



**MIPS32® 74K™ Processor Core Family
Software User's Manual**

Document Number: MD00519

Revision 01.05

March 30, 2011

Unpublished rights (if any) reserved under the copyright laws of the United States of America and other countries.

This document contains information that is proprietary to MIPS Tech, LLC, a Wave Computing company ("MIPS") and MIPS' affiliates as applicable. Any copying, reproducing, modifying or use of this information (in whole or in part) that is not expressly permitted in writing by MIPS or MIPS' affiliates as applicable or an authorized third party is strictly prohibited. At a minimum, this information is protected under unfair competition and copyright laws. Violations thereof may result in criminal penalties and fines. Any document provided in source format (i.e., in a modifiable form such as in FrameMaker or Microsoft Word format) is subject to use and distribution restrictions that are independent of and supplemental to any and all confidentiality restrictions. UNDER NO CIRCUMSTANCES MAY A DOCUMENT PROVIDED IN SOURCE FORMAT BE DISTRIBUTED TO A THIRD PARTY IN SOURCE FORMAT WITHOUT THE EXPRESS WRITTEN PERMISSION OF MIPS (AND MIPS' AFFILIATES AS APPLICABLE) reserve the right to change the information contained in this document to improve function, design or otherwise.

MIPS and MIPS' affiliates do not assume any liability arising out of the application or use of this information, or of any error or omission in such information. Any warranties, whether express, statutory, implied or otherwise, including but not limited to the implied warranties of merchantability or fitness for a particular purpose, are excluded. Except as expressly provided in any written license agreement from MIPS or an authorized third party, the furnishing of this document does not give recipient any license to any intellectual property rights, including any patent rights, that cover the information in this document.

The information contained in this document shall not be exported, reexported, transferred, or released, directly or indirectly, in violation of the law of any country or international law, regulation, treaty, Executive Order, statute, amendments or supplements thereto. Should a conflict arise regarding the export, reexport, transfer, or release of the information contained in this document, the laws of the United States of America shall be the governing law.

The information contained in this document constitutes one or more of the following: commercial computer software, commercial computer software documentation or other commercial items. If the user of this information, or any related documentation of any kind, including related technical data or manuals, is an agency, department, or other entity of the United States government ("Government"), the use, duplication, reproduction, release, modification, disclosure, or transfer of this information, or any related documentation of any kind, is restricted in accordance with Federal Acquisition Regulation 12.212 for civilian agencies and Defense Federal Acquisition Regulation Supplement 227.7202 for military agencies. The use of this information by the Government is further restricted in accordance with the terms of the license agreement(s) and/or applicable contract terms and conditions covering this information from MIPS Technologies or an authorized third party.

MIPS, MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MIPSr3, MIPS32, MIPS64, microMIPS32, microMIPS64, MIPS-3D, MIPS16, MIPS16e, MIPS-Based, MIPSsim, MIPSpro, MIPS-VERIFIED, Aptiv logo, microAptiv logo, interAptiv logo, microMIPS logo, MIPS Technologies logo, MIPS-VERIFIED logo, proAptiv logo, 4K, 4Kc, 4Km, 4Kp, 4KE, 4KEc, 4KEm, 4KEp, 4KS, 4KSc, 4KSd, M4K, M14K, 5K, 5Kc, 5Kf, 24K, 24Kc, 24Kf, 24KE, 24KEc, 24KEf, 34K, 34Kc, 34Kf, 74K, 74Kc, 74Kf, 1004K, 1004Kc, 1004Kf, 1074K, 1074Kc, 1074Kf, R3000, R4000, R5000, Aptiv, ASMACRO, Atlas, "At the core of the user experience.", BusBridge, Bus Navigator, CLAM, CorExtend, CoreFPGA, CoreLV, EC, FPGA View, FS2, FS2 FIRST SILICON SOLUTIONS logo, FS2 NAVIGATOR, HyperDebug, HyperJTAG, IASim, iFlowtrace, interAptiv, JALGO, Logic Navigator, Malta, MDMX, MED, MGB, microAptiv, microMIPS, Navigator, OCI, PDtrace, the Pipeline, proAptiv, Pro Series, SEAD-3, SmartMIPS, SOC-it, and YAMON are trademarks or registered trademarks of MIPS and MIPS' affiliates as applicable in the United States and other countries.

All other trademarks referred to herein are the property of their respective owners.

Table of Contents

Chapter 1: Introduction to the MIPS32® 74K™ Core Family	23
1.1: 74K™ Core Features	24
1.1.1: Pipeline	24
1.1.2: Instruction Set	25
1.1.3: Memory Management, Caches, and Scratchpad Memory	26
1.1.4: Interfaces	27
1.1.5: Power Control	27
1.1.6: Debug	27
1.1.7: Other	28
1.2: 74K™ Core Block Diagram	28
1.2.1: Instruction Fetch Unit (IFU)	30
1.2.2: Instruction Cache	30
1.2.3: Instruction Decode/Dispatch Unit (IDU)	30
1.2.4: Instruction Execution Unit (IEU)	31
1.2.5: Multiply Divide Unit (MDU)	31
1.2.6: CorExtend® User Defined Instructions (UDIs)	32
1.2.7: Load Store Unit (LSU)	32
1.2.8: System Control Coprocessor (CP0)	32
1.2.9: Memory Management Unit (MMU)	32
1.2.10: Data Cache	33
1.2.11: Scratchpad RAM	33
1.2.12: Graduation Unit (GRU)	33
1.2.13: Bus Interface Unit (BIU)	34
1.2.14: Coprocessor Interface Unit (CIU)	34
1.2.15: Power Management	34
1.2.16: EJTAG Debug	34
Chapter 2: Pipeline of the 74K™ Core	37
2.1: Integer Pipeline Description	37
2.1.1: IFU Pipeline	38
2.1.1.1: IT - Instruction Cache Tag Access	38
2.1.1.2: ID - Instruction Cache Data Access	39
2.1.1.3: IS - Instruction Select	39
2.1.1.4: IR - Instruction Recode	39
2.1.1.5: IK - Instruction	39
2.1.1.6: IX - Instruction Macro Expansion	39
2.1.1.7: IB - Instruction Buffer	40
2.1.2: Instruction Decode Unit Pipeline	40
2.1.2.1: DD - Dispatch Decode	40
2.1.2.2: DR - Dispatch Rename	40
2.1.2.3: DS - Dispatch Select	40
2.1.2.4: DM - DDQ Mux	41
2.1.3: ALU Pipeline	41
2.1.3.1: AF - ALU Pipe Register File Read	41
2.1.3.2: AM - ALU Pipe Operand Bypass Select Mux, This stage has the final operand bypass muxes for the ALU pipe. The functionality corresponding to this stage resides entirely in the IEU.	41
2.1.3.3: AC - ALU Compute	41

2.1.3.4: AB - ALU Bypass	41
2.1.4: MDU Pipeline	42
2.1.4.1: MB - Multiplier Booth Recode	42
2.1.4.2: M1- M3 Multiplier Array	42
2.1.4.3: M4 - Multiply Add	42
2.1.5: AGEN Pipeline	42
2.1.5.1: EM - Execute Operand Bypass Select Mux	42
2.1.5.2: EA - Execute and Address Generate	42
2.1.5.3: EC - Execute and Cache Access	43
2.1.5.4: ES - Execute and Cache Second	43
2.1.5.5: EB - Execute and Cache Data Bypass	43
2.1.6: GRU Pipeline	44
2.1.6.1: WB - Writeback.....	44
2.1.6.2: GC - Graduation Commit.....	44
2.2: Programming the 74K Core.....	44
2.3: Hazards	44
2.3.1: Types of Hazards	45
2.3.1.1: Execution Hazards, Execution hazards are those created by the execution of one instruction, and seen by the execution of another instruction. Table 2.2 lists possible execution hazards and whether they can be resolved via setting of the IHB bit in the CP0 Config7 register.	46
2.3.1.2: Instruction Hazards, Instruction hazards are those created by the execution of one instruction, and seen by the instruction fetch of another instruction. Table 2.3 lists instruction hazards. Because the fetch unit is decoupled from the execution unit, these hazards are rather large. The use of a hazard barrier instruction is required for reliable clearing of instruction hazards.	47
2.3.2: Instruction Listing	47
2.3.2.1: Instruction Encoding	48
2.3.3: Eliminating Hazards	48
Chapter 3: Floating-Point Unit of the 74Kf™ Core.....	49
3.1: Features Overview	49
3.1.1: IEEE Standard 754	50
3.2: Enabling the Floating-Point Coprocessor	50
3.3: Data Formats.....	51
3.3.1: Floating-Point Formats.....	51
3.3.1.1: Normalized and Denormalized Numbers.....	53
3.3.1.2: Reserved Operand Values—Infinity and NaN	53
3.3.1.3: Infinity and Beyond	53
3.3.1.4: Signalling Non-Number (SNaN)	53
3.3.1.5: Quiet Non-Number (QNaN)	53
3.3.2: Fixed-Point Formats.....	54
3.4: Floating-Point General Registers	55
3.4.1: FPRs and Formatted Operand Layout.....	55
3.4.2: Formats of Values Used in FP Registers	55
3.4.3: Binary Data Transfers (32-Bit and 64-Bit)	57
3.5: Floating-Point Control Registers.....	58
3.5.1: Floating-Point Implementation Register (FIR, CP1 Control Register 0).....	59
3.5.2: Floating-Point Condition Codes Register (FCCR, CP1 Control Register 25).....	60
3.5.3: Floating-Point Exceptions Register (FEXR, CP1 Control Register 26)	60
3.5.4: Floating-Point Enables Register (FENR, CP1 Control Register 28)	61
3.5.5: Floating-Point Control and Status Register (FCSR, CP1 Control Register 31).....	62
3.5.6: Operation of the FS/FO/FN Bits	64
3.5.6.1: Flush To Zero Bit	65
3.5.6.2: Flush Override Bit.....	65

3.5.6.3: Flush to Nearest	66
3.5.6.4: Recommended FS/FO/FN Settings	67
3.5.7: FCSR Cause Bit Update Flow.....	67
3.5.7.1: Exceptions Triggered by CTC1	67
3.5.7.2: Generic Flow	67
3.5.7.3: Multiply-Add Flow	67
3.5.7.4: Cause Update Flow for Input Operands	68
3.5.7.5: Cause Update Flow for Unimplemented Operations	68
3.6: Instruction Overview	68
3.6.1: Data Transfer Instructions	68
3.6.1.1: Data Alignment in Loads, Stores, and Moves	69
3.6.1.2: Addressing Used in Data Transfer Instructions	69
3.6.2: Arithmetic Instructions.....	70
3.6.3: Conversion Instructions.....	71
3.6.4: Formatted Operand-Value Move Instructions	71
3.6.5: Conditional Branch Instructions	72
3.6.6: Miscellaneous Instructions	73
3.7: Exceptions	73
3.7.1: Precise Exception Mode	73
3.7.2: Exception Conditions	74
3.7.2.1: Invalid Operation Exception.....	75
3.7.2.2: Division By Zero Exception.....	75
3.7.2.3: Underflow Exception.....	76
3.7.2.4: Overflow Exception.....	76
3.7.2.5: Inexact Exception	76
3.7.2.6: Unimplemented Operation Exception.....	76
3.8: Pipeline and Performance	77
3.8.1: Pipeline Overview	77
3.8.1.1: DR Stage - Dispatch Rename	77
3.8.1.2: C1 - Coprocessor Interface Unit Stage 1.....	77
3.8.1.3: CR Stage - Coprocessor Interface Unit Queue Read.....	78
3.8.1.4: CI Stage - Coprocessor 1 Interface	78
3.8.1.5: FR Stage - Decode, Register Read, and Unpack.....	78
3.8.1.6: M1 Stage - Multiply Tree	78
3.8.1.7: M2 Stage - Multiply Complete	78
3.8.1.8: A1 Stage - Addition First Step	78
3.8.1.9: A2 Stage - Addition Second and Final Step	78
3.8.1.10: FP Stage - Result Pack	79
3.8.1.11: FW Stage - Register Write.....	79
3.8.2: Bypassing.....	79
3.8.3: Repeat Rate and Latency	79

Chapter 4: The MIPS® DSP Application-Specific Extension to the MIPS32® Instruction Set..... 81

4.1: Additional Register State for the DSP ASE	81
4.1.1: HI-LO Registers	81
4.1.2: DSP Control Register.....	81
4.2: Software Detection of the DSP ASE Revision 2.....	83

Chapter 5: Memory Management of the 74K™ Core **85**

5.1: Introduction.....	85
5.2: Modes of Operation	87
5.2.1: Virtual Memory Segments.....	87
5.2.1.1: Unmapped Segments.....	88

5.2.1.2: Mapped Segments	89
5.2.2: User Mode.....	89
5.2.3: Supervisor Mode	90
5.2.4: Kernel Mode.....	92
5.2.4.1: Kernel Mode, User Space (kuseg)	94
5.2.4.2: Kernel Mode, Kernel Space 0 (kseg0).....	94
5.2.4.3: Kernel Mode, Kernel Space 1 (kseg1).....	94
5.2.4.4: Kernel Mode, Kernel/Supervisor Space 2 (ksseg/kseg2)	95
5.2.4.5: Kernel Mode, Kernel Space 3 (kseg3).....	95
5.2.5: Debug Mode.....	95
5.2.5.1: Debug Mode, Register (drseg)	96
5.2.5.2: Debug Mode, Memory (dmseg).....	96
5.3: Translation Lookaside Buffer	97
5.3.1: Joint TLB	97
5.3.2: Instruction TLB.....	99
5.4: Virtual-to-Physical Address Translation.....	100
5.4.1: Hits, Misses, and Multiple Matches.....	101
5.4.2: Memory Space	102
5.4.2.1: Page Sizes	102
5.4.2.2: Replacement Algorithm	102
5.4.3: TLB Instructions	103
5.5: Fixed Mapping MMU	104

Chapter 6: Exceptions and Interrupts in the 74K™ Core..... 107

6.1: Exception Conditions.....	107
6.2: Exception Priority.....	108
6.3: Interrupts	109
6.3.1: Interrupt Modes.....	110
6.3.1.1: Interrupt Compatibility Mode.....	110
6.3.1.2: Vectored Interrupt Mode.....	112
6.3.1.3: External Interrupt Controller Mode	115
6.3.2: Generation of Exception Vector Offsets for Vectored Interrupts	117
6.4: GPR Shadow Registers.....	118
6.5: Exception Vector Locations	120
6.6: General Exception Processing	122
6.7: Debug Exception Processing	125
6.8: Exception Descriptions	126
6.8.1: Reset Exception	127
6.8.2: Debug Single Step Exception	128
6.8.3: Debug Interrupt Exception	128
6.8.4: Non-Maskable Interrupt (NMI) Exception.....	129
6.8.5: Machine Check Exception.....	129
6.8.6: Interrupt Exception	130
6.8.7: Debug Instruction Break Exception.....	130
6.8.8: Watch Exception — Instruction Fetch or Data Access.....	130
6.8.9: Address Error Exception — Instruction Fetch/Data Access.....	131
6.8.10: TLB Refill Exception — Instruction Fetch or Data Access	132
6.8.11: TLB Invalid Exception — Instruction Fetch or Data Access.....	132
6.8.12: Cache Error Exception	133
6.8.13: Bus Error Exception — Instruction Fetch or Data Access.....	133
6.8.14: Debug Software Breakpoint Exception	134
6.8.15: Execution Exception — System Call.....	134
6.8.16: Execution Exception — Breakpoint.....	134

6.8.17: Execution Exception — Reserved Instruction	135
6.8.18: Execution Exception — Coprocessor Unusable	135
6.8.19: Execution Exception — CorExtend block Unusable	136
6.8.20: Execution Exception — DSP ASE State Disabled	136
6.8.21: Execution Exception — Floating Point Exception	136
6.8.22: Execution Exception — Integer Overflow.....	137
6.8.23: Execution Exception — Trap.....	137
6.8.24: Debug Data Break Exception.....	137
6.8.25: TLB Modified Exception — Data Access	137
6.9: Exception Handling and Servicing Flowcharts	138

