MIPS64™ Specification

Revision 1.0
November 15, 1999

MIPS Technologies, Inc.
1225 Charleston Road
Mountain View, CA 94043-1353

RESTRICTED DOCUMENT UNDER DOCUMENT CONTROL

Copy Number: *SiByte Controlled Copy* [redacted]

Registered to: *SiByte*

[redacted]

Revision History

Revision	Date	Who	Description
0.1	January 8, 1999	GMU	Release for first internal review
0.2	January 20, 1999	GMU	Update based on internal review in preparation for first external release. Changes in this revision: Update based on final internal review and software ISV feedback. Changes in this revision: • To reduce the number of software options, allow no FPU, one with S+D+W+L and one with S+D+PS+W+L. • Explain the reason for decommitting support for the Branch Likely instructions. • Explain why the SPEC2 variants of instruction multiply do not allow multiple destination registers. • Add L2 cache encodings to the Cache instruction. • To provide a stable software environment, upgrade the compliance level for PageMask, Count, Compare, Config1, PerfCnt, TagLo, and TagHi in situations as described in the PRA chapter. • Add the WR bit to Config1. • Reserve CP0 register 22 for implementations. • Require implementation of at least encodings 2 and 3 in the cache coherency attributes. • Add the first pass description of CP0 hazards and the ssnop instruction to support them. • Add a description of the PREF and PREFX instructions. • Virtual address mapping description recast in terms of implementation parameters for numbers of virtual and physical address bits, and generally expanded.
0.21	July 1, 1999	GMU	Modify clo and clz definitions to use rd as the target register instead of rt. Require that software duplicate the target register in both register fields to ensure compatibility.

Revision	Date	Who	Description
0.9	October 20, 1999	GMU	Update with all feedback since last major release. Changes in this version: • Clarify the difference between the use of Coprocessor Unusable Exceptions and Reserved Instruction Exceptions. The cases are now enumerated for each exception. • Add the COP2 interface instructions that allow the processor to communicate with a generic coprocessor. • Clarify the intent behind the pref instruction hints and allow implementation dependent hints. • Clarify the required Cache instruction encodings and the boundary condition between a locked cache line and an intervention that hits on the line. • Note that the VPN/PFN bits corresponding to the bits set in the PageMask register may be either preserved or zeroed during a TLB write, or a subsequent read. • Add descriptions of all of the floating point control registers. • Add the floating point control register descriptions to capture the small changes from the MIPS RISC Architecture documents. • Clarify the exact definition of all operating modes, including Debug Mode, and note that Debug Mode has full access to all Kernel Mode resources. • Move the BAT descriptions to Appendix A and add an alternative fixed mapping MMU description to the same Appendix. • Note undefined processor operation for illegal values of PageMask. • Clarify the order of exception priority. • Note that Soft Reset, NMI, and Machine Check are all optional. • Clarify the exception that occurs when a parity or ECC error is detected on the system bus. • Include a warning about potential live lock in the Random register description. • Clarify the required and the optional cache coherency attribute encodings and provide some historical perspective on use. • Note that the behavior of the Count register in low power modes is implementation dependent. • Clean up the Status register diagram and corresponding descriptions. • Allow implementation dependent exception cause encodings • Add a description of the PerfCnt register. • Update the CP0 hazards section.

Revision	Date	Who	Description
0.9, continued	October 20, 1999	GMU	Continue correction of minor errors: • Clarify delta instruction table encoding to indicate that Coprocessor Unusable Exceptions are taken on coprocessor interface instructions only if access to the coprocessor is not enabled. Otherwise, a Reserved Instruction Exception is taken on unimplemented coprocessor interface instructions. • Clarify the COPz instruction to note that a Reserved Instruction Exception is possible if access is allowed to coprocessor z and the COPz instruction is not implemented for that coprocessor. • Note that it is implementation dependent whether a watch exception occurs on a cache or prefetch instruction. The preferred implementation is not to cause a watch exception on these instructions. • Correct the reset state of the I, R, and W bits in WatchLo. Their reset state should be zero. • Update the list of initialized state described for the Reset and Soft Reset exceptions to be consistent with the values in the CP0 register descriptions. • Clean up the use of sign_extend in the pseudo-code. • Fix TLB Write Indexed that should have been Write Random in Random Register description. • Add restriction on setting Status$_{ERL}$ while executing in kuseg. • Rewrite the sections on virtual memory to correctly explain the nuances involved in implementing 64-bit addressing. • Augment the TLB-based address translation section to clarify the exact generation of the physical address from pfn and va, as a function of the page size in the matching TLB entry. • Reduce the size of the address range that transforms from user mapped to unmapped when ERL is a one. This increases implementation flexibility by allowing the logic that transforms kseg0 or kseg1 addresses to also be used in this instance. • Modify cache instruction encodings to support a unified secondary and a tertiary cache. • Add the PC bit in Config1 and the M bit in the Performance Counter Control Register to allow software to determine how many performance counters are implemented. • Generalize the description of the XContext register to use the new *SEGBITS* parameter. • Reserve Cause code value 18 for precise Coprocessor 2 exceptions. • Update the CacheErr register description to more accurately reflect previous MIPS implementations. • Make it clear that access to all floating point instructions is controlled by CU1, not CU3.

Revision	Date	Who	Description
1.0	November 15, 1999	GMU	Do final edits for Revision 1.0 release. Changes in this version: • Add assembler format for optional third (sel) operand of [D]MFCz and [D]MTCz instructions. • Correct the description of the function of the FR bit in the Status register. • Correct the inconsistencies describing the enabling of 64-bit operations in Supervisor Mode. When the processor is running in Supervisor Mode, 64-bit operations are always enabled. The PX and SX bits in the Status register do not affect 64-bit operations. • Add implementation and programming notes to the multiply-related instructions indicating that software should place short operands in GPR rt and hardware should check that register for data-dependent latency. • Add a section describing changes to the MIPS RISC Architecture specification that are a required part of MIPS64. • If a deferred watch exception occurs along with another exception on the same instruction, make it implementation dependent whether the WP bit is set. • Add optional cache instruction encodings for Secondary and Tertiary caches Also note that cache error exceptions can occur on some cache instruction operations. • Clean up the terminology around reset: The Cold Reset signal causes a Reset Exception and the Reset signal causes the Soft Reset Exception. • Add descriptions for the movn.ps and movz.ps instructions, which were added to MIPS64 in the previous release of this document, but for which there was no instruction description. • Clean up the redundant wording in the pref instruction description. • Note that any TLB instruction may generate a Machine Check. • Add missing set of ERL on a cache error exception. • Note that cache and bus errors may be imprecise in some cases. • Note that the preferred rate at which to increment the Count register is once per processor cycle. • Modify the definition of the xuseg/xsuseg/xkuseg Segments to refer to the area above useg/suseg/kuseg. This is simply a definition (not a functional) change.

Contents

List of Figures

List of Tables

1. The MIPS64™ Architecture

1.1 Architecture and Document Feedback

Comments or questions on the MIPS64™ Architecture or this document should be directed to

Director of MIPS Architecture
MIPS Technologies, Inc.
1225 Charleston Road
Mountain View, CA 94043

or via E-mail to architecture@mips.com.

1.2 MIPS64 Overview

1.2.1 Historical Perspective

The MIPS® Instruction Set Architecture (ISA) has evolved over time from the original MIPS I™ ISA to the most recent MIPS V™ ISA. As the ISA has evolved, all extensions have been backward compatible with previous versions of the ISA. With the MIPS III™ level of the ISA, 64-bit integers and addresses were added to the instruction set. The MIPS IV™ and MIPS V™ levels of the ISA added improved floating point operations, as well as a set of instructions intended to improve the efficiency of generated code and of data movement. Because of the strict backward-compatible requirement of the ISA, such changes were unavailable to 32-bit implementations of the ISA which were, by definition, MIPS I™ or MIPS II™ implementations.

While the user-mode ISA was always backward compatible, the privileged environment was allowed to change on a per-implementation basis. As a result, the R3000® privileged environment was different than the R4000® privileged environment, and subsequent implementations, while similar to the R4000 privileged environment, included subtle differences. Because the privileged environment was never part of the MIPS ISA, an implementation had the flexibility to make changes to suit that particular implementation. Unfortunately, this required kernel software changes to every operating system or kernel environment on which that implementation was intended to run.

Many of the original MIPS implementations were targeted at computer-like applications such as workstations and servers. In recent years MIPS implementations have had significant success in embedded applications to the extent that most of the MIPS parts that are shipped go into some sort of embedded application. Such applications tend to have different trade-offs than computer-like applications including a focus on cost of implementation, and performance as a function of cost and power.

1.2.2 The MIPS64 Architecture

The MIPS64 Architecture is intended to address the need for a high-performance but cost-sensitive 64-bit MIPS instruction set. It is based on the MIPS V ISA and is backward compatible with the 32-bit MIPS32 Architecture. It also brings the privileged environment into the Architecture definition to address the needs of operating systems and other kernel software. The MIPS64 Architecture therefore consists of the following components:

- The Instruction Set Architecture based on the MIPS V ISA, with backward compatibility to the MIPS32 Architecture.
- The 64-bit MIPS Privileged Resource Architecture which defines the requirements for the privileged environment.
- The provision for including MIPS Application Specific Extensions (ASEs) to address the specific needs of particular markets.

2. The MIPS64 ISA

The MIPS64 instruction set includes the following instructions:

- The MIPS V CPU instructions
- The MIPS V FPU instructions
- A set of new instructions targeted at embedded applications.
- The instructions which act as the ISA interface to the MIPS Privileged Resource Architecture

This specification does not describe many of these instructions in any detail because it is assumed that the reader also has access to the most recent copy of the two-volume set entitled MIPS RISC Architecture. Only differences and additions are described in this document. Upon completion of the review process, this document will be incorporated into the MIPS RISC Architecture document.

2.1 Compliance and Subsetting

To be compliant with the MIPS64 Architecture, designs must implement a set of required features, as described in this document. To allow flexibility in implementations, the MIPS64 Architecture does provide subsetting rules. An implementation that follows these rules is compliant with the MIPS64 Architecture as long as it adheres strictly to the rules, and fully implements the remaining instructions. Supersetting of the MIPS64 Architecture is only allowed by adding partner-specific functions to the *SPECIAL2* major opcode, by adding control for co-processors via the *COP2, LWC2, SWC2, LDC2*, and/or *SDC2* opcodes, or via the addition of approved Application Specific Extensions. The subsetting rules are:

- All CPU instructions must be implemented - no subsetting is allowed.
- The FPU and related support instructions, including the MOVF and MOVT CPU instructions, may be omitted. If the FPU is implemented, the paired single (PS) format is optional. Therefore, the following allowable FPU subsets are compatible with the MIPS64 architecture:
  - No FPU
  - FPU with S, D, W, and L formats and all supporting instructions
  - FPU with S, D, PS, W, and L formats and all supporting instructions
- Implementation of the full 64-bit address space is optional. The processor may implement 64-bit data and operations with a 32-bit only address space. In this case, the MMU acts as if 64-bit addressing is always disabled.
- Supervisor Mode is optional. If Supervisor Mode is not implemented, bit 3 of the *Status* register must be ignored on write and read as zero.
- The standard TLB-based memory management unit may be replaced with a simpler MMU (e.g., a Fixed Mapping MMU). If this is done, the rest of the interface to the Privileged Resource Architecture must be preserved. If a TLB-based memory management unit is implemented, it must be the standard TLB-based MMU as described in the Privileged Resource Architecture chapter.
- The Privileged Resource Architecture includes several implementation options and may be subsetted in accordance with those options.
- Instruction, CP0 Register, and CP1 Control Register fields that are marked "Reserved" or shown as "0" in the description of that field are reserved for future use by the architecture and are not available to implementations. Implementations may use those fields that are explicitly reserved for implementation dependent use.
- CP0 Register and CP1 Control Register encodings that are not currently used are reserved for the future use of the architecture and are not available to implementations. Implementations may use those encodings that are explicitly reserved for implementation dependent use.
- EJTAG and the ASEs are optional and may be subsetted out. If they are implemented, they must implement the entire ISA applicable to the component, or implement subsets that are approved by the EJTAG or ASE specifications.
- If any instruction is subsetted out based on the rules above, an attempt to execute that instruction must cause the appropriate exception (typically Reserved Instruction or Coprocessor Unusable).

2.2 Changes to Revision 5.1 of the MIPS RISC Architecture Specification

In addition to the MIPS64 Architecture described in this document, the following changes to Revision 5.1 of the MIPS RISC Architecture Specification are required for compliance with the MIPS64 Architecture:

- The MIPS IV ISA added a restriction to the load and store instructions which have natural alignment requirements (all but load and store byte and load and store left and right) in which the base register used by the instruction must also be naturally aligned (the restriction expressed in the MIPS RISC Architecture Specification is that the offset be aligned, but the implication is that the base register is also aligned, and this is more consistent with the indexed load/store instructions which have no offset field). The restriction that the base register be naturally-aligned is eliminated by the MIPS64 Architecture, leaving the restriction that the effective address be naturally-aligned.
- Early MIPS implementations required two instructions separating a mflo or mfhi from the next integer multiply or divide operation. This hazard was eliminated in the MIPS IV ISA, although the MIPS RISC Architecture Specification does not clearly explain this fact. The MIPS64 Architecture explicitly eliminates this hazard and requires that the hi and lo registers be fully interlocked in hardware for all integer multiply and divide instructions (including, but not limited to, the madd, maddu, msub, msubu, and mul instructions introduced in this specification).
- The Implementation and Programming Notes included in this specification for the madd, maddu, msub, msubu, and mul instructions should also be applied to all integer multiply and divide instructions in the MIPS RISC Architecture Specification.

2.3 CPU Architecture

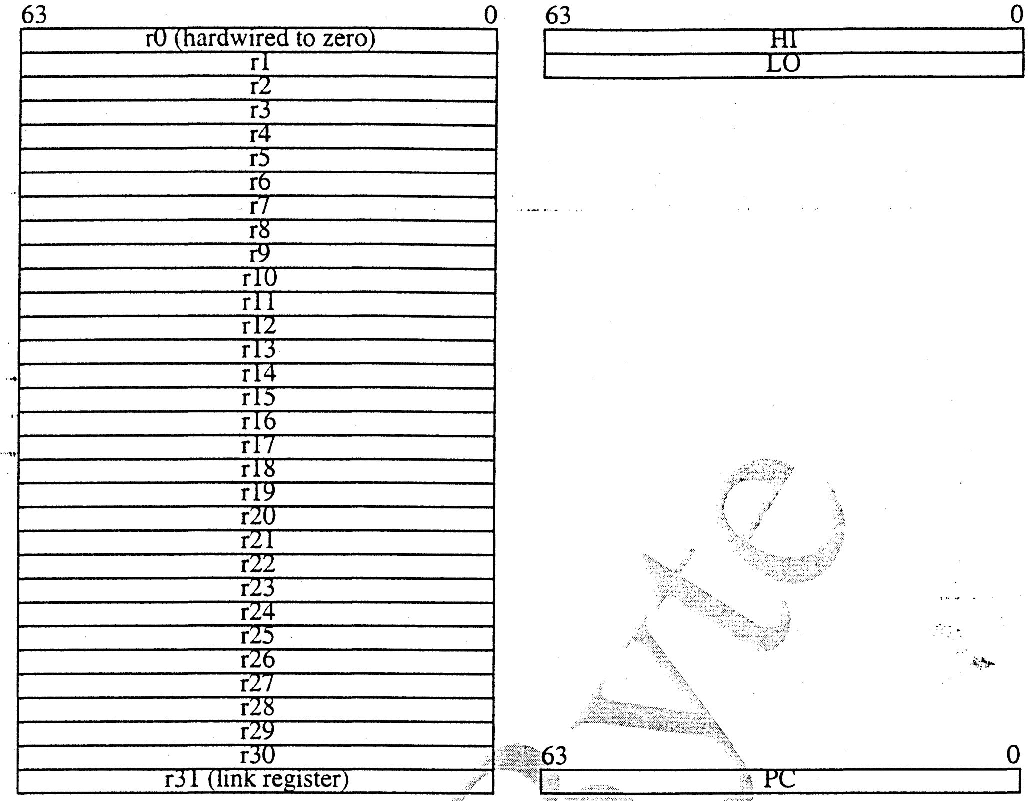
2.3.1 CPU Register Overview

The MIPS64 Architecture defines the following CPU registers:

- 32 64-bit general purpose registers (GPRs)
- a pair of special-purpose registers to hold the results of integer multiply, divide, and multiply-accumulate operations (HI and LO)
- a special-purpose program counter (PC), which is affected only indirectly by certain instructions - it is not an architecturally-visible register.

A MIPS64 processor always produces a 64-bit result, even for those instructions which are architecturally defined to operate on 32 bits. Such instructions typically sign-extend their 32-bit result into 64 bits. In so doing, 32-bit programs work as expected, even though the registers are actually 64 bits wide rather than 32.

Figure 1: CPU Registers in MIPS64 Native Mode



General Purpose Registers

Special Purpose Registers

2.3.2 Endianness

Compliant implementations of the MIPS64 Architecture must be bi-endian. That is, they must be capable of running in either a big-endian or a little-endian byte order, as selected by an implementation-specific power-up sequence. The BE bit in the *Config* register, set as part of the power-up sequence, indicates the endian mode in which the processor is running. It is implementation-dependent whether reverse-endian mode is implemented.

2.3.3 CPU Instruction Overview

Table 1 through Table 9 list the CPU instructions that are part of the MIPS64 ISA. If 64-bit operations are not enabled, certain instructions, as described in the Instruction Bit Encoding tables, are not legal and result in a Coprocessor Unusable Exception or Reserved Instruction Exception, as appropriate to the type of instruction.

Note

Although the Branch Likely instructions are included in this specification, software is strongly encouraged to avoid use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS64 architecture. The Branch Likely instructions were added to the ISA at a time when processor implementations were much simpler. Since that time, implementation of the Branch Likely instructions has been shown to be increasingly difficult and costly on processors with aggressive branch prediction. Continued use by software will result in serious performance issues as such processor designs penetrate the embedded market. The

Branch Likely instructions are listed in Table 8 and Table 15.

Table 1: CPU Load, Store, and Memory Control Instructions

Mnemonic	Instruction	Original MIPS ISA Level
LB	Load Byte	I
LBU	Load Byte Unsigned	I
LH	Load Halfword	I
LHU	Load Halfword Unsigned	I
LW	Load Word	I
LWL	Load Word Left	I
LWR	Load Word Right	I
SB	Store Byte	I
SH	Store Halfword	I
SW	Store Word	I
SWL	Store Word Left	I
SWR	Store Word Right	I
LL	Load Linked Word	II
SC	Store Conditional Word	II
SYNC	Synchronize Memory Operations	II
LD	Load Doubleword	III
LDL	Load Doubleword Left	III
LDR	Load Doubleword Right	III
LLD	Load Linked Doubleword	III
LWU	Load Word Unsigned	III
SCD	Store Conditional Doubleword	III
SD	Store Doubleword	III
SDL	Store Doubleword Left	III
SDR	Store Doubleword Right	III
PREF	Prefetch Memory Data	IV
PREFX	Prefetch Memory Data Indexed	IV

Table 2: CPU Arithmetic Instructions

Mnemonic	Instruction	Original MIPS ISA Level
ADD	Add Word	I
ADDI	Add Immediate Word	I
ADDIU	Add Immediate Unsigned Word	I
ADDU	Add Unsigned Word	I
DIV	Divide Word	I
DIVU	Divide Unsigned Word	I
MULT	Multiply Word	I
MULTU	Multiply Unsigned Word	I
SLT	Set on Let Than	I

Table 2: CPU Arithmetic Instructions

Mnemonic	Instruction	Original MIPS ISA Level
SLTI	Set on Less Than Immediate	I
SLTIU	Set on Less Than Immediate Unsigned	I
SLTU	Set on Less Than Unsigned	I
SUB	Subtract Word	I
SUBU	Subtract Unsigned Word	I
DADD	Add Doubleword	III
DADDI	Add Immediate Doubleword	III
DADDIU	Add Immediate Unsigned Doubleword	III
DADDU	Add Unsigned Doubleword	III
DDIV	Divide Doubleword	III
DDIVU	Divide Unsigned Doubleword	III
DMULT	Multiply Doubleword	III
DMULTU	Multiply Unsigned Doubleword	III
DSUB	Subtract Doubleword	III
DSUBU	Subtract Unsigned Doubleword	III

Table 3: CPU Logical Instructions

Mnemonic	Instruction	Original MIPS ISA Level
AND	Logical AND	I
ANDI	Logical AND Immediate	I
LUI	Load Upper Immediate	I
NOR	Logical NOR	I
OR	Logical OR	I
ORI	Logical OR Immediate	I
XOR	Logical XOR	I
XORI	Logical XOR Immediate	I

Table 4: CPU Move Instructions

Mnemonic	Instruction	Original MIPS ISA Level
MFHI	Move from HI	I
MFLO	Move from LO	I
MTHI	Move to HI	I
MTLO	Move to LO	I
MOVF[a]	Move Conditional on Floating Point False	IV
MOVN	Move Conditional on Not Zero	IV
MOVT[a]	Move Conditional on Floating Point True	IV
MOVZ	Move Conditional on Zero	IV

a. These instructions require a floating point unit and may be subsetted out if no floating point unit is implemented.

Table 5: CPU Shift Instructions

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

Table 6: CPU Branch and Jump Instructions

Mnemonic	Instruction	Original MIPS ISA Level
BEQ	Branch on Equal	I
BGEZ	Branch on Greater Than or Equal Zero	I
BGEZAL	Branch on Greater Than or Equal Zero and Link	I
BGTZ	Branch on Greater Than Zero	I
BLEZ	Branch on Less Than or Equal Zero	I
BLTZ	Branch on Less Than Zero	I
BLTZAL	Branch on Less Than Zero and Link	I
BNE	Branch on Not Equal	I
J	Jump	I
JAL	Jump and Link	I
JALR	Jump and Link Register	I
JR	Jump Register	I

Table 7: CPU Trap Instructions

Mnemonic	Instruction	Original MIPS ISA Level
BREAK	Breakpoint	I
SYSCALL	System Call	I

Table 7: CPU Trap Instructions

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

Table 8: Obsolete[a] Branch Instructions

Mnemonic	Instruction	Original MIPS ISA Level
BEQL	Branch on Equal Likely	II
BGEZALL	Branch on Greater Than or Equal Zero and Link Likely	II
BGEZL	Branch on Greater Than or Equal Zero Likely	II
BGTZL	Branch on Greater Than Zero Likely	II
BLEZL	Branch on Less Than or Equal Zero Likely	II
BLTZALL	Branch on Less Than Zero and Link Likely	II
BLTZL	Branch on Less Than Zero Likely	II
BNEL	Branch on Not Equal Likely	II

[a] Software is strongly encouraged to avoid use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS64 architecture.

Table 9: Embedded Application Instructions

Mnemonic	Instruction	Original MIPS ISA Level
CLO	Count Leading Ones in Word	MIPS32
CLZ	Count Leading Zeros in Word	MIPS32
DCLO	Count Leading Ones in Doubleword	MIPS64
DCLZ	Count Leading Zeros in Doubleword	MIPS64
MADD	Multiply and Add Word	MIPS32
MADDU	Multiply and Add Unsigned Word	MIPS32
MSUB	Multiply and Subtract Word	MIPS32
MSUBU	Multiply and Subtract Unsigned Word	MIPS32

Table 9: Embedded Application Instructions

Mnemonic	Instruction	Original MIPS ISA Level
MUL	Multiply Word to Register	MIPS32
SSNOP	Superscalar Inhibit NOP	MIPS32

2.4 FPU Architecture

2.4.1 FPU Register Overview

The MIPS64 Architecture defines the following FPU registers:

- 32 64-bit floating point registers (FPRs).
- Five FPU control registers

For compatibility with MIPS32 processors, a MIPS64 processor can be configured to run in a mode in which the FPRs are treated as 32 32-bit registers, each of which is capable of storing only 32-bit data types. In this mode, the double-precision floating point (type D) data type is stored in even-odd pairs of FPRs, and the long-integer (type L) and paired single (type PS) data types are not supported.

Figure 2: FPU Registers if Status$_{FR}$ is 1

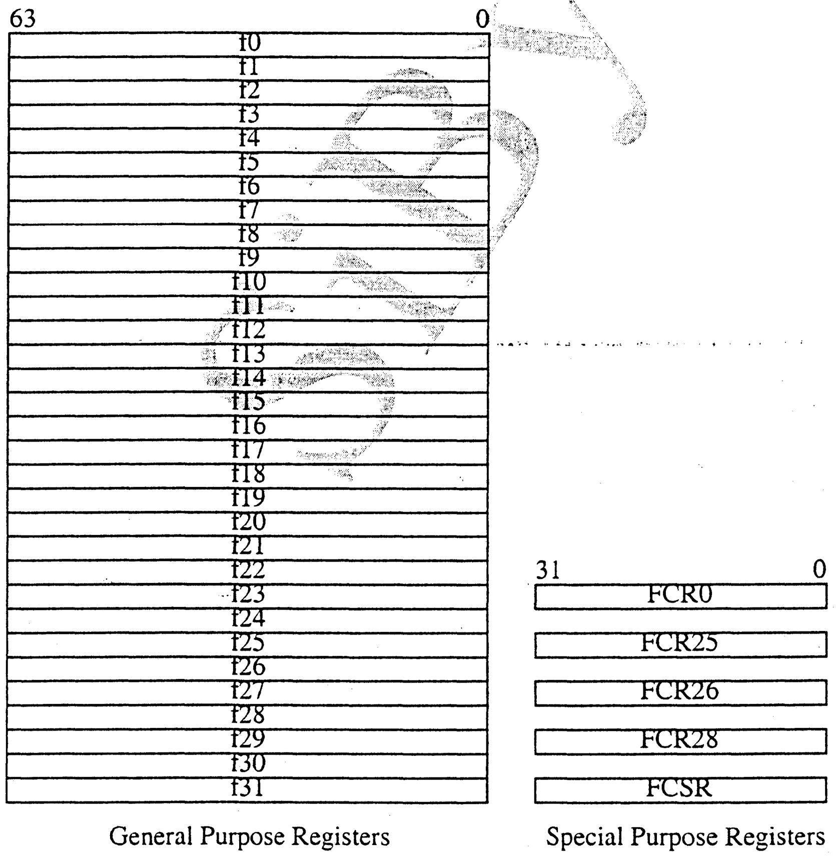
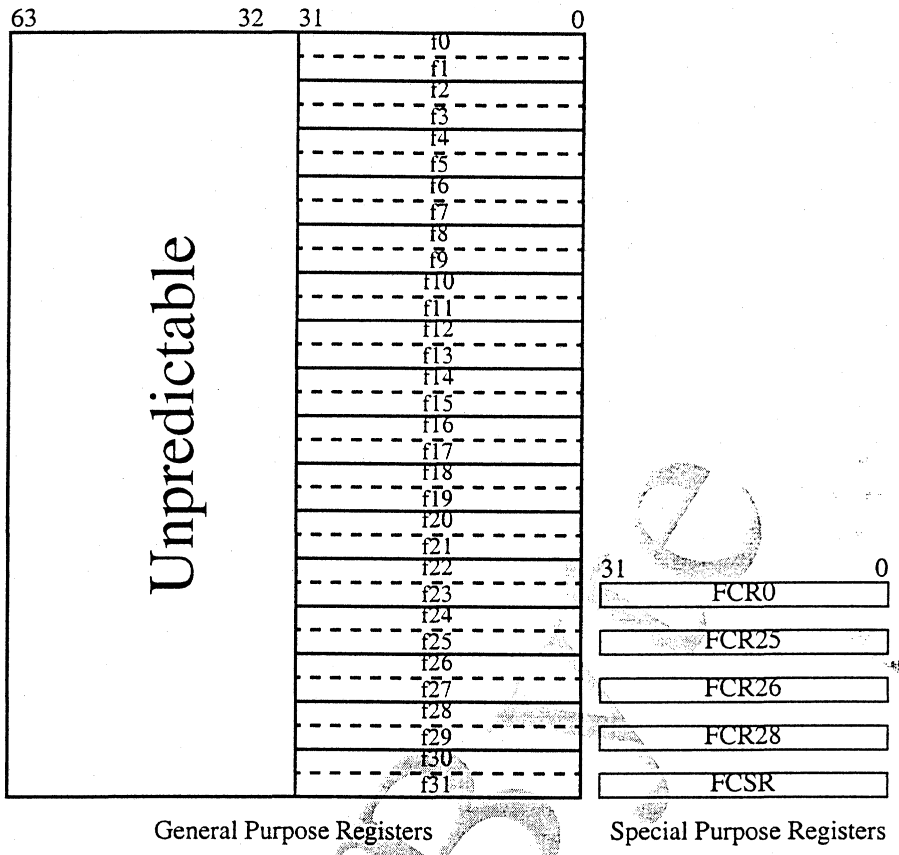


Figure 3: FPU Registers if Status$_{FR}$ is 0



2.4.2 FPU Instruction Overview

Table 10 through Table 15 list the FPU instructions that are part of the MIPS64 ISA. If the processor is configured to run in the mode providing backward compatibility to MIPS32 processors, certain instructions, as described in the Instruction Bit Encoding tables, are not legal and result in a Reserved Instruction exception. This includes those instructions which operate on paired single floating point (type PS) and 64-bit fixed point (type L) data types.

Table 10: FPU Load and Store Instructions

Mnemonic	Instruction	Original MIPS ISA Level
LWC1	Load Word to Floating Point	I
SWC1	Store Word to Floating Point	I
LDC1	Load Doubleword to Floating Point	II
SDC1	Store Doubleword to Floating Point	II
LDXC1	Load Doubleword Indexed to Floating Point	IV
LWXC1	Load Word Indexed to Floating Point	IV
SDXC1	Store Doubleword Indexed to Floating Point	IV

Table 10: FPU Load and Store Instructions

Mnemonic	Instruction	Original MIPS ISA Level
SWXC1	Store Word Indexed to Floating Point	IV
LUXC1	Load Doubleword Indexed Unaligned to Floating Point	V
SUXC1	Store Doubleword Indexed Unaligned to Floating Point	V

Table 11: FPU Arithmetic Instructions

Mnemonic	Instruction	Original MIPS ISA Level
ABS.fmt	Floating Point Absolute Value	I, V
ADD.fmt	Floating Point Add	I, V
C.cond.fmt	Floating Point Compare	I, V
DIV.fmt	Floating Point Divide	I
MUL.fmt	Floating Point Multiply	I, V
NEG.fmt	Floating Point Negate	I, V
SUB.fmt	Floating Point Subtract	I, V
SQRT.fmt	Floating Point Square Root	II
MADD.fmt	Floating Point Multiply Add	IV, V
MSUB.fmt	Floating Point Multiply Subtract	IV, V
NMADD.fmt	Floating Point Negative Multiply Add	IV, V
NMSUB.fmt	Floating Point Negative Multiply Subtract	IV, V
RECIP.fmt	Floating Point Reciprocal Approximation	IV
RSQRT.fmt	Floating Point Reciprocal Square Root Approximation	IV

Table 12: FPU Move Instructions

Mnemonic	Instruction	Original MIPS ISA Level
CFC1	Copy Word from Floating Point Control Register	I
CTC1	Copy Word to Floating Point Control Register	I
MFC1	Move Word from FPR	I
MOV.fmt	Floating Point Move	I
MTC1	Move Word to FPR	I
DMFC1	Move Doubleword from FPR	III
DMTC1	Move Doubleword to FPR	III
MOVF.fmt	Floating Point Conditional Move on FP False	IV, V
MOVN.fmt	Floating Point Conditional Move on Non-Zero	IV, V, MIPS64
MOVT.fmt	Floating Point Conditional Move on FP True	IV, V
MOVZ.fmt	Floating Point Conditional Move on Zero	IV, V, MIPS64

Table 13: FPU Convert Instructions

Mnemonic	Instruction	Original MIPS ISA Level
CVT.W.fmt	Floating Point Convert to Word Fixed Point	I
CVT.D.fmt	Floating Point Convert to Double Floating Point	I, III
CVT.S.fmt	Floating Point Convert to Single Floating Point	I,III,V
CEIL.W.fmt	Floating Point Ceiling to Word Fixed Point	II
FLOOR.W.fmt	Floating Point Floor to Word Fixed Point	II
ROUND.W.fmt	Floating Point Round to Word Fixed Point	II
TRUNC.W.fmt	Floating Point Truncate to Word Fixed Point	II
CEIL.L.fmt	Floating Point Ceiling to Long Fixed Point	III
CVT.L.fmt	Floating Point Convert to Long Fixed Point	III
FLOOR.L.fmt	Floating Point Floor to Long Fixed Point	III
ROUND.L.fmt	Floating Point Round to Long Fixed Point	III
TRUNC.L.fmt	Floating Point Truncate to Long Fixed Point	III
ALNV.PS	Floating Point Align Variable	V
CVT.PS.S	Floating Point Convert Pair to Pair Single	V
CVT.S.PL	Floating Point Convert Pair Lower to Single	V
CVT.S.PU	Floating Point Convert Pair Upper to Single	V
PLL.PS	Floating Point Pair Lower Lower	V
PLU.PS	Floating Point Pair Lower Upper	V
PUL.PS	Floating Point Pair Upper Lower	V
PUU.PS	Floating Point Pair Upper Upper	V

Table 14: FPU Branch Instructions

Mnemonic	Instruction	Original MIPS ISA Level
BC1F	Branch on Floating Point False	I, IV
BC1T	Branch on Floating Point True	I, IV

Table 15: Obsolete[a] FPU Branch Instructions

Mnemonic	Instruction	Original MIPS ISA Level
BC1FL	Branch on Floating Point False Likely	II, IV
BC1TL	Branch on Floating Point True Likely	II, IV

[a] Software is strongly encouraged to avoid use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS64 architecture.

2.5 Coprocessor Architecture

The MIPS64 Architecture supports the use of an additional coprocessor to perform application-specific tasks. Support for this coprocessor is provided by instructions, described below, that act as the control and data movement interface between the CPU and the coprocessor. The flexibility of this interface places almost no restrictions on the coprocessor, other than the fact that the coprocessor load and store instructions can only address 32 coprocessor-specific registers. It is implementation dependent whether all twelve coprocessor interface instructions are implemented - a subset of the instructions may be implemented as dictated by the requirements of the specific coprocessor. If the coprocessor is implemented and usage is enabled, an attempt to execute an unimplemented coprocessor interface instruction results in a Reserved Instruction Exception.

