned, the pipeline is restarted. The remainder of the cache line is returned in subsequent cycles. The dirty data in the write buffer is written out to memory after the cache line fill operations is completed.

2.4.3 Slip Conditions

During the 2R and 1A pipeline stages, internal logic determines whether it is possible to start the current instruction in this cycle. If all required source operands are available, as well as all hardware resources needed to complete the operation, 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.

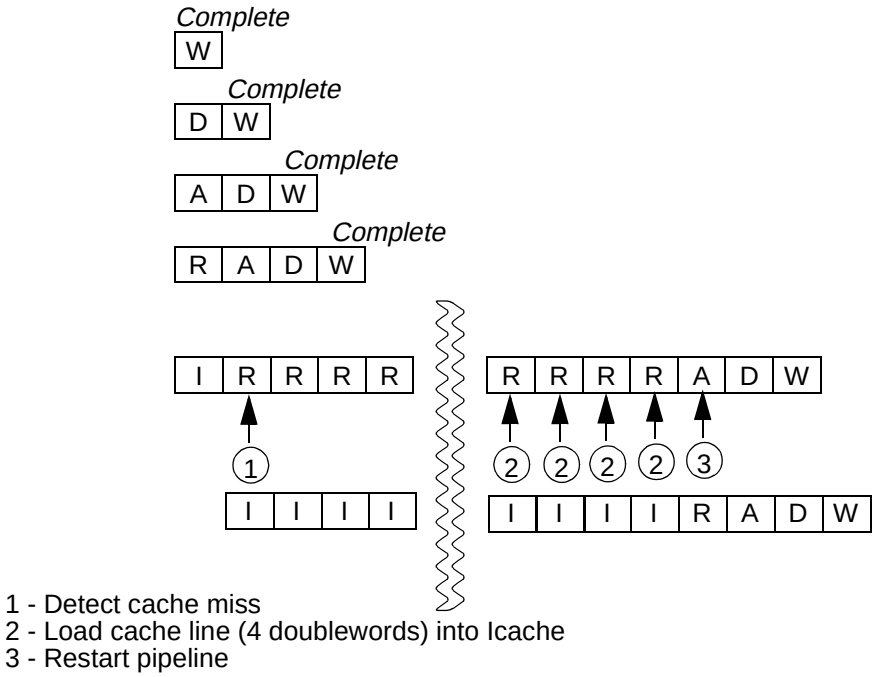


Figure 2.7 Slips During an Instruction Cache Miss

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

2.5 Write Buffer

The RM5200 processors contain 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 RM52xx processors incorporate an efficient asymmetric superscalar dispatch unit which allows them to issue an integer instruction and a floating-point computation instruction simultaneously. With respect to superscalar issue, integer instructions include alu, branch, load/store, and floating-point load/store, while floating-point computation instructions include floating-point add, subtract, combined multiply-add, converts, etc. In combination with its high throughput fully pipelined floating-point execution unit, the superscalar capability of the RM52xx provides unparalleled price/performance in computationally intensive embedded applications.

Figure 3.1 shows a simplified diagram of the dual issue mechanism.

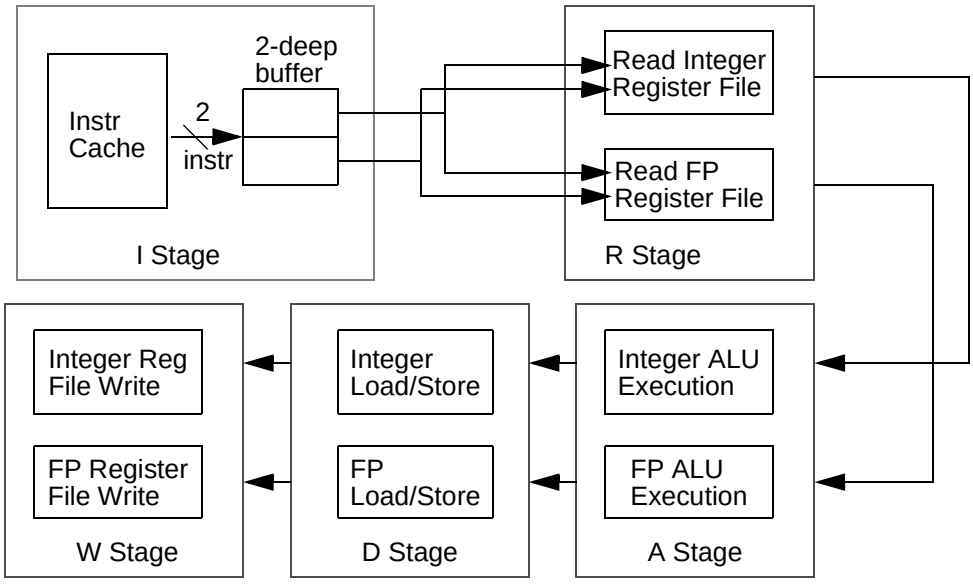


Figure 3.1 Dual Issue Mechanism

3.1 I - Stage

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

3.2 R - Stage

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

3.3 A - Stage

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

3.4 D - Stage

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

3.5 W - Stage

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

Section 4 Memory Management Unit

The RM5200 processors provide 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.

4.1 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 RM5200 processors have an Instruction TLB (ITLB) and a Data TLB (DTLB) to further improve performance.

4.1.1 Joint TLB

For fast virtual-to-physical address translation, the RM5200 processors use a large, fully associative TLB that maps 96 virtual pages to their corresponding physical addresses. As indicated by its name, the Joint TLB (JTLB) is used for both instruction and data translations. The JTLB is organized as 48 pairs of even-odd entries, and maps a virtual address and address space identifier into the large, 64GB physical address space.

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 RM5200 processors provide 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 RM5200 with data using write-back or write-through depending on the application. The write-through modes support the same efficient frame buffer handling as the R4600 and R4700.

Coherency attributes (4 - 7 in Table 4.5:) were defined in the R4000 for Multi-Processor (MP) capable microprocessors. The RM5200 processors are not MP capable, so these coherency attributes should never be used.

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 0x0000 0000 || VA[28:0].

4.1.2 Instruction TLB

The RM5200 use a 2-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.

4.1.3 Data TLB

The RM5200 use a 4-entry data TLB, or DTLB, for the same reasons cited for the ITLB. 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.

4.1.4 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 (see Figure 4.1).

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

4.1.5 Multiple Matches

The RM5200 processors do not provide any detection or shutdown mechanism for multiple matches in the TLB. Unlike earlier 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.

4.2 Processor Modes

The RM5200 processors have three operating modes, an instruction set mode, and an addressing mode. All five processor modes are described in this section.

4.2.1 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 to Kernel mode when the processor 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 4.1: 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*.

4.2.2 Instruction Set Mode

Associated with each of the processor instruction set modes specified in Table 4.1: is an implementation of two or more of the MIPS Instruction Set Architectures (ISA) for software compatibility. By default, the processor implements the MIPS I and II

ISA for all modes. Table 4.1: has two columns that indicate either the inclusion (1) or exclusion (0) for MIPS III and IV ISA for each of the processor modes based on the configuration of the *Status* register fields (noted below). Thus, for User Mode, there are three MIPS ISA compatibility modes. For (ISA III = 0 / ISA IV = 0) only MIPS ISA I and II are supported. For (ISA III = 1 / ISA IV = 0) MIPS ISA I/II/III are supported. For (ISA III = 1 / ISA IV = 1) MIPS ISA I/II/III/IV are supported.

The user can always execute implementation specific instructions.

Table 4.1: Processor Modes

XX 31	KX 7	SX 6	UX 5	KSU 4-3	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	0	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			

4.2.3 Addressing Modes

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

Refer to Table 4.1: for the following addressing mode encodings:

- In *Kernel* mode the ***KX*** bit selects 32-bit or 64-bit addressing.
- In *Supervisor* mode the ***SX*** bit selects 32-bit or 64-bit addressing and MIPS ISA compatibility.
- In *User* mode the ***UX*** bit selects 32-bit or 64-bit addressing. MIPS ISA compatibility is selected by both the ***UX*** and ***XX*** bits.

4.2.4 Endian Modes

The RM5200 processor can operate either in big-endian mode or little-endian mode. 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 halfword, 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 halfword, 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.

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 *SR.RE* bit while in kernel mode before the execution of such user mode processes.

4.3 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.

4.3.1 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 4.1 shows the translation of a virtual address into a physical address.

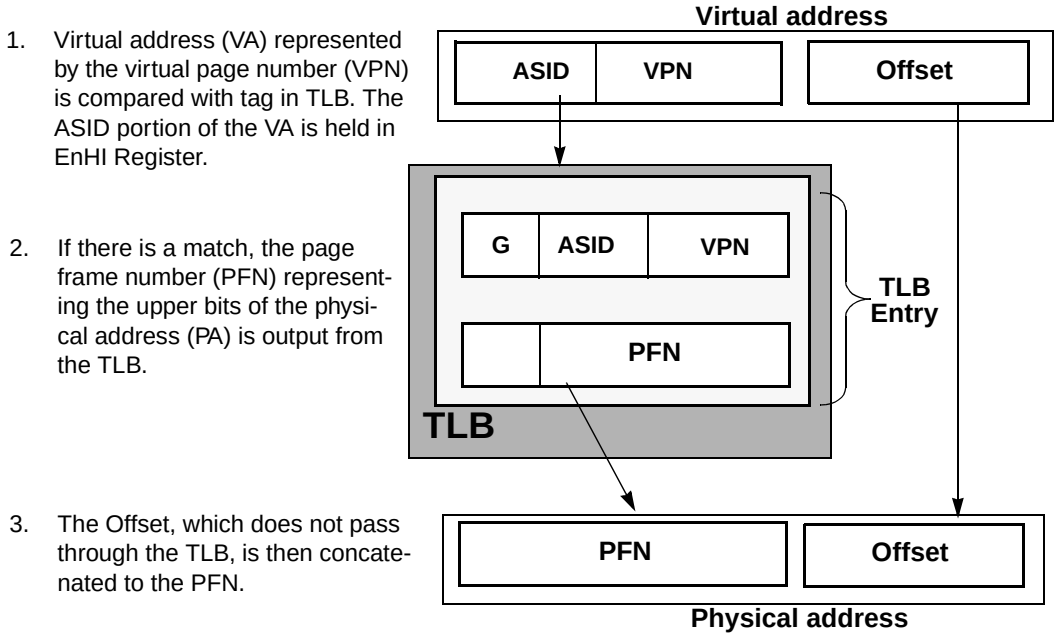


Figure 4.1 Overview of a Virtual-to-Physical Address Translation

As shown in Figure 4.1, 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.

4.3.2 Physical Address Space

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

4.3.3 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.

4.3.4 Virtual Address Translation

Figure 4.2 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 4.2 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 4.2 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.

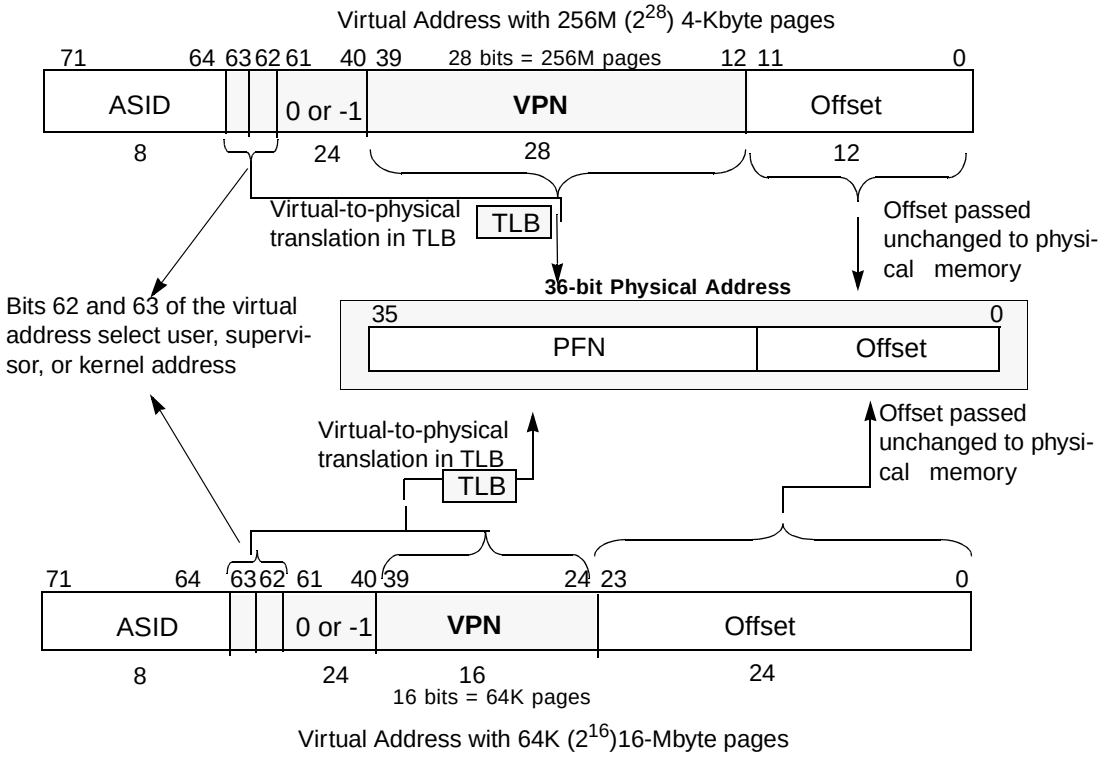


Figure 4.2 64-bit Mode Virtual Address Translation

4.3.5 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.

4.3.6 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 4.3 shows the range of User virtual address space.

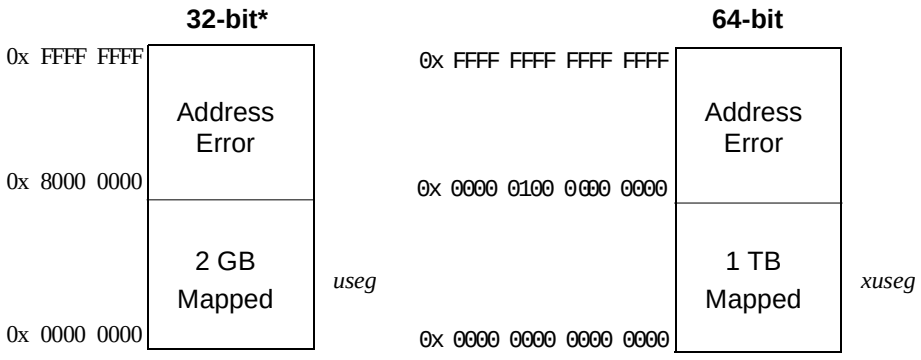


Figure 4.3 User Virtual Address Space as viewed from User Mode

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

Table 4.2: 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)

4.3.6.1 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.

4.3.6.2 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.

4.3.7 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

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 4.4 shows Supervisor address mapping. Table 4.3: lists the characteristics of the supervisor space segments; descriptions of the address spaces follow.

Figure 4.4 User and Supervisor Address Spaces as viewed from Supervisor mode

Table 4.3: 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)
11_2	X	X	sseg or csseg	0xFFFF FFFF C000 0000 through 0xFFFF FFFF DFFF FFFF	512 Mbytes (2^{29} bytes)

4.3.7.1 32-bit Supervisor, User Space (suseg)

In Supervisor space, when ***SX*** = 0 in the *Status* register and the most-significant bit of the 32-bit virtual address is set to 0, the *suseg* virtual address space is selected; it covers the full 2^{31} bytes (2 Gbytes) of the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

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

4.3.7.2 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.

4.3.7.3 64-bit Supervisor, User Space (xsuseg)

In Supervisor space, when ***SX*** = 1 in the *Status* register and bits 63:62 of the virtual address are set to 00_2, the *xsuseg* virtual address space is selected; it covers the full 2^{40} bytes (1 Tbyte) of the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space starts at virtual address 0x0000 0000 0000 0000 and runs through 0x0000 00FF FFFF FFFF.

4.3.7.4 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.

4.3.7.5 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.

4.3.8 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 ***EXL*** is cleared. The **ERET** instruction restores the processor to the address space existing prior to the exception.

Kernel virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 4.5. Table 4.4: lists the characteristics of the kernel mode segments.

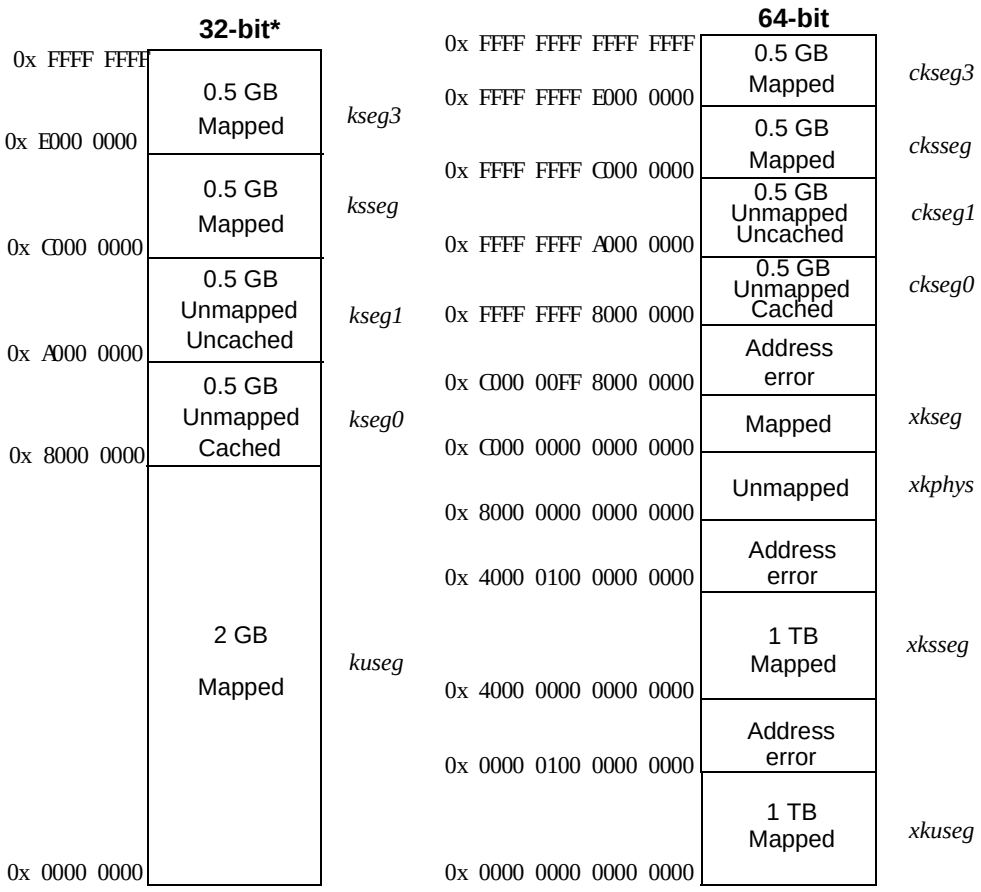


Figure 4.5 User, Supervisor, and Kernel Address Spaces viewed from Kernel mode

Table 4.4: 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)
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)
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)

4.3.8.1 32-bit Kernel, User Space (kuseg)

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

4.3.8.2 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.

4.3.8.3 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.

4.3.8.4 32-bit Kernel, Supervisor Space (ksseg)

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

4.3.8.5 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.

4.3.8.6 64-bit Kernel, User Space (xkuseg)

In Kernel space, when ***KX*** = 1 in the *Status* register and bits 63:62 of the 64-bit virtual address are 00_2, the *xkuseg* virtual address space is selected; it covers the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

When ***ERL*** = 1 in the *Status* register, the user address region becomes a 2^{31}-byte unmapped (that is, mapped directly to physical addresses) uncached address space.

4.3.8.7 64-bit Kernel, Current Supervisor Space (xksseg)

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

4.3.8.8 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 0 cause an address error.

References to this space are not mapped; the physical address selected is taken from bits 35:0 of the virtual address. Bits 61:59 of the virtual address specify the cacheability and coherency attributes, as shown in Table 4.5:.

Table 4.5: 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	**0x9000 0000 0000 0000**
3	Cacheable, noncoherent (writeback)	**0x9800 0000 0000 0000**
4-7	Reserved	**0xA000 0000 0000 0000**

4.3.8.9 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.

4.3.8.10 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 four bytes of the address, as shown in Figure 4.5, select one of the following 512-Mbyte compatibility spaces.

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

4.4 System Control Coprocessor

The System Control Coprocessor (CP0) is implemented as an integral part of the CPU, and supports memory management, address translation, exception handling, and other privileged operations. CP0 contains the registers shown in Figure 4.6 plus a 48-entry TLB. The sections that follow describe how the processor uses the memory management-related registers.

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

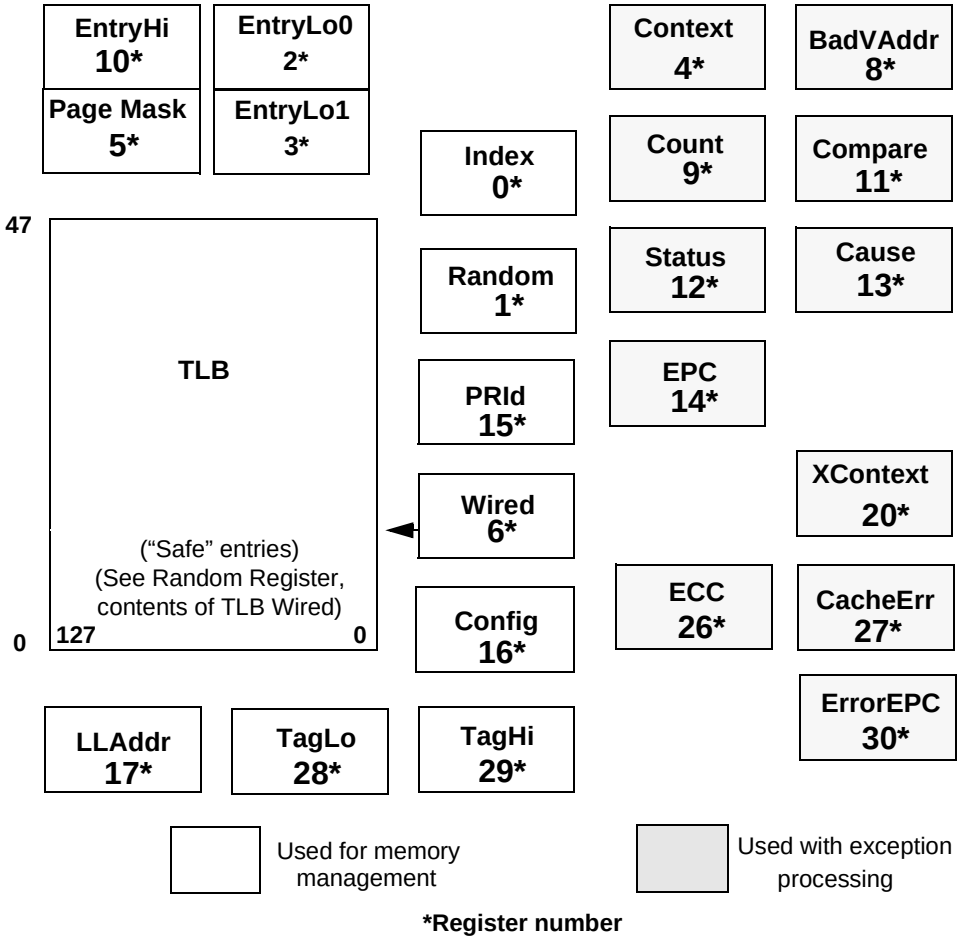


Figure 4.6 CP0 Registers and the TLB

4.4.1 Format of a TLB Entry

Figure 4.7 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.

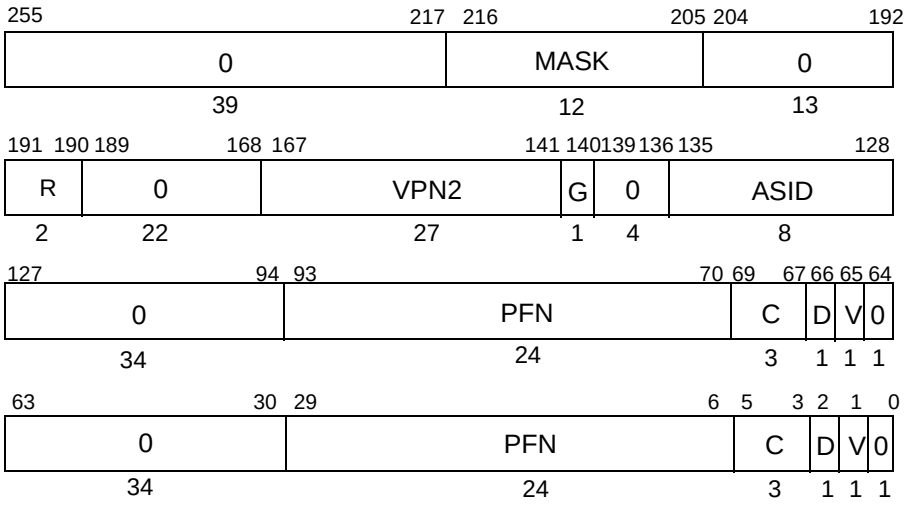
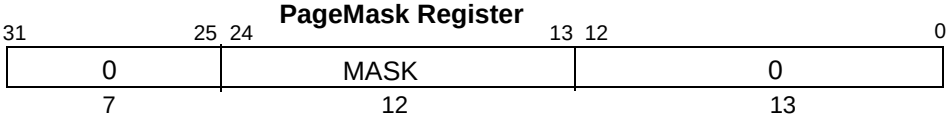


Figure 4.7 Format of a TLB Entry

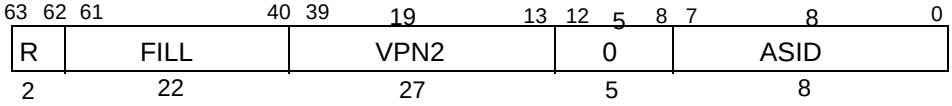
The format of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers are nearly the same as the TLB entry. The one exception is the ***Global*** field (***G*** bit), which is used in the TLB, but is reserved in the *EntryHi* register. Figures 4.8 and 4.9 describe the TLB entry fields shown in Figure 4.7.

PageMask Register



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

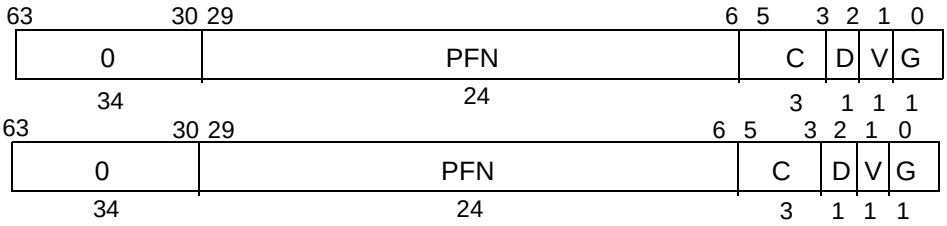
EntryHi Register



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

Figure 4.8 Fields of the PageMask and EntryHi Registers

EntryLo0 and EntryLo1 Registers



PFN Page frame number; the upper bits of the physical address.
C Specifies the TLB page coherency attribute; see Table 4.6:.
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 4.9 Fields of the EntryLo0 and EntryLo1 Registers

The TLB page coherency attribute (*C*) bits specify whether references to the page should be cached; if cached, the algorithm selects between several coherency attributes. Table 4.6: shows the coherency attributes selected by the *C* bits.

Table 4.6: 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
3	Cacheable, noncoherent, writeback
4	Reserved
5	Reserved
6	Reserved
7	Reserved

4.4.2 CP0 Registers

The following sections describe the CP0 registers that are assigned specifically as a software interface with memory management (each register is followed by its register number in parentheses).

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

4.4.2.1 Index Register (0)

The *Index* register is a 32-bit, read/write register containing six bits to index an entry in the TLB. The high-order bit of the register shows the success or failure of a TLB Probe (TLBP) instruction.

The *Index* register also specifies the TLB entry affected by TLB Read (TLBR) or TLB Write Index (TLBWI) instructions.

Figure 4.10 shows the format of the *Index* register; Table 4.7: describes the *Index* register fields.

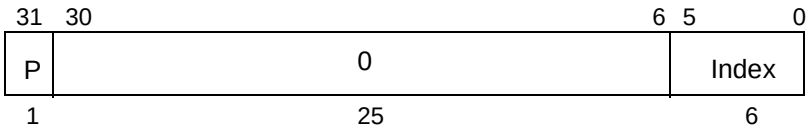


Figure 4.10 Index Register

Table 4.7: 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.

4.4.2.2 Random Register (1)

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

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

The *Random* register specifies the entry in the TLB that is affected by the TLB Write Random instruction. The register does not need to be read for this purpose; however, the register is readable to verify proper operation of the processor.

To simplify testing, the *Random* register is set to the value of the upper bound upon system reset. This register is also set to the upper bound when the *Wired* register is written.

Figure 4.11 shows the format of the *Random* register. Table 4.8: describes the *Random* register fields.

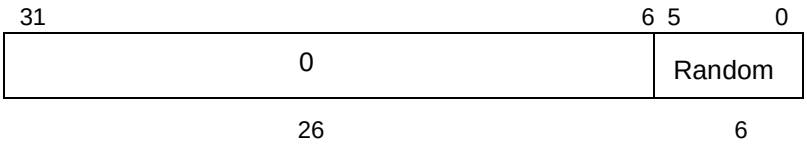


Figure 4.11 Random Register

Table 4.8: Random Register Field Descriptions

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

4.4.2.3 EntryLo0 (2), and EntryLo1 (3) Registers

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

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

The *EntryLo0* and *EntryLo1* registers are read/write registers. They hold the physical page frame number (PFN) of the TLB entry for even and odd pages, respectively, when performing TLB read and write operations. Figure 4.9 shows the format of these registers.

4.4.2.4 PageMask Register (5)

The *PageMask* register is a read/write register used for reading from or writing to the TLB; it holds a comparison mask that sets the variable page size for each TLB entry.

TLB read and write operations use this register as either a source or a destination; when virtual addresses are presented for translation into physical address, the corresponding bits in the TLB identify which virtual address bits among bits 24:13 are used in the comparison. When the ***Mask*** field is not one of the values shown in Table 4.9:, the operation of the TLB is undefined.

Table 4.9: 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

4.4.2.5 Wired Register (6)

The *Wired* register is a read/write register that specifies the boundary between the *wired* and *random* entries of the TLB as shown in Figure 4.12. Wired entries are fixed, nonreplaceable entries, which cannot be overwritten by a TLB write operation. Random entries can be overwritten.

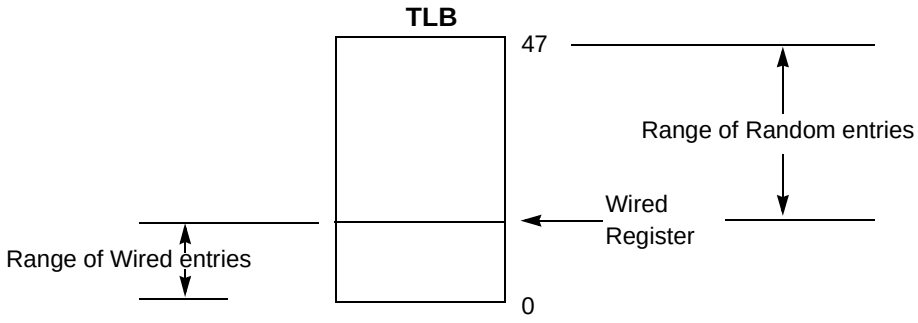


Figure 4.12 Wired Register Boundary

The *Wired* register is set to 0 upon system reset. Writing this register also sets the *Random* register to the value of its upper bound (see *Random* register, above). Figure 4.13 shows the format of the *Wired* register; Table 4.10: describes the register fields.

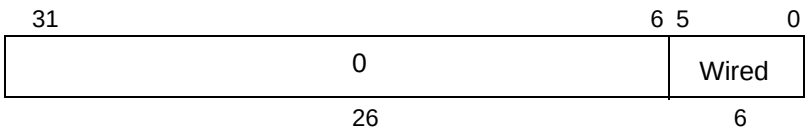


Figure 4.13 Wired Register

Table 4.10: Wired Register Field Descriptions

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

4.4.2.6 EntryHi Register (CP0 Register 10)

The *EntryHi* register holds the high-order bits of a TLB entry for TLB read and write operations.

The *EntryHi* register is accessed by the TLB Probe, TLB Write Random, TLB Write Indexed, and TLB Read Indexed instructions.

When either a TLB refill, TLB invalid, or TLB modified exception occurs, the *EntryHi* register is loaded with the virtual page number (VPN2) and the ASID of the virtual address that did not have a matching TLB entry.

4.4.2.7 Processor Revision Identifier (PRId) Register (15)

The 32-bit, read-only *Processor Revision Identifier* (*PRId*) register contains information identifying the implementation and revision level of the CPU and CP0. Figure 4.14 shows the format of the *PRId* register; Table 4.11: describes the *PRId* register fields.

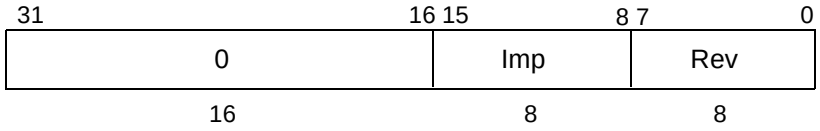


Figure 4.14 Processor Revision Identifier (PRId) Register Format

Table 4.11: PRId Register Fields

Field	Description
Imp	Implementation number (0x28)
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 while the high-order byte of the low-order halfword (bits 15:8) is interpreted as an implementation number. The content of the high-order halfword (bits 31:16) of the register are reserved.

The implementation number of the RM5200 processor is 0x28.

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.

4.4.2.8 Config Register (16)

The *Config* register reflects various configuration options.

Some configuration options are defined by the Mode Bits (see Table 9.2:, “Boot Mode Settings,” on page 85), and are therefore, dependent on the system design. Others are fixed in hardware by chip design. Two are software writable (SE,K0). These restrictions are defined in the descriptions in Table 4.12:. Additionally, caches should be reinitialized after any change is made.

Figure 4.15 shows the format of the *Config* register; Table 4.12: describes the *Config* register fields.

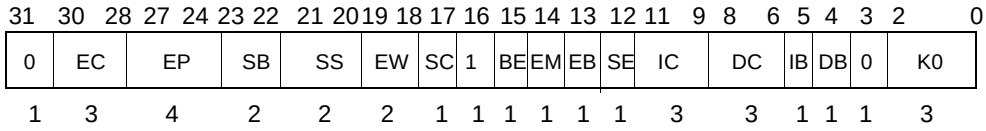


Figure 4.15 Config Register Format

Table 4.12: Config Register Fields

Bit(s)	Field	Description
30:28	EC	Pipeline clock frequency (loaded from Mode Bits 7..5): 0: SysClock frequency multiplied by 2 1: SysClock frequency multiplied by 3 2: SysClock frequency multiplied by 4 3: SysClock frequency multiplied by 5, multiplied by 2.5 (RM52x1 option) 4: SysClock frequency multiplied by 6 5: SysClock frequency multiplied by 7, multiplied by 3.5 (RM52x1 option) 6: SysClock frequency multiplied by 8 7: Reserved (RM52x0), SysClock frequency multiplied by 4.5 (RM52x1 option)
27:24	EP	Transmit data pattern (pattern for write-back data; values loaded from Mode Bits 4..1). ‘D’ = Doubleword, ‘x’ = wait state on external bus: 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. (On the RM5200 this is fixed at 8 words): 0: 4 words 1: 8 words 2: 16 words 3: 32 words
20:21	SI	System Identifiers. This field is user defined and are the value loaded from Mode Bits17:16. The system designer is free to choose encoding of these bits as the CPU is not effected by them.
19:18	EW	SysAD bus width. (Values loaded from Mode Bits for RM523x; for other RM5200 the value is fixed at 64-bit.): 00: 64-bit 01: 32-bit 10: Reserved 11: Reserved
17	SC	Secondary Cache Present. (Value loaded from Mode Bit 12 for RM527x; for other RM5200 this must be 1): 0: Secondary cache present 1: Secondary cache not present
15	BE	Big Endian Mode. (Value loaded from Mode Bit 8): 0: Little Endian 1: Big Endian

Bit(s)	Field	Description
14	EM	ECC mode enable. (On the RM5200 this is fixed at parity. ECC is not supported on-chip): 0: ECC mode 1: Parity mode
13	EB	Block ordering. (On the RM5200 this is fixed for sub-block.): 0: Sequential 1: Sub-block
12	SE	Secondary Cache Enable. (Software writable for RM527x only; for other RM5200 this must be 0) 0: Disabled 1: Enabled
11:9	IC	Primary I-cache Size (I-cache size = 2^{12+IC} bytes). In the RM52x0 processor, this field is fixed at 16 Kbytes (IC=2). In the RM52x1 processor, this field is fixed at 32 Kbytes (IC=3)
8:6	DC	Primary D-cache Size (D-cache size = 2^{12+DC} bytes). In the RM52x0 processor, this field is fixed at 16 Kbytes (IC=2). In the RM52x1 processor, this field is fixed at 32 Kbytes (IC=3).
5	IB	Primary I-cache line size. (In the RM5200 this is fixed at 32 bytes.): 0: 16 bytes 1: 32 bytes
4	DB	Primary D-cache line size. (In the RM5200 this is fixed at 32 bytes.): 0: 16 bytes 1: 32 bytes
2:0	K0	*kseg0* coherency algorithm (see *EntryLo0* and *EntryLo1* registers and the *C* field of Table 4.6:; software writable)

4.4.2.9 Load Linked Address (LLAddr) Register (17)

The read/write *Load Linked Address* (*LLAddr*) register contains the physical address read by the most recent Load Linked instruction.

This register is for diagnostic purposes only, and serves no function during normal operation.

Figure 4.16 shows the format of the *LLAddr* register; *PAddr* represents bits of the physical address, PA(35:4).

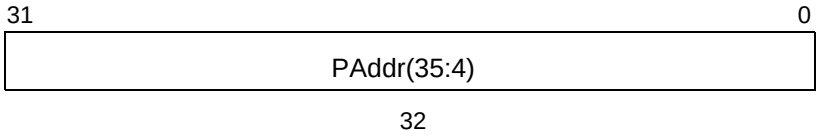


Figure 4.16 LLAddr Register Format

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

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

The ***P*** field of these registers are ignored on Index Store Tag operations. Parity is computed by the store operation.

Figure 4.17 shows the format of these registers for primary cache operations while Figure 4.18 shows the format of these registers for secondary cache operations.

Table 4.13: lists the field definitions of the *TagLo* and *TagHi* registers.

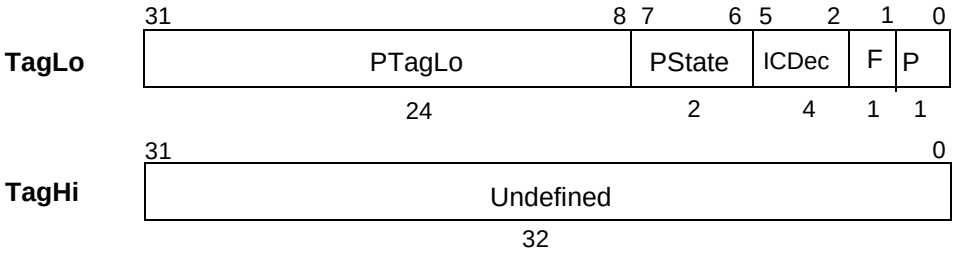


Figure 4.17 TagLo and TagHi Register (P-cache) Formats

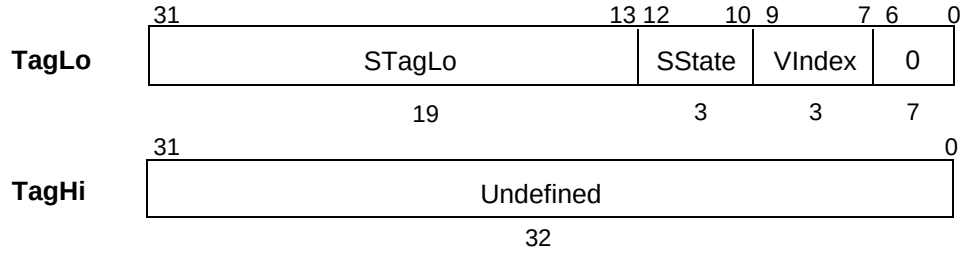


Figure 4.18 TagLo and TagHi Register (S-cache) Formats

Table 4.13: Cache Tag Register Fields

Field	Description
PTagLo	Specifies the physical address bits 35:12
PState	Specifies the primary cache state 0: Invalid 1: Reserved 2: Reserved 3: Valid
ICDec	ICache predecode bits (ICache only). Used by hardware to help decide what instructions go to the integer pipeline and which go to the FP pipeline. Software should NEVER modify these bits.
F	Fill bit used for FIFO set selection refill algorithm
P	Specifies the primary tag even parity bit
STagLo	Specifies the physical address bits 35:17
SState	Specifies the secondary cache state 0: Invalid 1: Reserved 2: Reserved 3: Reserved 4: Valid 5: Reserved 6: Reserved 7: Reserved
VIndex	Specifies the virtual index of the associated Primary cache line, vAddr(14:12)
0	Reserved. Must be written as zeroes, and returns zeroes when read.

4.4.3 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 4.19 illustrates the TLB address translation process.

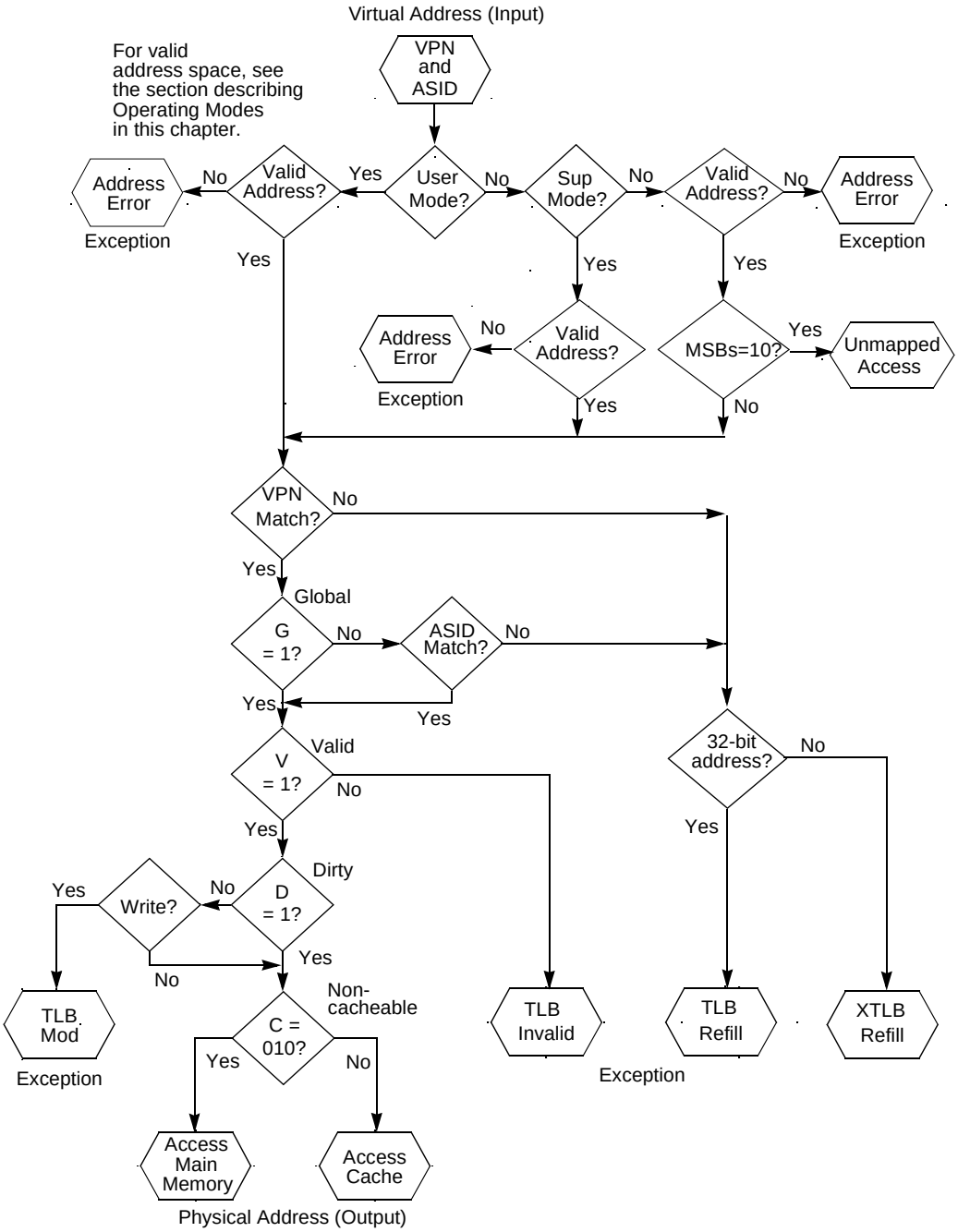


Figure 4.19 TLB Address Translation

4.4.4 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.