Chapter 7: CP0 Registers of the 74K™ Core **145**

7.1: CP0 Register Summary	145
7.2: CP0 Register Descriptions	150
7.2.1: Index (CP0 Register 0, Select 0): Index into TLB array	151
7.2.2: Random (CP0 Register 1, Select 0): Randomly Generated Index into the TLB Array.....	152
7.2.3: EntryLo0-1 (CP0 Registers 2 and 3, Select 0): Output (physical) side of TLB entry.....	152
7.2.4: Context (CP0 Register 4, Select 0): Mixture of Pre-programmed and BadVAddr Bits which can act as an OS Page Table Pointer.	154
7.2.5: ContextConfig Register (CP0 Register 4, Select 1)	155
7.2.6: UserLocal (CP0 Register 4, Select 2): Address Causing the Last TLB-related Exception	156
7.2.7: PageMask (CP0 Register 5, Select 0): Control for Variable Page Size in TLB Entries	157
7.2.8: Wired (CP0 Register 6, Select 0): Controls Number of Fixed ("wired") TLB Entries.....	158
7.2.9: HWREna (CP0 Register 7, Select 0): Bitmask Limiting User-mode Access to rdhwr Registers....	158
7.2.10: BadVAddr (CP0 Register 8, Select 0): Address Causing the Last TLB-related Exception	160
7.2.11: Count (CP0 Register 9, Select 0): Free-running Counter at Half the Pipeline Speed.....	160
7.2.12: EntryHi (CP0 Register10, Select 0): High-order Portion of TLB Entry	161
7.2.13: Compare (CP0 Register 11, Select 0): Timer Interrupt Control	162
7.2.14: Status (CP0 Register 12, Select 0): Processor Status and Control	162
7.2.14.1: Interruptibility	162
7.2.14.2: Operating Modes	163
7.2.14.3: Coprocessor Accessibility.....	163
7.2.15: IntCtl (CP0 Register 12, Select 1): Setup for Interrupt Vector and Interrupt Priority Features....	168
7.2.16: SRSCtl (CP0 Register12, Select 2): Shadow Register Set Selectors.....	169
7.2.17: SRSSMap (CP0 Register 12, Select 3): Shadow Set Choice for Each Interrupt Level in VI Mode	171
7.2.18: Cause (CP0 Register 13, Select 0): Cause of Last General Exception	172
7.2.19: EPC (CP0 Register 14, Select 0): Restart Address from Exception	176
7.2.20: PRId (CP0 Register 15, Select 0): Processor Identification and Revision	177
7.2.21: EBase (CP0 Register 15, Select 1): Exception entry point base address and CPU/VPE ID	178
7.2.22: CDMMBase Register (CP0 Register 15, Select 2).....	179
7.2.23: Config (CP0 Register 16, Select 0): Legacy Configuration Register.....	180
7.2.24: Config1-2 (CP0 Register 16, Select 1-2): MIPS32/64 Configuration Registers	181
7.2.24.1: Config1	181
7.2.24.2: Config2	183
7.2.25: Config3 (CP0 Register 16, Select 3): Configuration register showing ASEs	183
7.2.26: Config6 (CP0 Register 16, Select 6).....	184
7.2.27: Config7 (CP0 Register 16, Select 7): CPU-specific Configuration.....	187
7.2.28: WatchLo0-3 (CP0 Register 18, Select 0-3): Watchpoint Address and Qualifiers	190
7.2.29: WatchHi0-3 (CP0 Register 19, Select 0-3): Watchpoint Control/Status.....	190
7.2.30: Debug (CP0 Register 23, Select 0): EJTAG Debug Status/Control Register.....	191
7.2.31: Trace Control Register (CP0 Register 23, Select 1)	195
7.2.32: Trace Control2 Register (CP0 Register 23, Select 2)	197

7.2.33: User Trace Data1 Register (CP0 Register 23, Select 3) and User Trace Data2 Register (CP0 Register 24, Select 3).....	199
7.2.34: TraceIBPC Register (CP0 Register 23, Select 4)	200
7.2.35: TraceDBPC Register (CP0 Register 23, Select 5).....	201
7.2.36: DEPC (CP0 Register 24, Select 0): Restart Address from Last EJTAG Debug Exception.....	202
7.2.37: Trace Control3 Register (CP0 Register 24, Select 2)	203
7.2.38: PerfCtl0-3 (CP0 Register 25, Select 0, 2, 4, 6): Performance Counter Control	204
7.2.39: PerfCnt0-3 (CP0 Register 25, Select 1, 3, 5, 7): Performance Counters.....	209
7.2.40: ErrCtl (CP0 Register 26, Select 0): Software Parity Control and Test Modes for Cache RAM Arrays	209
7.2.41: CacheErr (CP0 Register 27, Select 0): Cache Parity Exception Status.....	210
7.2.42: ITagLo (CP0 Register 28, Select 0): Read/write Interface for Load/Store Tag Cacheops	212
7.2.42.1: ITagLo (ErrCtlWST = 0, ErrCtlSPR = 0)	212
7.2.42.2: ITagLo-WST (ErrCtlWST = 1, ErrCtlSPR = 0)	212
7.2.42.3: ITagLo-WST (ErrCtlWST = 0, ErrCtlSPR = 1)	213
7.2.43: IDataLo (CP0 Register 28, Select 1): Read/write Interface for I-cache Special Cacheops.....	213
7.2.44: DTagLo (CP0 Register 28, Select 2): Read/Write Interface for Load/Store Tag Cacheops	214
7.2.44.1: DTagLo (ErrCtlWST = 0, ErrCtlDYT = 0, ErrCtlSPR = 0)	214
7.2.44.2: DTagLo-WST(ErrCtlWST = 1, ErrCtlDYT = 0, ErrCtlSPR = 0)	214
7.2.44.3: DTagLo-DYT (ErrCtlWST = 0, ErrCtlDYT = 1, ErrCtlSPR = 0).....	215
7.2.45: DDataLo (CP0 Register 28, Select 3): Low-order Data Read/Write Interface for D-cache.....	216
7.2.46: L23TagLo (CP0 Register 28, Select 4): L2 and L3 Cacheop Tag Use	216
7.2.47: L23DataLo (CP0 Register 28, Select 5): Low-order Data Read/Write Interface for L2 or L3 cache..	216
7.2.48: ITagHi (CP0 Register 29, Select 0): I-cache Predecode Bits.....	217
7.2.49: IDataHi (CP0 Register 29, Select 1): High-order Data Read/write Interface for I-cache Special	Cacheops
	217
7.2.50: DTagHi (CP0 Register 29, Select 2): D-cache Virtual Index (including ASID).....	218
7.2.51: L23DataHi (CP0 Register 29, Select 5): High-order Data Read/Write Interface for L2 or L3 cache..	218
7.2.52: ErrorEPC (CP0 Register 30, Select 0): Restart Location from Reset or Cache Error Exception .	219
7.2.53: DESAVE (CP0 Register 31, Select 0): Scratch Read/Write Register for EJTAG Debug Exception	Handler.....
	219

Chapter 8: Hardware and Software Initialization of the 74K™ Core 221

8.1: Hardware-Initialized Processor State	221
8.1.1: Coprocessor 0 State	221
8.1.2: TLB Initialization.....	222
8.1.3: Bus State Machines	222
8.1.4: Static Configuration Inputs	222
8.1.5: Fetch Address.....	222
8.2: Software-Initialized Processor State.....	222
8.2.1: Register File.....	222
8.2.2: TLB.....	223
8.2.3: Caches	223
8.2.4: Coprocessor 0 State	223

Chapter 9: Caches of the 74K™ Core 225

9.1: Cache Configurations	225
9.2: Instruction Cache.....	225
9.2.1: Virtual Aliasing	226
9.2.2: Precode bits	226
9.2.3: Parity	227

9.3: Data Cache.....	227
9.3.1: Virtual Aliasing	228
9.3.2: Parity	229
9.4: Write Back Buffer.....	229
9.4.1: Uncached Accelerated Stores.....	230
9.5: Cache Protocols	231
9.5.1: Cache Organization	231
9.5.2: Cacheability Attributes	231
9.5.3: Replacement Policy	232
9.5.4: Line Locking	233
9.6: CACHE Instruction	233
9.7: Software Cache Testing	234
9.7.1: I-cache and D-cache Tag Arrays	234
9.7.2: I-cache Data Array	234
9.7.3: I-cache WS Array	234
9.7.4: D-cache Data Array.....	234
9.7.5: D-cache WS Array	235
9.7.6: D-cache DirtyArray.....	235
9.8: Memory Coherence Issues.....	235
Chapter 10: Power Management in the 74K™ Core	237
10.1: Register-Controlled Power Management	237
10.2: Instruction-Controlled Power Management	238
Chapter 11: EJTAG Debug Support in the 74K™ Core	239
11.1: Debug Control Register	240
11.2: Hardware Breakpoints	244
11.2.1: Features of Instruction Breakpoint	244
11.2.2: Features of Data Breakpoint	245
11.2.3: Instruction Breakpoint Registers Overview	245
11.2.4: Data Breakpoint Registers Overview	245
11.2.5: Conditions for Matching Breakpoints	246
11.2.5.1: Conditions for Matching Instruction Breakpoints	246
11.2.5.2: Conditions for Matching Data Breakpoints	246
11.2.6: Debug Exceptions from Breakpoints.....	248
11.2.6.1: Debug Exception by Instruction Breakpoint.....	248
11.2.6.2: Debug Exception by Data Breakpoint.....	248
11.2.7: Breakpoint used as TriggerPoint.....	250
11.2.8: Instruction Breakpoint Registers	250
11.2.8.1: Instruction Breakpoint Status (IBS) Register	250
11.2.8.2: Instruction Breakpoint Address n (IBAn) Register	251
11.2.8.3: Instruction Breakpoint Address Mask n (IBMn) Register	251
11.2.8.4: Instruction Breakpoint ASID n (IBASIDn) Register.....	252
11.2.8.5: Instruction Breakpoint Control n (IBCn) Register	252
11.2.9: Data Breakpoint Registers	254
11.2.9.1: Data Breakpoint Status (DBS) Register	254
11.2.9.2: Data Breakpoint Address n (DBAn) Register	255
11.2.9.3: Data Breakpoint Address Mask n (DBMn) Register	255
11.2.9.4: Data Breakpoint ASID n (DBASIDn) Register	256
11.2.9.5: Data Breakpoint Control n (DBCn) Register	256
11.2.9.6: Data Breakpoint Value n (DBVn) Register	258
11.2.9.7: Data Breakpoint Value High n (DBVHn) Register	258
11.3: Test Access Port (TAP)	259

11.3.1: EJTAG Internal and External Interfaces.....	259
11.3.2: Test Access Port Operation	260
11.3.2.1: Test-Logic-Reset State	261
11.3.2.2: Run-Test/Idle State.....	261
11.3.2.3: Select_DR_Scan State	261
11.3.2.4: Select_IR_Scan State	261
11.3.2.5: Capture_DR State	262
11.3.2.6: Shift_DR State	262
11.3.2.7: Exit1_DR State	262
11.3.2.8: Pause_DR State	262
11.3.2.9: Exit2_DR State	262
11.3.2.10: Update_DR State	262
11.3.2.11: Capture_IR State	262
11.3.2.12: Shift_IR State	263
11.3.2.13: Exit1_IR State.....	263
11.3.2.14: Pause_IR State	263
11.3.2.15: Exit2_IR State.....	263
11.3.2.16: Update_IR State	263
11.3.3: Test Access Port (TAP) Instructions	263
11.3.3.1: BYPASS Instruction.....	264
11.3.3.2: IDCODE Instruction	264
11.3.3.3: IMPCODE Instruction	264
11.3.3.4: ADDRESS Instruction.....	264
11.3.3.5: DATA Instruction	264
11.3.3.6: CONTROL Instruction	265
11.3.3.7: ALL Instruction.....	265
11.3.3.8: EJTAGBOOT Instruction	265
11.3.3.9: NORMALBOOT Instruction	265
11.3.3.10: FASTDATA Instruction	265
11.3.3.11: TCBCONTROLA Instruction.....	266
11.3.3.12: TCBCONTROLB Instruction.....	266
11.3.3.13: TCBCONTROLC Instruction.....	266
11.3.3.14: TCBCONTROLE Instruction.....	266
11.3.3.15: TCBDATA Instruction	266
11.3.3.16: PCSAMPLE Instruction	266
11.3.3.17: FDC Instruction.....	266
11.4: EJTAG TAP Registers	266
11.4.1: Instruction Register	266
11.4.2: Data Registers Overview	267
11.4.2.1: Bypass Register	267
11.4.2.2: Device Identification (ID) Register	267
11.4.2.3: Implementation Register.....	268
11.4.2.4: EJTAG Control Register	269
11.4.3: Processor Access Address Register.....	275
11.4.3.1: Processor Access Data Register.....	275
11.4.4: Fastdata Register (TAP Instruction FASTDATA).....	276
11.5: TAP Processor Accesses	277
11.5.1: Fetch/Load and Store From/To the EJTAG Probe Through dmseg.....	278
11.6: PC Sampling.....	279
11.6.1: PC Sampling in Wait State.....	280
11.7: Fast Debug Channel.....	280
11.7.1: Common Device Memory Map.....	280
11.7.2: Fast Debug Channel Interrupt.....	280

11.7.3: 74K™Core FDC Buffers.....	281
11.7.4: Sleep mode	283
11.7.5: FDC TAP Register	283
11.7.6: Fast Debug Channel Registers	284
11.7.6.1: FDC Access Control and Status (FDACSR) Register (Offset 0x0).....	284
11.7.6.2: FDC Configuration (FDCFG) Register (Offset 0x8)	285
11.7.6.3: FDC Status (FDSTAT) Register (Offset 0x10)	286
11.7.6.4: FDC Receive (FDRX) Register (Offset 0x18)	287
11.7.6.5: FDC Transmit n (FDTXn) Registers (Offset 0x20 + 0x8*n)	287
11.8: MIPS® Trace	288
11.8.1: Processor Modes	289
11.8.2: Software Versus Hardware Control.....	289
11.8.3: Trace Information	289
11.8.4: Load/Store Address and Data Trace Information.....	290
11.8.5: Programmable Processor Trace Mode Options	291
11.8.6: Programmable Trace Information Options	291
11.8.6.1: User Data Trace	291
11.8.7: Enable Trace to Probe On-chip Memory.....	292
11.8.8: TCB Trigger.....	292
11.8.9: Cycle-by-Cycle Information	292
11.8.10: Instruction and Data Cache Miss Tracing	292
11.8.11: Performance Counter Tracing.....	293
11.8.12: Filtered Data Trace Mode	294
11.8.13: PC Tracing Off	294
11.8.14: TMOAS Handling	295
11.8.15: Memory-mapped Access to On-Chip Trace RAM.....	297
11.8.16: Core-Specific Event Inefficiency Tracing	299
11.8.17: Trace Message Format	299
11.8.18: Trace Word Format	299
11.9: PDtrace™ Registers (Software Control).....	300
11.10: Trace Control Block (TCB) Registers (Hardware Control).....	300
11.10.1: TCBCONTROLA Register.....	301
11.10.2: TCBCONTROLB Register.....	304
11.10.3: TCBDATA Register	308
11.10.4: TCBCONTROLC Register	309
11.10.5: TCBCONTROLE Register.....	310
11.10.6: TCBCONFIG Register (Reg 0).....	311
11.10.7: TCBTW Register (Reg 4).....	312
11.10.8: TCBRDP Register (Reg 5)	313
11.10.9: TCBWRP Register (Reg 6)	313
11.10.10: TCBSTP Register (Reg 7).....	314
11.10.11: TCBTRIGx Register (Reg 16-23)	314
11.10.12: Register Reset State	317
11.11: Enabling MIPS Trace.....	317
11.11.1: Trace Trigger from EJTAG Hardware Instruction/Data Breakpoints	317
11.11.2: Turning On PDtrace™ Trace	318
11.11.3: Turning Off PDtrace™ Trace	319
11.11.4: TCB Trace Enabling.....	320
11.11.5: Tracing a Reset Exception	320
11.12: TCB Trigger Logic	320
11.12.1: Trigger Units Overview.....	320
11.12.2: Trigger Source Unit	321
11.12.3: Trigger Control Units	322

11.12.4: Trigger Action Unit	322
11.12.5: Simultaneous Triggers	322
11.12.5.1: Prioritized Trigger Actions	322
11.12.5.2: OR'ed Trigger Actions	323
11.13: MIPS Trace Cycle-by-Cycle Behavior	323
11.13.1: FIFO Logic in PDtrace and TCB Modules	323
11.13.2: Handling of FIFO Overflow in the PDtrace Module	323
11.13.3: Handling of FIFO Overflow in the TCB.....	324
11.13.3.1: Probe Width and Clock-ratio Settings.....	324
11.13.4: Adding Cycle Accurate Information to the Trace.....	325
11.14: TCB On-Chip Trace Memory	325
11.14.1: On-Chip Trace Memory Size.....	325
11.14.2: Trace-From Mode	325
11.14.3: Trace-To Mode.....	326
Chapter 12: Instruction Set Overview	327
12.1: CPU Instruction Formats	327
12.2: Load and Store Instructions.....	328
12.2.1: Scheduling a Load Delay Slot.....	328
12.2.2: Defining Access Types.....	328
12.3: Computational Instructions	329
12.3.1: Cycle Timing for Multiply and Divide Instructions.....	330
12.4: Jump and Branch Instructions	330
12.4.1: Overview of Jump Instructions	330
12.4.2: Overview of Branch Instructions	331
12.5: Control Instructions.....	331
12.6: Coprocessor Instructions.....	331
Chapter 13: 74K™ Processor Core Instructions	333
13.1: Understanding the Instruction Descriptions.....	333
13.2: 74K™ Opcode Map	333
13.3: Floating Point Unit Instruction Format Encodings	338
13.4: MIPS32™ Instruction Set for the 74K™ Core	339
CACHE.....	347
LL	355
PREF.....	357
SC	361
SYNC	363
TLBR	367
TLBWI	369
WAIT	371
TLBWR.....	373
Chapter 14: MIPS16e™ Application-Specific Extension to the MIPS32® Instruction Set	375
14.1: Instruction Bit Encoding.....	375
14.2: Instruction Listing.....	377
Appendix A: References	381
Appendix B: Revision History	383