2.5.1 Coprocessor Instruction Overview

Table 16 lists the interface instructions supported by the MIPS64 Architecture.

Table 16: Coprocessor Interface Instructions

Mnemonic	Instruction	Original MIPS ISA Level
BC2	Branch on Coprocessor 2 Condition	MIPS I
CFC2	Move Control from Coprocessor 2	MIPS I
COP2	Perform Coprocessor 2 Operation	MIPS I
CTC2	Move Control to Coprocessor 2	MIPS I
DMFC2	Move Doubleword from Coprocessor 2	MIPS III
DMTC2	Move Doubleword to Coprocessor 2	MIPS III
LDC2	Load Doubleword to Coprocessor 2	MIPS II
LWC2	Load Word to Coprocessor 2	MIPS I
MFC2	Move from Coprocessor 2	MIPS I
MTC2	Move to Coprocessor 2	MIPS I
SDC2	Store Doubleword to Coprocessor 2	MIPS II
SWC2	Store Word to Coprocessor 2	MIPS I

2.6 Privileged Instruction Set Architecture

2.6.1 Privileged Register Overview

The MIPS64 Architecture defines a set of privileged registers as described in Chapter 3.

2.6.2 Privileged Instruction Overview

Table 17 lists the privileged instructions which act as the ISA interface to the MIPS Privileged Resource Architecture.

Table 17: Privileged Instructions

Mnemonic	Instruction
CACHE	Perform Cache Operation
DMFC0	Move Doubleword From Coprocessor Zero
DMTC0	Move Doubleword To Coprocessor Zero
ERET	Exception Return
MFC0	Move Word From Coprocessor Zero
MTC0	Move Word To Coprocessor Zero
TLBP	Translation Look Aside Buffer Probe
TLBR	Translation Look Aside Buffer Read

Table 17: Privileged Instructions

Mnemonic	Instruction
TLBWI	Translation Look Aside Buffer Write Indexed
TLBWR	Translation Look Aside Buffer Write Random
WAIT	Enter Standby Mode

2.7 EJTAG Support Instructions

Table 18 lists the EJTAG support instructions that are supported by MIPS64. Refer to the EJTAG specification for more information about these instructions.

Table 18: EJTAG Support Instructions

Mnemonic	Instruction
DERET	Debug Exception Return
SDBBP	Software Debug Breakpoint

2.8 Instruction Bit Encoding

Table 20 through Table 34 describe the encoding used for the MIPS64 ISA. Table 19 describes the meaning of the symbols used in the tables.

Table 19: Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
*	Operation or field codes marked with this symbol are reserved for future use. Executing such an instruction must cause a Reserved Instruction Exception.
δ	(Also *italic* field name.) Operation or field codes marked with this symbol denotes a field class. The instruction word must be further decoded by examining additional tables that show values for another instruction field.
β	Operation or field codes marked with this symbol represent a valid encoding for a higher-order MIPS ISA level. Executing such an instruction must cause a Reserved Instruction Exception.
⊥	Operation or field codes marked with this symbol represent instructions which are not legal if the processor is configured to be backward compatible with MIPS32 processors. If the processor is executing in Kernel Mode, Debug Mode, or 64-bit instructions are enabled, execution proceeds normally. In other cases, executing such an instruction must cause a Reserved Instruction Exception (non-coprocessor encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).
θ	Operation or field codes marked with this symbol are available to licensed MIPS partners. To avoid multiple conflicting instruction definitions, the partner must notify MIPS Technologies, Inc. when one of these encodings is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction Exception (*SPECIAL2* encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).

Table 19: Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
σ	Field codes marked with this symbol represent an EJTAG support instruction and implementation of this encoding is optional for each implementation. If the encoding is not implemented, executing such an instruction must cause a Reserved Instruction Exception. If the encoding is implemented, it must match the instruction encoding as shown in the table.
ε	Operation or field codes marked with this symbol are reserved for MIPS Application Specific Extensions. If the ASE is not implemented, executing such an instruction must cause a Reserved Instruction Exception.
ϕ	Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS64 ISA. Software should avoid using these operation or field codes.

Table 20: MIPS64 Encoding of the Opcode Field

opcode		*bits 28..26*							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	*SPECIAL* δ	*REGIMM* δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	*COP0* δ	*COP1* δ	*COP2* θδ	*COP1X* δ⊥	BEQL ϕ	BNEL ϕ	BLEZL ϕ	BGTZL ϕ
3	011	DADDI ⊥	DADDIU ⊥	LDL ⊥	LDR ⊥	*SPECIAL2* δ	JALX ε	*MDMX* εδ	*
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	LWU ⊥
5	101	SB	SH	SWL	SW	SDL ⊥	SDR ⊥	SWR	CACHE
6	110	LL	LWC1	LWC2 θ	PREF	LLD ⊥	LDC1	LDC2 θ	LD ⊥
7	111	SC	SWC1	SWC2 θ	*	SCD ⊥	SDC1	SDC2 θ	SD ⊥

Table 21: MIPS64 *SPECIAL* Opcode Encoding of Function Field

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	SLL	*MOVCI* δ	SRL	SRA	SLLV	*	SRLV	SRAV
1	001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	*	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	DSLLV ⊥	*	DSRLV ⊥	DSRAV ⊥
3	011	MULT	MULTU	DIV	DIVU	DMULT ⊥	DMULTU ⊥	DDIV ⊥	DDIVU ⊥
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	*	*	SLT	SLTU	DADD ⊥	DADDU ⊥	DSUB ⊥	DSUBU ⊥
6	110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	111	DSLL ⊥	*	DSRL ⊥	DSRA ⊥	DSLL32 ⊥	*	DSRL32 ⊥	DSRA32 ⊥

Table 22: MIPS64 *REGIMM* Encoding of rt Field

rt		*bits 18..16*							
		0	1	2	3	4	5	6	7
bits 20..19		000	001	010	011	100	101	110	111
0	00	BLTZ	BGEZ	BLTZL ϕ	BGEZL ϕ	*	*	*	*
1	01	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	10	BLTZAL	BGEZAL	BLTZALL ϕ	BGEZALL ϕ	*	*	*	*
3	11	*	*	*	*	*	*	*	*

Table 23: MIPS64 *SPECIAL2* Encoding of Function Field

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	MADD	MADDU	MUL	θ	MSUB	MSUBU	θ	θ
1	001	θ	θ	θ	θ	θ	θ	θ	θ
2	010	θ	θ	θ	θ	θ	θ	θ	θ
3	011	θ	θ	θ	θ	θ	θ	θ	θ
4	100	CLZ	CLO	θ	θ	DCLZ ⊥	DCLO ⊥	θ	θ
5	101	θ	θ	θ	θ	θ	θ	θ	θ
6	110	θ	θ	θ	θ	θ	θ	θ	θ
7	111	θ	θ	θ	θ	θ	θ	θ	SDBBP σ

Table 24: MIPS64 *MOVCI* Encoding of tf Bit

tf	*bit 16*	
	0	1
	MOVF	MOVT

Table 25: MIPS64 *COPz* Encoding of rs Field

rs		*bits 23..21*							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFCz	DMFCz ⊥	CFCz	*	MTCz	DMTCz ⊥	CTCz	*
1	01	*BCz* δ	*	*	*	*	*	*	*
2	10	*CO* δ							
3	11								

Table 26: MIPS64 COPz Encoding of rt Field When rs=*BCz*

rt	*bits 16*	
bit 17	0	1
0	BCzF	BCzT
1	BCzFL ϕ	BCzTL ϕ

Table 27: MIPS64 *COP0* Encoding of rs Field

rs		*bits 23..21*							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC0	DMFC0 ⊥	*	*	MTC0	DMTC0 ⊥	*	*
1	01	*	*	*	*	*	*	*	*
2	10	*CO* δ							
3	11								

Table 28: MIPS64 *COP0* Encoding of Function Field When rs=*CO*

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	*	TLBR	TLBWI	*	*	*	TLBWR	*
1	001	TLBP	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	ERET	*	*	*	*	*	*	DERET σ
4	100	WAIT	*	*	*	*	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

Table 29: MIPS64 *COP1* Encoding of rs Field

rs		*bits 23..21*							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC1	DMFC1 ⊥	CFC1	*	MTC1	DMTC1 ⊥	CTC1	*
1	01	*BC1* δ	*BC1ANY2* δε⊥	BC1ANY4 δε⊥	*	*	*	*	*
2	10	*S* δ	*D* δ	*	*	*W* δ	*L* δ⊥	*PS* δ⊥	*
3	11	*	*	*	*	*	*	*	*

Table 30: MIPS64 *COP1* Encoding of Function Field When rs=*S*

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	ROUND.L ⊥	TRUNC.L ⊥	CEIL.L ⊥	FLOOR.L ⊥	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	*MOVCF* δ	MOVZ	MOVN	*	RECIP ⊥	RSQRT ⊥	*
3	011	*	*	*	*	RECIP2 ε⊥	RECIP1 ε⊥	RSQRT1 ε⊥	RSQRT2 ε⊥
4	100	*	CVT.D	*	*	CVT.W	CVT.L ⊥	CVT.PS ⊥	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F CABS.F ε⊥	C.UN CABS.UN ε⊥	C.EQ CABS.EQ ε⊥	C.UEQ CABS.UEQ ε⊥	C.OLT CABS.OLT ε⊥	C.ULT CABS.ULT ε⊥	C.OLE CABS.OLE ε⊥	C.ULE CABS.ULE ε⊥
7	111	C.SF CABS.SF ε⊥	C.NGLE CABS.NGLE ε⊥	C.SEQ CABS.SEQ ε⊥	C.NGL CABS.NGL ε⊥	C.LT CABS.LT ε⊥	C.NGE CABS.NGE ε⊥	C.LE CABS.LE ε⊥	C.NGT CABS.NGT ε⊥

Table 31: MIPS64 *COP1* Encoding of Function Field When rs=*D*

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	ROUND.L ⊥	TRUNC.L ⊥	CEIL.L ⊥	FLOOR.L ⊥	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	*MOVCF* δ	MOVZ	MOVN	*	RECIP ⊥	RSQRT ⊥	*
3	011	*	*	*	*	RECIP2 ε⊥	RECIP1 ε⊥	RSQRT1 ε⊥	RSQRT2 ε⊥
4	100	CVT.S	*	*	*	CVT.W	CVT.L ⊥	*	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F CABS.F ε⊥	C.UN CABS.UN ε⊥	C.EQ CABS.EQ ε⊥	C.UEQ| CABS.UEQ ε⊥	C.OLT CABS.OLT ε⊥	C.ULT CABS.ULT ε⊥	C.OLE CABS.OLE ε⊥	C.ULE CABS.ULE ε⊥
7	111	C.SF CABS.SF ε⊥	C.NGLE CABS.NGLE ε⊥	C.SEQ CABS.SEQ ε⊥	C.NGL CABS.NGL ε⊥	C.LT CABS.LT ε⊥	C.NGE CABS.NGE ε⊥	C.LE CABS.LE ε⊥	C.NGT CABS.NGT ε⊥

Table 32: MIPS64 *COP1* Encoding of Function Field When rs=*W or L*[a]

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	*	*	*	*	*	*	*	*
1	001	*	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	CVT.S	CVT.D	*	*	*	*	CVT.PS.PW ε⊥	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

a. Format type L is legal only if 64-bit operations are enabled.

Table 33: MIPS64 *COP1* Encoding of Function Field When rs=*PS*[a]

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	ADD	SUB	MUL	*	*	ABS	MOV	NEG
1	001	*	*	*	*	*	*	*	*
2	010	*	*MOVCF* δ	MOVZ	MOVN	*	*	*	*
3	011	ADDR ε	*	MULR ε	*	RECIP2 ε	RECIP1 ε	RSQRT1 ε	RSQRT2 ε
4	100	CVT.S.PU	*	*	*	CVT.PW.PS ε	*	*	*
5	101	CVT.S.PL	*	*	*	PLL.PS	PLU.PS	PUL.PS	PUU.PS
6	110	C.F CABS.F ε	C.UN CABS.UN ε	C.EQ CABS.EQ ε	C.UEQ CABS.UEQ ε	C.OLT CABS.OLT ε	C.ULT CABS.ULT ε	C.OLE CABS.OLE ε	C.ULE CABS.ULE ε
7	111	C.SF CABS.SF ε	C.NGLE CABS.NGLEε	C.SEQ CABS.SEQ ε	C.NGL CABS.NGL ε	C.LT CABS.LT ε	C.NGE CABS.NGE ε	C.LE CABS.LE ε	C.NGT CABS.NGT ε

a. Format type PS is legal only if 64-bit operations are enabled.

Table 34: MIPS64 *COP1* Encoding of tf Bit When rs=*S, D, or PS,* Function=*MOVCF*

tf	bit 16	
	0	1
	MOVF.fmt	MOVT.fmt

Table 35: MIPS64 *COP1X* Encoding of Function Field[a]

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	LWXC1	LDXC1	*	*	*	LUXC1	*	*
1	001	SWXC1	SDXC1	*	*	*	SUXC1	*	PREFX
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	ALNV.PS	*
4	100	MADD.S	MADD.D	*	*	*	*	MADD.PS	*
5	101	MSUB.S	MSUB.D	*	*	*	*	MSUB.PS	*
6	110	NMADD.S	NMADD.D	*	*	*	*	NMADD.PS	*
7	111	NMSUB.S	NMSUB.D	*	*	*	*	NMSUB.PS	*

a. COP1X instructions are legal only if 64-bit operations are enabled.

2.9 MIPS64 Instruction Descriptions

As described earlier, this specification does not include instruction descriptions for all instructions that are in the MIPS64 ISA. Rather, it includes by reference the MIPS RISC Architecture document for the majority of the instructions. The following sections describe only those ISA-related features and any instructions that are new or modified by their inclusion in MIPS64.

2.9.1 UNPREDICTABLE and UNDEFINED

The terms **UNPREDICTABLE** and **UNDEFINED** are used throughout this specification to describe the behavior of the processor in certain cases. **UNDEFINED** behavior or operations can occur only as the result of executing instructions in a privileged mode (i.e., in Kernel Mode or Debug Mode, or with the CP0 usable bit set in the Status register). Unprivileged software can never cause **UNDEFINED** behavior or operations. Conversely, both privileged and unprivileged software can cause **UNPREDICTABLE** results or operations.

2.9.1.1 UNPREDICTABLE

UNPREDICTABLE results may vary from implementation to implementation, instruction to instruction, or as a function of time on the same implementation or instruction. Software can never depend on results that are **UNPREDICTABLE. UNPREDICTABLE** operations may cause a result to be generated or not. If a result is generated, it is **UNPREDICTABLE. UNPREDICTABLE** operations may cause arbitrary exceptions.

UNPREDICTABLE results or operations have several implementation restrictions:

- **UNPREDICTABLE** results must not depend on any data source (memory or internal state) which are inaccessible in the current processor mode.
- **UNPREDICTABLE** operations must not read, write, or modify the contents of memory or internal state which is inaccessible in the current processor mode. For example, **UNPREDICTABLE** operations executed in user mode must not access memory or internal state that is only accessible in Kernel Mode or Debug Mode or in another process.
- **UNPREDICTABLE** operations must not halt or hang the processor.

2.9.1.2 UNDEFINED

UNDEFINED operations or behavior may vary from implementation to implementation, instruction to instruction, or as a function of time on the same implementation or instruction. **UNDEFINED** operations or behavior may vary from nothing to creating an environment in which execution can no longer continue. **UNDEFINED** operations or behavior may cause data loss.

UNDEFINED operations or behavior has one implementation restriction:

- **UNDEFINED** operations or behavior must not cause the processor to hang (that is, enter a state from which there is no exit other than powering down the processor). The assertion of any of the reset signals must restore the processor to an operational state.

2.9.2 Unprivileged Instructions

2.9.2.1 The BCz instruction

Branch on Coprocessor Condition **BCz**

31 26	25 21	20 16	15 0
COPz 0100zz	BC 01000	Cond	Offset
6	5	5	16

Format:

BCz*Cond* offset **MIPS64**

Purpose:

Branch to the specified address if the coprocessor condition is met.

Description:

A 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. Branch to that address if the coprocessor *z* condition is met. The *cond* field is specific to each coprocessor and determines the condition.

Note that the BCz instruction is actually a class of instructions, one for each coprocessor number specified by *z*, and including taken and not taken variants.

Restrictions:

If the coprocessor enable bit for coprocessor *z* is zero in the *Status* register, access to this coprocessor is not allowed, and execution of this instruction results in a Coprocessor Unusable Exception. If the processor is running in Kernel Mode or Debug Mode, access to coprocessor 0 is enabled even if the CU0 bit is zero in the *Status* register.

For coprocessor 0, this instruction is not valid and results in a Reserved Instruction Exception if access is allowed to coprocessor 0. A Reserved Instruction Exception is also initiated if BCz is not implemented for coprocessor *z*.

Operation:

```
I:  if ($Status_{CUz}$ = 1) or
        ((z = 0) and (($Status_{KSU}$= $00_2$) or ($Debug_{DM}$ = 1) or ($Status_{EXL}$ = 1) or ($Status_{ERL}$ = 1))) then
          # Access allowed to coprocessor z
          if ((z = 0) or (BCz Not Implemented)) then
              InitiateReservedInstructionException
          endif
          target_offset ← sign_extend(offset || $0^2$)
    else
          # Access not allowed to coprocessor z
          InitiateCoprocessorUnusableException(z)
    endif
I+1:If COPz.Condition[Cond] then
          PC ← PC + target_offset
    endif
```

Exceptions:

Coprocessor Unusable Exception (Access not allowed to coprocessor)

Reserved Instruction Exception (Access allowed to coprocessor and coprocessor 0 or BCz not implemented for this coprocessor)

2.9.2.2 The CFCz Instruction

Move Control from Coprocessor z **CFCz**

31 26	25 21	20 16	15 11	10 0
COPz 0 1 0 0 z z	CF 0 0 0 1 0	rt	rd	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

CFCz rt, rd **MIPS64**

Purpose:

Move the contents of a coprocessor control register to a general register.

Description:

The contents of the control register *rd* of coprocessor *z* are sign-extended and loaded into general register *rt*.

Restrictions:

If the coprocessor enable bit for coprocessor *z* is zero in the *Status* register, access to this coprocessor is not allowed, and execution of this instruction results in a Coprocessor Unusable Exception. If the processor is running in Kernel Mode or Debug Mode, access to coprocessor 0 is enabled even if the CU0 bit is zero in the *Status* register.

For coprocessor 0, this instruction is not valid and results in a Reserved Instruction Exception if access is allowed to coprocessor 0. A Reserved Instruction Exception is also initiated if CFCz is not implemented for coprocessor *z*.

The results are **UNPREDICTABLE** if coprocessor z does not contain a control register as specified by rd.

Operation:

```
if (Status$_{CUz}$ = 1) or
  ((z = 0) and ((Status$_{KSU}$= 00$_2$) or (Debug$_{DM}$ = 1) or (Status$_{EXL}$ = 1) or (Status$_{ERL}$ = 1))) then
    # Access allowed to coprocessor z
    if ((z = 0) or (CFCz Not Implemented)) then
        InitiateReservedInstructionException
    endif
    temp $\leftarrow$ CCR[z,rd]
    GPR[rt] $\leftarrow$ sign_extend(temp)
else
    # Access not allowed to coprocessor z
    InitiateCoprocessorUnusableException(z)
endif
```

Exceptions:

Coprocessor Unusable Exception (Access not allowed to coprocessor)

Reserved Instruction Exception (Access allowed to coprocessor and coprocessor 0 or CFCz not implemented for this coprocessor)

2.9.2.3 The CLO Instruction)

Count Leading Ones in Word **CLO**

31 26	25 21	20 16	15 11	10 6	5 0
SPEC2 011100	rs	rt	rd	0 00000	CLO 100001
6	5	5	5	5	6

Format:

CLO rd, rs **MIPS64**

Purpose:

Count the number of leading ones in a word

Description:

Bits 31..0 of GPR *rs* are scanned from most significant to least significant bit. The number of leading ones is counted and the result is written to GPR *rd*. If all of bits 31..0 were set in GPR *rs*, the result written to GPR *rd* is 32.

Architecture Change
Hardware/Software Implementation Note

In an earlier release of this document, the destination GPR of this instruction was specified by the *rt* field. In order to align the definition of this instruction with other similar instructions, the architecture has changed to specify the destination GPR with the *rd* field. The following items provide a transparent transition between previous and current architecture definitions:

- **Software must place the same GPR number in both the *rt* and *rd* fields of the instruction. This will guarantee correct execution on implementations of both previous and current architecture definitions. This is required to be compliant with the MIPS32 and MIPS64 architecture.**
- **New processor designs should use the *rd* field as the destination GPR number.**
- **Current processor designs should be changed to reflect the new definition to the extent that it is convenient to do so.**

Restrictions:

If GPR *rs* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

Operation:

```
if NotWordValue(GPR[rs]) then
    UNPREDICTABLE
endif
temp ← 32
for i in 31 .. 0
    if $GPR[rs]_i = 0$ then
        temp ← 31 - i
        break
    endif
endfor
GPR[rd] ← temp
```

Exceptions:

None

2.9.2.4 The CLZ Instruction

Count Leading Zeros in Word **CLZ**

31 26	25 21	20 16	15 11	10 6	5 0
SPEC2 011100	rs	rt	rd	0 00000	CLZ 100000
6	5	5	5	5	6

Format:

CLZ rd, rs **MIPS64**

Purpose

Count the number of leading zeros in a word

Description:

Bits 31..0 of GPR *rs* are scanned from most significant to least significant bit. The number of leading zeros is counted and the result is written to GPR *rd*. If none of bits 31..0 were set in GPR *rs*, the result written to GPR *rd* is 32.

Architecture Change
Hardware/Software Implementation Note

In an earlier release of this document, the destination GPR of this instruction was specified by the *rt* field. In order to align the definition of this instruction with other similar instructions, the architecture has changed to specify the destination GPR with the *rd* field. The following items provide a transparent transition between previous and current architecture definitions:

- **Software must place the same GPR number in both the *rt* and *rd* fields of the instruction. This will guarantee correct execution on implementations of both previous and current architecture definitions. This is required to be compliant with the MIPS32 and MIPS64 architecture.**
- **New processor designs should use the *rd* field as the destination GPR number.**
- **Current processor designs should be changed to reflect the new definition to the extent that it is convenient to do so.**

Restrictions:

If GPR *rs* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

Operation:

```
if NotWordValue(GPR[rs]) then
    UNPREDICTABLE
endif
temp ← 32
for i in 31 .. 0
    if $\text{GPR[rs]}_i$ = 1 then
        temp ← 31 - i
        break
    endif
endfor
GPR[rd] ← temp
```

Exceptions:

None

2.9.2.5 The COPz Instruction

Coprocessor Operation for Coprocessor z **COPz**

31 26	25	24 0
COPz 0 1 0 0 z z	CO 1	Coprocessor Function
6	1	25

Format:

COPz rt, rd **MIPS64**

Purpose:

Perform the coprocessor function specified by Bits [24:0].

Description:

A coprocessor function, as described by Bits [24:0], is performed that is specific to coprocessor *z*. Refer to the instruction descriptions for each coprocessor for more details.

Restrictions:

If the coprocessor enable bit for coprocessor *z* is zero in the *Status* register, access to this coprocessor is not allowed, and execution of this instruction results in a Coprocessor Unusable Exception. If the processor is running in Kernel Mode or Debug Mode, access to coprocessor 0 is enabled even if the CU0 bit is zero in the *Status* register.

A Reserved Instruction Exception is taken if access is allowed to coprocessor *z* and COPz is not implemented for that coprocessor.

Operation:

```
if ($Status_{CUz}$ = 1) or
  ((z = 0) and (($Status_{KSU}$= $00_2$) or ($Debug_{DM}$ = 1) or ($Status_{EXL}$ = 1) or ($Status_{ERL}$ = 1))) then
    # Access allowed to coprocessor z
    if (COPz Not Implemented) then
        InitiateReservedInstructionException
    endif
    CoprocessorOperation(z, CoprocessorFunction)
else
    # Access not allowed to coprocessor z
    InitiateCoprocessorUnusableException(z)
endif
```

Exceptions:

Coprocessor Unusable Exception (Access not allowed to coprocessor)

Reserved Instruction Exception (Access allowed, and COPz not implemented for this coprocessor)

2.9.2.6 The CTCz Instruction

Move Control to Coprocessor z **CTCz**

31 26	25 21	20 16	15 11	10 0
COPz 0100zz	CT 00110	rt	rd	0 00000000000
6	5	5	5	11

Format:

CTCz rt, rd **MIPS64**

Purpose:

Move the contents of a general register to a coprocessor control register.

Description:

Bits 31..0 of GPR *rt* are loaded into the control register *rd* of coprocessor *z*.

Restrictions:

If the coprocessor enable bit for coprocessor *z* is zero in the *Status* register, access to this coprocessor is not allowed, and execution of this instruction results in a Coprocessor Unusable Exception. If the processor is running in Kernel Mode or Debug Mode, access to coprocessor 0 is enabled even if the CU0 bit is zero in the *Status* register.

For coprocessor 0, this instruction is not valid and results in a Reserved Instruction Exception if access is allowed to coprocessor 0. A Reserved Instruction Exception is also initiated if CTCz is not implemented for coprocessor z.

The results are **UNPREDICTABLE** if coprocessor z does not contain a control register as specified by rd.

Operation:

```
if ($Status_{CUz}$ = 1) or
  ((z = 0) and (($Status_{KSU}$= $00_2$) or ($Debug_{DM}$ = 1) or ($Status_{EXL}$ = 1) or ($Status_{ERL}$ = 1))) then
    # Access allowed to coprocessor z
    if ((z = 0) or (CTCz Not Implemented)) then
        InitiateReservedInstructionException
    endif
    temp ← GPR[rt]
    CCR[z,rd] ← temp
else
    # Access not allowed to coprocessor z
    InitiateCoprocessorUnusableException(z)
endif
```

Exceptions:

Coprocessor Unusable Exception (Access not allowed to coprocessor)

Reserved Instruction Exception (Access allowed to coprocessor and coprocessor 0 or CTCz not implemented for this coprocessor)

2.9.2.7 The DCLO Instruction

Count Leading Ones in Doubleword **DCLO**

31 26	25 21	20 16	15 11	10 6	5 0
SPEC2 011100	rs	rt	rd	0 00000	DCLO 100101
6	5	5	5	5	6

Format:

DCLO rd, rs **MIPS64**

Purpose:

Count the number of leading ones in a doubleword

Description:

Bits 63..0 of GPR *rs* are scanned from most significant to least significant bit. The number of leading ones is counted and the result is written to GPR *rd*. If all of bits 63..0 were set in GPR *rs*, the result written to GPR *rd* is 64.

Architecture Change
Hardware/Software Implementation Note

In an earlier release of this document, the destination GPR of this instruction was specified by the *rt* field. In order to align the definition of this instruction with other similar instructions, the architecture has changed to specify the destination GPR with the *rd* field. The following items provide a transparent transition between previous and current architecture definitions:

- **Software must place the same GPR number in both the *rt* and *rd* fields of the instruction. This will guarantee correct execution on implementations of both previous and current architecture definitions. This is required to be compliant with the MIPS32 and MIPS64 architecture.**
- **New processor designs should use the *rd* field as the destination GPR number.**
- **Current processor designs should be changed to reflect the new definition to the extent that it is convenient to do so.**

Restrictions:

This instruction is not legal unless access to 64-bit operations is enabled. If access is not enabled, execution results in a Reserved Instruction Exception.

Operation:

```
if not MIPS64OperationsEnabled() then
    InitiateReservedInstructionException()
endif
temp ← 64
for i in 63 .. 0
    if $GPR[rs]_i$ = 0 then
        temp ← 63 - i
        break
    endif
endfor
GPR[rd] ← temp
```

Exceptions:

Reserved Instruction Exception

2.9.2.8 The DCLZ Instruction

Count Leading Zeros in Doubleword **DCLZ**

31 26	25 21	20 16	15 11	10 6	5 0
SPEC2 011100	rs	rt	rd	0 00000	DCLZ 100100
6	5	5	5	5	6

Format:

DCLZ rd, rs **MIPS64**

Purpose

Count the number of leading zeros in a doubleword

Description:

Bits 63..0 of GPR *rs* are scanned from most significant to least significant bit. The number of leading zeros is counted and the result is written to GPR *rd*. If none of bits 63..0 were set in GPR *rs*, the result written to GPR *rd* is 64.

Architecture Change
Hardware/Software Implementation Note

In an earlier release of this document, the destination GPR of this instruction was specified by the *rt* field. In order to align the definition of this instruction with other similar instructions, the architecture has changed to specify the destination GPR with the *rd* field. The following items provide a transparent transition between previous and current architecture definitions:

- **Software must place the same GPR number in both the *rt* and *rd* fields of the instruction. This will guarantee correct execution on implementations of both previous and current architecture definitions. This is required to be compliant with the MIPS32 and MIPS64 architecture.**
- **New processor designs should use the *rd* field as the destination GPR number.**
- **Current processor designs should be changed to reflect the new definition to the extent that it is convenient to do so.**

Restrictions:

This instruction is not legal unless access to 64-bit operations is enabled. If access is not enabled, execution results in a Reserved Instruction Exception.

Operation:

```
if not MIPS64OperationsEnabled() then
    InitiateReservedInstructionException()
endif
temp ← 64
for i in 63 .. 0
    if $GPR[rs]_i = 1$ then
        temp ← 63 - i
        break
    endif
endfor
GPR[rd] ← temp
```

Exceptions:

Reserved Instruction Exception

2.9.2.9 The DMFCz Instruction

Doubleword Move from Coprocessor z **DMFCz**

31 26	25 21	20 16	15 11	10 3	2 0
COPz 0100zz	DMF 00001	rt	rd	0 00000000	sel
6	5	5	5	8	3

Format:

DMFCz rt, rd **MIPS64**
DMFCz rt, rd, sel

Purpose:

Move the contents of a coprocessor register to a general register.

Description:

The contents of the coprocessor z register specified by the combination of *rd* and *sel* are loaded into general register *rt*. Not all coprocessors or registers within a coprocessor support the sub-selection specified by the *sel* field. In those instances, the *sel* field must be set to zero

Restrictions:

If the coprocessor enable bit for coprocessor *z* is zero in the *Status* register, access to this coprocessor is not allowed, and execution of this instruction results in a Coprocessor Unusable Exception. If the processor is running in Kernel Mode or Debug Mode, access to coprocessor 0 is enabled even if the CU0 bit is zero in the *Status* register.

A Reserved Instruction Exception is taken if access is allowed to coprocessor *z* and DMFCz is not implemented for that coprocessor or if access to 64-bit operations is not enabled.

The results are **UNPREDICTABLE** if coprocessor z does not contain a register as specified by *rd* and *sel* or if the coprocessor z register specified by *rd* and *sel* is a 32-bit register.

Operation:

```
if ($Status_{CUz}$ = 1) or
 ((z = 0) and (($Status_{KSU}$ = $00_2$) or ($Debug_{DM}$ = 1) or ($Status_{EXL}$ = 1) or ($Status_{ERL}$ = 1))) then
    if ((DMFCz Not Implemented) or
       (not MIPS64OperationsEnabled()) then
        InitiateReservedInstructionException
    endif
    data ← CPR[z,rd,sel]
    GPR[rt] ← data.
else
    InitiateCoprocessorUnusableException(z)
endif
```

Exceptions:

Coprocessor Unusable Exception (Access not allowed to coprocessor)

Reserved Instruction Exception (access allowed, and DMFCz not implemented for this coprocessor or access to 64-bit operations is not enabled)

2.9.2.10 The DMTCz Instruction

Doubleword Move to Coprocessor z **DMTCz**

31 26	25 21	20 16	15 11	10 3	2 0
COPz 0100zz	DMT 00101	rt	rd	0 0000000	sel
6	5	5	5	8	3

Format:

DMTCz rt, rd **MIPS64**
DMTCz rt, rd, sel

Purpose:

Move the contents of a general register to a coprocessor register.

Description:

The contents of general register *rt* are loaded into the coprocessor z register specified by the combination of *rd* and *sel*. Not all coprocessors or registers within a coprocessor support the sub-selection specified by the *sel* field. In those instances, the *sel* field must be set to zero.

Restrictions:

If the coprocessor enable bit for coprocessor *z* is zero in the *Status* register, access to this coprocessor is not allowed, and execution of this instruction results in a Coprocessor Unusable Exception. If the processor is running in Kernel Mode or Debug Mode, access to coprocessor 0 is enabled even if the CU0 bit is zero in the *Status* register.

A Reserved Instruction Exception is taken if access is allowed to coprocessor *z* and DMTCz is not implemented for that coprocessor or if access to 64-bit operations is not enabled.

The results are **UNPREDICTABLE** if coprocessor z does not contain a register as specified by *rd* and *sel* or if the coprocessor z register specified by *rd* and *sel* is a 32-bit register.

Operation:

```
if ($Status_{CUz}$ = 1) or
 ((z = 0) and (($Status_{KSU}$= $00_2$) or ($Debug_{DM}$ = 1) or ($Status_{EXL}$ = 1) or ($Status_{ERL}$ = 1))) then
    if ((DMTCz Not Implemented) or
        (not MIPS64OperationsEnabled()) then
         InitiateReservedInstructionException
    endif
    data ← GPR[rt]
    CPR[z,rd,sel] ← data
else
    InitiateCoprocessorUnusableException(z)
endif
```

Exceptions:

Coprocessor Unusable Exception (Access not allowed to coprocessor)

Reserved Instruction Exception (access allowed, and DMTCz not implemented for this coprocessor or access to 64-bit operations is not enabled)

2.9.2.11 The MADD Instruction

Multiply and Add Word to Hi,Lo **MADD**

31 26	25 21	20 16	15 11	10 6	5 0
SPEC2 011100	rs	rt	0 00000	0 00000	MADD 000000
6	5	5	5	5	6

Format:

MADD rs, rt **MIPS64**

Purpose:

Multiply two words and add the result to Hi, Lo

Description:

The 32-bit word value in GPR *rs* is multiplied by the 32-bit value in GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The product is added to the 64-bit concatenated values of $HI_{31..0}$ and $LO_{31..0.}$ The most significant 32 bits of the result is sign-extended and written into *HI* and the least significant 32 bits of the result is sign-extended and written into *LO*. No arithmetic exception occurs under any circumstances.