4.4.5 TLB Instructions

Table 4.14: lists the instructions that the CPU provides for working with the TLB.

Table 4.14: 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

Section 5 Floating-Point Unit

This section describes the floating-point unit (FPU) of theRM5200 processors, 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.

5.1 Overview

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

Figure 5.1 illustrates the functional organization of the FPU.

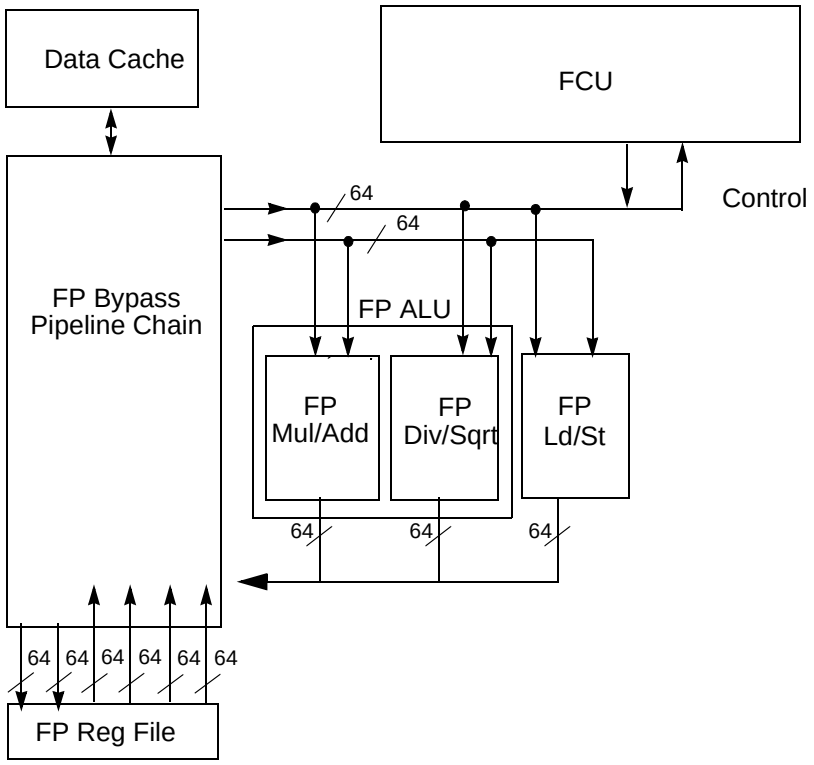


Figure 5.1 FPU Functional Block Diagram

5.2 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- and store-oriented instruction set, with single-cycle load and store operations.
- **Tightly Coupled Coprocessor Interface**. The FPU resides on-chip to form a tightly coupled unit with a seamless integration of floating-point and fixed-point instruction sets. Since each unit receives and executes instructions in parallel, some floating-point instructions can execute at the same single-cycle-per-instruction rate as fixed-point instructions.

5.3 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*.

5.4 Floating-Point General Registers (FGRs)

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 53.
- 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 5.2.

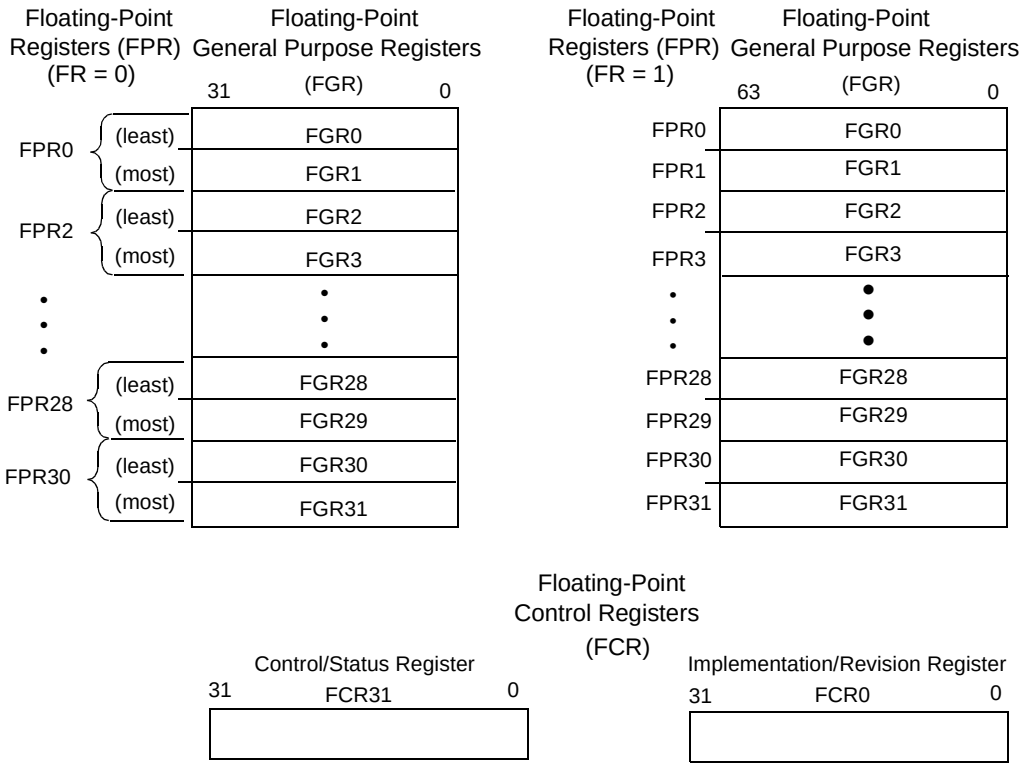


Figure 5.2 FPU Registers

5.5 Floating-Point Registers

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 5.2) 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*.

5.6 Floating-Point Control Registers

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

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

Table 5.1: lists the assignments of the *FCR*s.

Table 5.1: 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

5.6.1 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 5.3 shows the layout of the register; Table 5.2: describes the *Implementation and Revision* register (*FCR0*) fields.

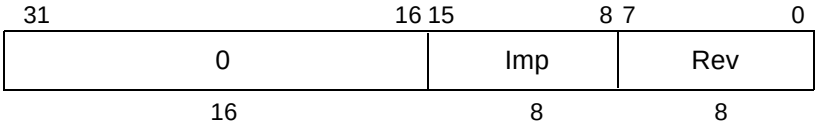


Figure 5.3 Implementation/Revision Register (FCR0)

Table 5.2: FCR0 Fields

Field	Description
Imp	Implementation number (0x28)
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.

5.6.2 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 5.4 shows the format of the *Control/Status* register, and Table 5.3: describes the *Control/Status* register fields. Figure 5.5 shows the *Control/Status* register *Cause*, *Flag*, and *Enable* fields.

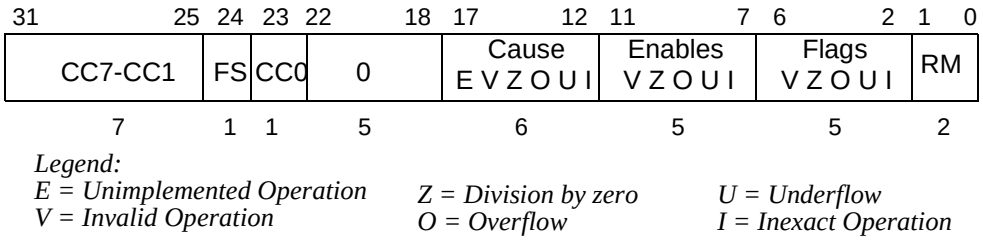


Figure 5.4 FP Control/Status Register (FCR31)

Table 5.3: Control/Status Register Fields

Field	Description
CC7-CC1	Condition bits 7-1
FS	When set, denormalized results are flushed to 0 instead of causing an unimplemented operation exception. On the RM5200, 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.

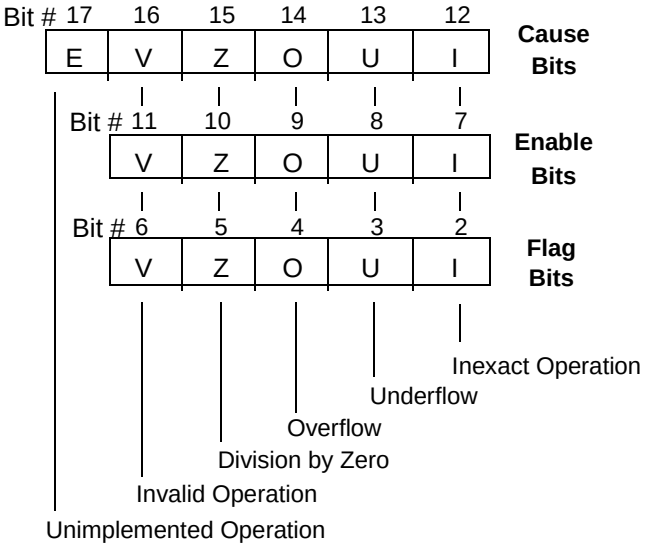


Figure 5.5 Control/Status Register Cause, Flag, and Enable Fields

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

5.6.2.2 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.

5.6.2.3 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.

5.6.2.4 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). The condition bit is set to 1 if the condition is true; the bit is cleared to 0 if the condition is false. The Condition bits are affected only by compare and Move Control To FPU instructions.

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

Figure 5.5 illustrates the *Cause*, *Flag*, and *Enable* fields of the *Control/Status* register.

Cause Bits

Bits 17:12 in the *Control/Status* register contain *Cause* bits, as shown in Figure 5.5, which reflect the results of the most recently executed instruction. The *Cause* bits are a logical extension of the CP0 *Cause* register; they identify the exceptions raised by the last floating-point operation and raise an interrupt or exception if the corresponding enable bit is set. If more than one exception occurs on a single instruction, each appropriate bit is set.

The *Cause* bits are written by each floating-point operation (but not by load, store, or move operations). The Unimplemented Operation (*E*) bit is set to a 1 if software emulation is required, otherwise it remains 0. The other bits are set to 0 or 1 to indicate the occurrence or non-occurrence (respectively) of an IEEE 754 exception.

When a floating-point exception is taken, no results are stored, and the only state affected is the *Cause* bit.

Enable Bits

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

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

Before returning from a floating-point exception, software must first clear the enabled *Cause* bits with a CTC1 instruction to prevent a repeat of the interrupt. Thus, User mode programs can never observe enabled *Cause* bits set; if this information is required in a User mode handler, it must be passed somewhere other than the *Status* register.

For a floating-point operation that sets only unenabled *Cause* bits, no exception occurs and the default result defined by IEEE 754 is stored. In this case, the exceptions that were caused by the immediately previous floating-point operation can be determined by reading the *Cause* field.

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 To Coprocessor Control 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.

5.6.2.6 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 5.4:, these bits specify the rounding mode that the FPU uses for all floating-point operations.

Table 5.4: 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.

5.7 Floating-Point Formats

The FPU performs both 32-bit (single-precision) and 64-bit (double-precision) IEEE standard floating-point operations. The 32-bit single-precision format has a 24-bit signed-magnitude fraction field (*f+s*) and an 8-bit exponent (*e*), as shown in Figure 5.6.

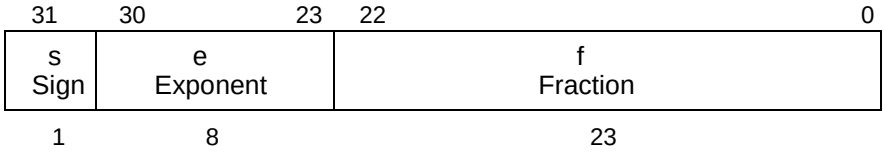


Figure 5.6 Single-Precision Floating-Point Format

The 64-bit double-precision format has a 53-bit signed-magnitude fraction field (*f+s)* and an 11-bit exponent, as shown in Figure 5.7.

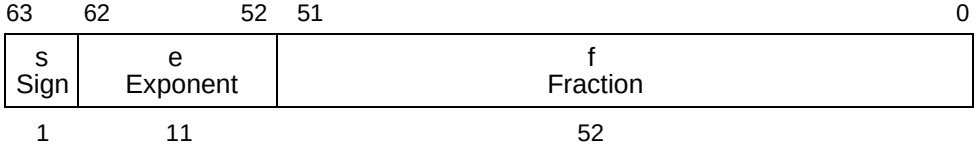


Figure 5.7 Double-Precision Floating-Point Format

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

- sign field, *s*

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

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

- E_{min} -1 (to encode ±0 and denormalized numbers)
- E_{max} +1 (to encode $\pm^{\infty}$ 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 5.5:.

Table 5.5: 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 5.6: defines the values for the format parameters; minimum and maximum floating-point values are given in Table 5.7:.

Table 5.6: 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 5.7: 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

5.8 Binary Fixed-Point Format

Binary fixed-point values are held in 2's complement format. Unsigned fixed-point values are not directly provided by the floating-point instruction set. Figure 5.8 illustrates binary fixed-point formats; Table 5.8: lists the binary fixed-point format fields.

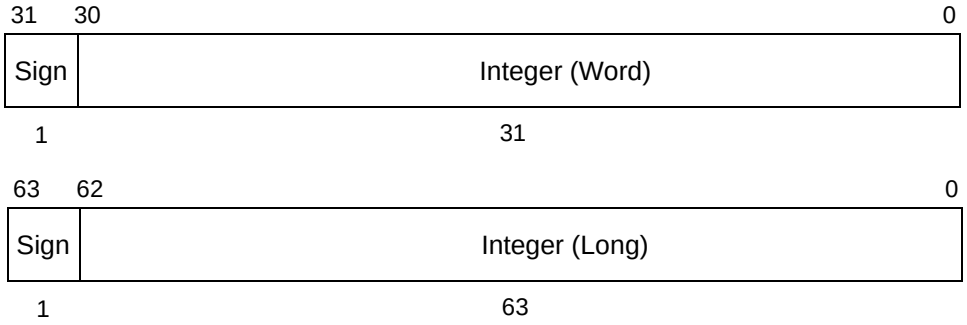


Figure 5.8 Binary Fixed-Point Formats

Field assignments of the binary fixed-point format are:

Table 5.8: Binary Fixed-Point Format Fields

Field	Description
sign	sign bit
integer	integer value

5.9 Floating-Point Instruction Set Overview

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

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

In the instruction formats shown in Table 5.9: through Table 5.12:, 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 5.9: 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 5.10: FPU Instruction Summary: Conversion Instructions

OpCode	Description
CVT.S.fmt	Floating-point Convert to Single FP
CVT.D.fmt	Floating-point Convert to Double FP
CVT.W.fmt	Floating-point Convert to 32-bit Fixed Point
CVT.L.fmt	Floating-point Convert to 64-bit Fixed Point
ROUND.W.fmt	Floating-point Round to 32-bit Fixed Point
ROUND.L.fmt	Floating-point Round to 64-bit Fixed Point
TRUNC.W.fmt	Floating-point Truncate to 32-bit Fixed Point
TRUNC.L.fmt	Floating-point Truncate to 64-bit Fixed Point
CEIL.W.fmt	Floating-point Ceiling to 32-bit Fixed Point
CEIL.L.fmt	Floating-point Ceiling to 64-bit Fixed Point
FLOOR.W.fmt	Floating-point Floor to 32-bit Fixed Point
FLOOR.L.fmt	Floating-point Floor to 64-bit Fixed Point

Table 5.11: 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 5.12: 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

5.9.1 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 5.9:.

5.9.1.1 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.

5.9.1.2 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.

5.9.1.3 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.

5.9.1.4 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.

5.9.1.5 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 halfword, 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 halfword, 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.

5.9.2 Floating-Point Conversion Instructions

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

5.9.3 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.

5.9.3.1 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.

5.9.3.2 Floating-Point Compare Operations

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

Table 5.13: lists the mnemonics for the compare instruction conditions.

Table 5.13: 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

5.10 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.

5.10.1 Instruction Execution

Figure 5.9 illustrates the 7-instruction overlap in the FPU Multiply/Add unit.

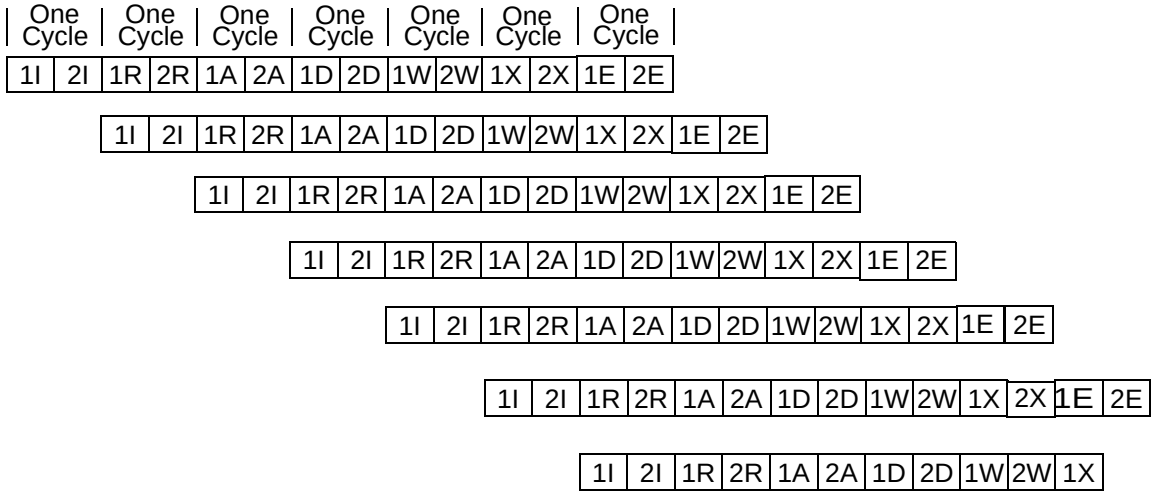


Figure 5.9 FPU Instruction Pipeline

Figure 5.9 assumes that one instruction is completed every PCycle.

5.10.2 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 A 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.

5.10.3 Instruction Scheduling Constraints

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 6 Cache Organization and Operation

In order to keep the RM5200 high-performance pipeline full and operating efficiently, the RM5200 processors incorporate on-chip primary instruction and primary data caches that can be accessed in a single processor cycle. Each cache has its own 64-bit data path allowing both caches to be accessed simultaneously. The cache subsystem provides the integer and floating-point units with an aggregate bandwidth of over 2GB per second.

This section describes the cache memory, its place in the RM5200 memory organization and individual operations of the primary cache.

6.1 Memory Organization

Figure 6.1 shows the RM5200 system memory hierarchy. In the logical memory hierarchy, caches lie between the CPU and main memory. The are designed to make the speedup of memory accesses transparent to the user. Each functional block in Figure 6.1 has the capacity to hold more data than the block above it. For instance, physical main memory has a larger capacity than the primary cache. At the same time, each functional block takes longer to access than any block above it. For instance, it takes longer to access data in main memory than in the CPU on-chip registers.

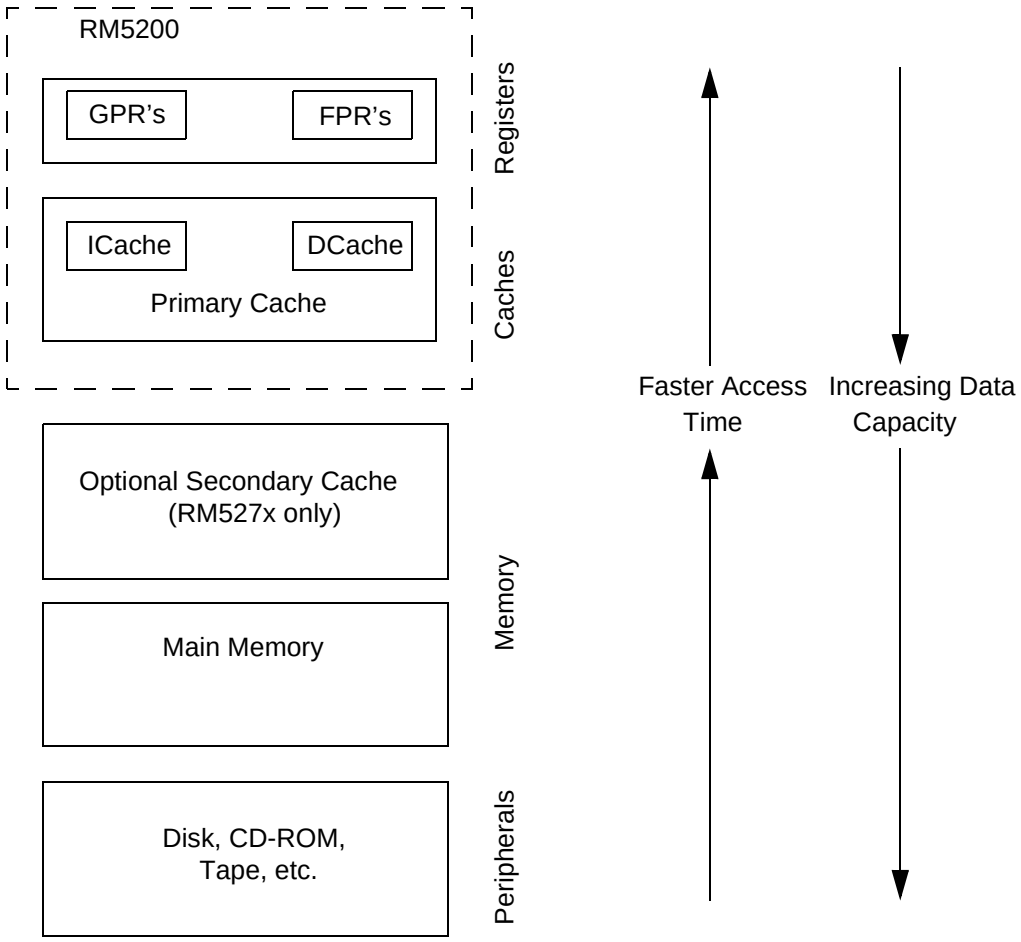


Figure 6.1 Logical Hierarchy of Memory

6.2 Overview of Cache Operations

The primary caches (ICache and DCache) provide fast temporary data storage, and they make the speedup of memory accesses transparent to the user. In general, the processor accesses cache-resident instructions or data through the following procedure: provides block diagrams of the RM5200 memory model.

1. The processor, through the on-chip cache controller, attempts to access the next instruction or data in the primary cache.
2. The cache controller checks to see if this instruction or data is present in the primary cache.
   a. If the instruction/data is present, the processor retrieves it. This is called a primary-cache *hit*.
   b. If the instruction/data is not present in the primary cache (a cache *miss*), it is retrieved as a cache line from memory and is written into the primary cache.
3. The processor retrieves the instruction/data from the primary cache and the operation continues. For a data cache miss, the processor can restart the pipeline after the first doubleword (the one at the miss address) is retrieved and continues the cache line refill in parallel.

Data is maintained in two places simultaneously: main memory and the primary cache. Data is kept consistent through the use of either a write-back or a write-through methodology. For a write-back cache, the modified data is not written back to memory until the cache line is replaced. In a write-through cache, the data is written to memory as the cached data is modified (with a possible delay due to the write buffer).

Table 6.1: RM5200 Cache Attributes

Characteristics	Instruction (ICache)	Data (DCache)	Secondary
Size	16KB (RM52x0) 32KB (RM52x1)	16KB (RM52x0) 32KB (RM52x1)	512KB to 2MB in powers of two
Set Associative	2-way	2-way	direct mapped
Replacement Algorithm	cyclic	cyclic	direct
Line size	32B	32B	32B
Index	$vAddr_{12..0}$(RM52x0) $vAddr_{13..0}$(RM52x1)	$vAddr_{12..0}$(RM52x0) $vAddr_{13..0}$(RM52x1)	$pAddr_{20..5}$
Tag	$pAddr_{35..12}$(RM52x0) $pAddr_{35..13}$(RM52x1)	$pAddr_{35..12}$(RM52x0) $pAddr_{35..13}$(RM52x1)	$pAddr_{35..19}$
Write policy	n.a.	write-back/write through	block write through
Read transfer order	sub-block	sub-block	sub-block
Write transfer order	sequential	sequential	sequential
Miss restart following:	Complete line	First double	n.a.
Parity	Word	Byte	Byte
Cache locking	Set A	Set A	none

6.3 RM5200 Cache Description

6.3.1 Organization of the Instruction Cache (ICache)

The RM5200 incorporates a two-way set associative on-chip instruction cache. This virtually indexed, physically tagged cache is 16KB in size in the RM52x0 and 32KB in size in the RM52x1 and is protected with word parity.

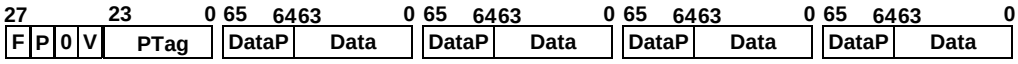
Since the cache is virtually indexed, the virtual-to-physical address translation can occur in parallel with the cache access, thus further increasing performance by allowing these two operations to occur simultaneously. The tag holds a 24-bit physical address and a valid bit and has a single bit of parity protection.

The instruction cache is 64-bits wide and can be accessed each processor cycle. Accessing 64 bits per cycle allows the instruction cache to supply two instructions per cycle to the superscalar dispatch unit. For typical code sequences where a floating-point load or store and a floating-point computation instruction are being issued together in a loop, the entire bandwidth available from the instruction cache will be consumed.

Cache miss refill writes 64 bits per cycle to minimize the cache miss penalty. The line size is eight instructions (32 bytes) to maximize the performance of communication between the processor and the memory system.

The RM5200 supports instruction cache locking. The contents of one set of the instruction cache, set A, can be *locked* by setting the IL bit in the coprocessor 0 *Status* Register. Locking the set prevents its contents from being overwritten by a subsequent cache miss. Refill will occur only into set B. This mechanism allows the programmer to lock critical code into the cache thereby guaranteeing deterministic behavior for the locked code sequence. Only non-invalid cache-lines can be locked. If a cacheline within set A is invalid while set A is locked, that cache-line can be changed by subsequent instruction fetches. Cache lines within set A can be invalidated with a Cache-Op while locked.

Refer to Table 6.1: for Cache Attributes. Figure 6.2 shows the format of a primary Instruction Cache line.



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 6.2 RM5200 Primary Instruction Cache Line Format

6.3.2 Organization of the primary Data Cache (DCache)

For fast, single cycle data access, the RM52x0 includes a 16KB on-chip data cache that is two-way set associative with a fixed 32-byte (eight words) line size. The RM52x1 has a 32KB on-chip data cache.

The data cache is protected with byte parity and its tag is protected with a single parity bit. It is virtually indexed and physically tagged to allow simultaneous address translation and data cache access.

The normal write policy is write-back, which means that a store to a cache line does not immediately cause memory to be updated. This increases system performance by reducing bus traffic and eliminating the bottleneck of waiting for each store operation to finish before issuing a subsequent memory operation. Software can, however, select write-through on a per-page basis when appropriate, such as for frame buffers. Software controls which cache policy to use through programming the TLB cache coherency bits.

Cache policies supported for the data cache:

- **Uncached.** Reads to addresses that are identified as uncached will not access the cache. Writes to such addresses will be written directly to main memory without updating the cache.
- **Write-Back.** Loads and instruction fetches will first search the cache, reading main memory only if the desired data is not cache resident. On data store operations, the cache is first searched to determine if the target address is cache resident. If it is resident, the cache contents will be updated, and the cache line marked for later write-back. If the cache lookup misses, the target line is first brought into the cache and then the write is performed as above.
- **Write-Through with write allocate.** Loads and instruction fetches will first search the cache, reading main memory only if the desired data is not cache resident. On data store operations, the cache is first searched to determine if the target address is cache resident. If it is resident, the cache contents will be updated and main memory will also be written leaving the *write-back* bit of the cache line unchanged. If the cache lookup misses, the target line is first brought into the cache and then the write is performed as above.
- **Write-through without write allocate.** Loads and instruction fetches will first search the cache, reading main memory only if the desired data is not cache resident. On data store operations, the cache is first searched to determine if the target address is cache resident. If it is resident, the cache contents will be updated and main memory will also be written leaving the *write-back* bit of the cache line unchanged. If the cache lookup misses, then only main memory is written.

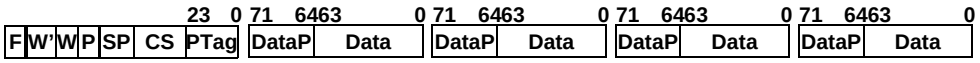
The RM5200 supports data cache locking on a per set basis. The contents of one set of the primary data cache, set A, can be *locked* by setting the DL bit in the coprocessor 0 *Status* Register. Locking the set prevents its contents from being overwritten by a subsequent cache miss. Refill will occur only into set B. This mechanism allows the programmer to lock critical data into the cache thereby guaranteeing deterministic behavior for the locked data values. Only non-invalid cache-lines can be locked. If a cacheline within set A is invalid while set A is locked, that cache-line can be changed by subsequent load instructions. Stores that hit into the locked set are allowed to update the primary data cache. Cache lines within set A can be invalidated with a Cache-Op while locked.

Associated with the Data Cache is the store buffer. When the RM5200 executes a Store instruction, this single-entry buffer gets written with the store data while the tag comparison is performed. If the tag matches, then the data is written into the Data Cache in the next cycle that the Data Cache is not accessed (the next non-load cycle). The store buffer allows the RM5200 to execute a store every processor cycle and to perform back-to-back stores without penalty. In the event of a store immediately followed by a load to the same address, a combined merge and cache write will occur such that no penalty is incurred.

In the RM5200 the W (write-back) bit, not the cache state, indicates whether or not the primary cache contains modified data that must be written back to memory.

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

Refer to Table 6.1: for primary Data Cache Attributes. Figure 6.3 shows the format of a primary Data Cache line.



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 = Shared 10 = Clean Exclusive; 11 = Dirty Exclusive
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 6.3 RM5200 Primary Data Cache Line Format

6.3.3 Accessing the Primary Caches

Figure 6.4 shows the virtual address (VA) index into the primary caches. For RM52x0 devices both instruction and data cache size is 16K bytes. For Rm52x1 devices both instruction and data cache size is 32K bytes.

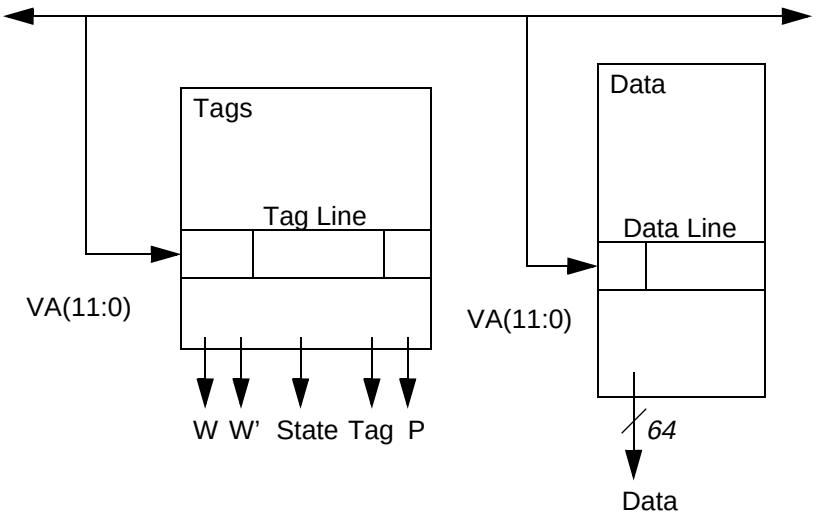


Figure 6.4 Primary Cache Data and Tag Organization

6.4 Primary Data Cache States

The following terms are used to describe the *state* of a primary data cache line:

- **Exclusive.** A cache line that is present in exactly one cache in the system is exclusive. This is always the case for the RM5200. All cache lines are in an exclusive state.
- **Dirty.** A cache line that contains data that has changed since it was loaded from memory is dirty.
- **Clean.** A cache line that contains data that has not changed since it was loaded from memory is clean.
- **Shared.** A cache line that is present in more than one cache in the system. As the RM5200 does not provide for hardware cache coherency this state should never happen in normal operations.

The RM5200 only supports the four cache states as shown in Table 6.2:. The only states that will occur in the RM5200 under normal operations are the Dirty Exclusive and Invalid states.

Note: Even though valid data is in the Dirty Exclusive state, it may still be consistent with memory. One must look at the dirty bit, W, to determine if the cache line is to be written back to memory when it is replaced.

Table 6.2: Cache States

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

6.4.1 Primary Cache States

Each primary data cache line is normally in one of the following states:

- Invalid
- Dirty Exclusive

Each primary instruction cache line is in one of the following states:

- Invalid
- Valid (Clean Exclusive)

6.5 Cache Line Ownership

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

The ownership of a cache line is set and maintained through the rules described as follows:

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

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

6.6 Data Cache State Transition Diagrams

Cache state diagrams illustrate the cache state transitions for the primary data cache. Figure 6.5 shows the state diagram of the primary data cache.

When an external agent supplies a cache line, it need not return the initial state of the cache line, for normal operation. This is because the only read request the RM5200 should issue are for non-coherent data, and the lower three bits for the data identifier are reserved. The initial state will automatically be set to Dirty Exclusive (DE) by the RM5200. Otherwise, the processor changes the state of the cache line during one of the following events:

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

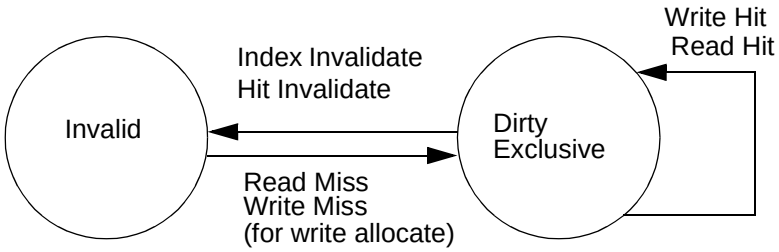


Figure 6.5 Primary Data Cache State Diagram

6.7 Secondary Cache

The RM527x provides support for an external Secondary cache. The Secondary cache interface is identical to the MIPS R5000 secondary cache interface. The RM527x supports Secondary cache sizes of 512 K byte, 1 M byte, and 2 M byte. The Secondary cache is direct-mapped and implements a block write-through protocol.

The RM527x uses the **SysAD** bus for address and data communication with the Secondary cache and provides a dedicated control interface which includes the following groups of signals:

- 16-bit Secondary cache line index (ScLine[15:0])
- 2-bit Secondary cache word index (ScLine[1:0])
- Eight enable signals for the Secondary cache tag and data RAMs
- Secondary cache valid signal (ScValid)
- Secondary cache tag match signal (ScMatch)
- Secondary cache block clear signal (ScCLR)

For a complete listing of Secondary pins and pin descriptions, refer to Section 12, “System Interface Protocol”, of this manual. For specific timing information refer to the QED RM527x Datasheet.

In addition to block read and block write operations, the Secondary cache also performs single tag invalidate, block tag invalidate, flash invalidate and tag probe operations. The single tag invalidate operation consists of writing to the tag RAM and invalidating the line in question by clearing the appropriate valid bit. This operation is done using the **Index_Load_Tag Cache** instruction with a zero in the *taglo* register. Block invalidates are the results of executing the **Page_Invalidate Cache** instruction with a zero in the *taglo* register. Single invalidates take two cycles on the system interface to execute. Block invalidates take 129 cycles on the system interface to invalidate 128 tag positions in the tag RAM. Invalidates are the only Secondary cache operations that can occur back-to-back.

The Secondary cache flash invalidate operation allows for the invalidation of an entire secondary cache 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 Secondary cache tag RAM probe is similar to a tag RAM read operation. The RM527x asserts the Secondary cache data and chip enables in the same clock that the address and line index busses are driven. The RM527x then tri-states the **SysAD** bus. One clock later the RM527x drives the tag RAM output enable and the tag RAM drives the tag information onto the **SysAD** bus.

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

6.7.1 Secondary Cache Organization

The Secondary cache is a direct-mapped cache that contains instruction and data information. The RM527x supports Secondary cache sizes of 512 Kbytes, 1Mbytes, and 2 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.

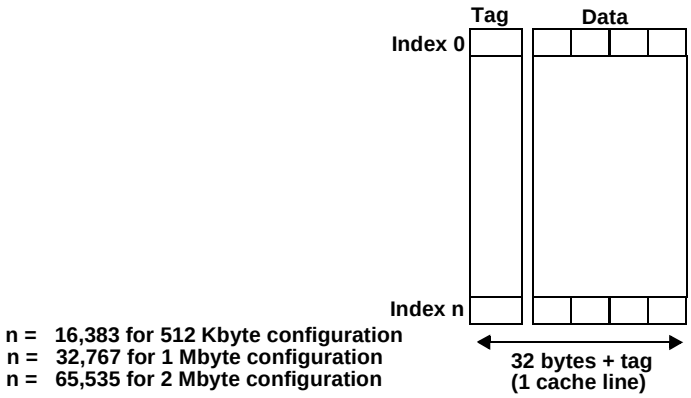
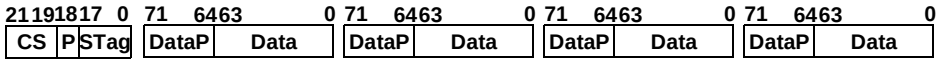


Figure 6.6 Secondary Cache Organization

Figure 6.6 shows the organization of the Secondary cache. When the Secondary 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 6.7 shows the Secondary cache line format.



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

Figure 6.7 Secondary Cache line Format

6.7.2 Accessing the Secondary Cache

The 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. The Secondary cache is physically indexed and physically tagged. The size of the index field varies depending on the size of the Secondary cache implemented. All three Secondary cache sizes are shown in Figure 6.8.

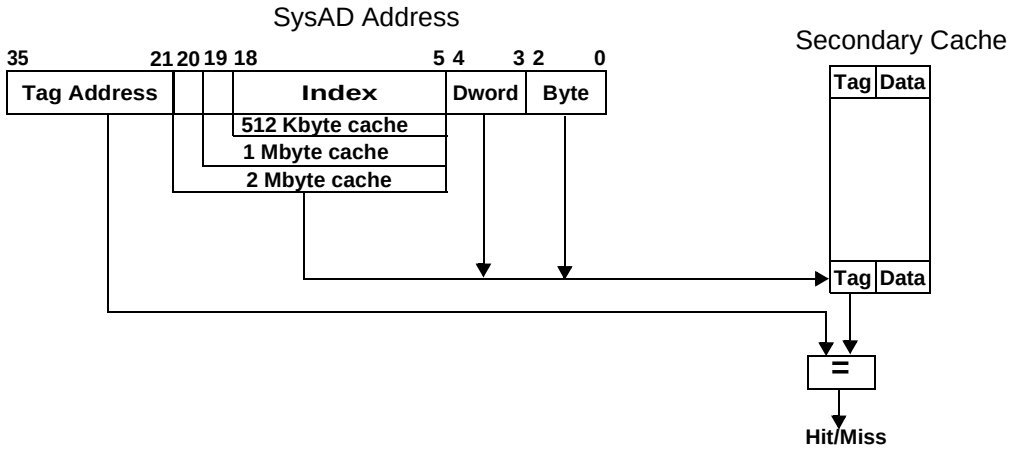


Figure 6.8 Accessing the Secondary Cache

The lower bits of address are used for indexing the data cache as shown in Figure 6.8. 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 con-

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

6.7.3 Secondary Cache States

The RM527x supports two Secondary cache states. These states are visible in bits 12:10 of the *TagLo* Register (Secondary cache format) after an **INDEX_LOAD_TAG_EC CACHE** instruction. The encoding is 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.
- **Valid (100):** A valid cache line contains valid information. A valid cache state is defined as a line that is in any state other than invalid.

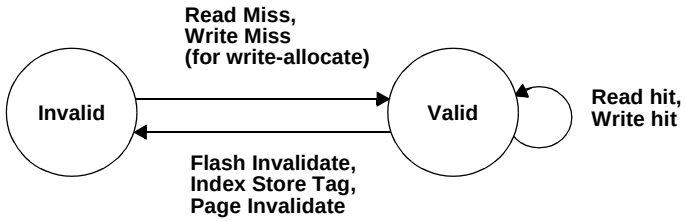


Figure 6.9 Secondary Cache State Transitions

Figure 6.9 shows a cache state transition diagram for the Secondary cache

6.8 Cache Coherency Overview

Systems using more than one master must have a mechanism to maintain data consistency throughout the system. This mechanism is called a cache coherency protocol. The RM5200 does not provide any hardware cache coherency. Cache coherency must be handled with software.

6.8.1 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 8 possible coherency attribute types. Four are reserved (see Table 4.5:, “Cacheability and Coherency Attributes,” on page 27). The available types are shown in Table 6.3::

Table 6.3: RM5200 Cache Coherency Attributes

Attribute Type	Coherency Code
Noncoherent Write-through without write-allocate	0
Noncoherent Write-through with write-allocate	1
Uncached	2
Noncoherent Writeback	3

Refer to Table 4.5:, “Cacheability and Coherency Attributes,” on page 27 for complete encodings of the different cache coherency attributes.

6.8.2 Uncached

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 the cache) for any load or store to a location within that page.

6.8.3 Noncoherent Writeback

Lines with the *Writeback* attribute can reside in any cache. On a primary data cache store hit, data are written only to the primary data cache. Data are returned to the secondary cache and main memory only when a dirty cache line is flushed from cache, either by a read that displaces this cache location, or by a block writeback cacheop.

This mode allows the primary data cache to be filled on either a load miss or a store miss. On a primary cache load miss or store miss, the RM527x checks the secondary cache for the requested address. For maximum performance, the RM527x accesses the secondary cache and main memory simultaneously with a block read request. If the requested data are present in the secondary cache, data are fetched and written to the primary cache and the pending memory access is aborted. If the requested data are not present in the secondary cache, the main memory access has already begun and data are fetched as soon as it becomes available. Once available the data are written to secondary and primary data caches.

Once in primary data cache, for a load miss the requested data is then moved to the register file. For a store miss, the doubleword, partial doubleword, word, or partial word data are written to the primary data cache. Data are returned to main memory only when the line is flushed from primary cache, either by a read that displaces this cache location, or by a block writeback cacheop.

6.8.4 Noncoherent Write-through with Write-allocate

Lines with a noncoherent *write-through with write-allocate* attribute type can reside only in the primary caches. On a load hit, data are fetched from the primary data cache to the register file.

On a load miss, data are fetched from main memory with a block read request and are written only to the primary data cache. The secondary cache is not accessed.

On a primary data cache store hit, doubleword or parital doubleword data are written to both the primary data cache and main memory. The secondary cache is not accessed.

On a primary data cache store miss, the corresponding cache line is fetched from main memory, modified with the store data, and then the doubleword or partial double-word data are written to main memory using non-block protocol.

6.8.5 Noncoherent Write-through without Write-allocate

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 cache 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 cache.

On a primary cache load miss, data are fetched from main memory with a block read request and are written only to the primary data cache. The secondary cache is not accessed.