List of Figures

Figure 1.1: 74K™ Core Block Diagram	29
Figure 1.2: Address Translation During a Cache Access	33
Figure 2.1: 74K™ Core Pipeline	38
Figure 3.1: FPU Block Diagram	50
Figure 3.2: Single-Precision Floating-Point Format (S)	52
Figure 3.3: Double-Precision Floating-Point Format (D)	52
Figure 3.4: Word Fixed-Point Format (W)	54
Figure 3.5: Longword Fixed-Point Format (L)	54
Figure 3.6: Single Floating-Point or Word Fixed-Point Operand in an FPR	55
Figure 3.7: Double Floating-Point or Longword Fixed-Point Operand in an FPR	55
Figure 3.8: Effect of FPU Operations on the Format of Values Held in FPRs	56
Figure 3.9: FPU Word Load and Move-to Operations	57
Figure 3.10: FPU Doubleword Load and Move-to Operations	57
Figure 3.11: FIR Format	59
Figure 3.12: FCCR Format	60
Figure 3.13: FEXR Format	61
Figure 3.14: FENR Format	61
Figure 3.15: FCSR Format	62
Figure 3.16: FS/FO/FN Bits Influence on Multiply and Addition Results	65
Figure 3.17: Flushing to Nearest when Rounding Mode is Round to Nearest	66
Figure 3.18: FPU Pipeline	77
Figure 3.19: Arithmetic Pipeline Bypass Paths	79
Figure 4.1: MIPS32® DSP ASE Control Register (DSPControl) Format	81
Figure 5.1: Address Translation For Cache Access with TLB MMU	86
Figure 5.2: Address Translation For Cache Access with FM MMU	86
Figure 5.3: 74K™ Processor Core Virtual Memory Map	88
Figure 5.4: User Mode Virtual Address Space	89
Figure 5.5: Supervisor Mode Virtual Address Space	91
Figure 5.6: Kernel Mode Virtual Address Space	93
Figure 5.7: Debug Mode Virtual Address Space	95
Figure 5.8: JTLB Entry (Tag and Data)	97
Figure 5.9: Overview of Virtual-to-Physical Address Translation	100
Figure 5.10: 32-bit Virtual Address Translation	101
Figure 5.11: TLB Address Translation Flow in the 74K™ Processor Core	103
Figure 5.12: FM Memory Map (ERL=0) in the 74K™ Processor Core	105
Figure 5.13: FM Memory Map (ERL=1) in the 74K™ Processor Core	106
Figure 6.1: Interrupt Generation for Vectored Interrupt Mode	114
Figure 6.2: Interrupt Generation for External Interrupt Controller Interrupt Mode	116
Figure 6.3: DebugVectorAddr Register Format	126
Figure 6.4: General Exception Handler (HW)	139
Figure 6.5: General Exception Servicing Guidelines (SW)	140
Figure 6.6: TLB Miss Exception Handler (HW)	141
Figure 6.7: TLB Exception Servicing Guidelines (SW)	142
Figure 6.8: Reset and NMI Exception Handling and Servicing Guidelines	143
Figure 7.1: Register Format Color Coding of Access Field Types	151
Figure 7.2: Index Register Format	151
Figure 7.3: Random Register Format.....	152

Figure 7.4: EntryLo0, EntryLo1 Register Format	153
Figure 7.5: Context Register Format.....	154
Figure 7.6: ContextConfig Register Format	155
Figure 7.7: IUserLocal Register Format.....	156
Figure 7.8: PageMask Register Format	157
Figure 7.9: Wired Register Format	158
Figure 7.10: HWREna Register Format	159
Figure 7.11: BadVAddr Register Format.....	160
Figure 7.12: Count Register Format.....	161
Figure 7.13: EntryHi Register Format	161
Figure 7.14: Compare Register Format	162
Figure 7.15: Status Register Format.....	164
Figure 7.16: IntCtl Register Format.....	168
Figure 7.17: SRSCtl Register Format	170
Figure 7.18: SRSTMap Register Format.....	172
Figure 7.19: Cause Register Format	172
Figure 7.20: EPC Register Format	177
Figure 7.21: PRId Register Format	177
Figure 7.22: EBase Register Format.....	178
Figure 7.23: CDMMBase Register	179
Figure 7.24: Config Register Format.....	180
Figure 7.25: Config1 Register Format.....	181
Figure 7.26: Config2 Register Format.....	183
Figure 7.27: Config3 Register Format.....	184
Figure 7.28: Config6 Register Format.....	184
Figure 7.29: Config7 Register Format.....	187
Figure 7.30: WatchLo Register Format.....	190
Figure 7.31: WatchHi Register Format.....	191
Figure 7.32: Debug Register Format.....	192
Figure 7.33: TraceControl Register Format	195
Figure 7.34: TraceControl2 Register Format	197
Figure 7.35: User Trace Data1 / User Trace Data2 Register Format	200
Figure 7.36: TracelBPC Register Format	200
Figure 7.37: TraceDBPC Register Format	201
Figure 7.38: DEPC Register Format	202
Figure 7.39: TraceControl3 Register Format	203
Figure 7.40: PerfCtl0-3 Register Format.....	204
Figure 7.41: Performance Counter Count Register.....	209
Figure 7.42: CacheErr Register Format.....	210
Figure 7.43: ITagLo Register Format (ErrCtlWST = 0, ErrCtlSPR = 0).....	212
Figure 7.44: ITagLo Register Format (ErrCtlWST = 1, ErrCtlSPR = 0).....	212
Figure 7.45: ITagLo Register Format (ErrCtlWST = 0, ErrCtlSPR = 1).....	213
Figure 7.46: IDataLo Register Format.....	213
Figure 7.47: DTagLo Register Format (ErrCtlWST = 0, ErrCtlDYT = 0, ErrCtlSPR = 0)	214
Figure 7.48: DTagLo Register Format (ErrCtlWST = 1, ErrCtlDYT = 0, ErrCtlSPR = 0)	215
Figure 7.49: Field Descriptions for DTagLo-DYT Register.....	215
Figure 7.50: DTagLo-SPT (ErrCtlWST = 0, ErrCtlDYT = 0, ErrCtlSPR = 1).....	215
Figure 7.51: DTagLo Register Format (ErrCtlWST = 0, ErrCtlDYT = 0, ErrCtlSPR = 1)	216
Figure 7.52: DDataLo Register Format.....	216
Figure 7.53: L23TagLo Register Format.....	216
Figure 7.54: L23DataLo Register Format.....	217
Figure 7.55: ITagHi Register Format.....	217
Figure 7.56: IDataHi Register Format	217

Figure 7.57: DTagHi Register Format	218
Figure 7.58: L23DataHi Register Format	218
Figure 7.59: ErrorEPC Register Format	219
Figure 7.60: DeSave Register Format	219
Figure 9.1: Instruction Cache Organization	226
Figure 9.2: Data Cache Organization	228
Figure 11.1: Debug Control Register Format	240
Figure 11.2: IBS Register Format	250
Figure 11.3: IBAn Register Format	251
Figure 11.4: IBMn Register Format	252
Figure 11.5: IBASIDn Register Format	252
Figure 11.6: IBCn Register Format	253
Figure 11.7: DBS Register Format	254
Figure 11.8: DBAn Register Format	255
Figure 11.9: DBMn Register Format	255
Figure 11.10: DBASIDn Register Format	256
Figure 11.11: DBCn Register Format	256
Figure 11.12: DBVn Register Format	258
Figure 11.13: DBVHn Register Format	258
Figure 11.14: TAP Controller State Diagram	261
Figure 11.15: Concatenation of the EJTAG Address, Data and Control Registers	265
Figure 11.16: TDI to TDO Path When in Shift-DR State and FASTDATA Instruction is Selected	265
Figure 11.17: Device Identification Register Format	267
Figure 11.18: Implementation Register Format	268
Figure 11.19: EJTAG Control Register Format	270
Figure 11.20: Endian Formats for PAD Register	276
Figure 11.21: Fastdata Register Format	276
Figure 11.22: TAP Register PCsample Format	279
Figure 11.23: Fast Debug Channel Buffer Organization	282
Figure 11.24: FDC TAP Register Format	283
Figure 11.25: FDC Access Control and Status Register Format	284
Figure 11.26: FDC Configuration Register Format	285
Figure 11.27: FDC Status Register Format	286
Figure 11.28: FDC Receive Register Format	287
Figure 11.29: FDC Transmit Register Format	287
Figure 11.30: MIPS® Trace Functional Blocks in the 74K™ Core	289
Figure 11.31: A TMOAS Trace Record	296
Figure 11.32: TCBCONTROLA Register Format	301
Figure 11.33: TCBCONTROLB Register Format	304
Figure 11.34: TCBDATA Register Format	308
Figure 11.35: TCBCONTROLC Register Format	309
Figure 11.36: TCBCONTROLE Register Format	310
Figure 11.37: TCBCONFIG Register Format	311
Figure 11.38: TCBTW Register Format	313
Figure 11.39: TCBRDP Register Format	313
Figure 11.40: TCBWRP Register Format	314
Figure 11.41: TCBSTP Register Format	314
Figure 11.42: TCBTRIGx Register Format	315
Figure 11.43: TCB Trigger Processing Overview	321
Figure 12.1: Instruction Formats	328
Figure 13.1: Usage of Address Fields to Select Index and Way	347

List of Tables

Table 2.1: 74K™ Core Pipeline Stages Descriptions.....	38
Table 2.2: Execution Hazards.....	46
Table 2.3: Instruction Hazards.....	47
Table 2.4: Hazard Instruction Listing.....	47
Table 3.1: Parameters of Floating-Point Data Types.....	51
Table 3.2: Value of Single or Double Floating-Point Data Type Encoding.....	52
Table 3.3: Value Supplied When a New Quiet NaN is Created.....	54
Table 3.4: Coprocessor 1 Register Summary.....	58
Table 3.5: Read/Write Properties.....	58
Table 3.6: FIR Bit Field Descriptions.....	59
Table 3.7: FCCR Bit Field Descriptions.....	60
Table 3.8: FEXR Bit Field Descriptions.....	61
Table 3.9: FENR Bit Field Descriptions.....	61
Table 3.10: FCSR Bit Field Descriptions.....	62
Table 3.11: Cause, Enables, and Flags Definitions.....	63
Table 3.12: Rounding Mode Definitions.....	64
Table 3.13: Handling Denormalized Floating-point Numbers.....	64
Table 3.14: Zero Flushing for Tiny Results.....	65
Table 3.15: Handling of Denormalized Operand Values and Tiny Results Based on FS Bit Setting.....	65
Table 3.16: Handling of Tiny Intermediate Result Based on the FO and FS Bit Settings.....	65
Table 3.17: Handling of Tiny Final Result Based on FN and FS Bit Settings.....	66
Table 3.18: Recommended FS/FO/FN Settings.....	67
Table 3.19: FPU Data Transfer Instructions.....	69
Table 3.20: FPU Loads and Stores Using Register+Offset Address Mode.....	69
Table 3.21: FPU Loads and Stores Using Register+Register Address Mode.....	69
Table 3.22: FPU Move To and From Instructions.....	69
Table 3.23: FPU IEEE Arithmetic Operations.....	70
Table 3.24: FPU-Approximate Arithmetic Operations.....	70
Table 3.25: FPU Multiply-Accumulate Arithmetic Operations.....	71
Table 3.26: FPU Conversion Operations Using the FCSR Rounding Mode.....	71
Table 3.27: FPU Conversion Operations Using a Directed Rounding Mode.....	71
Table 3.28: FPU Formatted Operand Move Instruction.....	72
Table 3.29: FPU Conditional Move on True/False Instructions.....	72
Table 3.30: FPU Conditional Move on Zero/Non-Zero Instructions.....	72
Table 3.31: FPU Conditional Branch Instructions.....	73
Table 3.32: Deprecated FPU Conditional Branch Likely Instructions.....	73
Table 3.33: CPU Conditional Move on FPU True/False Instructions.....	73
Table 3.34: Result for Exceptions Not Trapped.....	74
Table 3.35: 74Kf Core FPU Latency and Repeat Rate.....	79
Table 4.1: MIPS® DSP ASE Control Register (DSPControl) Field Descriptions.....	82
Table 4.2: DSPControl ouflag Bits.....	83
Table 5.1: User Mode Segments.....	90
Table 5.2: Supervisor Mode Segments.....	92
Table 5.3: Kernel Mode Segments.....	94
Table 5.4: Physical Address and Cache Attributes for dseg, dmseg, and drseg.....	96
Table 5.5: CPU Access to drseg.....	96
Table 5.6: CPU Access to dmseg.....	96

Table 5.7: TLB Tag Entry Fields	98
Table 5.8: TLB Data Entry Fields	98
Table 5.9: Machine Check Exception	102
Table 5.10: TLB Instructions	103
Table 5.11: Cache Coherency Attributes	104
Table 5.12: Cacheability of Segments with Fixed Mapping Translation.....	104
Table 6.1: Priority of Exceptions	108
Table 6.2: Interrupt Modes	110
Table 6.3: Relative Interrupt Priority for Vectored Interrupt Mode.....	113
Table 6.4: Exception Vector Offsets for Vectored Interrupts.....	118
Table 6.5: Exception Vector Base Addresses, SI_UseExceptionBase = 0.....	120
Table 6.6: Exception Vector Base Addresses, SI_UseExceptionBase = 1.....	120
Table 6.8: Exception Vectors	121
Table 6.7: Exception Vector Offsets	121
Table 6.9: Value Stored in EPC, ErrorEPC, or DEPC on Exception.....	123
Table 6.10: Debug Exception Vector Addresses	126
Table 6.11: DebugVectorAddr Register Field Descriptions.....	126
Table 6.12: Register States an Interrupt Exception	130
Table 6.13: Register States on Watch Exception.....	131
Table 6.14: CP0 Register States on Address Exception Error.....	132
Table 6.15: CP0 Register States on TLB Refill Exception	132
Table 6.16: CP0 Register States on TLB Invalid Exception.....	133
Table 6.17: CP0 Register States on Cache Error Exception	133
Table 6.18: Register States on Coprocessor Unusable Exception	135
Table 6.19: Register States on Floating Point Exception.....	136
Table 6.20: Register States on TLB Modified Exception.....	138
Table 7.1: CP0 Registers in Alphabetical Order	145
Table 7.2: CP0 Registers in Numerical Order.....	146
Table 7.3: CP0 Registers Grouped by Function	149
Table 7.4: CP0 Register Field R/W Access Types.....	150
Table 7.6: Field Descriptions for Random Register	152
Table 7.5: Field Descriptions for Index Register	152
Table 7.7: Field Descriptions for EntryLo0-1 Register	153
Table 7.8: Cache Coherency Attributes encoding of C field of EntryLo0-1 and K0 field of Config Register	153
Table 7.9: Context Register Field Descriptions.....	154
Table 7.11: Recommended ContextConfig Values	156
Table 7.12: UserLocal Register Field Description.....	156
Table 7.10: ContextConfig Register Field Descriptions	156
Table 7.13: Field Descriptions for PageMask Register	157
Table 7.14: Field Descriptions for Wired Register.....	158
Table 7.15: Field Descriptions for HWREna Register	159
Table 7.16: BadVAddr Register Field Description.....	160
Table 7.17: Count Register Field Description	161
Table 7.18: Field Descriptions for EntryHi Register	161
Table 7.19: Compare Register Field Description	162
Table 7.20: Field Descriptions for Status Register.....	164
Table 7.21: Field Descriptions for IntCtl Register.....	168
Table 7.22: Field Descriptions for SRSCtl Register	170
Table 7.23: Sources for SRSCtl _{CSS} on an Exception or Interrupt.....	171
Table 7.24: Field Descriptions for SRSSMap Register.....	172
Table 7.25: Field Descriptions for Cause Register.....	173
Table 7.26: Exception Code values in ExcCode Field of Cause Register	175
Table 7.27: EPC Register Field Description.....	177