Restrictions:

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE.**

Note that this instruction does not provide the capability of writing directly to a target GPR.

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← ($HI_{31..0}$ || $LO_{31..0}$) + ($GPR[rs]_{31..0}$ * $GPR[rt]_{31..0}$)
HI ← sign_extend($temp_{63..32}$)
LO ← sign_extend($temp_{31..0}$)
```

Exceptions:

None

Implementation Note:

Processors which implement a multiplier array which is not square (e.g., 32 x 16), and which therefore has an operation latency which is data dependent, should assume that the shorter operand is in GPR *rt*.

Programming Note:

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

2.9.2.12 The MADDU Instruction

Multiply and Add Unsigned Word to Hi,Lo **MADDU**

31 26	25 21	20 16	15 11	10 6	5 0
SPEC2 011100	rs	rt	0 00000	0 00000	MADDU 000001
6	5	5	5	5	6

Format:

MADDU rs, rt **MIPS64**

Purpose:

Multiply two unsigned words and add the result to Hi, Lo

Description:

The 32-bit word value in GPR *rs* is multiplied by the 32-bit value in GPR *rt*, treating both operands as unsigned values, to produce a 64-bit result. The product is added to the 64-bit concatenated values of $HI_{31..0}$ and $LO_{31..0}$. The most significant 32 bits of the result is sign-extended and written into *HI* and the least significant 32 bits of the result is sign-extended and written into *LO*. No arithmetic exception occurs under any circumstances.

Restrictions:

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

Note that this instruction does not provide the capability of writing directly to a target GPR.

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← $(HI_{31..0} \| LO_{31..0}) + ((0^{32} \| GPR[rs]_{31..0}) * (0^{32} \| GPR[rt]_{31..0}))$
HI ← sign_extend($temp_{63..32}$)
LO ← sign_extend($temp_{31..0}$)
```

Exceptions:

None

Implementation Note:

Processors which implement a multiplier array which is not square (e.g., 32 x 16), and which therefore has an operation latency which is data dependent, should assume that the shorter operand is in GPR *rt*.

Programming Note:

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

2.9.2.13 The MFCz Instruction

Move from Coprocessor z **MFCz**

31 26	25 21	20 16	15 11	10 3	2 0
COPz 0100zz	MF 00000	rt	rd	0 00000000	sel
6	5	5	5	8	3

Format:

MFCz rt, rd **MIPS64**
MFCz rt, rd, sel

Purpose:

Move the contents of a coprocessor register to a general register.

Description:

The contents of the least significant 32 bits of the coprocessor z register specified by the combination of *rd* and *sel* are sign-extended and loaded into general register *rt*. Not all coprocessors or registers within a coprocessor support the sub-selection specified by the *sel* field. In those instances, the *sel* field must be set to zero

Restrictions:

If the coprocessor enable bit for coprocessor *z* is zero in the *Status* register, access to this coprocessor is not allowed, and execution of this instruction results in a Coprocessor Unusable Exception. If the processor is running in Kernel Mode or Debug Mode, access to coprocessor 0 is enabled even if the CU0 bit is zero in the *Status* register.

A Reserved Instruction Exception is taken if access is allowed to coprocessor *z* and MFCz is not implemented for that coprocessor.

The results are **UNPREDICTABLE** if coprocessor z does not contain a register as specified by *rd* and *sel*.

Operation:

```
if ($\text{Status}_{\text{CUz}}$ = 1) or
  ((z = 0) and (($\text{Status}_{\text{KSU}} = 00_2$) or ($\text{Debug}_{\text{DM}}$ = 1) or ($\text{Status}_{\text{EXL}}$ = 1) or ($\text{Status}_{\text{ERL}}$ = 1))) then
    if (MFCz Not Implemented) then
        InitiateReservedInstructionException
    endif
    data ← CPR[z,rd,sel]
    GPR[rt] ← sign_extend(data)
else
    InitiateCoprocessorUnusableException(z)
endif
```

Exceptions:

Coprocessor Unusable Exception (Access not allowed to coprocessor)
Reserved Instruction Exception (access allowed, and MFCz not implemented for this coprocessor)

2.9.2.14 The MOVN.PS Instruction

Floating Point Move Conditional on Not Zero **MOVN.PS**

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PS 10110	rt	fs	fd	MOVN 010011
6	5	5	5	5	6

Format:

MOVN.PS fd, fs, rt **MIPS64**

Purpose:

To test a GPR then conditionally move an FP value.

Description:

If the value in GPR *rt* is not equal to zero, then the value in FPR *fs* is placed in FPR *fd*. The source and destination are values in format PS.

If GPR *rt* contains zero, then FPR *fs* is not copied and FPR *fd* contains its previous value in format PS. If FPR *fd* did not contain a value either in format PS or previously unused data from a load or move-to operation that could be interpreted in format PS, then the value of FPR *fd* becomes **UNPREDICTABLE**.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type PS; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format PS; if it is not, the result is **UNPREDICTABLE** and the value of FPR *fs* becomes **UNPREDICTABLE**.

Operation:

```
if GPR[rt] ≠ 0 then
    StoreFPR(fd, PS, ValueFPR(fs, PS))
else
    StoreFPR(fd, PS, ValueFPR(fd, PS))
endif
```

Exceptions:

Coprocessor Unusable Exception (Access not allowed to coprocessor)
Reserved Instruction Exception (access allowed, but 64-bit operations are not enabled or the paired single format is not implemented)

2.9.2.15 The MOVZ.PS Instruction

Floating Point Move Conditional on Zero **MOVZ.PS**

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	PS 10110	rt	fs	fd	MOVZ 010010
6	5	5	5	5	6

Format:

MOVZ.PS fd, fs, rt **MIPS64**

Purpose:

To test a GPR then conditionally move an FP value.

Description:

If the value in GPR *rt* is equal to zero, then the value in FPR *fs* is placed in FPR *fd*. The source and destination are values in format PS.

If GPR *rt* is not zero, then FPR *fs* is not copied and FPR *fd* contains its previous value in format PS. If FPR *fd* did not contain a value either in format PS or previously unused data from a load or move-to operation that could be interpreted in format PS, then the value of FPR *fd* becomes **UNPREDICTABLE.**

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields *fs* and *fd* must specify FPRs valid for operands of type PS; if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format PS; if it is not, the result is **UNPREDICTABLE** and the value of FPR *fs* becomes **UNPREDICTABLE**.

Operation:

```
if GPR[rt] = 0 then
    StoreFPR(fd, PS, ValueFPR(fs, PS))
else
    StoreFPR(fd, PS, ValueFPR(fd, PS))
endif
```

Exceptions:

Coprocessor Unusable Exception (Access not allowed to coprocessor)
Reserved Instruction Exception (access allowed, but 64-bit operations are not enabled or the paired single format is not implemented)

2.9.2.16 The MSUB Instruction

Multiply and Subtract Word to Hi,Lo **MSUB**

31 26	25 21	20 16	15 11	10 6	5 0
SPEC2 011100	rs	rt	0 00000	0 00000	MSUB 000100
6	5	5	5	5	6

Format:

MSUB rs, rt **MIPS64**

Purpose:

Multiply two words and subtract the result from Hi, Lo

Description:

The 32-bit word value in GPR *rs* is multiplied by the 32-bit value in GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values of $HI_{31..0}$ and $LO_{31..0}$. The most significant 32 bits of the result is sign-extended and written into *HI* and the least significant 32 bits of the result is sign-extended and written into *LO*. No arithmetic exception occurs under any circumstances.

Restrictions:

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.

Note that this instruction does not provide the capability of writing directly to a target GPR.

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← $(HI_{31..0} || LO_{31..0}) - (GPR[rs]_{31..0} * GPR[rt]_{31..0})$
HI ← sign_extend($temp_{63..32}$)
LO ← sign_extend($temp_{31..0}$)
```

Exceptions:

None

Implementation Note:

Processors which implement a multiplier array which is not square (e.g., 32 x 16), and which therefore has an operation latency which is data dependent, should assume that the shorter operand is in GPR *rt*.

Programming Note:

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

2.9.2.17 The MSUBU Instruction

Multiply and Subtract Unsigned Word to Hi,Lo **MSUBU**

31 26	25 21	20 16	15 11	10 6	5 0
SPEC2 011100	rs	rt	0 00000	0 00000	MSUBU 000101
6	5	5	5	5	6

Format:

MSUBU rs, rt **MIPS64**

Purpose:

Multiply two unsigned words and subtract the result from Hi, Lo

Description:

The 32-bit word value in GPR *rs* is multiplied by the 32-bit value in GPR *rt*, treating both operands as unsigned values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values of $HI_{31..0}$ and $LO_{31..0}$. The most significant 32 bits of the result is sign-extended and written into *HI* and the least significant 32 bits of the result is sign-extended and written into *LO*. No arithmetic exception occurs under any circumstances.

Restrictions:

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE.**

Note that this instruction does not provide the capability of writing directly to a target GPR.

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← $(HI_{31..0} \| LO_{31..0}) - ((0^{32} \| GPR[rs]_{31..0}) * (0^{32} \| GPR[rt]_{31..0}))$
HI ← sign_extend($temp_{63..32}$)
LO ← sign_extend($temp_{31..0}$)
```

Exceptions:

None

Implementation Note:

Processors which implement a multiplier array which is not square (e.g., 32 x 16), and which therefore has an operation latency which is data dependent, should assume that the shorter operand is in GPR *rt*.

Programming Note:

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

2.9.2.18 The MTCz Instruction

Move to Coprocessor z **MTCz**

31 26	25 21	20 16	15 11	10 3	2 0
COPz 0100zz	MT 00100	rt	rd	0 0000000	sel
6	5	5	5	8	3

Format:

MTCz rt, rd **MIPS64**
MTCz rt, rd, sel

Purpose:

Move the contents of a general register to a coprocessor register.

Description:

The contents of general register *rt* are loaded into the coprocessor z register specified by the combination of *rd* and *sel*. Not all coprocessors or registers within a coprocessor support the sub-selection specified by the *sel* field. In those instances, the *sel* field must be set to zero.

Restrictions:

If the coprocessor enable bit for coprocessor *z* is zero in the *Status* register, access to this coprocessor is not allowed, and execution of this instruction results in a Coprocessor Unusable Exception. If the processor is running in Kernel Mode or Debug Mode, access to coprocessor 0 is enabled even if the CU0 bit is zero in the *Status* register.

A Reserved Instruction Exception is taken if access is allowed to coprocessor *z* and MTCz is not implemented for that coprocessor.

The results are **UNPREDICTABLE** if coprocessor z does not contain a register as specified by *rd* and *sel*.

For coprocessor 0, this instruction writes all 64 bits of register *rt* into the coprocessor register specified by *rd* and *sel* if that register is a 64-bit register.

Operation:

```
if ($Status_{CUz}$ = 1) or
  ((z = 0) and (($Status_{KSU}$ = $00_2$) or ($Debug_{DM}$ = 1) or ($Status_{EXL}$ = 1) or ($Status_{ERL}$ = 1))) then
    if (MTCz Not Implemented) then
        InitiateReservedInstructionException
    endif
    data ← GPR[rt]
    if (z = 0) and (Width(CPR[z,rd,sel]) = 64) then
        CPR[z,rd,sel] ← data
    else
        CPR[z,rd,sel] ← $data_{31..0}$
    endif
else
    InitiateCoprocessorUnusableException(z)
endif
```

Exceptions:

Coprocessor Unusable Exception (Access not allowed to coprocessor)

Reserved Instruction Exception (access allowed, and MTCz not implemented for this coprocessor)

2.9.2.19 The MUL Instruction

Multiply Word to GPR **MUL**

31 26	25 21	20 16	15 11	10 6	5 0
SPEC2 011100	rs	rt	rd	0 00000	MUL 000010
6	5	5	5	5	6

Format:

MUL rd, rs, rt **MIPS64**

Purpose:

Multiply two words write the result to a GPR

Description:

The 32-bit word value in GPR *rs* is multiplied by the 32-bit value in GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The least significant 32 bits of the product are written to GPR *rd*. The contents of *HI* and *LO* are not defined after the operation. No arithmetic exception occurs under any circumstances.

Restrictions:

If GPRs *rs* or *rt* do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE.**

Note that this instruction does not provide the capability of writing the result to the HI and LO registers.

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← $GPR[rs]_{31..0}$ * $GPR[rt]_{31..0}$
GPR[rd] ← sign_extend($temp_{31..0}$)
HI ← UNPREDICTABLE
LO ← UNPREDICTABLE
```

Exceptions:

None

Implementation Note:

Processors which implement a multiplier array which is not square (e.g., 32 x 16), and which therefore has an operation latency which is data dependent, should assume that the shorter operand is in GPR *rt*.

Programming Note:

Where the size of the operands are known, software should place the shorter operand in GPR *rt*. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.

2.9.2.20 The PREF Instruction

Prefetch **PREF**

31 26	25 21	20 16	15 0
PREF 110011	base	hint	Offset
6	5	5	16

Format:

PREF hint, offset(base) **MIPS IV**

Purpose:

To move data between memory and cache

Description:

PREF adds the 16-bit signed *offset* to the contents of GPR *base* to form an effective byte address. The *hint* field supplies information about how the addressed data is to be manipulated.

PREF enables the processor to take some action as specified by the *hint* field, to improve program performance. The action taken for a specific PREF instruction is both system and context dependent. Any action, including doing nothing, is permitted as long as it does not change architecturally visible state or alter the meaning of a program. Implementations are expected either to do nothing, or take an action that increases the performance of the program.

PREF does not cause addressing-related exceptions. If it does happen to raise an exception condition, the exception condition is ignored. If an addressing-related exception condition is raised and ignored, no data movement occurs.

PREF never generates a memory operation for a location with an uncached memory access type.

For a cached location, the expected and useful action for the processor is to move a block of data between cache and the memory hierarchy. The size of the block transferred is implementation dependent, but software may assume that it is at least one cache block.

The following table defines the hint field values.

Table 36: PREF hint field encodings

Value	Name	Data Use and Desired PREF action
0	load	Use: Prefetched data is expected to be read (not modified) Action: Fetch data as if for a load.
1	store	Use: Prefetched data is expected to be stored or modified Action: Fetch data as if for a store.
2-3	Reserved	Reserved for future use - not available to implementations.

Table 36: PREF hint field encodings

Value	Name	Data Use and Desired PREF action
4	load_streamed	Use: Prefetched data is expected to be read (not modified) but not reused extensively; it "streams" through the cache Action: Fetch data as if for a load and place it in the cache so that it does not displace data prefetched as "retained"
5	store_streamed	Use: Prefetched data is expected to be stored or modified but not reused extensively; it "streams" through the cache Action: Fetch data as if for a store and place it in the cache so that it does not displace data prefetched as "retained"
6	load_retained	Use: Prefetched data is expected to be read (not modified) and reused extensively; it should be "retained" in the cache Action: Fetch data as if for a load and place it in the cache so that it is not displaced by data prefetched as "streamed"
7	store_retained	Use: Prefetched data is expected to be stored or modified and reused extensively; it should be "retained" in the cache Action: Fetch data as if for a store and place it in the cache so that it is not displaced by data prefetched as "streamed"
8-24	Reserved	Reserved for future use - not available to implementations.
25	writeback_invalidate (also known as nudge)	Use: Data is no longer to be expected to be used Action: For a writeback cache, schedule a writeback of any dirty data. At the completion of the writeback, mark as invalid the state of any cache lines written back.
26-31	Implementation Dependent	Unassigned by the Architecture - available for implementation dependent use

Restrictions:

None

Operation:

```
vAddr ← GPR[base] + sign_extend(offset)
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

Exceptions:

Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

Programming Notes:

Prefetch cannot access a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch does not cause addressing exceptions. It does not cause an exception to prefetch using a pointer before the validity of the pointer is determined.

Hint field encodings whose function is described as "streamed" or "retained" convey usage intent from software to hardware. Software should not assume that hardware will always prefetch data in an optimal way. If data is to be truly retained, software should use the Cache instruction to lock data into the cache.

Implementation Notes:

It is implementation dependent whether encodings of the hint field listed as "Implementation Dependent" or "Unimplemented" are treated as a NOP, or mapped to another valid encoding of the hint field.

Hint field encodings whose function is described as "streamed" or "retained" convey usage intent from software to hardware. Processors should make an attempt to take this information into account when prefetching data, but are not obligated to do so.

Processors should never implement the writeback_invalidate encoding of the hint field in such a way that the action moves data from memory hierarchy to the cache. This function should either take the action intended for the encoding (to schedule a possible writeback and subsequent invalidation) or treat the function as a NOP.

It is implementation dependent whether a data watch is triggered by a prefetch instruction whose address matches the Watch register address match conditions. The preferred implementation is not to match on the prefetch instruction.

2.9.2.21 The PREFX Instruction

Prefetch Indexed **PREFX**

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 010011	base	index	hint	0 00000	PREFX 001111
6	5	5	5	5	6

Format:

PREFX hint, index(base) **MIPS IV**

Purpose:

To move data between memory and cache

Description:

PREFX adds the contents of GPR index to the contents of GPR base to form an effective byte address. The *hint* field supplies information about how the addressed data is to be manipulated.

The only functional difference between the PREF and PREFX instructions is the addressing mode implemented by the two. Refer to the PREF instruction description for all other details, including the encoding of the *hint* field.

Note, however, that the prefx instruction is only available on processors that implement floating point, and should only be generated by compilers in situations in which the corresponding load and store indexed floating point instructions are generated.

Restrictions:

None

Operation:

```
if ($\text{Status}_{CU1}$ = 0) then
    InitiateCoprocessorUnusableException(1)
endif
vAddr ← GPR[base] + GPR[index]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

Exceptions:

Coprocessor Unusable Exception

Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

Programming Notes:

Refer to the corresponding section in the PREF instruction description.

Implementation Notes:

Refer to the corresponding section in the PREF instruction description.

2.9.2.22 The SSNOP Instruction

Superscalar Inhibit NOP **SSNOP**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	0 00000	0 00000	1 00001	SLL 000000
6	5	5	5	5	6

Format:

SSNOP **MIPS32**

Purpose:

Break superscalar issue on a superscalar processor

Description:

This instruction alters the instruction issue behavior on a superscalar processor by forcing the SSNOP instruction to single-issue. Notwithstanding implementation dependent issue rules, a processor must end the current instruction issue between the instruction previous to the SSNOP and the SSNOP. The SSNOP then issues alone in the next issue slot.

SSNOP is intended for use primarily to allow the programmer control over CP0 hazards by converting instructions into cycles in a superscalar processor. For example, to insert at least two cycles between an MTC0 and an ERET, one would use the following sequence:

```
mtc0
ssnop
ssnop
eret
```

Based on the normal issues rules of the processor, the MTC0 issues in cycle T. Because the SSNOP instructions must issue alone, they may issue no earlier than cycle T+1 and cycle T+2, respectively. Finally, the ERET issues no earlier than cycle T+3. Note that although the instruction after an SSNOP may issue no earlier than the cycle after the SSNOP is issued, that instruction may issue **later**. This is because other implementation-dependent issue rules may apply that prevent an issue in the next cycle. Processors should not introduce any unnecessary delay in issuing SSNOP instructions.

On a single-issue processor, this instruction is a nop that takes an issue slot.

Restrictions:

None

Operation:

Exceptions:

None

2.9.3 Privileged Instructions

2.9.3.1 The CACHE Instruction

Perform Cache Operation **CACHE**

31 26	25 21	20 16	15 0
CACHE 101111	base	op	Offset
6	5	5	16

Format:

CACHE op, offset(base) **MIPS64**

Purpose:

To perform the cache operation specified by op.

Description:

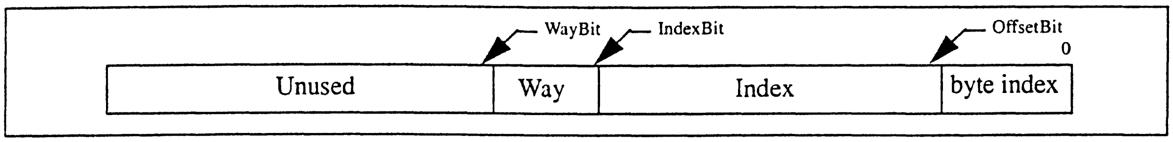
The 16-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used in one of three ways based on the operation to be performed and the type of cache as described in Table 37.

Table 37: Usage of Effective Address

Operation Requires an	Type of Cache	Usage of Effective Address
Address	Virtual	The effective address is used to address the cache. It is implementation dependent whether an address translation is performed on the effective address (with the possibility that a TLB Refill or TLB Invalid exception might occur)
Address	Physical	The effective address is translated by the MMU to a physical address. The physical address is then used to address the cache

Index	N/A	The effective address may be translated by the MMU to a physical address. It is implementation dependent whether the effective address or the translated physical address are used to index the cache.
		Assuming that the total cache size in bytes is CS, the associativity is A, and the number of bytes per tag is BPT, the following calculations give the fields of the address which specify the way and the index:
		OffsetBit ← Log2(BPT)
		IndexBit ← Log2(CS / A)
		WayBit ← IndexBit + Ceiling(Log2(A))
		Way ← $\text{Addr}_{\text{WayBit-1..IndexBit}}$
		Index ← $\text{Addr}_{\text{IndexBit-1..OffsetBit}}$
		For a direct-mapped cache, the Way calculation is ignored and the Index value fully specifies the cache tag. This is shown symbolically in Figure 4.

Figure 4: Usage of Address Fields to Select Index and Way



A TLB Refill and TLB Invalid (both with cause code equal TLBL) exception can occur on any operation. For index operations (where the address is used to index the cache but need not match the cache tag) software should use unmapped addresses to avoid TLB exceptions. This instruction never causes TLB Modified exceptions nor TLB Refill exceptions with a cause code of TLBS nor data Watch exceptions.

A Cache Error exception may occur as a byproduct of some operations performed by this instruction. For example, if a Writeback operation detects a cache or bus error during the processing of the operation, that error is reported via a Cache Error exception.

An Address Error Exception (with cause code equal AdEL) may occur if the effective address references a portion of the kernel address space which would normally result in such an exception. It is implementation dependent whether such an exception does occur.

It is implementation dependent whether a data watch is triggered by a cache instruction whose address matches the Watch register address match conditions. The preferred implementation is not to match on the cache instruction.

Bits [17:16] of the instruction specify the cache on which to perform the operation, as follows:

Table 38: Encoding of Bits[17:16] of CACHE Instruction

Code	Name	Cache
0 0	I	Primary Instruction
0 1	D	Primary Data or Unified Primary
1 0	T	Tertiary
1 1	S	Secondary

Bits [20:18] of the instruction specify the operation to perform. To provide software with a consistent base of cache operations, certain encodings must be supported on all processors. The remaining encodings are recommended.

Table 39: Encoding of Bits [20:18] of the CACHE Instruction

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance
0 0 0	I	Index Invalidate	Index	Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire instruction cache by stepping through all valid indices.	Required
	D	Index Writeback Invalidate / Index Invalidate	Index	For a write-back cache: If the state of the cache block at the specified index is valid and dirty, write the block back to the memory address specified by the cache tag. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid. For a write-through cache: Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire data cache by stepping through all valid indices.	Required
	S, T	Index Writeback Invalidate / Index Invalidate	Index		Optional
0 0 1	All	Index Load Tag	Index	Read the tag for the cache block at the specified index into the TagLo and TagHi COP0 registers. If the DataLo and DataHi registers are implemented, also read the data corresponding to the byte index into the DataLo and DataHi registers. The granularity and alignment of the data read into the DataLo and DataHi registers is implementation-dependent, but is typically the result of an aligned access to the cache, ignoring the appropriate low-order bits of the byte index.	Recommended

Table 39: Encoding of Bits [20:18] of the CACHE Instruction

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance
0 1 0	All	Index Store Tag	Index	Write the tag for the cache block at the specified index from the TagLo and TagHi COP0 registers. This required encoding may be used by software to initialize the entire instruction of data caches by stepping through all valid indices. Doing so requires that the *TagLo* and *TagHi* registers associated with the cache be initialized first.	Required
0 1 1				Available for implementation-dependent operation	Optional
1 0 0	I, D	Hit Invalidate	Address	If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the instruction cache by stepping through the address range by the line size of the cache.	Required (Instruction Cache Encoding Only), Recommended otherwise
	S, T	Hit Invalidate	Address		Optional
1 0 1	I	Fill	Address	Fill the cache from the specified address	Recommended
	D	Hit Writeback Invalidate / Hit Invalidate	Address	For a write-back cache: If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid. For a write-through cache: If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the data cache by stepping through the address range by the line size of the cache.	Required
	S, T	Hit Writeback Invalidate / Hit Invalidate	Address		Optional
1 1 0	D	Hit Writeback	Address	If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After the operation is completed, leave the state of the line valid, but clear the dirty state. For a write-through cache, this operation may be treated as a nop.	Recommended
	S, T	Hit Writeback	Address		Optional

Table 39: Encoding of Bits [20:18] of the CACHE Instruction

Code	Caches	Name	Effective Address Operand Type	Operation	Compliance
1 1 1	I, D	Fetch and Lock	Address	If the cache does not contain the specified address, fill it from memory, performing a writeback if required, and set the state to valid and locked. If the cache already contains the specified address, set the state to locked. In set-associative or fully-associative caches, the way selected on a fill from memory is implementation specific. The lock state may be cleared by executing an Index Invalidate, Index Writeback Invalidate, Hit Invalidate, or Hit Writeback Invalidate operation to the locked line, or via an Index Store Tag operation to the line that clears the lock bit. Note that clearing the lock state via Index Store Tag is dependent on the implementation-dependent cache tag and cache line organization, and that Index and Index Writeback Invalidate operations are dependent on cache line organization. Only Hit and Hit Writeback Invalidate operations are generally portable across implementations. It is implementation dependent whether a locked line is displaced as the result of an external invalidate or intervention that hits on the locked line. Software must not depend on the locked line remaining in the cache if an external invalidate or intervention would invalidate the line if it were not locked.	Recommended

Restrictions:

Execution of this instruction is legal only if the processor is operating in Kernel Mode or Debug Mode, or if the CP0 enable bit is set in the Status register. In other circumstances, a Coprocessor Unusable Exception is taken.

The operation of this instruction is **UNDEFINED** for any operation/cache combination that is not implemented.

The operation of this instruction is **UNDEFINED** for uncacheable addresses.

Operation:

```
if (Status$_{CU0}$ = 1) or (Status$_{KSU}$= $00_2$) or (Debug$_{DM}$ = 1) or (Status$_{EXL}$ = 1) or (Status$_{ERL}$ = 1) then
    vAddr ← GPR[base] + sign_extend(offset)
    (pAddr, uncached) ← AddressTranslation(vAddr, DataReadReference)
    CacheOp(op, vAddr, pAddr)
else
    InitiateCoprocessorUnusableException(0)
endif
```

Exceptions:

TLB Refill Exception.
TLB Invalid Exception
Coprocessor Unusable Exception
Address Error Exception
Cache Error Exception

2.9.3.2 The ERET Instruction

Exception Return **ERET**

31 26	25	24	0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	ERET 011000
6	1	19	6

Format:

ERET **MIPS64**

Purpose:

Return from interrupt, exception, or error trap

Description:

ERET returns to the interrupted instruction at the completion of interrupt, exception, or error trap processing. ERET does not execute the next instruction (i.e., it has no delay slot).

Restrictions:

The operation of the processor is **UNDEFINED** if an ERET is placed in the delay slot of a branch or jump instruction.

An ERET placed between an LL and SC instruction will always cause the SC to fail.

This instruction is legal only if the processor is in Kernel Mode or Debug Mode, or if the CP0 usable bit is set in the Status register. In other circumstances, execution of this instruction results in a Coprocessor Unusable Exception.

ERET implements a software barrier for all changes in the CP0 state that could affect the fetch and decode of the instruction at the PC to which the ERET returns, such as changes to the effective ASID, user-mode state, and addressing mode.

Operation:

```
if ($Status_{CU0} = 1$) or ($Status_{KSU} = 00_2$) or ($Debug_{DM} = 1$) or ($Status_{EXL} = 1$) or ($Status_{ERL} = 1$) then
    if $Status_{ERL} = 1$ then
        PC ← ErrorEPC
        $Status_{ERL} \leftarrow 0$
    else
        PC ← EPC
        $Status_{EXL} \leftarrow 0$
    endif
    LLbit ← 0
else
    InitiateCoprocessorUnusableException(0)
endif
```

Exceptions:

Coprocessor Unusable Exception

2.9.3.3 The TLBP Instruction

Probe TLB for Matching Entry **TLBP**

31 26	25	24	0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBP 001000
6	1	19	6

Format:

TLBP **MIPS64**

Purpose:

Find a matching entry in the TLB.

Description:

The *Index* register is loaded with the index of the TLB entry whose contents match the contents of the *EntryHi* register. If no TLB entry matches, the high-order bit of the *Index* register is set.

Restrictions:

This instruction is legal only if the processor is in Kernel Mode or Debug Mode, or if the CP0 usable bit is set in the *Status* register. In other circumstances, execution of this instruction results in a Coprocessor Unusable Exception.

For processors that do not include the standard TLB MMU, the operation of this instruction is **UNDEFINED**. However, the preferred implementation is a Reserved Instruction Exception.

Operation:

```
if (Status$_{CU0}$ = 1) or (Status$_{KSU}$= 00$_2$) or (Debug$_{DM}$ = 1) or (Status$_{EXL}$ = 1) or (Status$_{ERL}$ = 1) then
    Index ← 1 || UNPREDICTABLE$^{31}$
    for i in 0...TLBEntries-1
        if (TLB[i]$_R$ = EntryHi$_R$) and
          ((TLB[i]$_{VPN2}$ and not (TLB[i]$_{Mask}$)) =
           (EntryHi$_{VPN2}$ and not (TLB[i]$_{Mask}$))) and
          (TLB[i]$_G$ or (TLB[i]$_{ASID}$ = EntryHi$_{ASID}$)) then
            Index ← i
        endif
    endfor
else
    InitiateCoprocessorUnusableException(0)
endif
```

Exceptions:

Coprocessor Unusable Exception
Reserved Instruction Exception (if not implemented)
Machine Check (if implemented and a TLB shutdown condition is detected on a TLB read)

2.9.3.4 The TLBR Instruction

Read Indexed TLB Entry **TLBR**

31 26	25	24	0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBR 000001
6	1	19	6

Format:

TLBR **MIPS64**

Purpose:

Read an entry from the TLB.

Description:

The *EntryHi*, *EntryLo0*, *EntryLo1,* and *PageMask* registers are loaded with the contents of the TLB entry pointed to by the *Index* register. Note that the value written to the *EntryHi*, *EntryLo0*, and *EntryLo1* registers may be different from that originally written to the TLB via these registers in that:

- The value returned in the VPN2 field of the *EntryHi* register may have those bits set to zero corresponding to the one bits in the Mask field of the TLB entry (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed after a TLB entry is written and then read.
- The value returned in the PFN field of the *EntryLo0* and *EntryLo1* registers may have those bits set to zero corresponding to the one bits in the Mask field of the TLB entry (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed after a TLB entry is written and then read.
- The value returned in the G bit in both the *EntryLo0* and *EntryLo1* registers comes from the single G bit in the TLB entry. Recall that this bit was set from the logical AND of the two G bits in *EntryLo0* and *EntryLo1* when the TLB was written.

Restrictions:

This instruction is legal only if the processor is in Kernel Mode or Debug Mode, or if the CP0 usable bit is set in the Status register. In other circumstances, execution of this instruction results in a Coprocessor Unusable Exception.

The operation is **UNDEFINED** if the contents of the *Index* register are greater than or equal to the number of TLB entries in the processor.

For processors that do not include the standard TLB MMU, the operation of this instruction is **UNDEFINED**. However, the preferred implementation is a Reserved Instruction Exception.

Operation:

```
i ← Index
if ($Status_{CU0}$ = 1) or ($Status_{KSU}$= $00_2$) or ($Debug_{DM}$ = 1) or ($Status_{EXL}$ = 1) or ($Status_{ERL}$ = 1) then
    if i > TLBEntries -1 then
        UNDEFINED
    endif
    $PageMask_{Mask}$ ← $TLB[i]_{Mask}$
    EntryHi ← $TLB[i]_R$ || $0^{Fill}$ ||
                  ($TLB[i]_{VPN2}$ and not $TLB[i]_{Mask}$) ||      # Masking of VPN2 is implementation dependent
                  $0^5$ || $TLB[i]_{ASID}$
    EntryLo1 ← $0^{Fill}$ || ($TLB[i]_{PFN1}$ and not $TLB[i]_{Mask}$) ||   # Masking of PFN is implementation dependent
                  $TLB[i]_{C1}$ || $TLB[i]_{D1}$ || $TLB[i]_{V1}$ || $TLB[i]_G$
    EntryLo0 ← $0^{Fill}$ || ($TLB[i]_{PFN0}$ and not $TLB[i]_{Mask}$) ||   # Masking of PFN is implementation dependent
                  $TLB[i]_{C0}$ || $TLB[i]_{D0}$ || $TLB[i]_{V0}$ || $TLB[i]_G$
else
    InitiateCoprocessorUnusableException(0)
endif
```

Exceptions:

Coprocessor Unusable Exception

Reserved Instruction Exception (if not implemented)

Machine Check (if implemented and a TLB shutdown condition is detected on a TLB read)

2.9.3.5 The TLBWI Instruction

Write Indexed TLB Entry **TLBWI**

31 26	25	24	0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBWI 000010
6	1	19	6

Format:

TLBWI **MIPS64**

Purpose:

Write a TLB entry indexed by the *Index* register.

Description:

The TLB entry pointed to by the *Index* register is written from the contents of the *EntryHi*, *EntryLo0*, *EntryLo1,* and *PageMask* registers. The information written to the TLB entry may be different from that in the *EntryHi*, *EntryLo0*, and *EntryLo1* registers, in that:

- The value written to the VPN2 field of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of the *PageMask* register (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The value written to the PFN0 and PFN1 fields of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of *PageMask* register (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The single G bit in the TLB entry is set from the logical AND of the G bits in the *EntryLo0* and *EntryLo1* registers.

Restrictions:

This instruction is legal only if the processor is in Kernel Mode or Debug Mode, or if the CP0 usable bit is set in the Status register. In other circumstances, execution of this instruction results in a Coprocessor Unusable Exception.

The operation is **UNDEFINED** if the contents of the *Index* register are greater than or equal to the number of TLB entries in the processor.

For processors that do not include the standard TLB MMU, the operation of this instruction is **UNDEFINED**. However, the preferred implementation is a Reserved Instruction Exception.

Operation:

```
i ← Index
if ($Status_{CU0}$ = 1) or ($Status_{KSU}$ = $00_2$) or ($Debug_{DM}$ = 1) or ($Status_{EXL}$ = 1) or ($Status_{ERL}$ = 1) then
    if i > TLBEntries -1 then
        UNDEFINED
    endif
    $TLB[i]_{Mask}$ ← $PageMask_{Mask}$
    $TLB[i]_R$ ← $EntryHi_R$
    $TLB[i]_{VPN2}$ ← $EntryHi_{VPN2}$ and not $PageMask_{Mask}$      # Masking of VPN2 is implementation dependent
    $TLB[i]_{ASID}$ ← $EntryHi_{ASID}$
    $TLB[i]_G$ ← $EntryLo1_G$ and $EntryLo0_G$
    $TLB[i]_{PFN1}$ ← $EntryLo1_{PFN}$ and not $PageMask_{Mask}$      # Masking of PFN is implementation dependent
    $TLB[i]_{C1}$ ← $EntryLo1_C$
    $TLB[i]_{D1}$ ← $EntryLo1_D$
    $TLB[i]_{V1}$ ← $EntryLo1_V$
    $TLB[i]_{PFN0}$ ← $EntryLo0_{PFN}$ and not $PageMask_{Mask}$      # Masking of PFN is implementation dependent
    $TLB[i]_{C0}$ ← $EntryLo0_C$
    $TLB[i]_{D0}$ ← $EntryLo0_D$
    $TLB[i]_{V0}$ ← $EntryLo0_V$
else
    InitiateCoprocessorUnusableException(0)
endif
```

Exceptions:

Coprocessor Unusable Exception
Reserved Instruction Exception (if not implemented)
Machine Check (if implemented and a TLB shutdown condition is detected on a TLB write)

2.9.3.6 The TLBWR Instruction

Write Random TLB Entry **TLBWR**

31 26	25	24	0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBWR 000110
6	1	19	6

Format:

TLBWR **MIPS64**

Purpose:

Write a TLB entry indexed by the *Random* register.

Description:

The TLB entry pointed to by the *Random* register is written from the contents of the *EntryHi*, *EntryLo0*, *EntryLo1,* and *PageMask* registers. The information written to the TLB entry may be different from that in the *EntryHi*, *EntryLo0*, and *EntryLo1* registers, in that:

- The value written to the VPN2 field of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of the *PageMask* register (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The value written to the PFN0 and PFN1 fields of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of *PageMask* register (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.
- The single G bit in the TLB entry is set from the logical AND of the G bits in the *EntryLo0* and *EntryLo1* registers.

Restrictions:

This instruction is legal only if the processor is in Kernel Mode or Debug Mode, or if the CP0 usable bit is set in the Status register. In other circumstances, execution of this instruction results in a Coprocessor Unusable Exception.

For processors that do not include the standard TLB MMU, the operation of this instruction is **UNDEFINED**. However, the preferred implementation is a Reserved Instruction Exception.

Operation:

```
i ← Random
if ($Status_{CU0}$ = 1) or ($Status_{KSU}$= $00_2$) or ($Debug_{DM}$ = 1) or ($Status_{EXL}$ = 1) or ($Status_{ERL}$ = 1) then
    $TLB[i]_{Mask}$ ← $PageMask_{Mask}$
    $TLB[i]_R$ ← $EntryHi_R$
    $TLB[i]_{VPN2}$ ← $EntryHi_{VPN2}$ and not $PageMask_{Mask}$     # Masking of VPN2 is implementation dependent
    $TLB[i]_{ASID}$ ← $EntryHi_{ASID}$
    $TLB[i]_G$ ← $EntryLo1_G$ and $EntryLo0_G$
    $TLB[i]_{PFN1}$ ← $EntryLo1_{PFN}$ and not $PageMask_{Mask}$     # Masking of PFN is implementation dependent
    $TLB[i]_{C1}$ ← $EntryLo1_C$
    $TLB[i]_{D1}$ ← $EntryLo1_D$
    $TLB[i]_{V1}$ ← $EntryLo1_V$
    $TLB[i]_{PFN0}$ ← $EntryLo0_{PFN}$ and not $PageMask_{Mask}$     # Masking of PFN is implementation dependent
    $TLB[i]_{C0}$ ← $EntryLo0_C$
    $TLB[i]_{D0}$ ← $EntryLo0_D$
    $TLB[i]_{V0}$ ← $EntryLo0_V$
else
    InitiateCoprocessorUnusableException(0)
endif
```

Exceptions:

Coprocessor Unusable Exception
Reserved Instruction Exception (if not implemented)
Machine Check (if implemented and a TLB shutdown condition is detected on a TLB write)

2.9.3.7 The WAIT Instruction

Enter Standby Mode **WAIT**

31 26	25	24	0
COP0 010000	CO 1	Implementation-Dependent Information	WAIT 100000
6	1	19	6

Format:

WAIT **MIPS64**

Purpose:

Wait for Event

Description:

The WAIT instruction performs an implementation-dependent operation, usually involving a lower power mode. Software may use bits 24..6 of the instruction to communicate additional information to the processor, and the processor may use this information as control for the lower power mode. A value of zero for bits 24..6 is the default, and must be valid in all implementations.

The WAIT instruction is typically implemented by stalling the pipeline at the completion of the instruction and entering a lower power mode. The pipeline is restarted when an external event, such as an interrupt or external request occurs, and execution continues with the instruction following the WAIT instruction. It is implementation-dependent whether the pipeline restarts when a non-enabled interrupt is requested. In this case, software must poll for the cause of the restart. If the pipeline restarts as the result of an enabled interrupt, that interrupt is taken between the WAIT instruction and the following instruction (EPC for the interrupt points at the instruction following the WAIT instruction).

The assertion of any reset or NMI signal (if not masked by EJTAG) must restart the pipeline and the corresponding exception must be taken.

Restrictions:

The operation of the processor is **UNDEFINED** if a wait instruction is placed in the delay slot of a branch or a jump.

This instruction is legal only if the processor is in Kernel Mode or Debug Mode, or if the CP0 usable bit is set in the Status register. In other circumstances, execution of this instruction results in a Coprocessor Unusable Exception.

Operation:

```
if ($Status_{CU0}$ = 1) or ($Status_{KSU}$= $00_2$) or ($Debug_{DM}$ = 1) or ($Status_{EXL}$ = 1) or ($Status_{ERL}$ = 1) then
    Enter implementation dependent lower power mode
else
    InitiateCoprocessorUnusableException(0)
endif
```

Exceptions:

Coprocessor Unusable Exception

3. Floating Point Control Registers

Although all five floating point control registers are included in the MIPS RISC Architecture documentation for the MIPS V ISA, several changes are included in MIPS64. As such, the registers are described below. Refer to the MIPS RISC Architecture documentation for a full description of the MIPS Floating Point Architecture.

3.0.1 Floating Point Implementation Register (CP1 Register 0)

The Floating Point Implementation Register (*FIR*) is a 32-bit read-only register that contains information identifying the capabilities of the floating point unit, the floating point processor identification, and the revision level of the floating point unit. Figure 5 shows the format of the *FIR* register; Table 40 describes the *FIR* register fields.

Figure 5: FIR Register Format

31 20	19	18	17	16	15 8	7 0
0	3D	PS	D	S	ProcessorID	Revision

Table 40: FIR Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
0	31:20	Reserved for future use; reads as zero	0	0	Reserved
3D	19	Indicates that the MIPS-3D ASE is implemented: 0: MIPS-3D not implemented 1: MIPS-3D implemented	R	Preset	Required
PS	18	Indicates that the paired single (PS) floating point data type and instructions are implemented: 0: PS floating not implemented 1: PS floating implemented	R	Preset	Required
D	17	Indicates that the double-precision (D) floating point data type and instructions are implemented: 0: D floating not implemented 1: D floating implemented	R	Preset	Required
S	16	Indicates that the single-precision (S) floating point data type and instructions are implemented: 0: S floating not implemented 1: S floating implemented	R	Preset	Required
ProcessorID	15:8	Identifies the floating point processor. This value should normally match the corresponding field of the *PRId* CP0 register unless there are different floating point implementations used by a single CPU.	R	Preset	Required

Table 40: FIR Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
Revision	7:0	Specifies the revision number of the floating point unit. This field allows software to distinguish between one revision and another of the same floating point processor type. If this field is not implemented, it must read as zero.	R	Preset	Optional

3.0.2 Floating Point Control and Status Register (CP1 Register 31)

The Floating Point Control and Status Register (*FCSR*) is a 32-bit register that controls the operation of the floating point unit. Access to *FCSR* is not privileged; it can be read or written by any program that has access to the floating point unit (via the coprocessor enables in the *Status* register). Figure 6 shows the format of the *FCSR* register; Table 41 describes the *FCSR* register fields.

Figure 6: FCSR Register Format

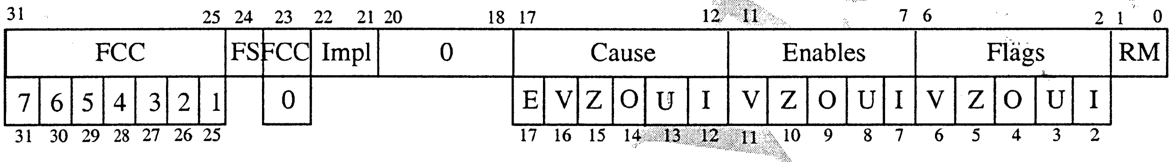


Table 41: FCSR Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
FCC	31:25, 23	Floating point condition codes. These bits record the result of floating point compares and are tested for floating point conditional branches and conditional moves. The FCC bit to use is specified in the compare, branch, or conditional move instruction. For backward compatibility with previous MIPS ISAs, the FCC bits are separated into two, non-contiguous fields.	R/W	Undefined	Required
FS	24	Flush to Zero. When FS is one, denormalized results are flushed to zero instead of causing an Unimplemented Operation exception. It is implementation dependent whether denormalized operand values are flushed to zero before the operation is carried out.	R/W	Undefined	Required
Impl	22:21	Available to control implementation dependent features of the floating point unit. If these bits are not implemented, they must be ignored on write and read as zero.	R/W	Undefined	Optional

Table 41: FCSR Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
0	20:18	Reserved for future use; Must be written as zero; returns zero on read.	0	0	Reserved
Cause	17:12	Cause bits. These bits indicate the exception conditions that arise during execution of an FPU arithmetic instruction. A bit is set to 1 if the corresponding exception condition arises during the execution of an instruction and is set to 0 otherwise. By reading the registers, the exception condition caused by the preceding FPU arithmetic instruction can be determined. Refer to Table 42 for the meaning of each bit.	R/W	Undefined	Required
Enables	11:7	Enable bits. These bits control whether or not a trap is taken when an IEEE exception condition occurs for any of the five conditions. The trap occurs when both an Enable bit and the corresponding Cause bit are set either during an FPU arithmetic operation or by moving a value to FCSR or one of its alternative representations. Note that Cause bit E has no corresponding Enable bit; the non-IEEE Unimplemented Operation exception is defined by MIPS as always enabled. Refer to Table 42 for the meaning of each bit.	R/W	Undefined	Required
Flags	6:2	Flag bits. This field shows any exception conditions that have occurred for completed instructions since the flag was last reset by software. When a FPU arithmetic operation raises an IEEE exception condition that does not result in a Floating Point Exception (i.e., the Enable bit was off), the corresponding bit(s) in the Flag field are set, while the others remain unchanged. Arithmetic operations that result in a Floating Point Exception (i.e., the Enable bit was on) do not update the Flag bits. This field is never reset by hardware and must be explicitly reset by software. Refer to Table 42 for the meaning of each bit.	R/W	Undefined	Required

Table 41: FCSR Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
RM	1:0	Rounding mode. This field indicates the rounding mode used for most floating point operations (some operations use a specific rounding mode). Refer to Table 43 for the meaning of the encodings of this field.	R/W	Undefined	Required.

Table 42: Cause, Enable, and Flag Bit Definitions

Bit Name	Bit Meaning
E	Unimplemented Operation (this bit exists only in the Cause field)
V	Invalid Operations
Z	Divide by Zero
O	Overflow
U	Underflow
I	Inexact

Table 43: Rounding Mode Definitions

RM Field Encoding	Meaning
0	RN - Round to Nearest Rounds the result to the nearest representable value. When two representable values are equally near, the result is rounded to the value whose least significant bit is zero (that is, even)
1	RZ - Round Toward Zero Rounds the result to the value closest to but not greater than in magnitude than the result.
2	RP - Round Towards Plus Infinity Rounds the result to the value closest to but not less than the result.

Table 43: Rounding Mode Definitions

RM Field Encoding	Meaning
3	RM - Round Towards Minus Infinity Rounds the result to the value closest to but not greater than the result.

3.0.3 Floating Point Condition Codes Register (CP1 Register 25)

The Floating Point Condition Codes Register (*FCCR*) is an alternative way to read and write the floating point condition code values that also appear in *FCSR*. Unlike *FCSR*, all eight FCC bits are contiguous in *FCCR*. Figure 7 shows the format of the *FCCR* register; Table 44 describes the *FCCR* register fields.

Figure 7: FCCR Register Format

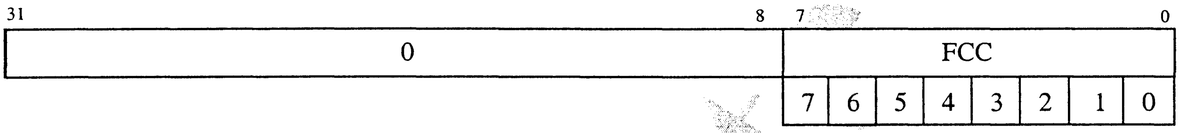


Table 44: FCCR Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
0	31:8	Must be written as zero; returns zero on read	0	0	Reserved
FCC	7:0	Floating point condition code. Refer to the description of this field in the *FCSR* register.	R/W	Undefined	Required

3.0.4 Floating Point Exceptions Register (CP1 Register 26)

The Floating Point Exceptions Register (*FEXR*) is an alternative way to read and write the Cause and Flags fields that also appear in *FCSR*. Figure 8 shows the format of the *FEXR* register; Table 45 describes the *FEXR* register fields.

Figure 8: FEXR Register Format

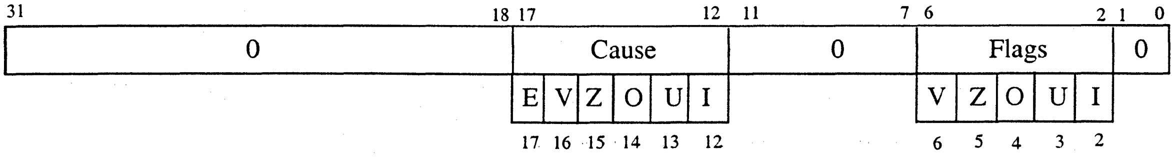


Table 45: FEXR Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
0	31:18, 11:7, 1:0	Must be written as zero; returns zero on read	0	0	Reserved
Cause	17:12	Cause bits. Refer to the description of this field in the *FCSR* register.	R/W	Undefined	Required
Flags	6:2	Flags bits. Refer to the description of this field in the *FCSR* register.	R/W	Undefined	Optional

3.0.5 Floating Point Enables Register (CP1 Register 28)

The Floating Point Enables Register (*FENR*) is an alternative way to read and write the Enables, FS, and RM fields that also appear in *FCSR*. Figure 9 shows the format of the *FENR* register; Table 46 describes the *FENR* register fields.

Figure 9: FENR Register Format

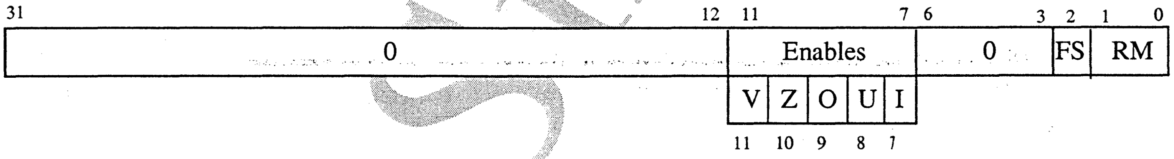


Table 46: FENR Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
0	31:12, 6:3	Must be written as zero; returns zero on read	0	0	Reserved
Enables	11:7	Enable bits. Refer to the description of this field in the *FCSR* register.	R/W	Undefined	Required
FS	2	Flush to Zero bit. Refer to the description of this field in the *FCSR* register.	R/W	Undefined	Required

Table 46: FENR Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
RM	1:0	Rounding mode. Refer to the description of this field in the *FCSR* register.	R/W	Undefined	Required

4. The MIPS64 Privileged Resource Architecture

4.1 Introduction

The MIPS64 Privileged Resource Architecture (PRA) is a set of environments and capabilities on which the Instruction Set Architecture operates. The effects of some components of the PRA are user-visible, for instance, the virtual memory layout. Many other components are visible only to the operating system kernel and to systems programmers. The PRA provides the mechanisms necessary to manage the resources of the CPU: virtual memory, caches, exceptions and user contexts. This chapter describes these mechanisms.

4.2 Compliance

Features described as *Required* in this document are required of all processors claiming compatibility with the MIPS64 Architecture. Features described as *Recommended* should be implemented unless there is an overriding need not to do so. Features described as *Optional* provide a standardization of features that may or may not be appropriate for a particular MIPS processor implementation. If such a feature is implemented, it must be implemented as described in this document if a processor claims compatibility with the MIPS64 Architecture.

In some cases, there are features within features that have different levels of compliance. For example, if there is an *Optional* field within a *Required* register, this means that the register must be implemented, but the field may or may not be, depending on the needs of the implementation. Similarly, if there is a *Required* field within an *Optional* register, this means that if the register is implemented, it must have the specified field.

4.3 The MIPS Coprocessor Model

The MIPS ISA provides for up to 4 coprocessors. A coprocessor extends the functionality of the MIPS ISA, while sharing the instruction fetch and execution control logic of the CPU. Some coprocessors, such as the system coprocessor and the floating point unit are standard parts of the ISA, and are specified as such in the architecture documents. Coprocessors are generally optional, with one exception: CP0, the system coprocessor, is required. CP0 is the ISA interface to the Privileged Resource Architecture and provides full control of the processor state and modes.

4.3.1 CP0 - The System Coprocessor

CP0 provides an abstraction of the functions necessary to support an operating system: exception handling, memory management, scheduling, and control of critical resources. The interface to CP0 is through various instructions encoded with the *COP0* opcode, including the ability to move data to and from the CP0 registers, and specific functions that modify CP0 state. The CP0 registers and the interaction with them make up much of the Privileged Resource Architecture.

4.3.2 CP0 Register Summary

Table 47 lists the CP0 registers in numerical order. The individual registers are described later in this document. If the compliance level is qualified (e.g., "*Required* (TLB MMU)"), it applies only if the qualifying condition is true. The

Sel column indicates the value to be used in the field of the same name in the MFC0 and MTC0 instructions.

Table 47: Coprocessor 0 Registers in Numerical Order

Register Number	Sel	Register Name	Function	Reference	Compliance Level
0	0	Index	Index into the TLB array	Section 4.9.1 on page 105	*Required (TLB MMU); Optional* (others)
1	0	Random	Randomly generated index into the TLB array	Section 4.9.2 on page 106	*Required* (TLB MMU); *Optional* (others)
2	0	EntryLo0	Low-order portion of the TLB entry for even-numbered virtual pages	Section 4.9.3 on page 107	*Required* (TLB MMU); *Optional* (others)
3	0	EntryLo1	Low-order portion of the TLB entry for odd-numbered virtual pages	Section 4.9.3 on page 107	*Required* (TLB MMU); *Optional* (others)
4	0	Context	Pointer to page table entry in memory	Section 4.9.4 on page 110	*Required* (TLB MMU); *Optional* (others)
5	0	PageMask	Control for variable page size in TLB entries	Section 4.9.5 on page 111	*Required* (TLB MMU); *Optional* (others)
6	0	Wired	Controls the number of fixed ("wired") TLB entries	Section 4.9.6 on page 112	*Required* (TLB MMU); *Optional* (others)
7	all		Reserved for future extensions		*Reserved*
8	0	BadVAddr	Reports the address for the most recent address-related exception	Section 4.9.7 on page 113	*Required*
9	0	Count	Processor cycle count	Section 4.9.8 on page 114	*Required*
10	0	EntryHi	High-order portion of the TLB entry	Section 4.9.9 on page 114	*Required* (TLB MMU); *Optional* (others)
11	0	Compare	Timer interrupt control	Section 4.9.10 on page 116	*Required*

Table 47: Coprocessor 0 Registers in Numerical Order

Register Number	Sel	Register Name	Function	Reference	Compliance Level
12	0	Status	Processor status and control	Section 4.9.11 on page 116	*Required*
13	0	Cause	Cause of last general exception	Section 4.9.12 on page 123	*Required*
14	0	EPC	Program counter at last exception	Section 4.9.13 on page 126	*Required*
15	0	PRId	Processor identification and revision	Section 4.9.14 on page 127	*Required*
16	0	Config	Configuration register	Section 4.9.15 on page 128	*Required*
16	1	Config1	Configuration register 1	Section 4.9.16 on page 130	*Required*
17	0	LLAddr	Load linked address	Section 4.9.17 on page 132	*Optional*
18	0-n	WatchLo	Watchpoint address	Section 4.9.18 on page 132	*Optional*
19	0-n	WatchHi	Watchpoint control	Section 4.9.19 on page 134	*Optional*
20	0	XContext	Extended Addressing Page Table Context	Section 4.9.20 on page 135	*Required* (64-bit TLB MMU) *Optional* (Others)
21	all		Reserved for future extensions		*Reserved*
22	all		Available for implementation dependent use	Section 4.9.21 on page 136	*Implementation-Dependent*
23	0	Debug	EJTAG Debug register	EJTAG Specification	*Optional*
24	0	DEPC	Program counter at last EJTAG debug exception	EJTAG Specification	*Optional*
25	0-n	PerfCnt	Performance counter interface	Section 4.9.24 on page 137	*Recommended*
26	0	ErrCtl	Parity/ECC error control and status	Section 4.9.25 on page 140	*Optional*
27	0-3	CacheErr	Cache parity error control and status	Section 4.9.26 on page 140	*Optional*

Table 47: Coprocessor 0 Registers in Numerical Order

Register Number	Sel	Register Name	Function	Reference	Compliance Level
28	0	TagLo	Low-order portion of cache tag interface	Section 4.9.27 on page 142	*Required (Cache)*
28	1	DataLo	Low-order portion of cache data interface	4.9.28 on page 143	*Optional*
29	0	TagHi	High-order portion of cache tag interface	Section 4.9.29 on page 143	*Required (Cache)*
29	1	DataHi	High-order portion of cache data interface	4.9.30 on page 144	*Optional*
30	0	ErrorEPC	Program counter at last error	Section 4.9.31 on page 144	*Required*
31	0	DESAVE	EJTAG debug exception save register	EJTAG Specification	*Optional*

4.4 Operating Modes

The MIPS64 PRA requires two operating mode: User Mode and Kernel Mode. When operating in User Mode, the programmer has access to the CPU and FPU registers that are provided by the ISA and to a flat, uniform virtual memory address space. When operating in Kernel Mode, the systems programmer has access to the full capabilities of the processor, including the ability to change virtual memory mapping, control the system environment, and context switch between processes.

In addition, the MIPS64 PRA supports the implementation of two additional modes: Supervisor Mode and EJTAG Debug Mode. Refer to the EJTAG specification for a description of Debug Mode.

Finally, the MIPS64 PRA provides backward compatible support for 32-bit programs by providing enables for both 64-bit addressing and 64-bit operations. If access is not enabled, an attempt to reference a 64-bit address or an instruction that implements a 64-bit operation results in an exception.

4.4.1 Debug Mode

For processors that implement EJTAG, the processor is operating in Debug Mode if the DM bit in the *Debug* register is a one. If the processor is running in Debug Mode, it has full access to all resources that are available to Kernel Mode operation.

4.4.2 Kernel Mode

The processor is operating in Kernel Mode when the DM bit in the *Debug* register is a zero (if the processor implements Debug Mode), and any of the following three conditions is true:

- The KSU field in the *Status* register contains 00_2
- The EXL bit in the *Status* register is one
- The ERL bit in the *Status* register is one

The processor enters Kernel Mode at power-up, or as the result of an interrupt, exception, or error. The processor leaves Kernel Mode and enters User Mode or Supervisor Mode when all of the previous three conditions are false, usually as the result of an ERET instruction.

4.4.3 Supervisor Mode

The processor is operating in Supervisor Mode (if that optional mode is implemented by the processor) when all of the following conditions are true:

- The DM bit in the *Debug* register is a zero (if the processor implements Debug Mode)
- The KSU field in the *Status* register contains 01_2
- The EXL and ERL bits in the *Status* register are both zero

4.4.4 User Mode

The processor is operating in User Mode when all of the following conditions are true:

- The DM bit in the *Debug* register is a zero (if the processor implements Debug Mode)
- The KSU field in the *Status* register contains 10_2
- The EXL and ERL bits in the *Status* register are both zero

4.5 Other Modes

4.5.1 64-bit Address Enable

Access to 64-bit addresses are enabled under any of the following conditions:

- A legal reference to a kernel address space occurs and the KX bit in the *Status* register is a one
- A legal reference to a supervisor address space occurs and the SX bit in the *Status* register is a one
- A legal reference to a user address space occurs and the UX bit in the *Status* register is a one

Note that the operating mode of the processor is not relevant to 64-bit address enables. That is, a reference to user address space made while the processor is operating in Kernel Mode is controlled by the state of the UX bit, not by the KX bit.

An attempt to reference a 64-bit address space when 64-bit addresses are not enabled results in an Address Error Exception (either AdEL or AdES, depending on the type of reference).

When a TLB miss occurs, the choice of the Exception Vector is also determined by the 64-bit address enable. If 64-bit addresses are not enabled for the reference, the TLB Refill Vector is used. If 64-bit addresses are enabled for the reference, the XTLB Refill Vector is used.

4.5.2 64-bit Operations Enable

Instructions that perform 64-bit operations are legal under any of the following conditions:

- The processor is operating in Kernel Model, Supervisor Mode, or Debug Mode, as described above.
- The PX bit in the *Status* register is a one
- The processor is operating in User Mode, as described above, and the UX bit in the *Status* register is a one.

An attempt to execute an instruction which performs 64-bit operations when such instructions are not enabled results in a Reserved Instruction Exception.

4.5.3 64-bit FPR Enable

Access to 64-bit FPRs is controlled by the FR bit in the *Status* register. If the FR bit is one, the FPRs are interpreted as 32 64-bit registers that may contain any data type. If the FR bit is zero, the FPRs are interpreted as 32 32-bit registers, any of which may contain a 32-bit data type (W, S). In this case, 64-bit data types are contained in even-odd pairs of registers.

The operation of the processor is **UNPREDICTABLE** under any of the following conditions:

- The FR bit is a zero and an odd register is referenced by an instruction whose datatype is 64-bits

- The FR bit is a zero and a floating point instruction is executed whose data type is L or PS
- 64-bit operations are not enabled, the FR bit is a one, and an instruction references the floating point registers.

4.6 Virtual Memory

4.6.1 Terminology

4.6.1.1 Address Space

An *Address Space* is the range of all possible addresses that can be generated for a particular addressing mode. There is one 64-bit Address Space and one 32-bit Compatibility Address Space that is mapped into a subset of the 64-bit Address Space.

4.6.1.2 Segment and Segment Size (SEGBITS)

A *Segment* is a defined subset of an Address Space that has self-consistent reference and access behavior. A 32-bit Compatibility Segment is part of the 32-bit Compatibility Address Space and is either 2^{29} or 2^{31} bytes in size, depending on the specific Segment. A 64-bit Segment is part of the 64-bit Address Space and is no larger than 2^{62} bytes in size, but may be smaller on an implementation dependent basis. The symbol *SEGBITS* is used to represent the actual number of bits implemented in each 64-bit Segment. As such, if 40 virtual address bits were implemented, the actual size of the Segment would be $2^{SEGBITS} = 2^{40}$ bytes.

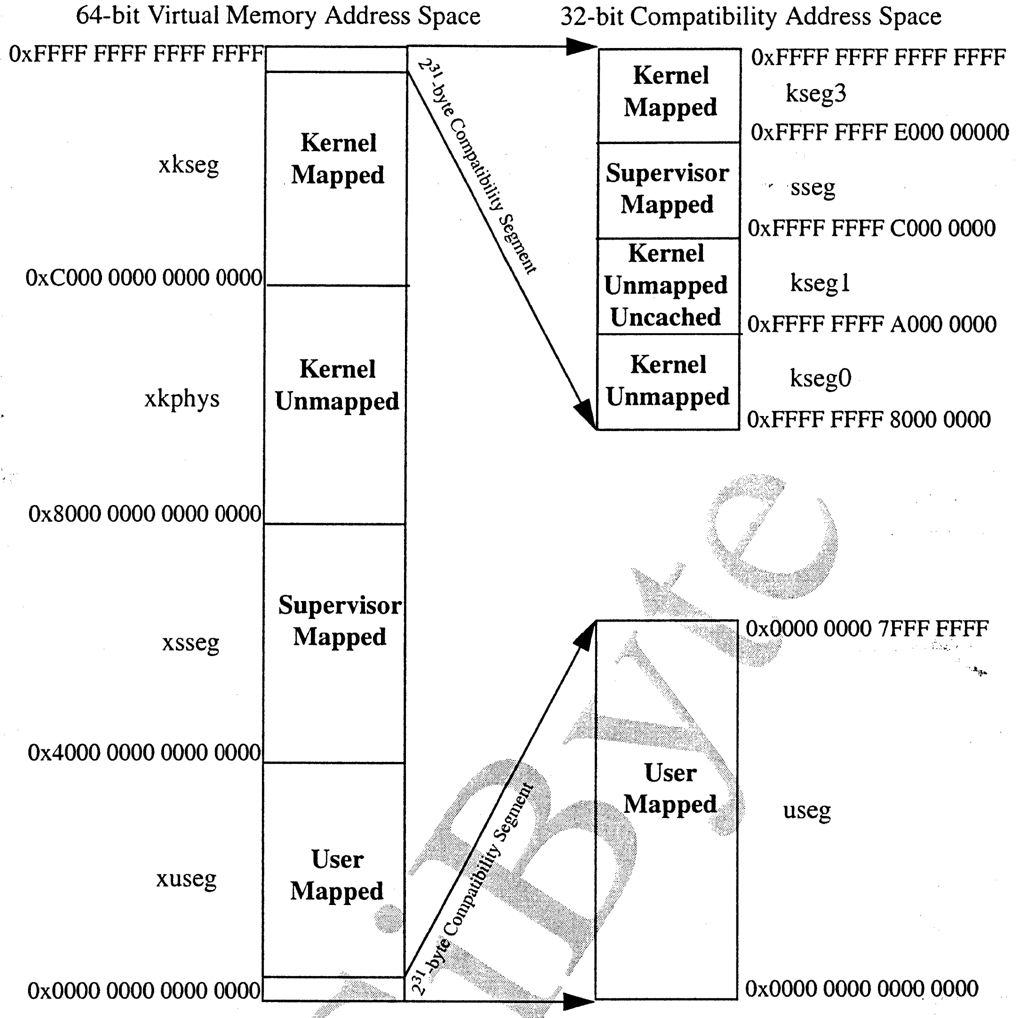
4.6.1.3 Physical Address Size (PABITS)

The number of physical address bits implemented is represented by the symbol *PABITS*. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{PABITS} = 2^{36}$ bytes.

4.6.2 Virtual Address Spaces

With support for 64-bit operations and address calculation, the MIPS64 architecture implicitly defines and provides support for a 64-bit virtual Address Space, sub-divided into four Segments selected by bits 63:62 of the virtual address. To provide compatibility for 32-bit programs and MIPS32 processors, a 2^{32}-byte Compatibility Address Space is defined, separated into two non-contiguous ranges in which the upper 32 bits of the 64-bit address are the sign extension of bit 31. The Compatibility Address Space is similarly sub-divided into Segments selected by bits 31:29 of the virtual address. Figure 10 shows the layout of the Address Spaces, including the Compatibility Address Space and the segmentation of each Address Space.

Figure 10: Virtual Address Spaces



Each Segment of an Address Space is classified as "Mapped" or "Unmapped". A "Mapped" address is one that is translated through the TLB or other memory management translation unit. An "Unmapped" address is one which is not translated through the TLB and which provides a window into the lowest portion of the physical address space, starting at physical address zero, and with a size corresponding to the size of the unmapped Segment.

Additionally, the kseg1 Segment is classified as "Uncached". References to this Segment bypass all levels of the cache hierarchy and allow direct access to memory without any interference from the caches.

Table 48 lists the same information in tabular form.

Table 48: Virtual Memory Address Spaces

$VA_{63..62}$	Segment Name(s)	Maximum Address Range	64-bit Address Enable	Associated with Mode	Reference Legal from Mode(s)	Actual Segment Size	Segment Type
11_2	kseg3	0xFFFF FFFF FFFF FFFF through 0xFFFF FFFF E000 0000	Always	Kernel	Kernel	2^{29} bytes	32-bit Compatibility
	sseg ksseg	0xFFFF FFFF DFFF FFFF through 0xFFFF FFFF C000 0000	Always	Supervisor	Supervisor Kernel	2^{29} bytes	32-bit Compatibility
	kseg1	0xFFFF FFFF BFFF FFFF through 0xFFFF FFFF A000 0000	Always	Kernel	Kernel	2^{29} bytes	32-bit Compatibility
	kseg0	0xFFFF FFFF 9FFF FFFF through 0xFFFF FFFF 8000 0000	Always	Kernel	Kernel	2^{29} bytes	32-bit Compatibility
	xkseg	0xFFFF FFFF 7FFF FFFF through 0xC000 0000 0000 0000	KX	Kernel	Kernel	$(2^{SEGBITS} - 2^{31})$ bytes[a]	64-bit
10_2	xkphys	0xBFFF FFFF FFFF FFFF through 0x8000 0000 0000 0000	KX	Kernel	Kernel	8 2^{PABITS} byte[a] regions within the 2^{62} byte Segment	64-bit
01_2	xsseg xksseg	0x7FFF FFFF FFFF FFFF through 0x4000 0000 0000 0000	SX	Supervisor	Supervisor Kernel	$2^{SEGBITS}$ bytes[a]	64-bit
00_2	xuseg xsuseg xkuseg	0x3FFF FFFF FFFF FFFF through 0x0000 0000 8000 0000	UX	User	User Supervisor Kernel	$(2^{SEGBITS} - 2^{31})$ bytes[a]	64-bit
	useg suseg kuseg	0x0000 0000 7FFF FFFF through 0x0000 0000 0000 0000	Always	User	User Supervisor Kernel	2^{31} bytes	32-bit Compatibility

[a] See Section 4.6.1.2 on page 73 and Section 4.6.1.3 on page 73 for an explanation of the symbols *SEGBITS* and *PABITS*, respectively

Each Segment of an Address Space is associated with one of the three processor operating modes (User, Supervisor, or Kernel). A Segment that is associated with a particular mode is accessible if the processor is running in that or a more-privileged mode. For example, a Segment associated with User Mode is accessible when the processor is running in User, Supervisor, or Kernel Modes. A Segment is not accessible if the processor is running in a less privileged mode than that associated with the Segment. For example, a Segment associated with Supervisor Mode is not accessible when the processor is running in User Mode and such a reference results in an Address Error Exception. The "Reference Legal from Mode(s)" column in Table 48 lists the modes from which each Segment may be legally referenced.

If a Segment has more than one name, each name denotes the mode from which the Segment is referenced. For example, the Segment name "useg" denotes a reference from user mode, while the Segment name "kuseg" denotes a reference to the same Segment from kernel mode.

References to 64-bit Segments (as shown in the "Segment Type" column of Table 48) are enabled only if the appropriate 64-bit Address Enable is on (see 4.5.1 on page 72, and the "64-bit Enable" column of Table 48). References to 32-bit Compatibility Segments are always enabled.

4.6.3 Compliance

A MIPS64 compliant processor must implement the following 32-bit Compatibility Segments:

- useg/kuseg
- kseg0
- kseg1

In addition, a MIPS64 compliant processor using the TLB-based address translation mechanism must also implement the kseg3 32-bit Compatibility Segment. It is also strongly recommended that the sseg segment be implemented, whether Supervisor Mode is implemented or not.

It is implementation dependent whether a MIPS64 compliant processor implements 64-bit addressing and the 64-bit Segments associated with the 64-bit Address Space. If 64-bit addressing is implemented, it must be implemented as described here.

It is implementation dependent whether a MIPS64 compliant processor implements Supervisor Mode and the Segments associated with that mode. If Supervisor Mode is implemented, it must be implemented as described here. If Supervisor Mode is not implemented a processor may implement the sseg and xsseg segments, or treat references to them as address error exceptions. If the xsseg segment is implemented, access to it is controlled by the SX bit in the Status register, just as it would be if Supervisor Mode was implemented.

A MIPS64 compliant processor may implement fewer than 64 bits in the virtual address by restricting all 64-bit Segments to be less than 2^{62} bytes in size. A MIPS64 compliant processor that implements 64-bit virtual addressing must implement a value of *SEGBITS* that is no smaller than 40 bits. An attempt to reference an unimplemented region of a segment (that between $2^{SEGBITS}$ and 2^{62}-1) results in an Address Error Exception.

4.6.4 Access Control as a Function of Address and Operating Mode

Table 49 enumerates the action taken by the processor for each section of the 64-bit Address Space as a function of the operating mode of the processor. The selection of TLB Refill vector and other special-cased behavior is also listed for each reference.

Table 49: Address Space Access and TLB Refill Selection as a Function of Operating Mode

Virtual Address Range		Segment Name(s)	Action when Referenced from Operating Mode		
Symbolic	Assuming *SEGBITS* = 40, *PABITS* = 36		User Mode[a]	Supervisor Mode	Kernel Mode
0xFFFF FFFF FFFF FFFF through 0xFFFF FFFF E000 0000	0xFFFF FFFF FFFF FFFF through 0xFFFF FFFF E000 0000	kseg3	Address Error	Address Error	Mapped Refill Vector: TLB (KX=0) XTLB(KX=1) See 4.6.8 on page 83 for special behavior when $\text{Debug}_{DM} = 1$
0xFFFF FFFF DFFF FFFF through 0xFFFF FFFF C000 0000	0xFFFF FFFF DFFF FFFF through 0xFFFF FFFF C000 0000	sseg ksseg	Address Error	Mapped Refill Vector[b]: TLB (KX=0) XTLB(KX=1)	Mapped Refill Vector[b]: TLB (KX=0) XTLB(KX=1)
0xFFFF FFFF BFFF FFFF through 0xFFFF FFFF A000 0000	0xFFFF FFFF BFFF FFFF through 0xFFFF FFFF A000 0000	kseg1	Address Error	Address Error	Unmapped, Uncached See Section 4.6.5 on page 79
0xFFFF FFFF 9FFF FFFF through 0xFFFF FFFF 8000 0000	0xFFFF FFFF 9FFF FFFF through 0xFFFF FFFF 8000 0000	kseg0	Address Error	Address Error	Unmapped See Section 4.6.5 on page 79
0xFFFF FFFF 7FFF FFFF through 0xC000 0000 0000 0000 + $2^{SEGBITS} - 2^{31}$	0xFFFF FFFF 7FFF FFFF through 0xC000 00FF 8000 0000		Address Error	Address Error	Address Error

Table 49: Address Space Access and TLB Refill Selection as a Function of Operating Mode

Virtual Address Range			Action when Referenced from Operating Mode		
Symbolic	**Assuming *SEGBITS* = 40, *PABITS* = 36**	**Segment Name(s)**	**User Mode[a]**	**Supervisor Mode**	**Kernel Mode**
0xC000 0000 0000 0000 + $2^{SEGBITS} - 2^{31} - 1$ through 0xC000 0000 0000 0000	0xC000 00FF 7FFF FFFF through 0xC000 0000 0000 0000	xkseg	Address Error	Address Error	Address Error if KX = 0 Mapped if KX = 1 Refill Vector: XTLB
0xBFFF FFFF FFFF FFFF through 0x8000 0000 0000 0000	0xBFFF FFFF FFFF FFFF through 0x8000 0000 0000 0000	xkphys	Address Error	Address Error	Address Error if KX = 0 or in certain address ranges within the Segment Unmapped See Section 4.6.6 on page 80
0x7FFF FFFF FFFF FFFF through 0x4000 0000 0000 0000 + $2^{SEGBITS}$	0x7FFF FFFF FFFF FFFF through 0x4000 0100 0000 0000		Address Error	Address Error	Address Error
0x4000 0000 0000 0000 + $2^{SEGBITS} - 1$ through 0x4000 0000 0000 0000	0x4000 00FF FFFF FFFF through 0x4000 0000 0000 0000	xsseg xksseg	Address Error	Address Error if SX = 0 Mapped if SX = 1 Refill Vector: XTLB	Address Error if SX = 0 Mapped if SX = 1 Refill Vector: XTLB
0x3FFF FFFF FFFF FFFF through 0x0000 0000 0000 0000 + $2^{SEGBITS}$	0x3FFF FFFF FFFF FFFF through 0x0000 0100 0000 0000		Address Error	Address Error	Address Error

Table 49: Address Space Access and TLB Refill Selection as a Function of Operating Mode

Virtual Address Range			Action when Referenced from Operating Mode		
Symbolic	**Assuming *SEGBITS* = 40, *PABITS* = 36**	**Segment Name(s)**	**User Mode[a]**	**Supervisor Mode**	**Kernel Mode**
0x0000 0000 0000 0000 + $2^{SEGBITS}$ - 1 through 0x0000 0000 8000 0000	0x0000 00FF FFFF FFFF through 0x0000 0000 8000 0000	xuseg xsuseg xkuseg	Address Error if UX = 0 Mapped if UX = 1 Refill Vector: XTLB	Address Error if UX = 0 Mapped if UX = 1 Refill Vector: XTLB	Address Error if UX = 0 Mapped if UX = 1 Refill Vector: XTLB See Section 4.6.7 on page 83 for implementation dependent behavior when Status$_{ERL}$=1
0x0000 0000 7FFF FFFF through 0x0000 0000 0000 0000	0x0000 0000 7FFF FFFF through 0x0000 0000 0000 0000	useg suseg kuseg	Mapped Refill Vector: TLB (UX=0) XTLB(UX=1)	Mapped Refill Vector: TLB (UX=0) XTLB(UX=1)	Unmapped if Status$_{ERL}$=1 See Section 4.6.7 on page 83 Mapped if Status$_{ERL}$=0 Refill Vector: TLB (UX=0) XTLB(UX=1)

a. See Section 4.6.9 on page 84 for the special treatment of the address for data references when the processor is running in User Mode and the UX bit is zero.

b. Note that the Refill Vector for references to sseg/ksseg is determined by the state of the KX bit, not the SX bit. This simplifies the processor implementation by allowing them to treat the entire quadrant of the address space in which $VA_{63..62}$ are 11_2 in the same manner, as well as simplifying operating system software design which does not use Supervisor Mode.

4.6.5 Address Translation and Cache Coherency Attributes for the kseg0 and kseg1 Segments

The kseg0 and kseg1 Unmapped Segments provide a window into the least significant 2^{29} bytes of physical memory, and, as such, are not translated using the TLB or other address translation unit. The cache coherency attribute of the kseg0 Segment is supplied by the K0 field of the *Config* register. The cache coherency attribute for the kseg1 Segment is always Uncached. Table 50 describes how this transformation is done, and the source of the cache coherency attributes for each Segment.

Table 50: Address Translation and Cache Coherency Attributes for the kseg0 and ksegl Segments

Segment Name	Virtual Address Range	Generates Physical Address	Cache Attribute
ksegl	0xFFFF FFFF BFFF FFFF through 0xFFFF FFFF A000 0000	0x0000 0000 1FFF FFFF through 0x0000 0000 0000 0000	Uncached
kseg0	0xFFFF FFFF 9FFF FFFF through 0xFFFF FFFF 8000 0000	0x0000 0000 1FFF FFFF through 0x0000 0000 0000 0000	From K0 field of *Config* Register

4.6.6 Address Translation and Cache Coherency Attributes for the xkphys Segment

The xkphys Unmapped Segment is actually composed of 8 address ranges, each of which provides a window into the entire 2^{PABITS} bytes of physical memory and, as such, is not translated using the TLB or other address translation unit. For this Segment, the cache coherency attribute is taken from $VA_{61..59}$ and has the same encoding as that shown in Table 62. An Address Error Exception occurs if $VA_{58..PABITS}$ are non-zero. If no Address Error Exception occurs, the physical address is taken from $VA_{PABITS-1..0}$. Figure 11 shows the interpretation of the various fields of the virtual address when referencing the xkphys Segment.

Figure 11: Address Interpretation for the xkphys Segment

63 62	61 59	58 PABITS	PABITS - 1 0
10	CCA	Address Error if Non-Zero	Physical Address

Table 51: Address Translation and Cacheability Attributes for the xkphys Segment

Virtual Address Range		Generates Physical Address	Cache Attribute
Symbolic	Assuming *PABITS* = 36		
0xBFFF FFFF FFFF FFFF through 0xB800 0000 0000 0000 + 2^{PABITS}	0xBFFF FFFF FFFF FFFF through 0xB800 0010 0000 0000	Address Error	N/A

Table 51: Address Translation and Cacheability Attributes for the xkphys Segment

Virtual Address Range		Generates Physical Address	Cache Attribute
Symbolic	Assuming *PABITS* = 36		
0xB800 0000 0000 0000 + $2^{PABITS} - 1$ through 0xB800 0000 0000 0000	0xB800 000F FFFF FFFF through 0xB800 0000 0000 0000	0x0000 0000 0000 0000 + $2^{PABITS} - 1$ through 0x0000 0000 0000 0000	Uses encoding 7 of Table 62
0xB7FF FFFF FFFF FFFF through 0xB000 0000 0000 0000 + 2^{PABITS}	0xB7FF FFFF FFFF FFFF through 0xB000 0010 0000 0000	Address Error	N/A
0xB000 0000 0000 0000 + $2^{PABITS} - 1$ through 0xB000 0000 0000 0000	0xB000 000F FFFF FFFF through 0xB000 0000 0000 0000	0x0000 0000 0000 0000 + $2^{PABITS} - 1$ through 0x0000 0000 0000 0000	Uses encoding 6 of Table 62
0xAFFF FFFF FFFF FFFF through 0xA800 0000 0000 0000 + 2^{PABITS}	0xAFFF FFFF FFFF FFFF through 0xA800 0010 0000 0000	Address Error	N/A
0xA800 0000 0000 0000 + $2^{PABITS} - 1$ through 0xA800 0000 0000 0000	0xA800 000F FFFF FFFF through 0xA800 0000 0000 0000	0x0000 0000 0000 0000 + $2^{PABITS} - 1$ through 0x0000 0000 0000 0000	Uses encoding 5 of Table 62
0xA7FF FFFF FFFF FFFF through 0xA000 0000 0000 0000 + 2^{PABITS}	0xA7FF FFFF FFFF FFFF through 0xA000 0010 0000 0000	Address Error	N/A

Table 51: Address Translation and Cacheability Attributes for the xkphys Segment

Virtual Address Range			
Symbolic	**Assuming *PABITS* = 36**	**Generates Physical Address**	**Cache Attribute**
0xA000 0000 0000 0000 + 2^{PABITS} - 1 through 0xA000 0000 0000 0000	0xA000 000F FFFF FFFF through 0xA000 0000 0000 0000	0x0000 0000 0000 0000 + 2^{PABITS} - 1 through 0x0000 0000 0000 0000	Uses encoding 4 of Table 62
0x9FFF FFFF FFFF FFFF through 0x9800 0000 0000 0000 + 2^{PABITS}	0x9FFF FFFF FFFF FFFF through 0x9800 0010 0000 0000	Address Error	N/A
0x9800 0000 0000 0000 + 2^{PABITS} - 1 through 0x9800 0000 0000 0000	0x9800 000F FFFF FFFF through 0x9800 0000 0000 0000	0x0000 0000 0000 0000 + 2^{PABITS} - 1 through 0x0000 0000 0000 0000	Cacheable (see encoding 3 of Table 62)
0x97FF FFFF FFFF FFFF through 0x9000 0000 0000 0000 + 2^{PABITS}	0x97FF FFFF FFFF FFFF through 0x9000 0010 0000 0000	Address Error	N/A
0x9000 0000 0000 0000 + 2^{PABITS} - 1 through 0x9000 0000 0000 0000	0x9000 000F FFFF FFFF through 0x9000 0000 0000 0000	0x0000 0000 0000 0000 + 2^{PABITS} - 1 through 0x0000 0000 0000 0000	Uncached (see encoding 2 of Table 62)
0x8FFF FFFF FFFF FFFF through 0x8800 0000 0000 0000 + 2^{PABITS}	0x8FFF FFFF FFFF FFFF through 0x8800 0010 0000 0000	Address Error	N/A

Table 51: Address Translation and Cacheability Attributes for the xkphys Segment

Virtual Address Range		Generates Physical Address	Cache Attribute
Symbolic	Assuming *PABITS* = 36		
0x8800 0000 0000 0000 + 2^{PABITS} - 1 through 0x8800 0000 0000 0000	0x8800 000F FFFF FFFF through 0x8800 0000 0000 0000	0x0000 0000 0000 0000 + 2^{PABITS} - 1 through 0x0000 0000 0000 0000	Uses encoding 1 of Table 62
0x87FF FFFF FFFF FFFF through 0x8000 0000 0000 0000 + 2^{PABITS}	0x87FF FFFF FFFF FFFF through 0x8000 0010 0000 0000	Address Error	N/A
0x8000 0000 0000 0000 + 2^{PABITS} - 1 through 0x8000 0000 0000 0000	0x8000 000F FFFF FFFF through 0x8000 0000 0000 0000	0x0000 0000 0000 0000 + 2^{PABITS} - 1 through 0x0000 0000 0000 0000	Uses encoding 0 of Table 62

4.6.7 Address Translation for the kuseg Segment when $Status_{ERL}$ = 1

To provide support for the cache error handler, the kuseg Segment becomes an unmapped, uncached Segment, similar to the kseg1 Segment, if the ERL bit is set in the *Status* register. This allows the cache error exception code to operate uncached using GPR R0 as a base register to save other GPRs before use.

All processors must transform at least the lower 2^{29} bytes of kuseg. It is implementation dependent whether $VA_{31..29}$ participates in the transformation, allowing implementations the flexibility of using the same transformation on these bits as would be used to transform kseg0 or kseg1.

If 64-bit addressing is implemented and the UX bit is a one in the *Status* register, it is implementation dependent whether the range of addresses between 2^{31} and $2^{SEGBITS}$-1 are also treated as an unmapped, uncached Segment. That is, an implementation may choose to treat the entire xkuseg Segment in the same manner as the kuseg Segment.

4.6.8 Special Behavior for the kseg3 Segment when $Debug_{DM}$ = 1

If EJTAG is implemented on the processor, the EJTAG block may treat the virtual address range 0xFFFF FFFF FF20 0000 through 0xFFFF FFFF FF3F FFFF, inclusive, as a special memory-mapped region in Debug Mode. A MIPS64 compliant implementation that also implements EJTAG must:

- explicitly range check the address range as given and not assume that the entire region between 0xFFFF FFFF FF20 0000 and 0xFFFF FFFF FFFF FFFF is included in the special memory-mapped region.
- not enable the special EJTAG mapping for this region in any mode other than in EJTAG Debug mode.

Even in Debug mode, normal memory rules may apply in some cases. Refer to the EJTAG specification for details on this mapping.

4.6.9 Special Behavior for Data References in User Mode with $\text{Status}_{UX} = 0$

When the processor is running in User Mode, legal addresses have VA_{31} equal zero, and the 32-bit virtual address is sign-extended (really zero-extended because VA_{31} is zero) into a full 64-bit address. As such, one would expect that the normal address bounds checks on the sign-extended 64-bit address would be sufficient. Unfortunately, there are cases in which a program running on a 32-bit processor can generate a data address that is legal in 32 bits, but which is not appropriately sign-extended into 64-bits. For example, consider the following code example:

```
la      r10, 0x80000000
lw      r10, -4(r10)
```

The results of executing this address calculation on 32-bit and 64-bit processors with UX equal zero is shown below:

32-bit Processor	64-bit Processor
0x8000 0000	0xFFFF FFFF 8000 0000
+0xFFFF FFFC	+0xFFFF FFFF FFFF FFFC
0x7FFF FFFC	0xFFFF FFFF 7FFF FFFC

On a 32-bit processor, the result of this address calculation results in a valid, useg address. On a 64-bit processor, however, the sign-extended address in the base register is added to the sign-extended displacement as a 64-bit quantity which results in a carry-out of bit 31, producing an address that is not properly sign extended.

To provide backward compatibility with 32-bit User Mode code, MIPS64 compliant processors must implement the following special case for data references (and explicitly *not* for instruction references) when the processor is running in User Mode and the UX bit is zero in the *Status* register:

The effective address calculated by a load, store, or prefetch instruction must be sign extended from bit 31 into bits 63..32 of the full 64-bit address, ignoring the previous contents of bits 63..32 of the address, before the final address is checked for address error exceptions or used to access the TLB or cache. This special-case behavior is not performed for instruction references.

This results in a properly zero-extended address for all legal data addresses (which cleans up the address shown in the example above), and results in a properly sign-extended address for all illegal data addresses (those in which bit 31 is a one). Code running in Debug Mode, Kernel Mode, or Supervisor Mode with the appropriate 64-bit address enable off is prohibited from generating an effective address in which there is a carry-out of bit 31. If such an address is produced, the operation of the instruction generating such an address is **UNPREDICTABLE**.

4.6.10 TLB-Based Virtual Address Translation

This section describes the TLB-based virtual address translation mechanism. If a TLB-based translation mechanism is implemented, it must be the one described below. Note that sufficient TLB entries must be implemented to avoid a TLB exception loop on load and store instructions. The absolute minimum is therefore two entries, but the realistic minimum is a function of the operating system running on the processor. Sixteen entries is a realistic minimum for simple operating systems. More may be required for complex operating systems.

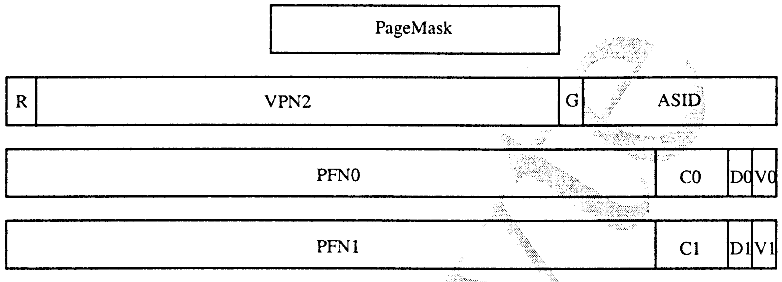
4.6.10.1 Address Space Identifiers (ASID)

The TLB-based translation mechanism supports Address Space Identifiers to uniquely identify the same virtual address across different processes. The operating system assigns ASIDs to each process and the TLB keeps track of the ASID when doing address translation. In certain circumstances, the operating system may wish to associate the same virtual address with all processes. To address this need, the TLB includes a global (G) bit which over-rides the ASID comparison during translation.

4.6.10.2 TLB Organization

The TLB is a fully-associative structure which is used to translate virtual addresses. Each entry contains two logical components: a comparison section and a physical translation section. The comparison section includes the mapping region specifier (R) and the virtual page number (actually, the virtual page number/2 since each entry maps two physical pages, VPN2) of the entry, the ASID, the G(lobal) bit and a recommended mask field which provides the ability to map different page sizes with a single entry. The physical translation section contains a pair of entries, each of which contains the physical page frame number (PFN), a valid (V) bit, a dirty (D) bit, and a cache coherency field (C). There are two entries in the translation section for each TLB entry because each TLB entry maps an aligned pair of virtual pages and the pair of physical translation entries corresponds to the even and odd pages of the pair. Figure 12 shows the logical arrangement of a TLB entry. The physical arrangement of the TLB entry data is implementation dependent, and the implemented size of the R, VPN2, PFN0, and PFN1 fields can vary as a function of virtual address modes supported (32-bit versus 64-bit) and of the needs of the implementation.

Figure 12: Contents of a TLB Entry



The fields of the TLB entry correspond exactly to the fields in the CP0 PageMask, EntryHi, EntryLo0 and EntryLo1 registers. The even page entries in the TLB (e.g., PFN0) come from EntryLo0. Similarly, odd page entries come from EntryLo1.

4.6.10.3 Address Translation

When an address translation is requested, the virtual page number and the current process ASID are presented to the TLB. All entries are checked simultaneously for a match, which occurs when all of the following conditions are true:

- The current process ASID (as obtained from the *EntryHi* register) matches the ASID field in the TLB entry, or the G bit is set in the TLB entry.
- Bits 63:62 of the virtual address match the region code in the R field of the TLB entry.
- The appropriate bits of the virtual page number match the corresponding bits of the VPN2 field stored within the TLB entry. The "appropriate" number of bits is determined by the PageMask field in each entry by performing an ANDNOT operation on both the virtual page number and the TLB VPN2 field. This allows each entry of the TLB to support a different page size, as determined by the PageMask register at the time that the TLB entry was written. If the recommended PageMask register is not implemented, the TLB operation is as if the PageMask register was written with a zero.

If a TLB entry matches the address and ASID presented, the corresponding PFN, C, V, and D bits are read from the translation section of the TLB entry. Which of the two PFN entries is read is a function of the virtual address bit immediately to the right of the section masked with the PageMask entry.

The valid and dirty bits determine the final success of the translation. If the valid bit is off, the entry is not valid and a TLB Invalid exception is raised. If the dirty bit is off and the reference was a store, a TLB Modified exception is raised. If there is an address match with a valid entry and no dirty exception, the PFN and the cache attribute bits are appended to the offset-within-page bits of the address to form the final physical address with attributes.

The TLB lookup process can be described as follows:

```
found ← 0
for i in 0...TLBEntries-1
    if (TLB[i]R = $va_{63..62}$) and
      (($TLB[i]_{VPN2}$ and not ($TLB[i]_{Mask}$)) = ($va_{SEGBITS\text{-}1..13}$ and not ($TLB[i]_{Mask}$))) and
      ($TLB[i]_G$ or ($TLB[i]_{ASID}$ = $EntryHi_{ASID}$)) then
        # EvenOddBit selects between even and odd halves of the TLB as a function of
        # the page size in the matching TLB entry
        case $TLB[i]_{Mask}$
            $000000000000_2$: EvenOddBit ← 12
            $000000000011_2$: EvenOddBit ← 14
            $000000001111_2$: EvenOddBit ← 16
            $000000111111_2$: EvenOddBit ← 18
            $000011111111_2$: EvenOddBit ← 20
            $001111111111_2$: EvenOddBit ← 22
            $111111111111_2$: EvenOddBit ← 24
            otherwise:      UNDEFINED
        endcase
        if $va_{EvenOddBit}$ = 0 then
            pfn ← $TLB[i]_{PFN0}$
            v ← $TLB[i]_{V0}$
            c ← $TLB[i]_{C0}$
            d ← $TLB[i]_{D0}$
        else
            pfn ← $TLB[i]_{PFN1}$
            v ← $TLB[i]_{V1}$
            c ← $TLB[i]_{C1}$
            d ← $TLB[i]_{D1}$
        endif
        if v = 0 then
            InitiateTLBInvalidException(reftype)
        endif
        if (d = 0) and (reftype = store) then
            InitiateTLBModifiedException()
        endif
        # $pfn_{PABITS\text{-}1\text{-}12..0}$ corresponds to $pa_{PABITS\text{-}1..12}$
        pa ← $pfn_{PABITS\text{-}1\text{-}12..EvenOddBit\text{-}12}$ || $va_{EvenOddBit\text{-}1..0}$
        found ← 1
        break
    endif
endfor
if found = 0 then
    InitiateTLBMissException(reftype, VA64Enable)
endif
```

It is implementation dependent whether the VPN2, PFN0, and PFN1 fields of the TLB are stored with the original value, or are pre-masked by the Mask value on a TLB write. This provides implementations with the flexibility of eliminating the "and not $TLB[i]_{Mask}$" terms in the pseudo code above. Note that the virtual address must still be masked with the $TLB[i]_{Mask}$ value in either case.

Table 52 demonstrates how the physical address is generated as a function of the page size of the TLB entry that matches the virtual address. The "Even/Odd Select" column of Table 52 indicates which virtual address bit is used to select between the even (EntryLo0) or odd (EntryLo1) entry in the matching TLB entry. The "PA generated from" column specifies how the physical address is generated from the selected PFN and the offset-in-page bits in the virtual address. In this column, PFN is the physical page number as loaded into the TLB from the EntryLo0 or EntryLo1 registers, and has the bit range $\text{PFN}_{PABITS\text{-}1\text{-}12..0}$, corresponding to $\text{PA}_{PABITS\text{-}1..12}$.

Table 52: Physical Address Generation

Page Size	Even/Odd Select	PA generated from
4K Bytes	VA_{12}	$\text{PFN}_{PABITS\text{-}1\text{-}12..0}$ || $\text{VA}_{11..0}$
16K Bytes	VA_{14}	$\text{PFN}_{PABITS\text{-}1\text{-}12..2}$ || $\text{VA}_{13..0}$
64K Bytes	VA_{16}	$\text{PFN}_{PABITS\text{-}1\text{-}12..4}$ || VA15..0
256K Bytes	VA_{18}	$\text{PFN}_{PABITS\text{-}1\text{-}12..6}$ || $\text{VA}_{17..0}$
1M Bytes	VA_{20}	$\text{PFN}_{PABITS\text{-}1\text{-}12..8}$ || $\text{VA}_{19..0}$
4M Bytes	VA_{22}	$\text{PFN}_{PABITS\text{-}1\text{-}12..10}$ || $\text{VA}_{21..0}$
16M Bytes	VA_{24}	$\text{PFN}_{PABITS\text{-}1\text{-}12..12}$ || $\text{VA}_{23..0}$

4.7 Interrupts

The processor supports eight interrupt requests, broken down into four categories:

- Software interrupts - Two software interrupt requests are made via software writes to bits IP0 and IP1 of the *Cause* register.
- Hardware interrupts - Six hardware interrupt requests numbered 0 through 5 are made via implementation-dependent external requests to the processor.
- Timer interrupt - A timer interrupt is raised when the *Count* and *Compare* registers reach the same value.
- Performance counter interrupt - A performance counter interrupt is raised when the most significant bit of the counter is a one, and the interrupt is enabled by the IE bit in the performance counter control register.

Timer interrupts, performance counter interrupts, and hardware interrupt 5 are combined in an implementation-dependent way to create the ultimate hardware interrupt 5.

The current interrupt requests are visible via the IP field in the *Cause* register on any read of that register (not just after an interrupt exception has occurred). The mapping of *Cause* register bits to the various interrupt requests is shown in Table 53.

Table 53: Mapping of Interrupts to the *Cause* and *Status* Registers

		Cause Register Bit		*Status* Register Bit	
Interrupt Type	**Interrupt Number**	**Number**	**Name**	**Number**	**Name**
Software Interrupt	0	8	IP0	8	IM0
	1	9	IP1	9	IM1

Table 53: Mapping of Interrupts to the *Cause* and *Status* Registers

		Cause Register Bit		*Status* Register Bit	
Interrupt Type	**Interrupt Number**	**Number**	**Name**	**Number**	**Name**
Hardware Interrupt	0	10	IP2	10	IM2
	1	11	IP3	11	IM3
	2	12	IP4	12	IM4
	3	13	IP5	13	IM5
	4	14	IP6	14	IM6
Hardware Interrupt, Timer Interrupt, or Performance Counter Interrupt	5	15	IP7	15	IM7

For each bit of the IP field in the *Cause* register there is a corresponding bit in the IM field in the *Status* register. An interrupt is only taken when all of the following are true:

- An interrupt request bit is a one in the IP field of the *Cause* register.
- The corresponding mask bit is a one in the IM field of the *Status* register. The mapping of bits is shown in Table 53.
- The IE bit in the *Status* register is a one.
- The DM bit in the *Debug* register is a zero (for processor implementing EJTAG)
- The EXL and ERL bits in the *Status* register are both zero.

Logically, the IP field of the *Cause* register is bit-wise ANDed with the IM field of the *Status* register, the eight resultant bits are ORed together and that value is ANDed with the IE bit of the *Status* register. The final interrupt request is then asserted only if both the EXL and ERL bits in the *Status* register are zero, and the DM bit in the *Debug* register is zero, corresponding to a non-exception, non-error, non-debug processing mode.

4.8 Exceptions

Normal execution of instructions may be interrupted when an exception occurs. Such events can be generated as a by-product of instruction execution (e.g., an integer overflow caused by an add instruction or a TLB miss caused by a load instruction), or by an event not directly related to instruction execution (e.g., an external interrupt). When an exception occurs, the processor stops processing instructions, saves sufficient state to resume the interrupted instruction stream, enters kernel mode, and starts a software exception handler. The saved state and the address of the software exception handler are a function of both the type of exception, and the current state of the processor.

4.8.1 Exception Priority

Table 54 lists all possible exceptions, and the relative priority of each, highest to lowest.

Table 54: Priority of Exceptions

Exception	Description	Type
Reset	The Cold Reset signal was asserted to the processor	Asynchronous Reset
Soft Reset	The Reset signal was asserted to the processor	

Table 54: Priority of Exceptions

Exception	Description	Type
Debug Single Step	An EJTAG Single Step occurred. Prioritized above other exceptions, including asynchronous exceptions, so that one can single-step into interrupt (or other asynchronous) handlers	Synchronous Debug
Debug Interrupt	An EJTAG interrupt (EjtagBrk or DINT) was asserted	Asynchronous Debug
Imprecise Debug Data Break	An imprecise EJTAG data break condition was asserted	
Nonmaskable Interrupt (NMI)	The NMI signal was asserted to the processor	Asynchronous
Machine Check	An internal inconsistency was detected by the processor	
Interrupt	An enabled interrupt occurred	
Deferred Watch	A watch exception, deferred because EXL was one when the exception was detected, was asserted after EXL went to zero	
Debug Instruction Break	An EJTAG instruction break condition was asserted. Prioritized above instruction fetch exceptions to allow break on illegal instruction addresses.	Synchronous Debug
Watch - Instruction fetch	A watch address match was detected on an instruction fetch. Prioritized above instruction fetch exceptions to allow watch on illegal instruction addresses.	Synchronous
Address Error - Instruction fetch	A non-word-aligned address was loaded into PC	
TLB/XTLB Refill - Instruction fetch	A TLB miss occurred on an instruction fetch	
TLB Invalid - Instruction fetch	The valid bit was zero in the TLB entry mapping the address referenced by an instruction fetch	
Cache Error - Instruction fetch	A cache error occurred on an instruction fetch	
Bus Error - Instruction fetch	A bus error occurred on an instruction fetch	
SDBBP	An EJTAG SDBBP instruction was executed	Synchronous Debug
Instruction Validity Exceptions	An instruction could not be completed because it was not allowed access to the required resources, or was illegal: Coprocessor unusable, reserved instruction. If both exceptions occur on the same instruction, the Coprocessor Unusable Exception takes priority.	Synchronous
Execution Exception	An instruction-based exception occurred: Integer overflow, trap, system call, breakpoint, floating point exception	

Table 54: Priority of Exceptions

Exception	Description	Type
Precise Debug Data Break	A precise EJTAG data break on load/store (address match only) or a data break on store (address+data match) condition was asserted. Prioritized above data fetch exceptions to allow break on illegal data addresses.	Synchronous Debug
Watch - Data access	A watch address match was detected on the address referenced by a load or store. Prioritized above data fetch exceptions to allow watch on illegal data addresses.	Synchronous
Address error - Data access	An unaligned address, or an address that was inaccessible in the current processor mode was referenced, by a load or store instruction	
TLB/XTLB Refill - Data access	A TLB miss occurred on a data access	
TLB Invalid - Data access	The valid bit was zero in the TLB entry mapping the address referenced by a load or store instruction	
TLB Modified - Data access	The dirty bit was zero in the TLB entry mapping the address referenced by a store instruction	
Cache Error - Data access	a cache error occurred on a load or store data reference	
Bus Error - Data access	A bus error occurred on a load or store data reference	
Precise Debug Data Break	A precise EJTAG data break on load (address+data match only) condition was asserted. Prioritized last because all aspects of the data fetch must complete in order to do data match.	Synchronous Debug

The "Type" column of Table 54 describes the type of exception. Table 55 explains the characteristics of each exception type.

Table 55: Exception Type Characteristics

Exception Type	Characteristics
Asynchronous Reset	Denotes a reset-type exception that occurs asynchronously to instruction execution. These exceptions always have the highest priority to guarantee that the processor can always be placed in a runnable state.
Asynchronous Debug	Denotes an EJTAG debug exception that occurs asynchronously to instruction execution. These exceptions have very high priority with respect to other exceptions because of the desire to enter Debug Mode, even in the presence of other exceptions, both asynchronous and synchronous.
Asynchronous	Denotes any other type of exception that occurs asynchronously to instruction execution. These exceptions are shown with higher priority than synchronous exceptions mainly for notational convenience. If one thinks of asynchronous exceptions as occurring between instructions, they are either the lowest priority relative to the previous instruction, or the highest priority relative to the next instruction. The ordering of the table above considers them in the second way.

Table 55: Exception Type Characteristics

Exception Type	Characteristics
Synchronous Debug	Denotes an EJTAG debug exception that occurs as a result of instruction execution, and is reported precisely with respect to the instruction that caused the exception. These exceptions are prioritized above other synchronous exceptions to allow entry to Debug Mode, even in the presence of other exceptions.
Synchronous	Denotes any other exception that occurs as a result of instruction execution, and is reported precisely with respect to the instruction that caused the exception. These exceptions tend to be prioritized below other types of exceptions, but there is a relative priority of synchronous exceptions with each other.

4.8.2 Exception Vector Locations

The Reset, Soft Reset, and NMI exceptions are always vectored to location 0xFFFF FFFF BFC0 0000. EJTAG Debug exceptions are vectored to location 0xFFFF FFFF BFC0 0480 or to location 0xFFFF FFFF FF20 0200 if the ProbEn bit is zero or one, respectively, in the EJTAG_Control_register. Addresses for all other exceptions are a combination of a vector offset and a base address. Table 56 gives the base address as a function of the exception and whether the BEV bit is set in the *Status* register. Table 57 gives the offsets from the base address as a function of the exception. Table 58 combines these two tables into one that contains all possible vector addresses as a function of the state that can affect the vector selection.

Table 56: Exception Vector Base Addresses

Exception	$\text{Status}_{\text{BEV}}$	
	0	1
Reset, Soft Reset, NMI	0xFFFF FFFF BFC0 0000	
EJTAG Debug (with ProbEn = 0 in the EJTAG_Control_register)	0xFFFF FFFF BFC0 0480	
EJTAG Debug (with ProbEn = 1 in the EJTAG_Control_register)	0xFFFF FFFF FF20 0200	
Cache Error	0xFFFF FFFF A000 0000	0xFFFF FFFF BFC0 0200
Other	0xFFFF FFFF 8000 0000	0xFFFF FFFF BFC0 0200

Table 57: Exception Vector Offsets

Exception	Vector Offset
TLB Refill, EXL = 0	0x000
64-bit XTLB Refill, EXL = 0	0x080
Cache error	0x100
General Exception	0x180
Interrupt, Cause_{IV} = 1	0x200
Reset, Soft Reset, NMI	None (Uses Reset Base Address)

Table 58: Exception Vectors

Exception	BEV	EXL	IV	EJTAG ProbEn	Vector
Reset, Soft Reset, NMI	x	x	x	x	0xFFFF FFFF BFC0 0000
EJTAG Debug	x	x	x	0	0xFFFF FFFF BFC0 0480
EJTAG Debug	x	x	x	1	0xFFFF FFFF FF20 0200
TLB Refill	0	0	x	x	0xFFFF FFFF 8000 0000
XTLB Refill	0	0	x	x	0xFFFF FFFF 8000 0080
TLB Refill	0	1	x	x	0xFFFF FFFF 8000 0180
XTLB Refill	0	1	x	x	0xFFFF FFFF 8000 0180
TLB Refill	1	0	x	x	0xFFFF FFFF BFC0 0200
XTLB Refill	1	0	x	x	0xFFFF FFFF BFC0 0280
TLB Refill	1	1	x	x	0xFFFF FFFF BFC0 0380
XTLB Refill	1	1	x	x	0xFFFF FFFF BFC0 0380
Cache Error	0	x	x	x	0xFFFF FFFF A000 0100
Cache Error	1	x	x	x	0xFFFF FFFF BFC0 0300
Interrupt	0	0	0	x	0xFFFF FFFF 8000 0180
Interrupt	0	0	1	x	0xFFFF FFFF 8000 0200
Interrupt	1	0	0	x	0xFFFF FFFF BFC0 0380
Interrupt	1	0	1	x	0xFFFF FFFF BFC0 0400
All others	0	x	x	x	0xFFFF FFFF 8000 0180
All others	1	x	x	x	0xFFFF FFFF BFC0 0380
'x' denotes don't care					

4.8.3 General Exception Processing

With the exception of Reset, Soft Reset, and NMI exceptions, which have their own special processing as described below, exceptions have the same basic processing flow:

- If the EXL bit in the *Status* register is zero, the *EPC* register is loaded with the PC at which execution will be restarted and the BD bit is set appropriately in the *Cause* register. The value loaded into the *EPC* register is the current PC if the instruction is not in the delay slot of a branch, or PC-4 if the instruction is in the delay slot of a branch. If the EXL bit in the *Status* register is set, the *EPC* register is not loaded and the BD bit is not changed in the *Cause* register.
- The CE, and ExcCode fields of the *Cause* registers are loaded with the values appropriate to the exception. The CE field is loaded, but not defined, for any exception type other than a coprocessor unusable exception.
- The EXL bit is set in the *Status* register.
- The processor is started at the exception vector.

The value loaded into EPC represents the restart address for the exception and need not be modified by exception handler software in the normal case. Software need not look at the BD bit in the Cause register unless is wishes to identify the address of the instruction that actually caused the exception.

Note that individual exception types may load additional information into other registers. This is noted in the description of each exception type below.

Operation:

```
if $Status_{EXL} = 0$
    if InstructionInBranchDelaySlot then
        $EPC \leftarrow PC - 4$
        $Cause_{BD} \leftarrow 1$
    else
        $EPC \leftarrow PC$
        $Cause_{BD} \leftarrow 0$
    endif
    if ExceptionType = TLBRefill then
        vectorOffset ← 0x000
    elseif (ExceptionType = XTLBRefill) then
        vectorOffset ← 0x080
    elseif (ExceptionType = Interrupt) and
         ($Cause_{IV} = 1$) then
        vectorOffset ← 0x200
    else
        vectorOffset ← 0x180
    endif
else
    vectorOffset ← 0x180
endif
$Cause_{CE} \leftarrow$ FaultingCoprocessorNumber
$Cause_{ExcCode} \leftarrow$ ExceptionType
$Status_{EXL} \leftarrow 1$
if $Status_{BEV} = 1$ then
    PC ← 0xFFFF FFFF BFC0 0200 + vectorOffset
else
    PC ← 0xFFFF FFFF 8000 0000 + vectorOffset
endif
```

4.8.4 EJTAG Debug Exception

An EJTAG Debug Exception occurs when one of a number of EJTAG-related conditions is met. Refer to the EJTAG Specification for details of this exception.

Entry Vector Used

0xFFFF FFFF BFC0 0480 if the ProbEn bit is zero in the EJTAG_Control_register; 0xFFFF FFFF FF20 0200 if the ProbEn bit is one.

4.8.5 Reset Exception

A Reset Exception occurs when the Cold Reset signal is asserted to the processor. This exception is not maskable. When a Reset Exception occurs, the processor performs a full reset initialization, including aborting state machines, establishing critical state, and generally placing the processor in a state in which it can execute instructions from uncached, unmapped address space. On a Reset Exception, the state of the processor in not defined, with the following exceptions:

- The *Random* register is initialized to the number of TLB entries - 1.
- The *Wired* register is initialized to zero.
- The *Config* and *Config1* registers are initialized with their boot state.
- The BEV, TS, SR, NMI, ERL, and RP fields of the *Status* register are initialized to a specified state.
- Watch register enables and Performance Counter register interrupt enables are cleared.
- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC. Note that this value may or may not be predictable if the Reset Exception was taken as the result of power being applied to the processor because PC may not have a valid value in that case. In some implementations, the value loaded into *ErrorEPC* register may not be predictable on either a Reset or Soft Reset Exception.
- PC is loaded with 0xFFFF FFFF BFC0 0000.

***Cause* Register ExcCode Value**

None

Additional State Saved

None

Entry Vector Used

Reset (0xFFFF FFFF BFC0 0000)

Operation

```
Random ← TLBEntries - 1
Wired ← 0
Config ← ConfigurationState
$\text{Config}_{K0}$ ← 2                    # Suggested - see Config register description
Config1 ← ConfigurationState
$\text{Status}_{BEV}$ ← 1
$\text{Status}_{TS}$ ← 0
$\text{Status}_{SR}$ ← 0
$\text{Status}_{NMI}$ ← 0
$\text{Status}_{ERL}$ ← 1
$\text{Status}_{RP}$ ← 0
$\text{WatchLo[n]}_{I}$ ← 0              # For all implemented Watch registers
$\text{WatchLo[n]}_{R}$ ← 0              # For all implemented Watch registers
$\text{WatchLo[n]}_{W}$ ← 0              # For all implemented Watch registers
$\text{PerfCnt.Control[n]}_{IE}$ ← 0     # For all implemented PerfCnt registers
if InstructionInBranchDelaySlot then
    ErrorEPC ← PC - 4
else
    ErrorEPC ← PC
endif
PC ← 0xFFFF FFFF BFC0 0000
```

4.8.6 Soft Reset Exception

A Soft Reset Exception occurs when the Reset signal is asserted to the processor. It is implementation dependent whether Soft Reset is implemented. If the Soft Reset Exception is not implemented, the Reset Exception should be used instead. This exception is not maskable. When a Soft Reset Exception occurs, the processor performs a subset of the full reset initialization. Although a Soft Reset Exception does not unnecessarily change the state of the processor, it may be forced to do so in order to place the processor in a state in which it can execute instructions from uncached, unmapped address space. Since bus, cache, or other operations may be interrupted, portions of the cache, memory, or other processor state may be inconsistent. In addition to any hardware initialization required, the following state is established on a Soft Reset Exception:

- The BEV, TS, SR, NMI, ERL, and RP fields of the *Status* register are initialized to a specified state.
- Watch register enables and Performance Counter register interrupt enables are cleared.
- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC. Note that this value may or may not be predictable.
- PC is loaded with 0xFFFF FFFF BFC0 0000.

Cause **Register ExcCode Value**

None

Additional State Saved

None

Entry Vector Used

Reset (0xFFFF FFFF BFC0 0000)

Operation

```
$Config_{K0} \leftarrow 2$                      # Suggested - see Config register description
$Status_{BEV} \leftarrow 1$
$Status_{TS} \leftarrow 0$
$Status_{SR} \leftarrow 1$
$Status_{NMI} \leftarrow 0$
$Status_{ERL} \leftarrow 1$
$Status_{RP} \leftarrow 0$
$WatchLo[n]_{I} \leftarrow 0$                   # For all implemented Watch registers
$WatchLo[n]_{R} \leftarrow 0$                   # For all implemented Watch registers
$WatchLo[n]_{W} \leftarrow 0$                   # For all implemented Watch registers
$PerfCnt.Control[n]_{IE} \leftarrow 0$          # For all implemented PerfCnt registers
if InstructionInBranchDelaySlot then
    $ErrorEPC \leftarrow PC - 4$
else
    $ErrorEPC \leftarrow PC$
endif
$PC \leftarrow$ 0xFFFF FFFF BFC0 0000
```

4.8.7 Non Maskable Interrupt (NMI) Exception

A non maskable interrupt exception occurs when the NMI signal is asserted to the processor. It is implementation dependent whether the NMI exception is implemented. However, several embedded operating systems make use of the NMI exception, so its implementation is strongly recommended.

Unlike all other interrupts, this exception is not maskable. An NMI occurs only at instruction boundaries, so does not do any reset or other hardware initialization. The state of the cache, memory, and other processor state is consistent and all registers are preserved, with the following exceptions:

- The BEV, TS, SR, NMI, and ERL fields of the *Status* register are initialized to a specified state.
- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC.
- PC is loaded with 0xFFFF FFFF BFC0 0000.

Cause **Register ExcCode Value**

None

Additional State Saved

None

Entry Vector Used

Reset (0xFFFF FFFF BFC0 0000)

Operation

```
$\text{Status}_{\text{BEV}} \leftarrow 1$
$\text{Status}_{\text{TS}} \leftarrow 0$
$\text{Status}_{\text{SR}} \leftarrow 0$
$\text{Status}_{\text{NMI}} \leftarrow 1$
$\text{Status}_{\text{ERL}} \leftarrow 1$
if InstructionInBranchDelaySlot then
    ErrorEPC ← PC - 4
else
    ErrorEPC ← PC
endif
PC ← 0xFFFF FFFF BFC0 0000
```

4.8.8 Machine Check Exception

A machine check exception occurs when the processor detects an internal inconsistency. It is implementation dependent whether the Machine Check Exception is implemented. If no internal consistency checking is performed by the processor, the Machine Check Exception need not be implemented.

The following conditions cause a machine check exception:

- Detection of multiple matching entries in the TLB in a TLB-based MMU. It is implementation dependent whether this condition is detected on the TLB write that creates multiple matching entries, or on a reference that detects them. In either case, the TS bit in the *Status* register is set to indicate this condition. It is implementation dependent whether this condition can be corrected in the software exception handler, perhaps by flushing the entire TLB. If the condition can be corrected, software must clear this bit before resuming normal operation.

  If the condition is detected during a TLB write, processors should attempt to preserve the entry already in the TLB, if possible, as that provides the most information for software debug of the problem.

***Cause* Register ExcCode Value**

MCheck

Additional State Saved

Depends on the condition that caused the exception. See the descriptions above.

Entry Vector Used

General exception vector (offset 0x180)

4.8.9 Address Error Exception

An address error exception occurs under the following circumstances:

- A load or store doubleword instruction is executed in which the address is not aligned on a doubleword boundary.
- An instruction is fetched from an address that is not aligned on a word boundary.
- A load or store word instruction is executed in which the address is not aligned on a word boundary.
- A load or store halfword instruction is executed in which the address is not aligned on a halfword boundary.
- A reference is made to a kernel address space from User Mode or Supervisor Mode.

- A reference is made to a supervisor address space from User Mode.
- A reference is made to a a 64-bit address that is outside the range of the 32-bit Compatibility Address Space when 64-bit address references are not enabled.
- A reference is made to an undefined or unimplemented 64-bit address when 64-bit address references are enabled.

Note that in the case of an instruction fetch that is not aligned on a word boundary, PC is updated before the condition is detected. Therefore, both EPC and BadVAddr point at the unaligned instruction address.

***Cause* Register ExcCode Value**

AdEL: Reference was a load or an instruction fetch
AdES: Reference was a store

Additional State Saved

Register State	Value
BadVAddr	failing address
$\text{Context}_{\text{VPN2}}$	**UNPREDICTABLE**
$\text{XContext}_{\text{VPN2}}$ $\text{XContext}_{\text{R}}$	**UNPREDICTABLE**
$\text{EntryHi}_{\text{VPN2}}$ $\text{EntryHi}_{\text{R}}$	**UNPREDICTABLE**
EntryLo0	**UNPREDICTABLE**
EntryLo1	**UNPREDICTABLE**

Entry Vector Used

General exception vector (offset 0x180)

4.8.10 TLB Refill and XTLB Refill Exceptions

A TLB Refill or XTLB Refill exception occurs in a TLB-based MMU when no TLB entry matches a reference to a mapped address space and the EXL bit is zero in the *Status* register. Note that this is distinct from the case in which an entry matches but has the valid bit off, in which case a TLB Invalid exception occurs. Refill exceptions have distinct exception vector offsets: 0x000 for a 32-bit TLB Refill and 0x080 for a 64-bit extended TLB ("XTLB") refill. The XTLB refill handler is used whenever a reference is made to an enabled 64-bit address space.

***Cause* Register ExcCode Value**

TLBL: Reference was a load or an instruction fetch
TLBS: Reference was a store

Additional State Saved

Register State	Value
BadVAddr	failing address

Register State	Value
Context	The BadVPN2 field contains $VA_{31:13}$ of the failing address
XContext	The XContext BadVPN2 field contains $VA_{\mathit{SEGBITS}-1:13}$, and the XContext R field contains $VA_{63:62}$ of the failing address.
EntryHi	The EntryHi VPN2 field contains $VA_{\mathit{SEGBITS}-1:13}$ of the failing address and the EntryHi R field contains $VA_{63:62}$ of the failing address; the ASID field contains the ASID of the reference that missed
EntryLo0	**UNPREDICTABLE**
EntryLo1	**UNPREDICTABLE**

Entry Vector Used

- TLB Refill vector (offset 0x000) if 64-bit addresses are not enabled and $Status_{EXL}$ = 0 at the time of exception.
- XTLB Refill vector (offset 0x080) if 64-bit addresses are enabled and $Status_{EXL}$ = 0 at the time of exception.
- General exception vector (offset 0x180) in either case if $Status_{EXL}$ = 1 at the time of exception

4.8.11 TLB Invalid Exception

A TLB invalid exception occurs when a TLB entry matches a reference to a mapped address space, but the matched entry has the valid bit off.

Note that the condition in which no TLB entry matches a reference to a mapped address space and the EXL bit is one in the *Status* register is indistinguishable from a TLB Invalid Exception in the sense that both use the general exception vector and supply an ExcCode value of TLBL or TLBS. The only way to distinguish these two cases is by probing the TLB for a matching entry (using TLBP).

***Cause* Register ExcCode Value**

TLBL: Reference was a load or an instruction fetch
TLBS: Reference was a store

Additional State Saved

Register State	Value
BadVAddr	failing address
Context	The BadVPN2 field contains $VA_{31:13}$ of the failing address
XContext	The XContext BadVPN2 field contains $VA_{\mathit{SEGBITS}-1:13}$, and the XContext R field contains $VA_{63:62}$ of the failing address.

Register State	Value
EntryHi	The EntryHi VPN2 field contains $VA_{SEGBITS\text{-}1:13}$ of the failing address and the EntryHi R field contains $VA_{63:62}$ of the failing address; the ASID field contains the ASID of the reference that missed
EntryLo0	**UNPREDICTABLE**
EntryLo1	**UNPREDICTABLE**

Entry Vector Used

General exception vector (offset 0x180)

4.8.12 TLB Modified Exception

A TLB modified exception occurs on a *store* reference to a mapped address when the matching TLB entry is valid, but the entry's D bit is zero, indicating that the page is not writable.

***Cause* Register ExcCode Value**

Mod

Additional State Saved

Register State	Value
BadVAddr	failing address
Context	The BadVPN2 field contains $VA_{31:13}$ of the failing address
XContext	The XContext BadVPN2 field contains $VA_{SEGBITS\text{-}1:13}$, and the XContext R field contains $VA_{63:62}$ of the failing address.
EntryHi	The EntryHi VPN2 field contains $VA_{SEGBITS\text{-}1:13}$ of the failing address and the EntryHi R field contains $VA_{63:62}$ of the failing address; the ASID field contains the ASID of the reference that missed
EntryLo0	**UNPREDICTABLE**
EntryLo1	**UNPREDICTABLE**

Entry Vector Used

General exception vector (offset 0x180)

4.8.13 Cache Error Exception

A cache error exception occurs when an instruction or data reference detects a cache tag or data error, or a parity or ECC error is detected on the system bus when a cache miss occurs. This exception is not maskable. Because the error was in a cache, the exception vector is to an unmapped, uncached address. It is implementation dependent whether a cache error exception resulting from an access to the data cache is reported precisely with respect to the instruction that caused the cache error.

***Cause* Register ExcCode Value**
N/A

Additional State Saved

Register State	Value
CacheErr	Error state
ErrorEPC	PC

Entry Vector Used
Cache error vector (offset 0x100)

Operation