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

Table 6.4: Cache Events and Coherency Behavior

Event	Cache Coherency Attribute			
	0	1	2	3
	W/T without W/A	W/T with W/A	Uncached, Blocking	WB
Dcache lookup	Yes	Yes	No	Yes
Fill Dcache on load miss	Yes	Yes	No	Yes
Fill Dcache on store miss	No	Yes	No	Yes
Store data modifies?	(1)	(1)	MM only	(2)
Load miss, lookup Scache. If Scache miss, fill Scache	No	No	No	Yes
Store miss, lookup Scache. If Scache miss, fill Scache	No	No	No	Yes
Dcache WB, line written back to the Scache	No WB's	No WB's	No WB's	Yes (4)
Dcache WB, written back to main memory	No WB's	No WB's	No WB's	Yes (4)
Weakly Ordered Memory Model (load and stores can occur out of program order)	Yes	Yes	No	Yes

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

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

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

Note 4: Data is written back to the secondary cache only if the secondary cache contains the cache line.

6.9 Secondary Cache Configuration

A Secondary Cache Configuration is only applicable to the RM527x. References are made below to section 4.4.2.8 (Config Register) and Table 9.2: (Boot Mode Settings).

The existence of a secondary cache is loaded in at boot time via the serial mode bit [12] and is reflected in CP0 config register *SC* bit [17]. When *SC* bit = 0, secondary cache is present.

The secondary cache is enabled or disable by seting or clearing CP0 config register *SE* bit [12]. When *SE* bit = 1, the secondary cache is enabled. The *SE* bit is cleared at reset. When the secondary cache is enabled by setting the *SE* bit, the state of the cache is undefined, thus software must explicitly invalidate the entire secondary cache before using it.

If no secondary cache is present, or the secondary cache is disabled, the processor drives all secondary cache signals to their inactive state. If no secondary cache is present the **ScMatch** and **ScDOE*** signals become don't-care inputs and must be terminated to valid logic levels. If the secondary cache is present and enabled, then the **SysADC** signals must implement valid parity during block read responses.

The doublewords transferred on **SysAD** during secondary cache block read transactions are in sub-block order. The doublewords transferred on **SysAD** during secondary cache block write transactions are in sequential order.

The secondary cache synchronous burst SRAM type is specified by the boot time serial mode bit [15]. Bit [15] = 1 indicates single cycle deselect (SCD) SRAM timing; bit [15] = 0 indicates dual cycle deselect (DCD) SRAM timing. The difference

between these two settings is in the timing of the signal **ScDCE*** on a secondary cache read hit. As Figure 11.21 shows, with **ScDCE*** asserted at t1 time and bit [15] = 0, **ScDCE*** is negated at t5 time. Inversely, if bit [15] = 1, **ScDCE*** is negated at t6 time. Boot time mode bit [15] = 1, allows for the use of commodity SRAM devices in the system design.

The size of the secondary cache is set by the system design. The size can be from 512KB, 1MB or 2MB. The external cache size may be indicated to the system software by the boot time serial mode bit [17:16]. The state of these bits have no effect on the hardware other then being reflected in CP0 config register bits [21:20]. Therefore the system designer is free to choose the encoding of these bits.

Figure 6.10 shows a block diagram of how a secondary cache might be configured for the RM527x processor.

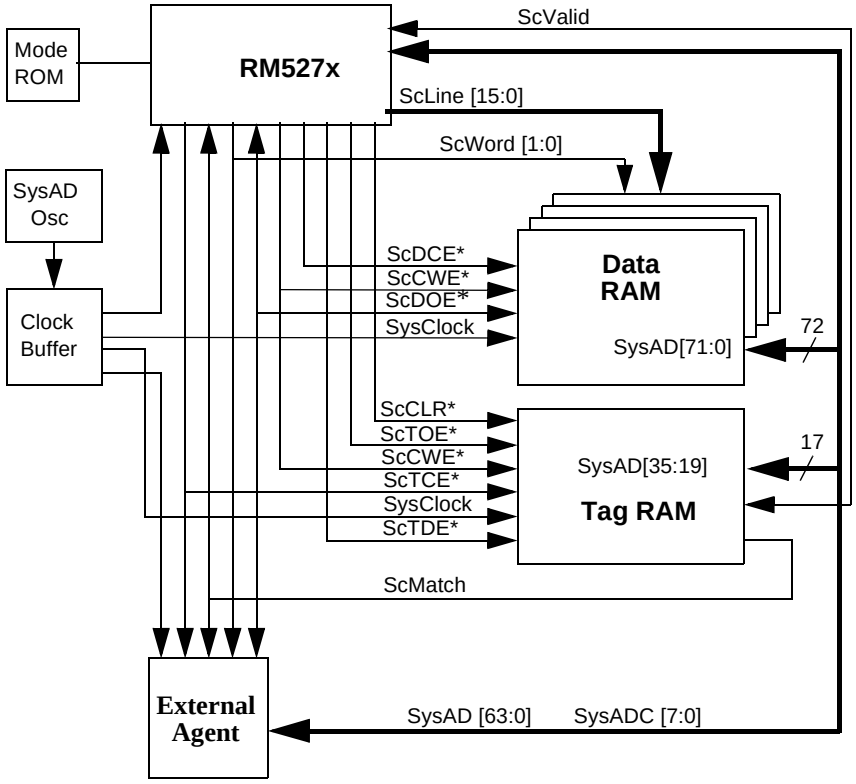


Figure 6.10 Secondary Cache Block Diagram

6.9.1 SRAM

The data RAMs are pipelined synchronous burst SRAMs with registered inputs and outputs. The chip enable and write enable signals are pipelined. The output enable signal is asynchronous. Figure 6.11 shows a block diagram of such a data RAMs.

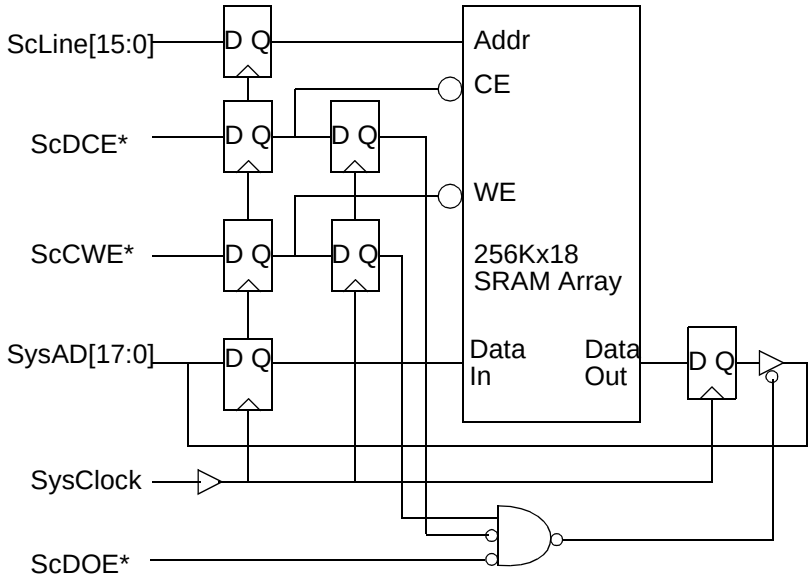


Figure 6.11 Data RAM Block Diagram

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.

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 6.12 shows a block diagram of the tag RAM.

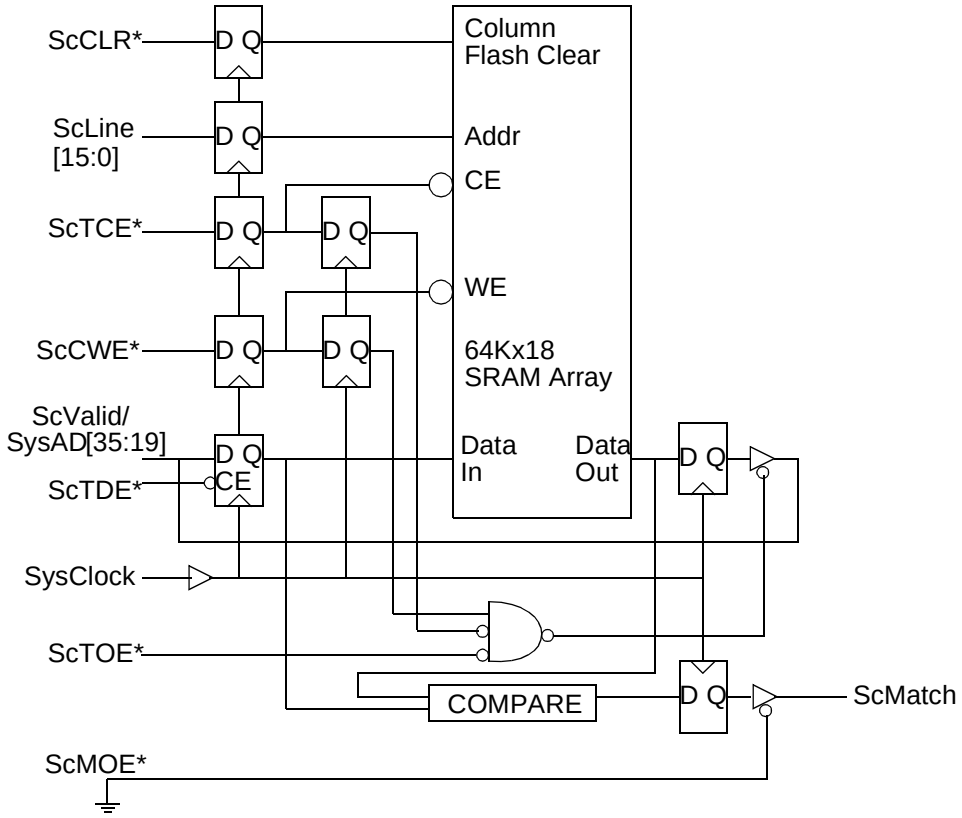


Figure 6.12 Tag RAM Block Diagram

The Secondary cache Match Output Enable (ScMOE*) signal would normally be generated by an ASIC. Here it is simply grounded.

6.10 RM5200 Processor Synchronization Support

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 RM5200 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.

6.10.1 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 6.13 illustrates a test-and-set synchronization procedure that uses a semaphore; when the semaphore is set to 0, the shared data is unlocked, and when the semaphore is set to 1, the shared data is locked.

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.

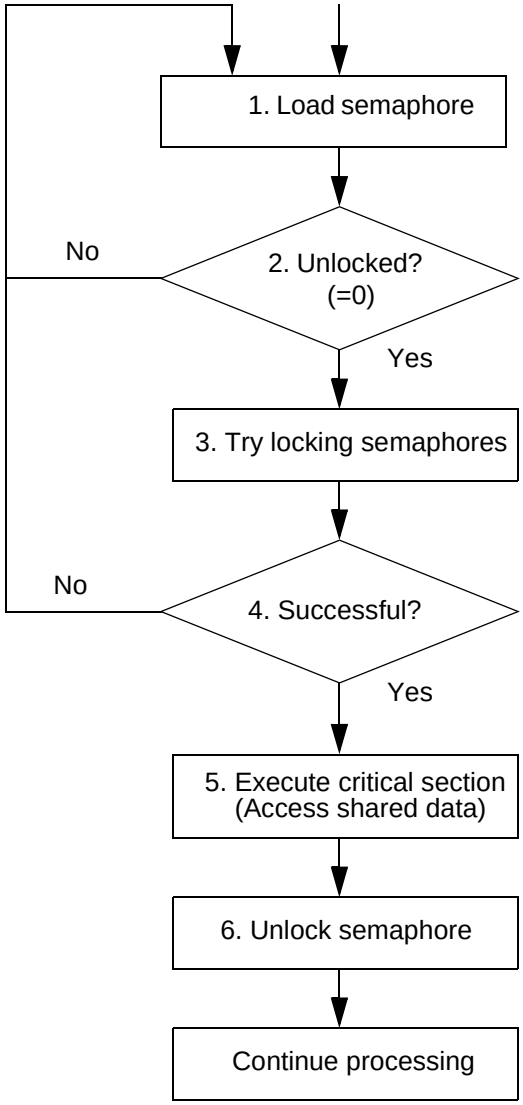


Figure 6.13 Synchronization with Test-and-Set

6.10.2 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.).

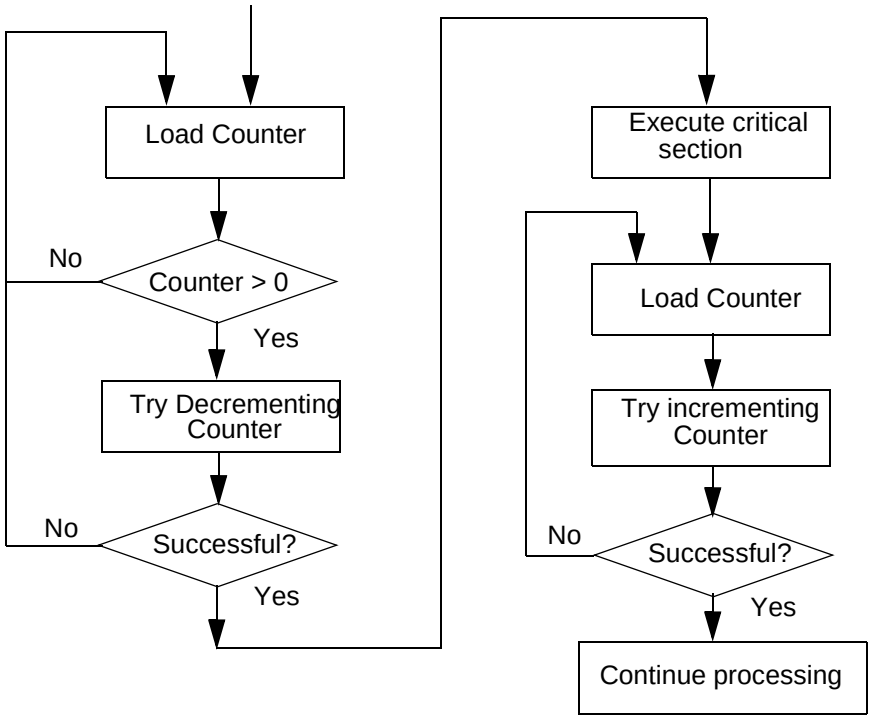


Figure 6.14 Synchronization Using a Counter

6.10.3 Load Linked and Store Conditional

The RM5200 instructions *Load Linked* (**LL**) and *Store Conditional* (**SC**) provide support for processor synchronization. These two instructions work very much like their simpler counterparts, load and store. The **LL** instruction, in addition to doing a simple load, has the side effect of setting a bit called the *link bit*. This link bit forms a breakable link between the **LL** instruction and the subsequent **SC** instruction. The **SC** performs a simple store if the link bit is set when the store executes. If the link bit is not set, then the store fails to execute. The success or failure of the **SC** is indicated in the target register of the store. The link is broken upon completion of an **ERET** (return from exception) instruction.

The most important features of **LL** and **SC** are:

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

Figure 6.15 shows how to implement test-and-set using **LL** and **SC** instructions; Figure 6.16 shows synchronization using a counter implemented with **LL** and **SC** instructions.

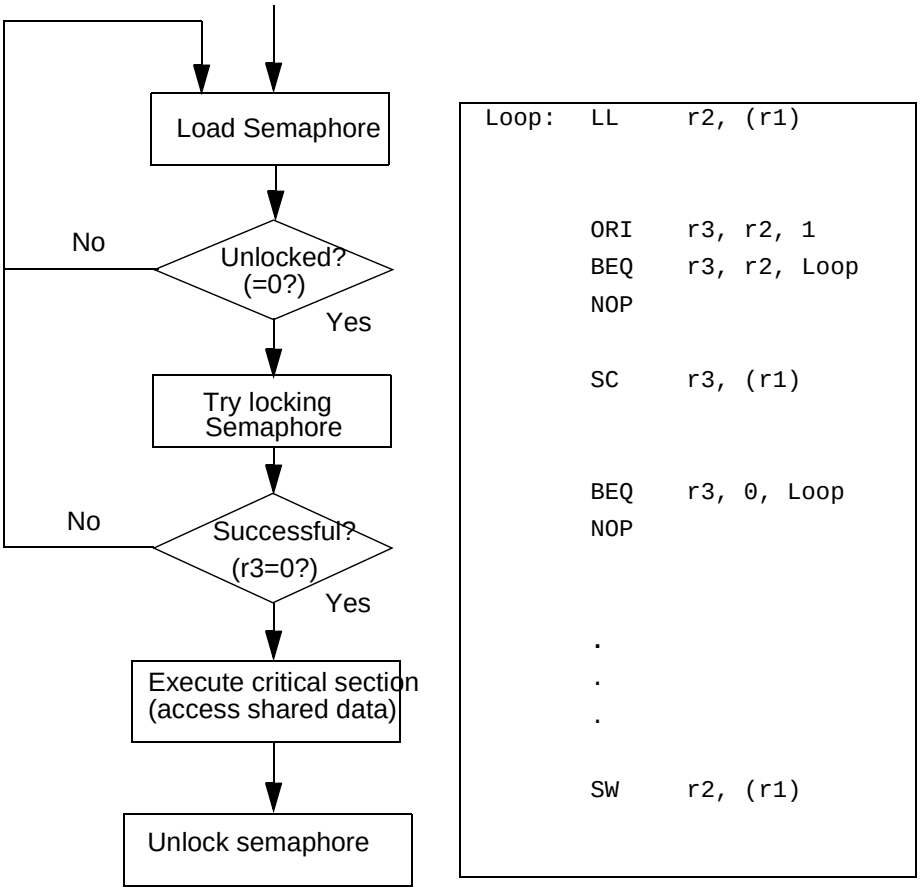


Figure 6.15 Test-and-Set using LL and SC

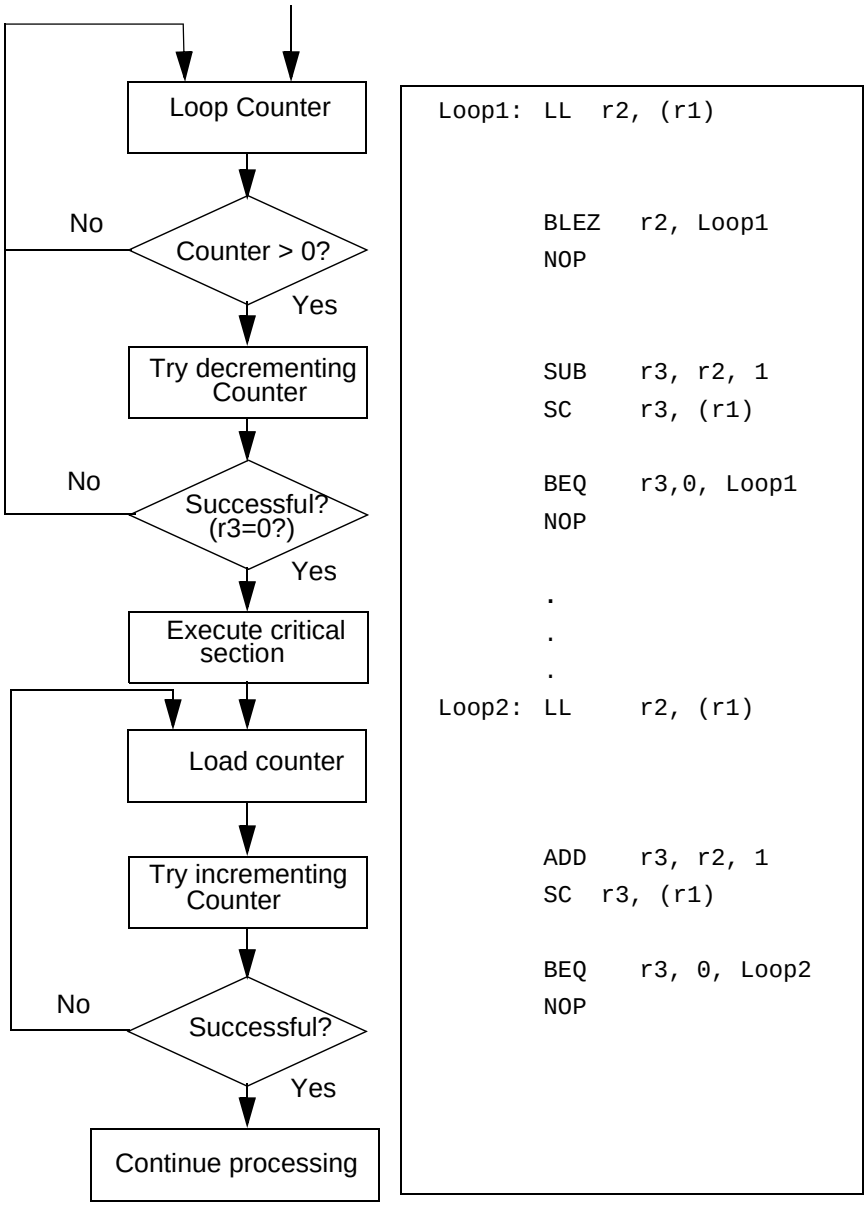


Figure 6.16 Counter Using LL and SC

Section 7 RM5200 Processor Bus Interface

The RM5200 processors provide an efficient system interface for the transfer of instruction and data between the processor and the system. This interface is compatible with the R4600, R4700, and the R5000 system interfaces.

The RM5200 uses the SysClock input to synchronize all bus transactions with itself. It multiplies SysClock by an integer between 2 and 8, inclusive, to produce the internal pipeline clock.

The processor bus interface consists of an Address/Data bus (32-bits for the RM523x; 64-bits for the RM526x/7x) with check bits (4-bits for the RM523x; 8-bits for the RM526x/7x) and a 9-bit command bus. In addition, there are 6 handshake signals and 6 interrupt signals. The interface has a simple timing specification and is capable of transferring data between the processor and memory.

In addition, the RM527x has an integrated cache controller, which can be used to implement an external, unified, write-through secondary cache. The RM527x support secondary cache sizes of 512KB, 1MB, and 2MB.

7.1 Interface Buses

Figure 7.1 shows a typical embedded system using the RM5200. In this example, a bank of DRAMs and a memory controller ASIC share the processor's **SysAD** bus while the memory controller provides separate ports to a boot ROM and an I/O system.

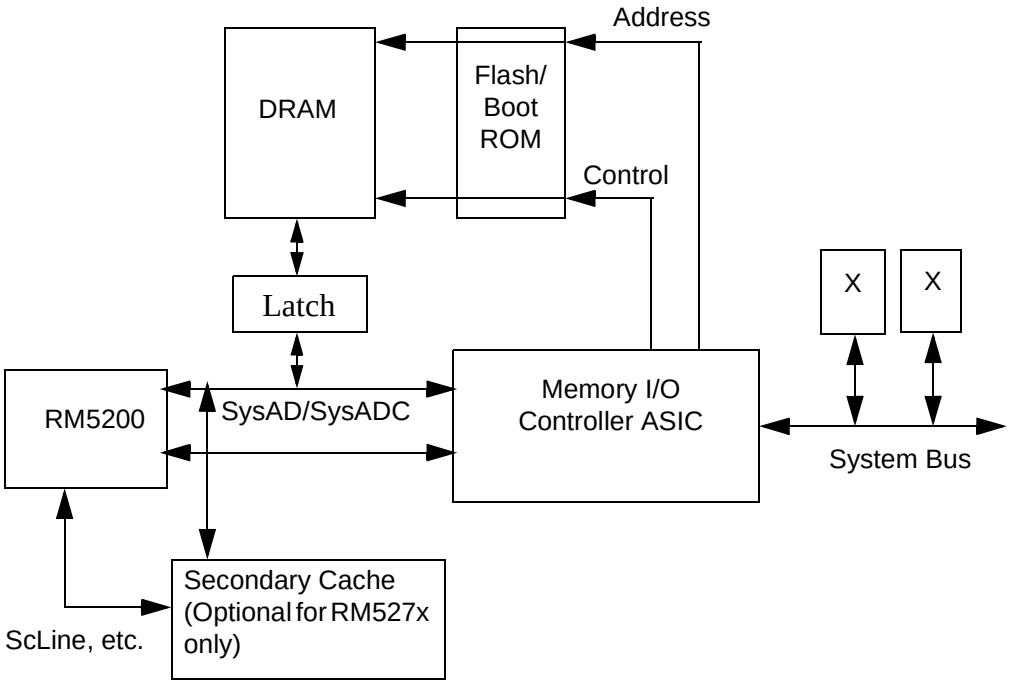


Figure 7.1 Typical Embedded System Block Diagram

7.2 RM5200 Processor Signal Descriptions

The signals used by the RM5200 include the System interface, the Clock interface, the Interrupt interface, the Joint Test Action Group (JTAG) interface, and the Initialization interface. The RM527x has the additional signals for the Secondary Cache Interface.

Signals are listed in bold, and low active signals have a trailing asterisk—for instance, the low-active Read Ready signal is **RdRdy***. The arrows used for each signal tells if the signal is an input (the processor receives it), an output (the processor sends it out), or bidirectional.

Figure 7.2 illustrates the functional groupings of the processor signals for the RM526x/7x. The Secondary Cache Interface only applies to the RM527x. The RM523x is identical to the RM526x except for **SysAD** is only 32 lines (**SysAD[31:0]**) and **SysADC** is only 4 lines (**SysADC[3:0]**).

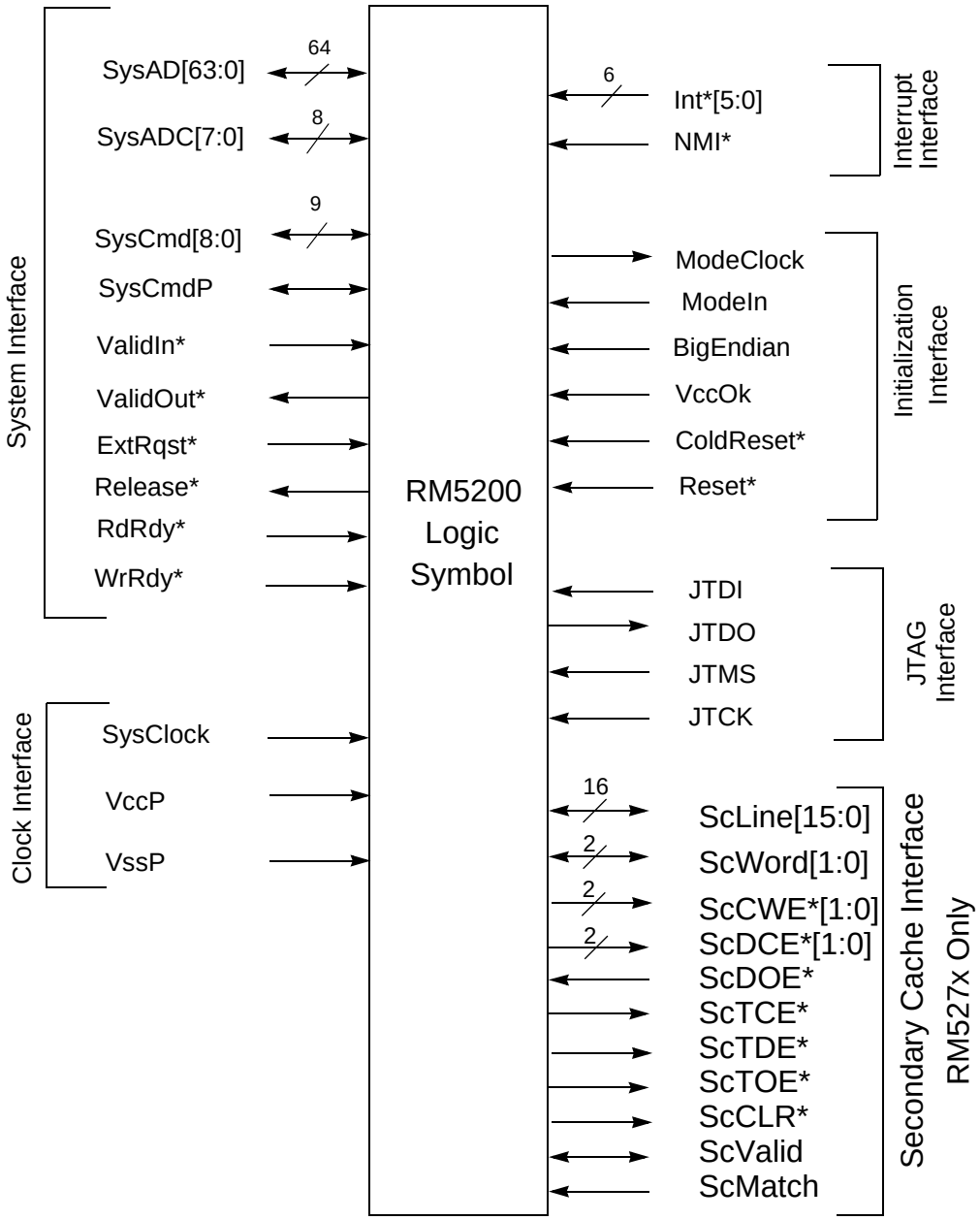


Figure 7.2 RM526x/7x Processor Signals

7.3 System Interface Signals

System interface signals provide the connection between the RM5200 processors and the other components in the system. Table 7.1: lists the system interface signals.

7.3.1 System Address/Data Bus

The System Address Data (**SysAD**) bus is used to transfer addresses and data between the RM5200 and the rest of the system. The **SysAD** for the RM523x is 32-bits and for the RM526x/7x it is 64-bits. The **SysAD** is protected with a parity check bus, **SysADC**. **SysADC** is 4-bits for the RM523x and 8-bits for the RM526x/7x.

The system interface is easily configurable to allow easy interfacing to memory and I/O systems of varying frequencies. The data rate and the bus frequency at which the RM5200 transmits data to the system interface are programmable at boot time via mode control bits. Also, the rate at which the processor receives data is fully controlled by the external device. Therefore, either a low cost interface requiring no read or write buffering or a faster, high performance interface can be designed to communicate with the RM5200.

7.3.2 System Command Bus

The RM5200 interface has a 9-bit System Command (**SysCmd**) bus. The command bus indicates whether the **SysAD** bus carries an address or data. If the **SysAD** carries an address, then the **SysCmd** bus also indicates what type of transaction is to take place (for example, a read or write). If the **SysAD** carries data, then the **SysCmd** bus also gives information about the data (for example, this is the 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 RM5200. Processor requests are initiated by the RM5200 and responded to by an external device. External requests are issued by an external device and require the RM5200 to respond.

The RM5200 supports one to eight byte and 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.

7.3.3 Handshake Signals

There are six handshake signals on the system interface. Two of these, **RdRdy*** and **WrRdy***, are used by an external device to indicate to the RM5200 whether it can accept a new read or write transaction. The RM5200 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. When an external device needs to control the interface, it asserts **ExtRqst***. The RM5200 responds by asserting **Release*** to release the system interface to slave state.

ValidOut* and **ValidIn*** are used by the RM5200 and the external device respectively to indicate that there is a valid command or data on the **SysAD** and **SysCmd** buses. The RM5200 asserts **ValidOut*** when it is driving these busses 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.

Table 7.1: System Interface Signals

Name	Definition	Direction	Description
SysAD(63:0) *RM526x/7x* SysAD(31:0) *RM523x*	System address/data bus	Input/Output	Address and data bus for communication between the processor and an external agent.
SysADC(7:0) *RM526x/7x* SysADC(3:0) *RM523x*	System address/ data check bus	Input/Output	Bus containing parity for the SysAD bus during data cycles.

Name	Definition	Direction	Description
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.
RdRdy*	Read ready	Input	An external agent asserts RdRdy* to indicate that it can accept a processor read.
WrRdy*	Write ready	Input	The external agent asserts WrRdy* when it can accept a processor write request.
ExtRqst*	External request	Input	An external agent asserts ExtRqst* to request use of the System interface. The processor grants the request by asserting Release*.
Release*	Release interface	Output	In response to the assertion of ExtRqst*, the processor asserts Release*, signalling to the requesting device that the System interface is available.
ValidIn*	Valid input	Input	The external agent asserts ValidIn* when it is driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.
ValidOut*	Valid output	Output	The processor asserts ValidOut* when it is driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus to the external agent.
SysCmdP	Reserved for System command/data identifier bus parity	Input/Output	Unused on input and zero on output. This signal is defined to maintain R4000 compatibility.

7.4 Clock Interface Signals

The Clock interface signals make up the interface for clocking. Table 7.2: lists the Clock interface signals.

Table 7.2: 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 (see Appendix E).
VssP	Quiet Vss for PLL	Input	Quiet Vss for the internal phase locked loop (see Appendix E).

7.5 Secondary Cache Interface Signals

Secondary Cache interface signals constitute the interface between the RM527x processor and secondary cache. Table 7.3: lists the Secondary Cache interface signals in alphabetical order.

Table 7.3: Secondary Cache Interface Signals

Name	Definition	Direction	Description
ScCLR*	Secondary Cache Flash Clear	Output	Clears all valid bits in those Tag RAMs which support this function.
ScCWE*(1:0)	Secondary Cache Write Enable	Output	Asserted during writes to the secondary cache. Two signals are provided to minimize loading from the cache RAMs.
ScDCE*(1:0)	Data RAM Chip Enable	Output	Chip Enable for Secondary Cache Data RAM. Two signals are provided to minimize loading from the cache RAMs.

Name	Definition	Direction	Description
ScDOE*	Data RAM Output Enable	Input	Asserted by the external agent to enable data onto the **SysAD** bus
ScLine (15:0)	Secondary Cache Line Index	Output	Cache line index for secondary cache
ScMatch	Secondary cache Tag Match	Input	Asserted by Tag RAM on Secondary cache tag match
ScTCE*	Secondary cache Tag RAM Chip Enable	Output	Chip enable for secondary cache tag RAM.
ScTDE*	Secondary cache Tag RAM Data Enable	Output	Data Enable for Secondary Cache Tag RAM.
ScTOE*	Secondary cache Tag RAM Output Enable	Output	Tag RAM Output enable for Secondary Cache Tag RAM
ScWord (1:0)	Secondary cache Word Index	Input/Output	Determines the double-word within the indexed secondary cache Index
ScValid	Secondary cache Valid	Input/Output	Always driven by the CPU except during a CACHE Probe operation, where it is driven by the Tag RAM.

7.6 Interrupt Interface Signals

The Interrupt interface signals make up the interface used by external agents to interrupt the RM5200 processor. Table 7.4: lists the Interrupt interface signals.

Table 7.4: Interrupt Interface Signals

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

7.7 Initialization Interface Signals

The Initialization interface signals make up the interface by which an external agent initializes the processor operating parameters. The fundamental operational modes for the processor are initialized by the boot-time mode control interface. The boot-time mode control interface is a serial interface operating at a very low frequency (**SysClock** divided by 256). The low frequency allows the initialization information to be kept in a low cost EPROM; alternatively the twenty or so bits could be generated by the system interface ASIC.

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 7.5: lists the Initialization interface signals.

Table 7.5: 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, ground the **BigEndian** pin.
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 data in	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 +3.3 volt power supply has been above 3.0 volts for more than 100 milliseconds and will remain stable. The assertion of **VccOk** initiates the reading of the boot-time mode control serial stream.

7.8 JTAG Interface Signals

The RM5200 interface supports JTAG boundary scan in conformance with IEEE 1149.1. The JTAG interface is especially helpful for checking the integrity of the processors pin connections. Table 7.6: lists the JTAG interface signals.

Table 7.6: JTAG Interface Signals

Name	Definition	Direction	Description
JTDI	JTAG Data In	Input	Data is serially scanned in through this pin.
JTCK	JTAG Clock input	Input	The processor accepts a serial clock on **JTCK**. On the rising edge of **JTCK**, both **JTDI** and **JTMS** are sampled.
JTDO	JTAG Data Out	Output	Data is serially scanned out through this pin on the falling edge of **JTCK**.
JTMS	JTAG Mode Select	Input	JTAG command signal, indicating the incoming serial data is command data.

Section 8 Clock Interface

8.1 Basic System Clocks

The various clock signals used in the RM5200 processors are described below, starting with **SysClock**, upon which the processor bases all internal and external clocking.

8.1.1 SysClock

The processor bases all internal and external clocking on the single **SysClock** input signal.

8.1.2 PClock

The processor generates an internal clock, PClock, at the initialization-interface-specified frequency multiplier of **SysClock** and phase-aligned to **SysClock**. All internal registers and latches use **PClock**.

8.1.3 Alignment to SysClock

- Processor output data changes a minimum of T_{dm} ns and becomes stable a maximum of T_{do} 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.
- 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**.

8.1.4 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_{jo} 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 8.1 shows the **SysClock** timing parameters.

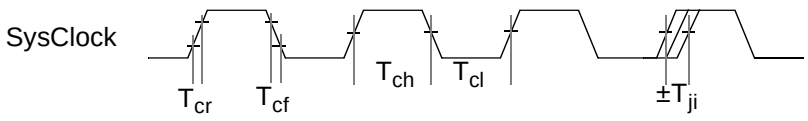


Figure 8.1 SysClock Timing

8.1.5 Input and Output Timing Diagrams

Figure 8.2 shows the input timing parameters.

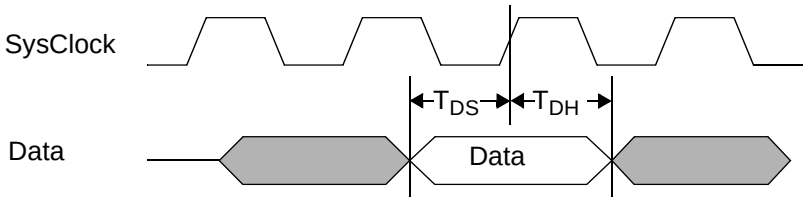


Figure 8.2 Input Timing

Figure 8.3 shows the output timing parameters measured at the midpoint of the rising clock edge.

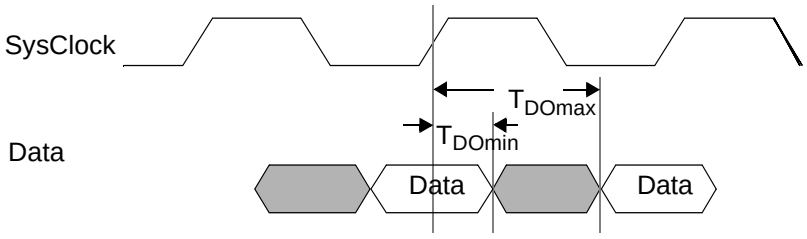


Figure 8.3 Output Timing

The **SysClock** input must meet the maximum rise time (T_{cr}), maximum fall time (T_{cf}), minimum T_{ch} time, minimum T_{cl} time, and T_{ji} input jitter parameters for proper operation of the PLL.

Please refer to the individual RM5200 datasheets for the values of the **SysAD** timing parameters.

Section 9 Initialization Interface

The RM5200 processors have the following three types of resets; they use the **VccOk, ColdReset***, and **Reset*** input signals.

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

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.

9.1 Processor Reset Signals

This section describes the three reset signals, **VccOK, ColdReset***, and **Reset***.

VccOk: When asserted[1], **VccOk** indicates to the processor that the +3.3 volt power supply (Vcc) has been above 3.0 volts for more than 100 milliseconds (ms) and is expected to remain stable. The assertion of **VccOk** initiates the reading of the boot-time mode control serial stream (described in 9.2, in this chapter).

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, at the **SysClock** frequency divided by 256 (**SysClock**/256).

Table 9.1: RM5200 Processor Signal Summary

Description	Name	I/O	Asserted State	Tristate	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

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

Description	Name	I/O	Asserted State	Tristate	Reset State
Write ready	WrRdy*	I	Low	No	N/A
Interrupts	Int[5:0]*	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
Secondary Cache Flash Clear	ScCLR*	O	Low	Yes	(2)
Secondary Cache Chip Write Enable	ScCWE(1:0)*	O	Low	Yes	(2)
Secondary Cache Data Chip Enable	ScDCE(1:0)*	O	Low	Yes	(2)
Secondary Cache Data Output Enable	ScDOE*	I	Low	No	N/A
Secondary Cache TAG Match	ScMatch	I	High	No	N/A
Secondary Cache TAG Chip Enable	ScTCE*	O	Low	Yes	(2)
Secondary Cache Device Enable	ScTDE*	O	Low	Yes	(2)
Secondary Cache TAG Output Enable	ScTOE*	O	Low	Yes	(2)
Secondary Cache Line index	ScLine[15:0]	O	High	Yes	(1)
Secondary Cache Word index	ScWord[1:0]	I/O	High	Yes	(1)
Secondary Cache TAG Valid	ScValid	I/O	High	Yes	(2)

Key to Reset State Column:

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

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

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

Note 4: ModeClock is always driven

Note 5: Na - Not Applicable to input pins.

9.1.1 Power-on Reset

The sequence for a power-on reset is listed below.

1. Power-on reset applies a stable **VccIO** of at least +3.0 volts from the +3.3 volt power supply to the processor. And for the RM52x1 devices, it applies a stable **VccInt** of at least 2.25 volts from the 2.5 supply to the processor. It also supplies a stable, continuous system clock at the processor operational frequency.
2. After at least 100 ms of stable Vcc and **SysClock**, the **VccOk** signal is asserted to the processor. The assertion of **VccOk** 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**.
4. After **ColdReset*** is deasserted synchronously, **Reset*** is deasserted to allow the processor to begin running. **Reset*** must be held asserted for at least 64 **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 9.1 shows the power-on system reset timing diagram.

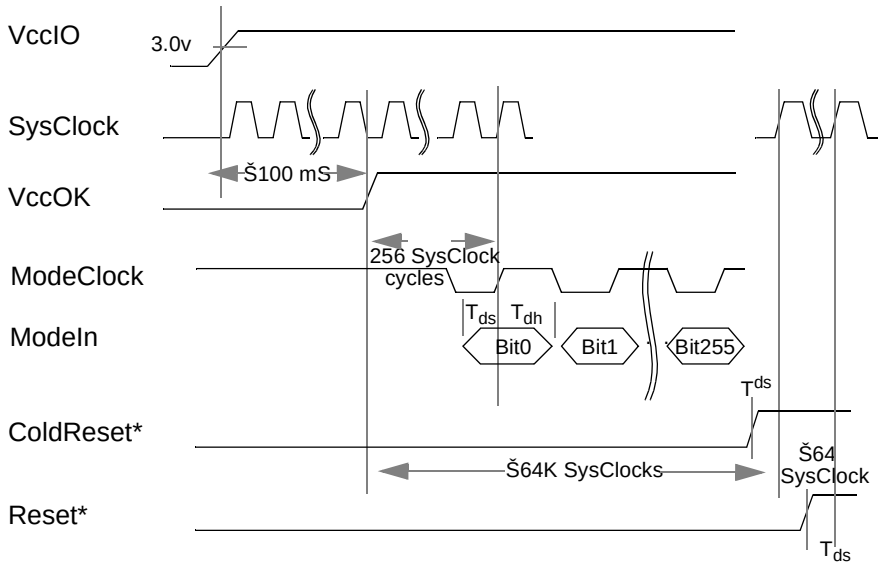


Figure 9.1 Power-On Reset Timing Diagram

9.1.2 Cold Reset

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

To begin the reset sequence, **VccOk** must be deasserted for a minimum of at least 64 MasterClock cycles before reassertion.

Figure 9.2 shows the cold reset timing diagram.

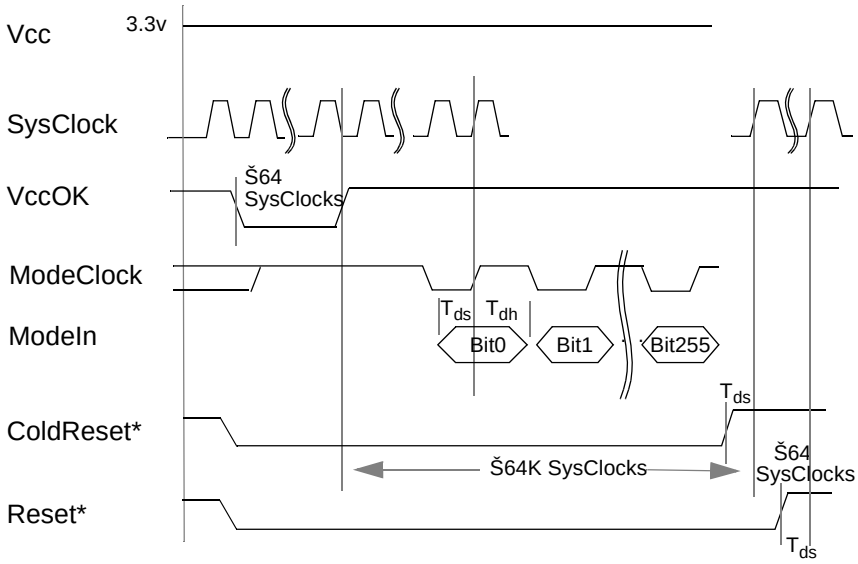


Figure 9.2 Cold Reset Timing Diagram

9.1.3 Warm Reset

To execute a warm reset, the **Reset*** input is asserted synchronously with **SysClock**. It is then held 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 start with a Soft Reset exception.