Table 7.28: Field Descriptions for PRId Register	177
Table 7.29: Field Descriptions for EBase Register	178
Table 7.30: CDMMBase Register Field Descriptions	179
Table 7.31: Field Descriptions for Config Register.....	180
Table 7.32: Field Descriptions for Config1 Register.....	182
Table 7.33: Field Descriptions for Config2 Register.....	183
Table 7.34: Field Descriptions for Config3 Register.....	184
Table 7.35: Field Descriptions for Config6 Register.....	185
Table 7.36: Field Descriptions for Config7 Register.....	187
Table 7.37: Field Descriptions for WatchLo0-3 Register.....	190
Table 7.38: Field Descriptions for WatchHi0-3 Register	191
Table 7.39: Field Descriptions for Debug Register	192
Table 7.40: TraceControl Register Field Descriptions	195
Table 7.41: TraceControl2 Register Field Descriptions	198
Table 7.42: UserTraceData1 / UserTraceData2 Register Field Descriptions	200
Table 7.43: TracelBPC Register Field Descriptions.....	200
Table 7.44: TraceDBPC Register Field Descriptions	201
Table 7.45: BreakPoint Control Modes: IBPC and DBPC.....	202
Table 7.46: DEPC Register Formats.....	202
Table 7.47: TraceControl3 Register Field Descriptions	203
Table 7.48: Field Descriptions for PerfCtl0-3 Register.....	204
Table 7.49: Performance Counter Events and Codes	204
Table 7.50: Performance Counter Count Register Field Descriptions	209
Table 7.51: Field Descriptions for ErrCtl Register	209
Table 7.52: Field Descriptions for CacheErr Register.....	210
Table 7.53: Field Descriptions for ITagLo Register.....	212
Table 7.55: Field Descriptions for ITagLo-SPR Register	213
Table 7.56: IDataLo Register Field Description	213
Table 7.54: Field Descriptions for ITagLo-WST Register.....	213
Table 7.57: Field Descriptions for DTagLo Register	214
Table 7.58: Field Descriptions for DTagLo-WST Register	215
Table 7.59: Field Descriptions for DTagLo-DYT Register.....	215
Table 7.60: Field Descriptions for DTagLo-SPT Register.....	216
Table 7.61: DDataLo Register Field Description.....	216
Table 7.62: L23DataLo Register Field Description	217
Table 7.63: Field Descriptions for ITagHi Register	217
Table 7.65: Field Descriptions for DTagHi Register.....	218
Table 7.66: L23DataHi Register Field Description	218
Table 7.64: IDataHi Register Field Description	218
Table 7.67: ErrorEPC Register Field Description.....	219
Table 7.68: DeSave Register Field Description	219
Table 9.1: Instruction Cache Attributes	225
Table 9.2: Data Cache Attributes	227
Table 9.3: Potential Virtual Aliasing Bits	228
Table 9.4: Way Selection Encoding, 4 Ways	233
Table 11.1: Debug Control Register Field Descriptions	240
Table 11.2: Overview of Status Register for Instruction Breakpoints.....	245
Table 11.3: Overview of Registers for Each Instruction Breakpoint.....	245
Table 11.4: Overview of Status Register for Data Breakpoints.....	245
Table 11.5: Overview of Registers for Each Data Breakpoint.....	246
Table 11.6: Rules for Update of BS Bits on Data Breakpoint Exceptions	249
Table 11.7: Addresses for Instruction Breakpoint Registers	250
Table 11.8: IBS Register Field Descriptions	251

Table 11.9: IBAn Register Field Descriptions	251
Table 11.10: IBMn Register Field Descriptions	252
Table 11.11: IBASIDn Register Field Descriptions	252
Table 11.12: BCn Register Field Descriptions	253
Table 11.13: Addresses for Data Breakpoint Registers	254
Table 11.14: DBS Register Field Descriptions	254
Table 11.15: DBAn Register Field Descriptions	255
Table 11.16: DBMn Register Field Descriptions	255
Table 11.17: DBASIDn Register Field Descriptions	256
Table 11.18: DBCn Register Field Descriptions	256
Table 11.19: DBVn Register Field Descriptions	258
Table 11.21: EJTAG Interface Pins	259
Table 11.20: DBVHn Register Field Descriptions	259
Table 11.22: Implemented EJTAG Instructions	263
Table 11.24: Implementation Register Descriptions	268
Table 11.23: Device Identification Register Field Descriptions	268
Table 11.25: EJTAG Control Register Descriptions	270
Table 11.26: Fastdata Register Field Description	276
Table 11.27: Operation of the FASTDATA Access	277
Table 11.28: FDC TAP Register Field Descriptions	283
Table 11.29: FDC Register Mapping	284
Table 11.30: FDC Access Control and Status Register Field Descriptions	284
Table 11.31: FDC Configuration Register Field Descriptions	285
Table 11.32: FDC Status Register Field Descriptions	286
Table 11.33: FDC Receive Register Field Descriptions	287
Table 11.35: FDTXn Address Decode	288
Table 11.34: FDC Transmit Register Field Descriptions	288
Table 11.36: TMOAS Trace Record Field Descriptions	296
Table 11.37: Mapping TCB Registers in drseg	298
Table 11.38: A List of Coprocessor 0 Trace Registers	300
Table 11.39: TCB EJTAG Registers	300
Table 11.40: Registers Selected by TCBCONTROLB	301
Table 11.41: TCBCONTROLA Register Field Descriptions	302
Table 11.42: TCBCONTROLB Register Field Descriptions	305
Table 11.43: Clock Ratio encoding of the CR field	308
Table 11.45: TCBCONTROLC Register Field Descriptions	309
Table 11.44: TCBDATA Register Field Descriptions	309
Table 11.46: TCBCONTROLE Register Field Descriptions	310
Table 11.47: TCBCONFIG Register Field Descriptions	311
Table 11.48: TCBTW Register Field Descriptions	313
Table 11.49: TCBRDP Register Field Descriptions	313
Table 11.50: TCBWRP Register Field Descriptions	314
Table 11.51: TCBSTP Register Field Descriptions	314
Table 11.52: TCBTRIGx Register Field Descriptions	315
Table 12.1: Byte Access Within a Doubleword	329
Table 13.1: Symbols Used in the Instruction Encoding Tables	333
Table 13.2: MIPS32 Encoding of the Opcode Field	334
Table 13.3: MIPS32 SPECIAL Opcode Encoding of Function Field	334
Table 13.4: MIPS32 REGIMM Encoding of rt Field	334
Table 13.5: MIPS32 SPECIAL2 Encoding of Function Field	335
Table 13.6: MIPS32 Special3 Encoding of Function Field for Release 2 of the Architecture	335
Table 13.7: MIPS32 MOVCI Encoding of tf Bit	335
Table 13.8: MIPS32 SRL Encoding of Shift/Rotate	335

Table 13.9: MIPS32 SRLV Encoding of Shift/Rotate	335
Table 13.10: MIPS32 BSHFLEncoding of sa Field	336
Table 13.11: MIPS32 COP0 Encoding of rs Field.....	336
Table 13.12: MIPS32COP0 Encoding of Function Field When rs=CO	336
Table 13.13: MIPS32 COP1 Encoding of rs Field.....	336
Table 13.14: MIPS32 COP1 Encoding of Function Field When rs=S	337
Table 13.15: MIPS32 COP1 Encoding of Function Field When rs=D	337
Table 13.16: MIPS32 COP1 Encoding of Function Field When rs=W or L.....	337
Table 13.17: MIPS32 COP1 Encoding of tf Bit When rs=S or D, Function=MOVCF.....	337
Table 13.18: COP1X Encoding of Function Field	338
Table 13.19: MIPS32 COP2 Encoding of rs Field.....	338
Table 13.20: Floating Point Unit Instruction Format Encodings	338
Table 13.21: 74K™ Core Instruction Set	339
Table 13.22: Usage of Effective Address.....	347
Table 13.23: Encoding of Bits[17:16] of CACHE Instruction	348
Table 13.24: Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST, DYT, SPR] Cleared.....	348
Table 13.25: Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST] Set, ErrCtl[DYT, SPR] Cleared ...	351
Table 13.26: Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[SPR] Set, ErrCtl[DYT, WST] Cleared ...	352
Table 13.27: Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[DYT] Set, ErrCtl[SPR, WST] Cleared ...	352
Table 13.28: Values of <i>hint</i> Field for PREF Instruction	357
Table 13.29: Encodings of the Bits[10:6] of the SYNC instruction; the SType Field.....	364
Table 14.1: Symbols Used in the Instruction Encoding Tables.....	375
Table 14.2: MIPS16e Encoding of the Opcode Field	376
Table 14.3: MIPS16e JAL(X) Encoding of the x Field.....	376
Table 14.4: MIPS16e SHIFT Encoding of the f Field	376
Table 14.5: MIPS16e RRI-A Encoding of the f Field.....	376
Table 14.6: MIPS16e I8 Encoding of the funct Field.....	376
Table 14.7: MIPS16e RRR Encoding of the f Field.....	377
Table 14.8: MIPS16e RR Encoding of the Funct Field	377
Table 14.9: MIPS16e I8 Encoding of the s Field when funct=SVRS	377
Table 14.10: MIPS16e RR Encoding of the ry Field when funct=J(AL)R(C).....	377
Table 14.11: MIPS16e RR Encoding of the ry Field when funct=CNVT	377
Table 14.12: MIPS16e Load and Store Instructions	378
Table 14.13: MIPS16e Save and Restore Instructions	378
Table 14.14: MIPS16e ALU Immediate Instructions	378
Table 14.15: MIPS16e Arithmetic Two or Three Operand Register Instructions	378
Table 14.16: MIPS16e Special Instructions	379
Table 14.17: MIPS16e Multiply and Divide Instructions.....	379
Table 14.18: MIPS16e Jump and Branch Instructions.....	379
Table 14.19: MIPS16e Shift Instructions.....	380

Introduction to the MIPS32® 74K™ Core Family

The 74K™ core from MIPS Technologies is a high-performance, low-power, 32-bit MIPS® RISC processor core family intended for use in custom system-on-silicon applications. The core is designed for semiconductor manufacturing companies, ASIC developers, and system OEMs who want to rapidly integrate their own custom logic and peripherals with a high-performance RISC processor. A 74K core is fully synthesizable to allow maximum flexibility; it is highly portable across processes and can easily be integrated into full system-on-silicon designs. This allows developers to focus their attention on end-user specific characteristics of their product.

The 74K core is ideally positioned to support new products for emerging segments of the digital consumer, network, systems, and information management markets, enabling new tailored solutions for embedded applications.

The 74K family has two members: the MIPS32® 74Kc™ core and the MIPS32® 74Kf™ core:

- The 74Kc 32-bit RISC core is optimized for high-performance applications.
- The 74Kf core adds an IEEE-754 compliant floating point unit.

The core implements the MIPS32 Release 2 Instruction Set Architecture (ISA). It also implements the following Application-Specific Extensions (ASEs):

- The MIPS® DSP Application-Specific Extension (ASE) is optimized for signal-processing applications.
- The MIPS16e™ Application-Specific Extension (ASE) is optimized for code compression.

The 74K core achieves its high performance through the implementation of advanced superscalar and out-of-order dispatch techniques in a deeply pipelined implementation of the MIPS32® architecture. The superscalar dispatch allows the core to dispatch two instructions per cycle to two pipelines: a 15-stage AGEN pipeline that executes all load/store and control transfer instructions, and a 14-stage ALU pipeline that executes all the rest of the instructions (arithmetic, logic, and general computations). The out-of-order approach allows each pipeline to operate independently and select from a pool of instructions for dispatch; and to ensure the availability of two instructions for dispatch, twice that number of instructions are fetched every cycle from an instruction cache. The 74K core also implements sophisticated branch prediction techniques that minimize the cost of a mispredicted branch in such a deeply pipelined core.

On the 74K core, instruction caches are configurable as 0, 16, 32, or 64 KB, and data caches are configurable as 0, 16, 32, or 64 KB in size. Each cache is organized as a 4-way set-associative data structure. The 74K core supports prefetching of sequential cache lines on instruction cache misses. The extent of prefetching can be configured via software to prefetch between 0 and 2 additional lines. The data cache features non-blocking load misses and can handle up to 9 outstanding load misses in up to 4 unique cache lines. On a cache miss, the processor can continue executing instructions while the load data is being fetched until a dependent instruction is reached. Both caches are virtually indexed and physically tagged. The MMU of the 74K core may contain a 4-entry instruction TLB (ITLB) and a dual-entry joint TLB (JTLB), with variable page sizes. The JTLB can be configured to have 16, 32, 48, or 64 dual entries. Optionally, the TLB can be replaced with a simplified fixed mapping (FM) translation mechanism for applications that do not require the full capabilities of a TLB.

The Multiply Divide Unit (MDU) is fully pipelined and supports a maximum issue rate of one 32x32 multiply (MUL/MULT/MULTU), multiply-add (MADD/MADDU), or multiply-subtract (MSUB/MSUBU) operation per clock.

The basic Enhanced JTAG (EJTAG) features provide CPU run control with stop, single-stepping, and re-start, as well as software breakpoints using the SDBBP instruction. Support for connection to an external EJTAG probe through the Test Access Port (TAP) and the Fast Debug Channel mechanism for efficient data transfer are also included. Instruction and data virtual address hardware breakpoints can be optionally included. In addition, optional PDtrace™ hardware provides the ability to trace program flow, load/store addresses, and load/store data. Several run-time options exist for the level of information which is traced, including tracing only when in specific processor modes (e.g., User Mode or Kernel Mode).

The bus interface implements the Open Core Protocol (OCP) using 64-bit read and write data buses. The bus interface may operate at the same rate or at a lower clock rate than the core itself.

1.1 74K™ Core Features

1.1.1 Pipeline

- Superscalar, dual-issue core supports two integer execution pipes
 - 15-stage AGEN pipe: supports load/store, control transfer, and conditional move instructions
 - 14-stage ALU pipe: supports all other arithmetic, logic, and computation instructions
- Out-of-order integer instruction dispatch
 - Selects one of eight instructions in each pipe
- Multiply Divide Unit
 - Offshoot of the ALU Pipe
 - Maximum issue rate of one 32x32 multiply per clock
 - Early-in divide control. Minimum 11, maximum 50-cycle clock latency on divide
- Dual- Issue Floating Point Unit supports two pipes (74Kf only)
 - Arithmetic pipe
 - To/From or Data transfer pipe
 - Floating point instruction dispatch is maintained in-order
- Dynamic branch/return prediction
 - Majority predictor featuring three tables of 256 entries, each with global history
 - 8-entry return prediction stack

1.1.2 Instruction Set

- MIPS32 Compatible Instruction Set
 - Multiply-add and multiply-subtract instructions (MADD, MADDU, MSUB, MSUBU)
 - Targeted multiply instruction (MUL)
 - Zero and one detect instructions (CLZ, CLO)
 - Wait instruction (WAIT)
 - Conditional move instructions (MOVZ, MOVN)
 - Prefetch instruction (PREF)
- MIPS32® Enhanced Architecture (Release 2) Features
 - Vectored interrupts and support for an external interrupt controller
 - Programmable exception vector base
 - Atomic interrupt enable/disable
 - GPR shadow sets: 1 to 4 sets are supported
 - Bit field manipulation instructions
- MIPS DSP ASE Rev 2
 - Fractional data types (Q15, Q31)
 - Saturating arithmetic
 - SIMD instructions operated on 2x16b or 4x8b simultaneously
 - 3 additional pairs of accumulator registers
- MIPS16e™ Application-Specific Extension
 - 16-bit encodings of 32-bit instructions to improve code density
 - Special PC-relative instructions for efficient loading of addresses and constants
 - Data type conversion instructions (ZEB, SEB, ZEH, SEH)
 - Compact jumps (JRC, JALRC)
 - Stack frame set-up and tear-down macro instructions (SAVE and RESTORE)
- Floating Point Instruction support (74Kf)
 - IEEE-754 compliant floating point unit

Introduction to the MIPS32® 74K™ Core Family

- Compliant with MIPS 64-bit FPU standards
- Supports single and double precision datatypes
- CorExtend® User-Defined Instruction capability
 - Support for the CorExtend feature allows users to define and add instructions to the core
 - 3, 5, or multi-cycle latencies supported
 - Source operands from register, immediate field, or local state
 - Destination to a register or local state
 - Interface to multiply-divide unit, allowing sharing of accumulator registers

1.1.3 Memory Management, Caches, and Scratchpad Memory

- Standard Memory Management Unit
 - 16, 32, 48, or 64 dual-entry MIPS32-style JTLB with variable page sizes
 - 4-entry instruction TLB
- Optional Memory Management Unit
 - Simple Fixed Mapping Translation (FM)
 - Address spaces mapped using register bits
- Caches
 - Individually configurable instruction and data caches
 - Instruction cache sizes of 0, 16, 32, or 64 KB
 - Data cache sizes of 0, 16, 32, or 64 KB
 - Data cache access widths can be configured to be 64b or 128b
 - 4-way set associative
 - 256-bit (32-byte) cache line size
 - Configurable instruction cache line prefetch, to fetch between 0 to 2 additional cache lines on a miss
 - Non-blocking data cache
 - Up to 4 data cache line misses or 9 unique load misses
 - Data cache supports write-back with write-allocation and write-through without write-allocation
 - Virtually indexed, physically tagged

- Cache line locking support
- Configurable support for parity
- Support for front-side external L2 cache
- Configurable support for elimination of aliases in 32 and 64KB data caches
- Independent Instruction and Data Scratch Pad RAMs
 - Address range of 4K - 1MB supported
 - 64-bit OCP interfaces for external access

1.1.4 Interfaces

- OCP Interface
 - 32b address and 64b data
 - Core/bus ratios of 1, 1.5, 2, 2.5, 3, 3.5, 4, 5, and 10 are supported
 - Supports bursts of 4x64b
 - 4-entry write buffer that handles eviction data, write-through, uncached, and uncached accelerated store data
 - Simple Byte Enable mode allows easy bridging to other bus standards
 - Extensions for management of front-side L2 cache
 - Critical data first and sub-block ordering support
- IEEE standard JTAG interface

