Integrated Device Technology, Inc.

IDT R5000™ RISC Microprocessor

Instruction Set Reference Manual

Version 1.0
February 1996

2975 Stender Way, Santa Clara, California 95054
Telephone: (800) 345-7015 • TWX: 910-338-2070 • FAX: (408) 492-8674
Printed in U.S.A.

CPU Instruction Set Summary

Chapter 1

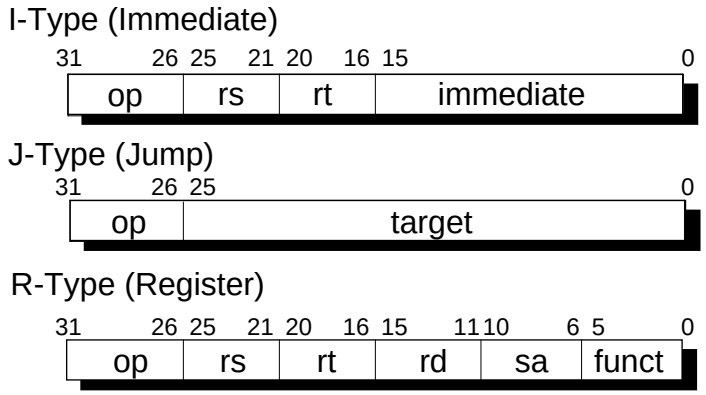
Introduction

The R5000 processor executes the MIPS IV instruction set, which is a superset of the MIPS III instruction set and is backward compatible. 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.

Types of Instruction Sets

There are three types of instruction types as shown in Figure 1.1.



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 1.1 CPU Instruction Formats

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

Load and Store Instructions

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

Scheduling a Load Delay Slot

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

In the R5000 processor, the instruction immediately following a load instruction can use the contents of the loaded register, however in such cases hardware interlocks insert additional real cycles. Consequently, scheduling load delay slots can be desirable, both for performance and R-Series processor compatibility. However, the scheduling of load delay slots is not absolutely required.

Defining Access Types

Access type indicates the size of a R5000 processor 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 byte is the most-significant byte; for a little-endian configuration, the low-order byte is the least-significant byte.

The access type, together with the three low-order bits of the address, define the bytes accessed within the addressed doubleword (shown in Table 1.1). Only the combinations shown in Table 1.1 are permissible; other combinations cause address error exceptions.

Table 1.1 Byte Access within a Doubleword

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

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

64-bit Operations

When operating in 64-bit mode, 32-bit 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.

Cycle Timing for Multiply and Divide Instructions

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

Table 1.2 gives the number of processor cycles (PCycles) required to resolve an interlock or stall between various multiply or divide instructions, and a subsequent MFHI or MFLO instruction.

Table 1.2 Multiply/Divide Instruction Latency and Repeat Rates

Instruction	Latency	Repeat Rate
MULT	5	4
MULTU	5	4
DIV	36	36
DIVU	36	36
DMULT	9	8
DMULTU	9	8
DDIV	68	68
DDIVU	68	68

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.

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 32-bit or 64-bit byte address contained in one of the general purpose registers.

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 32 bits). All branches occur with a delay of one instruction.

If a conditional branch is not taken, the instruction in the delay slot is nullified.

Special Instructions

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

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.

Individual coprocessor instructions are described in Appendices A (for CP0) and B (for the FPU, CP1).

CP0 instructions perform operations specifically on the System Control Coprocessor registers to manipulate the memory management and exception handling facilities of the processor.

MIPS IV Instruction Set Additions

The R5000 Microprocessor runs the MIPS IV instruction set, which is a superset of the MIPS III instruction set and is backward compatible. The additions of these new instructions enables the MIPS architecture 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 immediate 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. 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 R5000 microprocessor provides an 8-bit condition code field to give the compiler increased flexibility in scheduling the comparison and the conditional moves. Table 1.3 lists in alphabetical order the new instructions which comprise the MIPS IV instruction set.

Instruction	Definition
BC1F	Branch on FP Condition Code False
BC1T	Branch on FP Condition Code True
BC1FL	Branch on FP Condition Code False Likely

Table 1.3 MIPS IV Instruction Set Additions and Extensions

Instruction	Definition
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.sd	Floating PointMultiply-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.sd	Floating Point Multiply-Subtract
NMADD.sd	Floating Point Negative Multiply-Add
NMSUB.sd	Floating Point Negative Multiply-Subtract
PREFX[a]	Prefetch Indexed --- Register + Register
PREF[a]	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

Table 1.3 MIPS IV Instruction Set Additions and Extensions (Continued)

a. Prefetch is not implemented in the R5000 microprocessor and these instructions are no-ops.

Table 1.4 lists the COP0 instructions for the R5000 processor. COP0 instructions are those which are not architecturally visible and are used by the kernel.

COP0 Instruction	Definition
ERET	Return from Exception
TLBP	Probe for TLB Entry
TLBR	Read TLB Entry
TLBW	Write TLB Entry
DCTR	Data Cache Tag Read
DCTW	Data Cache Tag Write

Table 1.4 R5000 COP0 Instructions

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.

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.

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.

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 PFETCH instruction is considered a standard Coprocessor 1 instruction. The R5000 microprocessor does not implement prefetch and these instruction are executed as no-ops.

Branch on Floating Point Coprocessor

BC1T - Branch on FP condition True
BC1F - Branch on FP condition False
BC1TL - Branch on FP condition True Likely
BC1FL - Branch on FP condition False Likely

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

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

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.

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.

Floating Point Compare

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

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

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

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

Floating Point Conditional Moves

MOVT.fmt - Floating Point Conditional Move on condition code true

MOVF.fmt - Floating Point Conditional Move on condition code false

MOVN.fmt - Floating Point Conditional Move on register not equal to zero

MOVZ.fmt - Floating Point Conditional Move on register equal to zero

The four floating point conditional move instructions are 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.

Reciprocal's

RECIP.fmt - Reciprocal Approximation

RSQRT.fmt - Reciprocal Square Root Approximation

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

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

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

On the R5000 microprocessor, the RECIP instruction has the same latency as a DIV instruction, but a RSQRT is faster than a SQRT followed by a RECIP.

Table 1.5 shows the integer instruction latencies in the R5000 Microprocessor.

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

Table 1.5 R5000 Integer Instruction Latencies

Table 1.6 shows the floating point instruction latencies in the R5000 CPU.

Instruction Group	Latency	Repeat
Load	2	1
Load Indexed	3	2
Store	N/A	1
Store Indexed	N/A	2
Prefetch	N/A	1
Prefetch Indexed	N/A	2
Compare	1	1
Absolute	1	1
Negative	1	1
Move	1	1
Conditional Moves	1	1
Add	4	1
Subtract	4	1
MUL.s	4	1
MUL.d	5	2

Table 1.6 R5000 Floating Point Instruction Latencies

Instruction Group	Latency	Repeat
MADD.s	4	1
MADD.d	5	2
DIV.s	21	19
DIV.d	36	34
SQRT.s	21	19
SQRT.d	36	34
RECIP.s	21	19
RECIP.d	36	34
RSQRT.s	38	36
RSQRT.d	68	66
mtc1, dmtc1	2	1
mfc1, dmfc1	2	1
CTC1	6	3
CFC1	2	1
ROUND.w	4	1
ROUND.l[a]	4	1
TRUNC.w	4	1
TRUNC.l	4	1
CEIL.w	4	1
CEIL.l	4	1
FLOOR.w	4	1
FLOOR.l	4	1
CVT.s.d	4	1
CVT.s.w	6	3
CVT.s.l[b]	6	3
CVT.d.s	4	1
CVT.d.w	4	1
CVT.d.l[b]	4	1
CVT.w.s	4	1
CVT.w.d	4	1
CVT.l.s	4	1
CVT.l.d	4	1

Table 1.6 R5000 Floating Point Instruction Latencies

a. Trap on greater than 53 bits of significance.
b. Trap on greater than 52 bits of significance.

Introduction

The R5000 processor has a five-stage instruction pipeline. 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.

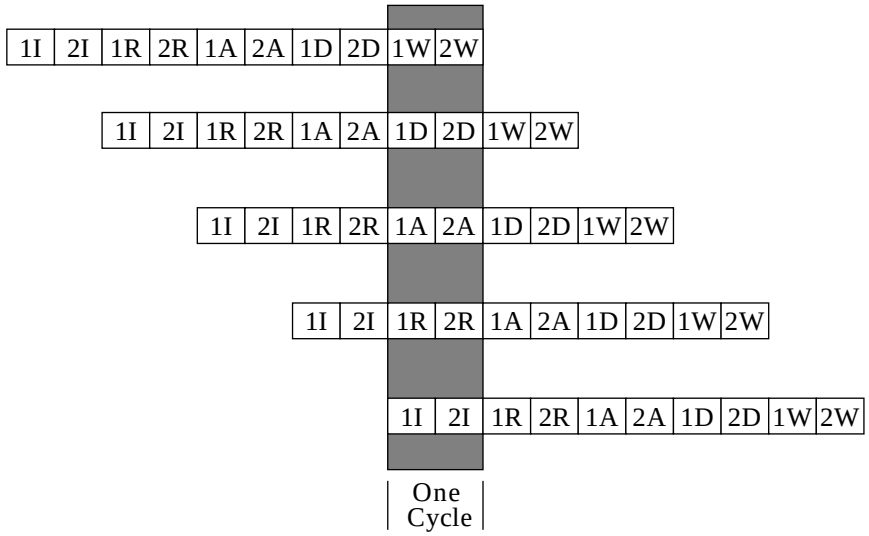


Figure 2.1 Instruction Pipeline Stages

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

II - 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.

2I - Instruction Fetch, Phase Two

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

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

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.
- Calculate branch address (if applicable).

1A - Execution - Phase One

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

- Any result from th 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.

2A - Execution - Phase Two

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

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

1D - Data Fetch - Phase One

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

The DTLB data address translation completes.

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.

1W - Write Back, Phase One

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

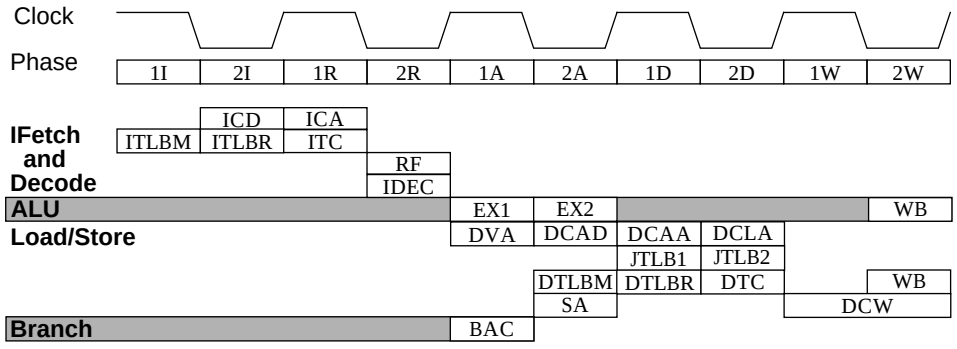
2W - Write Back, Phase Two

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

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	ICD	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

Branch Delay

The CPU pipeline has a branch delay of one cycle and a load delay of one cycles. 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.

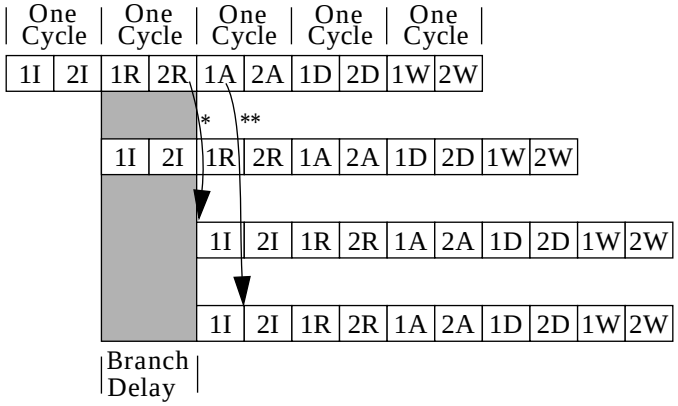


Figure 2.3 CPU Pipeline Branch Delay

Load Delay

The completion of a load at the end of the DS 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.

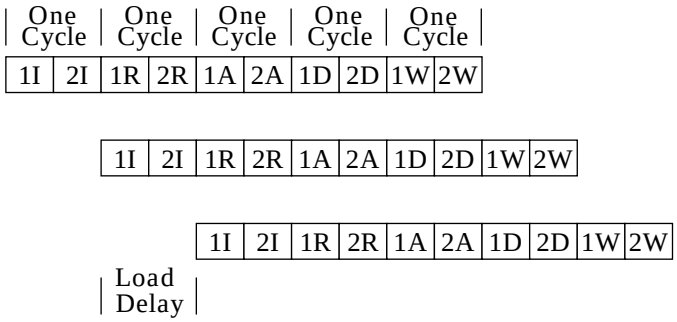


Figure 2.4 CPU Pipeline Load Delay

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
Stall	ITM	ICM		DCM	
				CPE	
Slip		LDI			
		MDSt			
		FCBusy			
Exceptions	ITLB	IBE	RI	DBE	
		IPErr	CUn	NMI	
			BP	Reset	
			SC	DPErr	
			DTLB	OVF	
			DTMod	Trap	
			Intr		

Table 1:

Table 2.2 Pipeline Exceptions

Exception	Description
ITLB	Instruction Translation or Address Exception
Intr	External Interrupt
IBE	IBus Error
RI	Reserved Instruction
BP	Breakpoint
SC	System Call
CUn	Coprocessor Unusable
IPErr	Instruction Parity Error
OVF	Integer Overflow

Exception	Description
FPE	FP Interrupt
ExTrap	EX Stage Traps
DTLB	Data Translation or Address Exception
TLBMod	TLB Modified
DBE	Data Bus Error
DPErr	Data Parity Error
NMI	Non-maskable Interrupt
Reset	Reset

Table 2.3 Pipeline Interlocks

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

Exception Conditions

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

When an exception condition is detected, the R5000 processor aborts the instruction which caused the exception, as well as all subsequent instructions. When this instruction reaches the W stage, three events occur;

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

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

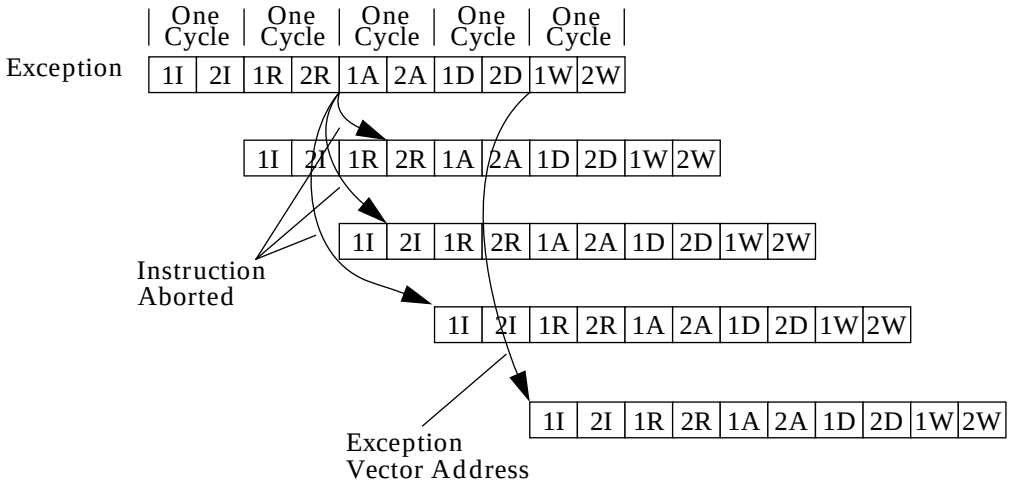
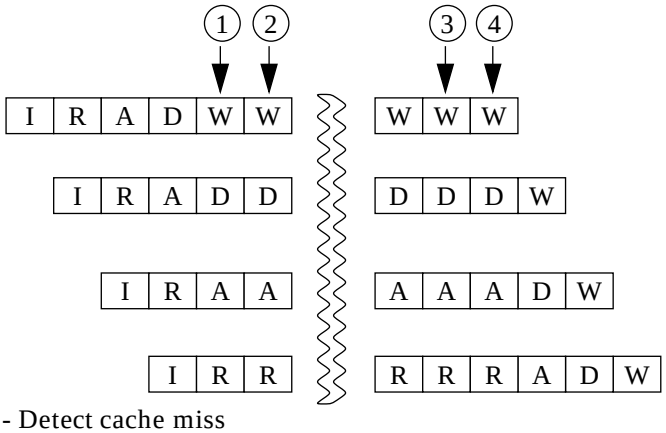


Figure 2.5 Exception Detection Mechanism

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.