Figure 9.3 shows the warm reset timing diagram.

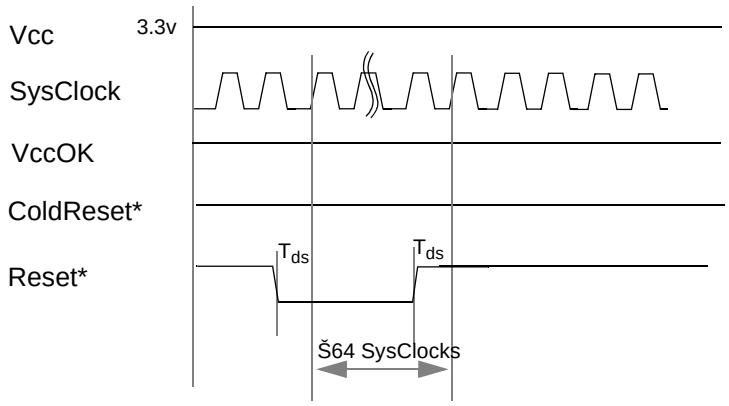


Figure 9.3 Warm Reset Timing Diagram

9.1.4 Processor Reset State

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

9.2 Initialization Sequence

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

The 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 the **ModeClock**. The processor samples 256 initialization bits from the **ModeIn** input.

9.3 Boot-Mode Settings

The rate at which the RM5200 writes block data to the external agent is controlled by boot mode bit 1 to 4. Boot mode bits 9 and 10 select the non-block write rate.

The ratio of the system clock to the pipeline clock is set by mode bits 5 to 7. For the RM52x1, if mode bit 20 is a zero, then only integer multipliers are available. If mode bit 20 is a one, then half-integer multipliers are also available. Half-integer multipliers are not available on the RM52x0 - mode bit 20 must be a zero.

The drive strength of the RM5200 output drivers is statically controlled at boot time. The output driver strength can be from 100% (fastest) to 50% (slowest), based on the value of boot mode bits 13 and 14.

On the RM527x, boot mode bit 15 selects between Dual Cycle Deselect (DCE) and Single Cycle Deselect (SCD) SRAMs for the external cache. Setting this bit allows commodity SRAM to be used for the external cache.

On the R5000, boot mode bits 16 and 17 were used to specify the external cache size. These bits, both on the R5000 and the RM5200, are software visible in processor configuration register bits 20 and 21. They have no other affect on the hardware and, therefore, can be use for any system design purpose.

The following rules apply to the boot-mode settings:

- Bit 0 of the stream is latched by the processor on the first rising edge of **ModeClock** following **VccOk** being asserted.
- Selecting a reserved value results in undefined processor behavior.
- Zeros must be scanned in for all reserved bits.
- A ‘D’ implies a doubleword (64-bit) datum
- A ‘W’ implies a word (32-bit) datum

Table 9.2: shows the boot mode settings for the RM523x, RM526x, and RM527x.

Table 9.2: Boot Mode Settings

Bit	Value	RM5230	RM5231
0	0	Reserved: Must be zero	
4:1	**32-bit Write-back data rate**		
	0	WWWWWWWW	
	1	WWxWWxWWxWWx	
	2	WWxxWWxxWWxxWWxx	
	3	WxWxWxWxWxWxWxWx	
	4	WWxxxWWxxxWWxxxWWxxx	
	5	WWxxxxWWxxxxWWxxxxWWxxxx	
	6	WxxWxxWxxWxxWxxWxxWxxWxx	
	7	WWxxxxxxWWxxxxxxWWxxxxxxWWxxxxxx	
	8	WxxxWxxxWxxxWxxxWxxxWxxxWxxxWxxx	
	9:15	Reserved	
7:5	**SysClkRatio: Pclock to SysClock Multiplier**		
	0	Multiply by 2	Multiply by 2
	1	Multiply by 3	Multiply by 3
	2	Multiply by 4	Multiply by 4
	3	Multiply by 5	Multiply by 5, 2.5
	4	Multiply by 6	Multiply by 6
	5	Multiply by 7	Multiply by 7, 3.5
	6	Multiply by 8	Multiply by 8
	7	Reserved	Multiply by 9, 4.5
8	**EndBit: Specifies byte ordering. Logically OR’ed with the BigEndian signal**		
	0	Little-Endian	
	1	Big Endian	
10:9	**Non-Block Write: Determines how non-block writes are handled.**		
	0	R4000 compatible	
	1	Reserved	
	2	Pipelined non-block writes	
	3	Non-block Write re-issue	

Bit	Value	RM5230	RM5231
11	**TmrIntEn: Disables Timer Interrupt on Int*[5]**		
	0	Timer Interrupt Enabled	
	1	Timer Interrupt Disabled	
12	0	Reserved: Must be zero	
14:13	**DrvOut: Output driver slew rate control**		
	00	67%	
	01	50% (slowest)	
	10	100% (fastest)	
	11	83%	
15	0	Reserved: Must be zero	
17:16	**User configuration identifiers - software visible in processor Config[21..20] register**		
19..18	0	Reserved: Must be a zero	
20		Reserved: Must be a zero	**Select SysClock to Pclock multiply mode**
	0		Integer Multiplier
	1		Half-integer Multiplier
21	1	Reserved: Must be one	
255:22	0	Reserved: Must be zero	

Bit	Value	RM5260	RM5261
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	**SysClkRatio: Pclock to SysClock Multiplier**		
	0	Multiply by 2	Multiply by 2
	1	Multiply by 3	Multiply by 3
	2	Multiply by 4	Multiply by 4
	3	Multiply by 5	Multiply by 5, 2.5
	4	Multiply by 6	Multiply by 6
	5	Multiply by 7	Multiply by 7, 3.5
	6	Multiply by 8	Multiply by 8
	7	Reserved	Multiply by 9, 4.5
8	**EndBit: Specifies byte ordering. Logically OR'ed with the BigEndian signal**		
	0	Little-Endian	
	1	Big Endian	

Bit	Value	RM5260	RM5261
10:9	**Non-Block Write: Determines how non-block writes are handled.**		
	0	R4000 compatible	
	1	Reserved	
	2	Pipelined non-block writes	
	3	Non-block write re-issue	
11	**TmrIntEn: Disables Timer Interrupt on Int*[5]**		
	0	Timer Interrupt Enabled	
	1	Timer Interrupt Disabled	
12	0	Reserved: Must be zero	
14:13	**DrvOut: Output driver slew rate control**		
	00	67%	
	01	50% (slowest)	
	10	100% (fastest)	
	11	83%	
15	0	Reserved: Must be zero	
17:16	**User configuration identifiers - software visible in processor Config[21..20] register**		
19..18		Reserved: Must be zero	
20		Reserved: Must be zero	**Select SysClock to Pclock multiply mode**
	0		Integer Multiplier
	1		Half-integer Multiplier
255:21	0	Reserved: Must be zero	

Bit	Value	RM5270	RM5271
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	**SysClkRatio: Pclock to SysClock Multiplier**		
	0	Multiply by 2	Multiply by 2
	1	Multiply by 3	Multiply by 3
	2	Multiply by 4	Multiply by 4
	3	Multiply by 5 or 2.5	Multiply by 5, 2.5
	4	Multiply by 6	Multiply by 6
	5	Multiply by 7 or 3.5	Multiply by 7, 3.5
	6	Multiply by 8	Multiply by 8
	7	reserved	Multiply by 9, 4.5

Bit	Value	RM5270	RM5271
8	**EndBit: Specifies byte ordering. Logically OR'ed with the BigEndian signal**		
	0	Little-Endian	
	1	Big Endian	
10:9	**Non-Block Write: Determines how non-block writes are handled.**		
	00	R4000 compatible	
	01	Reserved	
	10	Pipelined non-block writes	
	11	Non-block write-reissue	
11	**TmrIntEn: Disables Timer Interrupt on Int5***		
	0	Timer Interrupt Enabled	
	1	Timer Interrupt Disabled	
12	**Secondary cache present**		
	0	Not Present	
	1	Present	
14:13	**DrvOut: Output driver slew rate control**		
	00	67%	
	01	50% (slowest)	
	10	100% (fastest)	
	11	83%	
15	**Select secondary cache RAM type**		
	0	Dual Cycle Deselect SRAMs	
	1	Single Cycle Deselect SRAMs	
17:16	**User configuration identifiers - software visible in processor Config[21..20] register**		
19..18	0	Reserved: Must be zero	
20		Reserved: Must be zero	**Select SysClock to Pclock multiply mode**
	0		Integer Multiplier
	1		Half-integer Multiplier (RM5271
255:21	0	Reserved: Must be zero	

Section 10 System Interface Transactions

There are two broad categories of transactions: *processor requests* and *external requests*. *Processor requests* are issued, either singularly or in series, by the RM5200 through the System Interface to access an external resource. An *external request* is issued by an external resource to access a processor internal resource. This section describes these request along with the tertiary cache transactions.

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

10.2 Processor Requests

The RM5200 issues either a single request or a series of requests—called *processor requests*—through the System interface, to access an external resource. For this to work, the processor System interface must be connected to an external agent that is compatible with the System interface protocol, and can coordinate access to system resources.

An external agent requesting access to a processor internal resource generates an *external request*. This access request passes through the System interface. System events and request cycles are shown in Figure 10.1.

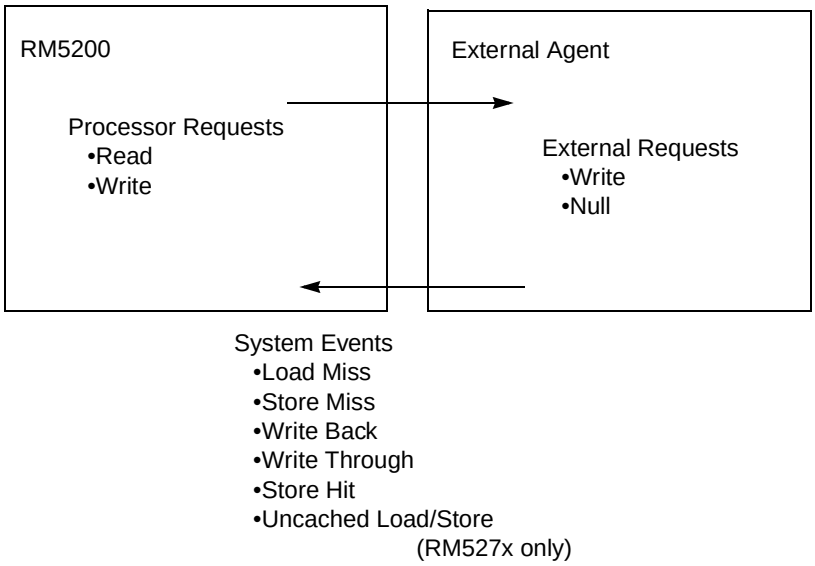


Figure 10.1 Requests and System Events

10.2.1 Rules for Processor Requests

A processor request is a request or a series of requests, through the System interface, to access some external resource. As shown in Figure 10.2, processor requests include read and write.

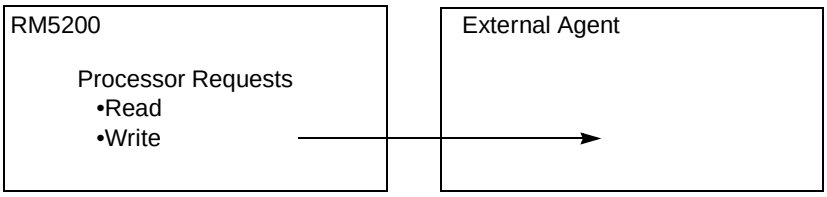


Figure 10.2 Processor Requests to External Agent

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

Write request provides a block, doubleword, partial doubleword, word, or partial word of data to be written either to main memory or to another system resource.

The RM5200 requires a non-overlapping system interface, compatible with the 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 RM5200 issues another request. The RM5200 can issue read and write requests to an external device, whereas an external device can issue null and write requests to the RM5200.

For processor reads the RM5200 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***. The external device can then drive data via the **SysAD** bus to the RM5200.

The processor has the input signals **RdRdy*** and **WrRdy*** to allow an external agent to manage the flow of processor requests. **RdRdy*** controls the flow of processor read requests, while **WrRdy*** controls the flow of processor write requests. The processor request cycle sequence is shown in Figure 10.3.

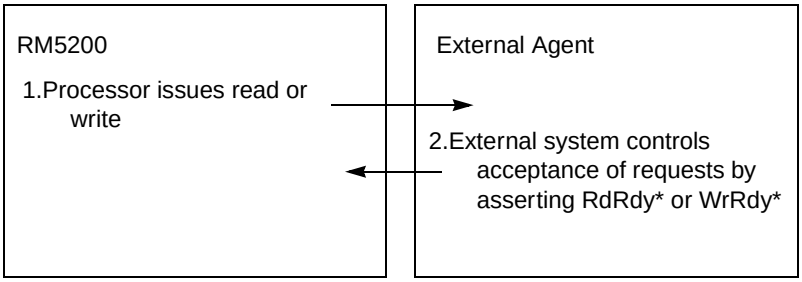


Figure 10.3 Processor Request Flow Control

10.2.2 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 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.

10.2.3 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.

Like the R4600, R4700, and R5000, the RM5200 processors support two enhancements to the original R4000 write mechanism: *Write Reissue* and *Pipelined Writes*. In write reissue mode, a write rate of one write every two bus cycles can be achieved. A write issues if **WrRdy*** is asserted two cycles earlier and is still 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***.

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.

10.3 External Requests

External requests include write and null requests, as shown in Figure 10.4. This section also includes a description of read response, a special case of an external request.

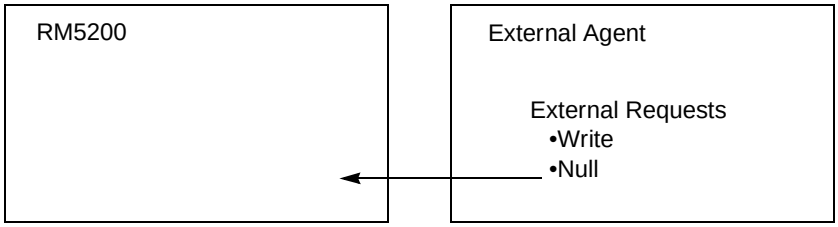


Figure 10.4 External Requests to Processor

Write request is executed by an external device when it wishes to update one of the processors writable resources such as the internal interrupt register.

Null request is executed when the external device wishes the processor to resume ownership of the processor external interface. That is, the external device wants the processor interface to go from slave state to master state. Typically, a null request will be executed after an external device that has acquired control of the processor interface via **ExtRqst*** and has completed a transaction between itself and system memory in a system where memory is connected directly to the SysAD bus. Normally this transaction would be a DMA read or write from the I/O system.

The processor controls the flow of external requests through the arbitration signals **ExtRqst*** and **Release***, as shown in Figure 10.5. 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 secondary cache is enabled, the secondary cache access must be resolved and be a miss. Otherwise the system interface returns to the master state.

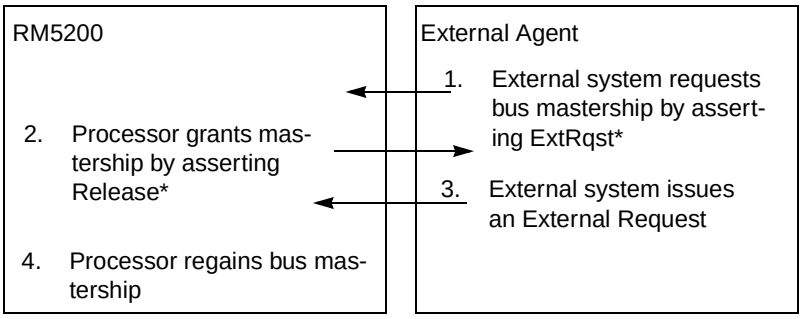


Figure 10.5 External Request Arbitration

Mastership of the System interface always returns to the processor after an external request is issued. The processor does not accept a subsequent external request until it has completed the current request.

If there are no processor requests pending, the processor decides, based on its internal state, whether to accept the external request, or to issue a new processor request. The processor can issue a new processor request even if the external agent is requesting access to the system interface.

The external agent asserts **ExtRqst*** indicating that it wishes to begin an external request. The external agent then waits for the processor to signal that it is ready to accept this request by asserting **Release***. The processor signals that it is ready to accept an external request based on the criteria listed below.

- The processor completes any request in progress.
- While waiting for the assertion of **RdRdy*** to issue a processor read request, the processor can accept an external request if the external request is delivered to the processor one or more cycles before **RdRdy*** is asserted.
- While waiting for the assertion of **WrRdy*** to issue a processor write request, the processor can accept an external request provided the external request is delivered to the processor one or more cycles before **WrRdy*** is asserted.
- If waiting for the response to a read request after the processor has made an uncompelled change to a slave state, the external agent can issue an external request before providing the read response data.

- Note that an external read request is undefined. The behavior of the processor to an external read request is undefined.

10.3.1 External Write Request

When an external agent issues a write request, the specified resource is accessed and the data is written to it. An external write request is complete after the word of data has been transmitted to the processor.

The only processor resource available to an external write request is the Interrupt register.

10.3.2 Read Response

A *read response* returns data in response to a processor read request, as shown in Figure 10.6. 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, read responses are handled separately from all other external requests, and are simply called read responses.

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

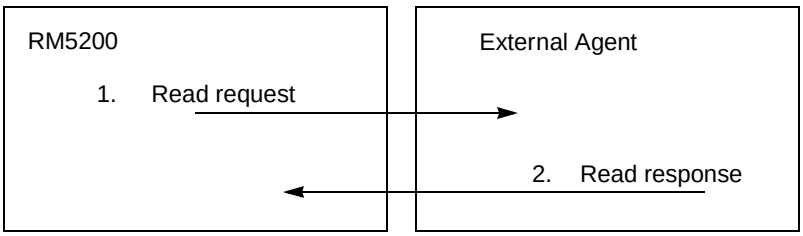


Figure 10.6 External Agent Read Response to Processor

10.4 Secondary Cache Transactions

For RM5270 and RM5271 processors configured with a secondary cache, the secondary cache is a special form of external agent that is jointly controlled by both the processor and the external agent. Figure 10.7 illustrates a processor request to the secondary cache and external agent.

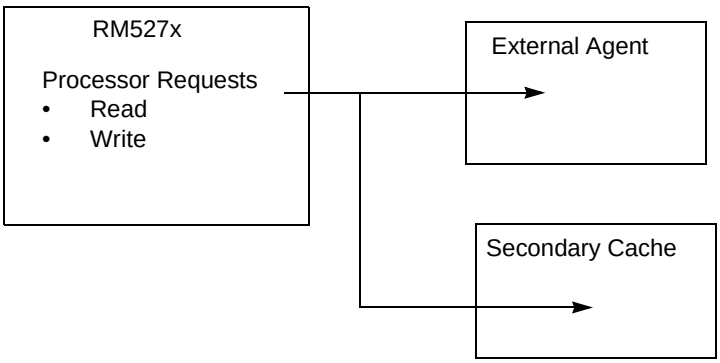


Figure 10.7 Processor Requests to Secondary Cache and External Agent

10.4.1 Secondary Cache Probe, Invalidate, and Clear

For secondary cache invalidate, clear, and probe operations, the secondary cache is controlled by the processor and the external agent is not involved in these operations. Issuance of secondary cache invalidate, clear, and probe operations is not flow-controlled and proceeds at the maximum data rate. Figure 10.8 and Figure 10.9 shows the secondary cache invalidate and tag probe operations.

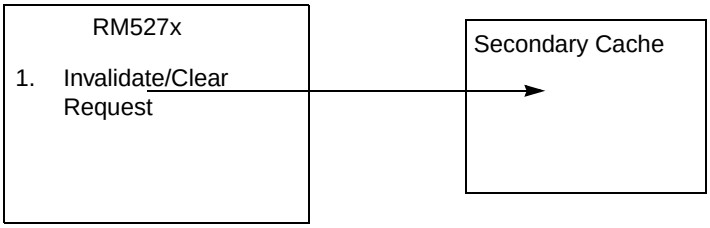


Figure 10.8 Secondary Cache Invalidate and Clear

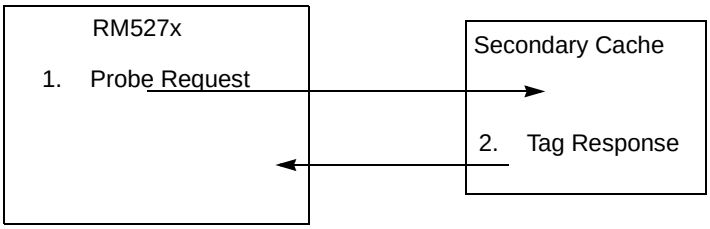


Figure 10.9 Secondary Cache Tag Probe

10.4.2 Secondary Cache Write

For secondary cache write-through, the processor issues a block write operation that is directed to both the secondary cache and the external agent. Issuance of secondary cache writes is controlled by the normal **WrRdy*** flow control mechanism. Secondary cache write data transfers proceed at the data transfer rate specified in the Mode ROM for block writes. Figure 10.10 illustrates a secondary cache write operation.

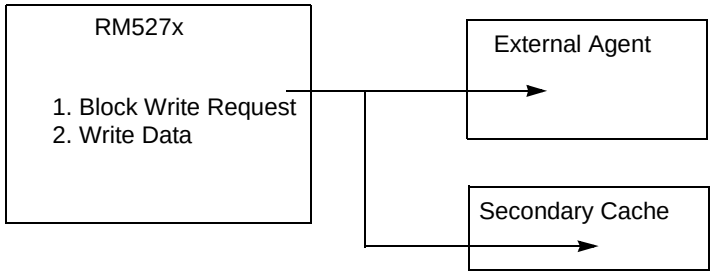


Figure 10.10 Secondary Cache Write Through

10.4.3 Secondary Cache Read

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

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

Figure 10.11 and Figure 10.12 shows a secondary cache read hit and miss respectively.

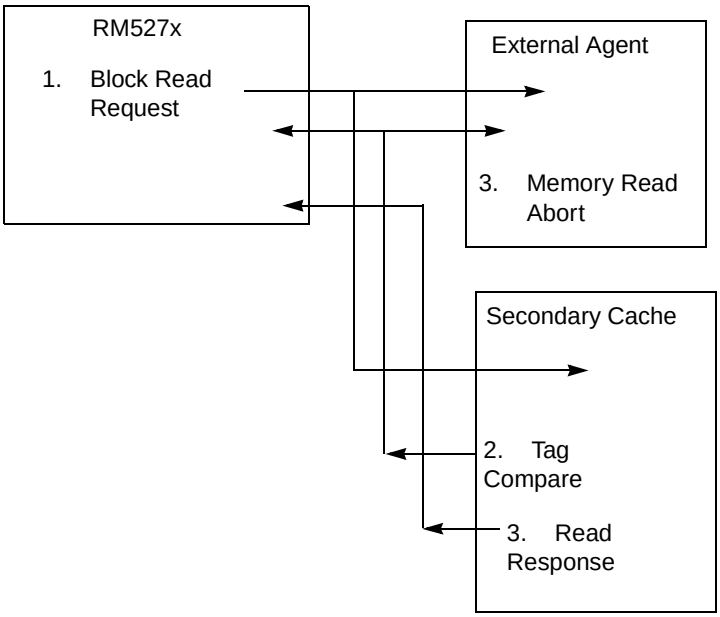


Figure 10.11 Secondary Cache Read Hit

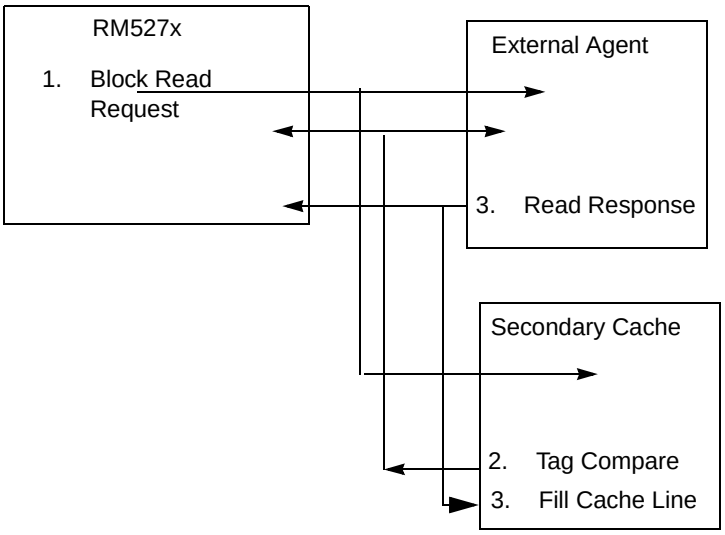


Figure 10.12 Secondary Cache Read Miss

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

10.5 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

10.5.1 Load Miss

When a processor load misses in the primary cache, before the processor can proceed it must obtain the cache line that contains the data element to be loaded from the external agent.

If the new cache line replaces a current dirty exclusive or dirty shared 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 10.1: shows the actions taken on a load miss to primary cache.

Table 10.1: Load Miss to Primary 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. If the secondary cache is enabled and the page coherency attribute is write-back, the response data will also be written into the secondary cache.
2. The processor then waits for an external agent to provide the read response.
3. The processor restarts the pipeline after the first doubleword of the data cache miss is received. The remaining three doublewords are placed in the cache after all three doublewords have been received and the dcache 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 secondary cache is enabled and the page attribute is write-back, the write back data will also be written into the secondary cache.

10.5.2 Store Miss

When a processor store misses in the primary cache, the processor may request, from the external agent, the cache line that contains the target location of the store for pages that are either write-back or write-through with write-allocate only. The processor examines the coherency attribute in the TLB entry for the page that contains the requested cache line to see if the cache line is being maintained with either a write-allocate or no-write-allocate policy.

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 *noncoherent* write-through with no write-allocate, a non-block write request is issued.

Table 10.2: shows the actions taken on a store miss to the primary cache.

Table 10.2: Store Miss to Primary Cache

Page Attribute	State of Data Cache Line Being Replaced	
	Clean/Invalid	Dirty (W=1)
Noncoherent-write-back or noncoherent-write-through with write-allocate	NCBR	NCBR/W
Noncoherent-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 non-coherent block read request for the cache line that contains the data element to be loaded, then waits for the external agent to provide read data in response to the read request. If the secondary cache is enabled and the page coherency attribute is write-back, the response data will also be written into the secondary cache. 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, no-write-allocate store miss, the processor issues a non-block write request only.

10.5.3 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.

10.5.4 Uncached Loads or Stores

When the processor performs an uncached load, it issues a noncoherent doubleword, partial doubleword, word, or partial word read request. When the processor performs an uncached store, it issues a doubleword, partial doubleword, word, or partial word write request. All writes by the processor are buffered from the system interface by a 4-entry write buffer. The write requests are sent to the system bus only when no other requests are in progress. However, once the emptying of the write buffer has begun, it is allowed to complete. Therefore, if the write buffer contains any entries when a block read is requested, the write buffer is allowed to empty before the block read request is serviced. Uncached loads and stores do not affect the secondary cache.

10.5.5 Uncached Instruction Fetch

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

To access the Boot ROM in an uncached address space the RM523x does two 32-bit instruction fetches and then issues both instructions.

The RM526x/7x accesses Boot ROM in this space with two 32-bit instruction fetches as well. In little endian mode, the first instruction (even word) will be from **SysAD[31:0]** with **SysAD[63:32]** being ignored. The second instruction (odd word) will be from **SysAD[63:32]** with **SysAD[31:0]** being ignored.

In big endian mode, the first instruction (even word) will be from **SysAD[63:32]** with **SysAD[31:0]** being ignored while the second instruction (odd word) will be 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 RM526x/7x 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.

10.5.6 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 11 System Interface Protocols

The following sections contain a cycle-by-cycle description of the system interface protocols for each type of processor and external request.

11.1 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 determined by the state of the **ValidOut*** signal. Validity of addresses and data from the external agent is determined by the state of the **ValidIn*** signal. Validity of data from the secondary cache is determined by the state of the pipelined **ScDCE*** and **ScCWE*** signals from the processor and the **ScDOE*** 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 **SysCmd(8)** = 0. The remainder of the **SysCmd** bus, **SysCmd(7:0)**, contains the encoded system interface command.
- During data cycles [**SysCmd(8)** = 1], the remainder of the **SysCmd** bus, **SysCmd(7:0)**, contains an encoded data identifier. There is no **SysCmd** associated with a secondary cache read response.

11.2 Issue Cycles

There are two types of processor issue cycles:

- processor read request.
- processor write request.

The processor samples the signal **RdRdy*** to determine the issue cycle for a processor read; the processor samples the signal **WrRdy*** to determine the issue cycle of a processor write request.

As shown in Figure 11.1, the issue cycle of a processor read request is the third cycle following the assertion of **RdRdy*.** This cycle will contain a valid address and command. The **SysAD** bus, the **SysCmd** bus, and **ValidOut*** may assert a valid address and command prior to this cycle.

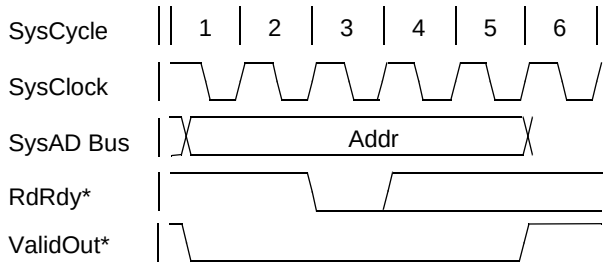


Figure 11.1 State of RdRdy* Signal for Read Requests

As shown in Figure 11.2, the issue cycle of a processor write request is the third cycle following the assertion of **WrRdy***. This cycle will contain a valid address and command. The **SysAD** bus, the **SysCmd** bus, and **ValidOut*** may assert a valid address and command prior to this cycle.

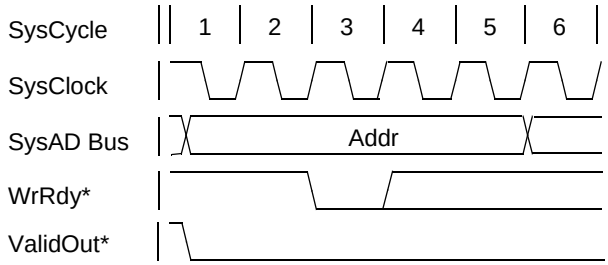


Figure 11.2 State of WrRdy* Signal for Write Requests

The processor repeats the address cycle for the request until the conditions for a valid issue cycle are met. After the issue cycle, if the processor request requires data to be sent, the data transmission begins. There is only one issue cycle for any processor request.

The processor accepts external requests, even while attempting to issue a processor request, by releasing the System interface to slave state in response to an assertion of **ExtRqst*** by the external agent.

Note that the rules governing the issue cycle of a processor request are strictly applied to determine 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.

11.3 Handshake Signals

The RM5200 processor manages the flow of requests through the following six control signals:

- **RdRdy***, **WrRdy*** are used by the external agent to indicate when it can accept a new read (**RdRdy***) or write (**WrRdy***) transaction.
- **ExtRqst***, **Release*** are used to transfer control of the **SysAD** and **SysCmd** buses. **ExtRqst*** is used by an external agent to indicate a need to control the interface. **Release*** is asserted by the processor when it transfers the mastership of the System interface to the external agent. For secondary cache reads, assertion of **Release*** to the external agent is speculative, and is aborted if there is a hit in the secondary cache.
- The RM5200 processors assert **ValidOut*** and the external agent asserts **ValidIn*** to indicate valid command/data on the **SysCmd**/**SysAD** buses.

11.4 System Interface Operation

Figure 11.3 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.**

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.

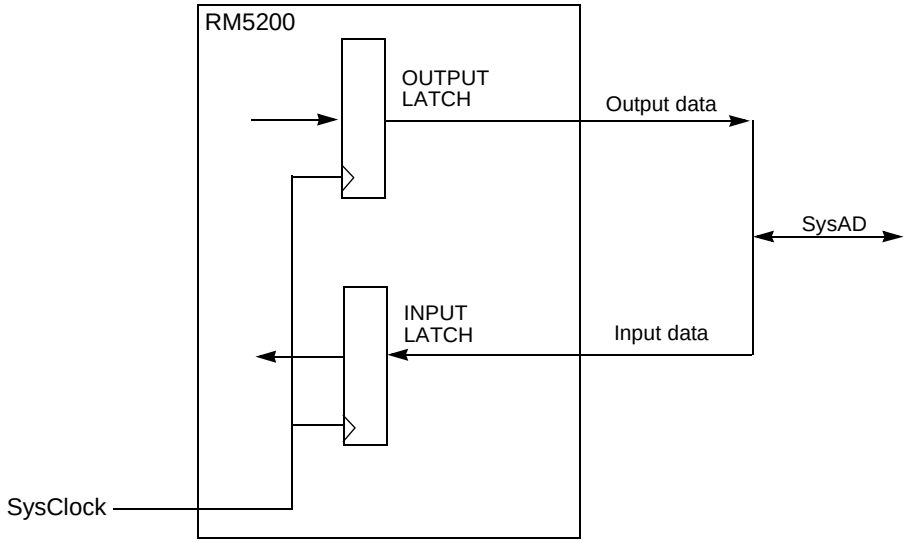


Figure 11.3 System Interface Register-to-Register Operation

11.4.1 Master and Slave States

When the RM5200 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 secondary cache is driving the **SysAD** and **SysADC** buses, the System interface is in slave 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 and the secondary cache drives the **SysAD** and **SysADC** buses in response to the **ScDCE***, **ScCWE***, and **ScDOE*** signals.

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

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

11.4.2 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.

11.4.3 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 after a read request 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 secondary cache is enabled and a secondary cache hit occurs, then the bus is returned to master state.

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

Table 11.1: 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 external request release null 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

11.5 Processor Request Protocols

Processor request protocols described in this section include:

- read
- write

Note: In the timing diagrams, the two closely spaced, wavy vertical lines, such as those shown in Figure 11.4, indicate one or more identical cycles which are not illustrated due to space constraints.

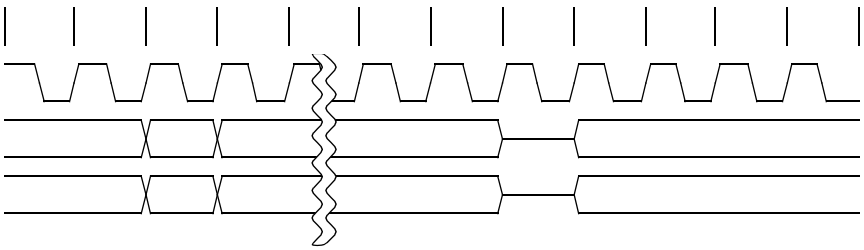


Figure 11.4 Symbol for Undocumented Cycles

11.5.1 Processor Read Request Protocol

The following sequence describes the protocol for doubleword, partial doubleword, word and partial word processor read requests. The secondary cache block read request protocol is described later in this section. The numbered steps below correspond to Figure 11.5.

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. For the RM526x/7x 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. For the RM523x the physical address is driven onto **SysAD[31:0]**. The RM523x does not drive virtual address bits [13:12].

   *Note: At the same time, the processor asserts **ValidOut*** for one cycle, indicating valid data is present on the **SysCmd** and the **SysAD** buses.*

3. Only one processor read request can be pending at a time.
4. The processor makes an uncompelled change to slave state during the issue cycle of the read request. The external agent must not assert the signal **ExtRqst*** for the purposes of returning a read response, but rather must wait for the uncompelled change to slave state. The signal **ExtRqst*** can be asserted before or during a read response to perform an external request other than a read response.
5. The processor releases the **SysCmd** and the **SysAD** buses one **SysClock** after the assertion of **Release***.
6. The external agent drives the **SysCmd** and the **SysAD** buses within two cycles after the assertion of **Release***.

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.

Figure 11.5 illustrates a processor read request, coupled with an uncompelled change to slave state, that occurs as the read request is issued.

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

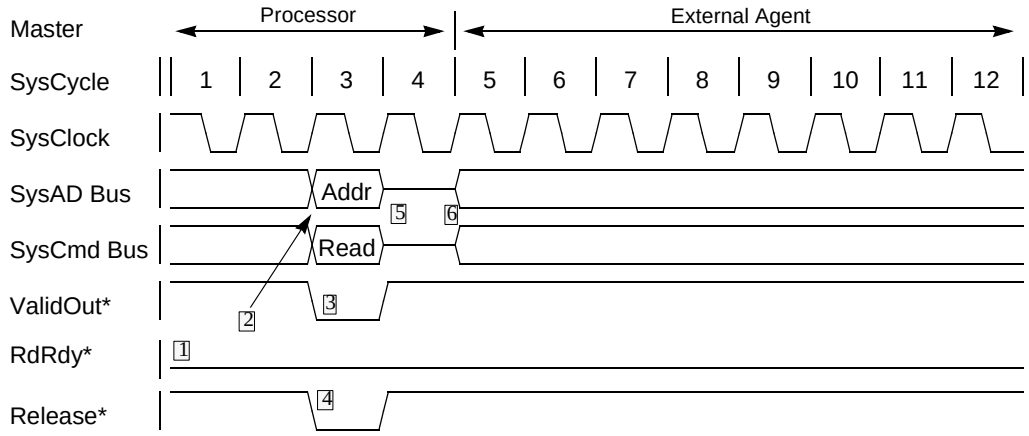


Figure 11.5 Processor Read Request Protocol

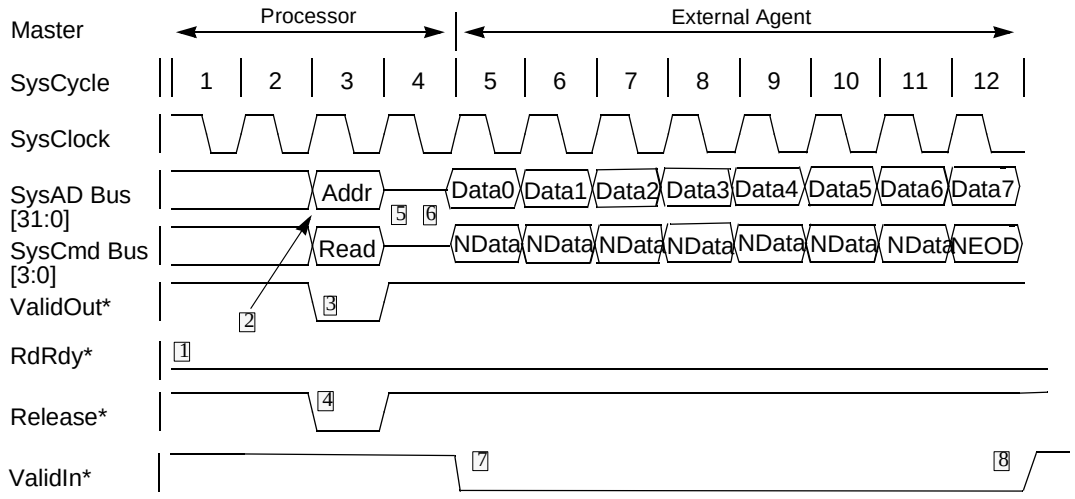


Figure 11.6 Processor Non-coherent, Non-Secondary Cache, Block Read Request, 32-bit bus (RM523x)

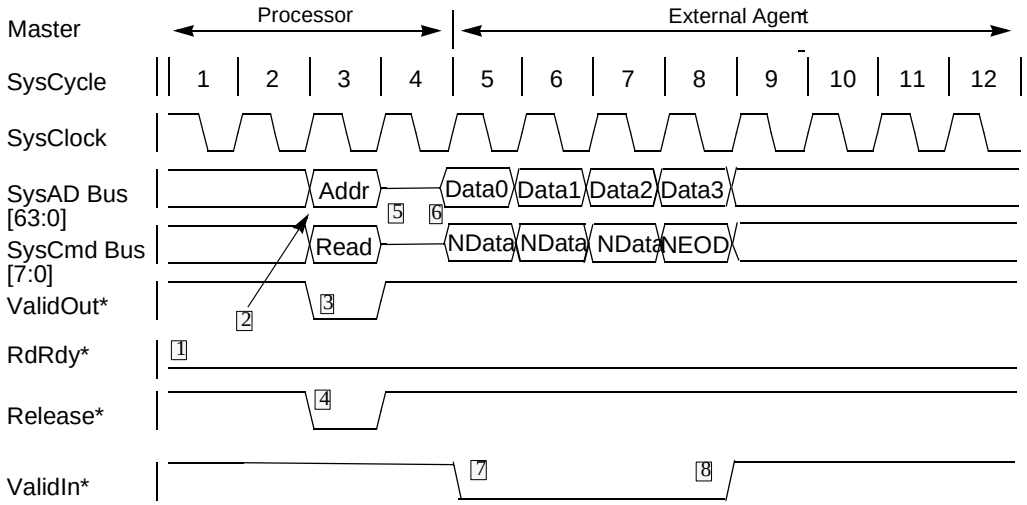


Figure 11.7 Processor Non-coherent, Non-Secondary Cache, Block Read Request, 64-bit bus (RM526x/7x)

Any time a read request has been issued (indicating a read request is pending), the processor will assert **Release*** 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 (see section 11.15). When an external request is accepted while a read is pending the processor will always perform an uncompelled change to slave state following the external request. An uncompelled change to slave state is signaled by **Release*** being asserted for one clock cycle.

Figure 11.6 shows a Processor Block Read transaction on the 32-bit Bus Interface (RM523x) with zero wait states. Figure 11.7 shows a Processor Block Read transaction on the 64-bit Bus Interface (RM526x/7x) with zero wait states. After the bus has granted to the External Agent by the uncompelled released:

7. The External Agent notifies the Processor that valid data is on the bus by asserting **ValidIn*** low and placing a Noncoherent Data identifier on the **SysCmd** bus.
8. 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 deasserting **ValidIn***. The processor only accepts data on the **SysAD** bus when **ValidIn*** is asserted and the **SysCmd** bus contains a data identifier.

11.5.2 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 protocol. The protocol used is selected by mode bit 9 and 10 at power-up.
- Cache block writes use a block write request protocol.
- Secondary cache block write request protocol.

Processor non-block write requests are issued with the System interface in master state, as described below in the steps below; Figure 11.8 shows a processor noncoherent non-block write request cycle.

1. **WrRdy*** is asserted low, indicating the external agent is ready to accept a write request.
2. A processor single non-block write request is issued by driving a write command on the **SysCmd** bus and a write address on the **SysAD** bus. For the RM526x/7x 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. For the RM523x the physical address is driven onto **SysAD[31:0]** and the virtual address bits [13:12] are not driven.
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 data cycle must contain a last data cycle indication. At the end of the cycle, **ValidOut*** is deasserted.