1.1.5 Power Control

- No minimum frequency
- Power-down mode (triggered by WAIT instruction)
- Support for software-controlled clock divider
- Support for extensive use of fine-grain clock gating

1.1.6 Debug

- EJTAG Debug Support via JTAG interface
 - CPU control with start, stop, and single stepping
 - Software breakpoints via the SDBBP instruction

Introduction to the MIPS32® 74K™ Core Family

- Optional hardware breakpoints on virtual addresses: 4 instruction and 2 data breakpoints
- Test Access Port (TAP) facilitates high-speed download of application code
- Fast Debug Channel with configurable FIFO depth for efficient data transfer to and from probe
- Optional MIPS PDtrace™ hardware to enable real-time tracing of executed code

1.1.7 Other

- R4000 Style Privileged Resource Architecture
 - Count/Compare registers for real-time timer interrupts
 - Instruction and data watch registers for software breakpoints
- Relocatable Bootstrap Exception Vector support
 - Support for hardware selectable exception base in a multi-core environment

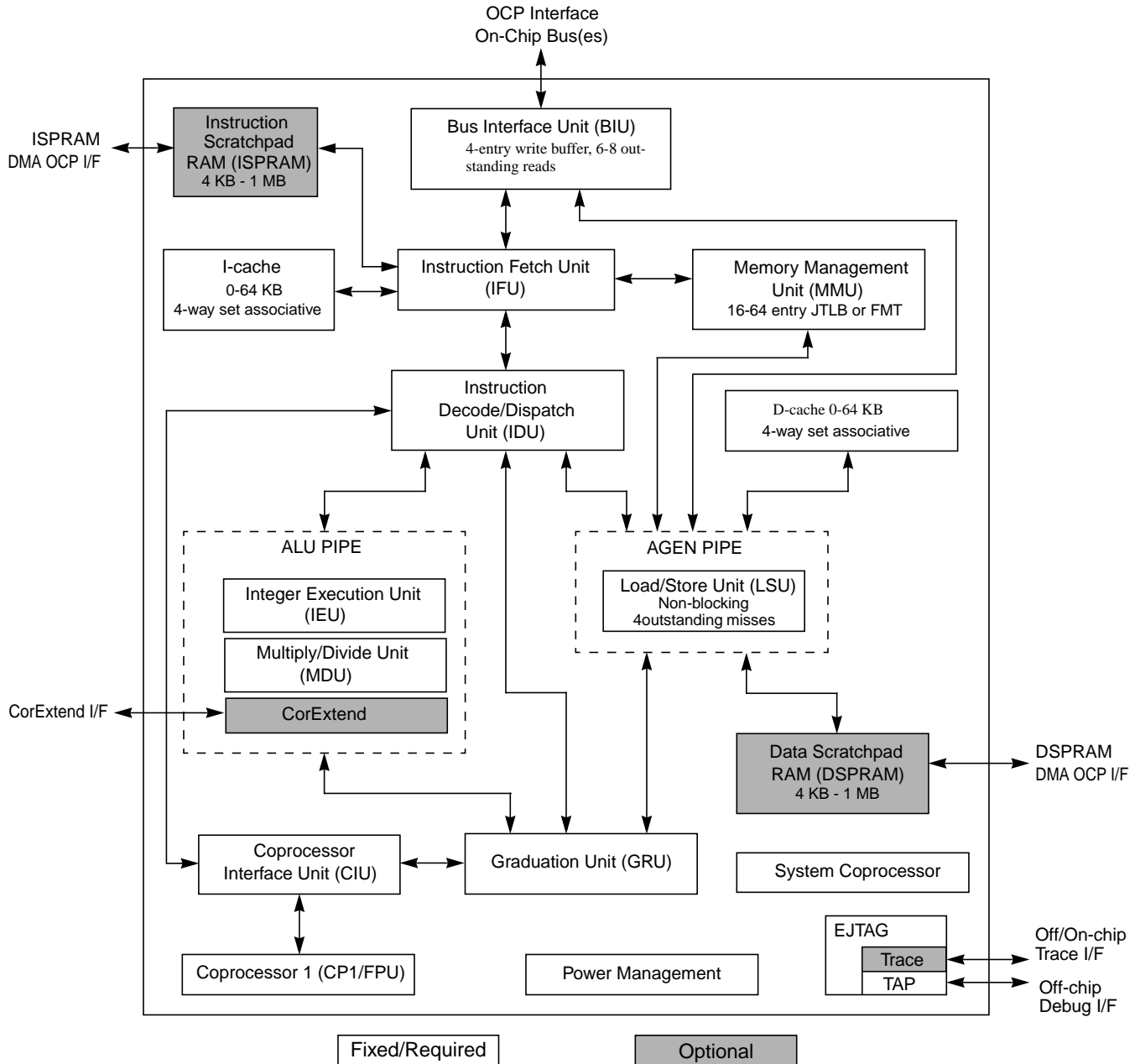
1.2 74K™ Core Block Diagram

The 74K core contains a number of blocks, shown in [Figure 1.1](#). The major blocks are:

- Instruction Fetch Unit (IFU)
- Instruction Cache (I-cache)
- Instruction Decode and Dispatch Unit (IDU)
- Instruction Execution Unit (IEU)
- Multiply/Divide Unit (MDU)
- CorExtend® User Defined Instructions (UDI)
- System Control Coprocessor (CP0)
- Memory Management Unit (MMU)
- Load Store Unit (LSU)
- Data Cache (D-cache)
- Graduation unit (GRU)
- Bus Interface Unit (BIU)
- Coprocessor Interface unit (CIU) (only in 74Kf)
- Floating Point Unit (FPU) (only in 74Kf)

- Power Management
- Enhanced JTAG (EJTAG) Controller

Figure 1.1 74K™ Core Block Diagram



The functional blocks shown in Figure 1.1 are described in the following subsections.

1.2.1 Instruction Fetch Unit (IFU)

The Instruction Fetch Unit (IFU) is responsible for fetching instructions from the instruction cache/memory and providing them to the IDU. The IFU can fetch up to 4 instructions at a time from an aligned fetch address. The IFU has a 4-entry microTLB which is used to translate the virtual fetch address into the physical fetch address. This translated physical address is used to compare against tags in the instruction cache to determine a hit.

The IFU uses majority branch prediction based on gshare predictors. There are three, 256-entry Branch History Tables that are indexed by different combinations of the instruction PC and Global History. The majority of these three predictions are used to determine the predicted direction of a conditional branch. The IFU also has an 8-entry Return Prediction Stack to predict subroutine return addresses.

There is a 12-entry Instruction Buffer to decouple the instruction fetch from execution. Up to 4 instructions at a time can be written into this buffer, but a maximum of 2 instructions at a time can be read from this buffer by the IDU.

The 74K core includes supports for the MIPS16e ASE. This ASE improves code density through the use of 16-bit encoding of many MIPS32 instructions plus some MIPS16e-specific instructions. The IFU contains the logic for the handling of MIPS16e instructions.

1.2.2 Instruction Cache

The instruction cache is an on-chip memory block of 0/16/32/64 KB, with 4-way associativity. The instruction cache is virtually indexed and physically tagged, allowing the virtual-to-physical address translation to occur in parallel with the cache access, rather than having to wait for the physical address translation.

A tag entry holds 21 bits of physical address, a valid bit, a lock bit, and an optional parity bit. There are 7 precode bits per instruction pair, making a total of 28 bits per tag entry. The data array line consists of 256 bits (8 MIPS32 instructions) of data. Each instruction doubleword (64 bits) has 8 bits of byte parity. The IFU interface consists of 128 bits (4 MIPS32 instructions) with 16 bits of parity. The LRU replacement bits (6 bits) are shared among the 4 ways of the data and tag array and are stored in a separate array.

The core supports instruction cache locking. Cache locking allows critical code to be locked into the cache on a “per-line” basis, enabling the system designer to maximize the efficiency of the system cache. Cache locking is always available on all instruction cache entries.

The instruction cache also supports cache line prefetching on miss. This feature can be configured by software to prefetch anywhere between 0 and 2 lines. The control for this prefetch resides in the Coprocessor Register field *Config7_{PREF}*. Refer to [Appendix 7, “Config7 \(CP0 Register 16, Select 7\): CPU-specific Configuration” on page 187](#) for details on programming this feature. The default setting will fetch one additional cache line. This can be modified by the user of the core. Increasing the number of cache lines prefetched to 2 will typically provide increased performance for applications with a high I-cache miss rate. Decreasing the number of cache lines prefetched to 0 may be appropriate for applications with very low I-cache miss rates, or when there is a desire to reduce memory bandwidth for power or other reasons.

1.2.3 Instruction Decode/Dispatch Unit (IDU)

This unit is responsible for receiving instructions from the IFU and dispatching them out-of-order to the execution units when their operands and required resources are available. Up to two instructions per cycle can be received in-order from the IFU. As the IDU dispatches instructions speculatively and out-of-order, results from these instructions are stored in temporary storage buffers referred to as *completion buffers*. The IDU assigns the completion buffer ID, as well as a separate instruction ID for each instruction. The instruction is also renamed by looking up in a Rename

Map, and the source registers are replaced (if necessary) by completion buffer IDs of producer instructions, so that operands may be bypassed as soon as possible.

Renamed instructions are assigned to one of two pipes (ALU or AGEN) and written into the Decode and Dispatch Queue (DDQ) for that pipe. The oldest instruction that has all the operands ready and meets all resource requirements is dispatched independently to the corresponding pipe. Instructions may be dispatched out-of-order relative to program order. Dispatched instructions do not stall in the pipe and write the results into the completion buffer.

The IDU also keeps track of the progress of the instruction through the pipe, updating the availability of operands in the Rename Map and in all dependent instructions in the DDQ.

The IDU also writes the instruction ID, completion buffer ID, and related information into structures in the Graduation Unit (GRU). The GRU reads instructions and corresponding results from the completion buffer, graduates the instructions, and updates the architectural state of the machine.

1.2.4 Instruction Execution Unit (IEU)

The Instruction Execution Unit implements the entire ALU pipe and parts of the AGEN pipe (parts of the AGEN pipe also reside in the LSU). The IEU provides data inputs to the multiply/divide unit (MDU) and CorExtend units and receives outputs from them. The LSU, MDU, and CorExtend Unit are described in subsequent sections.

The architecturally-defined General Purpose Registers (GPRs) reside in the IEU. In addition to these, the IEU also contains the completion buffers (CBs) used to store computed results. There is a dedicated completion buffer per pipeline. Each pipe of the IEU has input bypass muxes to select data from the GPRs, CBs, or from the pipeline when data forwarding is required. The IEU also contains the output muxes that generate final output data. In addition, the IEU has certain pipe-specific execution units described below.

ALU Pipe

The ALU pipe contains the ALU for performing arithmetic and logical operations, the Shifter, and the Leading Zero/One detector. The ALU pipe implements a subset of the DSP ASE instructions.

AGEN Pipe

The AGEN pipe contains the adder required for address computation in case of load/store and control transfer instructions. It also contains all the branch resolution logic.

The IEU also provides data inputs to the multiply/divide unit (MDU) and CorExtend units and receives outputs from them.

1.2.5 Multiply Divide Unit (MDU)

The multiply/divide unit implements the multiply and divide operations. This unit also executes multiply class instructions in the DSP ASE.

The MDU consists of a pipelined 32 \times 32 multiplier, result/accumulation registers (HI and LO), a divide state machine, and the necessary multiplexors and control logic. The MDU supports execution of one multiply or multiply-accumulate operation every clock cycle. Divide operations are implemented with a simple 1 bit per clock radix 2 iterative SRT algorithm.

1.2.6 CorExtend® User Defined Instructions (UDIs)

This module contains support for CorExtend user-defined instructions. These instructions are defined at build-time for the 74K core. This feature makes 15 instructions in the opcode map available for user-defined customer use, and the latency of each instruction can be selected to be 3, 5, or >5 cycles. A CorExtend instruction can operate on any one or two general purpose registers or immediate data contained within the instruction, and can write the result of each instruction back to a general purpose register or a local register. Further details regarding CorExtend can be found in the *CorExtend® Instruction Integrator's Guide for MIPS32 74K™ Cores* [9].

Refer to [Table 13.5](#) for a specification of the opcode map available for user-defined instructions.

1.2.7 Load Store Unit (LSU)

The Load Store Unit, as the name implies, is primarily responsible for the implementation of Load/Store instructions. In addition to the Load/Store instructions, it also implements the Prefetch, CACHE, and some other special instructions. The LSU contains all the control logic for the data cache. In addition it contains several holding structures for address and data information. These are primarily a 4-cache-line Fill Store Buffer (FSB) to service cache line misses, a 9-entry Load Data Queue (LDQ) to support 9 load misses, a 14-entry Load Store Queue (LSQ), and a 10-entry Load Store Graduation Buffer (LSGB) to hold information for Loads and Store instructions in flight.

1.2.8 System Control Coprocessor (CP0)

In the MIPS architecture, CP0 is responsible for virtual-to-physical address translation, cache protocols, the exception control system, the processor's diagnostics capability, operating mode selection (kernel vs. user mode), power management, and the enabling/disabling of interrupts. Configuration information such as cache size, set associativity, and the presence of build-time options are available by accessing the CP0 registers. Refer to [Chapter 7, "CP0 Registers of the 74K™ Core"](#) for more information on the CP0 registers. Refer to [Chapter 11, "EJTAG Debug Support in the 74K™ Core"](#) for more information on EJTAG debug registers.

1.2.9 Memory Management Unit (MMU)

The 74K core contains an MMU that interfaces between the execution unit and the cache controllers. Although the 74K core implements a 32-bit architecture, the Memory Management Unit (MMU) is modeled after the MMU found in the 64-bit R4000 family, as defined by the MIPS32 architecture.

By default, the 74K core implements its MMU based on a Translation Lookaside Buffer (TLB). The TLB consists of two translation buffers: a configurable 16/32/48/64 dual-ported, dual-entry fully associative Joint TLB (JTLB) and a 4-entry fully associative Instruction TLB (ITLB).

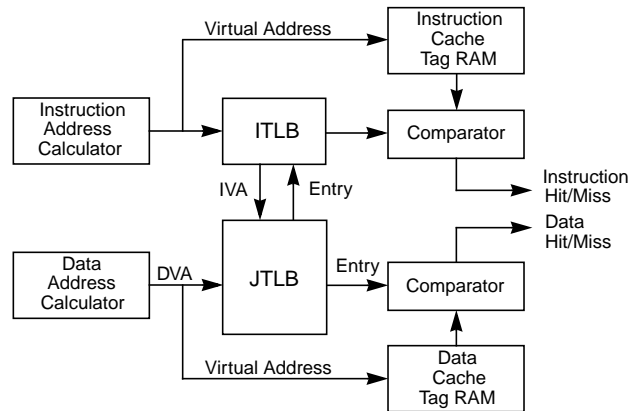
The ITLB (Instruction micro TLB), which contains a subset of the JTLB, is managed by the hardware and is not visible to software. When translating an instruction fetch address, the ITLB is accessed first. If there is no matching entry, the JTLB is used to translate the address and refill the ITLB. If the entry is not found in the JTLB, then an exception is taken.

In order to translate an address for a data access, the MMU looks in the JTLB directly, as there is no Data micro TLB present in the 74K core. The JTLB is dual-ported so as to avoid contention between instruction and data accesses.

The core optionally implements an FMT-based MMU instead of a TLB-based MMU. The FMT replaces the ITLB and JTLB blocks in [Figure 1.2](#). The FMT performs a simple translation to obtain the physical address from the virtual address. Refer to [Chapter 5, "Memory Management of the 74K™ Core"](#) for more information on the FMT.

Figure 1.2 shows how the address translation mechanism interacts with cache accesses.

Figure 1.2 Address Translation During a Cache Access



1.2.10 Data Cache

The data cache is an on-chip memory array of 0/16/32/64 KBytes. The cache is virtually indexed and physically tagged, allowing the virtual-to-physical address translation to occur in parallel with the cache access. The tag holds 20 bits of the physical address, a valid bit, a lock bit, and optionally a parity bit. For each entry there is also a corresponding 20-bit virtual tag. The virtual tag is used to determine the way selected on a cache access. As a result of this, data selection does not have to wait for the JTLB translation to complete.

The data entry is configurable to hold 64 or 128 bits of data per way, with optional parity per byte. The 128-bit option allows faster refill and evictions of the data cache and is recommended where data cache bandwidth is critical. In the 64-bit configuration, there are 4 data entries per tag entry, and in the 128-bit configuration there are 2 data entries for each tag entry. The tag and data entries exist for each way of the cache. A separate array holds the dirty and LRU bits (6b), dirty bits (4b), and optional dirty parity bits (4b) for all 4 ways. A data cache locking mechanism is available that is similar to the mechanism in the instruction cache.

The physical data cache memory must be byte-writable to support sub-word store operations. The LRU/dirty bit array must be bit-writable.