1 - Detect cache miss
2 - Start moving dirty cache line data to write buffer
3 - Fetch first doubleword into cache and restart pipeline
4 - Load remainder of cache line into cache

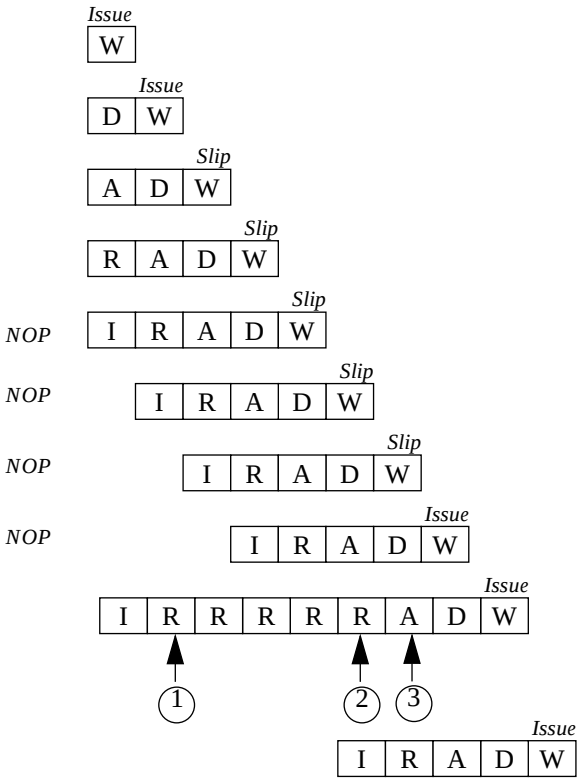
Figure 2.6 Servicing a Data Cache Miss

The data cache miss is detected in the D stage of the pipeline. If the cache line to be replaced is dirty, the W bit is set and data is moved to the internal write buffer in the next cycle. The squiggly line in Figure 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.

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 cycles are retried on subsequent cycles until they are issued. Pipeline stages D and W advance normally during slips in an attempt to resolve the conflict. NOP's are inserted into the bubbles which are created in the pipeline. Instructions caused by "branch likely" instructions, ERET, or exceptions do not cause clips.

Figure 8 shows how instruction can slip during an instruction cache miss.



1 - Detect cache miss
2 - Start moving dirty cache line data to write buffer
3 - Fetch first doubleword into cache and restart pipeline
4 - Load remainder of cache line into cache

Figure 2.7 Slips During and Instruction Cache Miss

Instruction cache misses are detected in the R-stage of the pipeline. Slips are detected in the A stage. Instruction cache misses never require a writeback operation 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.

Write Buffer

The R5000 processor contains a write buffer which improves the performance of write operations to external memory. All write cycles use the write buffer. The write buffer holds up to four 64-bit address and data pairs.

On a cache miss requiring a write-back, the entire buffer is used for the write-back data and allows the processor to proceed in parallel with the memory update. For uncached and write-through stores, the write buffer decouples the CPU from the write to memory. If the write buffer is full, additional stores are stalled until there is room for them in the write buffer.

Introduction

The R5000 processor incorporates a simple dual-issue mechanism which allows two instructions to be dispatched per cycle under certain conditions. A FPU ALU operation can be dispatched along with any other type of instruction, as long as the other instruction is not another FP ALU operation.

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

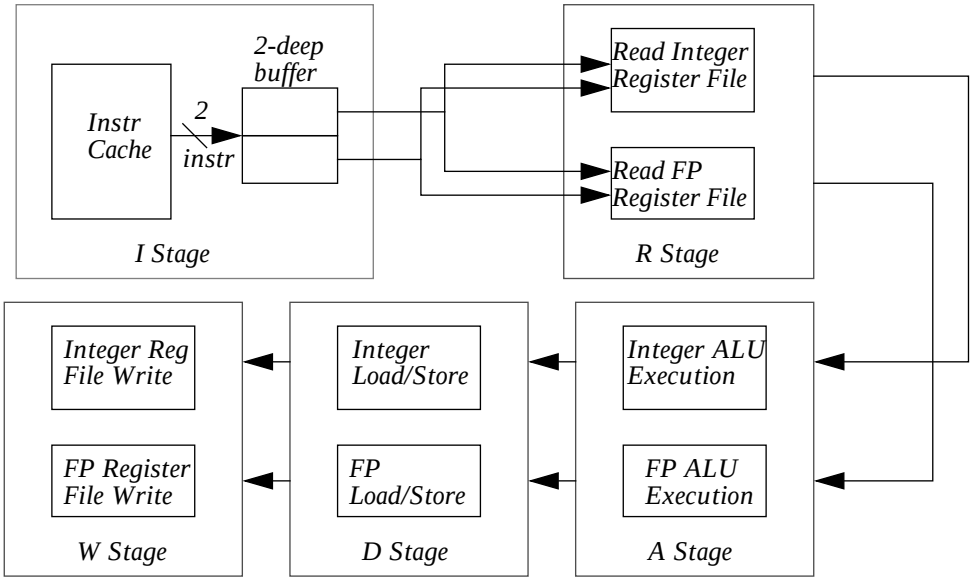


Figure 3.1 Dual Issue Mechanism

I - Stage

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

R - Stage

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

A - Stage

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

D - Stage

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

W - Stage

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

Introduction

The IDT R5000 processor provides a full-featured memory management unit (MMU) which uses an on-chip translation lookaside buffer (TLB) to translate virtual addresses into physical addresses.

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

Translation Lookaside Buffer (TLB)

Mapped virtual addresses are translated into physical addresses using an on-chip TLB.[1] The TLB is a fully associative memory that holds 48 entries, which provide mapping to 48 odd/even page pairs (96 pages). When address mapping is indicated, each TLB entry is checked simultaneously for a match with the virtual address that is extended with an ASID stored in the *EntryHi* register.

The address mapped to a page ranges in size from 4 Kbytes to 16 Mbytes, in multiples of 4—that is, 4K, 16K, 64K, 256K, 1M, 4M, 16M.

Hits and Misses

If there is a virtual address match, or hit, in the TLB, the physical page number is extracted from the TLB and concatenated with the offset to form the physical address.

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

Multiple Matches

The R5000 processor does not provide any detection of shutdown mechanism for multiple matches in the TLB. The result of this condition is undefined, and software is expected to never allow this to occur.

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.

Virtual Address Space

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

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

1. There are virtual-to-physical address translations that occur outside of the TLB. For example, addresses in *the kseg0* and *kseg1* spaces are unmapped translations. In these spaces the physical address is 0x000 0000 0 11 VA[28:0].

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

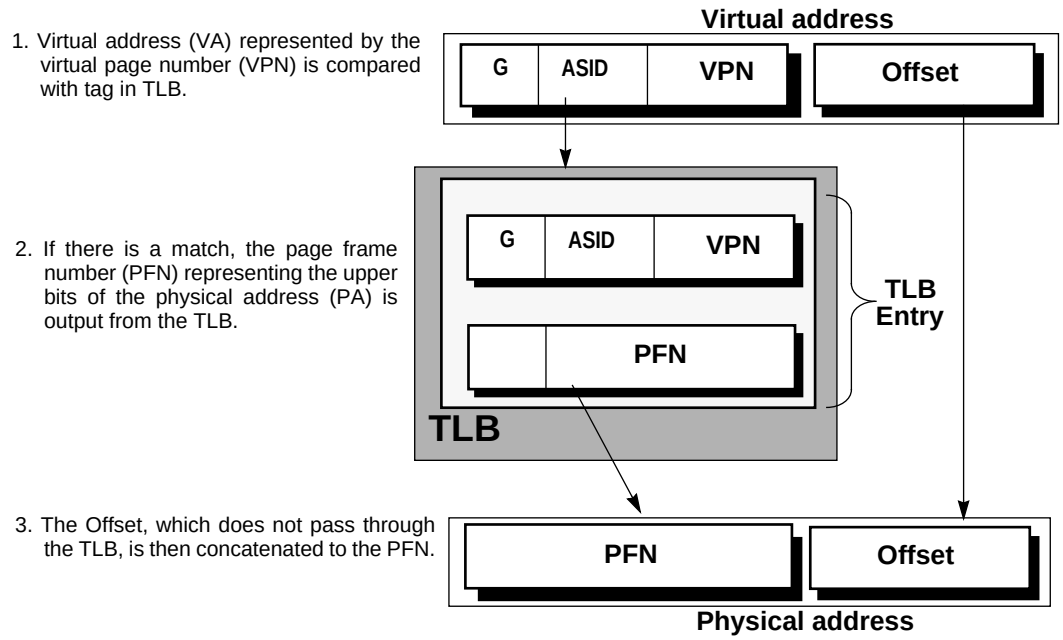


Figure 4.1 Overview of a Virtual-to-Physical Address Translation

As shown in Figures 11 and 12, the virtual address is extended with an 8-bit address space identifier (ASID), which reduces the frequency of TLB flushing when switching contexts. This 8-bit ASID is in the CP0 *EntryHi* register. The *Global* bit (*G*) is in the *EntryLo0* and *EntryLo1* registers.

Physical Address Space

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

Virtual-to-Physical Address Translation

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

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

This match is referred to as a *TLB hit*. If there is no match, a TLB Miss exception is taken by the processor and software is allowed to refill the TLB from a page table of virtual/physical addresses in memory.

If there is a virtual address match in the TLB, the physical address is output from the TLB and concatenated with the *Offset*, which represents an address within the page frame space. The *Offset* does not pass through the TLB.

Virtual-to-physical translation is described in greater detail throughout the remainder of this chapter.

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

32-bit Mode Virtual Address Translation

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

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

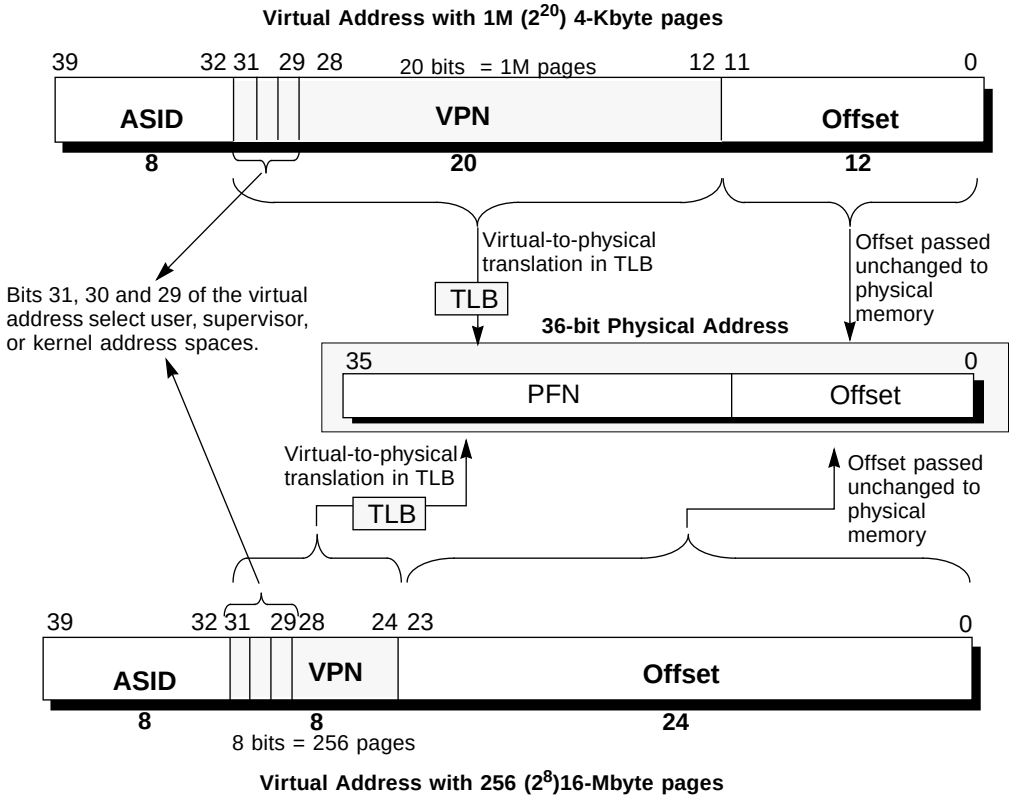


Figure 4.2 32-bit Mode Virtual Address Translation

64-bit Mode Virtual Address Translation

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

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

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

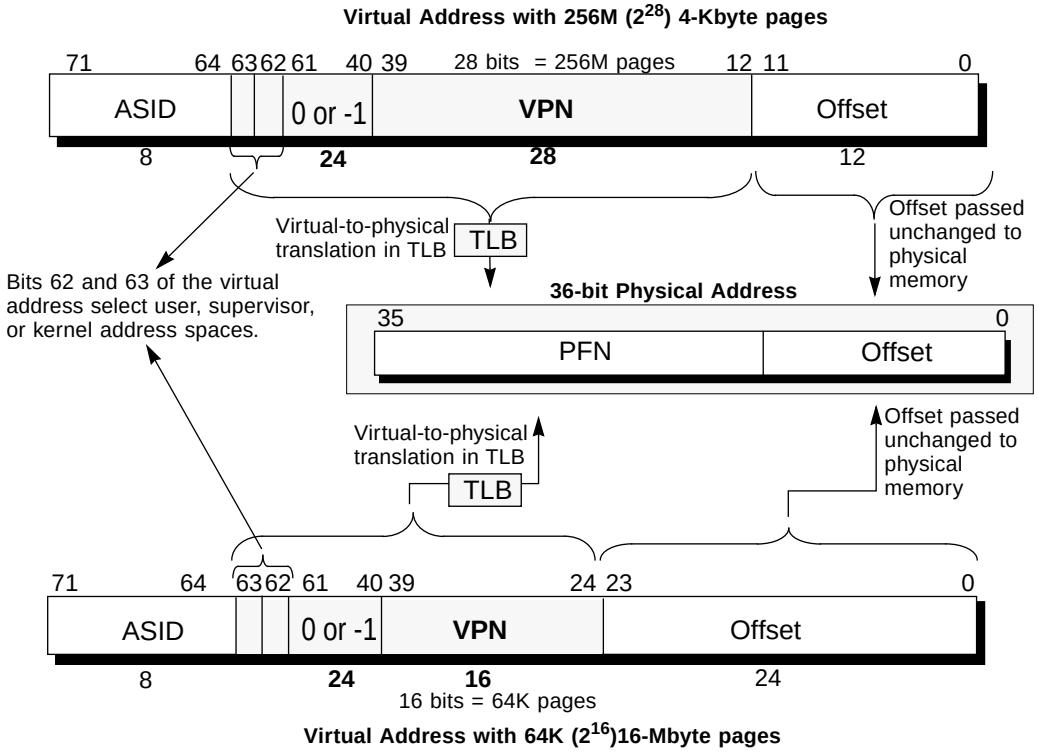


Figure 4.3 64-bit Mode Virtual Address Translation

Operating Modes

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

- User mode
- Supervisor mode
- Kernel mode

These modes are described in the next three sections.

User Mode Operations

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

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

Figure 4.4 shows User mode virtual address space.

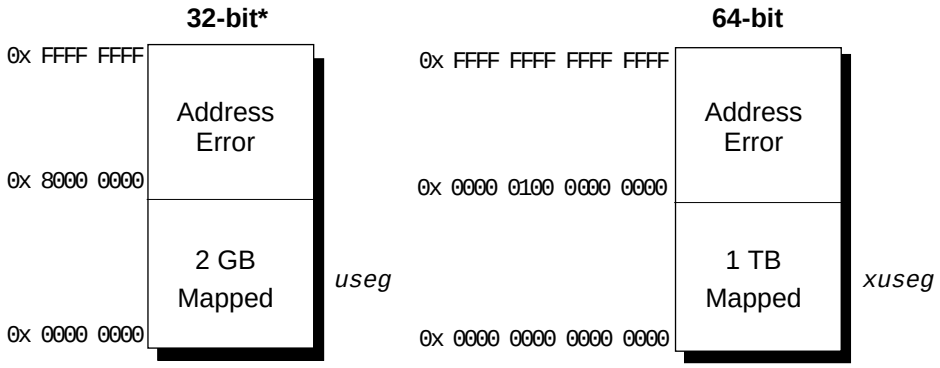


Figure 4.4 User Mode Virtual Address Space

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

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

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

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

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

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

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

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

32-bit User Mode (useg)

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

All valid User mode virtual addresses have their most-significant bit cleared to 0; any attempt to reference an address with the most-significant bit set while in User mode causes an Address Error exception.