```
CacheErr ← ErrorState
Status$_{ERL}$ ← 1
if InstructionInBranchDelaySlot then
    ErrorEPC ← PC - 4
else
    ErrorEPC ← PC
endif
if Status$_{BEV}$ = 1 then
    PC ← 0xFFFF FFFF BFC0 0200 + 0x100
else
    PC ← 0xFFFF FFFF A000 0000 + 0x100
endif
```

4.8.14 Bus Error Exception

A bus error occurs when an instruction, data, or prefetch access makes a bus request (due to a cache miss or an uncacheable reference) and that request is terminated in an error. Note that parity errors detected during bus transactions are reported as cache error exceptions, not bus error exceptions. It is implementation dependent whether a data bus error exception is reported precisely with respect to the instruction that caused the bus error.

***Cause* Register ExcCode Value**

IBE:	Error on an instruction reference
DBE:	Error on a data reference

Additional State Saved
None

Entry Vector Used
General exception vector (offset 0x180)

4.8.15 Integer Overflow Exception

An integer overflow exception occurs when selected integer instructions result in a 2's complement overflow.

***Cause* Register ExcCode Value**

Ov

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

4.8.16 Trap Exception

A trap exception occurs when a trap instruction results in a TRUE value.

***Cause* Register ExcCode Value**

Tr

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

4.8.17 System Call Exception

A system call exception occurs when a SYSCALL instruction is executed.

***Cause* Register ExcCode Value**

Sys

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

4.8.18 Breakpoint Exception

A breakpoint exception occurs when a BREAK instruction is executed.

***Cause* Register ExcCode Value**

Bp

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

4.8.19 Reserved Instruction Exception

A Reserved Instruction Exception occurs if any of the following conditions is true:

- An instruction was executed that specifies an encoding of the opcode field (Table 20) that is flagged with "*" (reserved), "β" (higher-order ISA), "⊥" (64-bit) if 64-bit operations are not enabled, or an unimplemented "ε" (ASE).
- An instruction was executed that specifies a *SPECIAL* opcode encoding of the function field (Table 21) that

is flagged with "*" (reserved), "β" (higher-order ISA), or "⊥" (64-bit) if 64-bit operations are not enabled.

- An instruction was executed that specifies a *REGIMM* opcode encoding of the rt field (Table 22) that is flagged with "*" (reserved).
- An instruction was executed that specifies an unimplemented *SPECIAL2* opcode encoding of the function field (Table 23) that is flagged with an unimplemented "θ" (partner available), "⊥" (64-bit) if 64-bit operations are not enabled, or an unimplemented "σ" (EJTAG).
- An instruction was executed that specifies a *COPz* opcode encoding of the rs field (Table 25, Table 27, Table 29) that is flagged with "*" (reserved), "β" (higher-order ISA), "⊥" (64-bit) if 64-bit operations are not enabled, or an unimplemented "ε" (ASE), assuming that access to the coprocessor is allowed. If access to the coprocessor is not allowed, a Coprocessor Unusable Exception occurs instead. For the *COP1* opcode, some implementations of previous ISAs reported this case as a Floating Point Exception, setting the Unimplemented Operation bit in the Cause field of the *FCSR* register.
- An instruction was executed that specifies an unimplemented *COP0* opcode encoding of the function field when rs is *CO* (Table 28) that is flagged with "*" (reserved), or an unimplemented "σ" (EJTAG), assuming that access to coprocessor 0 is allowed. If access to the coprocessor is not allowed, a Coprocessor Unusable Exception occurs instead.
- An instruction was executed that specifies a *COP1* opcode encoding of the function field when rs is S, D, or W (Table 30, Table 31, Table 32) that is flagged with "*" (reserved), "β" (higher-order ISA), "⊥" (64-bit) if 64-bit operations are not enabled, or an unimplemented "ε" (ASE), assuming that access to coprocessor 1 is allowed. If access to the coprocessor is not allowed, a Coprocessor Unusable Exception occurs instead. Some implementations of previous ISAs reported this case as a Floating Point Exception, setting the Unimplemented Operation bit in the Cause field of the *FCSR* register.
- An instruction was executed that specifies a *COP1* opcode encoding when rs is L or PS (Table 32, Table 33) and 64-bit operations are not enabled, or with a function field encoding that is flagged with "*" (reserved), "β" (higher-order ISA), or an unimplemented "ε" (ASE), assuming that access to coprocessor 1 is allowed. If access to the coprocessor is not allowed, a Coprocessor Unusable Exception occurs instead. Some implementations of previous ISAs reported this case as a Floating Point Exception, setting the Unimplemented Operation bit in the Cause field of the *FCSR* register.
- An instruction was executed that specifies a COP1X opcode encoding of the function field (Table 35) that is flagged with "*" (reserved), or any execution of the COP1X opcode when 64-bit operations are not enabled, assuming that access to coprocessor 1 is allowed. If access to the coprocessor is not allowed, a Coprocessor Unusable Exception occurs instead. Some implementations of previous ISAs reported this case as a Floating Point Exception, setting the Unimplemented Operation bit in the Cause field of the *FCSR* register.

***Cause* Register ExcCode Value**

RI

Additional State Saved

None

Entry Vector Used

General exception vector (offset 0x180)

4.8.20 Coprocessor Unusable Exception

A coprocessor unusable exception occurs if any of the following conditions is true:

- A COP0 or Cache instruction was executed while the processor was running in a mode other than Debug Mode or Kernel Mode, and the CU0 bit in the *Status* register was a zero
- A COP1, COP1X, LWC1, SWC1, LDC1, SDC1 or MOVCI (Special opcode function field encoding) instruction was executed and the CU1 bit in the *Status* register was a zero.
- A COP2, LWC2, SWC2, LDC2, or SDC2 instruction was executed, and the CU2 bit in the *Status* register was a zero.

***Cause* Register ExcCode Value**

CpU

Additional State Saved

Register State	Value
Cause_{CE}	unit number of the coprocessor being referenced

Entry Vector Used

General exception vector (offset 0x180)

4.8.21 Floating Point Exception

A floating point exception is initiated by the floating point coprocessor to signal a floating point exception.

Register ExcCode Value

FPE

Additional State Saved

Register State	Value
FCSR	indicates the cause of the floating point exception

Entry Vector Used

General exception vector (offset 0x180)

4.8.22 Watch Exception

The watch facility provides a software debugging vehicle by initiating a watch exception when an instruction or data reference matches the address information stored in the *WatchHi* and *WatchLo* registers. A watch exception is taken immediately if the EXL and ERL bits of the *Status* register are both zero. If either bit is a one at the time that a watch exception would normally be taken, the WP bit in the *Cause* register is set, and the exception is deferred until both the EXL and ERL bits in the Status register are zero. Software may use the WP bit in the *Cause* register to determine if the EPC register points at the instruction that caused the watch exception, or if the exception actually occurred while in kernel mode.

If the EXL or ERL bits are one in the *Status* register and a single instruction generates both a watch exception (which is deferred by the state of the EXL and ERL bits) and a lower-priority exception, the lower priority exception is taken. It is implementation dependent whether the WP bit is set in this case. The preferred implementation is to set the WP bit only if the instruction completes with no other exception.

It is implementation dependent whether a data watch exception is triggered by a prefetch or cache instruction whose address matches the Watch register address match conditions. The preferred implementation is not to match on these instructions.

Register ExcCode Value

WATCH

Additional State Saved

Register State	Value
Cause_{WP}	indicates that the watch exception was deferred until after both $\text{Status}_{\text{EXL}}$ and $\text{Status}_{\text{ERL}}$ were zero. This bit directly causes a watch exception, so software must clear this bit as part of the exception handler to prevent a watch exception loop at the end of the current handler execution.

Entry Vector Used

General exception vector (offset 0x180)

4.8.23 Interrupt Exception

The interrupt exception occurs when one or more of the eight interrupt requests is enabled by the Status registers. See Section 4.7 on page 87 for more information.

Register ExcCode Value

Int

Additional State Saved

Register State	Value
Cause_{IP}	indicates the interrupts that are pending.

Entry Vector Used

General exception vector (offset 0x180) if the IV bit in the *Cause* register is zero.
Interrupt vector (offset 0x200) if the IV bit in the *Cause* register is one.

4.9 CP0 Registers

The CP0 registers provide the interface between the ISA and the PRA. Each register is discussed below, with the registers presented in numerical order, first by register number, then by select field number.

For each register described below, field descriptions include the read/write properties of the field, and the reset state of the field. For the read/write properties of the field, the following notation is used:

<table>
<tr><th>Read/Write Notation</th><th>Hardware Interpretation</th><th>Software Interpretation</th></tr>
<tr><td>R/W</td><td colspan="2">A field in which all bits are readable and writable by software and, potentially, by hardware.

Hardware updates of this field are visible by software read. Software updates of this field are visible by hardware read.

If the Reset State of this field is “Undefined”, either software or hardware must initialize the value before the first read will return a predictable value. This should not be confused with the formal definition of <b>UNDEFINED</b> behavior.</td></tr>
<tr><td>R</td><td>A field which is either static or is updated only by hardware.

If the Reset State of this field is either “0” or “Preset”, hardware initializes this field to zero or to the appropriate state, respectively, on powerup.

If the Reset State of this field is “Undefined”, hardware updates this field only under those conditions specified in the description of the field.</td><td>A field to which the value written by software is ignored by hardware. Software may write any value to this field without affecting hardware behavior. Software reads of this field return the last value updated by hardware.

If the Reset State of this field is “Undefined”, software reads of this field result in an <b>UNPREDICTABLE</b> value except after a hardware update done under the conditions specified in the description of the field.</td></tr>
<tr><td>0</td><td>A field which hardware does not update, and for which hardware can assume a zero value.</td><td>A field to which the value written by software must be zero. Software writes of non-zero values to this field may result in <b>UNDEFINED</b> behavior of the hardware. Software reads of this field return zero as long as all previous software writes are zero.

If the Reset State of this field is “Undefined”, software must write this field with zero before it is guaranteed to read as zero.</td></tr>
</table>

4.9.1 Index Register (CP0 Register 0, Select 0)

Compliance Level: *Required* for TLB-based MMUs; *Optional* otherwise.

The *Index* register is a 32-bit read/write register which contains the index used to access the TLB for TLBP, TLBR, and TLBWI instructions. The width of the index field is implementation-dependent as a function of the number of TLB entries that are implemented. The minimum value for TLB-based MMUs is Ceiling(Log2(TLBEntries)).

The operation of the processor is **UNDEFINED** if a value greater than or equal to the number of TLB entries is written to the *Index* register.

Figure 13 shows the format of the *Index* register; Table 59 describes the *Index* register fields.

Figure 13: Index Register

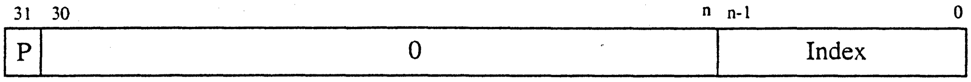


Table 59: Index Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
P	31	Probe Failure. Hardware writes this bit during execution of the TLBP instruction to indicate whether a TLB match occurred: 0: A match occurred, and the Index field contains the index of the matching entry 1: No match occurred and the Index field is **UNPREDICTABLE**	R	Undefined	Required
Index	n-1:0	TLB index. Software writes this field to provide the index to the TLB entry referenced by the TLBR and TLBWI instructions. Hardware writes this field with the index of the matching TLB entry during execution of the TLBP instruction. If the TLBP fails to find a match, the contents of this field are **UNPREDICTABLE.**	R/W	Undefined	Required
0	30:n	Must be written as zero; returns zero on read.	0	0	Reserved

4.9.2 Random Register (CP0 Register 1, Select 0)

Compliance Level: *Required* for TLB-based MMUs; *Optional* otherwise.

The *Random* register is a read-only register whose value is used to index the TLB during a TLBWR instruction. The width of the Random field is calculated in the same manner as that described for the *Index* register above.

The value of the register varies between an upper and lower bound as follow:

- 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). The entry indexed by the *Wired* register is the first entry available to be written by a TLB Write Random operation.
- An upper bound is set by the total number of TLB entries minus 1.

Within the required constraints of the upper and lower bounds, the manner in which the processor selects values for the Random register is implementation-dependent. However, designers should be aware of a potential live lock condition for implementations that simply increment the Random field every 'n' cycles. With such an implementation, the TLB/XTLB refill handler can fall into synchronization with the Random field such that the same entry is used during each pass through the refill handler. If the instruction causing the TLB/XTLB refill requires more than a single entry to complete (e.g., a load instruction requiring both an instruction and a data translation), no forward progress is made and a live lock condition is created. In most cases, some other event, such as an interrupt, breaks the condition. However, if the offending instruction is executed in Kernel Mode with interrupts disabled, breaking the live lock may not be possible. Designers are encouraged to introduce some pseudo-random behavior on top of a counter implementation of the Random field, such as might be provided by, for example, an LFSR.

The processor initializes the *Random* register to the upper bound on a Reset Exception, and when the *Wired* register is written.

Figure 14 shows the format of the *Random* register; Table 60 describes the *Random* register fields.

Figure 14: Random Register Format

31 … n	n-1 … 0
0	Random

Table 60: Random Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
Random	n-1:0	TLB Random Index	R	TLB Entries - 1	Required
0	31:n	Must be written as zero; returns zero on read.	0	0	Reserved

4.9.3 EntryLo0, EntryLo1 (CP0 Registers 2 and 3, Select 0)

Compliance Level: EntryLo0 is *Required* for a TLB-based MMU; *Optional* otherwise.
Compliance Level: EntryLo1 is *Required* for a TLB-based MMU; *Optional* otherwise.

The pair of EntryLo registers act as the interface between the TLB and the TLBR, TLBWI, and TLBWR instructions. EntryLo0 holds the entries for even pages and EntryLo1 holds the entries for odd pages.

The contents of the EntryLo0 and EntryLo1 registers are not defined after an address error exception and some fields may be modified by hardware during the address error exception sequence.

Figure 15 shows the format of the EntryLo0 and EntryLo1 registers; Table 61 describes the EntryLo0 and EntryLo1 register fields.

Figure 15: Entrylo0, EntryLo1 Register Format

63 … 30	29 … 6	5 … 3	2	1	0
Fill	PFN	C	D	V	G

Table 61: EntryLo0, EntryLo1 Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
Fill	63:30	Ignored on write; returns zero on read.	R	0	Required

Table 61: EntryLo0, EntryLo1 Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
PFN	29:6	Page Frame Number. Corresponds to bits[*PABITS*-1:12] of the physical address. The width of this field implicitly limits the size of the physical address to 36 bits. If the processor implements fewer physical address bits than this limit, the unimplemented bits must be written as zero, and return zero on read. If the processor implements more physical address bits that this limit, the PFN field boundary moves to the left, compressing out bits of the Fill field.	R/W	Undefined	Required
C	5:3	Coherency attribute of the page. See Table 62 below.	R/W	Undefined	Required
D	2	"Dirty" bit, indicating that the page is writable. If this bit is a one, stores to the page are permitted. If this bit is a zero, stores to the page cause a TLB Modified exception. Kernel software may use this bit to implement paging algorithms that require knowing which pages have been written. If this bit is always zero when a page is initially mapped, the TLB Modified exception that results on any store to the page can be used to update kernel data structures that indicate that the page was actually written.	R/W	Undefined	Required
V	1	Valid bit, indicating that the TLB entry, and thus the virtual page mapping are valid. If this bit is a one, accesses to the page are permitted. If this bit is a zero, accesses to the page cause a TLB Invalid exception.	R/W	Undefined	Required
G	0	Global bit. On a TLB write, the logical AND of the G bits from both EntryLo0 and EntryLo1 becomes the G bit in the TLB entry. If the TLB entry G bit is a one, ASID comparisons are ignored during TLB matches. On a read from a TLB entry, the G bits of both EntryLo0 and EntryLo1 reflect the state of the TLB G bit.	R/W	Undefined	Required (TLB MMU)

Table 62 lists the encoding of the C field of the *EntryLo0* and *EntryLo1* registers and the K0 field of the *Config* register. An implementation may choose to implement a subset of the cache coherency attributes shown, but must implement at least encodings 2 and 3 such that software can always depend on these encodings working appropriately. In other cases, the operation of the processor is **UNDEFINED** if software specifies an unimplemented encoding.

Table 62 lists the required and optional encodings for the coherency attributes, in addition to giving an historical perspective on the encodings implemented by various MIPS processors, as obtained from the processor chip specification.

Table 62: Cache Coherency Attributes

C(5:3) Value	Cache Coherency Attributes With Historical Usage	Compliance
0	Available for implementation dependent use Historical usage: • Reserved (R4000®, VR5400, R10000®) • Unused, defaults to cached (R4300™) • Cacheable, noncoherent, write through, no write allocate (RC32364, RM5200)	Optional
1	Available for implementation dependent use Historical usage: • Reserved (R4000) • Unused, defaults to cached (R4300) • Cacheable, noncoherent, write through, write allocate (RC32364, RM5200) • Cacheable write-through, write allocate (VR5400)	Optional
2	Uncached Historical usage: • Uncached (all processors)	Required
3	Cacheable Historical usage: • Cacheable noncoherent (noncoherent) (R4000, R10000) • Cached (R4300) • Cacheable, noncoherent (writeback) (RC32364, RM5200) • Cacheable, writeback (VR5400)	Required
4	Available for implementation dependent use Historical usage: • Cacheable coherent exclusive (exclusive) (R4000, R10000) • Unused, defaults to cached (R4300) • Reserved (RC32364, RM5200, VR5400)	Optional

Table 62: Cache Coherency Attributes

C(5:3) Value	Cache Coherency Attributes With Historical Usage	Compliance
5	Available for implementation dependent use Historical usage: • Cacheable coherent exclusive on write (sharable) (R4000, R10000) • Unused, defaults to cached (R4300) • Reserved (RC32364, RM5200, VR5400)	Optional
6	Available for implementation dependent use Historical usage: • Cacheable coherent update on write (update) (R4000) • Unused, defaults to cached (R4300) • Reserved (RC32364, RM5200, R10000)	Optional
7	Available for implementation dependent use Historical usage: • Reserved (R4000) • Unused, defaults to cached (R4300) • Reserved (RC32364, RM5200) • Uncached accelerated (VR5400, R10000)	Optional

4.9.4 Context Register (CP0 Register 4, Select 0)

Compliance Level: *Required* for TLB-based MMUs; *Optional* otherwise.

The *Context* register is a read/write register containing a pointer to an entry in the page table entry (PTE) array. This array is an operating system data structure that stores virtual-to-physical translations. During a TLB miss, the operating system loads the TLB with the missing translation from the PTE array. The Context register is primarily intended for use with the TLB Refill handler, but is also loaded by hardware on an XTLB Refill and may be used by software in that handler. The *Context* register duplicates some of the information provided in the *BadVAddr* register, but is organized in such a way that the operating system can directly reference a 16-byte structure in memory that describes the mapping.

A TLB exception (TLB Refill, XTLB Refill, TLB Invalid, or TLB Modified) causes bits $VA_{31:13}$ of the virtual address to be written into the *BadVPN2* field of the *Context* register. The *PTEBase* field is written and used by the operating system.

The BadVPN2 field of the *Context* register is not defined after an address error exception and this field may be modified by hardware during the address error exception sequence.

Processor implementations must not assume that software will write the same value into the PTEBase fields of the Context and XContext registers (i.e., the PTEBase fields of the two registers may be set to different values, thus cannot share storage).

Figure 16 shows the format of the *Context* Register; Table 63 describes the *Context* register fields.

Figure 16: Context Register Formats

63 23	22 4	3 0
PTEBase	BadVPN2	0

Table 63: Context Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
PTEBase	63:23	This field is for use by the operating system and 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	R/W	Undefined	Required
BadVPN2	22:4	This field is written by hardware on a TLB exception. It contains bits $VA_{31:13}$ of the virtual address that caused the exception.	R	Undefined	Required
0	3:0	Must be written as zero; returns zero on read.	0	0	Reserved

4.9.5 PageMask Register (CP0 Register 5, Select 0)

Compliance Level: *Required* for TLB-based MMUs; *Optional* otherwise.

The *PageMask* register is a read/write register used for reading from and writing to the TLB. It holds a comparison mask that sets the variable page size for each TLB entry, as shown in Table 65. Figure 17 shows the format of the *PageMask* register; Table 64 describes the *PageMask* register fields.

Figure 17: PageMask Register Format

31 25	24 13	12 0
0	Mask	0

Table 64: PageMask Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
Mask	24:13	The Mask field is a bit mask in which a "1" bit indicates that the corresponding bit of the virtual address should not participate in the TLB match.	R/W	Undefined	Required
0	31:25, 12:0	Must be written as zero; returns zero on read.	0	0	Reserved

Table 65: Values for the Mask Field of the PageMask Register

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 MByte	0	0	1	1	1	1	1	1	1	1	1	1
16 Mbyte	1	1	1	1	1	1	1	1	1	1	1	1

It is implementation dependent how many of the encodings described in Table 65 are implemented. All processors must implement the 4KB page size (an encoding of all zeros). If a particular page size encoding is not implemented by a processor, a read of the *PageMask* register must return zeros in all bits that correspond to encodings that are not implemented. Software can determine which page sizes are supported by writing the encoding for a 16MB page to the *PageMask* register, then examine the value returned from a read of the *PageMask* register. If a pair of bits reads back as ones, the processor implements that page size. The operation of the processor is **UNDEFINED** if software loads the PageMask register with a value other than one of those listed in Table 65.

4.9.6 Wired Register (CP0 Register 6, Select 0)

Compliance Level: *Required* for TLB-based MMUs; *Optional* otherwise.

The *Wired* register is a read/write register that specifies the boundary between the wired and random entries in the TLB as shown in Figure 18. The width of the Wired field is calculated in the same manner as that described for the *Index* register above. Wired entries are fixed, non-replaceable entries which are not overwritten by a TLBWR instruction. Wired entries can be overwritten by a TLBWI instruction.

The *Wired* register is set to zero by a Reset Exception. Writing the *Wired* register causes the *Random* register to reset to its upper bound.

The operation of the processor is **UNDEFINED** if a value greater than or equal to the number of TLB entries is written to the *Wired* register.

Figure 19 shows the format of the *Wired* register; Table 66 describes the *Wired* register fields.

Figure 18: Wired And Random Entries In The TLB

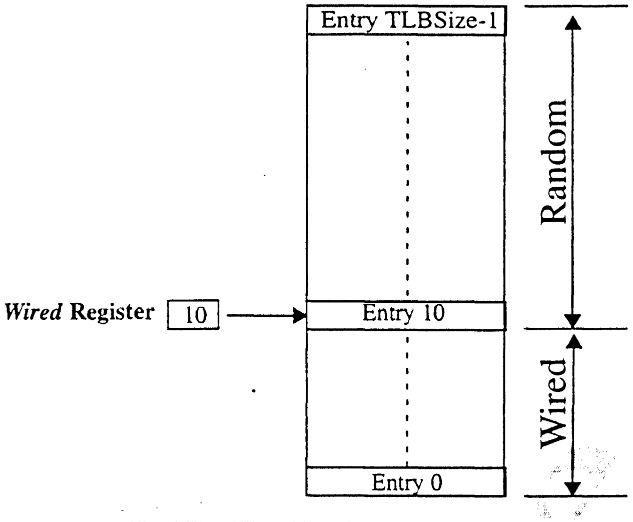


Figure 19: Wired Register

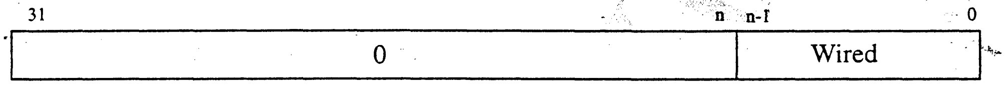


Table 66: Wired Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
Wired	n-1:0	TLB wired boundary	R/W	0	Required
0	31:n	Must be written as zero; returns zero on read.	0	0	Reserved

4.9.7 BadVAddr Register (CP0 Register 8, Select 0)

Compliance Level: *Required.*

The *BadVAddr* register is a read-only register that captures the most recent virtual address that caused one of the following exceptions:

- Address error (AdEL or AdES)
- TLB/XTLB Refill
- TLB Invalid (TLBL, TLBS)
- TLB Modified

The *BadVAddr* register does not capture address information for cache or bus errors, or for Watch exceptions, since none is an addressing error.

Figure 20 shows the format of the *BadVAddr* register; Table 67 describes the BadVAddr register fields.

Figure 20: BadVAddr Register Format

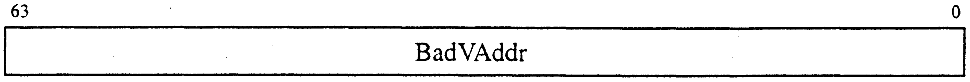


Table 67: BadVAddr Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
BadVAddr	63:0	Bad virtual address	R	Undefined	Required

4.9.8 Count Register (CP0 Register 9, Select 0)

Compliance Level: *Required.*

The Count register acts as a timer, incrementing at a constant rate, whether or not an instruction is executed, retired, or any forward progress is made through the pipeline. The rate at which the counter increments is implementation-dependent, and is a function of the frequency of the processor, not the issue width of the processor. The preferred implementation is to increment the *Count* register once per processor cycle.

It is implementation dependent whether the *Count* register continues to count or stops when the processor enters a low power mode, as might occur after executing the Wait instruction.

The Count register can be written for functional or diagnostic purposes, including at reset or to synchronize processors.

Figure 21 shows the format of the Count register; Table 68 describes the Count register fields.

Figure 21: Count Register Format

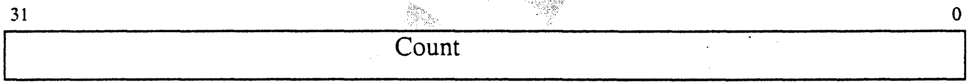


Table 68: Count Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
Count	31:0	Interval counter	R/W	Undefined	Required

4.9.9 EntryHi Register (CP0 Register 10, Select 0)

Compliance Level: *Required* for TLB-based MMU; *Optional* otherwise.

The *EntryHi* register contains the virtual address match information used for TLB read, write, and access operations.

A TLB exception (TLB Refill, XTLB Refill, TLB Invalid, or TLB Modified) causes the bits of the virtual address corresponding to the R and VPN2 fields to be written into the *EntryHi* register. The ASID field is written by software with the current address space identifier value and is used during the TLB comparison process to determine TLB match.

Software may determine the size of the virtual address space implemented by a processor by writing all ones to the EntryHi register and then reading the value back.

The VPN2 and R fields of the *EntryHi* register are not defined after an address error exception and these fields may be modified by hardware during the address error exception sequence. Software writes of the EntryHi register (via MTC0 or DMTC0) do not cause the implicit write of address-related fields in the *BadVAddr*, *Context*, or *XContext* registers.

Figure 22 shows the format of the *EntryHi* register; Table 69 describes the *EntryHi* register fields.

Figure 22: EntryHi Register Format

63 62	61 40	39 13	12 8	7 0
R	Fill	VPN2	0	ASID

Table 69: EntryHi Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
R	63:62	Virtual memory region, corresponding to $VA_{63:62}$. 00: xuseg: user address region 01: xsseg: supervisor address region. If supervisor mode is not implemented, this encoding is reserved. 10: Reserved 11: xkseg: kernel address region This field is written by hardware on a TLB exception or on a TLB read, and is written by software before a TLB write.	R/W	Undefined	Required
Fill	61:40	Fill bits reserved for expansion of the virtual address space. See below. Returns zeros on read, ignored on write.	R	0	Required
VPN2	39:13	$VA_{39:13}$ of the virtual address (virtual page number / 2). This field is written by hardware on a TLB exception or on a TLB read, and is written by software before a TLB write. The default width of this field implicitly limits the size of each virtual address space to 40 bits. If the processor implements fewer virtual address bits than this default, the Fill field must be extended to take up the unimplemented VPN2 bits. If the processor implements more virtual address bits than this default, the VPN2 field must be extended to take up some or all of the Fill bits.	R/W	Undefined	Required

Table 69: EntryHi Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
ASID	7:0	Address space identifier. This field is written by hardware on a TLB read and by software to establish the current ASID value for TLB write and against which TLB references match each entry's TLB ASID field.	R/W	Undefined	Required (TLB MMU)
0	12:8	Must be written as zero; returns zero on read.	0	0	Reserved

4.9.10 Compare Register (CP0 Register 11, Select 0)

Compliance Level: *Required.*

The *Compare* register acts in conjunction with the *Count* register to implement a timer and timer interrupt function. The *Compare* register maintains a stable value and does not change on its own.

When the value of the *Count* register equals the value of the *Compare* register, an interrupt request is combined in an implementation-dependent way with hardware interrupt 5 to set interrupt bit IP(7) in the *Cause* register. This causes an interrupt as soon as the interrupt is enabled.

For diagnostic purposes, the *Compare* register is a read/write register. In normal use however, the *Compare* register is write-only. Writing a value to the *Compare* register, as a side effect, clears the timer interrupt. Figure 23 shows the format of the *Compare* register; Table 70 describes the Compare register fields.

Figure 23: Compare Register Format

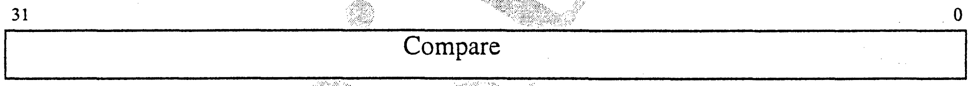


Table 70: Compare Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
Compare	31:0	Interval count compare value	R/W	Undefined	Required

4.9.11 Status Register (CP Register 12, Select 0)

Compliance Level: *Required.*

The *Status* register is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. Fields of this register combine to create operating modes for the processor. Refer to Section 4.4 on page 71 and Section 4.7 on page 87 for a discussion of operating modes and interrupt enable, respectively.

Figure 24 shows the format of the Status register; Table 71 describes the Status register fields.

Figure 24: Status Register Format

31 28	27	26	25	24	23	22	21	20	19	18	17 16	15 8	7	6	5	4	3	2	1	0
CU3-CU0	RP	FR	RE	MX	PX	BEV	TS	SR	NMI	0	Impl	IM7-IM0	KX	SX	UX	UM	R0	ERL	EXL	IE
																KSU				

Table 71: Status Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
CU (CU3.. CU0)	31:28	Controls access to coprocessors 3, 2, 1, and 0, respectively: 0: access not allowed 1: access allowed Coprocessor 0 is always usable when the processor is running in Kernel Mode or Debug Mode, independent of the state of the CU_0 bit. Execution of all floating point instructions, including those encoded with the COP1X opcode, is controlled by the CU1 enable. CU3 is not currently used by MIPS64 implementations and is reserved for future use by the Architecture. If there is no provision for connecting a coprocessor, the corresponding CU bit must be ignored on write and read as zero. However, for backward compatibility with earlier MIPS processors, CU3 may be implemented as a read/write bit, even though its state does not affect the operation of the processor.	R/W	Undefined	Required for all implemented coprocessors
RP	27	Enables reduced power mode on some implementations. The specific operation of this bit is implementation dependent. If this bit is not implemented, it must be ignored on write and read as zero. If this bit is implemented, the reset state must be zero so that the processor starts at full performance.	R/W	0	Optional

Table 71: Status Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
FR	26	Controls the floating point register mode: 0: Floating point registers can contain any 32-bit datatype. 64-bit datatypes are stored in even-odd pairs of registers. 1: Floating point registers can contain any datatype Certain combinations of the FR bit and other state or operations can cause **UNPREDICTABLE** behavior. See Section 4.5.3 on page 72 for a discussion of these combinations.	R/W	Undefined	Required
RE	25	Used to enable reverse-endian memory references while the processor is running in user mode: 0: User mode uses configured endianness 1: User mode uses reversed endianness Neither Kernel Mode nor Supervisor Mode references are affected by the state of this bit. If this bit is not implemented, it must be ignored on write and read as zero.	R/W	Undefined	Optional
MX	24	Enable access to MDMX™ resources on processors implementing MDMX. If MDMX is not implemented, the bit must be ignored on write and read as zero.	R/W	Undefined	Optional
PX	23	Enable access to 64-bit operations in User mode, without enabling 64-bit addressing: 0: 64-bit operations are not enabled 1: 64-bit operations are enabled	R/W	Undefined	Required
BEV	22	Controls the location of exception vectors: 0: Normal 1: Bootstrap See Section 4.8.2 on page 91 for details.	R/W	1	Required

Table 71: Status Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
TS	21	Indicates that the TLB has detected a match on multiple entries. It is implementation dependent whether this detection occurs at all, on a write to the TLB, or an access to the TLB. When such a detection occurs, the processor initiates a machine check exception and sets this bit. It is implementation dependent whether this condition can be corrected by software. If the condition can be corrected, this bit should be cleared before resuming normal operation. If this bit is not implemented, it must be ignored on write and read as zero. Software writes to this bit may not cause a 0-to-1 transition. Hardware may ignore software attempts to cause such a transition.	R/W	0	Required if TLB Shutdown is implemented
SR	20	Indicates that the entry through the reset exception vector was due to a Soft Reset: 0: Not Soft Reset (NMI or Reset) 1: Soft Reset	R/W	1 for Soft Reset; 0 otherwise	Required if Soft Reset is implemented
NMI	19	Indicates that the entry through the reset exception vector was due to an NMI 0: Not NMI (Soft Reset or Reset) 1: NMI	R/W	1 for NMI; 0 otherwise	Required if NMI is implemented
0	18	Must be written as zero; returns zero on read.	0	0	Reserved
Reserved for Implementations	17:16	These bits are implementation dependent and not defined by the architecture. If they are not implemented, they must be ignored on write and read as zero.		Undefined	Optional
IM	15:8	Interrupt Mask: Controls the enabling of each of the external, internal and software interrupts. Refer to Section 4.7 on page 87 for a complete discussion of enabled interrupts. 0: interrupt request disabled 1: interrupt request enabled	R/W	Undefined	Required

Table 71: Status Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
KX	7	Enables the following behavior: • Access to 64-bit Kernel Segments • Use of the XTLB Refill Vector for references to Kernel Segments 0: Access to 64-bit Kernel Segments disabled, TLB Refill Vector used for references to Kernel Segments 1: Access to 64-bit Kernel Segments enabled, XTLB Refill Vector used for references to Kernel Segments If 64-bit addressing is not implemented, this bit must be ignored on write and read as zero.	R/W	Undefined	Required for 64-bit Addressing
SX	6	If Supervisor Mode is implemented, enables the following behavior: • Access to 64-bit Supervisor Segments • Use of the XTLB Refill Vector for references to Supervisor Segments 0: Access to 64-bit Supervisor Segments disabled, TLB Refill Vector used for references to Supervisor Segments 1: Access to 64-bit Supervisor Segments enabled, XTLB Refill Vector used for references to Supervisor Segments If Supervisor Mode is not implemented, it is implementation dependent whether access to what would normally be 64-bit supervisor address space is enabled with the SX or KX bit. If 64-bit addressing is not implemented, this bit must be ignored on write and read as zero.	R/W	Undefined	Required if both Supervisor Mode and 64-bit addressing are implemented

Table 71: Status Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
UX	5	Enables the following behavior: • Access to 64-bit User Segments • Use of the XTLB Refill Vector for references to User Segments • Execution of instructions which perform 64-bit operations while the processor is operating in User Mode 0: Access to 64-bit User Segments disabled, TLB Refill Vector used for references to User Segments, execution of instructions which perform 64-bit operations is disallowed while the processor is running in User Mode 1: Access to 64-bit User Segments enabled, XTLB Refill Vector used for references to User Segments, execution of instructions which perform 64-bit operations is allowed while the processor is running in User Mode If 64-bit addressing is not implemented, this bit must be ignored on write and read as zero.	R/W	Undefined	Required for 64-bit Addressing
KSU	4:3	If Supervisor Mode is implemented, the encoding of this field denotes the base operating mode of the processor. See Section 4.4 for a full discussion of operating modes. The encoding of this field is: 00_2: Base mode is Kernel Mode 01_2: Base mode is Supervisor Mode 10_2: Base mode is User Mode 11_2: Reserved. The operation of the processor is **UNDEFINED** if this value is written to the KSU field Note: This field overlaps the UM and R0 fields, described below.	R/W	Undefined	Required if Supervisor Mode is implemented; Optional otherwise

Table 71: Status Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
UM	4	If Supervisor Mode is not implemented, this bit denotes the base operating mode of the processor. See Section 4.4 on page 71 for a full discussion of operating modes. The encoding of this bit is: 0: Base mode is Kernel Mode 1: Base mode is User Mode Note: This bit overlaps the KSU field, described above.	R/W	Undefined	Required
R0	3	If Supervisor Mode is not implemented, this bit is reserved. This bit must be ignored on write and read as zero. Note: This bit overlaps the KSU field, described above.	R	0	Reserved
ERL	2	Error Level; Set by the processor when a Reset, Soft Reset, NMI or Cache Error exception are taken. 0: normal level 1: error level When ERL is set: • The processor is running in kernel mode • Interrupts are disabled • The ERET instruction will use the return address held in ErrorEPC instead of EPC • The lower 2^{29} bytes of kuseg are treated as an unmapped and uncached region. See Section 4.6.7 on page 83. This allows main memory to be accessed in the presence of cache errors. The operation of the processor is **UNDEFINED** if the ERL bit is set while the processor is executing instructions from kuseg.	R/W	1	Required

Table 71: Status Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
EXL	1	Exception Level; Set by the processor when any exception other than Reset, Soft Reset, NMI or Cache Error exception are taken. 0: normal level 1: exception level When EXL is set: • The processor is running in Kernel Mode • Interrupts are disabled. • TLB/XTLB Refill exceptions will use the general exception vector instead of the TLB/XTLB Refill vectors. • EPC and Cause$_{BD}$ will not be updated if another exception is taken	R/W	Undefined	Required
IE	0	Interrupt Enable: Acts as the master enable for software and hardware interrupts: 0: disable interrupts 1: enables interrupts	R/W	Undefined	Required

4.9.12 Cause Register (CP0 Register 13, Select 0)

Compliance Level: *Required.*

The *Cause* register primarily describes the cause of the most recent exception. In addition, fields also control software interrupt requests and the vector through which interrupts are dispatched. With the exception of the IP[1:0], IV, and WP fields, all fields in the Cause register are read-only.

Figure 25 shows the format of the Cause register; Table 72 describes the Cause register fields.

Figure 25: Cause Register Format

31	30	29 28	27 24	23	22	21 16	15 8	7	6 2	1 0
BD	0	CE	0	IV	WP	0	IP7:IP0	0	Exc Code	0

Table 72: Cause Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
BD	31	Indicates whether the last exception taken occurred in a branch delay slot: 0: Not in delay slot 1: In delay slot The processor updates BD only if Status$_{EXL}$ was zero when the exception occurred.	R	Undefined	Required
CE	29:28	Coprocessor unit number referenced when a Coprocessor Unusable exception is taken. This field is loaded by hardware on every exception, but is **UNPREDICTABLE** for all exceptions except for Coprocessor Unusable.	R	Undefined	Required
IV	23	Indicates whether an interrupt exception uses the general exception vector or a special interrupt vector: 0: Use the general exception vector (0x180) 1: Use the special interrupt vector (0x200)	R/W	Undefined	Required
WP	22	Indicates that a watch exception was deferred because Status$_{EXL}$ or Status$_{ERL}$ were a one at the time the watch exception was detected. This bit both indicates that the watch exception was deferred, and causes the exception to be initiated once Status$_{EXL}$ and Status$_{ERL}$ are both zero. As such, software must clear this bit as part of the watch exception handler to prevent a watch exception loop. Software writes to this bit may not cause a 0-to-1 transition. Hardware may ignore software attempts to cause such a transition. If watch registers are not implemented, this bit must be ignored on write and read as zero.	R/W	Undefined	Required if watch registers are implemented

Table 72: Cause Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
IP[7:2]	15:10	Indicates an external interrupt is pending: 15: Hardware interrupt 5, timer or performance counter interrupt 14: Hardware interrupt 4 13: Hardware interrupt 3 12: Hardware interrupt 2 11: Hardware interrupt 1 10: Hardware interrupt 0	R	Undefined	Required
IP[1:0]	9:8	Controls the request for software interrupts: 9: Request software interrupt 1 8: Request software interrupt 0	R/W	Undefined	Required
ExcCode	6:2	Exception code - see Table 73	R	Undefined	Required
0	30, 27:24, 21:16, 7, 1:0	Must be written as zero; returns zero on read.	0	0	Reserved

Table 73: Cause Register ExcCode Field

Exception Code Value		Mnemonic	Description
Decimal	Hexidecimal		
0	0x00	Int	Interrupt
1	0x01	Mod	TLB modification exception
2	0x02	TLBL	TLB exception (load or instruction fetch)
3	0x03	TLBS	TLB exception (store)
4	0x04	AdEL	Address error exception (load or instruction fetch)
5	0x05	AdES	Address error exception (store)
6	0x06	IBE	Bus error exception (instruction fetch)
7	0x07	DBE	Bus error exception (data reference: load or store)
8	0x08	Sys	Syscall exception
9	0x09	Bp	Breakpoint exception
10	0x0a	RI	Reserved instruction exception
11	0x0b	CpU	Coprocessor Unusable exception
12	0x0c	Ov	Arithmetic Overflow exception

Table 73: Cause Register ExcCode Field

Exception Code Value			
Decimal	**Hexidecimal**	**Mnemonic**	**Description**
13	0x0d	Tr	Trap exception
14	0x0e	-	Reserved
15	0x0f	FPE	Floating point exception
16-17	0x10-0x11	-	Available for implementation dependent use
18	0x12	C2E	Reserved for precise Coprocessor 2 exceptions
19-22	0x13-0x16	-	Reserved
23	0x17	WATCH	Reference to WatchHi/WatchLo address
24	0x18	MCheck	Machine check
25-29	0x19-0x1d	-	Reserved
30	0x1e	CacheErr	Cache error. In normal mode, a cache error exception has a dedicated vector and the Cause register is not updated. If EJTAG is implemented and a cache error occurs while in Debug Mode, this code is used to indicate that re-entry to Debug Mode was caused by a cache error.
31	0x1f	VCED	Virtual Coherency Exception Data. This exception code was used on the R4000 and is listed here only for historical perspective.

4.9.13 Exception Program Counter (CP0 Register 14, Select 0)

Compliance Level: *Required.*

The *Exception Program Counter (EPC)* is a read/write register that contains the address at which processing resumes after an exception has been serviced. All bits of the *EPC* register are significant and must be writable.

For synchronous (precise) exceptions, *EPC* 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 exception causing instruction is in a branch delay slot, and the *Branch Delay* bit in the *Cause* register is set.

For asynchronous (imprecise) exceptions, *EPC* contains the address of the instruction at which to resume execution.

The processor does not write to the *EPC* register when the EXL bit in the *Status* register is set to one.

Figure 26 shows the format of the *EPC* register; Table 74 describes the *EPC* register fields.

Figure 26: EPC Register Format

63 0
EPC

Table 74: EPC Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
EPC	63:0	Exception Program Counter	R/W	Undefined	Required

4.9.14 Processor Identification (CP0 Register 15, Select 0)

Compliance Level: *Required.*

The *Processor Identification (PRId)* register is a 32 bit read-only register that contains information identifying the manufacturer, manufacturer options, processor identification and revision level of the processor. Figure 27 shows the format of the *PRId* register; Table 75 describes the *PRId* register fields.

Figure 27: PRId Register Format

31 24	23 16	15 8	7 0
Company Options	CompanyID	ProcessorID	Revision

Table 75: PRId Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
Company Options	31:24	Available to the designer or manufacturer of the processor for company-dependent options. The value in this field is not specified by the architecture. If this field is not implemented, it must read as zero.	R	Preset	Optional

Table 75: PRId Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	**Bits**				
Company ID	23:16	Identifies the company that designed or manufactured the processor. Software can distinguish a MIPS32 or MIPS64 processor from one implementing an earlier MIPS ISA by checking this field for zero. If it is non-zero the processor implements the MIPS32 or MIPS64 Architecture. Company IDs are assigned by MIPS Technologies when a MIPS32 or MIPS64 license is acquired. The encodings in this field are: 0: Not a MIPS32 or MIPS64 processor 1: MIPS Technologies, Inc. 2-255: Contact MIPS Technologies, Inc. for the list of CompanyID assignments	R	Preset	Required
Processor ID	15:8	Identifies the type of processor. This field allows software to distinguish between various processor implementations within a single company, and is qualified by the CompanyID field, described above. The combination of the CompanyID and ProcessorID fields creates a unique number assigned to each processor implementation.	R	Preset	Required
Revision	7:0	Specifies the revision number of the processor. This field allows software to distinguish between one revision and another of the same processor type. If this field is not implemented, it must read as zero.	R	Preset	Optional

4.9.15 Configuration Register (CP0 Register 16, Select 0)

Compliance Level: *Required.*

The *Config* register specifies various configuration and capabilities information. Most of the fields in the *Config* register are initialized by hardware during the Reset Exception process, or are constant. One field, K0, must be initialized by software in the reset exception handler.

Figure 28 shows the format of the *Config* register; Table 76 describes the *Config* register fields.

Figure 28: Config Register Format

31	30 16	15	14 13	12 10	9 7	6 3	2 0
M	Reserved for Implementations	BE	AT	AR	MT	0	K0

Table 76: Config Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
M	31	Denotes that the Config1 register is implemented at a select field value of 1.	R	Preset	Required
	30:16	This field is reserved for implementations. Refer to the processor specification for the format and definition of this field		Undefined	Optional
BE	15	Indicates the endian mode in which the processor is running: 0: Little endian 1: Big endian	R	Preset or Externally Set	Required
AT	14:13	Architecture type implemented by the processor: 0: MIPS32 1: MIPS64 with 32-bit addresses only 2: MIPS64 with 32/64-bit addresses 3: Reserved	R	Preset	Required
AR	12:10	Architecture revision level: 0: Revision 1 1-7: Reserved	R	Preset	Required
MT	9:7	MMU Type: 0: None 1: Standard TLB 2: Standard BAT (see Appendix A) 3: Standard fixed mapping (see Appendix A) 4: Reserved 5: Reserved 6: Reserved 7: Reserved	R	Preset	Required
K0	2:0	Kseg0 coherency algorithm. See Table 62 for the encoding of this field.	R/W	Undefined[a]	Optional
0	6:3	Must be written as zero; returns zero on read.	0	0	Reserved

[a] It is strongly recommended that the K0 field be initialized by hardware to a value that would allow the processor to operate correctly even if software references kseg0 before initializing this value. The suggested value is the uncached encoding of 2. Some operating systems have been seen to reference kseg0 before initializing the K0 field, causing processors who do not initialize the K0 field at reset to hang during boot. While this is certainly a software error, having to debug such errors during boot of a new processor may easily justify the minimal hardware necessary to initialize the K0 field at reset.

4.9.16 Configuration Register 1 (CP0 Register 16, Select 1)

Compliance Level: *Required.*

The *Config1* register is an adjunct to the *Config* register and encodes additional capabilities information. All fields in the *Config1* register are read-only.

The Icache and Dcache configuration parameters include encodings for the number of sets per way, the line size, and the associativity. The total cache size for a cache is therefore:

Associativity * Line Size * Sets Per Way

If the line size is zero, there is no cache implemented.

Figure 29 shows the format of the *Config1* register; Table 77 describes the *Config1* register fields.

Figure 29: Config1 Register Format

31	30 25	24 22	21 19	18 16	15 13	12 10	9 7	6 5	4	3	2	1	0
0	MMU Size - 1	IS	IL	IA	DS	DL	DA	0	PC	WR	CA	EP	FP

Table 77: Config1 Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
0	31	This bit is reserved to indicate that a Config2 register is present. With this revision of the architecture, writes to this bit must be ignored, and it must read as zero.	R	Preset	Required
MMU Size - 1	30:25	Number of entries in the TLB minus one. The values 0 through 63 is this field correspond to 1 to 64 TLB entries. The value zero is implied by $\text{Config}_{\text{MT}}$ having a value of 'none'.	R	Preset	Required
IS	24:22	Icache sets per way: 0: 64 1: 128 2: 256 3: 512 4: 1024 5: 2048 6: 4096 7: Reserved	R	Preset	Required
IL	21:19	Icache line size: 0: No Icache present 1: 4 bytes 2: 8 bytes 3: 16 bytes 4: 32 bytes 5: 64 bytes 6: 128 bytes 7: Reserved	R	Preset	Required

Table 77: Config1 Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
IA	18:16	Icache associativity: 0: Direct mapped 1: 2-way 2: 3-way 3: 4-way 4: 5-way 5: 6-way 6: 7-way 7: 8-way	R	Preset	Required
DS	15:13	Dcache sets per way: 0: 64 1: 128 2: 256 3: 512 4: 1024 5: 2048 6: 4096 7: Reserved	R	Preset	Required
DL	12:10	Dcache line size: 0: No Dcache present 1: 4 bytes 2: 8 bytes 3: 16 bytes 4: 32 bytes 5: 64 bytes 6: 128 bytes 7: Reserved	R	Preset	Required
DA	9:7	Dcache associativity: 0: Direct mapped 1: 2-way 2: 3-way 3: 4-way 4: 5-way 5: 6-way 6: 7-way 7: 8-way	R	Preset	Required
PC	4	Performance Counter registers implemented: 0: No performance counter registers implemented 1: At least one performance counter register implemented	R	Preset	Required

Table 77: Config1 Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
WR	3	Watch registers implemented: 0: No watch registers implemented 1: At least one watch register implemented	R	Preset	Required
CA	2	Code compression (MIPS16) implemented: 0: No code compression 1: Code compression	R	Preset	Required
EP	1	EJTAG implemented: 0: No EJTAG implemented 1: EJTAG implemented	R	Preset	Reserved
FP	0	FPU implemented: 0: No FPU 1: FPU	R	Preset	Required
0	6:5	Must be written as zero; returns zero on read.	0	0	Reserved

4.9.17 Load Linked Address (CP0 Register 17, Select 0)

Compliance Level: *Optional.*

The *LLAddr* register contains relevant bits of the physical address read by the most recent Load Linked instruction. This register is implementation dependent and for diagnostic purposes only and serves no function during normal operation.

Figure 30 shows the format of the *LLAddr* register; Table 78 describes the *LLAddr* register fields.

Figure 30: LLAddr Register Format

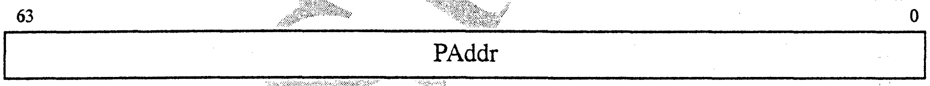


Table 78: LLAddr Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
PAddr	63:0	This field encodes the physical address read by the most recent Load Linked instruction. The format of this register is implementation-dependent, and an implementation may implement as many of the bits or format the address in any way that it finds convenient.	R	Undefined	Optional

4.9.18 WatchLo Register (CP0 Register 18)

Compliance Level: *Optional.*

The *WatchLo* and *WatchHi* registers together provide the interface to a watchpoint debug facility which initiates a watch exception if an instruction or data access matches the address specified in the registers. As such, they duplicate some functions of the EJTAG debug solution. Watch exceptions are taken only if the EXL and ERL bits are zero in the *Status* register. If either bit is a one, the WP bit is set in the *Cause* register, and the watch exception is deferred until both the EXL and ERL bits are zero.

An implementation may provide zero or more pairs of WatchLo and WatchHi registers, referencing them via the select field of the MTC0/MFC0 and DMTC0/DMFC0 instructions, and each pair of Watch registers may be dedicated to a particular type of reference (e.g., instruction or data). Software may determine if at least one pair of *WatchLo* and *WatchHi* registers are implemented via the WR bit of the *Config1* register. See the discussion of the M bit in the *WatchHi* register description below.

The *WatchLo* register specifies the base virtual address and the type of reference (instruction fetch, load, store) to match. If a particular Watch register only supports a subset of the reference types, the unimplemented enables must be ignored on write and return zero on read. Software may determine which enables are supported by a particular Watch register pair by setting all three enables bits and reading them back to see which ones were actually set.

It is implementation dependent whether a data watch is triggered by a prefetch or a cache instruction whose address matches the Watch register address match conditions. The preferred implementation is not to match on these instructions.

Figure 31 shows the format of the *WatchLo* register; Table 79 describes the *WatchLo* register fields.

Figure 31: WatchLo Register Format

63 … 3	2	1	0
VAddr	I	R	W

Table 79: WatchLo Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
VAddr	63:3	This field specifies the virtual address to match. Note that this is a doubleword address, since bits [2:0] are used to control the type of match.	R/W	Undefined	Required
I	2	If this bit is one, watch exceptions are enabled for instruction fetches that match the address and are actually issued by the processor (speculative instructions never cause Watch exceptions). If this bit is not implemented, writes to it must be ignored, and reads must return zero.	R/W	0	Optional
R	1	If this bit is one, watch exceptions are enabled for loads that match the address. If this bit is not implemented, writes to it must be ignored, and reads must return zero.	R/W	0	Optional

Table 79: WatchLo Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
W	0	If this bit is one, watch exceptions are enabled for stores that match the address. If this bit is not implemented, writes to it must be ignored, and reads must return zero.	R/W	0	Optional

4.9.19 WatchHi Register (CP0 Register 19)

Compliance Level: *Optional.*

The *WatchLo* and *WatchHi* registers together provide the interface to a watchpoint debug facility which initiates a watch exception if an instruction or data access matches the address specified in the registers. As such, they duplicate some functions of the EJTAG debug solution. Watch exceptions are taken only if the EXL and ERL bits are zero in the *Status* register. If either bit is a one, the WP bit is set in the *Cause* register, and the watch exception is deferred until both the EXL and ERL bits are zero.

An implementation may provide zero or more pairs of WatchLo and WatchHi registers, referencing them via the select field of the MTC0/MFC0 and DMTC0/DMFC0 instructions, and each pair of Watch registers may be dedicated to a particular type of reference (e.g., instruction or data). Software may determine if at least one pair of *WatchLo* and *WatchHi* registers are implemented via the WR bit of the *Config1* register. If the M bit is one in the *WatchHi* register reference with a select field of '*n*', another WatchHi/WatchLo pair are implemented with a select field of '*n+1*'.

The *WatchHi* register contains information that qualifies the virtual address specified in the *WatchLo* register: an ASID, a G(lobal) bit, and an optional address mask. If the G bit is one, any virtual address reference that matches the specified address will cause a watch exception. If the G bit is a zero, only those virtual address references for which the ASID value in the *WatchHi* register matches the ASID value in the *EntryHi* register cause a watch exception. The optional mask field provides address masking to qualify the address specified in *WatchLo*.

Figure 32 shows the format of the *WatchHi* register; Table 80 describes the *WatchHi* register fields.

Figure 32: WatchHi Register Format

31	30	29 24	23 16	15 12	11 3	2 0
M	G	0	ASID	0	MASK	0

Table 80: WatchHi Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
M	31	If this bit is one, another pair of *WatchHi/ WatchLo* registers is implemented at a MTC0 or MFC0 select field value of '*n+1*'	R	Preset	Required

Table 80: WatchHi Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
G	30	If this bit is one, any address that matches that specified in the *WatchLo* register will cause a watch exception. If this bit is zero, the ASID field of the *WatchHi* register must match the ASID field of the *EntryHi* register to cause a watch exception.	R/W	Undefined	Required
ASID	23:16	ASID value which is required to match that in the *EntryHi* register if the G bit is zero in the *WatchHi* register.	R/W	Undefined	Required
Mask	11:3	Optional bit mask that qualifies the address in the *WatchLo* register. If this field is implemented, any bit in this field that is a one inhibits the corresponding address bit from participating in the address match. If this field is not implemented, writes to it must be ignored, and reads must return zero. Software may determine how many mask bits are implemented by writing ones the this field and then reading back the result.	R/W	Undefined	Optional
0	29:24, 15:12, 2:0	Must be written as zero; returns zero on read.	0	0	Reserved

4.9.20 XContext Register (CP0 Register 20, Select 0)

Compliance Level: *Required* for 64-bit TLB-based MMUs. *Optional* otherwise.

The X*Context* register is a read/write register containing a pointer to an entry in the page table entry (PTE) array. This array is an operating system data structure that stores virtual-to-physical translations. During a TLB miss, the operating system loads the TLB with the missing translation from the PTE array. The *XContext* register is primarily intended for use with the XTLB Refill handler, but is also loaded by hardware on a TLB Refill. However, it is unlikely to be useful to software in the TLB Refill Handler. The X*Context* register duplicates some of the information provided in the *BadVAddr* register, but is organized in such a way that the operating system can directly reference a 16-byte structure in memory that describes the mapping.

A TLB exception (TLB Refill, XTLB Refill, TLB Invalid, or TLB Modified) causes bits 63:62 of the virtual address to be written into the R field and bits *SEGBITS*-1:13 of the virtual address to be written into the *BadVPN2* field of the X*Context* register. The *PTEBase* field is written and used by the operating system.

The BadVPN2 field of the *Context* register is not defined after an address error exception and this field may be modified by hardware during the address error exception sequence.

Processor implementations must not assume that software will write the same value into the PTEBase fields of the Context and XContext registers (i.e., the PTEBase fields of the two registers may be set to different values, thus cannot share storage).

Figure 33: shows the format of the *XContext* register; Table 81: describes the *XContext* register fields. In Figure 33,

bit numbers above the figure use the symbol *SEGBITS*; bit number under the figure assume that *SEGBITS* has the value 40.

Figure 33: XContext Register Format

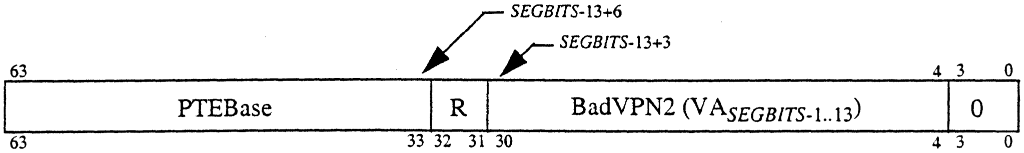


Table 81: XContext Register Fields

Field		Description	Read/ Write	Reset State	Compliance
Name	Bits				
PTEBase	63 : *SEGBITS*-13+6 (63:33 assuming *SEGBITS* is 40)	This field is for use by the operating system and 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	R/W	Undefined	Required
R	*SEGBITS*-13+5 : *SEGBITS*-13+4 (32:31 assuming *SEGBITS* is 40)	The *Region* field contains bits 63:62 of the virtual address. 00 = xuseg 01: xsseg: supervisor address region. If supervisor mode is not implemented, this encoding is reserved. 10 = Reserved 11 = xkseg	R	Undefined	Required
BadVPN2	*SEGBITS*-13+3 : 4 (30:4 assuming *SEGBITS* is 40)	The *Bad Virtual Page Number/2* field is written by hardware on a miss. It contains bits VA$_{SEGBITS-1:13}$ of the virtual address that missed.	R	Undefined	Required
0	3:0	Must be written as zero; returns zero on read.	0	0	Reserved

4.9.21 Reserved for Implementations (CP0 Register 22, all Select values)

Compliance Level: *Optional: Implementation Dependent.*

CP0 register 22 is reserved for implementation dependent use and is not defined by the architecture.

4.9.22 Debug Register (CP0 Register 23)

Compliance Level: *Optional.*

The *Debug* register is part of the EJTAG specification. Refer to that specification for the format and description of this register.

4.9.23 DEPC Register (CP0 Register 24)

Compliance Level: *Optional.*

The *DEPC* register is part of the EJTAG specification. Refer to that specification for the format and description of this register.

All bits of the *DEPC* register are significant and must be writable.

4.9.24 Performance Counter Register (CP0 Register 25)

Compliance Level: *Recommended.*

The MIPS64 Architecture supports implementation dependent performance counters that provide the capability to count events or cycles for use in performance analysis. If performance counters are implemented, each performance counter consists of a pair of registers: a 32-bit control register and a 32-bit counter register. To provide additional capability, multiple performance counters may be implemented.

Performance counters can be configured to count implementation dependent events or cycles under a specified set of conditions that are determined by the control register for the performance counter. The counter register increments once for each enabled event. When bit 31 of the counter register is a one (the counter overflows), the performance counter optionally requests an interrupt that is combined in an implementation dependent way with hardware interrupt 5 to set interrupt bit IP(7) in the *Cause* register. Counting continues after a counter register overflow whether or not an interrupt is requested or taken.

Each performance counter is mapped into even-odd select values of the *PerfCnt* register: Even selects access the control register and odd selects access the counter register. Table 82 shows an example of two performance counters and how they map into the select values of the *PerfCnt* register.

Table 82: Example Performance Counter Usage of the PerfCnt CP0 Register

Performance Counter	PerfCnt Register Select Value	PerfCnt Register Usage
0	PerfCnt, Select 0	Control Register 0
	PerfCnt, Select 1	Counter Register 0
1	PerfCnt, Select 2	Control Register 1
	PerfCnt, Select 3	Counter Register 1

More or less than two performance counters are also possible, extending the select field in an obvious way to obtain the desired number of performance counters. Software may determine if at least one pair of Performance Counter Control and Counter registers is implemented via the PC bit in the Config1 register. If the M bit is one in the Performance Counter Control register referenced via a select field of '*n*', another pair of Performance Counter Control and Counter registers is implemented at the select values of '*n+2*' and '*n+3*'.

The Control Register associated with each performance counter controls the behavior of the performance counter. Figure 34 shows the format of the Performance Counter Control Register; Table 83 describes the Performance Counter Control Register fields.

Figure 34: Performance Counter Control Register Format

31	30 11	10 5	4	3	2	1	0
M	0	Event	IE	U	S	K	EXL

Table 83: Performance Counter Control Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
M	31	If this bit is a one, another pair of Performance Counter Control and Counter registers is implemented at a MTC0 or MFC0 select field value of '*n+2*' and '*n+3*'.	R	Preset	Required
0	30:11	Must be written as zero; returns zero on read	0	0	Reserved
Event	10:5	Selects the event to be counted by the corresponding Counter Register. The list of events is implementation dependent, but typical events include cycles, instructions, memory reference instructions, branch instructions, cache and TLB misses, etc. If an implementation does not support all possible encodings of this field, it is implementation dependent how the unimplemented encodings are interpreted. The preferred implementation is to treat them as null events that enable no counts. Implementations that support multiple performance counters allow ratios of events, e.g., cache miss ratios if cache miss and memory references are selected as the events in two counters	R/W	Undefined	Required
IE	4	Interrupt Enable. Enables the interrupt request when the corresponding counter overflows (bit 31 of the counter is one). Note that this bit simply enables the interrupt request. The actual interrupt is still gated by the normal interrupt masks and enable in the *Status* register. 0: Performance counter interrupt disabled 1: Performance counter interrupt enabled	R/W	0	Required

Table 83: Performance Counter Control Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
U	3	Enables event counting in User Mode. Refer to Section 4.4.4 on page 72 for the conditions under which the processor is operating in User Mode. 0: Disable event counting in User Mode 1: Enable event counting in User Mode	R/W	Undefined	Required
S	2	Enables event counting in Supervisor Mode (for those processors that implement Supervisor Mode). Refer to Section 4.4.3 on page 72 for the conditions under which the processor is operating in Supervisor mode. If the processor does not implement Supervisor Mode, this bit must be ignored on write and return zero on read. 0: Disable event counting in Supervisor Mode 1: Enable event counting in Supervisor Mode	R/W	Undefined	Required
K	1	Enables event counting in Kernel Mode. Unlike the usual definition of Kernel Mode as described in Section 4.4.2 on page 71, this bit enables event counting only when the EXL and ERL bits in the *Status* register are zero. 0: Disable event counting in Kernel Mode 1: Enable event counting in Kernel Mode	R/W	Undefined	Required
EXL	0	Enables event counting when the EXL bit in the *Status* register is one and the ERL bit in the *Status* register is zero. 0: Disable event counting while EXL = 1, ERL = 0 1: Enable event counting while EXL = 1, ERL = 0 Counting is never enabled when the ERL bit in the *Status* register is one.	R/W	Undefined	Required

The Counter Register associated with each performance counter increments once for each enabled event. Figure 35 shows the format of the Performance Counter Counter Register; Table 84 describes the Performance Counter Counter Register fields.

Figure 35: Performance Counter Counter Register Format

<table>
<tr><td>31</td><td align="right">0</td></tr>
<tr><td colspan="2" align="center">Event Count</td></tr>
</table>

Table 84: Performance Counter Counter Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
Event Count	31:0	Increments once for each event that is enabled by the corresponding Control Register. When bit 31 is one, an interrupt request is made if the IE bit in the Control Register is one.	R/W	Undefined	Required

4.9.25 ErrCtl Register (CP0 Register 26, Select 0)

Compliance Level: *Optional.*

The *ErrCtl* register provides an implementation dependent diagnostic interface with the error detection mechanisms implemented by the processor. This register has been used in previous implementations to read and write parity or ECC information to and from the primary or secondary cache data arrays in conjunction with specific encodings of the Cache instruction or other implementation-dependent method. *The exact format of the ErrCtl register is implementation dependent and not specified by the architecture.* Refer to the processor specification for the format of this register and a description of the fields.

4.9.26 CacheErr Register (CP0 Register 27, Select 0)

Compliance Level: *Optional.*

The CacheErr register provides an interface with the cache error detection logic that may be implemented by a processor.

The exact format and operation of the CacheErr register is implementation dependent. The description below is an example of a format that is similar to previous implementations. Caches with substantially different sizes, organizations, and error correction/detection properties may require a different format from that shown below.

Figure 36 shows the example format of the *CacheErr* register; Table 85 describes the *CacheErr* register fields.

The *ErrorEPC* register is a read-write register, similar to the *EPC* register, except that *ErrorEPC* is used on error exceptions. All bits of the *ErrorEPC* register are significant and must be writable. It is also used to store the program counter on Reset, Soft Reset, Nonmaskable Interrupt (NMI), and Cache Error exceptions.

The *ErrorEPC* register contains the virtual address at which instruction processing can resume after servicing an error. *ErrorEPC* 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 error causing instruction is in a branch delay slot.

Unlike the *EPC* register, there is no corresponding branch delay slot indication for the *ErrorEPC* register.

Figure 41 shows the format of the *ErrorEPC* register; Table 90 describes the *ErrorEPC* register fields.

Figure 41: ErrorEPC Register Format

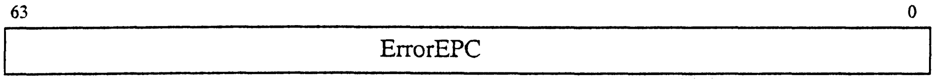


Table 90: ErrorEPC Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
ErrorEPC	63:0	Error Exception Program Counter	R/W	Undefined	Required

4.9.32 DESAVE Register (CP0 Register 31)

Compliance Level: *Optional.*

The *DESAVE* register is part of the EJTAG specification. Refer to that specification for the format and description of this register.

4.10 CP0 Hazards

Because resources controlled via Coprocessor 0 affect the operation of various pipeline stages of a MIPS64 processor, manipulation of these resources may produce results that are not detectable by subsequent instructions for some number of execution cycles. When no hardware interlock exists between one instruction that causes an effect that is visible to a second instruction, a *CP0 hazard* exists. Some MIPS implementations have placed the entire burden on the kernel programmer to pad the instruction stream in such a way that the second instruction is spaced far enough from the first that the effects of the first are seen by the second. Other MIPS implementations have added full hardware interlocks such that the kernel programmer need not pad. The trade-off is between kernel software changes for each new processor vs. more complex hardware interlocks required in the processor.

The MIPS64 Architecture does not dictate the solution that is required for a compatible implementation. The choice of implementation ranges from full hardware interlocks to full dependence on software padding, to some combination of the two. For an implementation choice that relies on software padding, Table 91 lists the "typical" spacing required to allow the consumer to eliminate the hazard. The "typical" values shown in this table represent spacing that is in common use by operating systems today. An implementation which requires less spacing to clear the hazard (including one which has full hardware interlocking) should operate correctly with and operating system which uses this hazard table. An implementation which requires more spacing to clear the hazard incurs the burden of validating kernel code against the new hazard requirements.

Note that, for superscalar MIPS implementations, the number of instructions issued per cycle may be greater than *one, and thus that the duration of the hazard in instructions may be greater than the duration in cycles. It is for this reason that MIPS64 defines the SSNOP instruction to convert instruction issues to cycles in a superscalar design.*

Table 91: "Typical" CP0 Hazard Spacing

Producer	→	Consumer	Hazard On	"Typical" Spacing (Cycles)
TLBWR, TLBWI	→	TLBP, TLBR	TLB entry	3
		Load/store using new TLB entry	TLB entry	3
		Instruction fetch using new TLB entry	TLB entry	5
MTCO Status[CU]	→	Coprocessor instruction needs CU set	Status[CU]	4
MTC0 Status	→	ERET	Status	3
MTC0 Status[IE]	→	Interrupted Instruction	Status[IE]	3
TLBR	→	MFC0 EntryHi MFC0 PageMask	EntryHi, PageMask	3
MTC0 EntryLo0 MTC0 EntryLo1 MTC0 Entry Hi MTC0 PageMask MTC0 Index	→	TLBP TLBR TLBWI TLBWR	EntryLo0 EntryLo1 EntryHi PageMask Index	2
TLBP	→	MFC0 Index	Index	2
MTC0 EPC	→	ERET	EPC	2

Appendix A Alternative MMU Organizations

The main body of this specification describes the TLB-based MMU organization. This appendix describes other potential MMU organizations.

A.1 Fixed Mapping MMU

As an alternative to the full TLB-based MMU, the MIPS64 Architecture supports a lightweight memory management mechanism with fixed virtual-to-physical address translation, and no memory protection beyond what is provided by the address error checks required of all MMUs. This may be useful for those applications which do not require the capabilities of a full TLB-based MMU. It is not anticipated that MIPS64 processors that implement a fixed-mapping MMU will require a 64-bit address capability. As a result, the description below is given assuming a 32-bit address.

A.1.1 Fixed Address Translation

Address translation using the Fixed Mapping MMU is done as follows:

- Kseg0 and Kseg1 addresses are translated in an identical manner to the TLB-based MMU: they both map to the low 512MB of physical memory.
- Useg/Suseg/Kuseg addresses are mapped by adding 1GB to the virtual address when the ERL bit is zero in the Status register, and are mapped using an identity mapping when the ERL bit is one in the Status register.
- Sseg/Ksseg/Kseg2/Kseg3 addresses are mapped using an identity mapping.

Table 92 lists all mappings from virtual to physical addresses. Note that address error checking is still done before the translation process. Therefore, an attempt to reference kseg0 from User Mode still results in an address error exception, just as it does with a TLB-based MMU.

Table 92: Physical Address Generation from Virtual Addresses

Segment Name	Virtual Address	Generates Physical Address	
		$Status_{ERL} = 0$	$Status_{ERL} = 1$
useg suseg kuseg	0x 0000 0000 through 0x 7FFF FFFF	0x 4000 0000 through 0x BFFF FFFF	0x 0000 0000 through 0x 7FFF FFFF
kseg0	0x 8000 0000 through 0x 9FFF FFFF	0x 0000 0000 through 0x 1FFF FFFF	
kseg1	0x A000 0000 through 0x BFFF FFFF	0x 0000 0000 through 0x 1FFF FFFF	
sseg ksseg kseg2	0x C000 0000 through 0x DFFF FFFF	0x C000 0000 through 0x DFFF FFFF	
kseg3	0x E000 0000 through 0x FFFF FFFF	0x E000 0000 through 0x FFFF FFFF	

Note that this mapping means that physical addresses 0x2000 0000 through 0x3FFF FFFF are inaccessible when the ERL bit is off in the *Status* register, and physical addresses 0x8000 0000 through 0xBFFF FFFF are inaccessible when the ERL bit is on in the *Status* register.

Figure 42 shows the memory mapping when the ERL bit in the *Status* register is zero; Figure 43 shows the memory mapping when the ERL bit is one.

Figure 42: Memory Mapping when ERL = 0

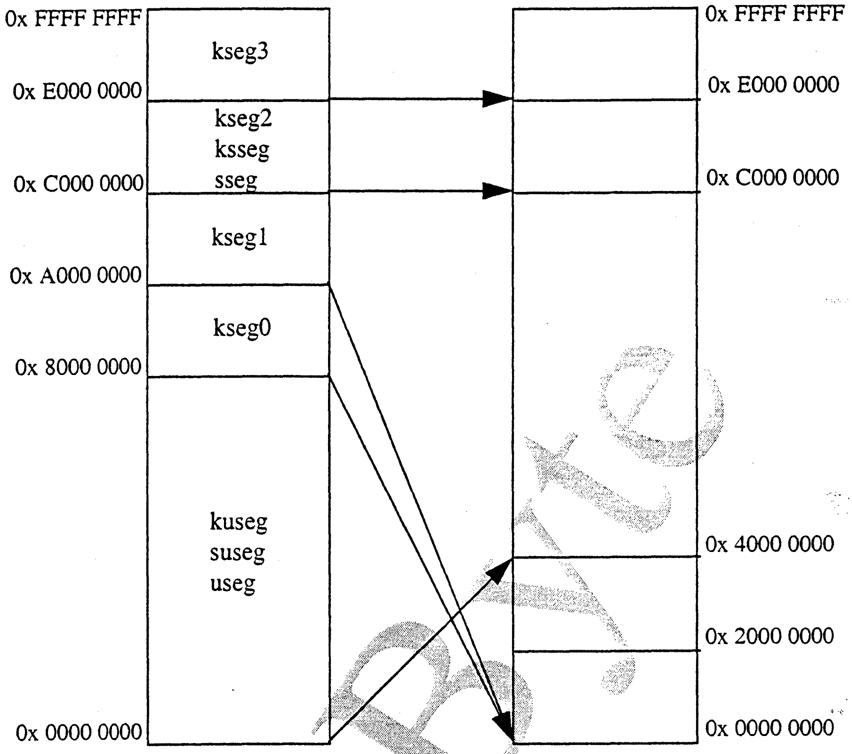
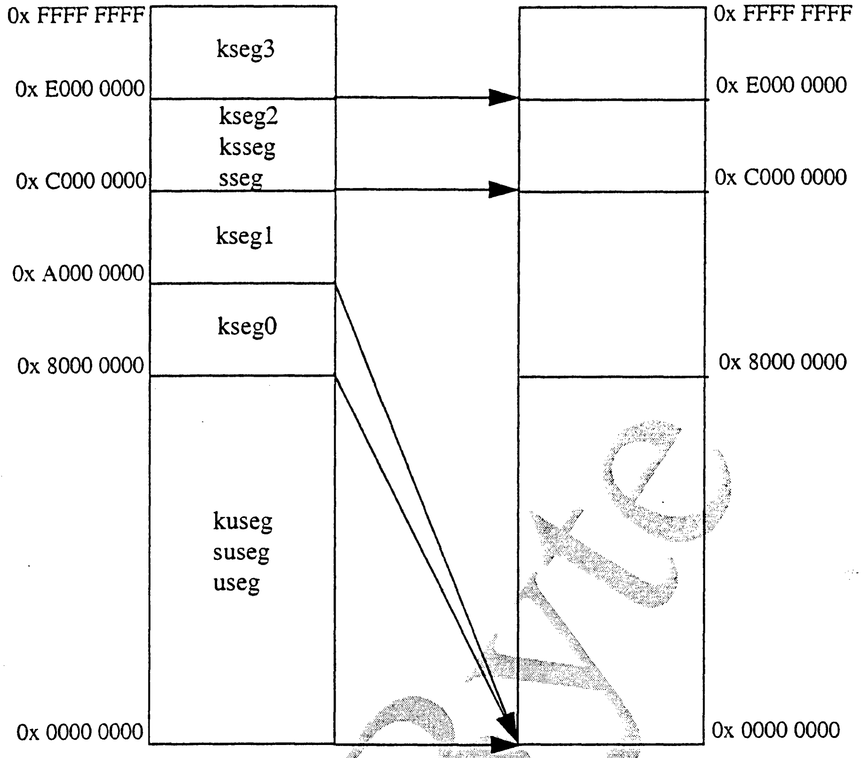


Figure 43: Memory Mapping when ERL = 1



A.1.2 Cacheability Attributes

Because the TLB provided the cacheability attributes for the kuseg, kseg2, and kseg3 segments, some mechanism is required to replace this capability when the fixed mapping MMU is used. Two additional fields are added to the *Config* register whose encoding is identical to that of the K0 field. These additions are the K23 and KU fields which control the cacheability of the kseg2/kseg3 and the kuseg segments, respectively. Note that when the ERL bit is on in the Status register, kuseg references are always treated as uncacheable references, independent of the value of the KU field.

The cacheability attributes for kseg0 and kseg1 are provided in the same manner as for a TLB-based MMU: the cacheability attribute for kseg0 comes from the K0 field of *Config*, and references to kseg1 are always uncached.

Figure 44 shows the format of the additions to the *Config* register; Table 93 describes the new *Config* register fields.

Figure 44: Config Register Additions

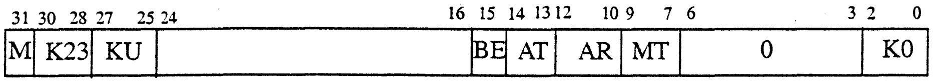


Table 93: Config Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
K23	30:28	Kseg2/Kseg3 coherency algorithm. See Table 62 for the encoding of this field.	R/W	Undefined	Optional
KU	27:25	Kuseg coherency algorithm when $\mathrm{Status}_{\mathrm{ERL}}$ is zero. See Table 62 for the encoding of this field.	R/W	Undefined	Optional

A.1.3 Changes to the CP0 Register Interface

Relative to the TLB-based address translation mechanism, the following changes are necessary to the CP0 register interface:

- The Index, Random, EntryLo0, EntryLo1, Context, PageMask, Wired, and EntryHi registers are no longer required and may be removed
- The TLBWR, TLBWI, TLBP, and TLBR instructions are no longer required and should cause a Reserved Instruction Exception

A.2 Block Address Translation

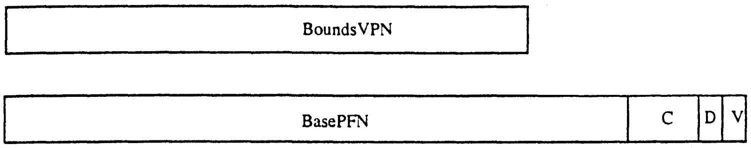
This section describes the architecture for a block address translation (BAT) mechanism that reuses much of the hardware and software interface that exists for a TLB-Based virtual address translation mechanism. This mechanism has the following features:

- It preserves as much as possible of the TLB-Based interface, both in hardware and software.
- It provides independent base-and-bounds checking and relocation for instruction references and data references.
- It provides optional support for base-and-bounds relocation of kseg2 and kseg3 virtual address regions

A.2.1 BAT Organization

The BAT is an indexed structure which is used to translate virtual addresses. It contains pairs of instruction/data entries which provide the base-and-bounds checking and relocation for instruction references and data references, respectively. Each entry contains a page-aligned bounds virtual page number, a base page frame number (whose width is implementation dependent), a cache coherence field (C), a dirty (D) bit, and a valid (V) bit. Figure 45 shows the logical arrangement of a BAT entry.

Figure 45: Contents of a BAT Entry



The BAT is indexed by the reference type and the address region to be checked as shown in Table 94.

Table 94: BAT Entry Assignments

Entry Index	Reference Type	Address Region
0	Instruction	useg/kuseg
1	Data	
2	Instruction	kseg2 (or kseg2 and kseg3)
3	Data	
4	Instruction	kseg3
5	Data	

Entries 0 and 1 are required. Entries 2 and 3 and 4 and 5 are optional and may be implemented as necessary to address the needs of the particular implementation. If entries for kseg2 and kseg3 are not implemented, it is implementation-dependent how, if at all, these address regions are translated. One alternative is to combine the mapping for kseg2 and kseg3 into a single pair of instruction/data entries. Software may determine how many BAT entries are implemented by looking at the MMU Size field of the *Config1* register.

A.2.2 Address Translation

When a virtual address translation is requested, the BAT entry that is appropriate to the reference type and address region is read. If the virtual address is greater than the selected bounds address, or if the valid bit is off in the entry, a TLB Invalid exception of the appropriate reference type is initiated. If the reference is a store and the D bit is off in the entry, a TLB Modified exception is initiated. Otherwise, the base PFN from the selected entry, shifted to align with bit 12, is added to the virtual address to form the physical address. The BAT process can be described as follows:

```
i ← SelectIndex (reftype, va)
bounds ← $\mathrm{BAT[i]}_{\mathrm{BoundsVPN}} \,||\, 1^{12}$
pfn ← $\mathrm{BAT[i]}_{\mathrm{BasePFN}}$
c ← $\mathrm{BAT[i]}_{\mathrm{C}}$
d ← $\mathrm{BAT[i]}_{\mathrm{D}}$
v ← $\mathrm{BAT[i]}_{\mathrm{V}}$
```

```
if (va > bounds) or (v = 0) then
    InitiateTLBInvalidException(reftype)
endif
if (d = 0) and (reftype = store) then
    InitiateTLBModifiedException()
endif
pa ← va + (pfn || $0^{12}$)
```

Making all addresses out-of-bounds can only be done by clearing the valid bit in the BAT entry. Setting the bounds value to zero leaves the first virtual page mapped.

A.2.3 Changes to the CP0 Register Interface

Relative to the TLB-based address translation mechanism, the following changes are necessary to the CP0 register interface:

- The *Index* register is used to index the BAT entry to be read or written by the TLBWI and TLBR instructions.
- The *EntryHi* register is the interface to the BoundsVPN field in the BAT entry.
- The *EntryLo0* register is the interface to the BasePFN and C, D, and V fields of the BAT entry. The register has the same format as for a TLB-based MMU.
- The *Random*, *EntryLo1*, *Context*, *PageMask*, and *Wired* registers are eliminated. The effects of a read or write to these registers is **UNDEFINED**.
- The TLBP and TLBWR instructions are unnecessary. The TLBWI and TLBR instructions reference the BAT entry whose index is contained in the *Index* register. The effects of executing a TLBP or TLBWR are **UNDEFINED**, but processors should prefer a Reserved Instruction Exception.