1.2.11 Scratchpad RAM

The 74K core allows blocks of scratchpad RAM to be attached to the load/store and instruction units. These allow low-latency access to a fixed block of memory.

1.2.12 Graduation Unit (GRU)

The Graduation Unit is responsible for graduating instructions in-order, even though they might have been dispatched and have completed their result computation out-of-order. It does so by reading data and associated control information from the completion buffers in program order and committing them into architectural state in that same program order. It then releases any completion buffers and resources used by these instructions. The GRU is also responsible for evaluating the exception conditions reported by execution units and taking the appropriate exception.

1.2.13 Bus Interface Unit (BIU)

The Bus Interface Unit controls the external interface signals. Additionally, it contains the implementation of a collapsing write buffer. This buffer is used to merge write-through transactions and to gather multiple writes from dirty line evictions and uncached accelerated stores. The write buffer consists of 4, 32-byte entries.

1.2.14 Coprocessor Interface Unit (CIU)

The coprocessor interface unit is responsible for maintaining an in-order interface between the integer core and the Floating Point Unit (FPU). The FPU is described in further detail in [Chapter 3, “Floating-Point Unit of the 74Kf™ Core”](#).

1.2.15 Power Management

The core offers a number of power-management features, including low-power design, active power management, and power-down modes of operation. The core is a static design that supports a WAIT instruction designed to signal the rest of the device that execution and clocking should be halted, hence reducing system power consumption during idle periods. The core provides two mechanisms for system-level, low-power support:

- Register-controlled power management
- Instruction-controlled power management

In register-controlled power management mode, the core provides three bits in the CP0 Status register for software control of the power-management function, and allows interrupts to be serviced even when the core is in power-down mode. In instruction-controlled power-down mode, execution of the WAIT instruction is used to invoke low-power mode.

Refer to [Chapter 10, “Power Management in the 74K™ Core”](#) for more information on power management.

1.2.16 EJTAG Debug

All cores provide basic EJTAG support with debug mode, run control, single-step, and software breakpoint instruction (SDBBP) as part of the core. These features allow basic software debugging of user and kernel code. A TAP controller is also included, enabling communication between an EJTAG probe and the CPU through a dedicated port. This provides the capability of debugging without debug code in the application, and for download of application code to the system.

An optional EJTAG feature is hardware breakpoints. A 74K core may have four instruction breakpoints and two data breakpoints, or no breakpoints. The hardware instruction breakpoints can be configured to generate a debug exception when an instruction is executed anywhere in the virtual address space. Bit mask and Address Space Identifier (ASID) values may apply in the address compare. These breakpoints are not limited to code in RAM, like the software instruction breakpoint (SDBBP). The data breakpoints can be configured to generate a debug exception on a data transaction. The data transaction may be qualified with both virtual address, data value, size, and load/store transaction type. Bit mask and ASID values may apply in the address compare, and byte mask may apply in the value compare.

An optional MIPS Trace feature has been added to aid software debugging. The trace logic implements PDtrace version 6, which allows tracing of the PC, load/store address, load/store data, and performance counter data, and also provides information about processor pipeline inefficiencies. The trace information can be stored to either an on-chip trace memory or an off-chip trace probe. The optional on-chip trace memory can be configured in various sizes.

These trace features provide a powerful software debugging mechanism. Refer to [Chapter 11, “EJTAG Debug Support in the 74K™ Core”](#) for more information on the EJTAG and tracing features.

Pipeline of the 74K™ Core

The 74K processor core is a superscalar processor capable of issuing two integer instructions every clock cycle. The integer dual issue is achieved through two pipelines referred to as the ALU and AGEN pipelines. The 15-stage AGEN pipeline implements the memory transfer and control transfer class of instructions, and the 14-stage ALU pipeline implements all the rest of the instructions involving arithmetic, logic, and computation. The pipelines allow the processor to achieve high frequency while minimizing device complexity, reducing both cost and power consumption.

This chapter contains the following sections:

- [Section 2.1 “Integer Pipeline Description”](#)
- [Section 2.2 “Programming the 74K Core”](#)
- [Section 2.3 “Hazards”](#)

2.1 Integer Pipeline Description

The two primary integer pipelines, the AGEN and ALU pipelines, are supported by a common front and back end. The common front end comprises of instruction fetch, decode, and dispatch, and represents the first eight stages of the pipeline, except in MIPS16e mode, where three more stages are present. The common back end comprises of instruction graduation and consumes the last two stages. The intermediate stages represent the instruction execution specific portions of the ALU and AGEN pipelines.

The two major pipelines are further made up of multiple mini-pipelines. The front end stages form the IFU and IDU pipelines. The backend forms the GRU pipeline. The functionality representing the IFU, IDU, and GRU pipelines reside in the units with the same name. The instruction execution specific functionality of the ALU and AGEN pipeline resides in the IEU and the LSU.

In addition to the above mentioned pipelines, there are a few other pipelines that exist in the 74K processor core. The Multiply Divide Unit (MDU) attaches to the ALU pipeline and is an offshoot of the ALU pipeline. The Floating Point Unit (FPU) attaches to the common front end and has a separate pipeline that is described later. There is also a separate post-graduation memory pipeline that comes into existence only for load/store instructions. This pipeline resides entirely in the Load Store Unit.

[Figure 2.1](#) shows the stages of the 74K processor core pipeline. The pipeline stages shown in the figure are described in [Table 2.1](#) and in the following subsections.

Figure 2.1 74K™ Core Pipeline

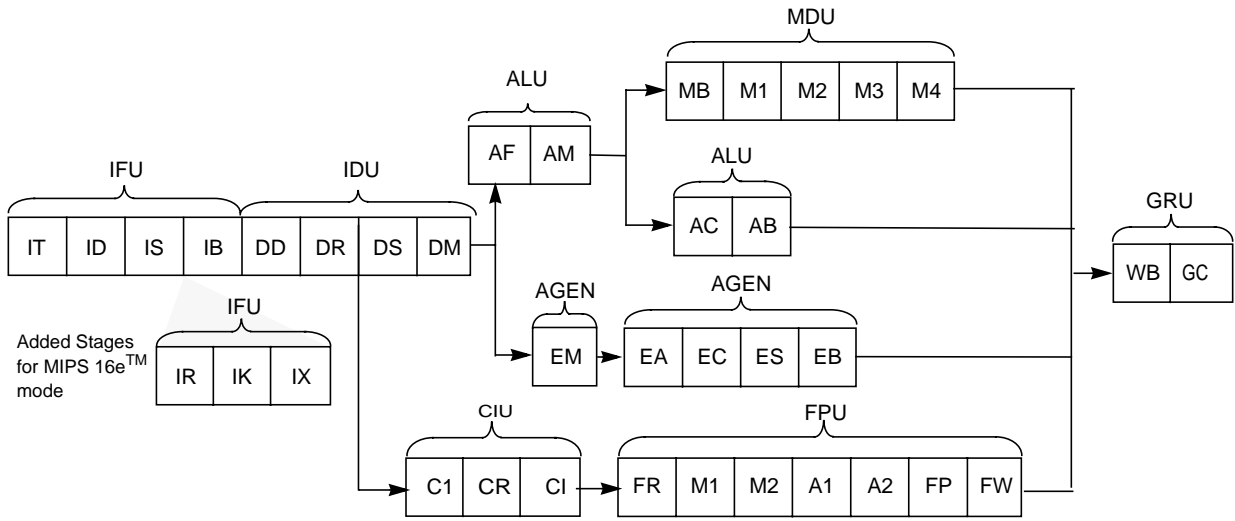


Table 2.1 74K™ Core Pipeline Stages Descriptions

Stage	Description	Stage	Description
IT	I-cache Tag read; ITLB Lookup; BHT Lookup	AF	ALU register File read
ID	I-cache Data read; Tag Compare	AM	ALU operand Mux select
IS	Way Select; Target Calculation Start	AC	ALU Compute
IB	Write Fetch Buffer; Target Calculation Done	AB	ALU Results Bypass
IR	MIPS16 Recode	(MB,M[1-4])	Booth recode, multiply stages 1-4
IK	MIPS16 Branch Decode and validate	EM	Execute operands Mux select
IX	MIPS16 macro expansion	EA	Address Generation; JTLB Access Start; Branch operand select
DD	Instruction Decode and Register Rename (read RMap)	EC	D-cache Read; JTLB access continue; Branch Compare
DR	Write RMap; Write DDQs; Issue to CP1/CP2	ES	PTag Compare start; Vhint selects Data; Branch Redirect
DS	Select 1 instruction from DDQ0 and 1 from DDQ1	EB	Load Align; Load data bypass; Ptag validate data select Branch/Jump link data pipe forward.
DM	Finish instruction selection and update DDQ entries.	WB	Completion buffer write; Exceptions determined
C1	Adjust InOrder instruction queue write pointer	GC	Graduation Complete
CR	Read instruction from InOrder instruction queue		
CI	InOrder instruction dispatch to Coprocessor1		

2.1.1 IFU Pipeline

2.1.1.1 IT - Instruction Cache Tag Access

The Instruction Fetch Unit (IFU) determines the address to be accessed in the cycle prior to IT. The I-cache tags, the branch history table (BHT), and the Return Prediction Stack (RPS) are accessed in this stage. The tag data contains some precode bits for each instruction pair indicating instruction type, etc. The I-cache is virtually indexed and the initial

lookup proceeds without any physical address bits. In parallel, the ITLB is accessed in this cycle in order to obtain the virtual to physical address translation.

In addition to this Instruction fetch, related Watch exception conditions and EJTAG breakpoint matches are also determined.

The functionality corresponding to this stage resides entirely in the IFU.

2.1.1.2 ID - Instruction Cache Data Access

In this stage, the I-cache arrays are accessed and 128 bits of instruction are read out for each cache way. While the I-cache data is being fetched, the tag outputs are compared against the physical address obtained from the ITLB lookup and the Refill Buffer tags, and one of the 4 associative ways or Refill Buffer entries is determined. The Refill Buffers represent the I-cache misses currently being serviced.

ITag parity error detection is also performed in this stage.

The functionality corresponding to this stage resides entirely in the IFU.

2.1.1.3 IS - Instruction Select

In this cycle, data from the I-cache or Refill Buffer is selected based on results of the tag compare in the previous cycle.

In case the instruction fetched from the I-cache is a Branch type instruction that is predicted taken, the computation of the target address is also started in this cycle.

The functionality corresponding to this stage resides entirely in the IFU.

2.1.1.4 IR - Instruction Recode

This stage comes into existence in the main pipeline only for MIPS16e instructions. MIPS16 instructions are recoded in this stage into MIPS32 equivalent instructions.

The functionality corresponding to this stage resides entirely in the IFU.

2.1.1.5 IK - Instruction

This stage also comes into existence in the main pipeline only for MIPS16e instructions. In this stage the instructions are decoded for control transfer instruction information and validated. The candidate branch/jump instruction is determined out of the 4 possible instructions in this stage.

The functionality corresponding to this stage resides entirely in the IFU.

2.1.1.6 IX - Instruction Macro Expansion

This stage also comes into existence in the main pipeline only for MIPS16e instructions. If a MIPS16e macro instruction is detected, it is expanded into multiple MIPS32 instructions in this stage. This stage is bypassed if there are no macro instructions.

The functionality corresponding to this stage resides entirely in the IFU.

2.1.1.7 IB - Instruction Buffer

The MIPS16e and MIPS32 pipelines converge at this stage. Up to 4 instructions from either the IS or IX stages are written into the Fetch Buffers. A maximum of 2 instructions can be read from any Fetch Buffer. The write of the Fetch Buffer can be bypassed in the same cycle.

The functionality corresponding to this stage resides entirely in the IFU.

2.1.2 Instruction Decode Unit Pipeline

2.1.2.1 DD - Dispatch Decode

In this stage the IDU receives up to 2 instructions and decodes them. In parallel with the decode operation the Rename Map is looked up to determine whether the source operands are in the Register File, Completion Buffer/pipeline, or Unavailable (pending long latency operation).

Lookups are done in this stage only if there are enough Completion Buffer resources available. Each instruction that crosses this stage receives a Completion Buffer ID (CBID) and Instruction ID. The CBID determines the location in AGCB or ALCB where the execution unit can write the results. The Instruction ID is a sequential ID that uniquely determines the age of the instruction in the pipe between the DD stage and Graduation.

A branch and its delay slot are always presented together by the IFU into the IDU.

This stage is also the synchronization point for several hazard preventing serializing operations such as EHB, branch mispredict redirect resolution, etc.

The functionality corresponding to this stage resides entirely in the IDU.

2.1.2.2 DR - Dispatch Rename

The instructions from the DD stage will arrive in the DR stage and any dependencies on older instructions and intra-dependencies among the two incoming instructions are resolved. The RenameMap is updated with the new destination CBID (if applicable) for both instructions.

The instructions are written into one of the two out-of-order dispatch queues, DDQ0 and DDQ1. The two queues are 6 entries deep. Each queue is associated with an execution pipe. The queues are divided as follows:

- DDQ1 (AGEN Pipe): Supports Memory Transfer (Load/Store), Control Transfer, and Conditional Move Instructions.
- DDQ0 (ALU Pipe): Supports all other instructions.

Coprocessor 1 instructions are handed off to the Coprocessor Interface Unit (CIU) in this stage. If necessary, some instructions (such as Coprocessor 1 loads and Coprocessor 1 branches, etc.) are written into DDQ1 as well the CIU.

The functionality corresponding to this stage resides entirely in the IDU.

2.1.2.3 DS - Dispatch Select

In this stage the IDU selects 1 instruction out of each DDQ to send down the corresponding execution pipe. It reads a counter associated with each source register to determine the availability for dispatch. Each register (CBID) has its own count down counters that is triggered by its producer that has been dispatched earlier. These counters are used only if the register value is to be bypassed from within the pipeline/Completion Buffer. Older available

instructions are given higher priority for selection. An instruction issues out-of-order only when preceding instructions are held up by source register or resource dependencies.

The functionality corresponding to this stage resides entirely in the IDU.

2.1.2.4 DM - DDQ Mux

The result vector from the selection in the previous stage is used to completely read out the selected DDQ entry. The destination CBID of the selected instruction is broadcast to the DDQ and to the RenameMap so that they can start their count down timers. The initial value of the timers is determined in DD stage based on the instruction/execution pipe properties.

In addition for instructions destined for the AGEN Pipe, the access of the Register File and Completion Buffers is started in this stage.

The functionality corresponding to this stage resides largely in the IDU and to a small extent in the IEU.

2.1.3 ALU Pipeline

2.1.3.1 AF - ALU Pipe Register File Read

In this stage the Register File or Completion Buffer read is performed for the source operands of the instructions selected in the DM stage. This functionality requires 2 read ports on the Register File and Completion Buffer, corresponding to two sources for the instruction in ALU pipe. Only 1 instruction can be in the AF stage in the ALU pipe at any time.

The functionality corresponding to this stage resides entirely in the IEU.

2.1.3.2 AM - ALU Pipe Operand Bypass Select Mux, This stage has the final operand bypass muxes for the ALU pipe. The functionality corresponding to this stage resides entirely in the IEU.

2.1.3.3 AC - ALU Compute

The 74K core's ALU is pipelined. Some ALU instructions complete the operation and bypass the results in this cycle. These instructions are referred to as single-cycle ops and they include all logical instructions (AND, ANDI, OR, ORI, XOR, XORI, LUI), some shift instructions (SLL sa<=8, SRL 31<=sa<=25), and some arithmetic instructions (ADD rt=0, ADDU rt=0, SLT, SLTI, SLTU, SLTIU, SEH, SEB, ZEH, ZEB). In addition, add instructions (ADD, ADDU, ADDI, ADDIU) complete the operation and bypass results to the ALU pipe in this cycle. Add instructions cannot bypass results to the AGEN pipe in this cycle, but will bypass to the AGEN pipe in the subsequent cycle. All other ALU instructions take 2 cycles to execute and bypass the results to both pipes. This stage corresponds to the first stage of execution for those instructions. The Multiply and Divide class of instructions start their execution in a separate MDU pipe, and the first stage of that pipe is aligned with this stage.

The functionality corresponding to this stage resides entirely in the IEU.

2.1.3.4 AB - ALU Bypass

This is the second stage of the ALU pipeline. Instructions whose latency exceeds a single cycle perform their second cycle of computation in this cycle.

All ALU operations can bypass their results from this stage to the muxes in AM, EM, and EA stages. All exception information is gathered in this stage to be written into the completion buffer.

The functionality corresponding to this stage resides entirely in the IEU.

2.1.4 MDU Pipeline

2.1.4.1 MB - Multiplier Booth Recode.

The MDU performs Booth recoding on the input operand in this stage.

2.1.4.2 M1- M3 Multiplier Array

These stages represent the actual core of the multiplier array.

2.1.4.3 M4 - Multiply Add

This stage is used to perform the add or subtract operation in the case of complex multiply-add or multiply-subtract type of instructions.

2.1.5 AGEN Pipeline

2.1.5.1 EM - Execute Operand Bypass Select Mux

In the presence of shadow set registers, this stage accommodates the final selectors of register data from the different GPRs. In addition, this stage accommodates the bypass muxes for the Load/Store class instructions in the AGEN Pipe.