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

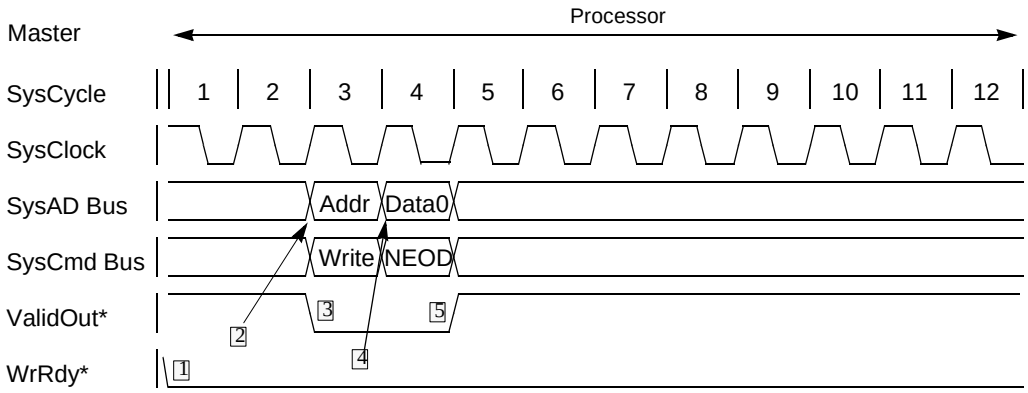


Figure 11.8 Processor Noncoherent Non-Block Write Request Protocol

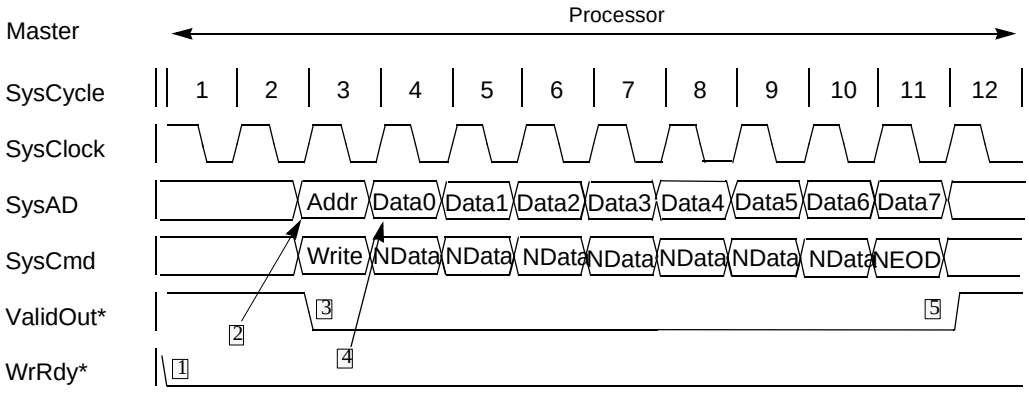


Figure 11.9 Processor Non-Coherent, Non-Secondary Cache Block Write Request, 32-bit bus (RM523x)

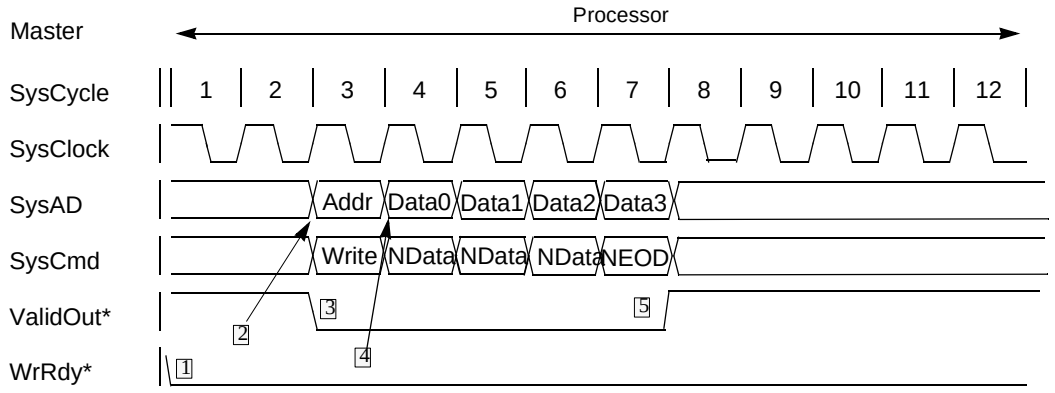


Figure 11.10 Processor Non-Coherent, Non-Secondary Cache Block Write Request, 64-bit bus (RM526x/7x)

11.5.3 System Interface Flow Control

The RM5200 implements two input pins to regulate the flow of instructions and data across it' s system interface.

The external agent generates **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. Negating the **RdRdy*** signal during a data cycle may delay any follow-on read cycle, but will not affect the present one since it has already issued.

Figure 11.11 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.

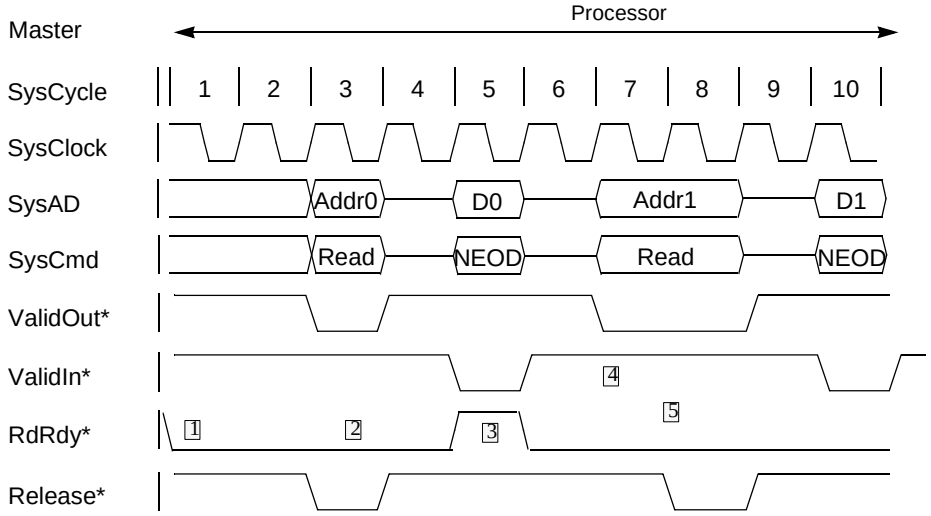


Figure 11.11 Processor Read Request Flow Control

The external agent generates **WrRdy*** to control the flow of processor writes. Figure 11.12 illustrates two processor write requests in which the issue of the second is delayed by the negation of **WrRdy***. Processor write requests are delayed by **WrRdy*** only at the Address cycle. Negating he **WrRdy*** signal during a data cycle may delay any follow-on read cycle, but will not affect the present one since it has already issued.

1. **WrRdy*** is asserted low, 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. This is followd by a data cycle during which **WrRdy*** is negated.
3. The second write request is delayed until two **SysClock** cycles after the **WrRdy*** signal is again asserted.
4. The processor does not complete the issue of a write request until it issues an address cycle in response to the write request for which the signal **WrRdy*** was asserted two cycles earlier.

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

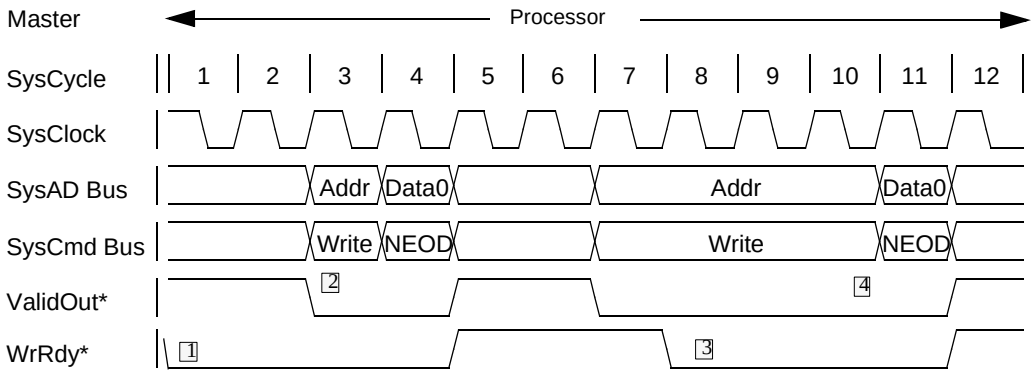


Figure 11.12 Two Processor Write Requests with Second Write Delayed

The processor interface requires that **WrRdy*** be asserted two system cycles prior to the issue of a write cycle. An external agent that negates **WrRdy*** immediately upon receiving the write that fills its buffer will suspend any subsequent writes for four system cycles in R4000 non-block write-compatible mode. The processor always inserts at least two unused system 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 controlled by **WrRdy*** in the same manner.

Figure 11.13 shows back-to-back write cycles in R4000-compatible mode. This 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.

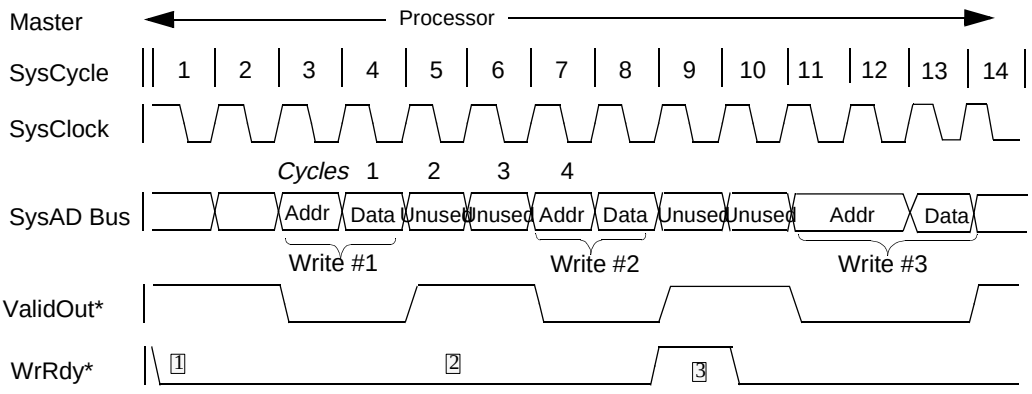


Figure 11.13 R4000-Compatible Back-to-Back Write Cycle Timing

An address/data pair every four system cycles is not sufficiently high performance for all applications. For this reason, the RM5200 processor provides two protocol options that modify the R4000 back-to-back write protocol to allow an address/data pair every two system cycles. These two protocols are as follows:

- *Write Reissue* allows **WrRdy*** to be negated during the address cycle, but forces the write cycle to be re-issued. This write mode is selected with mode bits 9 and 10 set to ones
- *Pipelined Writes* leave the sample point of **WrRdy*** unchanged, but requires that the external agent accept one more write than dictated by the R4000 protocol. This write mode is selected with mode bit 9 set to a one, and mode bit 10 set to a zero.

The write re-issue protocol is shown in Figure 11.14. Writes issue when **WrRdy*** is asserted two cycles prior to the address cycle. Writes re-issue when **WrRdy*** is not asserted at the end of a write issue address cycle.

If a processor 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 re-presented on the bus.

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 second address cycle. This write request is aborted, but will be reissued.
4. **WrRdy*** is asserted, indicating the external agent can accept a write request.
5. **WrRdy*** remains asserted, the write is re-issued, and the external agent is able to accept another write request.

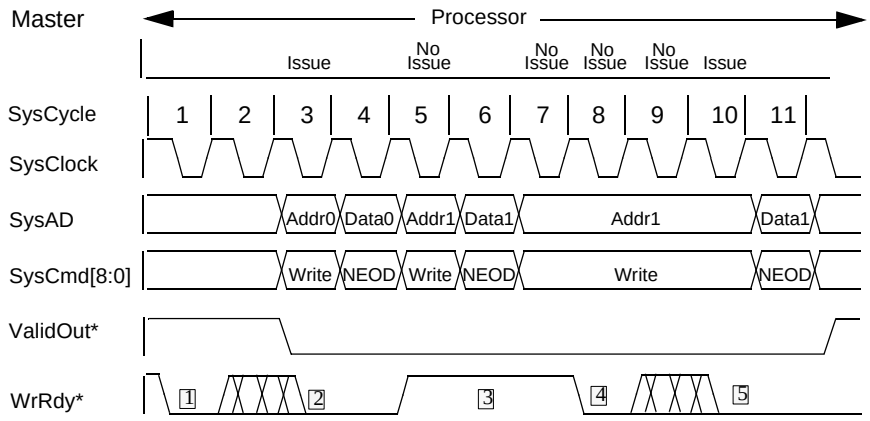


Figure 11.14 Write Reissue

The pipelined write protocol is shown in Figure 11.15. Writes issue when **WrRdy*** is asserted two cycles before the address cycle and the external agent is required to accept one more write after **WrRdy*** is negated.

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 during the second address cycle, indicating the external agent cannot accept a follow-on write request. It does, however, accept this write.
4. **WrRdy*** is asserted, indicating the external agent can accept a write request.

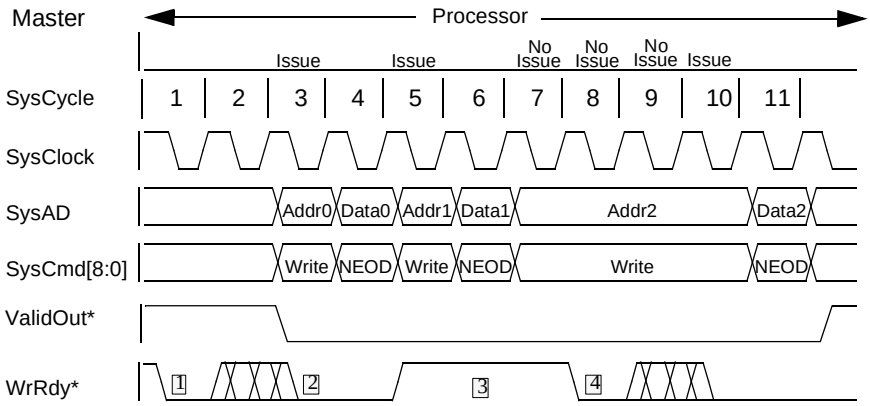


Figure 11.15 Pipelined Writes

11.6 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*** 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

11.6.1 External Arbitration Protocol

System interface arbitration uses the signals **ExtRqst*** and **Release*** as described above. Figure 11.16 is a timing diagram of the arbitration protocol, in which slave and master states are shown.

The arbitration cycle consists of the following steps:

1. The external agent asserts **ExtRqst*** when it wishes to submit an external request.
2. The processor waits until it is ready to handle an external request, whereupon it asserts **Release*** for one cycle.
3. The processor sets the **SysAD** and **SysCmd** buses to tristate.
4. The external agent must wait at least two cycles after the assertion of **Release*** before it drives the **SysAD** and **SysCmd** buses.
5. The external agent negates **ExtRqst*** two cycles after the assertion of **Release***, unless the external agent wishes to perform an additional external request.
6. The external agent sets the **SysAD** and the **SysCmd** buses to tristate at the completion of an external request.

The processor can start issuing a processor request one cycle after the external agent sets the bus to tristate.

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

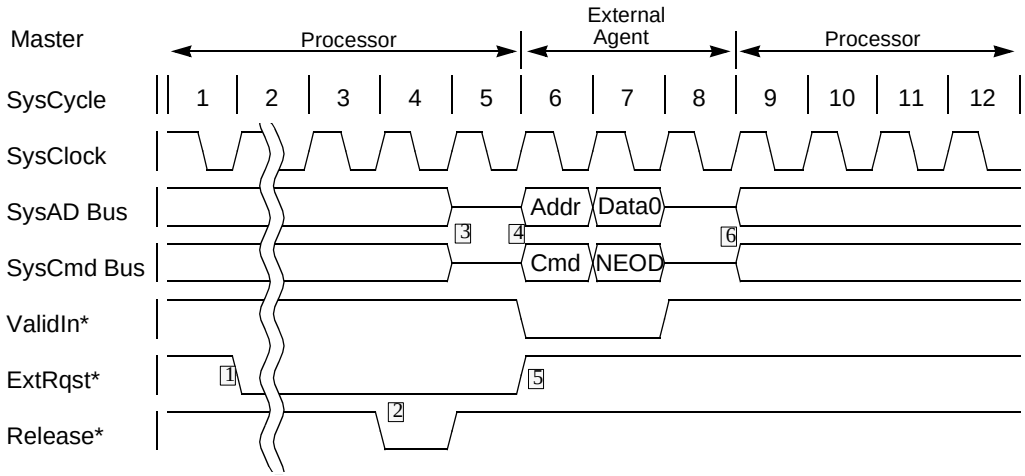


Figure 11.16 Arbitration Protocol for External Requests

11.6.2 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 11.17 shows a timing diagram of an external null request, which consist 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 ownership 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.

Fooolwing a *System interface external request release null* cycle, the external agent releases the **SysCmd** and **SysAD** buses, and expects the System interface to return to the master state.

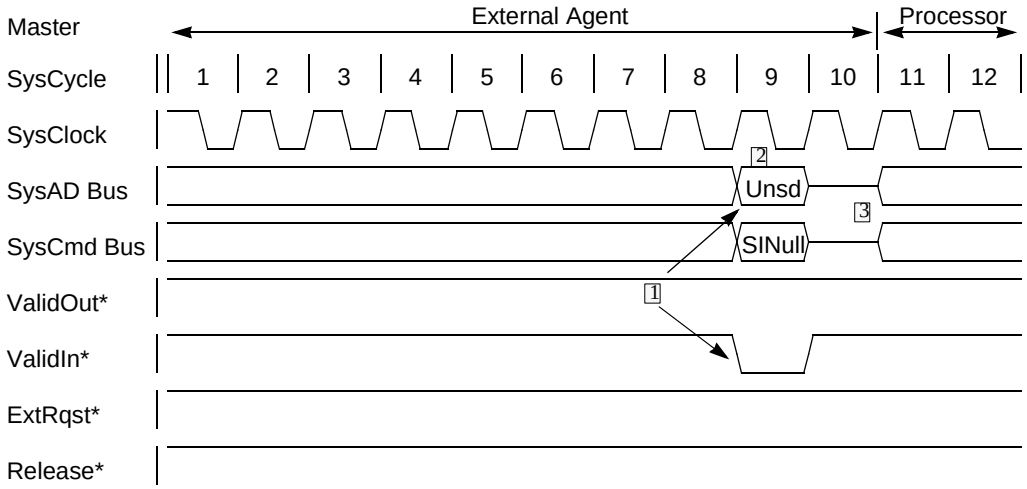


Figure 11.17 System Interface Release External Null Request

11.6.3 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 11.18 shows a timing diagram of an external write request, which consists of the following steps:

1. The external agent asserts **ExtRqst*** to arbitrate for the System interface.
2. The processor releases the System interface to slave state by asserting **Release***.
3. The external agent drives a write command on the **SysCmd** bus, a write address on the **SysAD** bus, and asserts **ValidIn***.
4. The external agent drives a data identifier on the **SysCmd** bus, data on the **SysAD** bus, and asserts **ValidIn***.
5. The data identifier associated with the data cycle must contain a coherent or noncoherent last data cycle indication.
6. After the data cycle is issued, the write request is complete and the external agent sets the **SysCmd** and **SysAD** buses to a tristate, allowing the System interface to return to master state. Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

External write requests are only allowed to write a 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.

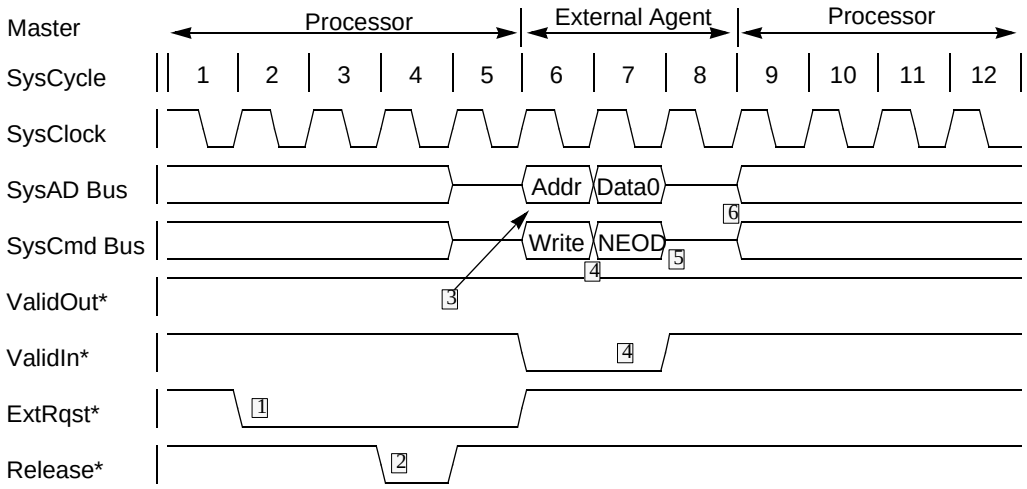


Figure 11.18 External Write Request; System Interface Initially a Bus Master

11.6.4 Read Response Protocol

An external agent must return data to the processor in response to a processor read request by using a read response protocol. A read response protocol consists of the following steps:

1. The external agent waits for the processor to perform an uncompelled change to slave state.
2. The processor returns the data through a single data cycle or a series of data cycles.
3. After the last data cycle is issued, the read response is complete and the external agent sets the **SysCmd** and **SysAD** buses to a tristate.
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, an external agent must return a data block of the correct size regardless of the fact that the data may be in error.

The processor only checks the error bit for the first doubleword of the block. The remaining error bits for the 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 when a read response is presented to it and there is no processor read pending.

Figure 11.19 illustrates a processor word read request followed by a word read response. Figure 11.20 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.*

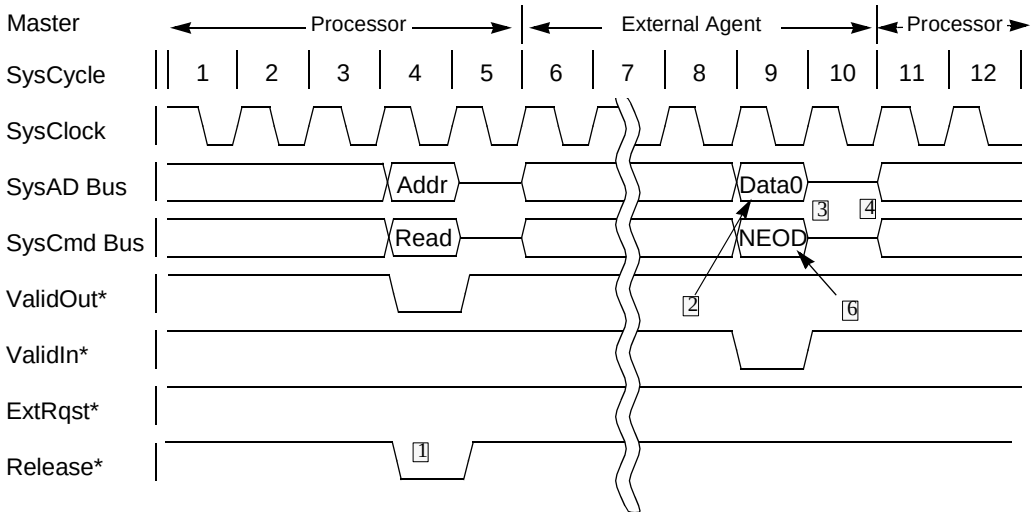


Figure 11.19 Processor Word Read Request, followed by a Word Read Response

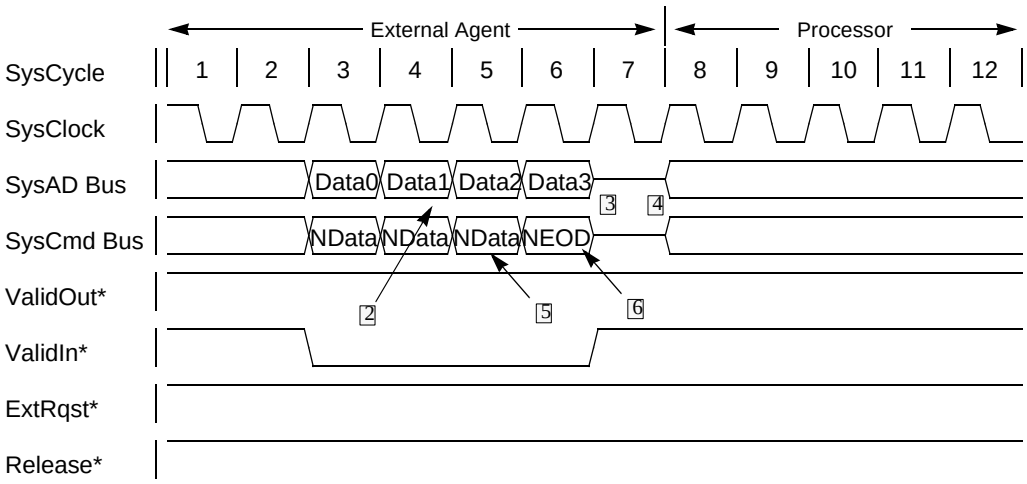


Figure 11.20 Block Read Response, System Interface already in Slave State

11.7 Secondary Cache Read Protocol (RM527x only)

There are three possible scenarios which can occur on a secondary cache access.

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

11.7.1 Secondary Cache Read Hit

Figure 11.21 shows the secondary cache read hit protocol. When a block read request is speculatively issued to both the secondary cache and the external agent, but completed by the secondary cache:

1. The processor issues a block read request and also asserts the **ScTCE***, **ScTDE***, and **ScDCE*** secondary cache control signals. In addition the processor drives the cache index onto **ScLine[15:0]** and the sub-block order doubleword onto **ScWord[1:0]**. Assertion of **ScTCE***, along with **ValidOut*** and **SysCmd**, indicates to the external agent that this is a secondary cache read request. In addition, the assertion of **ScTCE*** initiates a tag RAM probe. The assertion of **ScTDE*** loads the tag portion of the **SysAD** bus into the tag RAM. The **ScValid** signal is asserted to probe for a valid cache tag. The assertion of **ScDCE*** initiates a speculative read of the secondary cache data RAMs.
2. The **ScMatch** signal from the tag RAM is sampled by both the processor and the external agent. Assertion of **ScMatch** indicates a secondary cache tag hit, causing the external agent to abort the memory read. Hence the external cache data RAMs now own **SysAD** and supply the first of a 4 doubleword burst in response to the 4-cycle **ScDCE*** burst. The **SysCmd** bus is not driven during the secondary cache read.
3. Ownership of the **SysAD** bus is returned to the processor.
4. The timing of **ScDCE*** is dependent on the setting of the boot-time mode serial bit[15]. If mode bit[15] =0, **ScDCE*** will negate at time t5 to accommodate Dual Cycle De-select (DCD) SRAMs timing. If mode bit[15] =1, **ScDCE*** will negate at time t6 to accommodate Single Cycle De-select (SCD) SRAM timing.

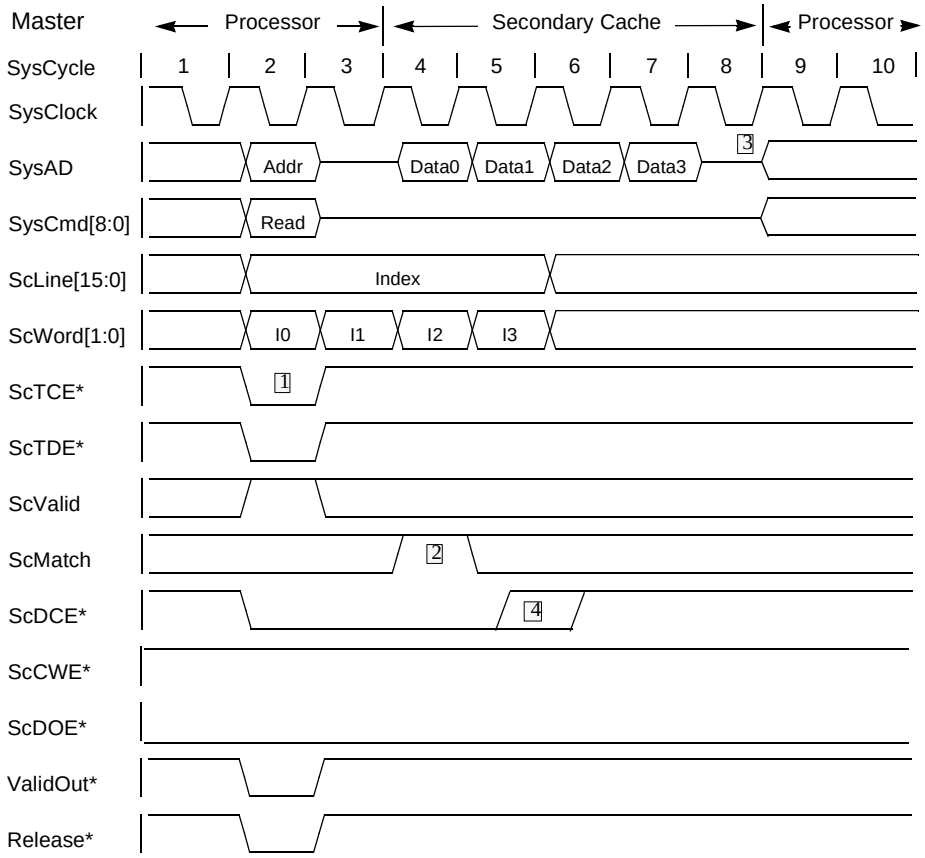


Figure 11.21 Secondary Cache Read Hit

11.7.2 Secondary Cache Read Miss

Figure 11.22 shows the secondary cache read miss protocol when a block read request is speculatively issued to both the secondary cache and the external agent, but is completed by the external agent with a response to both the secondary cache and the processor.

1. The processor issues a block read request and also asserts the **ScTCE***, **ScTDE***, **ScDCE***, and **ScValid** signals and drives the cache index onto **ScLine[15:0]** and **ScWord[1:0]**.
2. The **ScMatch** signal from the tag RAM is sampled by the processor and external agent. Since the signal is negated, indicating a secondary cache miss, the **SysAD** data from the secondary cache is invalid.
3. The external agent negates **ScDOE*** to tristate the data RAM outputs, indicating that it will be supplying the read response. The processor tristates its **ScWord[1:0]** outputs to allow the external agent to drive them during the read response.
4. The processor asserts **ScCWE*** 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 secondary 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 **ScTCE*** to write the tag value stored in the tag RAM data input register two cycles after **ValidIn*** is asserted.
7. The external agent asserts **ScDOE*** to indicate that it will supply the last doubleword of the read response in the next cycle.
8. The processor negates **ScDCE*** two cycles after the next assertion of **ScDOE*** in order to complete the secondary cache line fill.

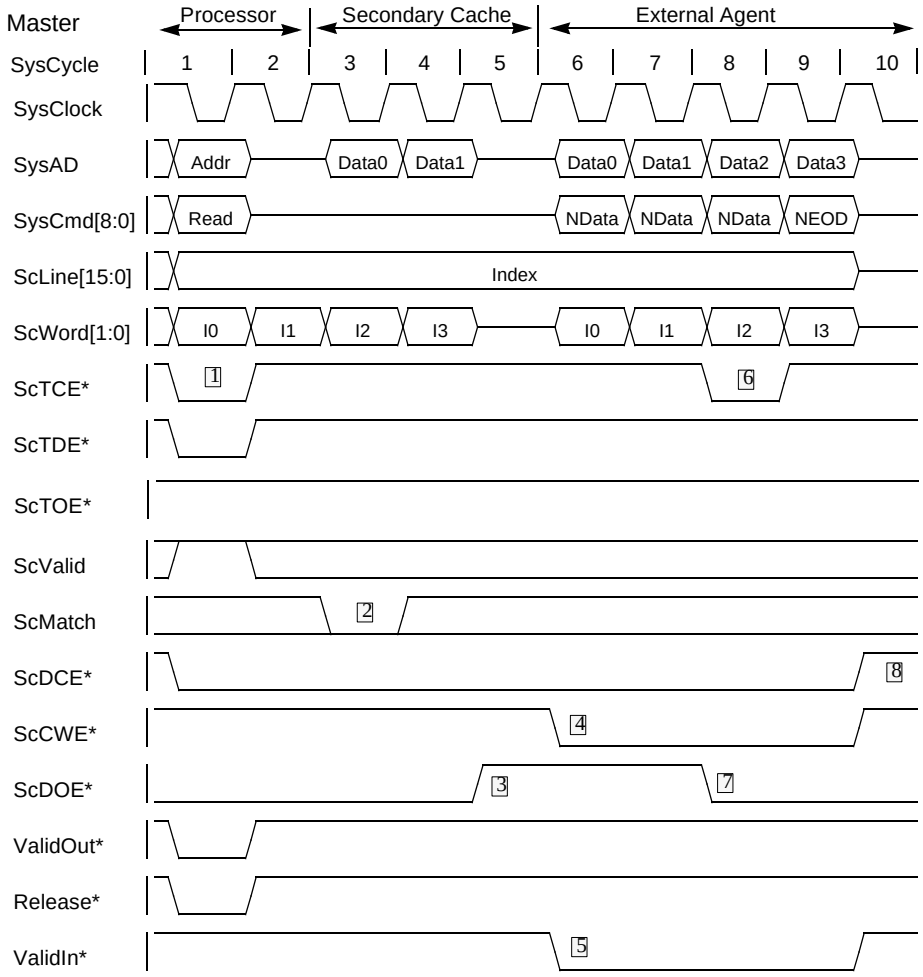


Figure 11.22 Secondary Cache Read Miss

11.7.3 Secondary Cache Read Miss with Bus Error

Figure 11.23 shows a secondary cache read miss with bus error protocol. This protocol is the same as the secondary 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 **ScTCE*** and **ScTDE*** to write the new tag value into the secondary cache tag RAM with **ScValid** negated to invalidate this line.

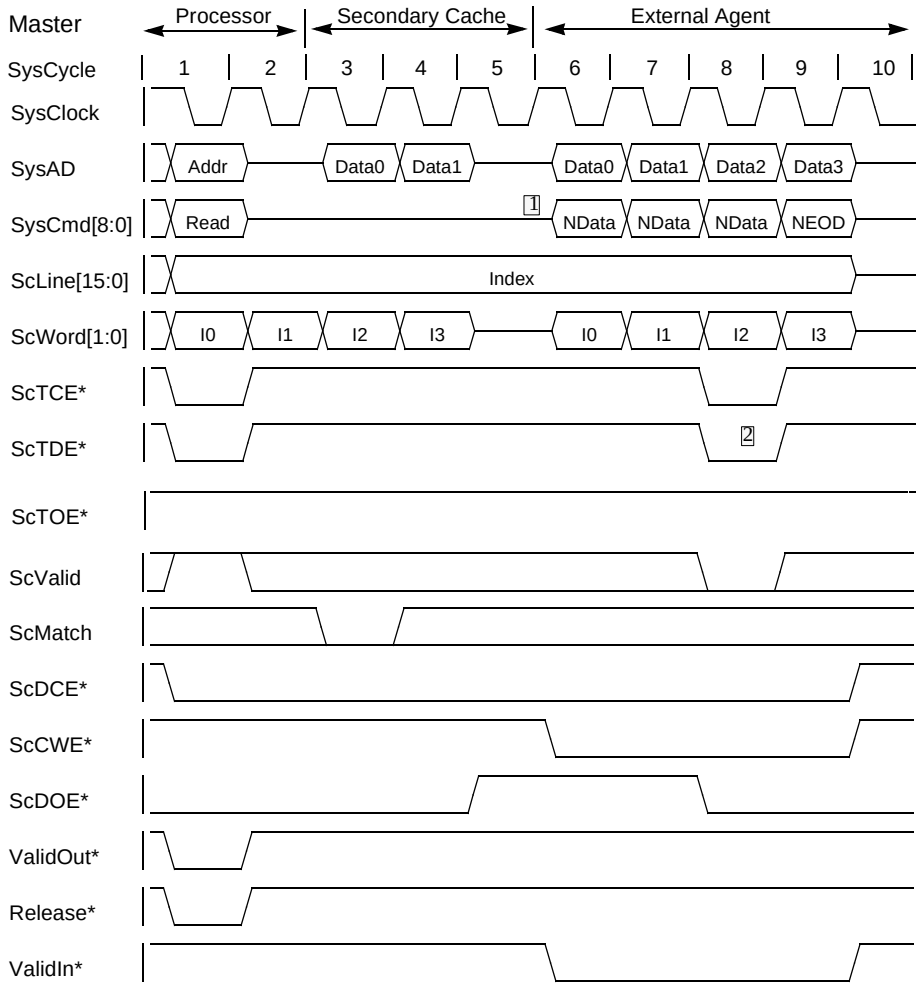


Figure 11.23 Secondary Cache Read Miss with Bus Error

11.8 Secondary Cache Write

Figure 11.24 shows a secondary cache write protocol. For the external agent, this protocol is the same as a non-secondary cache mode block write to the external agent, but the data is also written into the secondary cache.

1. The processor issues a block write and also asserts **ScTCE***, **ScTDE***, and **ScCWE*** in order to write the tag portion of the address on **SysAD** into the secondary cache tag RAM. The processor asserts **ScValid** to set the secondary cache tag to valid.
2. The processor asserts **ScDCE*** to write the block into the secondary cache data RAMs.

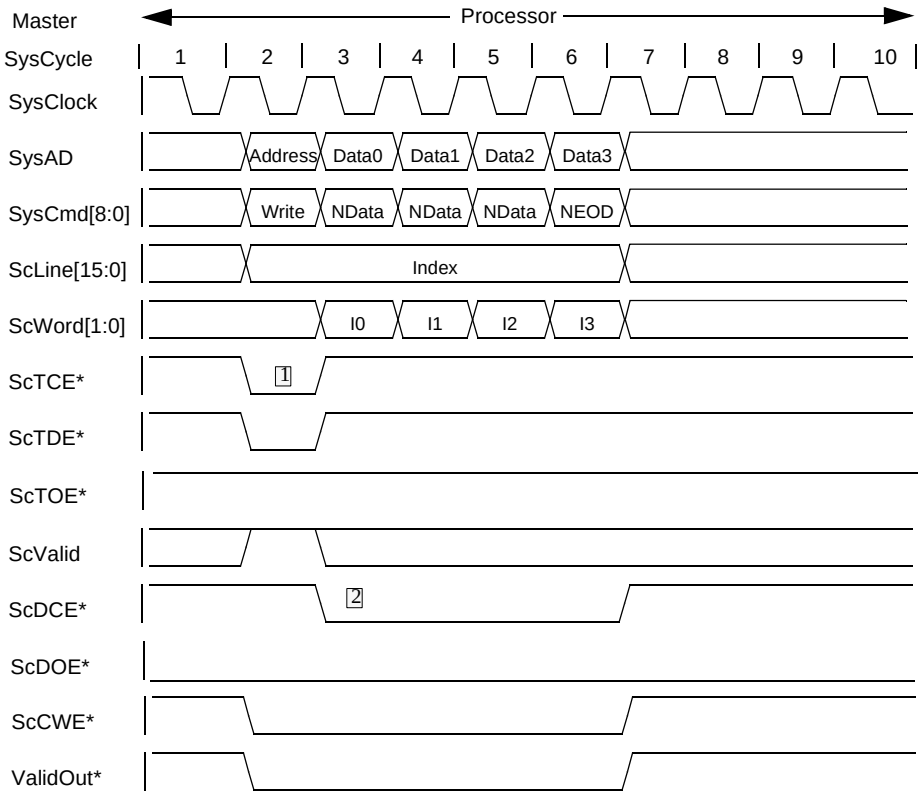


Figure 11.24 Secondary Cache Write Operation

11.9 Secondary Cache Line Invalidate

The RM527x processors have the ability, via the CACHE instruction, to invalidate either a single line or 128 consecutive lines (address aligned) of the secondary cache. The invalidate operation is analogous to writing to the Tag RAM and invalidating the line in question. The **ScTCE***, **ScTDE***, and **ScCWE*** signals are driven active in the same clock as the **SysAD** and **ScLine** busses with **ScValid** negated. Invalidates are the only cache operations which may occur back-to-back. Note that **ValidOut*** is not asserted during secondary cache invalidate operations as the external agent does not participate in secondary cache invalidates.

Figure 11.25 shows the secondary cache invalidate protocol.

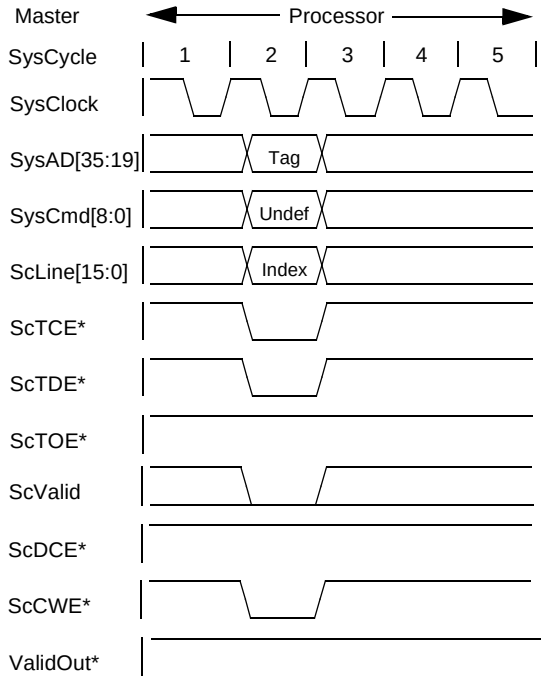


Figure 11.25 Secondary Cache Line Invalidate

The repeat rate for cache line invalidate instructions is two **SysClock** cycles. The repeat rate for cache page invalidate is one **SysClock** per line for 128 consecutive **SysClock** cycles.

11.10 Secondary Cache Probe Protocol

The secondary cache probe operation is analogous to a Tag RAM read operation. The **ScTCE*** and **ScTDE*** signals are asserted in the same clock as system address and the secondary cache line index. The processor then tristates the **SysAD** bus and asserts **ScTOE***. One clock later the tag information is driven onto the **SysAD** bus. **ValidOut*** is not asserted during a secondary cache probe operation as the external agent does not participate in secondary cache probes. The Tag RAM data are driven onto **SysAD[35:19]** and **ScValid**, which are the only **SysAD** signals valid during a probe operation.

A Secondary Cache Probe is initiated by executing an Index Load Tag CACHE instruction on the secondary cache.

Figure 11.26 shows a timing diagram of a secondary cache probe protocol.

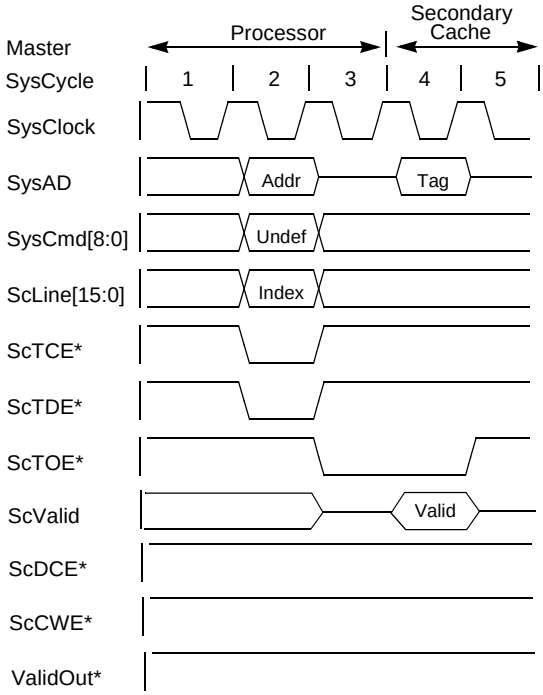


Figure 11.26 Secondary Cache Probe (Tag RAM Read)

11.11 Secondary Cache Flash Clear Protocol

In addition to the line invalidate operation, the RM527x processor also has, via the CACHE instruction, the ability to invalidate the entire secondary 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. 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 **ScTCE***, **ScTDE***, and **ScCWE*** signals need not be asserted. The assertion of **ScCLR*** is all that is necessary for the Tag RAM to perform the requested operation. Figure 11.27 illustrates the secondary cache flash clear protocol.

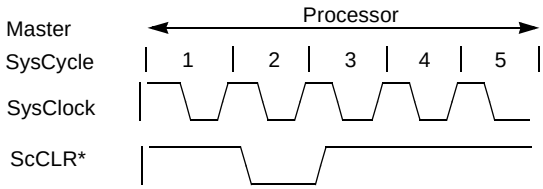


Figure 11.27 Secondary Cache Flash Clear

11.12 SysADC[7:0] Protocol

The following rules apply to the use of **SysADC[7:0]** during a block read response.

- Data is checked on only the first doubleword of the transfer. If the data is erroneous (**SysCmd[5]**=1), the primary cache line is invalidated and a bus error exception is generated.
- A parity error on the first doubleword will be detected as it is issued and will cause a cache parity error exception. The cache line will be valid. Parity errors in subsequent doubles will be detected if and when they are referenced.
- On the following three doublewords: The data erroneous 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 will fill the secondary cache must receive correct parity for all 4 doublewords (**SysCmd[4]**=0) for data going to the secondary cache.
- For a secondary cache mode read hit cycle: The data erroneous bit is ignored. Parity checking is implicitly ON, indicating that the secondary 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 transfer other than the first doubleword does not cause a bus error. Forcing bad parity will generate a cache error if any of the remaining three doublewords of the transfer are referenced.