The system maps all references to *useg* through the TLB, and bit settings within the TLB entry for the page determine the cacheability of a reference.

64-bit User Mode (*xuseg*)

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

All valid User mode virtual addresses have bits 63:40 equal to 0; an attempt to reference an address with bits 63:40 not equal to 0 causes an Address Error exception.

Supervisor Mode Operations

Supervisor mode is designed for layered operating systems in which a true kernel runs in Kernel mode, and the rest of the operating system runs in Supervisor mode.

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

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

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

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

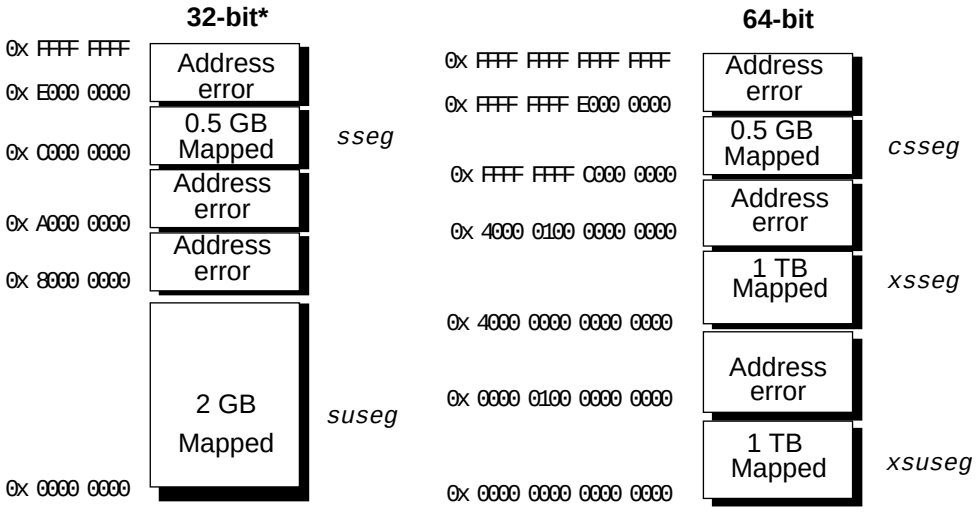


Figure 4.5 Supervisor Mode Address Space

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

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

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

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

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

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

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

This mapped space begins at virtual address 0xC000 0000 and runs through 0xDFFF FFFF.

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

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

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

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

In Supervisor mode, when *SX* = 1 in the *Status* register and bits 63:62 of the virtual address are set to 01_2, the *xsseg* current supervisor virtual address space is selected. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

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

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

In Supervisor mode, when *SX* = 1 in the *Status* register and bits 63:62 of the virtual address are set to 11_2, the *csseg* separate supervisor virtual address space is selected. Addressing of the *csseg* is compatible with addressing *sseg* in 32-bit mode. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space begins at virtual address 0xFFFF FFFF C000 0000 and runs through 0xFFFF FFFF DFFF FFFF.

Kernel Mode Operations

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

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

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

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

The processor enters Kernel mode whenever an exception is detected and it remains in Kernel mode until an Exception Return (ERET) instruction is executed. The ERET instruction restores the processor to the mode existing prior to the exception.

Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 4.6. Table 4.3 lists the characteristics of the 32-bit kernel mode segments, and Table 4.4 lists the characteristics of the 64-bit kernel mode segments.

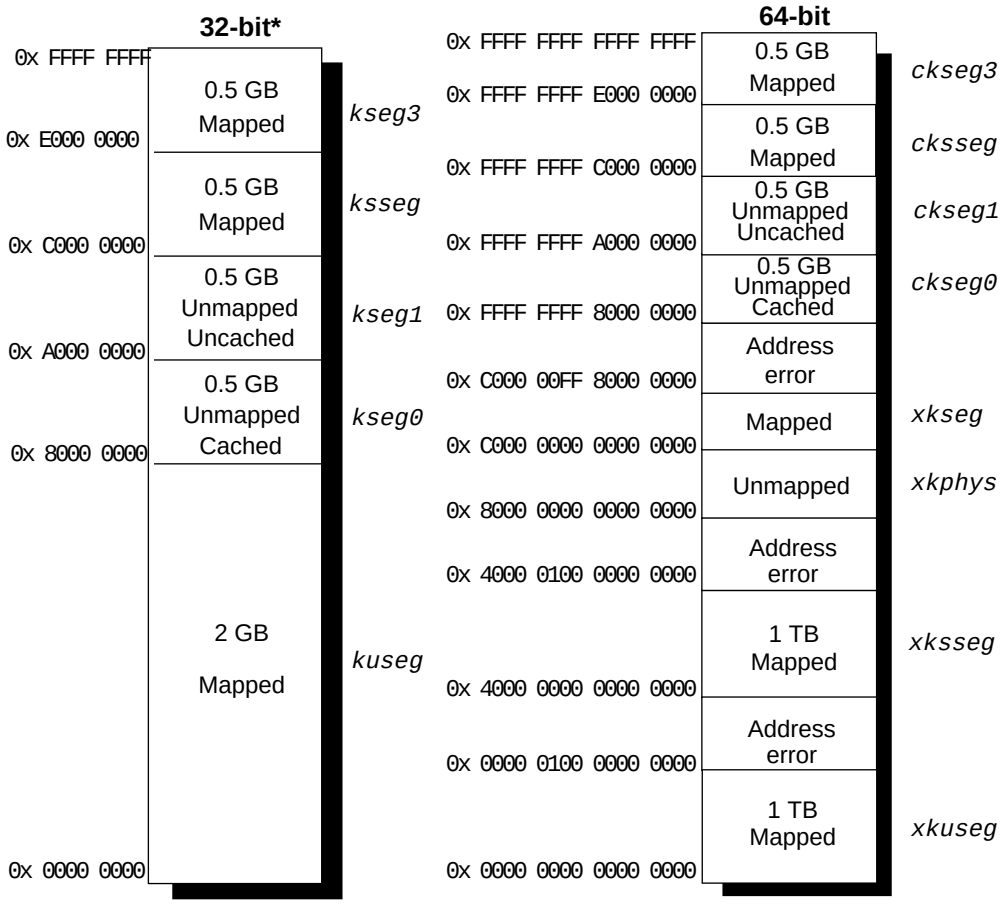


Figure 4.6 Kernel Mode Address Space

Table 4.3 32-bit Kernel Mode Segments

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

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

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

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

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

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

In Kernel mode, when *KX* = 0 in the *Status* register and the most-significant three bits of the 32-bit virtual address are 101_2, 32-bit *kseg1* virtual address space is selected; it is the 2^{29}-byte (512-Mbyte) kernel physical space.

References to *kseg1* are not mapped through the TLB; the physical address selected is defined by subtracting 0xA000 0000 from the virtual address.

Caches are disabled for accesses to these addresses, and physical memory (or memory-mapped I/O device registers) are accessed directly.

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

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

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

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

Table 4.4 64-bit Kernel Mode Segments

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

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

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

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

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

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

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

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63:62 of the 64-bit virtual address are 10_2, the *xkphys* virtual address space is selected; it is a set of eight 2^{36}-byte kernel physical spaces. Accesses with address bits 58:36 not equal to 0 cause an address error.

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

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	0x9800 0000 0000 0000
4-7	Reserved	0xA000 0000 0000 0000

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

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

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

64-bit Kernel Mode, Compatibility Spaces

In Kernel mode, when *KX* = 1 in the *Status* register, bits 63:62 of the 64-bit virtual address are 11_2, and bits 61:31 of the virtual address equal –1. The lower two bytes of address, as shown in figure 15, 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 *ksse*g.
- *ckseg3*. This 64-bit virtual address space is kernel virtual space, compatible with the 32-bit address model *kseg3*.

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 4.7 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.

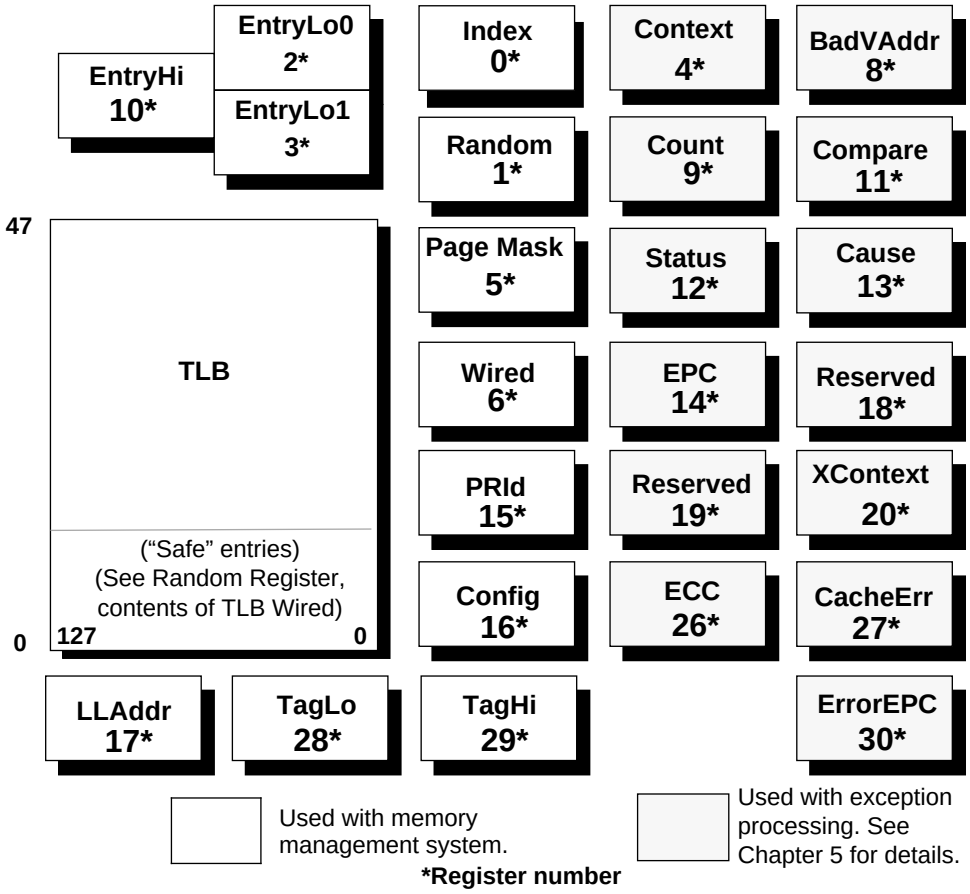


Figure 4.7 CP0 Registers and the TLB

Format of a TLB Entry

Figure 4.8 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.

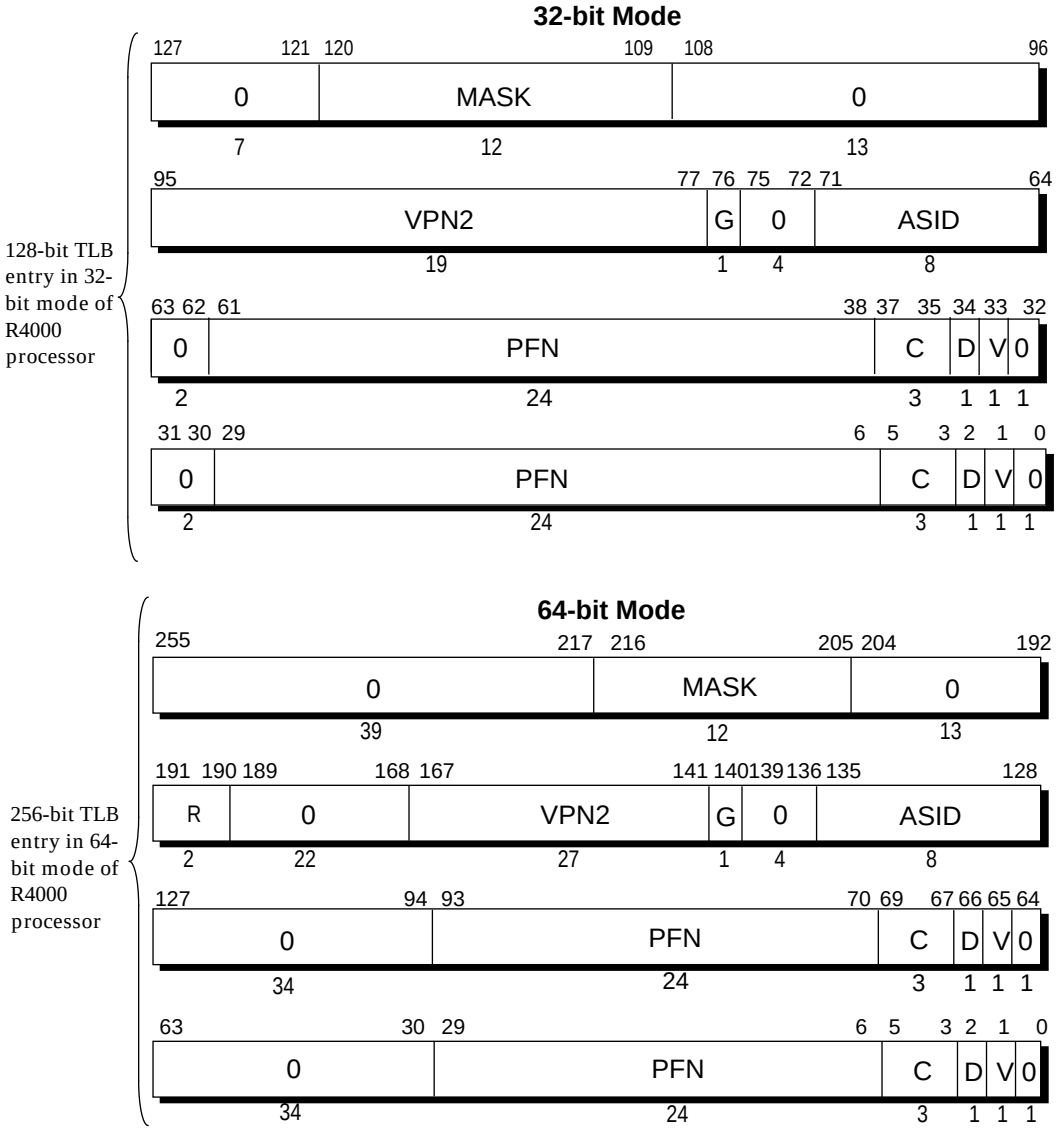
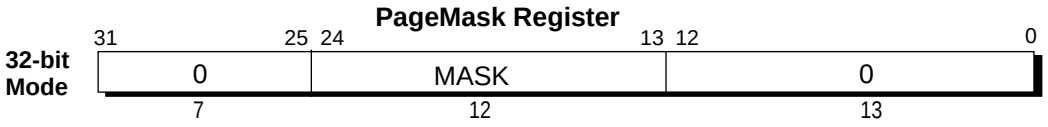
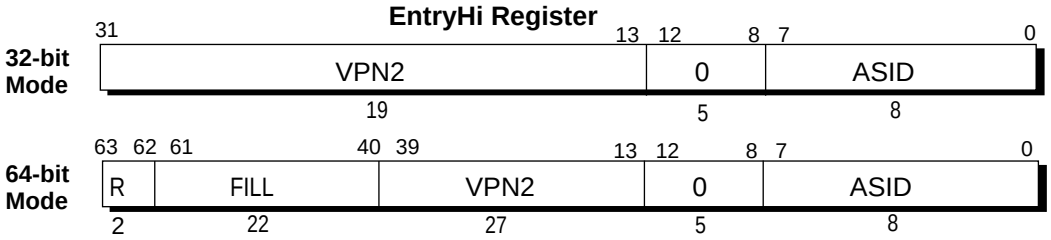


Figure 4.8 Format of a TLB Entry

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. Figure 4.9 and Figure 4.10 describe the TLB entry fields shown in Figure 4.8.