The functionality corresponding to this stage resides entirely in the IEU

2.1.5.2 EA - Execute and Address Generate

Load/Store Instructions

The effective address calculation for Load/Store class instructions is computed in this cycle. The JTLB access is also started in this cycle for Load/Store type instructions. The JTLB is accessed in parallel using source operands and a fast compare algorithm. This enables the JTLB access to start one cycle earlier than would otherwise be possible. It also makes it possible for the 74K core to access the JTLB inline and avoid a DTLB.

The functionality corresponding to this adder resides entirely in the IEU. The JTLB resides in the MMU.

Control Transfer Instructions

In the case of Control Transfer instructions, the effective redirect address computation is started in this cycle. The bypass muxes for source register operands for this class of instructions are also present in this stage. It is to be noted that there are two sets of bypass muxes in the AGEN Pipe. The first set is in the EM stage and is used for Load/Store instructions. The second set in the EA stage is used for Control Transfer instructions. While it would have been possible for Control transfer instructions to use the bypass muxes from EM and execute the branch in EA, performance simulation has shown that it would significantly reduce overall performance, as it increases the producer/consumer latency between most integer instructions and branches.

The functionality corresponding to Control Transfer Instructions for this stage resides entirely in the IEU.

2.1.5.3 EC - Execute and Cache Access

Load/Store Instructions

The Data Cache is accessed using the effective address calculated in the preceding EA stage. The local buffers in the Load Store Unit (LSU) are also compared against this effective address. The JTLB access is continued in this cycle. The Data cache tags are also accessed in this cycle. The Dtag entry will contain a physical tag and virtual tag.

The functionality corresponding to D-cache and Dtag resides entirely in the LSU. The JTLB resides in the MMU.

Control Transfer Instructions

Conditional Branch and jump instructions are resolved in this cycle. The branch comparison is done and compared against the predicted path. The branch instructions compute the alternate address and redirect the IFU if needed. Since branch instructions can be issued out-of-order the branch execution unit keeps track of the age of the last redirected branch. If the new resolution results in a mispredict the age of the new branch instruction is compared against the age of the last redirected branch that has not yet graduated. Only if the new branch is older than the previous branch the IFU will be redirected again. Register indirect jumps which have been predicted with the Return Stack are also compared against the real register value in this cycle. If there is a mispredict the correct target is sent to the IFU.

The functionality corresponding to Control Transfer Instructions for this stage resides entirely in the IEU.

2.1.5.4 ES - Execute and Cache Second

Load/Store Instructions

The JTLB access is completed in this cycle. The virtual tag is compared against the effective address in this cycle. The tag compare for the various internal LSU buffers is also completed in this cycle. Alignment of the D-cache data from all four ways is also done in parallel in this cycle. If a store to load bypass situation is detected by virtue of a load effective address matching one of the local buffers in the LSU (LSQ or FSB), those buffers are read in this stage.

The functionality corresponding to D-cache, DTag, LSQ and other buffers resides entirely in the LSU. The JTLB resides in the MMU.

Control Transfer Instructions

The result of all the branch decisions is prioritized and communicated to the IFU in this cycle. In addition to the branch unit redirects, there can be redirects from the graduating instructions. These redirects are prioritized over the branch redirects, and the final redirect is sent over to the IFU in this cycle. This redirect causes the IFU to kill its current fetch stream and all instructions in the IFU. A new fetch will be started with the newly received target address.

The functionality corresponding to Control Transfer Instructions for this stage resides entirely in the IEU.

2.1.5.5 EB - Execute and Cache Data Bypass

Load/Store Instructions

The D-cache data is selected based on the virtual tag comparison. The final Hit/Miss determination is done by the end of this cycle based on the physical tag comparison with the JTLB output. Alignment of data in case of data return from LSU buffers is done in this cycle. In case of a hit, the selection of data from the appropriate source is also completed in this cycle. The data return to the ALU bypass muxes in EM and EA is done in this cycle.

The functionality corresponding to the D-cache, DTag, LSQ, and other buffers resides entirely in the LSU. The JTLB resides in the MMU.

Pipeline of the 74K™ Core

All exception detection and prioritization within the LSU is completed in this cycle for the AGEN Pipe.

Control Transfer Instructions

In the case of branch and link or jump and link instructions, the link register update information is carried forward in the pipe to be written into the completion buffer. This data is eventually written into the Register File when the branch/jump instruction graduates. This data is written into the AGEN completion buffer and it uses the same write port as Load/Store instructions.

The functionality corresponding to this stage for Control Transfer Instructions resides entirely in the IEU.

2.1.6 GRU Pipeline

2.1.6.1 WB - Writeback

The AGEN Pipe writes the AGEN completion buffer (AGCB) in this stage. The ALU pipe and the MDU pipe write into the ALU completion buffer (ALCB) in this stage. All units will also have written their exception information at this stage. The highest priority pipeline exception status will be available for each instruction at this point in the completion buffer structure. In addition, the oldest 2 entries that have completed execution are identified as candidates for graduation in the subsequent stage.

The buffers and their controls reside across multiple units (LSU, IEU, GRU), depending on the functionality.

2.1.6.2 GC - Graduation Commit

This stage is the final stage of the graduation pipeline. In this stage, the two oldest ready instructions that have been identified in the previous stage are graduated. This implies that the result data corresponding to these two instructions is committed to the architecturally visible GPR if no flush and redirect due to exceptions or other special cases is required. If a redirect is required, the graduation/exception logic in the GRU will send the appropriate redirect information to the IFU in this stage.

A select class of privileged instructions such as MTC0, TLB operations, and CACHE instructions are actually executed at graduation. These instructions are sent to the different units for execution in this stage.

The functionality corresponding to this stage resides entirely in the GRU.

2.2 Programming the 74K Core

For guidelines on programming the 74K core and a better understanding of the impact of the pipeline on the software programmer, refer to the document titled *Programming the MIPS32® 1074K™ Coherent Processing System Family* (MD00750).

2.3 Hazards

In general, the 74K core ensures that instructions are executed following a fully sequential program model. Each instruction in the program sees the results of the previous instruction. There are some deviations to this model. These deviations are referred to as *hazards*.

Prior to Release 2 of the MIPS™ Architecture, hazards (primarily CP0 hazards) were relegated to implementation-dependent cycle-based solutions, primarily based on the SSNOP instruction. The 74K core implements an out-of-order dispatch technique that is incompatible with this concept of allocating cycles through fixed instruction spacing.

Release 2 defines new instructions which act as explicit barriers that eliminate hazards. The new instructions have been added in such a way that they are backward-compatible with existing MIPS processors.

The 74K core family requires that the programmer implement the hazard barrier instructions as defined in Release 2 of the architecture. This does not typically impact an application programmer and is relevant primarily to privileged software. The following sections describe the types of hazards that are addressed. The hazard descriptions in the subsequent sections are here to help the user in identifying hazards in his code.

2.3.1 Types of Hazards

With one exception, all hazards were eliminated in Release 1 of the Architecture for unprivileged software. The exception occurs when unprivileged software writes a new instruction sequence and then wishes to jump to it. Such an operation remained a hazard, and is addressed by the capabilities of Release 2.

In privileged software, there are two different types of hazards: *execution hazards* and *instruction hazards*. Both are defined below.

2.3.1.1 Execution Hazards, Execution hazards are those created by the execution of one instruction, and seen by the execution of another instruction. Table 2.2 lists possible execution hazards and whether they can be resolved via setting of the IHB bit in the CP0 Config7 register..

Table 2.2 Execution Hazards

Producer	→	Consumer	Hazard On	Does Config7.IHB=1 resolve this Hazard?
TLBWR, TLBWI	→	Load/store using new TLB entry	TLB entry	No
MTC0	→	Load/store affected by new state	WatchHi WatchLo	No
MTC0	→	MFC0	Any CP0 register	No
MTC0	→	EI/DI	Status	Yes
MTC0	→	RDHWR \$3	Count	No
MTC0	→	Coprocessor instruction execution depends on the new value of Status _{CU}	Status _{CU}	No
MTC0	→	ERET	EPC DEPC ErrorEPC	Yes
MTC0	→	ERET	Status	Yes
EI, DI	→	Interrupted instruction	Status _{IE}	No
MTC0	→	Interrupted instruction	Status	No
MTC0	→	User-defined instruction	Status _{ERL} Status _{EXL}	No
MTC0	→	Interrupted Instruction	Status _{IM} (Cause _{IP})	No
TLBR	→	MFC0	EntryHi, EntryLo0, EntryLo1, PageMask	Yes
TLBP	→	MFC0	Index	Yes
MTC0	→	TLBR	Index (not EntryHi)	No
MTC0	→	TLBWI TLBWR	EntryHi	No
MTC0	→	TLBP Load/store affected by new state	EntryHi _{ASID}	No
MTC0	→	TLBWI	Index	No
MTC0	→	TLBWR	Random (not Index)	No
MTC0 ContextConfig	→	MFC0	Context ContextConfig	Yes
MTC0	→	RDPGPR WRPGPR	SRSCtl _{PSS}	No

Table 2.2 Execution Hazards (Continued)

Producer	→	Consumer	Hazard On	Does Config7.IHB=1 resolve this Hazard?
MTC0	→	Instruction not seeing a Timer Interrupt	Compare update that clears Timer Interrupt	No
MTC0	→	Instruction affected by change	Any other CP0 register	No
CACHE	→	MFC0	<i>TagHi, TagLo, DataHi, DataLo</i>	Yes

2.3.1.2 Instruction Hazards, Instruction hazards are those created by the execution of one instruction, and seen by the instruction fetch of another instruction. Table 2.3 lists instruction hazards. Because the fetch unit is decoupled from the execution unit, these hazards are rather large. The use of a hazard barrier instruction is required for reliable clearing of instruction hazards.

Table 2.3 Instruction Hazards

Producer	→	Consumer	Hazard On
TLBWR, TLBWI	→	Instruction fetch using new TLB entry	TLB entry
MTC0	→	Instruction fetch seeing the new value including: <ul style="list-style-type: none"> • change to ERL followed by an instruction fetch from the useg segment and • change to ERL or EXL followed by a Watch exception 	<i>Status</i>
MTC0	→	Instruction fetch seeing the new value	<i>EntryHi_{ASID}</i>
MTC0	→	Instruction fetch seeing the new value	<i>WatchHi WatchLo</i>
MTC0 (write to Config7)	→	JR, JALR seeing the new value of <i>IHB</i> of <i>Config7</i>	<i>IHB</i> bit of <i>Config7</i>
Instruction stream write via CACHE	→	Instruction fetch seeing the new instruction stream	Cache entries
Instruction stream write via store	→	Instruction fetch seeing the new instruction stream	Cache entries

2.3.2 Instruction Listing

Table 2.4 lists the instructions designed to eliminate hazards. See the document titled *MIPS32® Architecture Reference Manual Volume II: The MIPS32® Instruction Set* (MD00084) for a more detailed description of these instructions.

Table 2.4 Hazard Instruction Listing

Mnemonic	Function
EHB	Clear execution hazard
ERET	Clears both execution and instruction hazards
JALR.HB	Clears both execution and instruction hazards
JR.HB	Clears both execution and instruction hazards

Table 2.4 Hazard Instruction Listing (Continued)

Mnemonic	Function
SYNCI	Synchronize caches after instruction stream write

2.3.2.1 Instruction Encoding

The EHB instruction is encoded using a variant of the NOP/SSNOP encoding. This encoding was chosen for compatibility with the Release 1 SSNOP instruction, such that existing software may be modified to be compatible with both Release 1 and Release 2 implementations. See the EHB instruction description for additional information.

The JALR.HB and JR.HB instructions are encoding using bit 10 of the *hint* field of the JALR and JR instructions. These encodings were chosen for compatibility with existing MIPS implementations, including many which pre-date the MIPS architecture. Because a pipeline flush clears hazards on most early implementations, the JALR.HB or JR.HB instructions can be included in existing software for backward and forward compatibility. See the JALR.HB and JR.HB instructions for additional information.

The SYNCI instruction is encoded using a new encoding of the REGIMM opcode. This encoding was chosen because it causes a Reserved Instruction exception on all Release 1 implementations. As such, kernel software running on processors that don't implement Release 2 can emulate the function using the CACHE instruction.

2.3.3 Eliminating Hazards

In order to eliminate hazards, use one of the instructions listed in [Table 2.4](#) between the producer and consumer of the hazard. Execution hazards can be removed by using the EHB, JALR.HB, or JR.HB instructions. Instruction hazards can be removed by using the JALR.HB or JR.HB instructions, in conjunction with the SYNCI instruction.

Floating-Point Unit of the 74Kf™ Core

This chapter describes the MIPS64® Floating-Point Unit (FPU) included in the 74Kf core. This chapter contains the following sections:

- Section 3.1 “Features Overview”
- Section 3.2 “Enabling the Floating-Point Coprocessor”
- Section 3.3 “Data Formats”
- Section 3.4 “Floating-Point General Registers”
- Section 3.5 “Floating-Point Control Registers”
- Section 3.6 “Instruction Overview”
- Section 3.7 “Exceptions”
- Section 3.8 “Pipeline and Performance”

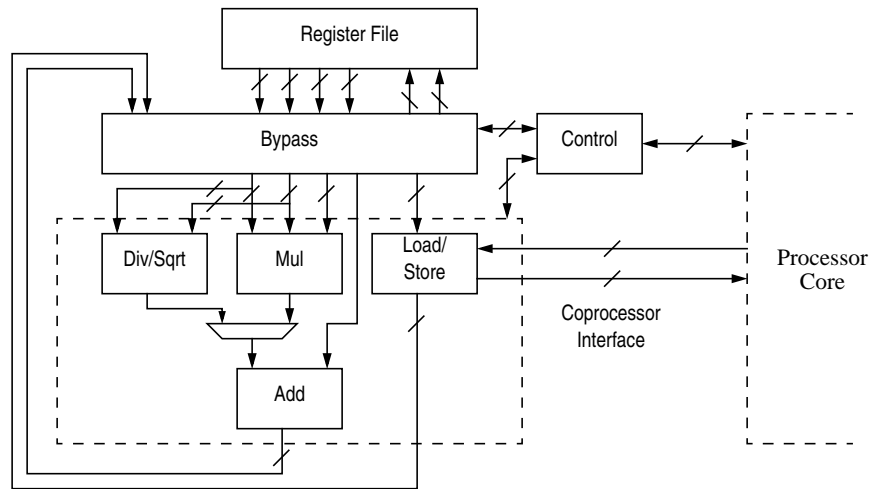
3.1 Features Overview

The FPU is provided via Coprocessor 1. Together with its dedicated system software, the FPU fully complies with the ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*. The MIPS architecture supports the recommendations of IEEE Standard 754, and the coprocessor implements a precise exception model. The key features of the FPU are listed below.

- Full 64-bit operation is implemented in both the register file and functional units.
- A 32-bit Floating-Point Control Register controls the operation of the FPU, and monitors condition codes and exception conditions.
- Like the main processor core, Coprocessor 1 is programmed and operated using a Load/Store instruction set. The processor core communicates with Coprocessor 1 using a dedicated coprocessor interface. The FPU functions as an autonomous unit. The hardware is completely interlocked such that, when writing software, the programmer does not have to worry about inserting delay slots after loads and between dependent instructions.
- Additional arithmetic operations not specified by IEEE Standard 754 (for example, reciprocal and reciprocal square root) are specified by the MIPS architecture and are implemented by the FPU. In order to achieve low latency counts, these instructions satisfy more relaxed precision requirements.
- The MIPS architecture further specifies compound multiply-add instructions. These instructions meet the IEEE accuracy specification, where the result is numerically identical to an equivalent computation using multiply, add, subtract, or negate instructions.

Figure 3.1 depicts a block diagram of the FPU.

Figure 3.1 FPU Block Diagram



The MIPS architecture is designed such that a combination of hardware and software can be used to implement the architecture. The 74K core FPU can operate on numbers within a specific range (in general, the IEEE normalized numbers), but it relies on a software handler to operate on numbers not handled by the FPU hardware (in general, the IEEE denormalized numbers). Supported number ranges for different instructions are described later in this chapter. A fast Flush To Zero mode is provided to optimize performance for cases where IEEE denormalized operands and results are not supported by hardware. The fast Flush to Zero mode is enabled through the CPI *FCSR* register; use of this mode is recommended for best performance.

3.1.1 IEEE Standard 754

The IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*, is referred to in this chapter as “IEEE Standard 754”. IEEE Standard 754 defines the following:

- Floating-point data types
- The basic arithmetic, comparison, and conversion operations
- A computational model

IEEE Standard 754 does not define specific processing resources nor does it define an instruction set.

For more information about this standard, see the IEEE web page at <http://stdsbbs.ieee.org/>.

3.2 Enabling the Floating-Point Coprocessor

Coprocessor 1 is enabled through the CU1 bit in the CP0 *Status* register. When Coprocessor 1 is not enabled, any attempt to execute a floating-point instruction causes a Coprocessor Unusable exception.