11.13 Data Rate Control

The System interface supports a maximum data rate of one doubleword per cycle for the RM526x/7x and a maximum data rate of one word per cycle for the RM523x. 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 cycles as valid when **ValidIn*** is asserted and the **SysCmd** bus contains a data identifier; thereafter, the processor continues to accept data until it receives the data word tagged as the last one.

Figure 11.28 shows how the External Agent can introduce wait states by deasserting the **ValidIn*** pin.

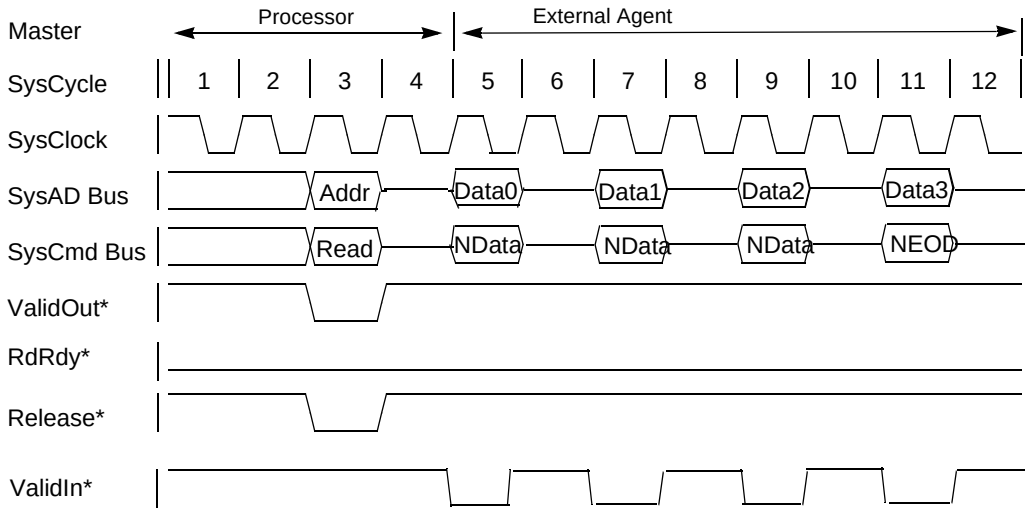


Figure 11.28 External Agent controlling Processor Read Data Flow by deasserting ValidIn*

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

11.14 Write Data Transfer Patterns

A data pattern is a sequence of letters indicating the *data* and *idle* 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 11.2: lists the possible block write data rate patterns, which may be specified at boot time for a 64-bit interface. Table 11.3: is for 32-bit interfaces.

Table 11.2: Block Write Data Rate Patterns for 64bit Bus Interface

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

In Table 11.2:, block write data patterns are specified using the letters **D** and **x**; **D** indicates a double-word-data cycle and **x** indicates an unused cycle.

Table 11.3: Block Write Data Rate Patterns for 32bit Bus Interface

Maximum Data Rate	Data Pattern
1 Double/1 SysClock Cycle	WWWWWWWW
2 Doubles/3 SysClock Cycles	WWxWWxWWxWWx
1 Double/2 SysClock Cycles	WWxxWWxxWWxxWWxx
1 Double/2 SysClock Cycles	WxWxWxWxWxWxWxWx
2 Doubles/5 SysClock Cycles	WWxxxWWxxxWWxxxWWxxx
1 Double/3 SysClock Cycles	WWxxxxWWxxxxWWxxxxWWxxxx
1 Double/3 SysClock Cycles	WxxWxxWxxWxxWxxWxxWxxWxx
1 Double/4 SysClock Cycles	WWxxxxxxWWxxxxxxWWxxxxxxWWxxxxxx
1 Double/4 SysClock Cycles	WxxxWxxxWxxxWxxxWxxxWxxxWxxxWxxx

In Table 11.3:, block write data patterns are specified using the letters **W** and **x**; **W** indicates a word-data cycle and **x** indicates an unused cycle.

Figure 11.29 shows a block write that provides data to the external agent at a data rate of two doublewords every three cycles using the data pattern **DDx**.

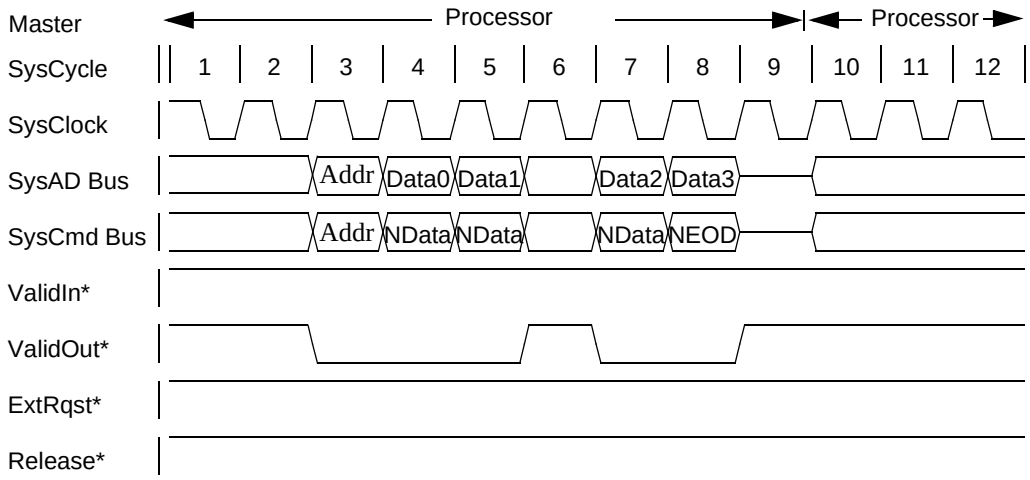


Figure 11.29 Block write at DDxDDx data transfer pattern rate.

11.15 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, to allow 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 coordinate control 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 request mastership of the **SysAD**.
2. The processor does an compelled change to slave state and signals so by asserting **Release*** for one clock cycle.
3. The external agent then performs it's independent transmission on the **SysAD** bus, making sure not to assert **ValidIn*** while the transmission is occurring.
4. When the independent transmission is complete, the external agent must issue a *System interface external request release null* command with **ValidIn*** asserted to return the System interface to master state.

11.16 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 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, ground the **BigEndian** pin. 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.

11.17 System Interface Cycle Time

The processor specifies minimum and maximum cycle counts for various processor transactions and for the processor response time to external requests. Processor requests themselves are constrained by the System interface request protocol, and request cycle counts can be determined by examining the protocol. The following System interface interactions can vary within minimum and maximum cycle counts:

- waiting period for the processor to release the System interface to slave state in response to an external request (*release latency*)
- response time for an external request that requires a response (*external response latency*).

The remainder of this section describes and tabulates the minimum and maximum cycle counts for these System interface interactions.

11.18 Release Latency

Release latency is generally defined as the number of cycles the processor can wait to release the System interface to slave state for an external request. When no processor requests are in progress, internal activity can cause the processor to wait some number of cycles before releasing the System interface. Release latency is therefore more specifically defined as the number of cycles that occur between the assertion of **ExtRqst*** and the assertion of **Release***.

There are three categories of release latency:

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 11.4: 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 11.4: Release Latency for External Requests

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

11.19 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.

11.19.1 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.

11.19.2 System Interface Command Syntax

This section describes the **SysCmd** bus encoding for System interface commands. Figure 11.30 shows a common encoding used for all System interface commands.

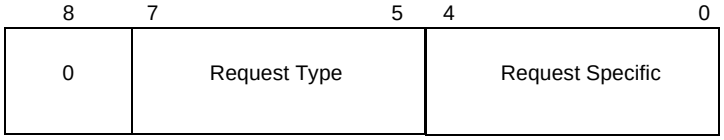


Figure 11.30 System Interface Command Syntax Bit Definition

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 11.5: 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 11.5: 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

11.19.2.1 Read Requests

Figure 11.31 shows the format of a **SysCmd** read request.

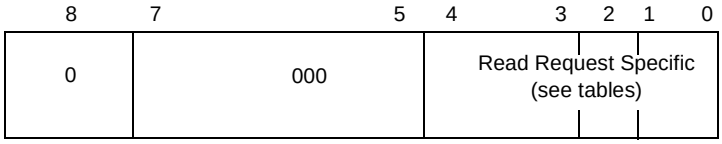


Figure 11.31 Read Request SysCmd Bus Bit Definition

Tables 11.6 through 11.8 list the encodings of **SysCmd(4:0)** for read requests.

Table 11.6: Encoding of SysCmd(4:3) for non-block Read Requests

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

Table 11.7: Encoding of SysCmd(2:0) for Block Read Request

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

Table 11.8: Read Request Data Size Encoding of SysCmd(2:0)

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)

11.19.2.2 Write Requests

Figure 11.32 shows the format of a **SysCmd** write request.

Table 11.9: lists the write attributes encoded in bits **SysCmd(4:3)**. Table 11.10: lists the block write replacement attributes encoded in bits **SysCmd(2:0)**. Table 11.11: lists the write request bit encodings in **SysCmd(2:0)**.

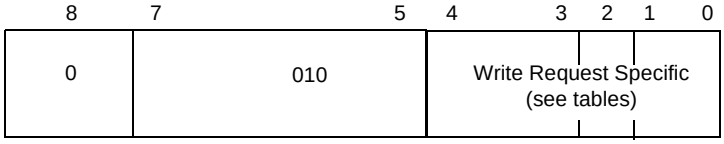


Figure 11.32 Write Request SysCmd Bus Bit Definition

Table 11.9: Write Request Encoding of SysCmd(4:3)

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

Table 11.10:Block Write Request Encoding of SysCmd(2:0)

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

Table 11.11:Write Request Data Size Encoding of SysCmd(2:0)

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)

11.19.2.3 Null Requests

Figure 11.33 shows the format of a **SysCmd** null request.

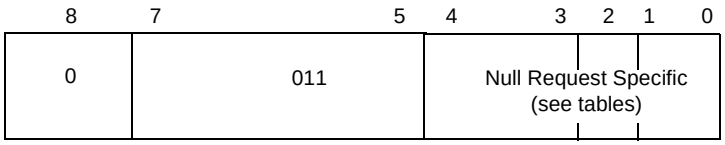


Figure 11.33 Null Request SysCmd Bus Bit Definition

System interface release external null requests use the null request command. Table 11.12: lists the encodings of **SysCmd(4:3)** for external null requests.

SysCmd(2:0) are reserved for null requests.

Table 11.12:External Null Request Encoding of SysCmd(4:3)

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

Table 11.13:SysCmd[8:0] Command Identifier Encoding summary

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	RdHalfword	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	WrHalfword	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 Word (4 bytes)
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 byte (cache line)
0	0	1	1	0	0	X	X	X	NullReq	Null Request Command

11.19.3 System Interface Data Identifier Syntax

This section defines the encoding of the **SysCmd** bus for System interface data identifiers. Figure 11.34 shows a common encoding used for all System interface data identifiers.

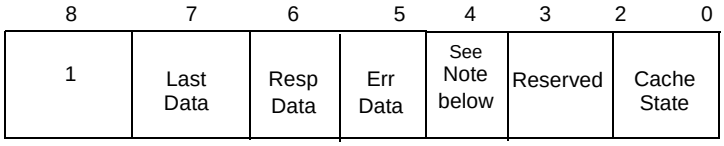


Figure 11.34 Data Identifier SysCmd Bus Bit Definition

SysCmd(8) must be set to 1 for all System interface data identifiers.

*Note: **SysCmd(4)** is reserved for processor data identifier. In an external data identifier, **SysCmd(4)** indicates whether or not to check the data and check bits for parity error. If **SysCmd(4)** is set, the check bits are ignored and the RM5200 generates even parity for the data it receives.*

11.19.3.1 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
- data that is returned in response to an external read request

11.19.3.2 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 of this data element, for both coherent and non-coherent external data identifiers.

SysCmd(3) is reserved for external data identifiers.

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

Table 11.14: lists the encodings of **SysCmd(7:3)** for processor data identifiers. Table 11.15: lists the encodings of **SysCmd(7:3)** for external data identifiers.

Table 11.14: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)	**Data Parity Checking Enable**
0	Check data parity
1	Ignore data parity
SysCmd(3)	**Reserved**

Table 11.15:External Agent 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)	**Data Parity Checking Enable**
0	Check the data and check bits
1	Do not check the data and check bits
SysCmd(3)	**Reserved**

11.20 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]** (RM526x/7x only). The remaining bits of the **SysAD** bus are unused during address cycles.

11.20.1 Addressing Conventions

Addresses associated with doubleword, partial doubleword, word, or partial word transactions and update requests, are aligned for the size of the data element. The system uses the following address conventions:

- Addresses associated with block requests are aligned to double-word boundaries; that is, the low-order 3 bits of address are 0.
- Doubleword requests set the low-order 3 bits of address to 0.
- Word requests set the low-order 2 bits of address to 0.
- Halfword requests set the low-order bit of address to 0.
- Byte, tribyte, quintibyte, sextibyte, and septibyte requests use the byte address.

11.20.2 Subblock Ordering

The order in which data is returned in response to a processor block read request is subblock ordering. In subblock ordering, the processor delivers the address of the requested doubleword 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 that form the block.

During data cycles, the valid byte lines depend upon the position of the data with respect to the aligned doubleword (this may be a byte, halfword, tribyte, quadbyte/word, quintibyte, sextibyte, septibyte, or an octalbyte/doubleword). For example, in little-endian mode, on a byte request where the address modulo 8 is 0, **SysAD(7:0)** are valid during the data cycles. Table 11.16: lists the byte lanes used by the RM526x/7x for partial word transfers for both big and little endian. Table 11.16: lists the byte lanes used by the RM523x for partial word transfers for both big and little endian.

Table 11.16:Partial Word Transfer Byte Lane Usage - RM526x/7x

# Bytes SysCmd[2:0]	Address Mod 8	SysAD byte lanes used (Big Endian)							
		63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
1 (000)	0	X							
	1		X						
	2			X					
	3				X				
	4					X			
	5						X		
	6							X	
	7								X
2 (001)	0	X	X						
	2			X	X				
	4					X	X		
	6							X	X
3 (010)	0	X	X	X					
	1		X	X	X				
	4					X	X	X	
	5						X	X	X
4 (011)	0	X	X	X	X				
	4					X	X	X	X
5 (100)	0	X	X	X	X	X			
	3				X	X	X	X	X
6 (101)	0	X	X	X	X	X	X		
	2			X	X	X	X	X	X
7 (110)	0	X	X	X	X	X	X	X	
	1		X	X	X	X	X	X	X
8 (111)	0	X	X	X	X	X	X	X	X
	Address Mod 8	7:0	15:8	23:16	31:24	39:32	47:40	55:48	63:56
		SysAD byte lanes used (Little Endian)							

Table 11.17:Partial Word Transfer Byte Lane Usage - RM523x

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

11.20.3 Processor Internal Address Map

External reads and writes provide access to processor internal resources that may be of interest to an external agent. The processor decodes bits **SysAD(6:4)** of the address associated with an external read or write request to determine which processor internal resource is the target. However, the processor does not contain any resources that are *readable* through an external read request. Therefore, in response to an external read request the processor returns undefined data and a data identifier with its *Erroneous Data* bit, **SysCmd(5)**, set. The *Interrupt* register is the only processor internal resource available for *write* access by an external request. The *Interrupt* register is accessed by an external write request with an address of 000_2 on bits 6:4 of the **SysAD** bus.

11.21 Error Checking

11.21.1 Parity Error Checking

The RM5200 processors uses only even parity error detection.

Parity is the simplest error detection scheme. By appending a bit to the end of an item of data—called a *parity bit*—single bit errors can be detected; however, these errors cannot be corrected.

There are two types of parity:

- **Odd Parity** adds 1 to any even number of 1s in the data, making the total number of 1s odd (including the parity bit).
- **Even Parity** adds 1 to any odd number of 1s in the data, making the total number of 1s even (including the parity bit).

Odd and even parity 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: it has an even number of 1s. However it is impossible to tell *which* bit is in error.

11.21.2 Error Checking Operation

The processor verifies data correctness by using even parity as it passes data from the System interface to/from the primary caches.

11.21.2.1 System Interface

The processor generates correct check bits for doubleword, word, or partial-word data transmitted to the System interface. As it checks for data correctness, the processor passes data check bits from the primary cache, directly without changing the bits, to the System interface.

The processor does not check data received from the System interface for external writes. By setting the **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 does not contain a data corrector; instead, the processor takes a cache error exception when it detects an error based on data check bits. Software is responsible for error handling.

11.21.2.2 System Interface Command Bus

In the RM5200 processors, 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 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 RM5200 processors.

11.21.2.3 Summary of Error Checking Operations

Error checking operations are summarized in Tables 11.18 and 11.19.

Table 11.18:Error Checking and Generation Summary for Internal Transactions

Bus	Uncached Load	Uncached Store	Primary Cache Load from System Interface	Primary Cache Write to System Interface	Cache Instruction
Processor Data	From system	Not checked	From system interface unchanged	Checked; Trap on error	Check on cache write-back; Trap on error
System Address, Command, and Check bits; Transmit	Not Generated	Not Generated	Not Generated	Not Generated	Not Generated
System Address, Command, and Check Bits; Receive	Not Checked	Not Checked	Not Checked	Not Checked	Not Checked
System Interface Data	Checked, Trap on error	From Processor	Checked on requested doubleword, Trap on error	From primary cache	From primary cache
System Interface Data Check Bits	Checked, Trap on error	Generated	Checked on requested doubleword, Trap on error	From primary cache	From primary cache

Table 11.19:Error Checking and Generation Summary for External Transactions

Bus	External Write
Processor Data	NA
System Address, Command, and Check bits; Transmit	NA
System Address, Command, and Check Bits; Receive	Not Checked
System Interface Data	Not Checked
System Interface Data Check Bits	Not Checked

Section 12 CPU Exception & Interrupt Processing

This section describes the CPU exception processing, including an explanation of exception processing, followed by the format and use of each CPU exception register.

12.1 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.

12.2 Exception Processing Registers

This section describes the CP0 registers that are used in exception processing. Figure 12.1 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 and are discussed in Section 4 on page 17.

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 12.1: are used in exception processing, and are described in the sections that follow.

Table 12.1: CP0 Exception Processing Registers

Register Name	Reg. No.
Context	4
BadVAddr (Bad Virtual Address)	8
Count	9
Compare register	11
Status	12
Cause	13
EPC (Exception Program Counter)	14
XContext	20
ECC	26
CacheErr (Cache Error and Status)	27
ErrorEPC (Error Exception Program Counter)	30

CPU general registers are interlocked and the result of an instruction can normally be used by the next instruction; if the result is not available right away, the processor stalls until it is available.

CP0 registers and the TLB are not interlocked with the pipeline. Data launched to these registers take a few more cycles to reach the registers (See Appendix C, “Instruction Hazards” on page 201). The programmer should keep this in mind and not have their code take action that depends on data being instantly available in these registers.

12.2.1 Context Register (4)

The *Context* register is a read/write register containing the pointer to an entry in the page table entry (PTE) array; this array is an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the 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 12.1 shows the format of the *Context* register; Table 12.2: describes the *Context* register fields.

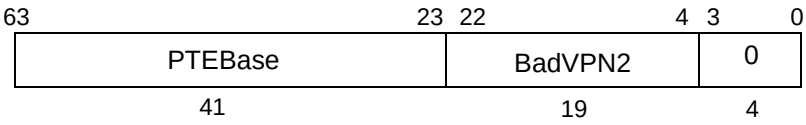


Figure 12.1 Context Register Format

Table 12.2: Context Register Fields

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

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.

12.2.2 Bad Virtual Address Register (BadVAddr) (8)

The *Bad Virtual Address* register (*BadVAddr*) is a read-only register that displays the most recent virtual address that caused one of the following exceptions: TLB Invalid, TLB Modified, TLB Refill, or Address Error.

Figure 12.2 shows the format of the *BadVAddr* register.

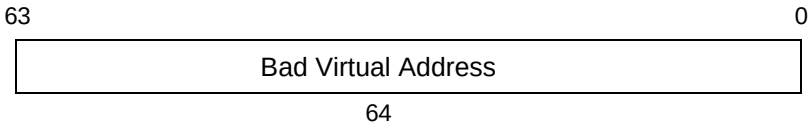


Figure 12.2 BadVAddr Register Format

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

12.2.3 Count Register (9)

The *Count* register acts as a timer incrementing at a constant rate whether or not an instruction is executed, retired, or any forward progress is made through the pipeline.

This register can be read or written. It can be written for diagnostic purposes or system initialization; for example, to synchronize processors.

Figure 12.3 shows the format of the *Count* register.

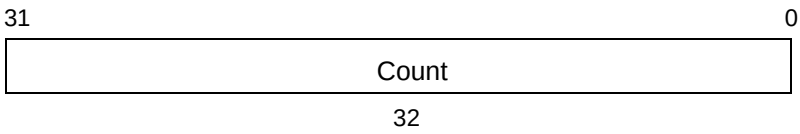


Figure 12.3 Count Register Format

12.2.4 Compare Register (11)

The *Compare* register acts as a timer (see also the *Count* register); it maintains a stable value that does not change on its own.

When the value of the *Count* register equals the value of the *Compare* register, and if the Timer has been selected, via the boot time serial mode bit 11 to be the interupt source instead of the **Int5*** pin, the interrupt bit ***IP(7)*** in the *Cause* register is set. This causes an interrupt provided the global interrupt enable bit (SR.IE) is set and SR.IM(7) is set (both in CP0_Status register)..

Writing a value to the *Compare* register, as a side effect, clears the timer interrupt.

For diagnostic purposes, the *Compare* register is a read/write register. In normal use however, the *Compare* register is write-only. Figure 12.4 shows the format of the *Compare* register.

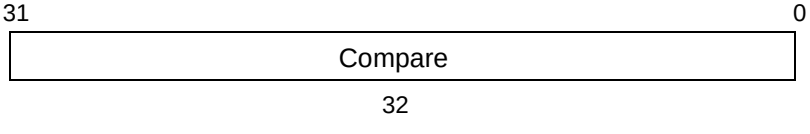


Figure 12.4 Compare Register Format

12.2.5 Status Register (12)

The *Status* register (*SR*) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. 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]**.
- The 3-bit ***Coprocessor Usability*** (***CU***) field controls the usability of 3 of the 4 possible coprocessors originally defined by the MIPS I - III opcode maps (the fourth, ***CU3***, was given up for the MIPS IV implementation, ***XX***, and is no longer available). For the RM5200 *CU0* is the MMU and exception reporting registers. ***CU1*** is the floating point unit. ***CU2*** is reserved. 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.

12.2.5.1 Status Register Format

Figure 12.5 shows the format of the *Status* register. Table 12.3: describes the *Status* register fields. Figure 12.6 and Table 12.4: provide additional information on the ***Diagnostic Status*** (***DS***) field. All bits in the ***DS*** field are readable and writable.

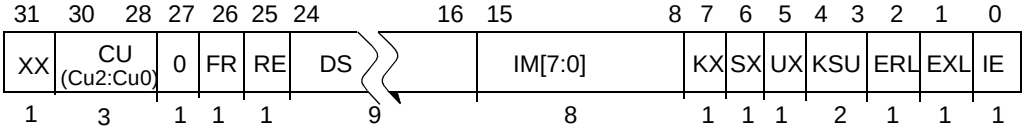


Figure 12.5 Status Register

Table 12.3: Status Register Fields

Field	Description
XX	Enables execution of MIPS IV instructions in user-mode 1: MIPS IV instructions usable 0: MIPS IV instructions unusable
CU	Controls the usability of each of the three coprocessor unit numbers. (see explanation above) 1: usable 0: unusable
0	Reserved. Set to 0.
FR	Enables additional floating-point registers 0: 16 registers 1: 32 registers
RE	*Reverse-Endian* bit, valid in User mode.
DS	*Diagnostic Status* field (see Figure 12.6 and Table 12.4:).
IM[7:0]	*Interrupt Mask*: controls the enabling of each of the external, internal, and software interrupts. An interrupt is taken if interrupts are enabled, and the corresponding bits are set in both the *Interrupt Mask* field of the *Status* register and the *Interrupt Pending* field of the *Cause* register. 0: disabled 1: enabled
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

Field	Description
SX	Enables 64-bit addressing and operations in Supervisor mode. The extended-addressing TLB refill exception is used for TLB misses on supervisor addresses. 0: 32–bit 1: 64–bit
UX	Enables 64-bit addressing and operations in User mode. The extended-addressing TLB refill exception is used for TLB misses on user addresses. 0: 32–bit 1: 64–bit
KSU	Mode bits 10_2: User 01_2: Supervisor 00_2: Kernel
ERL	Error Level; set by the processor when Reset, Soft Reset, NMI, or Cache Error exception are taken. 0: normal 1: error When ERL is set: Interrupts are disabled. The ERET instruction will use 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.
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 will not be updated if another exception is taken.
IE	Interrupt Enable 0: disable interrupts 1: enables interrupts

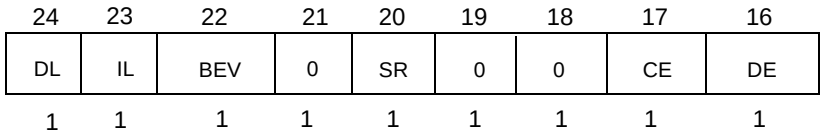


Figure 12.6 Status Register Diagnostics Status (DS) Field

Table 12.4: Status Register Diagnostic Status Bits

Bit	Description
DL	Data Cache Lock 0: Dcache set A normal operation 1: refill into Dcache set A disabled Locks contents of Dcache set A. Does not prevent refills into set A when set A's cache line is invalid. Does not inhibit update of set A on store operations.
IL	Instruction Cache Lock 0: Icache set A normal operation 1: refill into Icache set A disabled Locks contents of Icache set A. Does not prevent refills into set A when set A's cache line is invalid.
BEV	Controls the location of TLB refill and general exception vectors. 0: normal 1: bootstrap
0	Reserved. Must be written as zeroes. Returns zeroes when read.
SR	1: Indicates that a Soft Reset or NMI has occurred.
CE	Contents of the ECC register set or modify the check bits of the caches when CE = 1; see description of the *ECC* register.
DE	Specifies that cache parity or ECC errors cannot cause exceptions. 0: parity/ECC remain enabled 1: disables parity/ECC

12.2.5.2 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 MIPSIII opcodes and translation of 64-bit addresses. The 64-bit operation for User, Kernel and Supervisor modes can be set independently.

- 64-bit addressing for Kernel mode is enabled when ***KX*** = 1. The 64-bit operations are always valid in Kernel mode.
- 64-bit addressing and operations are enabled for Supervisor mode when ***SX*** = 1.
- 64-bit addressing and operations are enabled for User mode when ***UX*** = 1.

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.

12.2.5.3 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 is used to distinguishes between the Reset exception and the Soft Reset exception. When SR = 1 the exception was caused by the **Reset*** pin being asserted. When SR = 0 the exception was caused by the **NMI*** pin being asserted or an external agent, via the external request protocal, setting bit 6 in the Interrupt register..

12.2.6 Cause Register (13)

The 32-bit read/write *Cause* register describes the cause of the most recent exception.

Figure 12.7 shows the fields of this register. Table 12.5: describes the *Cause* register fields. A 5-bit exception code (***ExcCode***) indicates the cause of the most recent exception, as listed in Table 12.6:.

All bits in the *Cause* register, with the exception of the ***IP(1:0)*** and ***IV*** bits, are read-only. ***IP(1:0)*** bits are set to a one to cause a software interrupts. The Cause.IV bit is set to a one to remap interruts to interrupt base +180 instead of interrupt base +200, which is with exceptions. The *Cause* ***IV*** bit is set to zero by a Reset.

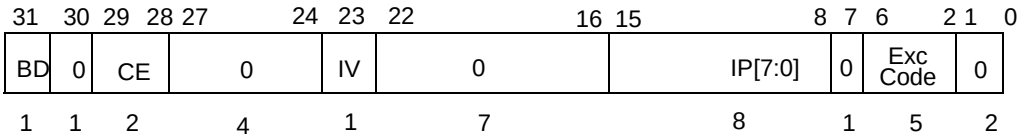


Figure 12.7 Cause Register Format

Table 12.5: Cause Register Fields

Field	Description
BD	Indicates whether the last exception taken occurred in a branch delay slot. 1: delay slot 0: normal
CE	Coprocessor unit number referenced when a Coprocessor Unusable exception is taken.
IV	Enables the new dedicated interrupt vector. 1: interrupts use new exception vector (0x200). 0: interrupts use common exception vector (0x180).
IP[7:0]	Indicates an interrupt is pending. 1: interrupt pending 0: no interrupt
ExcCode	Exception code field (see Table 12.6:)
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 12.6: 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
14	----	Reserved
15	FPE	Floating-Point exception
16-31	----	Reserved

12.2.7 Exception Program Counter (EPC) Register (14)

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

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

- the virtual address of the instruction that was the direct cause of the exception, or
- the virtual address of the immediately preceding branch or jump instruction (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. That is, the EPC register is not updated by an exception within the exception handler.

Figure 12.8 shows the format of the *EPC* register.

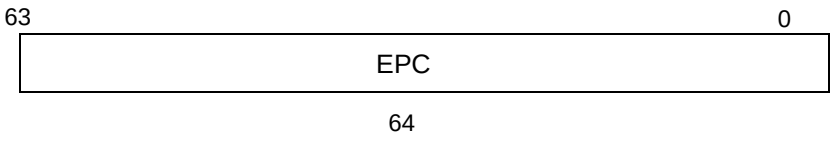


Figure 12.8 EPC Register Format

12.2.8 XContext Register (20)

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

XTLB refill handler, which loads TLB entries for references to a 64-bit address space, and is included solely for operating system use. The operating system sets the ***PTE base*** field in the register, as needed. Normally, the operating system uses the *Context* register to address the current page map, which resides in the kernel-mapped segment *kseg3*. Figure 12.9 shows the format of the *XContext* register; Table 12.7: describes the *XContext* register fields.

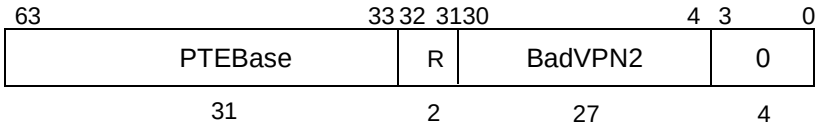


Figure 12.9 XContext Register Format

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

Table 12.7: XContext Register Fields

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

12.2.9 Error Checking and Correcting (ECC) Register (26)

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

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 12.10 shows the format of the *ECC* register; Table 12.8: describes the register fields.

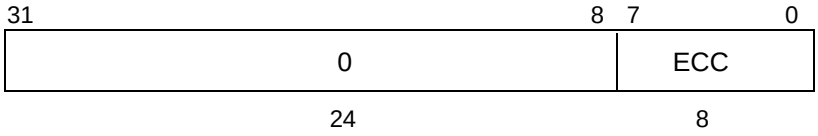


Figure 12.10 ECC Register Format

Table 12.8: ECC Register Fields

Field	Description
ECC	An 8-bit field specifying the parity bits read from or written to a primary cache. ECC field values for Index_Store_Tag_D, Index_Load_Tag_D cache operations: ECC[0] Even parity for least significant byte of requested doubleword ECC[1] Even parity for 2nd least significant byte ECC[2] Even parity for 3rd least significant byte ECC[3] Even parity for 4th least significant byte ECC[4] Even parity for 4th most significant byte ECC[5] Even parity for 3rd most significant byte ECC[6] Even parity for 2nd most significant byte ECC[7] Even parity for most significant byte of requested doubleword ECC field values for Index_Store_Tag_I, Index_Load_Tag_I cache operations: ECC[0] Even parity for least significant word of requested doubleword ECC[1] Even parity for most significant word of requested doubleword
0	Reserved. Must be written as zeroes, and returns zeroes when read.

12.2.10 Cache Error (CacheErr) Register (27)

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

The *CacheErr* register holds cache index and status bits that indicate the source and nature of the error; it is loaded when a Cache Error exception is asserted.

Figure 12.11 shows the format of the *CacheErr* register and Table 12.9: describes the *CacheErr* register fields.

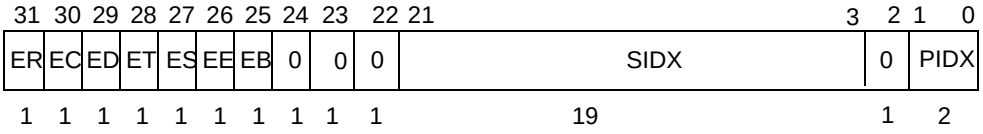


Figure 12.11 CacheErr Register Format

Table 12.9: CacheErr Register Fields

Field	Description
ER	Type of reference 0: instruction 1: data
EC	Cache level of the error 0: primary 1: reserved
ED	Indicates if a data field error occurred 0: no error 1: error
ET	Indicates if a tag field error occurred 0: no error 1: error
ES	Indicates that a parity error occurred in the first double-word of the read response data. 0: no cache miss parity error 1: cache miss parity error
EE	This bit is set if the error occurred on the SysAD bus.

Field	Description
EB	This bit is set if a data error occurred in addition to the instruction error (indicated by the remainder of the bits). If so, this requires flushing the data cache after fixing the instruction error.
SIDX	Physical address [21:3] of the reference that encountered the error
PIDX	Virtual address [13:12] of the double word in error. (used with SIDX to construct a virtual index for the primary caches)
0	Reserved. Must be written as zeroes, and returns zeroes when read.

12.2.11 Error Exception Program Counter 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 non-maskable 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 12.12 shows the format of the *ErrorEPC* register.

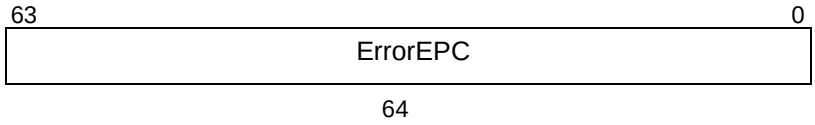


Figure 12.12 ErrorEPC Register Format

12.3 Processor Exceptions

This section describes the processor exceptions—it describes the cause of each exception, its processing by the hardware, and servicing by a handler (software). The types of exception, with exception processing operations, are described in the next section.

12.3.1 Exception Types

This section gives sample exception handler operations for the following exception types:

- reset
- soft reset
- non-maskable 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).

12.3.1.1 Reset Exception Process

Figure 12.13 shows the Reset exception process.

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

Figure 12.13 Reset Exception Processing

12.3.1.2 Cache Error Exception Process

Figure 12.14 shows the Cache Error exception process.

```
T:  ErrorEPC ← PC
    CacheErr ← ER || EC || ED || ET || ES || EE || ED || 0^25
    SR ← SR31:3 || 1 ||SR1:0
    if SR22 = 1 then                                  /* On 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 12.14 Cache Error Exception Processing

12.3.1.3 Soft Reset and NMI Exception Process

Figure 12.15 shows the Soft Reset and NMI exception process.

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

Figure 12.15 Soft Reset and NMI Exception Processing

12.3.1.4 General Exception Process

Figure 12.16 shows the process used for exceptions other than Reset, Soft Reset, NMI, and Cache Error.

```
T:  Cause ← BD || 0 || CE || 0^12 || Cause_15:8 || ExcCode || 0^2
    if SR_1 = 0 then /* System is in User or Supervisor mode with no current exception */
            EPC ← PC
    endif
    SR ← SR_31:2 || 1 || SR_0
    if Cause.IV then
       vector = 0x200
    else
       vector = 0x180
    if SR_22 = 1 then                                    /* On 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 12.16 General Exception Processing

12.3.1.5 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.

Table 12.10: shows the 64-bit-mode vector address for all exceptions when the ***BEV*** bit is set; 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 12.10: shows the 64-bit-mode vector address for all exceptions when the ***BEV*** bit is clear; 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 12.10: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[1]	0xFFFF FFFF BFC0 0300
Interrupt, IV=0	0xFFFF FFFF 8000 0180	0xFFFF FFFF BFC0 0380
Interrupt, IV=1	0xFFFF FFFF 8000 0200	0xFFFF FFFF BFC0 0400
Common	0xFFFF FFFF 8000 0180	0xFFFF FFFF BFC0 0380

Note 1: The Cache Error exception vector uses an uncached address to avoid using the caches in the presence of cache errors.

12.3.1.6 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 RM5200, 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 12.11: lists the TLB refill vector locations, based on the address that caused the TLB miss and its corresponding mode bit.

Table 12.11: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 (SX=0) or XRefill (SX=1)
Kernel	**0xC000 0000 0000 0000 to 0xC000 0FFE FFFF FFFF**	xkseg	XRefill (KX=1)
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)

12.3.1.7 Priority of Exceptions

Table 12.12: describes exceptions in the order of highest to lowest priority. While more than one exception can occur for a single instruction, only the exception with the highest priority is reported.

Table 12.12: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
Integer overflow, Trap, System Call, Breakpoint, Reserved Instruction, Coprocessor Unusable, or Floating-Point Exception
Address error — Data access
TLB refill — Data access
TLB invalid — Data access
TLB modified — Data write
Cache error — Data access
Bus error — Data access
Interrupt *(lowest priority)*

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

12.3.1.8 Reset Exception

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

Processing: The CPU provides a special interrupt vector for this exception:

- location 0xFFFF FFFF BFC0 0000 in 64-bit mode

The Reset vector resides in unmapped and uncached CPU address space, so the hardware need not initialize the TLB or the cache to process this exception. It also means the processor can fetch and execute instructions while the caches and virtual memory are in an undefined state.

The contents of all registers in the CPU are undefined when this exception occurs, except for the following register fields:

- In the *Status* register, ***SR*** is cleared to 0, and ***ERL*** and ***BEV*** are set to 1. 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.

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

12.3.1.9 Soft Reset Exception

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.

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 to 1.

Because the Soft Reset can abort cache and bus operations, the cache and memory states are undefined when the Soft Reset exception occurs.

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

12.3.1.10 Non Maskable Interrupt (NMI) Exception

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

Processing: The Reset vector is used for this exception. The Reset vector is located within uncached and unmapped address space. Hence the cache and TLB need not be initialized in order to process the exception. Regardless of the cause, when this exception occurs the ***SR*** bit of the *Status* register is set, distinguishing this exception from a Reset exception.

Because the NMI can occur in the midst of another exception, it is typically not possible to continue program execution after servicing an NMI. An NMI exception is taken only at instruction boundaries. The state of the caches and memory system are preserved.

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

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

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

12.3.1.11 Address Error Exception

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

- load or store a doubleword that is not aligned on a doubleword boundary
- load, fetch, or store a word that is not aligned on a word boundary
- load or store a halfword that is not aligned on a halfword boundary
- reference the kernel address space from User or Supervisor mode
- reference the supervisor address space from User mode

This exception is not maskable.

Processing: The common exception vector is used for this exception. The ***AdEL*** or ***AdES*** code in the *Cause* register is set, indicating whether the instruction caused the exception with an instruction reference, load operation, or store operation shown by the *EPC* register and ***BD*** bit in the *Cause* register.

When this exception occurs, the *BadVAddr* register retains the virtual address that was not properly aligned or that referenced protected address space. The contents of the ***VPN*** field of the *Context* and *EntryHi* registers are undefined, as are the contents of the *EntryLo* register.

The *EPC* register contains the address of the instruction that caused the exception, unless this instruction is in a branch delay slot. If it is in a branch delay slot, the *EPC* register contains the address of the preceding branch instruction and the ***BD*** bit of the *Cause* register is set as indication.

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

12.3.1.12 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.

TLB Refill Exception

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

Processing: There are two special exception vectors for this exception; one for references to 32-bit address spaces, and one for references to 64-bit address spaces. The ***UX***, ***SX***, and ***KX*** bits of the *Status* register determine whether the user, supervisor or kernel address spaces referenced are 32-bit or 64-bit spaces. All references use these vectors when the ***EXL*** bit is set to 0 in the *Status* register. This exception sets the ***TLBL*** or ***TLBS*** code in the ***ExcCode*** field of the *Cause* register. This code indicates whether the instruction, as shown by the *EPC* register and the ***BD*** bit in the *Cause* register, caused the miss by an instruction reference, load operation, or store operation.

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

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.

TLB Invalid Exception

Cause: The TLB invalid exception occurs when a virtual address reference matches a TLB entry that is marked invalid (TLB valid bit cleared). This exception is not maskable.

Processing: The common exception vector is used for this exception. The ***TLBL*** or ***TLBS*** code in the ***ExcCode*** field of the *Cause* register is set. This indicates whether the instruction, as shown by the *EPC* register and ***BD*** bit in the *Cause* register, caused the miss by an instruction reference, load operation, or store operation.

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers contain the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The *Random* register normally contains a valid location in which to put the replacement TLB entry. The contents of the *EntryLo* register is undefined.

The *EPC* register contains the address of the instruction that caused the exception unless this instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the ***BD*** bit of the *Cause* register is set.

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

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

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

TLB Modified Exception

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

Processing: The common exception vector is used for this exception, and the *Mod* code in the *Cause* register is set.

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers contain the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The contents of the *EntryLo* 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.

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.

12.3.1.13 Cache Error Exception

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

Processing: The processor sets the ***ERL*** bit in the *Status* register, saves the exception restart address in the *ErrorEPC* register, and then transfers the information to 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.

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

12.3.1.14 Bus Error Exception

Cause: A Bus Error exception is raised by board-level circuitry for events such as bus time-out, backplane bus parity errors, and invalid physical memory addresses or access types. This exception is not maskable.

A Bus Error exception occurs when a cache miss refill, uncached reference, or an unbuffered write occurs synchronously; a Bus Error exception resulting from a buffered write transaction must be reported using the general interrupt mechanism.

Processing: The common 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.

The *EPC* register contains the address of the instruction that caused the exception, unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the ***BD*** bit of the *Cause* register is set.

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

- If the ***IBE*** code in the *Cause* register is set (indicating an instruction fetch reference), the 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. The process executing at the time of this exception is handed a bus error signal, which is usually fatal.

12.3.1.15 Integer Overflow Exception

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

Processing: The common exception vector is used for this exception, and the ***OV*** code in the *Cause* register is set.

The *EPC* register contains the address of the instruction that caused the exception unless the instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the ***BD*** bit of the *Cause* register is set.

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

12.3.1.16 Trap Exception

Cause: The Trap exception occurs when a **TGE, TGEU, TLT, TLTU, TEQ, TNE, TGEI, TGEUI, TLTI, TLTUI, TEQI,** or **TNEI** instruction results in a TRUE condition. This exception is not maskable.

Processing: The common exception vector is used for this exception, and the ***Tr*** code in the *Cause* register is set.

The *EPC* register contains the address of the instruction causing the exception unless the instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the ***BD*** bit of the *Cause* register is set.

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

12.3.1.17 System Call Exception

Cause: A System Call exception occurs as the results of executing a **SYSCALL** instruction. This exception is not maskable.

Processing: The common exception vector is used for this exception, and the ***Sys*** code in the *Cause* register is set.

The *EPC* register contains the address of the **SYSCALL** instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

If the **SYSCALL** instruction is in a branch delay slot, the ***BD*** bit of the *Status* register is set; otherwise this bit is cleared.

Servicing: When this exception occurs, control is transferred to the applicable system routine.

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

If a **SYSCALL** instruction is in a branch delay slot, a more complicated algorithm, beyond the scope of this description, may be required.

12.3.1.18 Breakpoint Exception

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

Processing: The common exception vector is used for this exception, and the ***BP*** code in the *Cause* register is set.

The *EPC* register contains the address of the **BREAK** instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

If the BREAK instruction is in a branch delay slot, the ***BD*** bit of the *Status* register is set, otherwise the bit is cleared.

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

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

If a **BREAK** instruction is in a branch delay slot, interpretation of the branch instruction is required to resume execution.

12.3.1.19 Reserved Instruction Exception

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

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

64-bit operations are always valid in Kernel mode regardless of the value of the ***KX*** bit in the *Status* register.

This exception is not maskable.

Processing: The common exception vector is used for this exception, and the ***RI*** code in the *Cause* register is set.

The *EPC* register contains the address of the reserved instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

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

12.3.1.20 Coprocessor Unusable Exception

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

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

This exception is not maskable.

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

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

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

12.3.1.21 Floating-Point Exception

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

Processing: The common exception vector is used for this exception, and the ***FPE*** code in the *Cause* register is set.