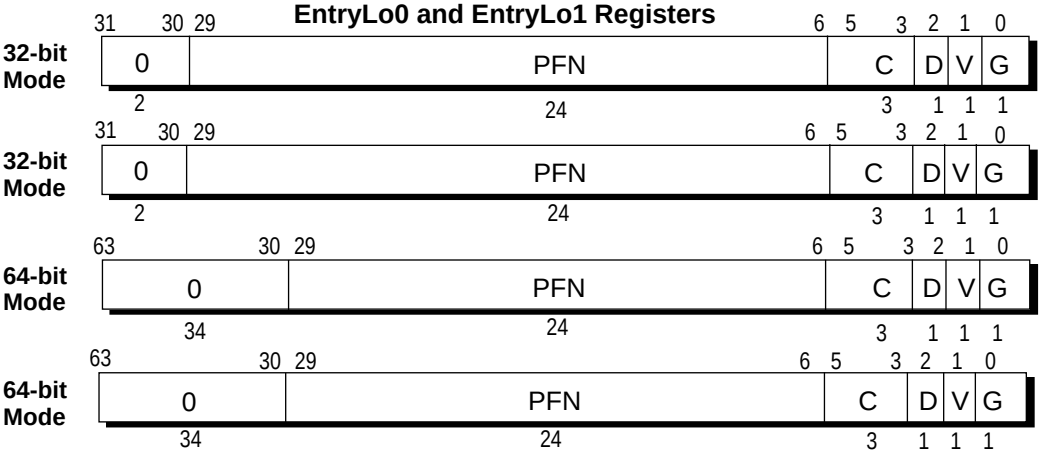


Mask .... Page comparison mask.
0..... Reserved. Must be written as zeroes, and returns zeroes when read.



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

Figure 4.9 Fields of the PageMask and EntryHi Registers



PFN.....Page frame number; the upper bits of the physical address.
C.....Specifies the TLB page coherency attribute; see Table 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 4.10 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, write-back
4 - 7	Reserved

CP0 Registers

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

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

Index Register (0)

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

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

Figure 4.11 shows the format of the *Index* register; Table 4.7 describes the *Index* register fields.

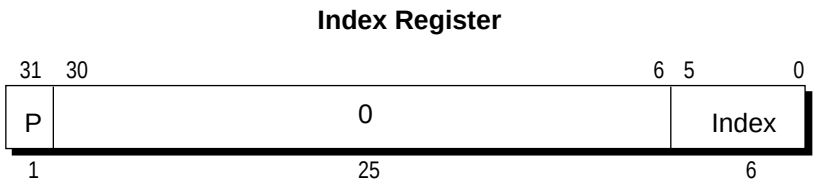


Figure 4.11 Index Register

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.

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 4.12 shows the format of the *Random* register. Table 4.8 describes the *Random* register fields.

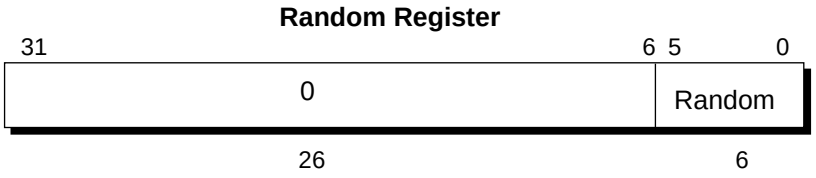


Figure 4.12 Random Register

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.

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 4.10 shows the format of these registers.

PageMask Register (5)

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

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

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 4.13. Wired entries are fixed, nonreplaceable entries, which cannot be overwritten by a TLB write operation. Random entries can be overwritten.

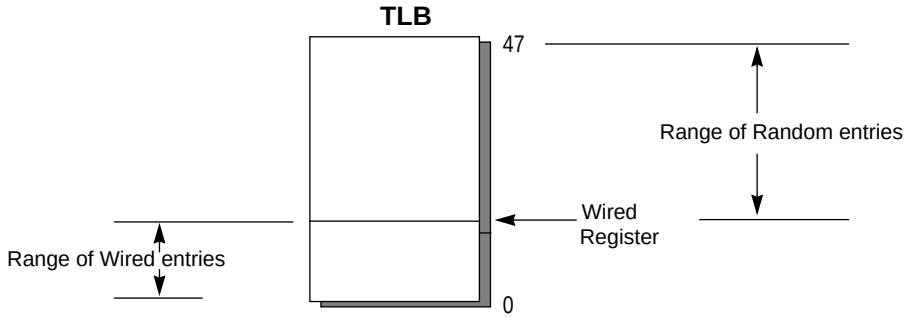


Figure 4.13 Wired Register Boundary

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 4.14 shows the format of the *Wired* register; Table 4.10 describes the register fields.

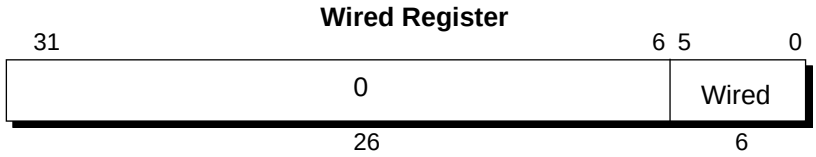


Figure 4.14 Wired Register

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.

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.

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 4.15 shows the format of the *PRId* register; Table 4.11 describes the *PRId* register fields.

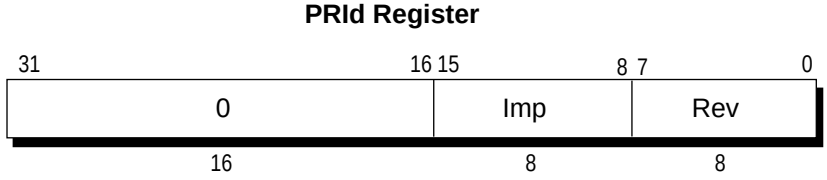


Figure 4.15 Processor Revision Identifier Register Format

Table 4.11 PRId Register Fields

Field	Description
Imp	Implementation number Imp=0x23
Rev	Revision number
0	Reserved. Must be written as zeroes, and returns zeroes when read.

The low-order byte (bits 7:0) of the *PRId* register is interpreted as a revision number, and the high-order byte (bits 15:8) is interpreted as an implementation number. The implementation number of the R5000 processor is 0x23. The content of the high-order halfword (bits 31:16) of the register are reserved.

The revision number is stored as a value in the form *y.x*, where *y* is a major revision number in bits 7:4 and *x* is a minor revision number in bits 3:0.

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

Config Register (16)

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

Some configuration options, as defined by *Config* bits 31:6, are set by the hardware during reset and are included in the *Config* register as read-only status bits for the software to access. Other configuration options are read/write (as indicated by *Config* register bits 5:0) and controlled by software; on reset these fields are undefined.

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

Figure 4.16 shows the format of the *Config* register; Table 4.12 describes the *Config* register fields.

Config Register

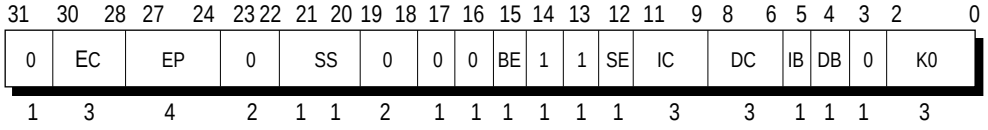


Figure 4.16 Config Register Format

Table 4.12 Config Register Fields

Field	Description
EC	System clock ratio: 0 → processor clock frequency divided by 2 1 → processor clock frequency divided by 3 2 → processor clock frequency divided by 4 3 → processor clock frequency divided by 5 4 → processor clock frequency divided by 6 5 → processor clock frequency divided by 7 6 → processor clock frequency divided by 8 7 → Reserved

Field	Description
EP	Transmit data pattern (pattern for write-back data): 0 → D Doubleword every cycle 1 → DDxDDx 2 Doublewords every 3 cycles 2 → DDxxDDxx 2 Doublewords every 4 cycles 3 → DxDxDxDx 2 Doublewords every 4 cycles 4 → DDxxxDDxxx 2 Doublewords every 5 cycles 5 → DDxxxxDDxxxx 2 Doublewords every 6 cycles 6 → DxxDxxDxxDxx 2 Doublewords every 6 cycles 7 → DDxxxxxxDDxxxxxx 2 Doublewords every 8 cycles 8 → DxxxDxxxDxxxDxxx 2 Doublewords every 8 cycles
SS	Secondary Cache Size 00 → 512 KByte 01 → 1 MByte 10 → 2 MByte 11 → None
BE	Big Endian Mode: 0 → Little Endian 1 → Big Endian
SE	Secondary Cache Enable 0 → Disabled 1 → Enabled
IC	Primary I-cache Size (I-cache size = 2^{12+IC} bytes). In the R5000 processor, this is set to 32 Kbytes.
DC	Primary D-cache Size (D-cache size = 2^{12+DC} bytes). In the R5000 processor, this is set to 32 Kbytes.
IB	Primary I-cache line size. In the R5000 processor, this is set to 32 bytes. 0 → 16 bytes 1 → 32 bytes
DB	Primary D-cache line size. In the R5000 processor, this is set to 32 bytes. 0 → 16 bytes 1 → 32 bytes
K0	*kseg0* coherency algorithm (see *EntryLo0* and *EntryLo1* registers and the *C* field of Table 4.6)

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

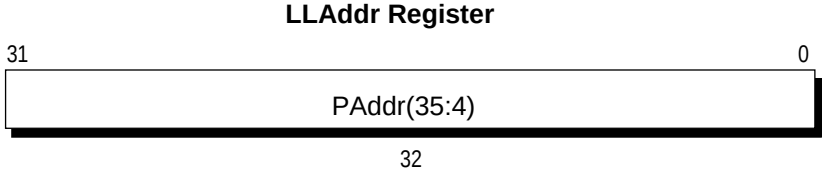


Figure 4.17 LLAddr Register Format

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

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

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

Figure 4.18 shows the format of these registers for primary cache operations. Figure 4.19 shows the format of these registers for secondary cache operations.

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

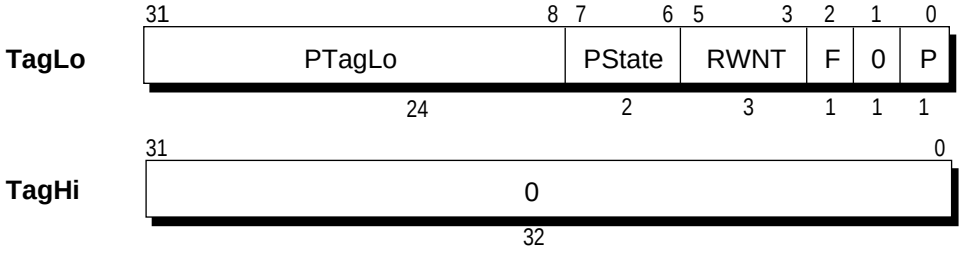


Figure 4.18 TagLo and TagHi Register (P-cache) Formats

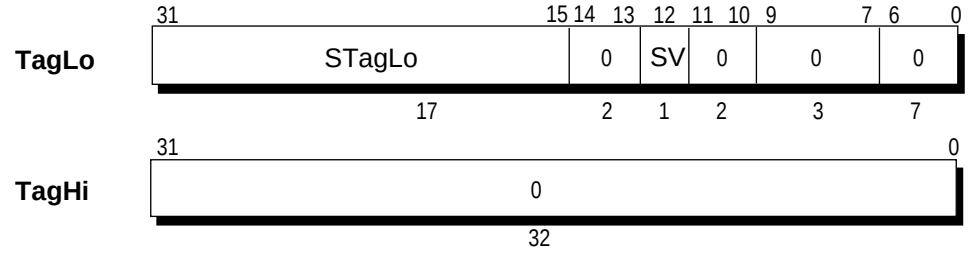


Figure 4.19 TagLo and TagHi Register (S-cache) Formats

Table 4.13 Cache Tag Register Fields

Field	Description
PTagLo	Specifies the physical address bits 35:12
PState	Specifies the primary cache state
P	Specifies the primary tag even parity bit
STagLo	Specifies the physical address bits 35:19
SV	Specifies the Valid bit for secondary cache
0	Reserved. Must be written as zeroes, and returns zeroes when read.

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 4.20 illustrates the TLB address translation process.

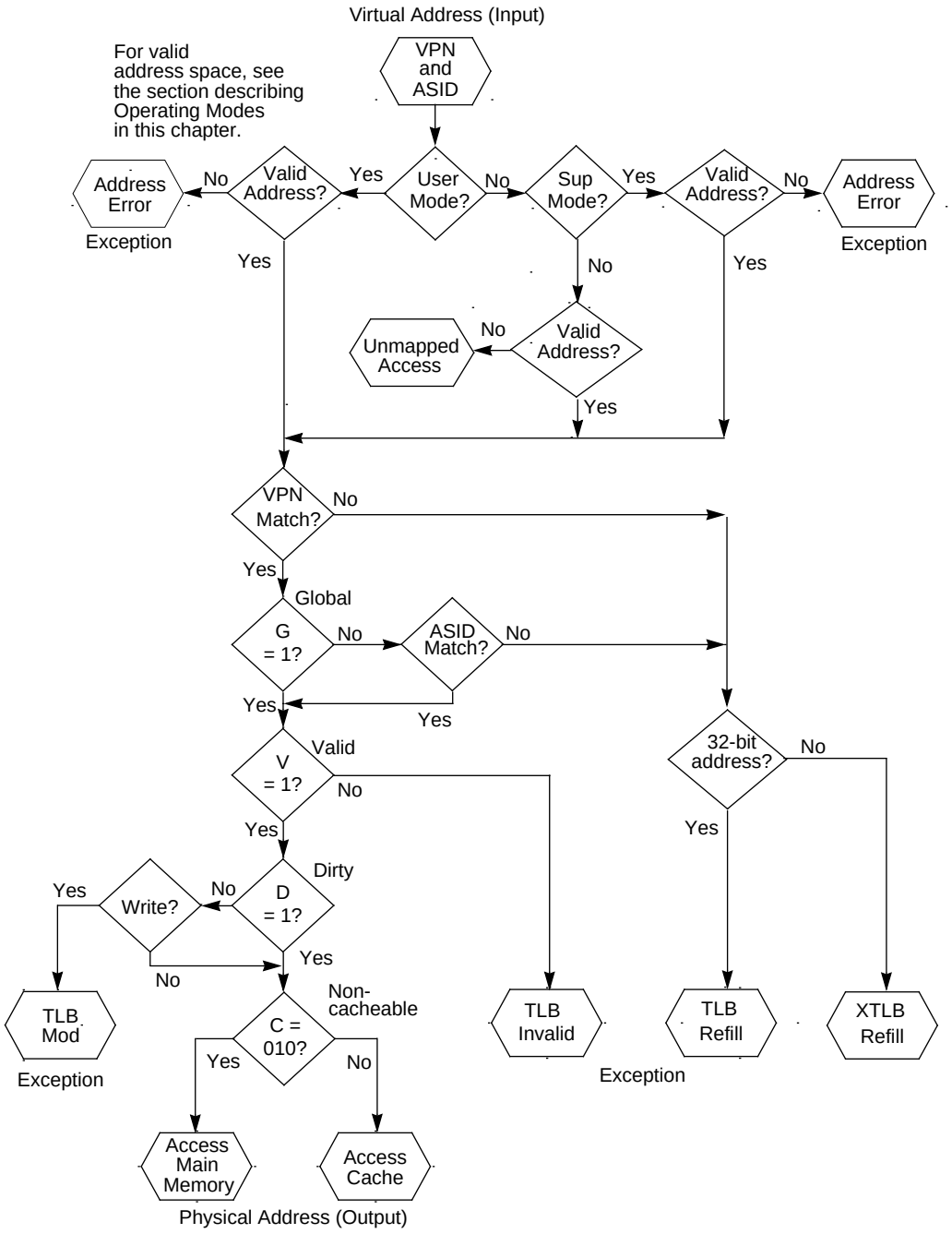


Figure 4.20 TLB Address Translation

TLB Misses

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

TLB Instructions

Table 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

Secondary Cache Operations

The CACHE instruction defines two operations for the secondary cache: *index load tag* and *index store tag*. The following orientation of the index bits determine the type of operation:

Index Bits [17:16] equal to 11b (3h) specifies the secondary cache.

Index bits [20:18] equal to 001b (1h) specifies the index load tag. The index load tag reads the secondary cache for the specified index and places it into the TagLo CP0 register.

Index bits [20:18] equal to 010b (2h) specify the index store tag. The index store tag writes the secondary cache for the specified index from the physical address generated by the CACHE instruction.

Index bits [20:18] equal to 000b (0h) generates a valid clear sequence to flush the entire cache in one operation.

Index bits [20:18] equal to 101b (5h) generates a cache page invalidate instruction to flush 128 lines of the cache in one operation with the tag value from the TagLo CP0 register. The index for the cache page invalidate must be page aligned.

Interrupts are deferred until a cache page invalidate instruction completes (up to 512 processor clocks for a SysClock ratio of 4).

TagLo[12] is the valid bit and TagLo[31:15] is the tag for all secondary cache operations.

Secondary Cache Software Enable

The secondary cache may be enabled or disabled by software control via CP0 config register bit 12 (SE). When the SE bit is set (1) the secondary cache is enabled. When the SE bit is cleared (0) the secondary cache is disabled. 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 and software must explicitly invalidate the entire secondary cache before using it.

Introduction

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

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.