The contents of the *Floating-Point Control/Status* register indicate the cause of this exception.

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.

12.3.1.22 Interrupt Exception

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

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

Processing: The common exception vector is used for this exception if ***Cause[IV]***=0, and the ***Int*** code in the *Cause* register is set. If ***Cause[IV]***=1, then the Interrupt exception vector is used.

The ***IP*** field of the *Cause* register indicates current interrupt requests. It is possible that more than one of the bits can be simultaneously set (or even *no* bits may be set) if the interrupt is asserted and then deasserted before this register is read.

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

If the interrupt is hardware-generated, the interrupt condition is cleared by correcting the condition causing the interrupt pin to be asserted.

Due to the on-chip write buffer, a store to an external device may not occur until after other instructions in the pipeline finish. Hence, the user must ensure that the store will occur before the *return from exception* instruction (**ERET**) is executed. Otherwise the interrupt may be serviced again even though there is no actual interrupt pending. Normally an uncached store is used for this purpose. The RM5200 will stall the pipeline until an uncached store has left the chip.

12.3.2 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).

Exceptions other than Reset, Soft Reset, NMI, CacheError or first-level TLB miss
Note: Interrupts can be masked by IE or IMs

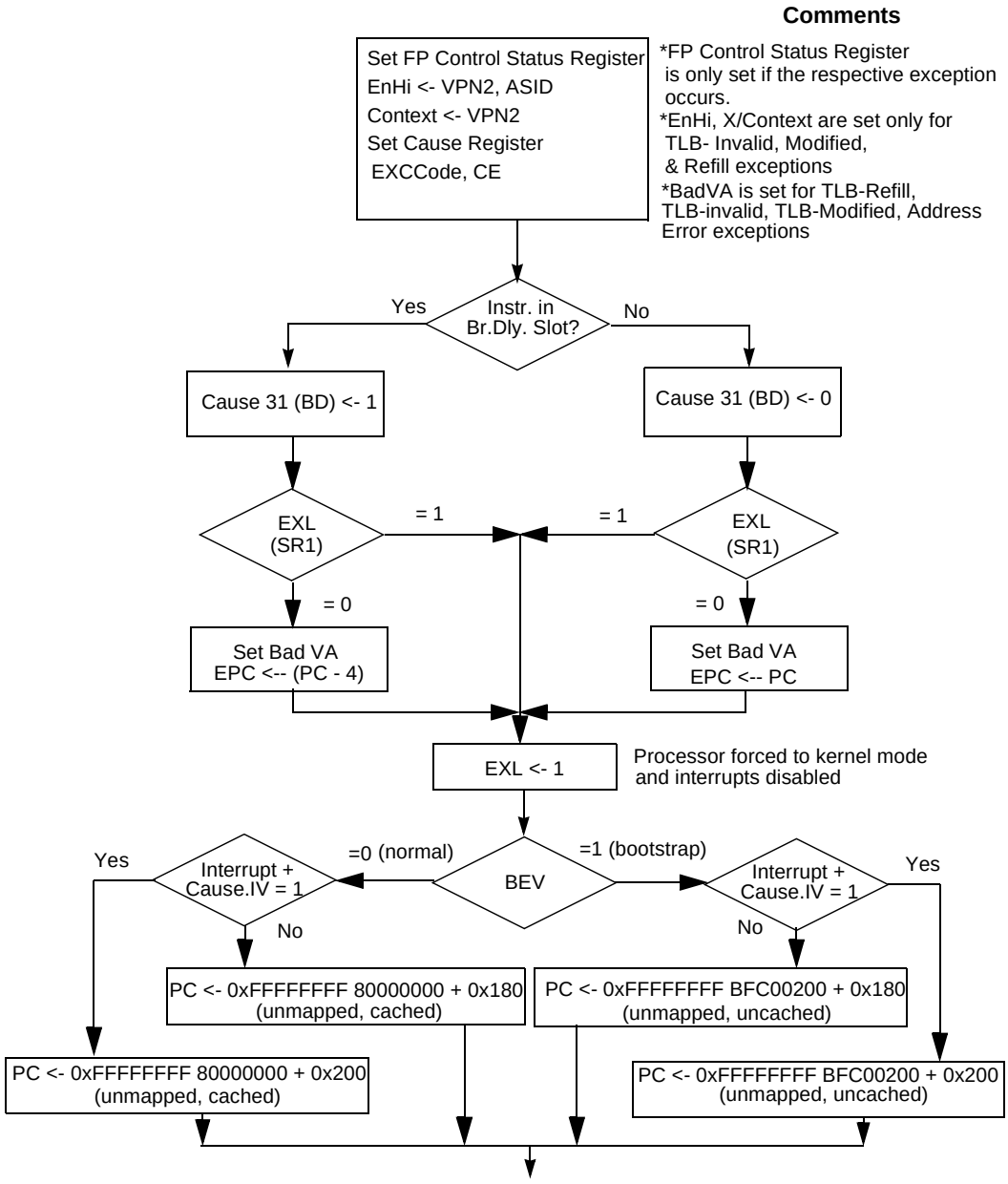


Figure 12.17 General Exception Handler (HW)

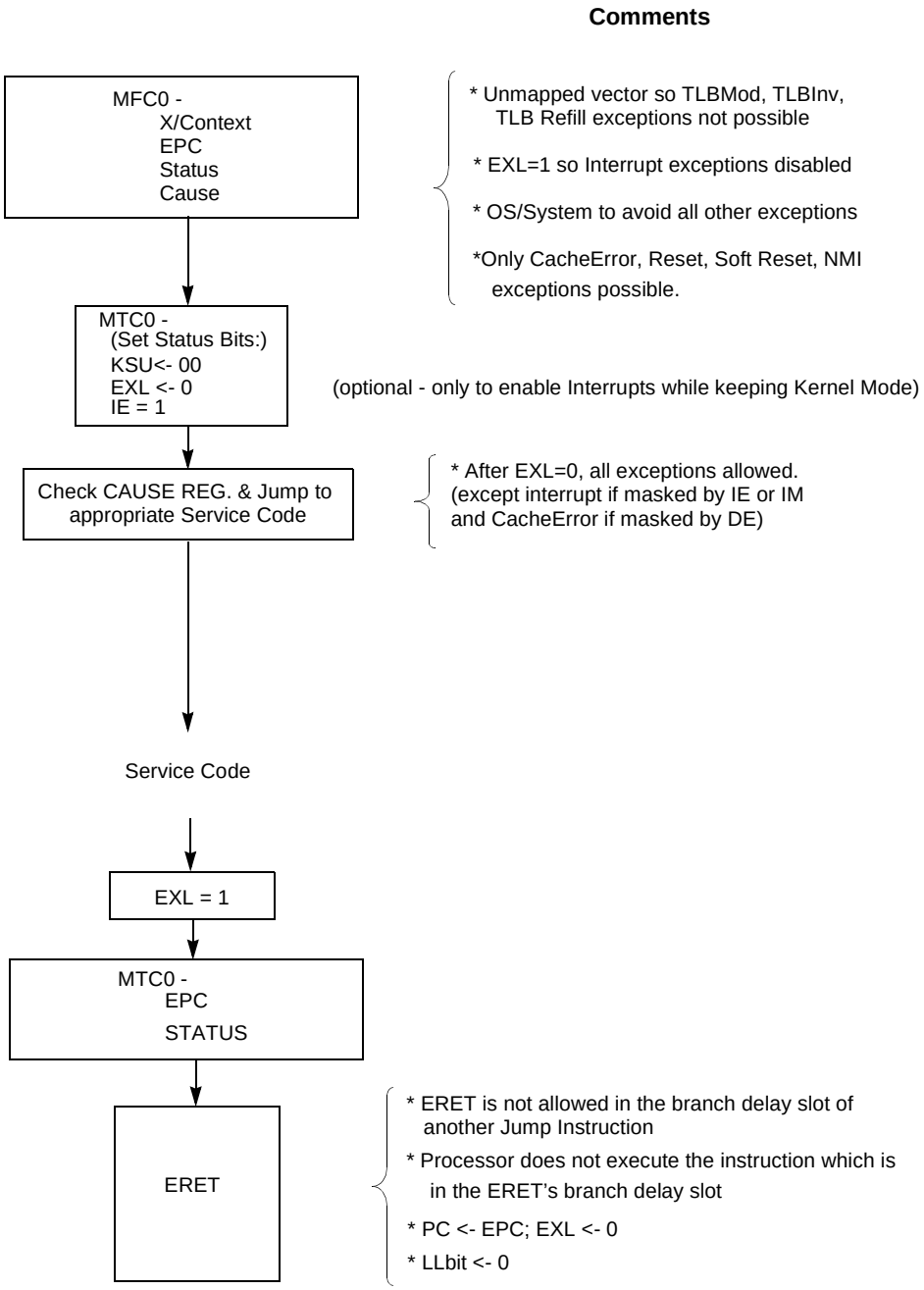


Figure 12.18 General Exception Servicing Guidelines (SW)

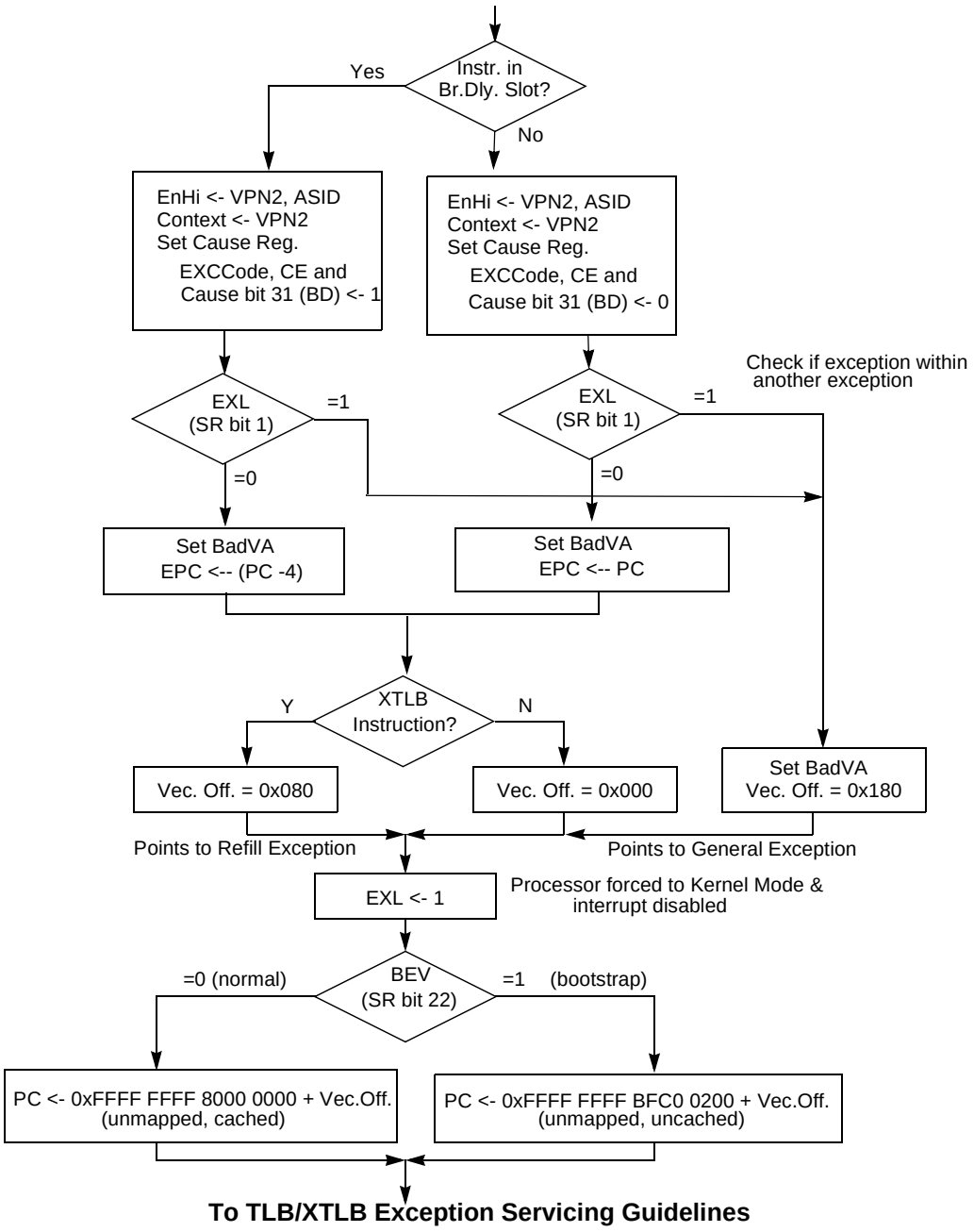


Figure 12.19 TLB/XTLB Miss Exception Handler (HW)

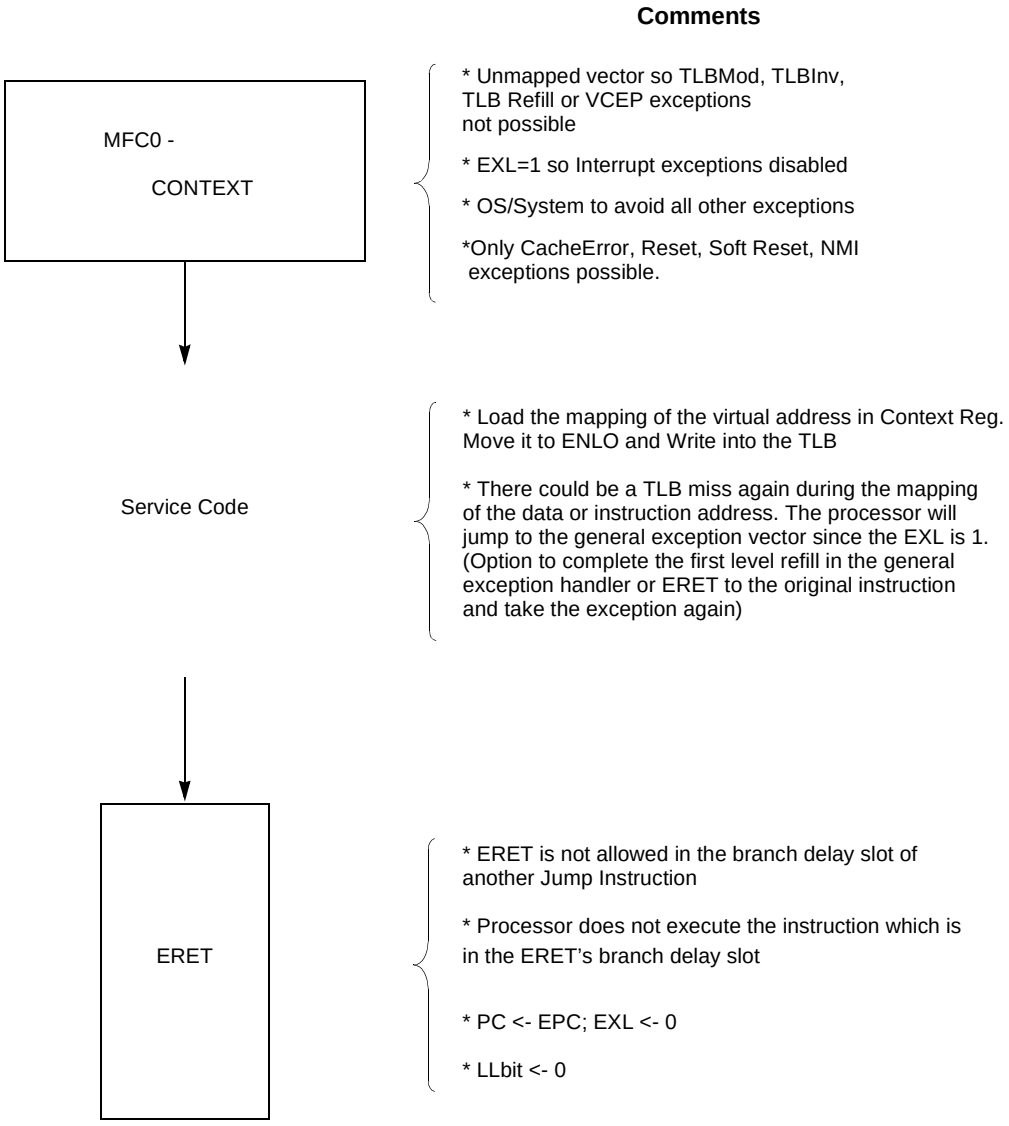


Figure 12.20 TLB/XTLB Exception Servicing Guidelines (SW)

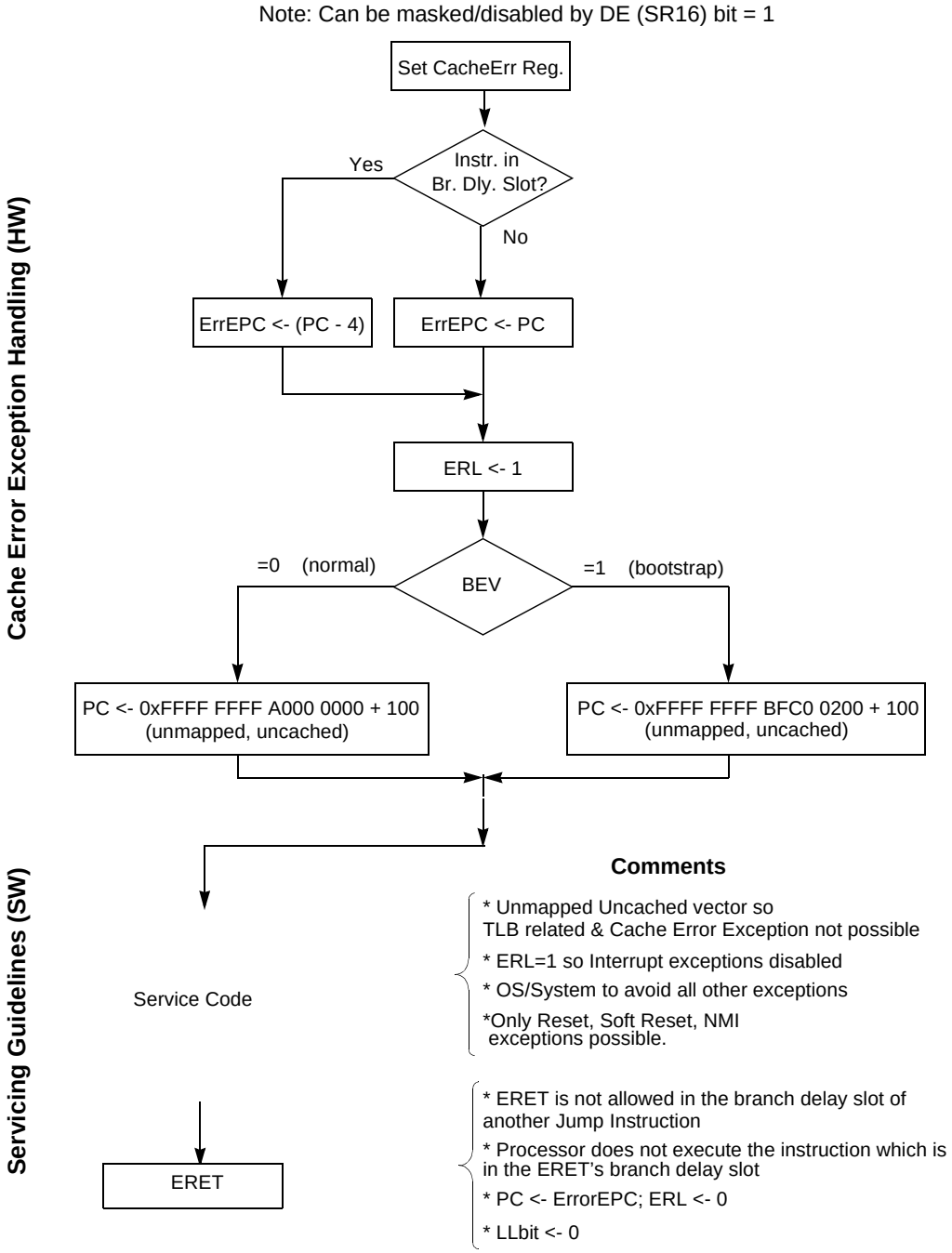


Figure 12.21 Cache Error Exception Handling (HW) and Servicing Guidelines

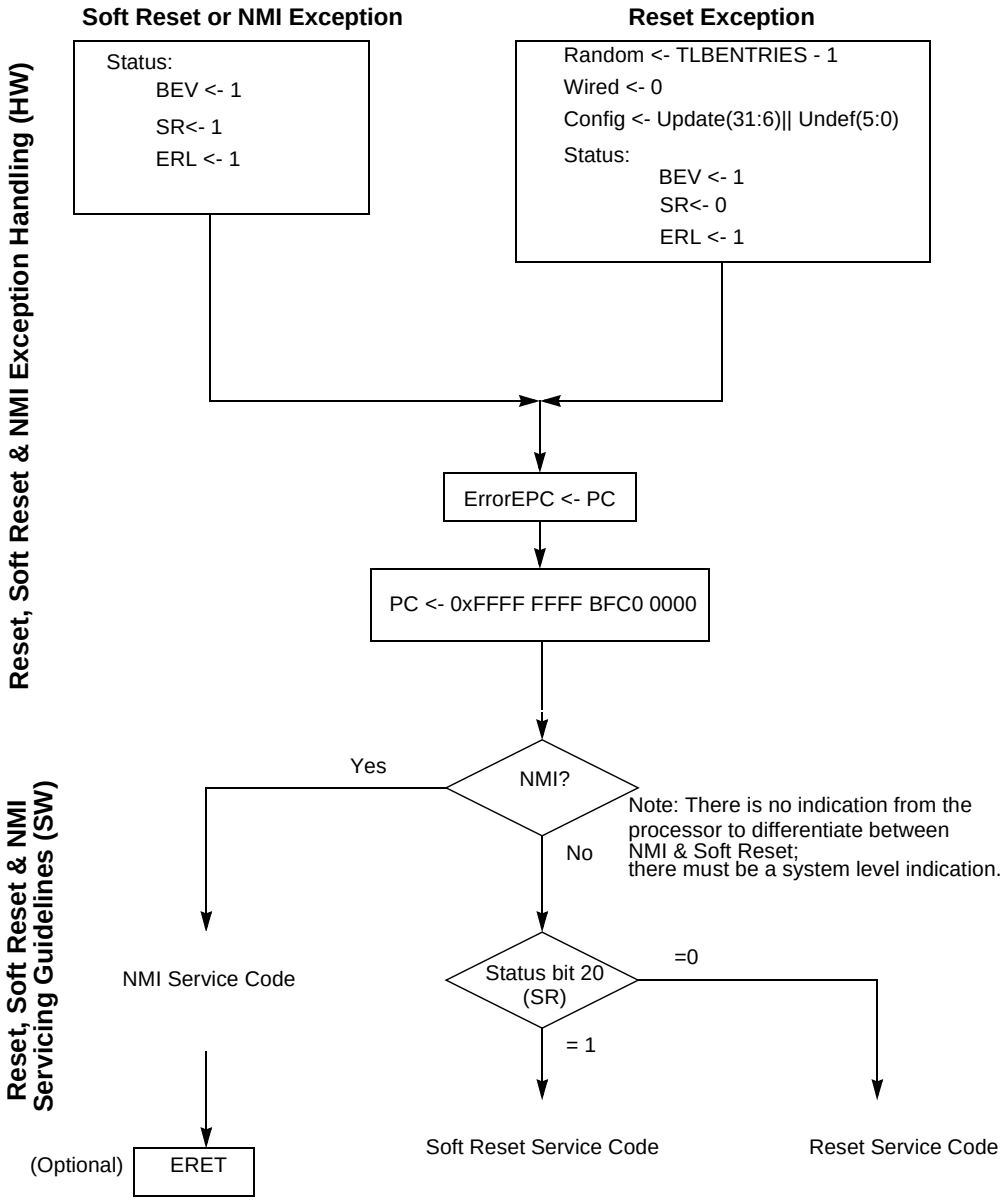


Figure 12.22 Reset, Soft Reset & NMI Exception Handling

12.4 Interrupts

The RM5200 processors support the following interrupts: six hardware interrupts, one internal “timer interrupt,” two software interrupts, and one nonmaskable interrupt. The processor takes an exception on any interrupt.

12.4.1 Hardware Interrupts

The six CPU hardware interrupts can be caused by either an external request write to the interrupt register, or by asserting dedicated interrupt pins. These pins are latched into an internal register by the rising edge of **SysClock**.

12.4.2 Nonmaskable Interrupt (NMI)

The nonmaskable interrupt is caused either by an external request write to bit 6 of the interrupt register or by asserting a the **NMI*** pin. This pin is latched into an internal register by the rising edge of **SysClock**.

12.4.3 Asserting Interrupts

External writes to the CPU are directed to various internal resources, based on an internal address map of the processor. When **SysAD[6:4]** = 0, an external write to any address writes to an architecturally transparent register called the *Interrupt* register; this register is available for external write cycles, but not for external reads.

During a data cycle, **SysAD[22:16]** are the write enables for the seven individual *Interrupt* register bits and **SysAD[6:0]** are the values to be written into these bits. This allows any subset of the *Interrupt* register to be set or cleared with a single write request. Figure 12.23 shows the mechanics of an external write to the *Interrupt* register.

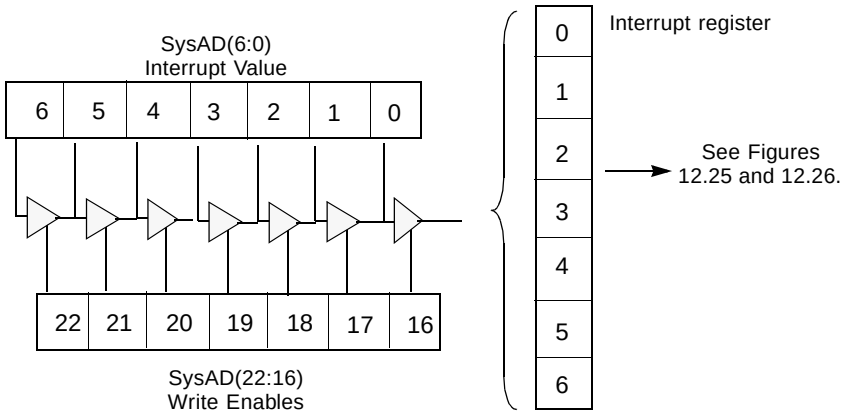


Figure 12.23 Interrupt Register Bits and Enables

Figure 12.24 shows how the RM5200 interrupts are readable through the Cause register.

- Bit 5 of the *Interrupt* register is ORed with the **Int5*** pin and then multiplexed with the *Timer Interrupt* signal. The result is directly readable as bit 15 of the *Cause* register. The selection betwee I**nt5*** and the *Timer Interrupt* is selected by boot time serial mode bit 11.
- Bits 4:0 of the *Interrupt* register are bit-wise ORed with the current value of interrupt pins **Int*[4:0]**. The result is directly readable as bits 14:10 of the *Cause* register.

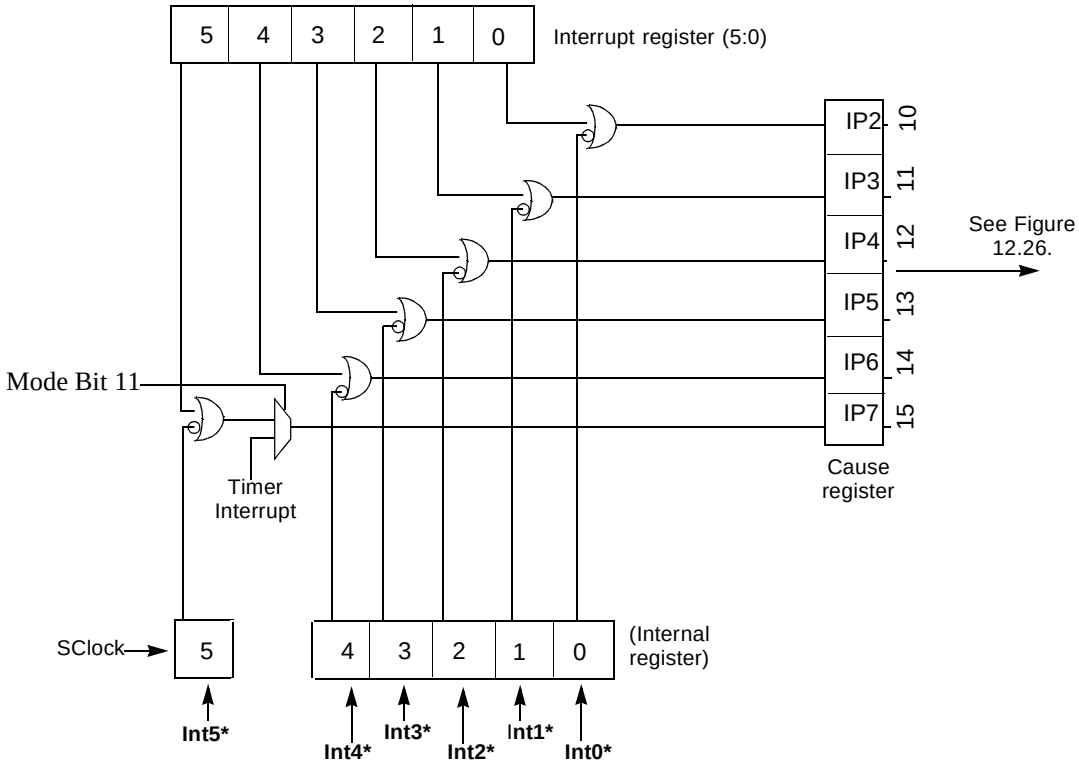


Figure 12.24 RM5200 Interrupt Signals

Figure 12.25 shows the internal derivation of the **NMI** signal for the RM5200 processor.

The **NMI*** pin is latched by the rising edge of **SysClock**. Bit 6 of the *Interrupt* register is then ORed with the inverted value of **NMI*** to form the nonmaskable interrupt. Only the falling edge of the latched signal will cause the NMI.

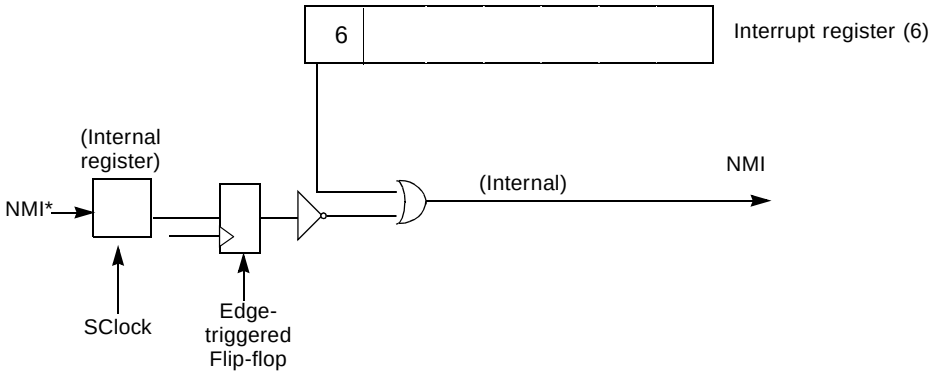


Figure 12.25 RM5200 Nonmaskable Interrupt Signal

Figure 12.26 shows the masking of the RM5200 interrupt signal.

- *Cause* register bits 15:8 (IP7-IP0) are AND-ORed with *Status* register interrupt mask bits 15:8 (IM7-IM0) to mask individual interrupts.
- *Status* register bit 0 is a global Interrupt Enable (IE). It is ANDed with the output of the AND-OR logic to produce the RM5200 interrupt signal.

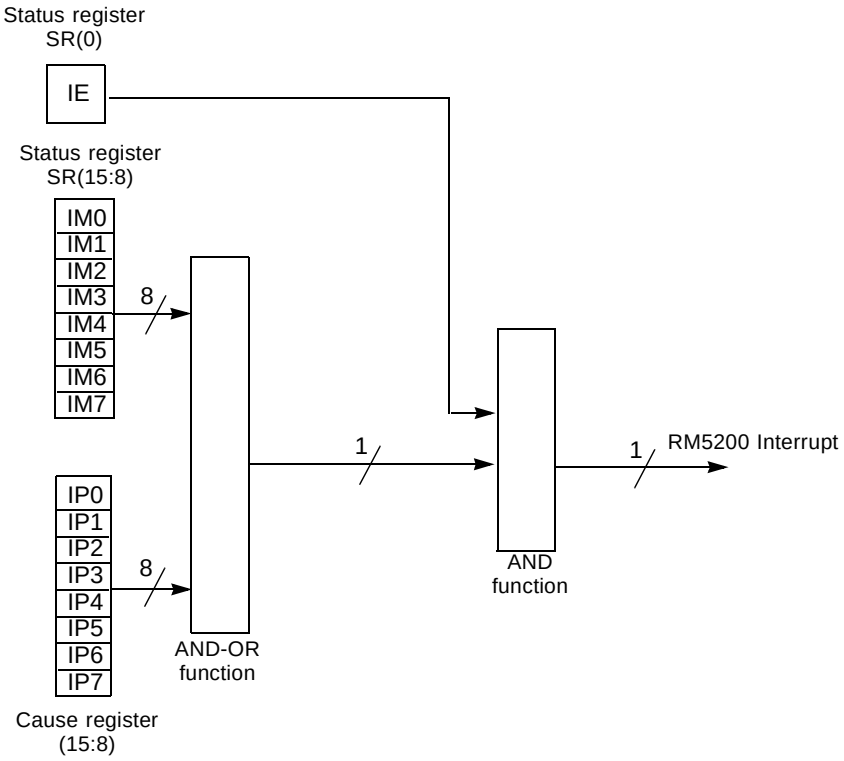


Figure 12.26 Masking of the RM5200 Interrupt

Source		Registered		External Request Write	
Name	**Generated by**	**Interrupt Mask (IM)**	**Interrupt Pending (IP)**	**Set/Clear**	**Enable**
Reset	**ColdReset***				
NMI	**NMI***			**SysAD15**	**SysAD31**
Soft Reset	**Reset***				
SWINT0	SW setting Cause.IP0	Status8	Cause8		
SWINT1	SW setting Cause.IP1	Status9	Causes9		
INT0	**Int0***	Status10	Cause10	**SysAD0**	**SysAD16**
INT1	**Int1***	Status11	Cause11	**SysAD1**	**SysAD17**
INT2	**Int2***	Status12	Cause12	**SysAD2**	**SysAD18**
INT3	**Int3***	Status13	Cause13	**SysAD3**	**SysAD19**
INT4	**Int4***	Status14	Cause14	**SysAD4**	**SysAD20**
INT5	**Int5*/Timer**	Status15	Cause15	**SysAD5**	**SysAD21**

Section 13 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.

13.1 Exception Types

The FP *Control/Status* register described in section 6 contains an ***Enable*** bit for each exception type; exception ***Enable*** bits determine whether an exception will cause the FPU to initiate a trap or set a status flag.

- If a trap is taken, the FPU remains in the state found at the beginning of the operation and a software exception handling routine executes.
- If no trap is taken, an appropriate value is written into the FPU destination register and execution continues.

The FPU supports the five IEEE Standard 754 exceptions:

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

Cause bits, ***Enables***, and ***Flag*** bits (status flags) are used.

The FPU adds a sixth exception type, Unimplemented Operation (E), to use when the FPU cannot implement the standard MIPS floating-point architecture, including cases in which the FPU cannot determine the correct exception behavior. This exception indicates the use of a software implementation. The Unimplemented Operation exception has no ***Enable*** or ***Flag*** bit; whenever this exception occurs, an unimplemented exception trap is taken.

Figure 13.1 illustrates the *Control/Status* register bits that support exceptions.

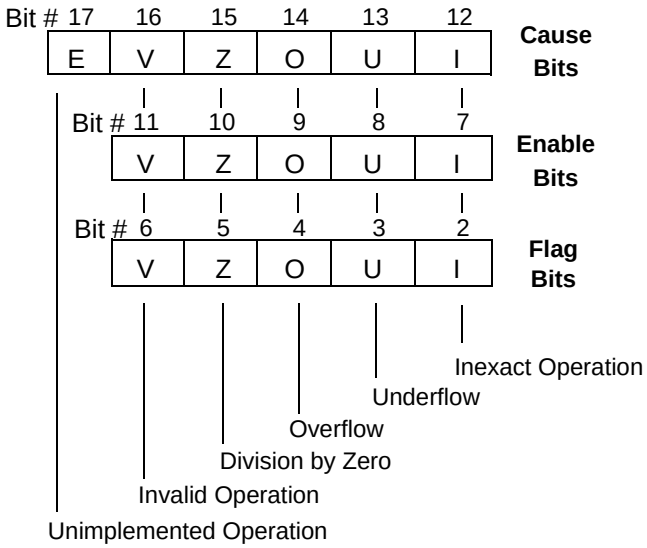


Figure 13.1 Control/Status Register Exception/Flag/Trap/Enable Bits

Each of the five IEEE Standard 754 exceptions (V, Z, O, U, I) is associated with a trap under user control, and is enabled by setting one of the five ***Enable*** bits. When an exception occurs, 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.

13.2 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.

13.3 Flags

A *Flag* bit is provided for each IEEE exception. This ***Flag*** bit is set to a 1 on the assertion of its corresponding exception, with no corresponding exception trap signaled.

The ***Flag*** bit is reset by writing a new value into the *Status* register; flags can be saved and restored by software either individually or as a group.

When no exception trap is signaled, 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 13.1: lists the default action taken by the FPU for each of the IEEE exceptions.

Table 13.1: 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 + ∞
		RM	Modify positive overflows to the format's largest finite number; modify negative overflows to – ∞
Z	Division by zero	Any	Supply a properly signed ∞
V	Invalid operation	Any	Supply a quiet Not a Number (NaN)

Table 13.2: lists the exception-causing situations and contrasts the behavior of the FPU with the requirements of the IEEE Standard 754.

Table 13.2: 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	E	E	Source out of integer range
Signaling NaN source	V	V	V	
Invalid operation	V	V	V	0/0, etc.
Exponent underflow	U	E	E	Normalized exponent < E_{min}
Denormalized or QNaN	None	E	E	Denormalized is (exponent = E_{min}-1 and mantissa <> 0)

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

13.4 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.

13.4.1 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.

13.4.2 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 one or both operands is a signaling NaN.
- Square root: $\sqrt{x}$, where x is less than zero

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

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.

13.4.3 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.

13.4.4 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 (as listed in Table 13.1:).

13.4.5 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 13.1:).

Unimplemented Instruction Exception (E)

Any attempt to execute an instruction with an operation code or format code that has been reserved for future definition sets the ***Unimplemented*** bit in the ***Cause*** field in the FPU *Control/Status* register and traps. The operand and destination registers remain undisturbed and the instruction is emulated in software. Any of the IEEE Standard 754 exceptions can arise from the emulated operation, and these exceptions in turn are simulated.

The Unimplemented Instruction exception can also be signaled when unusual operands or result conditions are detected that the implemented hardware cannot handle properly. These include:

- Denormalized operand, except for Compare instruction
- Quiet Not a Number operand, except for Compare instruction

- Denormalized result or Underflow, when either Underflow or Inexact *Enable* bits are set or the *FS* bit is not set.
- Reserved opcodes
- Unimplemented formats
- Operations which are invalid for their format (for instance, CVT.S.S)

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 RM5200 additional causes of the unimplemented exception include:

- If the multiply portion of the MADD, MSUB, NMADD, 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.

13.5 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.

13.6 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.

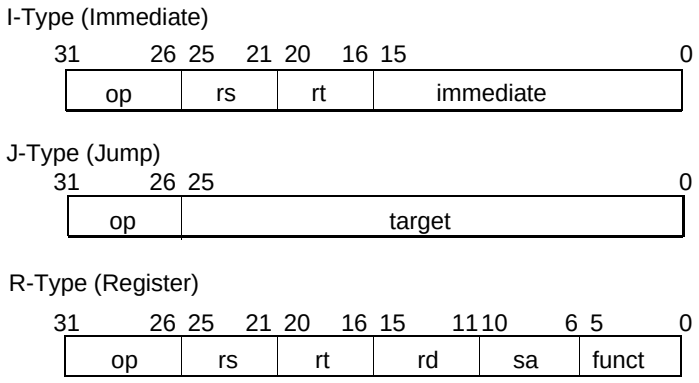
The IEEE Standard 754 recommends that, if enabled, the overflow and underflow traps take precedence over a separate inexact trap. This prioritization is accomplished in software; hardware sets both bits.

Section 14 CPU Instruction Set Summary

The RM523x/RM526x/RM527x processors (hereafter referred to as RM5200) execute the MIPS IV Instruction Set Architecture (ISA), a superset of the MIPS III ISA, and provide backward compatibility to MIPS I, II and III. Each CPU instruction consists of a single 32-bit word, aligned on a word boundary. There are three instruction formats—immediate (I-type), jump (J-type), and register (R-type). The use of a small number of instruction formats simplifies instruction decoding, allowing the compiler to synthesize more complicated (and less frequently used) operations and addressing modes from these three formats as needed.

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 instructions as shown in Figure 14.1.



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 14.1 CPU Instruction Formats

In the MIPS architecture, coprocessor instructions are implementation-dependent.

14.1 Implementation Specific CP0 Instructions

Some of the RM5200 instructions are implementation specific and therefore are not part of the MIPS IV Instruction Set. These are coprocessor instructions that perform operations in their respective coprocessors. Coprocessor loads and stores are I-type, and coprocessor computational instructions have coprocessor-dependent formats.

Table 14.1: lists the CP0 instructions for the RM5200 processors. CP0 instructions are those which are not architecturally visible and are used by the kernel.

Table 14.1: RM5200 CP0 Instructions

COP0 Instruction	Definition
CACHE	Cache Management
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

14.1.1 CACHE - Cache Management

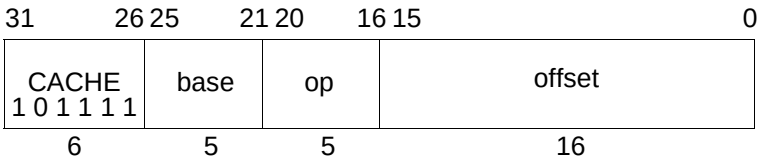


Figure 14.2 RM5200 CACHE Instruction Format

14.1.1.1 Format:

```
CACHE   op, offset(base)
```

14.1.1.2 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 CP0 is not usable (User or Supervisor mode), the CP0 enable bit in the *Status* register is clear, then a coprocessor unusable exception is taken. The operation of this instruction on any operation/cache combination not listed below, or on a secondary 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 caches on the RM5200 are 2-way set associative, which implies that they are divided into halves. Each half is a set.

For a 16Kbyte primary cache divided into two 8Kbyte sets with 32 bytes per tag, $vAddr_{12:5}$ specifies a block of 256 cache lines per set. $VAddr_{13}$ selects between the two sets within the cache. This is the primary cache size on the RM52x0.

For a 32Kbyte primary cache divided into two 16Kbyte sets with 32 bytes per tag, $vAddr_{13:5}$ specifies a block of 512 cache lines per set. $VAddr_{14}$ selects between the two sets within the cache. This is the primary cache size on the RM52x1.

For a secondary cache of $2^{CACHEBITS}$ bytes with 32 bytes per tag, $pAddr_{CACHEBITS...5}$ specifies the block. The secondary cache is direct-mapped, so there is no need to specify the cache set to operate on.

Index Load Tag also uses $vAddr_{LINEBITS...3}$ 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 $vAddr_{LINEBITS...3}$ 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.

Write back from a primary cache goes to the secondary cache and to memory. If no secondary cache is present, the data goes to memory. Data comes from the primary data cache, if present, and is modified (the ***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 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	--	Reserved
3	S	Secondary Cache

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 back to the secondary cache (if present) and to memory. The address to write is taken from the primary cache tag. Set the cache state of primary cache block to Invalid.
0	S	Flash Invalidate	Flash Invalidate the entire secondary cache in one operation for tag RAMs which support this function. (Asserts the ScCLR* pin)
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.
2	I, D	Index Store Tag	Write the tag for the cache block at the specified index from the *TagLo* and *TagHi* CP0 registers.
2	S	Index Store Tag	Write the valid bit from the TagLo CP0 register for the cache block at the specified index from the effective address generated by the CACHE instruction.