Exception Processing Registers

This section describes the CP0 registers that are used in exception processing. Table 5.1 lists these registers, along with their number—each register has a unique identification number that is referred to as its *register number*. For instance, the *ECC* register is register number 26. The remaining CP0 registers are used in memory management.

Software examines the CP0 registers during exception processing to determine the cause of the exception and the state of the CPU at the time the exception occurred. The registers in Table 5.1 are used in exception processing, and are described in the sections that follow.

Table 5.1 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
WatchLo (Memory Reference Trap Address Low)	18

Register Name	Reg. No.
WatchHi (Memory Reference Trap Address High)	19
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, however; there may be some delay before a value written by one instruction is available to following instructions.

Context Register (4)

The *Context* register is a read/write register containing the pointer to an entry in the page table entry (PTE) array; this array is an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the CPU loads the TLB with the missing translation from the PTE array. Normally, the operating system uses the *Context* register to address the current page map which resides in the kernel-mapped segment, *kseg3*. The *Context* register duplicates some of the information provided in the *BadVAddr* register, but the information is arranged in a form that is more useful for a software TLB exception handler. Figure 5.1 shows the format of the *Context* register; Table 5.2 describes the *Context* register fields.

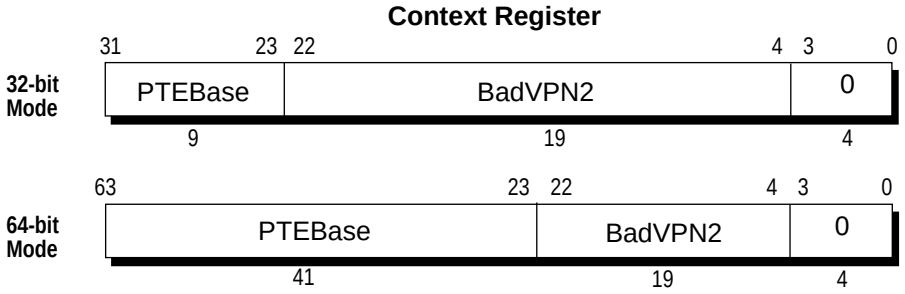


Figure 5.1 Context Register Format

Table 5.2 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.

Bad Virtual Address Register (BadVAddr) (8)

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

Figure 5.2 shows the format of the *BadVAddr* register.

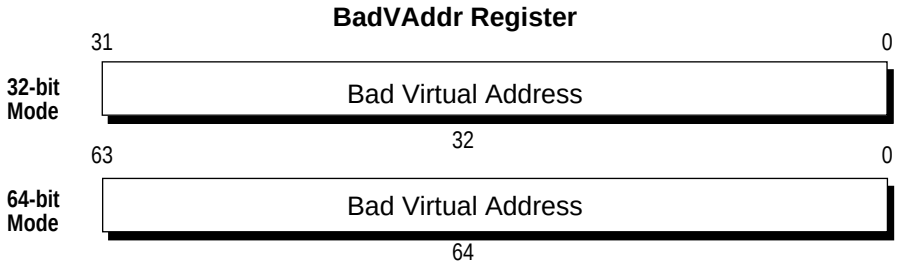


Figure 5.2 BadVAddr Register Format

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

Count Register (9)

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

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

Figure 5.3 shows the format of the *Count* register.

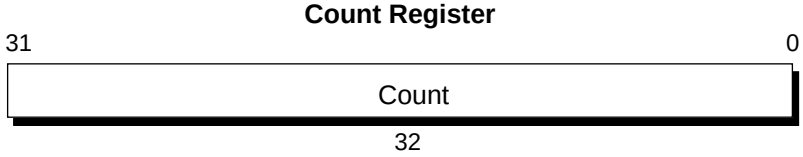


Figure 5.3 Count Register Format

Compare Register (11)

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

When the value of the *Count* register equals the value of the *Compare* register, interrupt bit *IP(7)* in the *Cause* register is set. This causes an interrupt as soon as the interrupt is enabled.

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

For diagnostic purposes, the *Compare* register is a read/write register. In normal use however, the *Compare* register is write-only. Figure 5.4 shows the format of the *Compare* register.

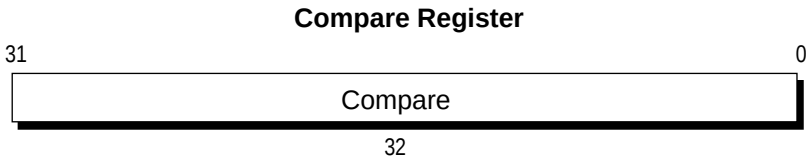


Figure 5.4 Compare Register Format

Status Register (12)

The *Status* register (SR) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. The following list describes the more important *Status* register fields; Figures 34 and 35 show the format of the entire register, including descriptions of the fields. Some of the important fields include:

- The 8-bit *Interrupt Mask* (*IM*) field controls the enabling of eight interrupt conditions. Interrupts must be enabled before they can 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 soft3ware interrupt masks, while IM[7:2] correspond to Int[5:0].
- The 4-bit *Coprocessor Usability* (*CU*) field controls the usability of 4 possible coprocessors. Regardless of the *CU0* bit setting, CP0 is always usable in Kernel mode. For all other cases, an access to an unusable coprocessor causes an exception.
- The 9-bit *Diagnostic Status* (*DS*) field is used for self-testing, and checks the cache and virtual memory system.
- The *Reverse-Endian (RE)* bit, bit 25, reverses the endianness of the machine. The processor can be configured as either little-endian or big-endian at system reset; reverse-endian selection is used in Kernel and Supervisor modes, and in the User mode when the *RE* bit is 0. Setting the *RE* bit to 1 inverts the User mode endianness.

Status Register Format

Figure 5.5 shows the format of the *Status* register. Table 5.3 describes the *Status* register fields. Figure 5.6 and Table 5.4 provide additional information on the *Diagnostic Status* (*DS*) field. All bits in the *DS* field except *TS* are readable and writable.

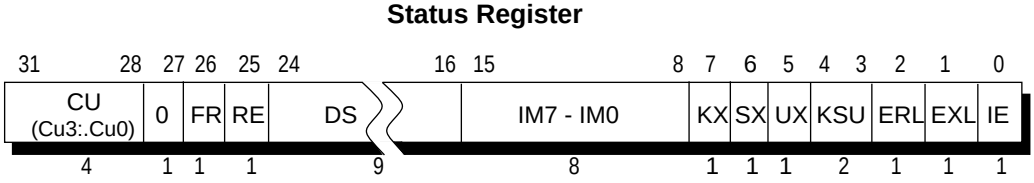


Figure 5.5 Status Register

Table 5.3 Status Register Fields

Field	Description
CU	Controls the usability of each of the four coprocessor unit numbers. CP0 is always usable when in Kernel mode, regardless of the setting of the CU_0 bit. Setting CU_3 enables the MIPS IV instruction set, 1 → usable 0 → unusable
0	Reserved. Set to 0.
FR	Enables additional floating-point registers 0 → 16 registers 1 → 32 registers
RE	*Reverse-Endian* bit, valid in User mode.
DS	*Diagnostic Status* field (see Figure 5.6).
IM	*Interrupt Mask*: controls the enabling of each of the external, internal, and software interrupts. An interrupt is taken if interrupts are enabled, and the corresponding bits are set in both the *Interrupt Mask* field of the *Status* register and the *Interrupt Pending* field of the *Cause* register. 0 → disabled 1→ enabled
KX	Enables 64-bit addressing in Kernel mode. The extended-addressing TLB refill exception is used for TLB misses on kernel addresses. 0 → 32–bit 1 → 64–bit
SX	Enables 64-bit addressing and operations in Supervisor mode. The extended-addressing TLB refill exception is used for TLB misses on supervisor addresses. 0 → 32–bit 1 → 64–bit
UX	Enables 64-bit addressing and operations in User mode. The extended-addressing TLB refill exception is used for TLB misses on user addresses. 0 → 32–bit 1 → 64–bit
KSU	Mode bits 10_2 → User 01_2 → Supervisor 00_2 → Kernel
ERL	Error Level; set by the processor when Reset, Soft Reset, NMI, or Cache Error exception are taken. 0 → normal 1 → error
EXL	Exception Level; set by the processor when any exception other than Reset, Soft Reset, NMI, or Cache Error exception are taken. 0 → normal 1 → exception
IE	Interrupt Enable 0 → disable interrupts 1 → enables interrupts

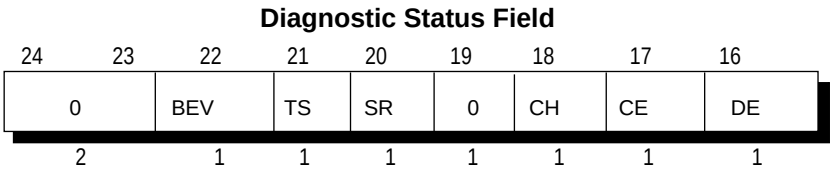


Figure 5.6 Status Register DS Field

Table 5.4 Status Register Diagnostic Status Bits

Bit	Description
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.
CH	Hit (tag match and valid state) or miss indication for last CACHE Hit Invalidate, Hit Write Back Invalidate, Hit Write Back, Hit Set Virtual, or Create Dirty Exclusive for a secondary cache. 0 → miss 1 → hit
CE	Contents of the ECC register set or modify the check bits of the caches when CE = 1; see description of the *ECC* register.
DE	Specifies that cache parity or ECC errors cannot cause exceptions. 0 → parity / ECC remain enabled 1 → disables parity / ECC
0	Reserved. Must be written as zeroes, and returns zeroes when read.

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

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

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 above titled, Operating Modes.

User Address Space Accesses: Access to the user address space is allowed in any of the three operating modes.

Status Register Reset

The contents of the *Status* register are undefined at reset, except for the following bits in the *Diagnostic Status* field:

- *ERL* and *BEV* = 1

The *SR* bit distinguishes between the Reset exception and the Soft Reset exception (caused either by **Reset*** or Nonmaskable Interrupt [NMI]).

Cause Register (13)

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

Figure 5.7 shows the fields of this register. Table 5.5 describes the *Cause* register fields.

All bits in the *Cause* register, with the exception of the *IP(1:0)* bits, are read-only; *IP(1:0)* are used for software interrupts.

Table 5.5 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.
IP	Indicates an interrupt is pending. 1 → interrupt pending 0 → no interrupt
ExcCode	Exception code field (see Table 5.6)
0	Reserved. Must be written as zeroes, and returns zeroes when read.

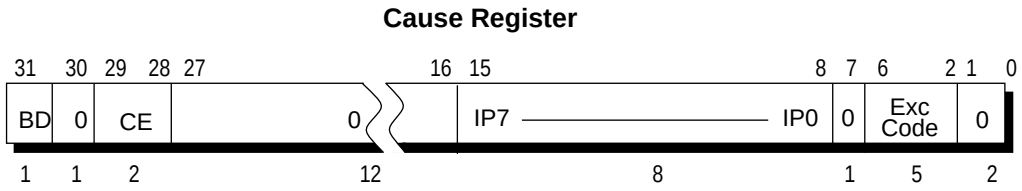


Figure 5.7 Cause Register Format

Table 5.6 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

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.

Figure 5.8 shows the format of the *EPC* register.

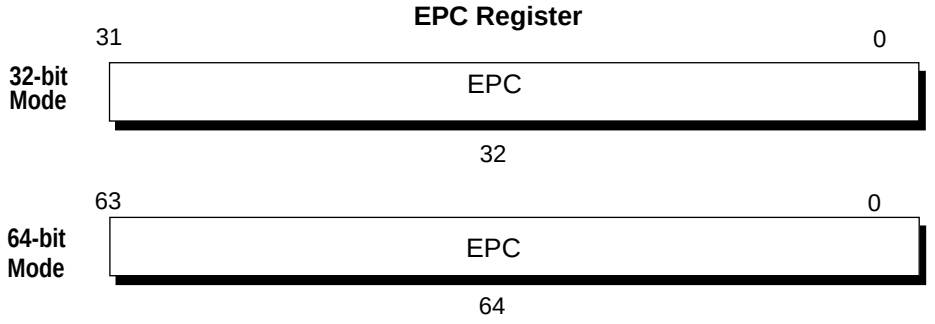


Figure 5.8 EPC Register Format

XContext Register (20)

The read/write *XContext* register contains a pointer to an entry in the page table entry (PTE) array, an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the operating system software loads the TLB with the missing translation from the PTE array. The *XContext* register duplicates some of the information provided in the *BadVAddr* register, and puts it in a form useful for a software TLB exception handler. The *XContext* register is for use with the XTLB refill handler, which loads TLB entries for references to a 64-bit address space, and is included solely for operating system use. The operating system sets the PTE base field in the register, as needed. Normally, the operating system uses the *Context* register to address the current page map, which resides in the kernel-mapped segment *kseg3*. Figure 5.9 shows the format of the *XContext* register; Table 5.7 describes the *XContext* register fields.

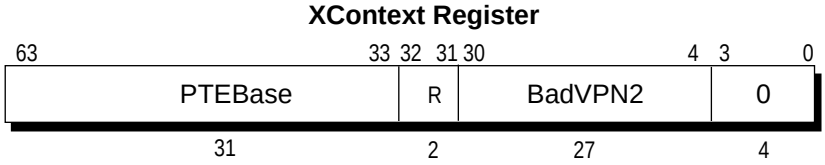


Figure 5.9 XContext Register Format

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 5.7 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.

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 5.10 shows the format of the *ECC* register; Table 5.8 describes the register fields.

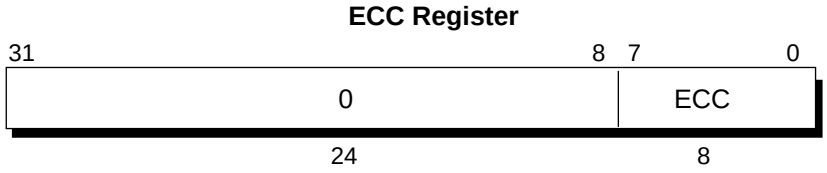


Figure 5.10 ECC Register Format

Table 5.8 ECC Register Fields

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

Cache Error (CacheErr) Register (27)

The 32-bit read-only *CacheErr* register processes ECC errors in the secondary cache and parity errors in the primary cache. 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 5.11 shows the format of the *CacheErr* register and Table 5.9 describes the *CacheErr* register fields.

CacheErr Register

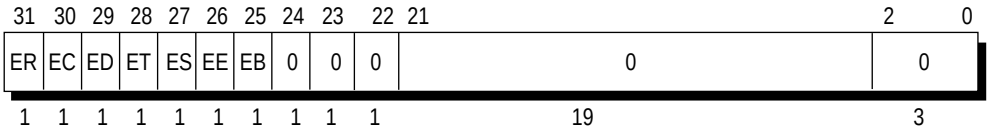


Figure 5.11 CacheErr Register Format

Table 5.9 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 doubleword 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.
EB	This bit is set if a data error occurred in addition to the instruction error (indicated by the remainder of the bits). If so, this requires flushing the data cache after fixing the instruction error.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

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

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

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

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

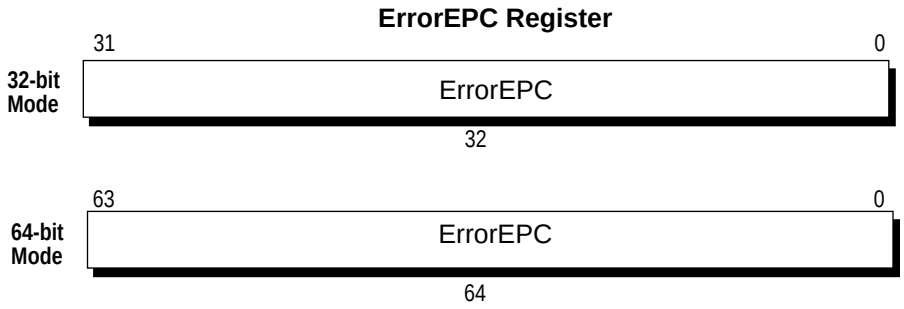


Figure 5.12 ErrorEPC Register Format

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.

Exception Types

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

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

When the *EXL* bit in the *Status* register is 0, either User, Supervisor, or Kernel operating mode is specified by the *KSU* bits in the *Status* register. When the *EXL* bit is a 1, the processor is in Kernel mode.

When the processor takes an exception, the *EXL* bit is set to 1, which means the system is in Kernel mode. After saving the appropriate state, the exception handler typically changes *KSU* to Kernel mode and resets the *EXL* bit back to 0. When restoring the state and restarting, the handler restores the previous value of the *KSU* field and sets the *EXL* bit back to 1.

Returning from an exception, also resets the *EXL* bit to 0.

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

Reset Exception Process

Figure 5.13 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 5.13 Reset Exception Processing

Cache Error Exception Process

Figure 5.14 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      /*What is the BEV bit setting*/
      PC ← 0xFFFF FFFF BFC0 0200 + 0x100  /*Access boot-PROM area*/
   else
      PC ← 0xFFFF FFFF A000 0000 + 0x100  /*Access main memory area*/
   endif
```

Figure 5.14 Cache Error Exception Processing

Soft Reset and NMI Exception Process

Figure 5.15 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 5.15 Soft Reset and NMI Exception Processing

General Exception Process

Figure 5.16 shows the process used for exceptions other than Reset, Soft Reset, NMI, and Cache Error.

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

Figure 5.16 General Exception Processing

Exception Vector Locations

The Reset, Soft Reset, and NMI exceptions are always vectored to location 0xFFFF FFFF BFC0 0000. Addresses for all other exceptions are a combination of a *vector offset* and a *base address*.

The base address is determined by the BEV bit of the *Status* register.

Table 5.10 shows the 64-bit-mode vector base address for all exceptions; the 32-bit mode address is the low-order 32 bits (for instance, the base address for NMI in 32-bit mode is 0xBFC0 0000).

Table 5.11 shows the vector offset added to the base address to create the exception address.

Table 5.10 Exception Vector Base Addresses

BEV Bit	R5000 Processor Vector Base Address
0	0xFFFF FFFF 8000 0000
1	0xFFFF FFFF BFC0 0200

Table 5.11 Exception Vector Offsets

Exception	R5000 Processor Vector Offset
TLB refill, EXL = 0	0x000
XTLB refill, EXL = 0 (X = 64-bit TLB)	0x080
Cache Error	0x100
Others	0x180
Reset, Soft Reset, NMI	none

When **BEV = 0**, the vector base address for the cache error exception changes from *kseg0* (0xFFFF FFFF 8000 0000) to *kseg1* (0xFFFF FFFF A000 0000). This change indicates that the caches are initialized and that the vector can be cached. When **BEV = 1**, the vector base for the cache error exception is 0xFFFF FFFF BFC0 0200. This is an uncached and unmapped space, allowing the exception to bypass the cache and the TLB.

Priority of Exceptions

Table 5.12 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 5.12 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.

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 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

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.

Non Maskable Interrupt (NMI) Exception

Cause

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

Processing

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

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

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

- ErrorEPC register, which contains the restart PC.

- ERL, BEV, and SR bits of the Status Register, each of which is set to 1.

Servicing

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

Address Error Exception

Cause

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

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

This exception is not maskable.

Processing

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

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

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

Servicing

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

TLB Exceptions

Three types of TLB exceptions can occur:

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

- TLB Invalid occurs when a virtual address reference matches a TLB entry that is marked invalid.
- TLB Modified occurs when a store operation virtual address reference to memory matches a TLB entry which is marked valid but is not dirty (the entry is not writable).

The following three sections describe these TLB exceptions.

TLB Refill Exception

Cause

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

Processing

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

When this exception occurs, the *BadVAddr, Context, XContext* and *EntryHi* registers hold the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The *Random* register normally contains a valid location in which to place the replacement TLB entry. The contents of the *EntryLo* register are undefined. The *EPC* register contains the address of the instruction that caused the exception, unless this instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

To service this exception, the contents of the *Context* or *XContext* register are used as a virtual address to fetch memory locations containing the physical page frame and access control bits for a pair of TLB entries. The two entries are placed into the E*ntryLo0/EntryLo1* register; the *EntryHi* and *EntryLo* registers are written into the TLB.

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

TLB Invalid Exception

Cause

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

Processing

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

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

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

Servicing

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

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

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

TLB Modified Exception

Cause

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

Processing

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

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

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

Servicing

The kernel uses the failed virtual address or virtual page number to identify the corresponding access control information. The page identified may or may not permit write accesses; if writes are not permitted, a write protection violation occurs.

If write accesses are permitted, the page frame is marked dirty/writable by the kernel in its own data structures. The TLBP instruction places the index of the TLB entry that must be altered into the *Index* register. The *EntryLo* register is loaded with a word containing the physical page frame and access control bits (with the *D* bit set), and the *EntryHi* and *EntryLo* registers are written into the TLB.

Cache Error Exception

Cause

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

Processing

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

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

If BEV = 0, 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.

Bus Error Exception

Cause

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

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

Processing

The common interrupt vector is used for a Bus Error exception. The *IBE* or *DBE* code in the *ExcCode* field of the *Cause* register is set, signifying whether the instruction (as indicated by the *EPC* register and *BD* bit in the *Cause* register) caused the exception by an instruction reference, load operation, or store operation.

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

Servicing

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

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

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

Integer Overflow Exception

Cause

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

Processing

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

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

Servicing

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

Trap Exception

Cause

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

Processing

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

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

Servicing

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

System Call Exception

Cause

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

Processing

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

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

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

Servicing

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

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

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

Breakpoint Exception

Cause

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

Processing

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

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