Code	Caches	Name	Operation
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.
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.
5	S	Page Invalidate	The processor performs a burst of 128 *Index_Store_Tag* operations to the secondary cache at the page specified by the effective address generated by the CACHE instruction, which must be page-aligned. To invalidate the page, the state bits within the *TagLo* register must be zero. Interrupts are deferred during this operation.
5	I	Fill	Fill the primary instruction cache block from secondary cache or memory.
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.
6	I	Hit Writeback	If the cache block contains the specified address, data is written back unconditionally.

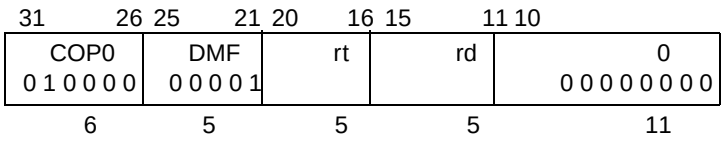
14.1.1.3 Operation:

32, 64 T: vAddr ← $((offset_{15})^{48}$ || $offset_{15...0})$ + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
CacheOp (op, vAddr, pAddr)

14.1.1.4 Exceptions:

TLB refill exception
TLB invalid exception
Buss error exception
Address error exception
Reserved instruction exception
Coprocessor unusable exception

14.1.2 DMFC0 - Doubleword Move From System Control Coprocessor



14.1.2.1 Format:

```
DMFC0   rt, rd
```

14.1.2.2 Description:

The contents of coprocessor register *rd* of the CP0 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.

14.1.2.3 Operation:

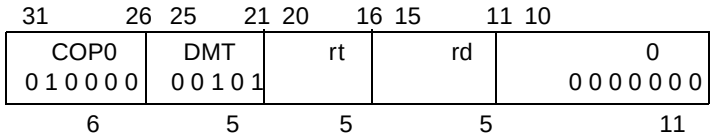
```
T:      data ← GPR[0,rd]

T+1:    CPR[rt] ← data
```

14.1.2.4 Exceptions:

Coprocessor unusable exception.
Reserved instruction exception for supervisor mode with ***Status.SX*** = 0 or user mode with ***Status.UX*** = 0.

14.1.3 DMTC0 - Doubleword Move To System Control Coprocessor



14.1.3.1 Format:

```
DMTC0   rt, rd
```

14.1.3.2 Description:

The contents of general register *rt* are loaded into coprocessor register *rd* of CP0.

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.

14.1.3.3 Operation:

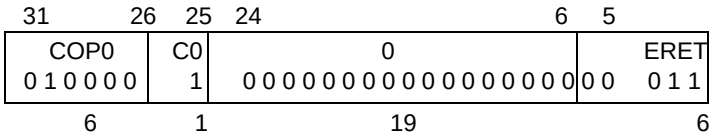
```
T:      data ← GPR[rt]

T+1:    CPR[0,rd] ← data
```

14.1.3.4 Exceptions:

Coprocessor unusable exception.
Reserved instruction exception for supervisor mode with ***Status.SX*** = 0 or user mode with ***Status.UX*** = 0.

14.1.4 ERET - Exception Return



14.1.4.1 Format:

```
ERET
```

14.1.4.2 Description:

ERET is the RM5200 instruction for returning from an interrupt, exception, or error trap. Unlike a branch or jump instruction, **ERET** does not execute the next instruction.

ERET must not itself 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.

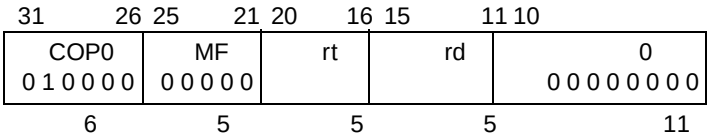
14.1.4.3 Operation:

```
T:    if SR2 = 1 then
          PC ← ErrorEPC
          SR ← SR31..3 || 0 || SR1..0
      else
          PC ← EPC
          SR ← SR31..2 || 0 || SR0
      endif
      LLbit ← 0
```

14.1.4.4 Exceptions:

Coprocessor unusable exception.

14.1.5 MFC0 - Move From CP0



14.1.5.1 Format:

```
MFC0    rt, rd
```

14.1.5.2 Description:

The contents of CP0 set 0 register *rd* are loaded into general register *rt*.

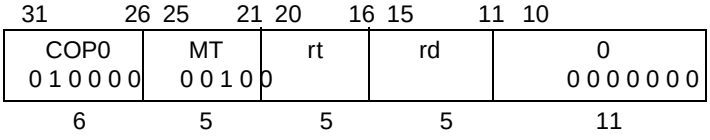
14.1.5.3 Operation:

```
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
```

14.1.5.4 Exceptions:

Coprocessor unusable exception.

14.1.6 MTC0 - Move To System Control Coprocessor



14.1.6.1 Format:

```
MTC0    rt, rd
```

14.1.6.2 Description:

The contents of general register *rt* are loaded into coprocessor register *rd* of CP0.

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. This instruction was first defined for MIPS I, but is not part of MIPS IV. The registers it references may change from processor to processor. The RM5200 have the same registers as have been implemented since the R4000.

14.1.6.3 Operation:

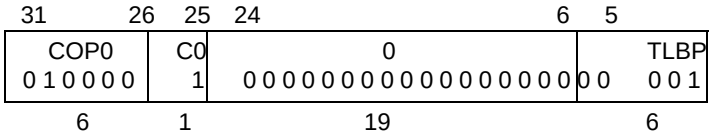
```
T:      data ← GPR[rt]

T+1:    CPR[0,rd] ← data
```

14.1.6.4 Exceptions:

Coprocessor unusable exception.

14.1.7 TLBP - Probe TLB For Matching Entry



14.1.7.1 Format:

```
TLBP
```

14.1.7.2 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.

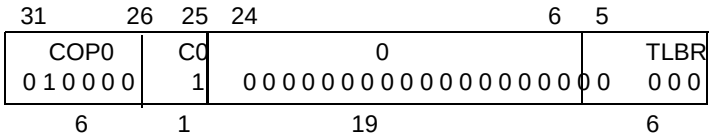
14.1.7.3 Operation:

```
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
```

14.1.7.4 Exceptions:

Coprocessor unusable exception.

14.1.8 TLBR - Read Indexed TLB Entry



14.1.8.1 Format:

```
TLBR
```

14.1.8.2 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.

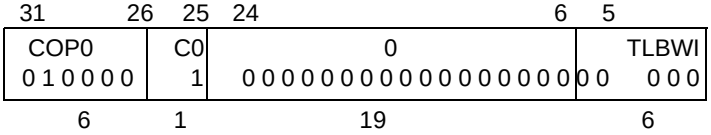
14.1.8.3 Operation:

```
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
```

14.1.8.4 Exceptions:

Coprocessor unusable exception.

14.1.9 TLBWI - Write Indexed TLB Entry



14.1.9.1 Format:

```
TLBWI
```

14.1.9.2 Description:

The TLB entry pointed at 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.

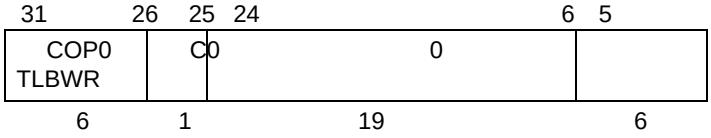
14.1.9.3 Operation:

```
T:   TLB[Index5..0] ←
        EntryHi[39:25] ||  (EntryHi[24:13] and not PageMask) ||  EntryLo1 || EntryLo0
```

14.1.9.4 Exceptions:

Coprocessor unusable exception.

14.1.10 TLBWR - Write Random TLB Entry



14.1.10.1 Format:

```
TLBWR
```

14.1.10.2 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.

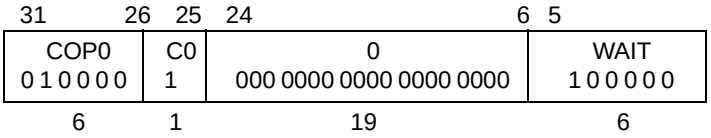
14.1.10.3 Operation:

```
T:   TLB[Random5..0] ←
        EntryHi[39:25] ||  (EntryHi[24:13] and not PageMask) ||  EntryLo1 || EntryLo0
```

14.1.10.4 Exceptions:

Coprocessor unusable exception.

14.1.11 WAIT - Enter Standby Mode



14.1.11.1 Format:

```
WAIT
```

14.1.11.2 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. See Appendix B, “Standby Mode Operation”, for more details.

14.1.11.3 Operation:

```
T:      if SysAD bus is idle then
            Enter Standby Mode
        endif
```

14.1.11.4 Exceptions:

Coprocessor unusable exception.

14.2 Implementation Specific Integer Instructions

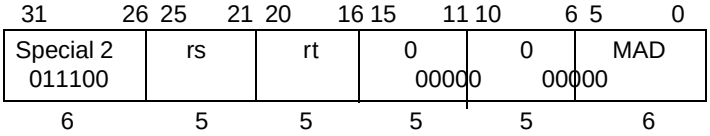
The instruction listed in Table 14.2: are implementation-specific instruction used by the RM5200 integer unit

Table 14.2: RM5200 Integer Unit Instruction.

Integer Instruction	Definition
MAD	Multiply/Add
MADU	Multiply/Add Unsigned
MUL	Multiply

The **MUL** instruction allows the RM5200 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.

14.2.1 MAD - Multiply/Add



14.2.1.1 Format:

```
MAD     rs, rt
```

14.2.1.2 Description:

The RM5200 adds a **MAD** instruction (Multiply-accumulate, with *HI* and *LO* 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 *LO*, while the upper 32-bits of the result are stored in the lower 32-bits of *HI*. 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, as explained in the appendix.

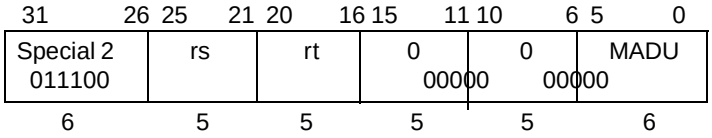
14.2.1.3 Operation:

$$T: \quad temp \leftarrow (HI_{31..0} \,||\, LO_{31..0}) + ((rs_{31})^{32} \,||\, rs_{31..0}) \times ((rt_{31})^{32} \,||\, rt_{31..0})$$
$$HI \leftarrow (temp_{63})^{32} \,||\, temp_{63..32}$$
$$LO \leftarrow (temp_{31})^{32} \,||\, temp_{31..0}$$

14.2.1.4 Exceptions:

None

14.2.2 MADU - Multiply/Add Unsigned



14.2.2.1 Format:

```
MADU    rs, rt
```

14.2.2.2 Description:

The RM5200 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 *LO*, while the upper 32-bits of the result are stored in the lower 32-bits of *HI*. 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, as explained in section Appendix A, “Cycle Time and Latency Tables”.

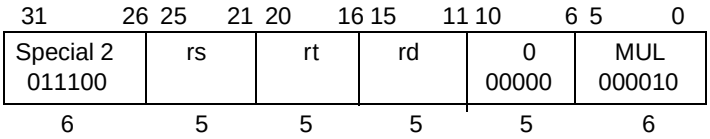
14.2.2.3 Operation:

$$T: \quad temp \leftarrow (HI_{31..0} \,||\, LO_{31..0}) + ((0)^{32} \,||\, rs_{31..0}) \times ((0)^{32} \,||\, rt_{31..0})$$
$$HI \leftarrow (temp_{63})^{32} \,||\, temp_{63..32}$$
$$LO \leftarrow (temp_{31})^{32} \,||\, temp_{31..0}$$

14.2.2.4 Exceptions:

None

14.2.3 MUL - Multiply



14.2.3.1 Format:

```
MUL     rd, rs, rt
```

14.2.3.2 Description:

The RM5200 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, as explained in section Appendix A, “Cycle Time and Latency Tables”.

The *HI* and *LO* registers are undefined after executing this instruction. For 16-bit operands, the latency of **MUL** is 3 cycles, with a repeat rate of 2 cycles. In addition, the **MUL** instruction will unconditionally slip or stall for all but 2 cycles of its latency.

14.2.3.3 Operation:

```
T:    Temp ← rs31..0 x rt31..0
      rd ← (temp31)32 || temp31..0
      HI ← undefined
      LO ← undefined
```

14.2.3.4 Exceptions:

None

14.3 MIPS IV Instruction Set Additions

The RM5200 microprocessors execute the MIPS IV Instruction Set Architecture (ISA), a superset of the MIPS III ISA, and provide backward compatibility to MIPS I, II and III. The additions of these new instructions enables the MIPS ISA to compete in the high-end numeric processing market which has 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. Register + register addressing 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 RM5200 microprocessors provide 8 condition codes to give the compiler increased flexibility in scheduling the comparison and the conditional moves. Table 14.3: lists in alphabetical order the new instructions which comprise the MIPS IV instruction set.

Table 14.3: 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[1]	Prefetch Indexed --- Register + Register
PREF[1]	Prefetch --- Register + Offset
RECIP.fmt	Reciprocal Approximation
RSQRT.fmt	Reciprocal Square Root Approximation
SDXC1	Store Double Word indexed to COP1
SWXC1	Store Word indexed to COP1

Note 1: Prefetch is not implemented in the RM5200 microprocessors and these instructions are treated as NOPs.

14.3.1 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.

14.3.1.1 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 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.

14.3.1.2 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.

14.3.1.3 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.

The RM5200 microprocessors do not implement prefetch. Therefore, the Prefetch instructions are executed as NOPs.

14.3.1.4 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 II. 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 16-bit offset (shifted left two bits and sign-extended to 64 bits) and the address of the instruction in the delay slot. If the contents of the floating-point condition code specified in the instruction are equal to the test value, the target address is branched to with a delay of one instruction. If the conditional branch is not taken and the nullify delay bit in the instruction (bit 17) is set, the instruction in the branch delay slot is nullified. Refer to the MIPS IV Instruction Set manual for more on the delay slot and nullify delay bit.

14.3.1.5 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.

14.3.1.6 Floating-Point Multiply-Add

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

14.3.1.7 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.

14.3.1.8 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.

14.3.1.9 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.

On the RM5200 microprocessors, the **RECIP** and **RSQRT** instructions fully meet IEEE accuracy. The rounding mode for the square-root intermediate value of the **RSQRT** instruction is RZ (round towards zero).

On the RM5200 microprocessors, 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 A, “Cycle Time and Latency Tables”, for operation latency tables.

14.4 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.

14.4.1 Scheduling a Load Delay Slot

MIPS I defines delayed loads: an instruction scheduling restriction requires that an instruction immediately following a load into register *Rn* cannot use *Rn* as a source register. The time between the load instruction and the time the data is available is the “load delay slot”. If no useful instruction can be put into the load delay slot, then a null operation (assembler mnemonic **NOP**) must be inserted. MIPS II ISA removed this restriction.

In the RM5200 processors programs will execute correctly when the loaded data is used by the instruction following the load, but this may require extra real cycles. Scheduling load delay slots can be desirable, both for performance and MIPS backward compatibility. However, the scheduling of load delay slots is not absolutely required.

14.4.2 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 (rightmost) byte is the most-significant byte; for a little-endian configuration, the low-order (rightmost) byte is the least-significant byte.

In either endianness, bit 0 is always the least-significant bit located at the rightmost bit position. Thus, bit designations are always little-endian.

In either endianness, instruction encodings remain 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]
- a 32-bit word
- a 16-bit halfword
- a 8-bit byte

a. a. Since the RM5230 only has a 32-bit SysAD bus, it cannot issue a doubleword access. It will use two word data cycles to transfer a doubleword.

The processor uses byte addressing for halfword, 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 doubleword (shown in Table 14.4:). Only the combinations shown in Table 14.4: are permissible; other combinations cause address error exceptions.

Table 14.4: Byte Access within a Doubleword

Access Type Mnemonic (*Value*)	Low Order Address Bits			Bytes Accessed															
				Big endian (63----------31----------0)								Little endian (63----------31----------0)							
	2	1	0	Byte								Byte							
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							

14.5 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

14.5.1 64-bit Operations

The RM5200 microprocessors have a 64-bit architecture which 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.

14.5.2 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 RM5200 have added the **MUL**, **MAD** and **MADU** instructions and have reduced the latencies and repeat rates of the existing multiply instructions to substantially improve the integer multiply performance. The new **MUL** instruction in the RM5200 returns the multiply result directly to the register file and eliminates the need for a separate instruction to read the *HI*/*LO* registers (as was the case for the R5000). The **MAD** (multiply/**MADU** (multiply/add unsigned) instructions on the RM5200 implement the atomic multiply-accumulate to the content of the *HI*/*LO* registers. The integer multiplies latencies and repeat rates have been reduced by one cycle for the word multiplies and by two cycles for the double word multiplies. These instructions are described later in this section.

Appendix A, “Cycle Time and Latency Tables”, gives the number of processor cycles (PCycles) required to resolve an interlock or stall between various multiply or divide instructions.

14.5.3 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.

14.5.3.1 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.

14.5.3.2 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 is not taken, the instruction in the delay slot is nullified.

14.5.4 Special Instructions

Special instructions allow the software to initiate traps; they are always R-type. Exception instructions are extensions to the MIPS ISA.

14.5.5 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.

Appendix A Cycle Time and Latency Tables

This Appendix lists the latency and repeat cycles for Integer and Floating Point operations, and the cycle counts for Cache operations.

A.1 Latency Tables

Table A.1: Integer Multiply and Divide

Opcode	Operand Size	Latency	Repeat Rate	Stall Cycles
MULT/U, MAD/U	16 bit	3	2	0
	32 bit	4	3	0
MUL	16 bit	3	2	1
	32 bit	4	3	2
DMULT, DMULTU	any	7	6	0
DIV, DIVD	any	36	36	0
DDIV, DDIVU	any	68	68	0

Table A.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
frecp (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

Opcode	Latency	Repeat
fmov (sngl/dbl)	1	1
fmovc (sngl/dbl)	1	1
fabs (sngl/dbl)	1	1
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

A.2 Cycle Counts for RM5200 Cache Misses

A.2.1 Mnemonics

To describe a processor sequences that include an external access, the number of cycles taken must be calculated based on the system response to such an access. These sequences are described with equations based on the following mnemonics:

- SYSDIV: SYStem DIVsor is defined as the number of processor cycles (pclocks) per **SysClock** cycles and ranges from 2 to 8.
- ML: Memory latency is defined as the number of cycles the SysAD bus is driven by the external agent before the first doubleword of data appears. This is the cycle count from SysAD release to first valid data.
- DD: Data Duration is 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)}: This pclock to **SysClock** synchronization term is used in many of the following equations. It has a value (number of pclock cycles) between 0 and (SYSDIV - 1) depending on the alignment of the execution unit with the system clock.

A.2.2 Primary Data Cache Misses

Caveats to Primary Data Cache Misses:

1. All Cycle counts are in processor cycles (pclocks).
2. Data Cache misses have lower priority than write backs, external requests, and Instruction Cache misses. If the write back buffer contains unwritten data when a dcache miss occurs, the write back buffer will be 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 will be handled first. External requests will be completed before beginning the handling of a data cache miss.
3. For all data cache misses handling of the returning cache miss data must wait for the store buffer and response buffer to empty (if they are filled) and for dirty data (if present) to be moved from the data cache to the write back buffer. It is possible that if all of the above occur, and the data cache miss hits in the secondary cache, the first doubleword of data will return before the data cache is available. In this case the first doubleword of data will hold in the response buffer for one or two cycles which will add to the latency of the data cache miss.
4. In handling a data cache miss a write back may be required which will fill the write back buffer. Write backs can affect subsequent cache misses since they will stall until the write back buffer is written back to memory.
5. All cycle counts are best case assuming no interference from the mechanisms described above.

The following equations yield the number of stall cycles for data cache misses under the specified circumstances.

Secondary cache hit:

Number_Of_Cycles_For_DCache_Miss_Secondary_Cache_Hit = 1 + {0 to (SYSDIV -1)} + (3 x SYSDIV) + 2

Secondary cache miss:

Number_Of_Cycles_For_DCache_Miss_Secondary _Cache_Miss = 1 + {0 to (SYSDIV - 1)} + (2 x SYSDIV) + (ML x SYSDIV) + (1 x SYSDIV) + 2

Note: Memory Latency (ML) has a minimum of 3 cycles to allow for the secondary cache probe. The 1 at the front of these equations is for the cycle during which the miss in the data cache took place. The 2 at the end of these equations is for one cycle to place the data in the data cache and one cycle to move it from the data cache to the pipeline.

A.2.3 Instruction Cache Misses

Caveats to Instruction Cache Misses

1. All cycle counts are in processor cycles (pclocks).
2. Instruction Cache misses have lower priority than write backs and external requests. If the write back buffer contains unwritten data when an instruction cache miss occurs, the write back buffer will be retired before the handling of the instruction cache miss is begun. External requests will be 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.

The following equations yield the number of stall cycles for instruction cache misses under the specified circumstances.

Secondary cache hit:

Number _Of_Cycles_For_ICache_Miss_Secondary_Cache_Hit = 1 + {0 to (SYSDIV - 1)} + (6 x SYSDIV) + 3

Secondary cache miss:

Number_Of_Cycles_For_ICache_Miss_Secondary_Cache_Miss = 1 + {0 to (SYSDIV - 1)} + (2 x SYSDIV) + (ML x SYSDIV) + (DD x SYSDIV) + 3

Note: Memory Latency (ML) has a minimum of 3 to allow for the secondary cache probe. The 1 at the front of these equations is for the cycle during which the miss in the instruction cache took place

A.3 Cycle Counts for RM5200 Cache Instructions

Caveats to Cache Instructions

1. All cycle counts are in processor cycles.
2. All cache instructions have lower priority than cache misses, write backs and external requests. If the write back buffer contains unwritten data when a cache instruction is executed, the write back buffer will be retired before the cache instruction is begun. If an instruction cache miss occurs at the same time as a cache instruction is executed, the instruction cache miss will be handled first. Cache instructions are mutually exclusive with respect to data cache misses. External requests will be completed before beginning a cache instruction.
3. For all data cache instructions the cache instruction machine waits for the store buffer and response buffer to empty before beginning the cache instruction. This can add 3 cycles to any data cache instruction 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 instructions of the form xxxx_Writeback_xxxx may perform a write back which fill the write back buffer. Write backs can affect subsequent cache instructions since they stall until the write back buffer is written back to memory. Cache instructions 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 A.3: Primary Data Cache Operations

Code	Name	Number of Cycles
0	Index_Writeback_Invalidate_D	10 Cycles if the cache line is clean. 12 Cycles if the cache line is dirty. (Write back)
1	Index_Load_Tag_D	7 Cycles
2	Index_Store_Tag_D	8 Cycles
3	Create_Dirty_Exclusive_D	10 Cycles for a cache hit. 13 Cycles for a cache miss if the cache line is clean. 15 Cycles for a cache miss if the cache line is dirty. (Writeback)
4	Hit_Invalidate_D	7 Cycles for a cache miss. 9 Cycles for a cache hit.
5	Hit_Writeback_Invalidate_D	7 Cycles for a cache miss. 12 Cycles for a cache hit if the cache line is clean. 14 Cycles for a cache hit if the cache line is dirty. (Writeback)
6	Hit_Writeback_D	7 Cycles for a cache miss. 10 Cycles for a cache hit if the cache line is clean. 14 Cycles for a cache hit if the cache line is dirty. (Writeback)

Table A.4: Primary Instruction Cache Operations

Code	Name	Number of Cycles
0	Index_Invalidate_I	7 Cycles.
1	Index_Load_Tag_I	7 Cycles.
2	Index_Store_Tag_I	8 Cycles.
3	NA	
4	Hit_Invalidate_I	7 Cycles for a cache miss. 9 Cycles for a cache hit.
5	Fill_I	This equation yields the number of processor cycles for a Fill_I cache operation: **Number_Of_Cycles_For_A_Fill_I_Cacheop = 10 + {0 to (SYSDIV - 1)} + (2 x SYSDIV) + (ML x SYSDIV) + (DD x SYSDIV).**
6	Hit_Writeback_I	7 Cycles for a cache miss. 20 Cycles for a cache hit. (Writeback)

Table A.5: Secondary Cache Operations

Code	Name	Number of Cycles
0	Flash_Invalidate_S	This equation yields the number of processor cycles for a Flash_Invalidate_S cache operation: **Number_Of_Cycles_For_Flash_Invalidate_S_Cacheop = 3 + {0 to (SYSDIV - 1)} + (1 x SYSDIV) + 3**
1	Index_Load_Tag_S	This equation yields the number of processor cycles for an Index_Load_Tag_S cache operation: **Number_Of_Cycles_For_Index_Load_Tag_S = 3 + {0 to (SYSDIV - 1)} + (4 x SYSDIV) + 3**
2	Index_Store_Tag_S	This equation yields the number of processor cycles for an Index_Store_Tag_S cache operation: **Number_Of_Cycles_For_Index_Store_Tag_S = 3 + {0 to (SYSDIV - 1)} + (1 x SYSDIV) + 3**
3	NA	
4	NA	

Code	Name	Number of Cycles
5	Page_Invalidate_S	This equation yields the number of processor cycles for a Page_Invalidate_S cache operation: **Number_Of_Cycles_For_Page_Invalidate_S = 3 + {0 to (SYSDIV - 1)} + (128 x SYSDIV) + 3**
6	NA	

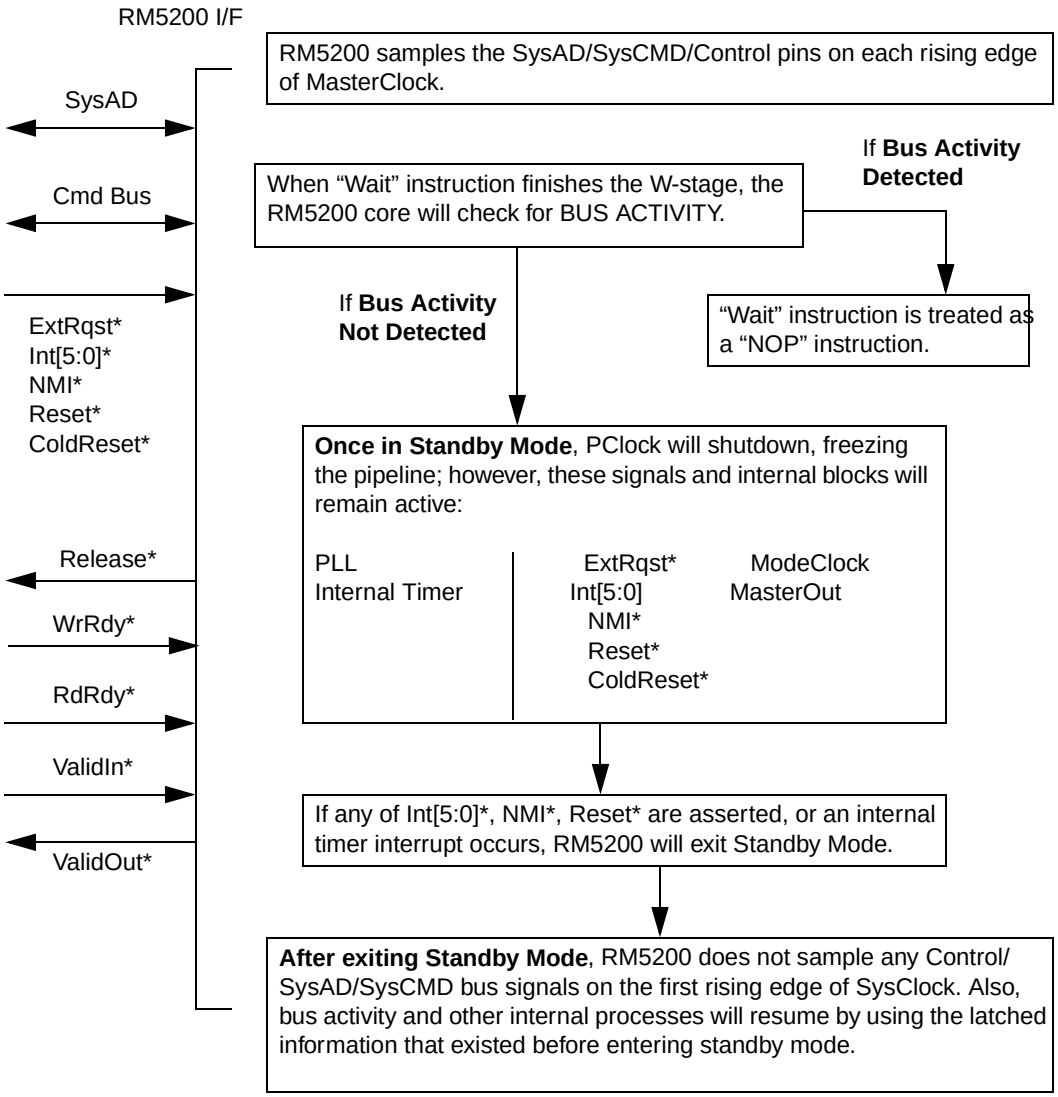
Appendix B Standby Mode Operation

The Standby Mode operation is a means of reducing the internal core's power consumption when the CPU is in a "standby" state. In this section, the Standby Mode operation is discussed.

B.1 Entering Standby Mode

To enter standby mode, first execute the **WAIT** instruction. When the **WAIT** instruction finishes the W pipe-stage, if the SysAD bus is currently idle, the internal clocks will shut down, thus freezing the pipeline. The PLL, internal timer/counter, and the "wake up" input pins: **Int[5:0]***, **NMI***, **ExtRqst***, **Reset*** and **ColdReset*** will continue to operate in their normal fashion, and the output clock (**ModeClock**) will continue to run. If these conditions are not correct when the **WAIT** instruction finishes the W pipe-stage (i.e., the **SysAD** bus is not idle), the **WAIT** is treated as a **NOP**.

Once the processor is in standby mode, any interrupt including the internally generated timer interrupt, will cause the processor to exit standby mode and resume operation where it left off. The **WAIT** instruction is typically inserted in the idle loop of the operating system or real time executive. Figure B.1 illustrates the Standby Mode Operation.



Note: During standby mode, all control signals for the CPU must be deasserted or put into the appropriate state, and all input signals, except Int[5:0], Reset*, ColdReset* and ExtRqst*, must remain unchanged.*

Figure B.1 Standby Mode Operation

Appendix C Instruction Hazards

C.1 Introduction

This appendix identifies the RM5200 Instruction Hazards. Certain combinations of instructions are not permitted because the results of executing such combinations are unpredictable in combination with some 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.

For the following code hazards, the behavior is undefined and unpredictable.

C.1.1 List of Instruction Hazards:

- Any instruction that would modify *PageMask* or *EntryHi* or *EntryLo0* or *EntryLo1* or *Random* CP0 Registers should not be followed by a **TLBWR** instruction. There should be at least two integer instructions between the register modification and the **TLBWR** instruction.
- Any instruction that would modify *PageMask* or *EntryHi* or *EntryLo0* or *EntryLo1* or *Index* CP0 Registers should not be followed by a **TLBWI** instruction. There should be at least two integer instructions 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 two integer instructions 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 two integer instructions between the register modification and the **TLBP** instruction.
- Any instruction that would modify the *PageMask* or *EntryHi* or CP0 Registers or the contents of the JTLB should not be followed by any instruction that uses the JTLB. There should be at least two integer instructions between the register modification and the use of the JTLB.
- Any instruction that would modify the *EPC* or *ErrorEPC* or *Status* CP0 Registers should not be followed by an **ERET** instruction. There should be at least two integer instructions between the register modification and the **ERET** instruction.
- The two instructions preceding any **DIV**, **DIVU**, **DDIV**, **DDIVU**, **MAD**, **MADU**, **MUL**, **MULT**, **MULTU**, **DMULT** or **DMULTU** instructions should not read the *HI* or *LO* registers. There should be at least two integer instructions between the register read and the register modification.
- If a **MTLO**/**MTHI** instruction is issued after any **DIV**, **DIVU**, **DDIV**, **DDIVU**, **MAD**, **MADU**, **MUL**, **MULT**, **MULTU**, **DMULT** or **DMULTU** instructions but before any **MFLO**/**MFHI** instruction, the results of the *HI*/*LO* registers is unpredictable.
- If any instruction modifies the ***Config.K0*** field, 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 5 integer instructions.
- If the dcache is to be locked, it is recommended that the two instructions that follow the instruction that sets ***SR.DL*** should not cause a dcache miss. This dcache miss might cause a cache refill into dcache set A that the user did not expect. The register modification and the next dcache miss should be separated by at least two integer instruction.
- If the icache is to be locked, it is recommended that the four instructions that follow the instruction that sets ***SR.IL*** should not cause an icache miss. This icache miss might cause a cache refill into icache set A that the user did not expect. The register modification and next icache miss should be separated by at least four integer instructions. The safest way is to run the instructions uncached. KSEG1 is always uncached. Or a page may be specified as uncached with the TLB.

- 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.

Appendix D Subblock Order

The order that data is returned in response to a processor block read request is *subblock ordering*. In subblock ordering, the processor delivers the address of the requested doubleword (RM526x/RM527x) or word (RM523x) within the block. An external agent must return the block of data using subblock ordering, starting with the addressed doubleword or word. Subblock ordering is use so that the pipeline can be restarted at the reception of required data instead of having to wait for the complete cache line. This is know as restart on critical data.

All of QED's MIPS CPU write their data in sequential order. In sequential data movement each double word or word is handled in a linear sequence.

Figure D.1 shows a sequential order in which doubleword 0 is taken first and doubleword 3 is taken last.

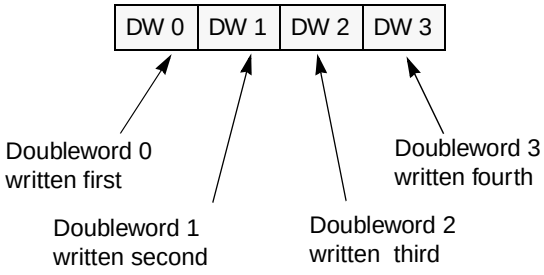


Figure D.1 Storing a Data Block in Sequential Order

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 RM526x and RM527x is a doubleword. Figure D.2 shows the retrieval of a block of data that consists of 4 doublewords, in which DW2 is taken first.

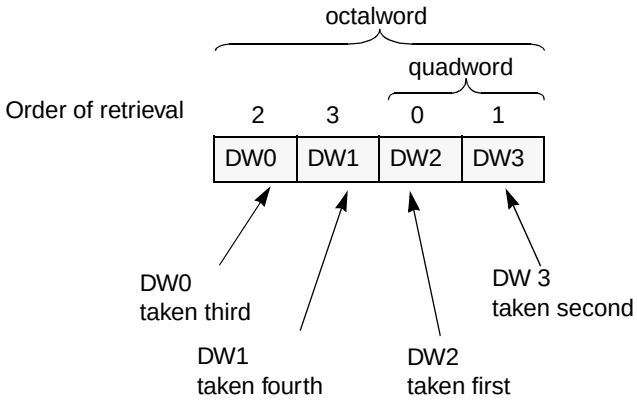


Figure D.2 Retrieving Data in a Subblock Order

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. 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: list the subblock ordering of doublewords for 8-word blocks. 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.

Table D.1: Subblock Ordering Sequence (doubleword)

First Address	x..x00	x..x01	x..x10	x..x11
Second Address	x..x01	x..x00	x..x11	x..x10
Third Address	x..x10	x..x11	x..x00	x..x01
Fourth Address	x..x11	x..x10	x..x01	x..x00

D.1 Generating Subblock Order of Words (RM523x)

Using the same scheme, Table D.2: list the subblock ordering of words for 8-word block. The subblock ordering is generated by an XOR of the subblock address with the binary count of the word.

Therefore, the third word retrieved from a block of data with a starting address of 010_2 is determined by taking the XOR of address 010_2 with the binary count of word 2, 010_2. The result is 000_2, or word 0, as shown in Table D.2:.

Table D.2: Subblock Ordering Sequence: (word)

First Address	x..x000	x..x001	x..x010	x..x011	x..x100	x..x101	x..x110	x..x111
Second Address	x..x001	x..x000	x..x011	x..x010	x..x101	x..x100	x..x111	x..x110
Third Address	x..x010	x..x011	x..x000	x..x001	x..x110	x..x111	x..x100	x..x101
Fourth Address	x..x011	x..010	x..x001	x..x000	x..x111	x..x110	x..x101	x..x100
Fifth Address	x..x100	x..x101	x..x110	x..x111	x..x000	x..x001	x..x010	x..x011
Sixth Address	x..x101	x..x100	x..x111	x..x110	x..x001	x..x000	x..x011	x..x010
Seventh Address	x..x110	x..x111	x..x100	x..x101	x..x010	x..x011	x..x000	x..x001
Eight Address	x..x111	x..x110	x..x101	x..x100	x..x011	x..x010	x..x001	x..x000

Appendix E PLL Analog Power Filtering

The Phase Lock Loop circuit requires several passive components for proper operation. The recommended configuration is shown in Figure E.1. It is also recommended that the smallest capacitor be placed the closest to the chip package as depicted.

For the RM52x0 devices, Board Vcc in Figure E.1 should be VccIO (3.3 volts). For the RM52x1 devices, Board Vcc in Figure E.1 should be VccInt (2.5 volts).

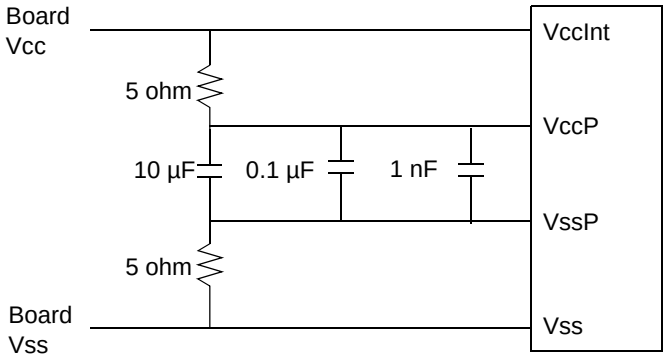


Figure E.1 PLL Filter Circuit

Appendix F JTAG Interface

The RM5200 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.

F.1 Test Data Registers

The RM5200 processor test logic contains three registers; Bypass, Boundary Scan, and Instruction.

F.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**.

F.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 RM5200 processor. This register allows for the testing of board interconnections to detect defects such as opens circuits and short circuits. Figure F.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.

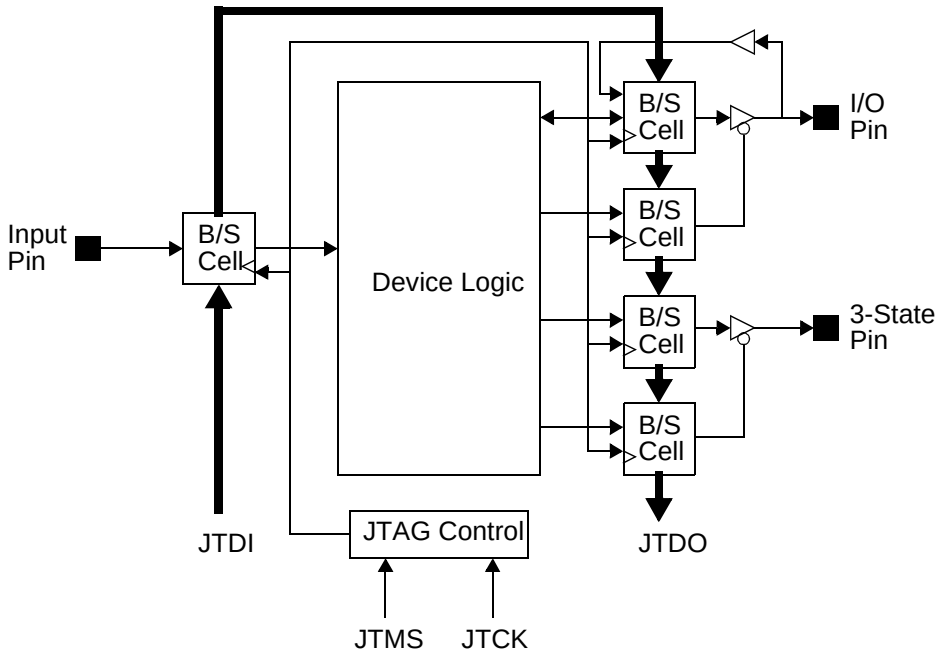


Figure F.1 JTAG Boundary Scan Cell Organization

F.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 F.3 for more information on the TAP controller and its states.

Table F.1: shows the encoding of the instructions supported by the RM5200.

Table F.1: JTAG Instruction Register Encoding

Instruction Code	Instruction
000	EXTEST
001-101	Reserved
110	SAMPLE/PRELOAD
111	BYPASS

F.2 Boundary Scan Instructions

The RM5200 processor supports three instructions; EXTEST, SAMPLE/PRELOAD, and BYPASS. These instructions are encoded in the instruction register as shown in section F.1.3. The following subsections describe these instructions in more detail.

F.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 RM5200 processor. Values scanned into the register become the output values. The input boundary scan cells sample the input pins of the RM5200 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 F.3 for more information on the TAP controller:

- The RM5200 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 RM5200 processor must be reset.

F.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 RM5200 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 RM5200 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.

F.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 RM5200 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.

F.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 F.2 shows a diagram of the TAP controller state machine.

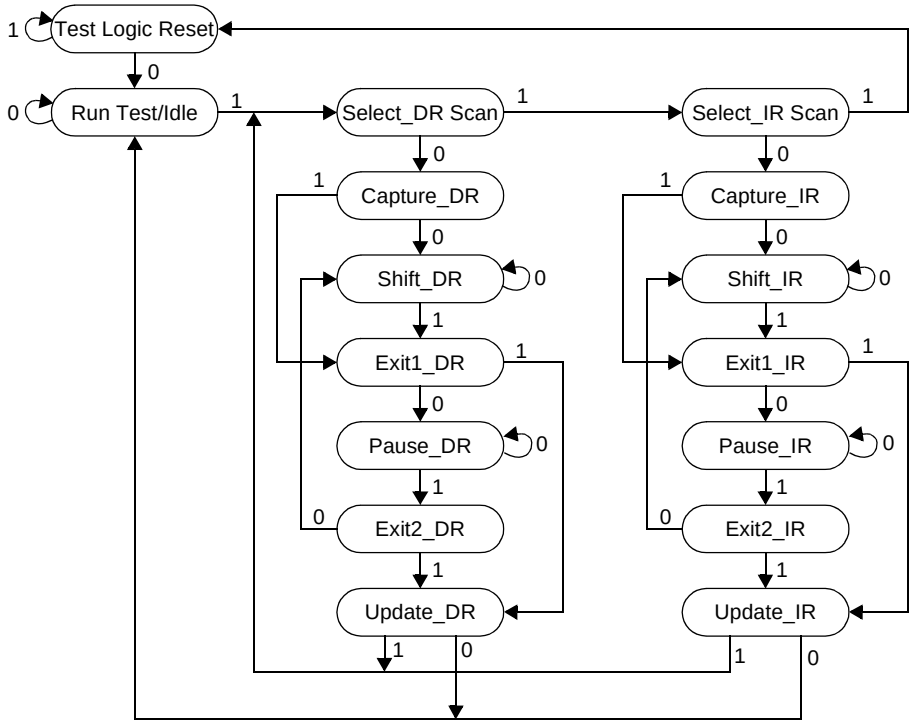


Figure F.2 TAP Controller State Diagram

F.3.1 Test-Logic-Reset State

In the *Test-Logic-Reset* state the boundary scan test logic is disabled, allowing normal operation of the RM5200 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.

F.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.

F.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.

F.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.

F.3.5 Capture_DR State

In this state the boundary scan register captures input pin data is the current instruction is either EXTEST of 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.

F.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.

F.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.

F.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.

F.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.

F.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.

F.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.

F.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.

F.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.

F.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.

F.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.

F.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.

F.4 TAP Controller Initialization

In the RM5200, 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 RM5200 does not support the JTAG **TRST*** signal.

F.5 Boundary Scan Signals

The RM5200 supports those signals listed in Table F.2:. The Test-Reset (**TRST***) input is not supported. The **JTMS** input must be used to initialize the TAP controller.

Table F.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

Index

Numerics

A

B

C

D

E

F

R

S

T

U

V

W

X

Disclaimers

When using this document, keep the following in mind:

6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. The QED logo and RISCMark are trademarks of Quantum Effect Devices, Inc.
12. 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)