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

Servicing

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

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

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

Reserved Instruction Exception

Cause

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

- an attempt is made to execute an instruction with an undefined major opcode (bits 31:26)
- an attempt is made to execute a SPECIAL instruction with an undefined minor opcode (bits 5:0)
- an attempt is made to execute a REGIMM instruction with an undefined minor opcode (bits 20:16)
- an attempt is made to execute 64-bit operations in 32-bit mode when in User or Supervisor modes

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

This exception is not maskable.

Processing

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

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

Servicing

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

Coprocessor Unusable Exception

Cause

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

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

This exception is not maskable.

Processing

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

Servicing

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

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

Floating-Point Exception

Cause

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

Processing

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

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

Servicing

This exception is cleared by clearing the appropriate bit in the *Floating-Point Control/Status* register.

For an unimplemented instruction exception, the kernel should emulate the instruction; for other exceptions, the kernel should pass the exception to the user program that caused the exception.

Interrupt Exception

Cause

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

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

Processing

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

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

Servicing

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

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

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

Exception Handling and Servicing Flowcharts

The remainder of this section contains flowcharts for the following exceptions and guidelines for their handlers:

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

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

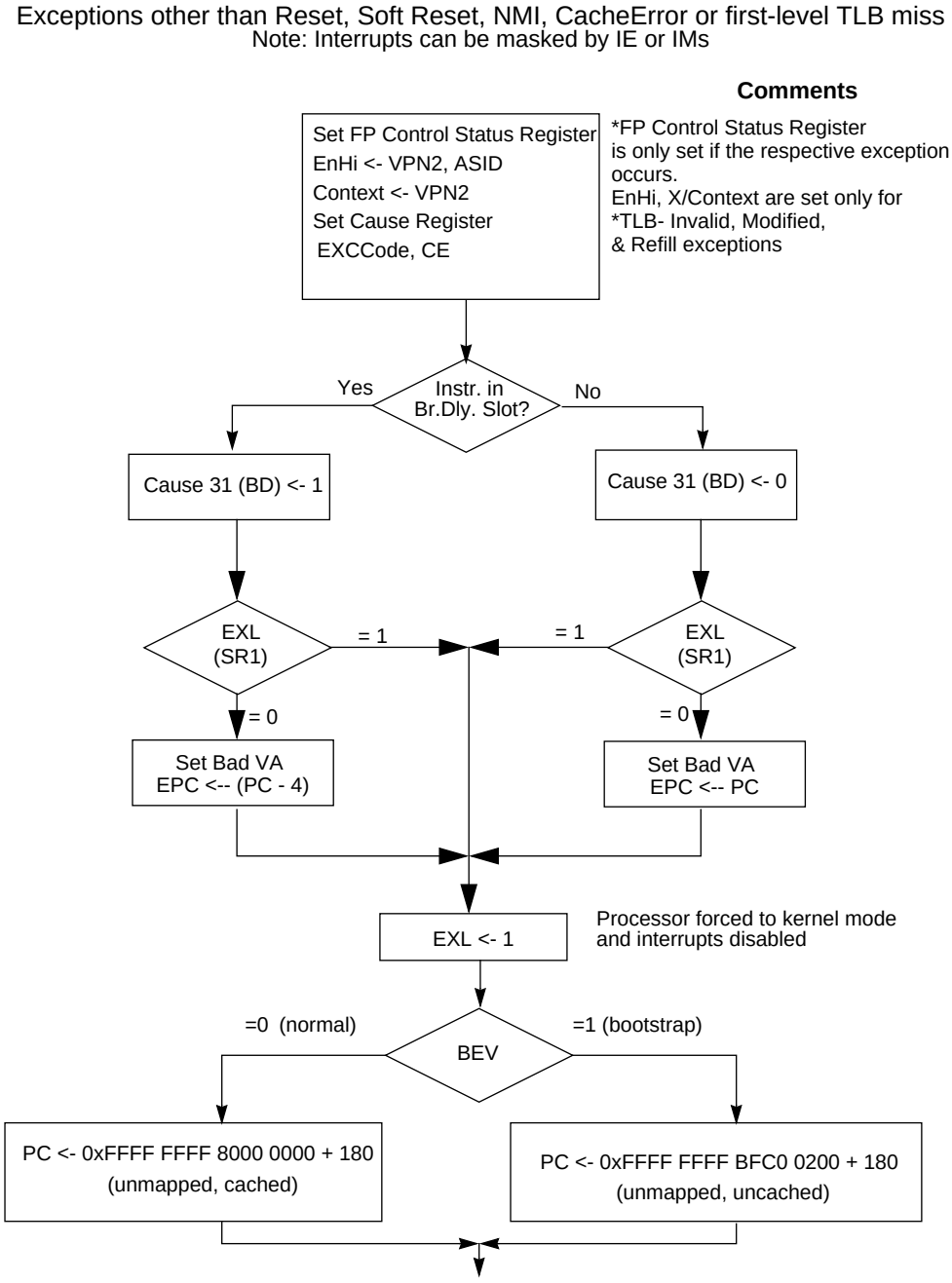


Figure 5.17 General Exception Handler (HW)

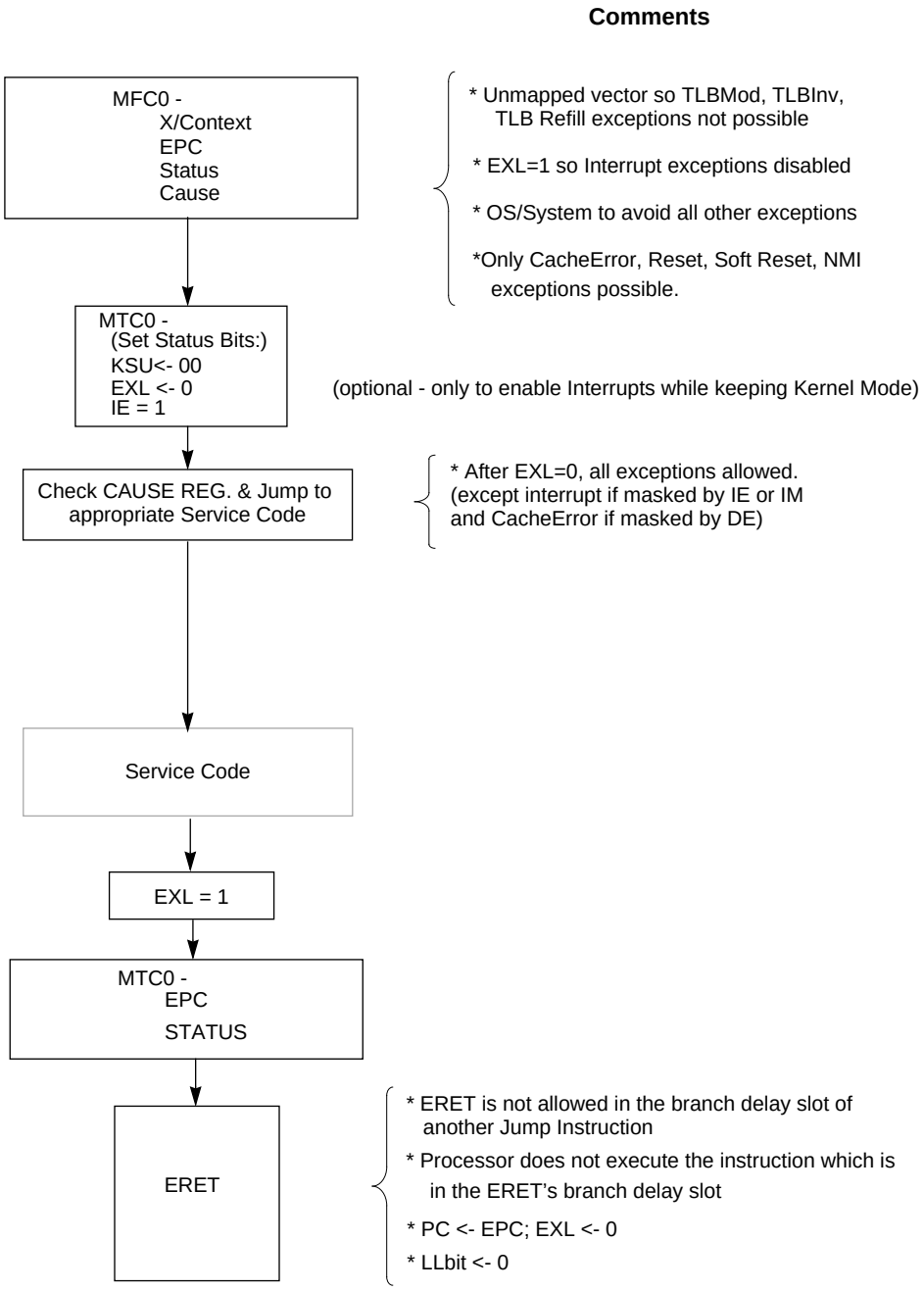


Figure 5.18 General Exception Servicing Guidelines (SW)

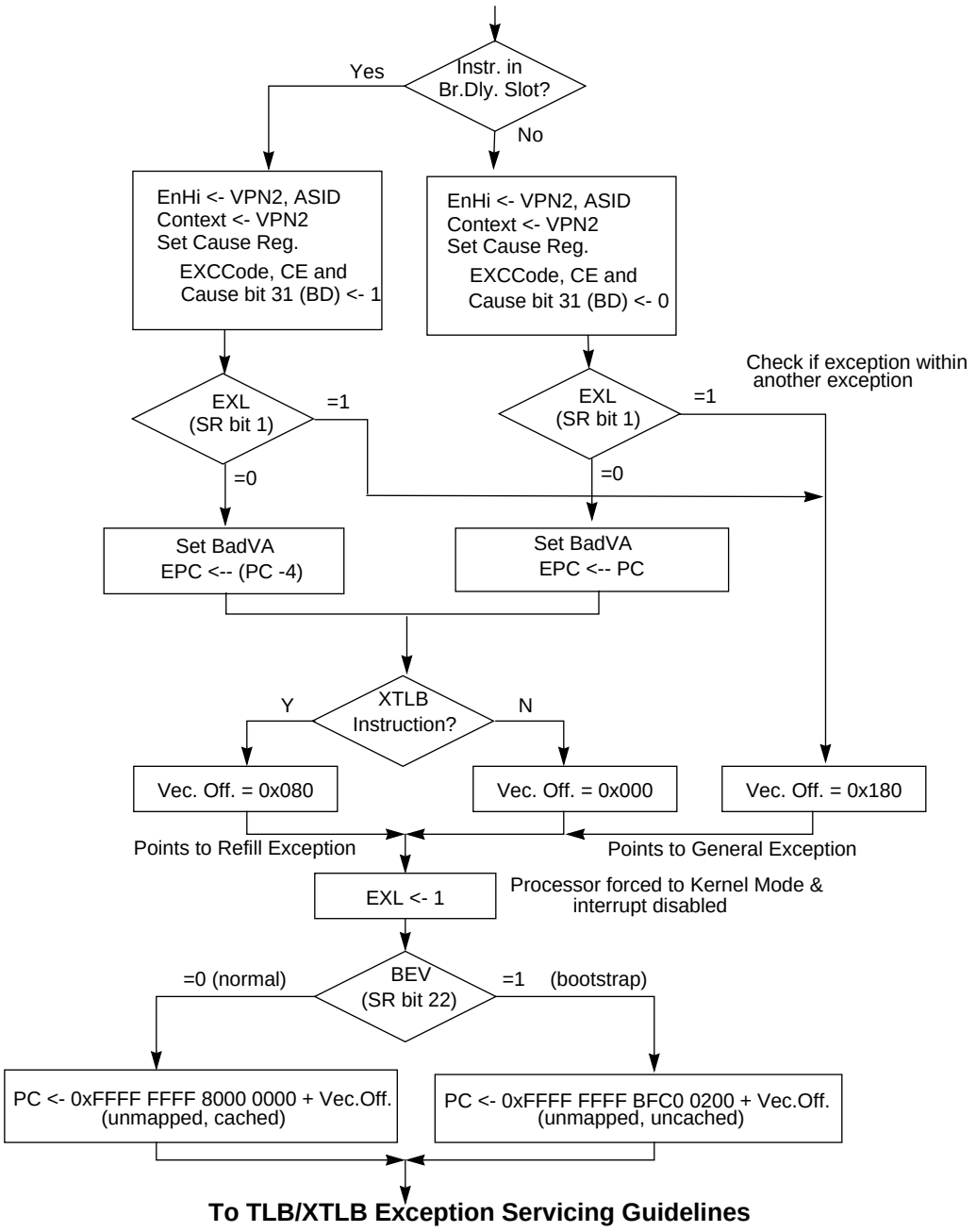


Figure 5.19 TLB/XTLB Miss Exception Handler (HW)

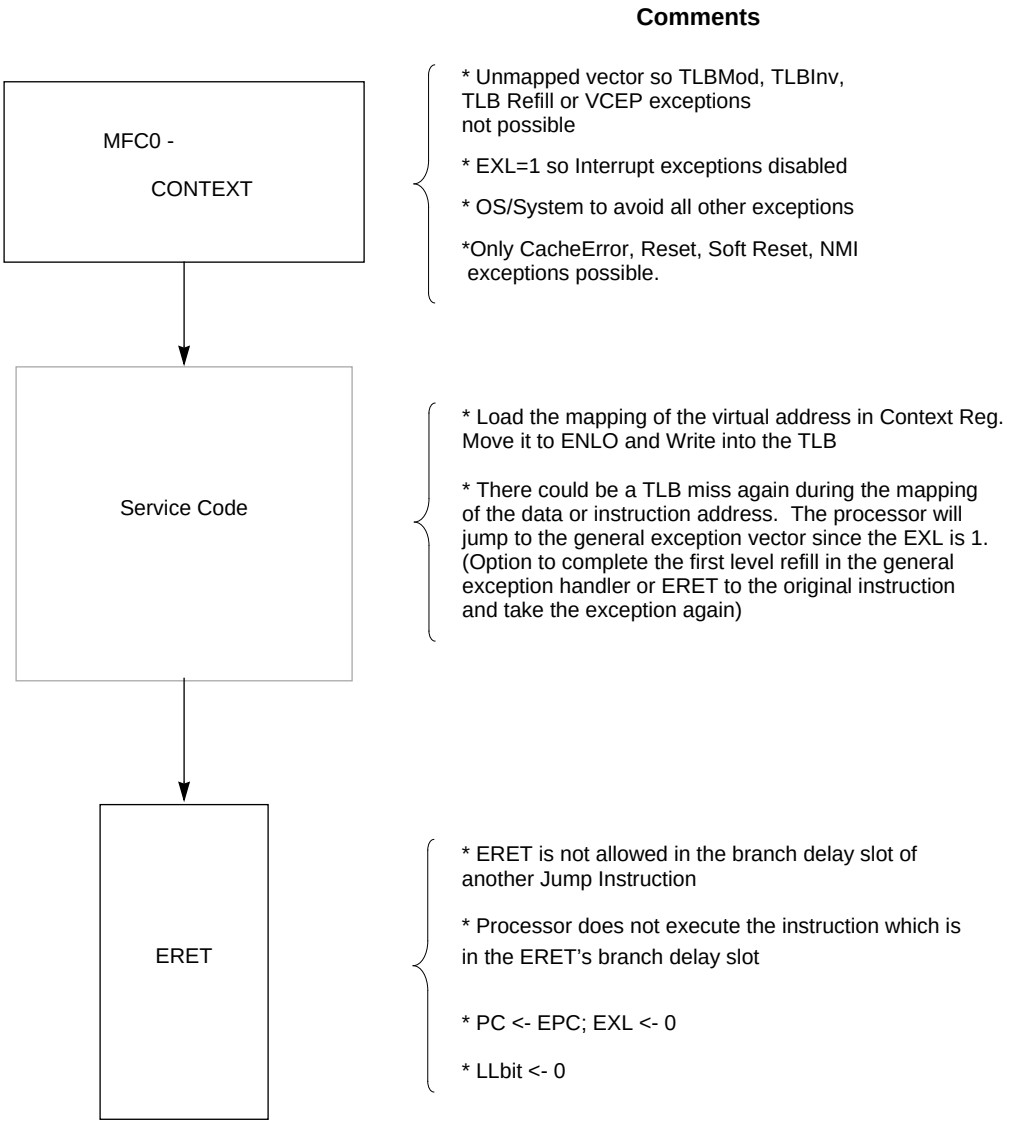


Figure 5.20 TLB/XTLB Exception Servicing Guidelines (SW)

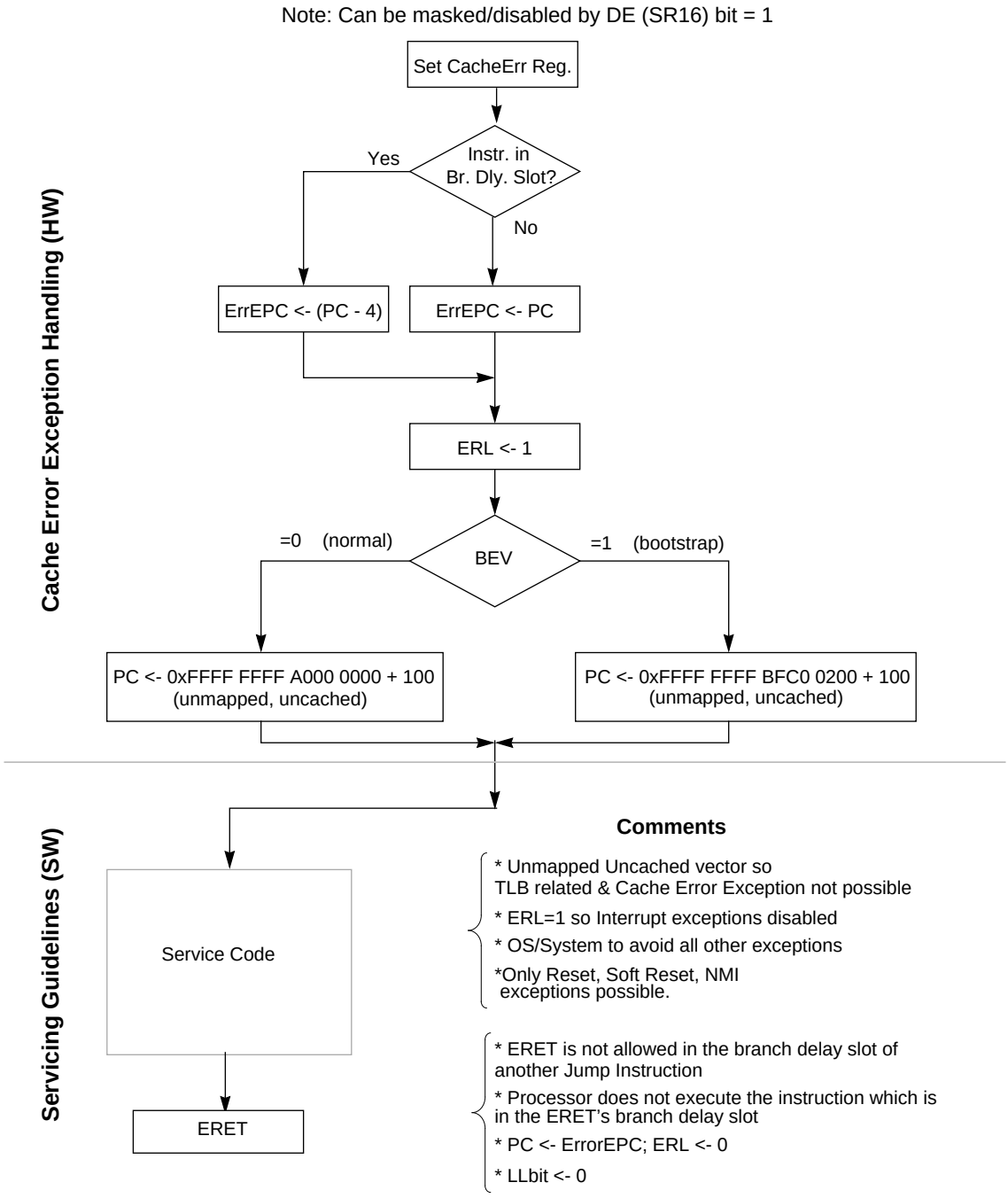


Figure 5.21 Cache Error Exception Handling (HW) and Servicing Guidelines

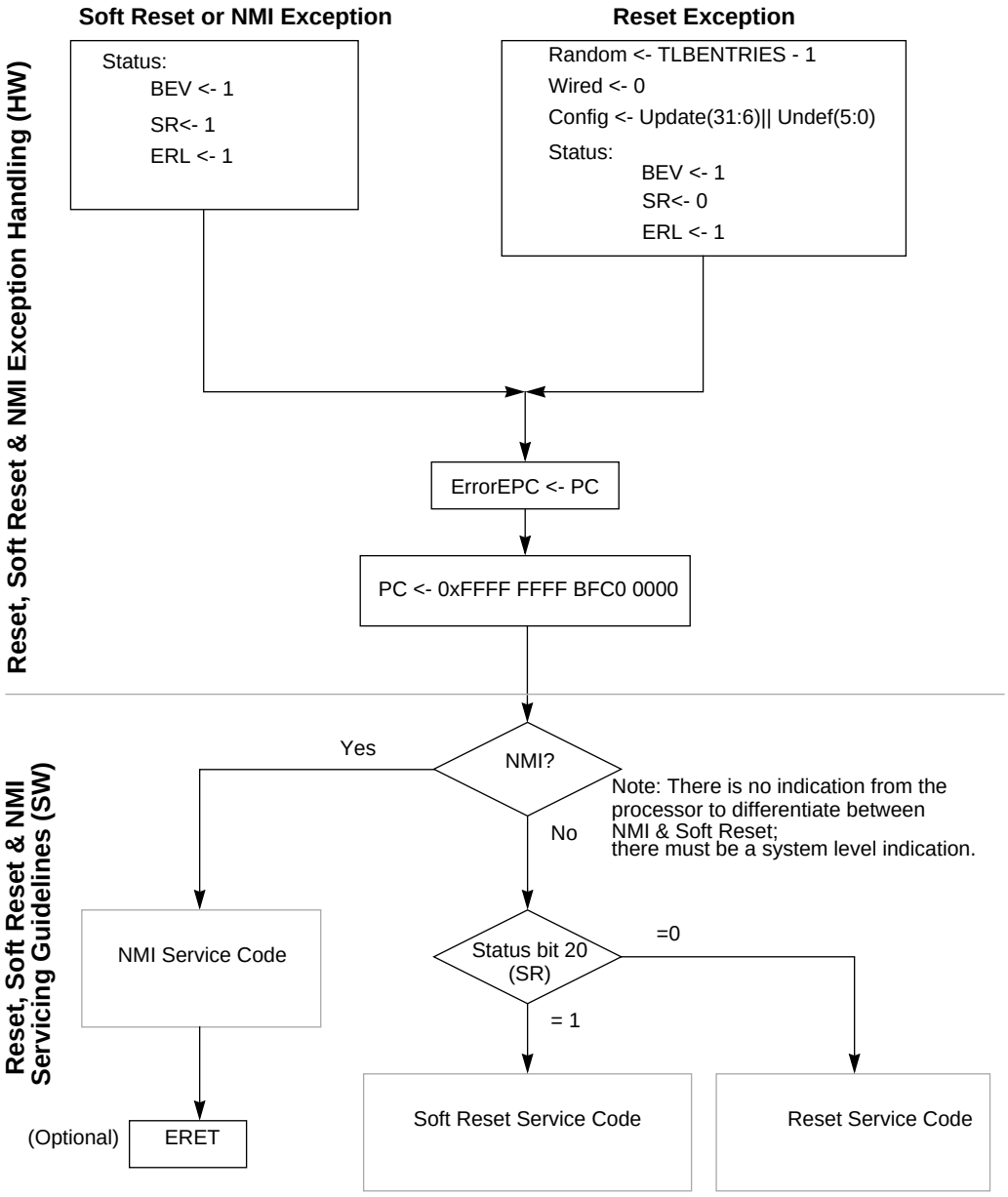


Figure 5.22 Reset, Soft Reset & NMI Exception Handling

Introduction

This section describes the floating-point unit (FPU) of the IDT R5000 processor, including the programming model, instruction set and formats, and the pipeline.

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

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 6.1 illustrates the functional organization of the FPU.

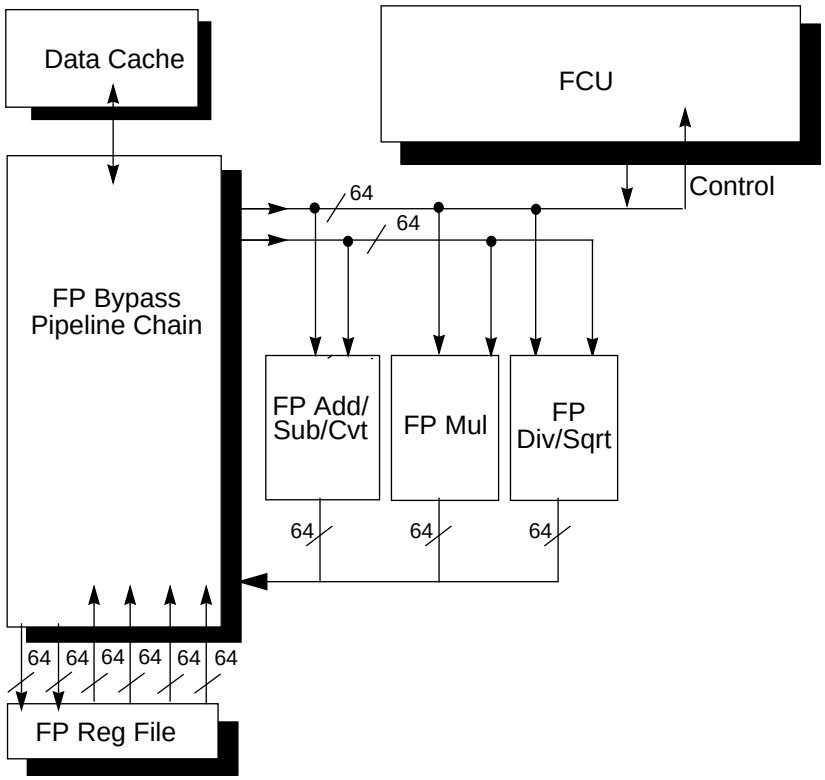


Figure 6.1 FPU Functional Block Diagram

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.

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*.

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 6.2.

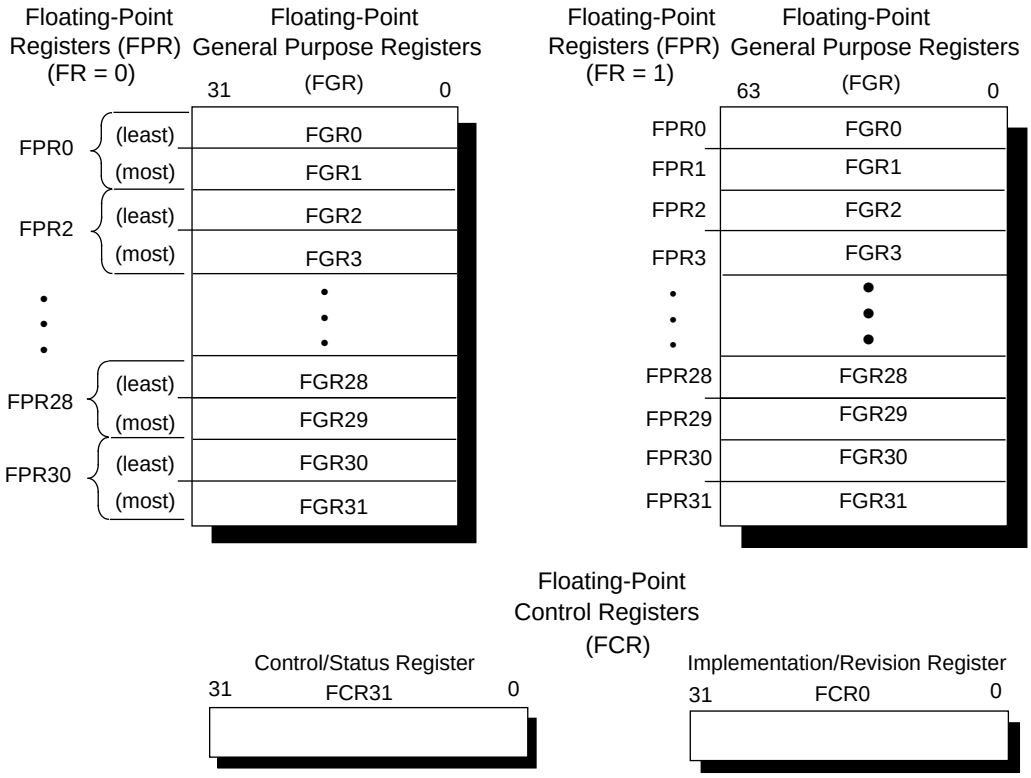


Figure 6.2 FPU Registers

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 6.2) can be used to address *FPR*s. When the *FR* bit is set to a 1, all *FPR* register numbers are valid.

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

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 6.1 lists the assignments of the *FCR*s.

Table 6.1 Floating-Point Control Register Assignments

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

Implementation and Revision Register, (FCR0)

The read-only *Implementation and Revision* register (*FCR0*) specifies the implementation and revision number of the FPU. This information can determine the coprocessor revision and performance level, and can also be used by diagnostic software.

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

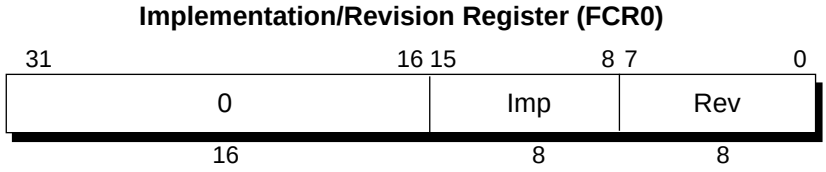


Figure 6.3 Implementation/Revision Register

Table 6.2 FCR0 Fields

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

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, MIPS does not guarantee that changes to its chips are necessarily reflected by the revision number, or that changes to the revision number necessarily reflect real chip changes. For this reason revision number values are not listed, and software should not rely on the revision number to characterize the chip.

Control/Status Register (FCR31)

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

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

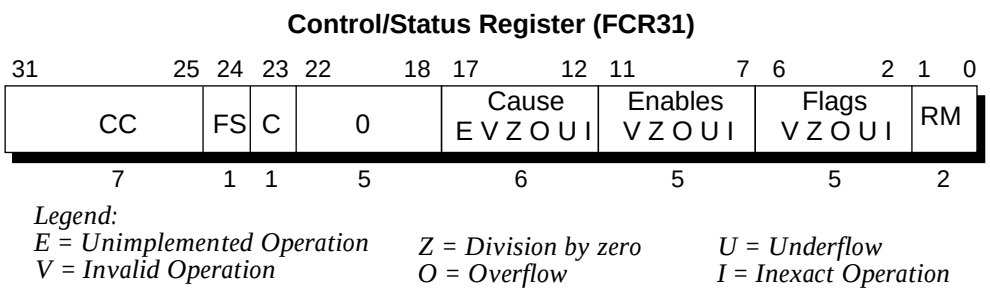


Figure 6.4 FP Control/Status Register Bit Assignments

Table 6.3 Control/Status Register Fields

Field	Description
CC	Condition code.
FS	When set, denormalized results are flushed to 0 instead of causing an unimplemented operation exception.
C	Condition bit. See description of *Control/Status* register *Condition* bit.
Cause	Cause bits. See description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
Enables	Enable bits. See description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
Flags	Flag bits. See description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
RM	Rounding mode bits. See description of *Control/Status* register *Rounding Mode Control* bits.

Accessing the Control/Status Register

When the *Control/Status* register is read by a Move Control From Coprocessor 1 (CFC1) instruction, all unfinished instructions in the pipeline are completed before the contents of the register are moved to the main processor. If a floating-point exception occurs as the pipeline empties, the FP exception is taken and the CFC1 instruction is re-executed after the exception is serviced.

The bits in the *Control/Status* register can be set or cleared by writing to the register using a Move Control To Coprocessor 1 (CTC1) instruction. *FCR31* must only be written to when the FPU is not actively executing floating-point operations; this can be ensured by reading the contents of the register to empty the pipeline.

IEEE Standard 754

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

Control/Status Register FS Bit

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

Control/Status Register Condition Bit

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

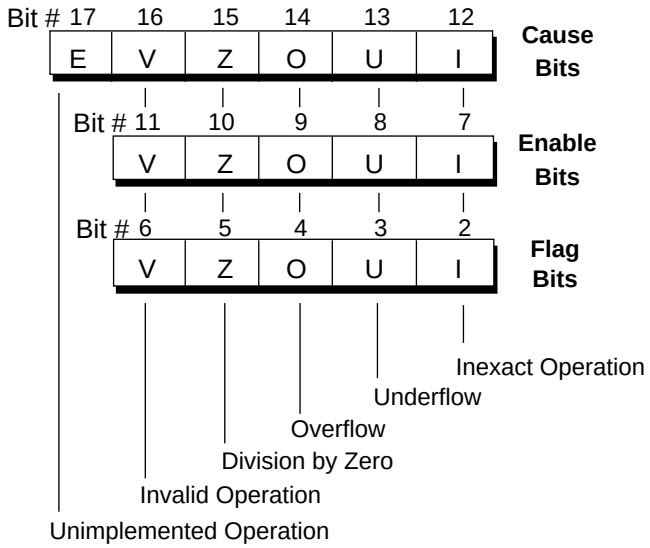


Figure 6.5 Control/Status Register Cause, Flag, and Enable Fields

Control/Status Register Cause, Flag, and Enable Fields

Figure 6.5 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 6.5, 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.

Control/Status Register Rounding Mode Control Bits

Bits 1 and 0 in the *Control/Status* register constitute the *Rounding Mode* (*RM*) field.

As shown in Table 6.4, these bits specify the rounding mode that the FPU uses for all floating-point operations.

Table 6.4 Rounding Mode Bit Decoding

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

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 6.6.

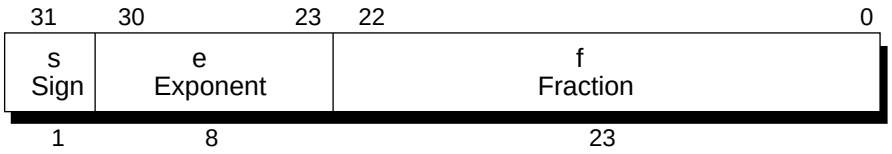


Figure 6.6 Single-Precision Floating-Point Format

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 6.7.

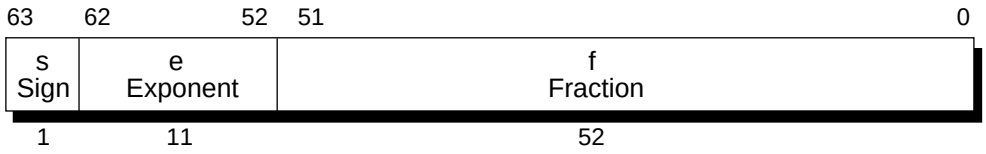


Figure 6.7 Double-Precision Floating-Point Format

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 6.5.

Table 6.5 Calculating Values in Single and Double-Precision Formats

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

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

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

Table 6.6 Floating-Point Format Parameter Values

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

Table 6.7 Minimum and Maximum Floating-Point Values

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

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

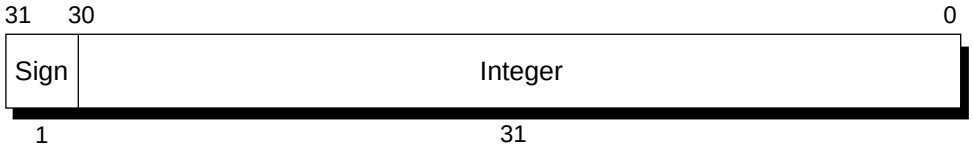


Figure 6.8 Binary Fixed-Point Format

Field assignments of the binary fixed-point format are:

Table 6.8 Binary Fixed-Point Format Fields

Field	Description
sign	sign bit
integer	integer value

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 6.9 through Table 6.12, the *fmt* appended to the instruction opcode specifies the data format: *S* specifies single-precision binary floating-point, *D* specifies double-precision binary floating-point, *W* specifies 32-bit binary fixed-point, and *L* specifies 64-bit (long) binary fixed-point.

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

OpCode	Description
LWC1	Load Word to FPU
LWXC1	Load Word Indexed to FPU
SWC1	Store Word from FPU
SWXC1	Store Word Indexed from FPU
LDC1	Load Doubleword to FPU
LDXC1	Load Doubleword Indexed to FPU
SDC1	Store Doubleword From FPU
SDXC1	Store Doubleword Indexed From FPU
MTC1	Move Word To FPU
MFC1	Move Word From FPU
CTC1	Move Control Word To FPU
CFC1	Move Control Word From FPU
DMTC1	Doubleword Move To FPU
DMFC1	Doubleword Move From FPU
PREF	Prefetch - Register + Offset
PREFX	Prefetch Indexed - Register + Register

Table 6.10 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

OpCode	Description
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 6.11 FPU Instruction Summary: Computational Instructions

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

Table 6.12 FPU Instruction Summary: Compare and Branch Instructions

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

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 44.

Transfers Between FPU and Memory

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

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

These load and store operations are unformatted; no format conversions are performed and therefore no floating-point exceptions can occur due to these operations.

Transfers Between FPU and CPU

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

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

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

Load Delay and Hardware Interlocks

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

Data Alignment

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

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

Endianness

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

Floating-Point Conversion Instructions

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

Floating-Point Computational Instructions

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

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

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

Branch on FPU Condition Instructions

The Branch on FPU (coprocessor unit 1) condition instructions that can test the result of the FPU compare (C.cond) instructions. For a detailed description of each instruction, refer to the MIPS IV instruction set manual.

Floating-Point Compare Operations

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

Table 6.13 lists the mnemonics for the compare instruction conditions.

Table 6.13 Mnemonics and Definitions of Compare Instruction Conditions

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

FPU Instruction Pipeline Overview

The FPU provides an instruction pipeline that parallels the CPU instruction pipeline. It shares the same eight-stage pipeline architecture with the CPU.

Instruction Execution

Figure 6.9 illustrates the 8-instruction overlap in the FPU pipeline.

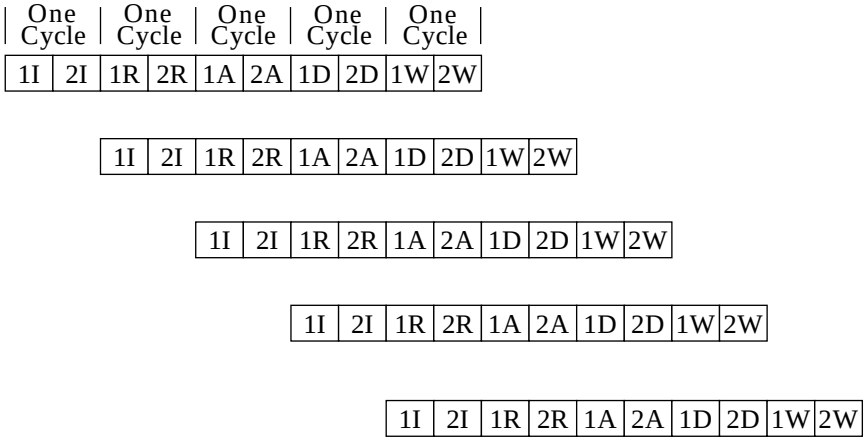


Figure 6.9 FPU Instruction Pipeline

Figure 6.9 assumes that one instruction is completed every PCycle. Most FPU instructions, however, require more than one cycle in the EX stage. This means the FPU must stall the pipeline if an instruction execution cannot proceed because of register or resource conflicts.

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.

Table 6.14 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.

Table 6.14 Floating-Point Operation Latencies

Operation	Pipeline Cycles Latency/Repeat				Operation	Pipeline Cycles Latency/Repeat	
	S	D	W	L		S	D
ADD.fmt	4/1	4/1			CVT.[W,L]	4/1	4/1
SUB.fmt	4/1	4/1			C.fmt.cond	1/1	1/1
MUL.fmt	4/1	5/2			BC1T	1	
DIV.fmt	21/19	36/34			BC1F	1	
SQRT.fmt	21/19	36/34			BC1TL	1	
RECIP	21/19	36/34			BC1FL	1	
RSQRT	38/36	68/66			LWC1	1	
ABS.fmt	1/1	1/1			SWC1, SDC1	1	
MOV.fmt	1/1	1/1			LDC1, SDC1	2	
NEG.fmt	1/1	1/1			MTC1, DMTC1	2	
ROUND.W /TRUNC.W	4/1	4/1			MFC1, DMFC1	2	
ROUND.L/ TRUNC.L	4/1**	4/1**			CTC1	3	
CEIL.W/ FLOOR.W	4/1	4/1			CFC1	2	
CEIL.L/ FLOOR.L	4/1**	4/1**			MADD	4/1	5/2
CVT.D.fmt	4/1	(a)	4/1	4/1*	MSUB	4/1	5/2
CVT.S.fmt	(a)	4/1	6/3	6/3*	NMADD	4/1	5/2
					NMSUB	4/1	5/2

(a) .....These operations are illegal.
* .....Trap on greater than 52 bits of significance.
** .....Trap on greater than 53 bits of significance.

Instruction Scheduling Constraints

The FPU resource scheduler is kept from issuing instructions to the FPU op units (adder, multiplier, and divider) by the limitations in their micro-architectures. 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.

Introduction

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.

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 (if the FPU interrupt input to the CPU is enabled).

Figure 7.1 illustrates the *Control/Status* register bits that support exceptions.

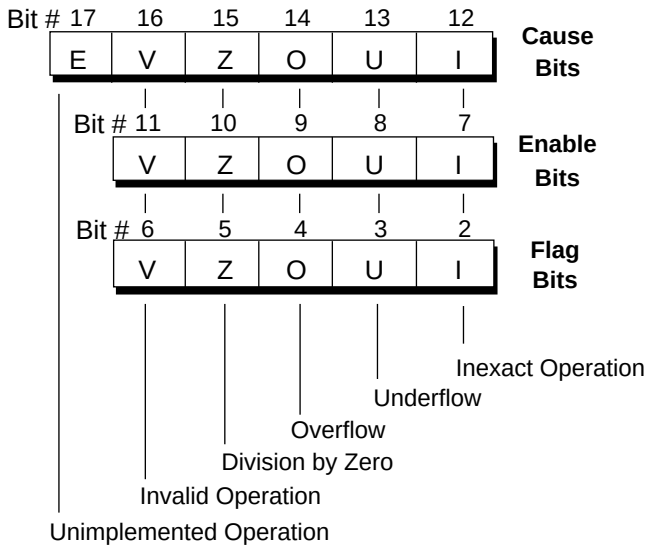


Figure 7.1 Control/Status Register Exception/Flag/Trap/Enable Bits

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.

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.

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 7.1 lists the default action taken by the FPU for each of the IEEE exceptions.

Table 7.1 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 7.2 lists the exception-causing situations and contrasts the behavior of the FPU with the requirements of the IEEE Standard 754.

Table 7.2 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[a]	O,I	O,I	Normalized exponent > E_{max}
Division by zero	Z	Z	Z	Zero is (exponent = E_{min}-1, mantissa = 0)
Overflow on convert	V	E	E	Source out of integer range
Signaling NaN source	V	V	V	
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)

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

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.

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.

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.

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($\pi/2$), csc(0), or $0^{-1.}$

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: The result, when no trap occurs, is a correctly signed infinity.

Overflow Exception (O)

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

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 50).

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 7.1).

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.

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.

Trap Enabled Results: The original operand values are undisturbed.

Trap Disabled Results: This trap cannot be disabled.

Saving and Restoring State

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

When the coprocessor *Control/Status* register (*FCR31*) is read, and the coprocessor is executing one or more floating-point instructions, the instruction(s) in progress are either completed or reported as exceptions. The architecture requires that no more than one of these pending instructions can cause an

exception. 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.

Trap Handlers for IEEE Standard 754 Exceptions

The IEEE Standard 754 strongly recommends that users be allowed to specify a trap handler for any of the five standard exceptions that can compute; the trap handler can either compute or specify a substitute result to be placed in the destination register of the operation.

By retrieving an instruction using the processor *Exception Program Counter (EPC*) register, the trap handler determines:

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

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

On Overflow or Underflow exceptions encountered on floating-point conversions, and on Invalid Operation and Divide-by-Zero exceptions, the trap handler gains access to the operand values by examining the source registers of the instruction.

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