3.3 Data Formats

The FPU provides both floating-point and fixed-point data types, which are described below:

- The single- and double-precision floating-point data types are those specified by IEEE Standard 754.
- The fixed-point types are signed integers provided by the CPU architecture.

3.3.1 Floating-Point Formats

The FPU provides the following two floating-point formats:

- a 32-bit single-precision floating point (type S, shown in [Figure 3.2](#))
- a 64-bit double-precision floating point (type D, shown in [Figure 3.3](#))

The floating-point data types represent numeric values as well as the following special entities:

- Two infinities, $+\infty$ and $-\infty$
- Signaling non-numbers (SNaNs)
- Quiet non-numbers (QNaNs)
- Numbers of the form: $(-1)^s 2^E b_0.b_1 b_2..b_{p-1}$, where:
 - $s = 0$ or 1
 - $E =$ any integer between E_{\min} and E_{\max} , inclusive
 - $b_i = 0$ or 1 (the high bit, b_0 , is to the left of the binary point)
 - p is the signed-magnitude precision

The single and double floating-point data types are composed of three fields—sign, exponent, fraction—whose sizes are listed in [Table 3.1](#).

Table 3.1 Parameters of Floating-Point Data Types

Parameter	Single	Double
Bits of mantissa precision, p	24	53
Maximum exponent, E_{\max}	+127	+1023
Minimum exponent, E_{\min}	-126	-1022
Exponent <i>bias</i>	+127	+1023
Bits in exponent field, e	8	11
Representation of b_0 integer bit	hidden	hidden
Bits in fraction field, f	23	52
Total format width in bits	32	64
Magnitude of largest representable number	3.4028234664e+38	1.7976931349e+308

Table 3.1 Parameters of Floating-Point Data Types (Continued)

Parameter	Single	Double
Magnitude of smallest normalized representable number	1.1754943508e-38	2.2250738585e-308

Layouts of these three fields are shown in Figures 3.2 and 3.3 below. The fields are:

- 1-bit sign, s
- Biased exponent, $e = E + bias$
- Binary fraction, $f = .b_1 b_2 \dots b_{p-1}$ (the b_0 bit is *hidden*; it is not recorded)

Figure 3.2 Single-Precision Floating-Point Format (S)



Figure 3.3 Double-Precision Floating-Point Format (D)



Values are encoded in the specified format using the unbiased exponent, fraction, and sign values listed in Table 3.2. The high-order bit of the Fraction field, identified as b_1 , is also important for NaNs.

Table 3.2 Value of Single or Double Floating-Point Data Type Encoding

Unbiased E	f	s	b_1	Value V	Type of Value	Typical Single Bit Pattern ¹	Typical Double Bit Pattern ¹
$E_{max} + 1$	$\neq 0$		1	SNaN	Signaling NaN	0x7fffffff	0x7fffffff ffffffff
			0	QNaN	Quiet NaN	0x7fbfffffff	0x7ff7ffff ffffffff
$E_{max} + 1$	0	1		$-\infty$	Minus infinity	0xff800000	0xfff00000 00000000
		0		$+\infty$	Plus infinity	0x7f800000	0x7ff00000 00000000
E_{max} to E_{min}		1		$-(2^E)(1.f)$	Negative normalized number	0x80800000 through 0xff7fffff	0x80100000 00000000 through 0xffefffff ffffffff
		0		$+(2^E)(1.f)$	Positive normalized number	0x00800000 through 0x7f7fffff	0x00100000 00000000 through 0x7fefffff ffffffff
$E_{min} - 1$	$\neq 0$	1		$-(2^{E_{min}})(0.f)$	Negative denormalized number	0x807fffff	0x800fffff ffffffff
		0		$+(2^{E_{min}})(0.f)$	Positive denormalized number	0x007fffff	0x00fffff ffffffff
$E_{min} - 1$	0	1		- 0	Negative zero	0x80000000	0x80000000 00000000
		0		+ 0	positive zero	0x00000000	0x00000000 00000000

1. The “Typical” nature of the bit patterns for the NaN and denormalized values reflects the fact that the sign might have either value (NaN) and that the fraction field might have any non-zero value (both). As such, the bit patterns shown are one value in a class of potential values that represent these special values.

3.3.1.1 Normalized and Denormalized Numbers

For single and double data types, each representable nonzero numerical value has just one encoding; numbers are kept in normalized form. The high-order bit of the p-bit mantissa, which lies to the left of the binary point, is “hidden,” and not recorded in the *Fraction* field. The encoding rules permit the value of this bit to be determined by looking at the value of the exponent. When the unbiased exponent is in the range E_{min} to E_{max} , inclusive, the number is normalized and the hidden bit must be 1. If the numeric value cannot be normalized because the exponent would be less than E_{min} , then the representation is denormalized, the encoded number has an exponent of $E_{min} - 1$, and the hidden bit has the value 0. Plus and minus zero are special cases that are not regarded as denormalized values.

3.3.1.2 Reserved Operand Values—Infinity and NaN

A floating-point operation can signal IEEE exception conditions, such as those caused by uninitialized variables, violations of mathematical rules, or results that cannot be represented. If a program does not trap IEEE exception conditions, a computation that encounters any of these conditions proceeds without trapping but generates a result indicating that an exceptional condition arose during the computation. To permit this case, each floating-point format defines representations (listed in [Table 3.2](#)) for plus infinity ($+\infty$), minus infinity ($-\infty$), quiet non-numbers (QNaN), and signaling non-numbers (SNaN).

3.3.1.3 Infinity and Beyond

Infinity represents a number with magnitude too large to be represented in the given format; it represents a magnitude overflow during a computation. A correctly signed ∞ is generated as the default result in division by zero operations and some cases of overflow as described in [Section 3.7.2 “Exception Conditions”](#).

Once created as a default result, ∞ can become an operand in a subsequent operation. The infinities are interpreted such that $-\infty < (\text{every finite number}) < +\infty$. Arithmetic with ∞ is the limiting case of real arithmetic with operands of arbitrarily large magnitude, when such limits exist. In these cases, arithmetic on ∞ is regarded as exact, and exception conditions do not arise. The out-of-range indication represented by ∞ is propagated through subsequent computations. For some cases, there is no meaningful limiting case in real arithmetic for operands of ∞ . These cases raise the Invalid Operation exception condition as described in [Section 3.7.2.1 “Invalid Operation Exception”](#).

3.3.1.4 Signalling Non-Number (SNaN)

SNaN operands cause an Invalid Operation exception for arithmetic operations. SNaNs are useful values to put in uninitialized variables. An SNaN is never produced as a result value.

IEEE Standard 754 states that “Whether copying a signaling NaN without a change of format signals the Invalid Operation exception is the implementor’s option.” The MIPS architecture makes the formatted operand move instructions (MOV.fmt, MOVT.fmt, MOVF.fmt, MOVN.fmt, MOVZ.fmt) non-arithmetic; they do not signal IEEE 754 exceptions.

3.3.1.5 Quiet Non-Number (QNaN)

QNaNs provide retrospective diagnostic information inherited from invalid or unavailable data and results. Propagation of the diagnostic information requires information contained in a QNaN to be preserved through arithmetic operations and floating-point format conversions.

QNaN operands do not cause arithmetic operations to signal an exception. When a floating-point result is to be delivered, a QNaN operand causes an arithmetic operation to supply a QNaN result. When possible, this QNaN result is one¹ of the operand QNaN values. QNaNs do have effects similar to SNaNs on operations that do not deliver a floating-point result—specifically, comparisons. (For more information, see the detailed description of the floating-point compare instruction, C.cond.fmt.).

When certain invalid operations not involving QNaN operands are performed but do not trap (because the trap is not enabled), a new QNaN value is created. Table 3.3 shows the QNaN value generated when no input operand QNaN value can be copied. The values listed for the fixed-point formats are the values supplied to satisfy IEEE Standard 754 when a QNaN or infinite floating-point value is converted to fixed point. There is no other feature of the architecture that detects or makes use of these “integer QNaN” values.

Table 3.3 Value Supplied When a New Quiet NaN is Created

Format	New QNaN value
Single floating point	0x7fbf ffff
Double floating point	0x7ff7 ffff ffff ffff
Word fixed point	0x7fff ffff
Longword fixed point	0x7fff ffff ffff ffff

3.3.2 Fixed-Point Formats

The FPU provides two fixed-point data types:

- a 32-bit Word fixed point (type W), shown in Figure 3.4
- a 64-bit Longword fixed point (type L), shown in Figure 3.5

The fixed-point values are held in 2’s complement format, which is used for signed integers in the CPU. Unsigned fixed-point data types are not provided by the architecture; application software can synthesize computations for unsigned integers from the existing instructions and data types.

Figure 3.4 Word Fixed-Point Format (W)

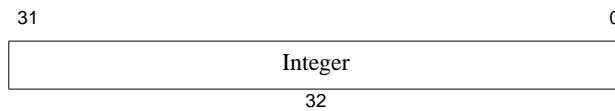
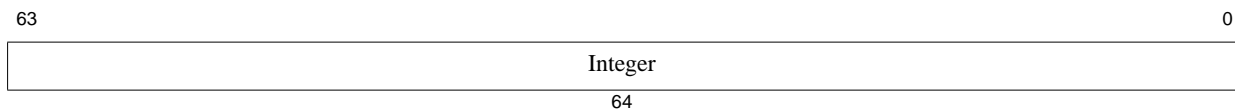


Figure 3.5 Longword Fixed-Point Format (L)



1. In case of one or more QNaN operands, a QNaN is propagated from one of the operands according to the following priority: 1: fs, 2: ft, 3: fr.

3.4 Floating-Point General Registers

This section describes the organization and use of the Floating-Point general Registers (FPRs). The FPU is a 64b FPU, but a 32b register mode for backwards compatibility is also supported. The FR bit in the CP0 *Status* register determines which mode is selected:

- When the FR bit is a 1, the 64b register model is selected, which defines 32 64-bit registers with all formats supported in a register.
- When the FR bit is a 0, the 32b register model is selected, which defines 32 32-bit registers with D-format values stored in even-odd pairs of registers; thus the register file can also be viewed as having 16 64-bit registers.

These registers transfer binary data between the FPU and the system, and are also used to hold formatted FPU operand values.

3.4.1 FPRs and Formatted Operand Layout

FPU instructions that operate on formatted operand values specify the Floating-Point Register (FPR) that holds the value. Operands that are only 32 bits wide (*W* and *S* formats) use only half the space in an FPR.

Figures 3.6 and 3.7 show the FPR organization and the way that operand data is stored in them.

Figure 3.6 Single Floating-Point or Word Fixed-Point Operand in an FPR

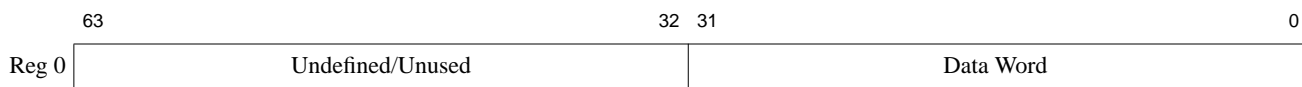
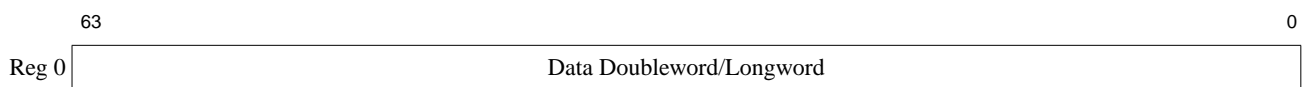


Figure 3.7 Double Floating-Point or Longword Fixed-Point Operand in an FPR



3.4.2 Formats of Values Used in FP Registers

Unlike the CPU, the FPU neither interprets the binary encoding of source operands nor produces a binary encoding of results for every operation. The value held in a floating-point operand register (FPR) has a format, or type, and it can be used only by instructions that operate on that format. The format of a value is either *uninterpreted*, *unknown*, or one of the valid numeric formats: *single* or *double* floating point, and *word* or *long* fixed point.

The value in an FPR is always set when a value is written to the register as follows:

- When a data transfer instruction writes binary data into an FPR (a load), the FPR receives a binary value that is *uninterpreted*.
- A computational or FP register move instruction that produces a result of type *fmt* puts a value of type *fmt* into the result register.

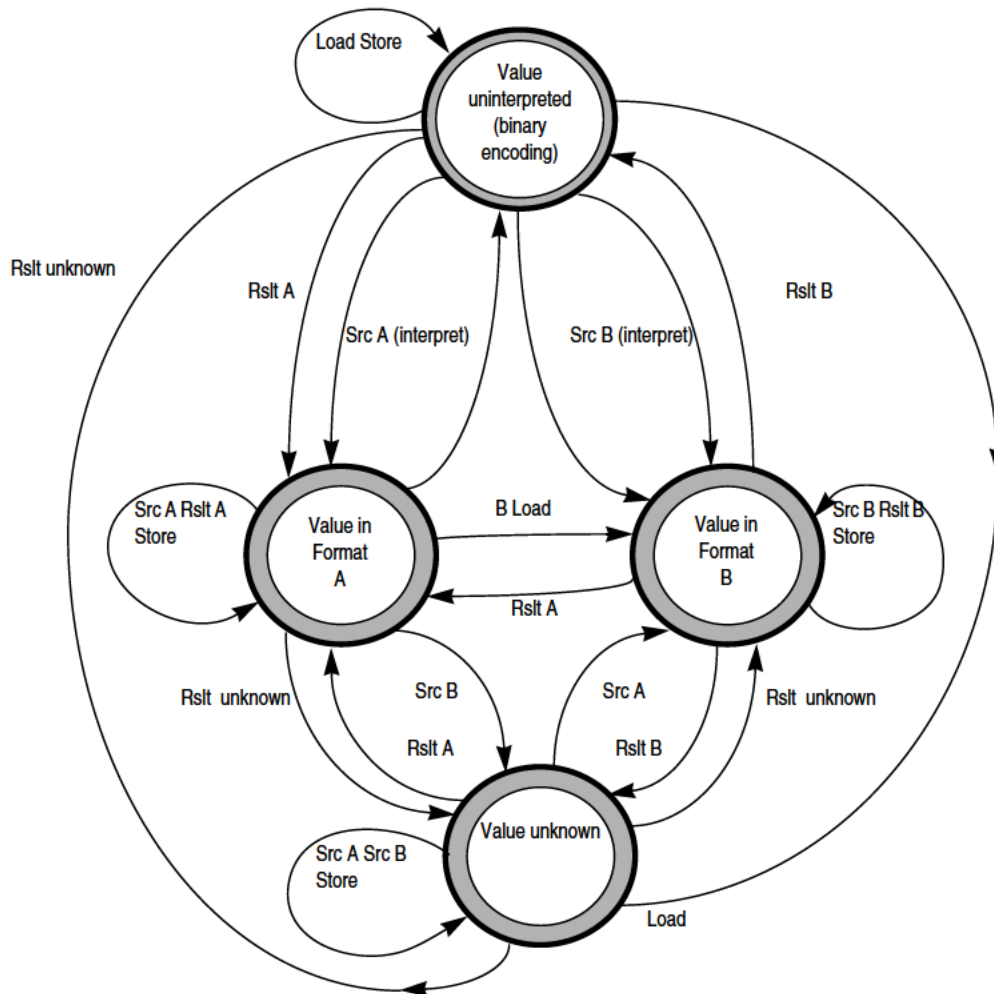
When an FPR with an *uninterpreted* value is used as a source operand by an instruction that requires a value of format *fmt*, the binary contents are interpreted as an encoded value in format *fmt*, and the value in the FPR changes to a value of format *fmt*. The binary contents cannot be reinterpreted in a different format.

If an FPR contains a value of format *fmt*, a computational instruction must not use the FPR as a source operand of a different format. If this case occurs, the value in the register becomes *unknown*, and the result of the instruction is also a value that is *unknown*. Using an FPR containing an *unknown* value as a source operand produces a result that has an *unknown* value.

The format of the value in the FPR is unchanged when it is read by a data transfer instruction (a store). A data transfer instruction produces a binary encoding of the value contained in the FPR. If the value in the FPR is *unknown*, the encoded binary value produced by the operation is not defined.

The state diagram in Figure 3.8 illustrates the manner in which the formatted value in an FPR is set and changed.

Figure 3.8 Effect of FPU Operations on the Format of Values Held in FPRs



A, B: Example formats
 Load: Destination of LWC1, LDC1, MTC1 instructions.
 Store: Source operand of SWC1, SDC1, MFC1 instructions.
 Src *fmt*: Source operand of computational instruction expecting format "*fmt*."
 Rslt *fmt*: Result of computational instruction producing value of format "*fmt*."

3.4.3 Binary Data Transfers (32-Bit and 64-Bit)

The data transfer instructions move words and doublewords between the FPU FPRs and the remainder of the system. The operations of the word and doubleword load and move-to instructions are shown in Figure 3.9 and Figure 3.10, respectively.

The store and move-from instructions operate in reverse, reading data from the location that the corresponding load or move-to instruction had written.

Figure 3.9 FPU Word Load and Move-to Operations

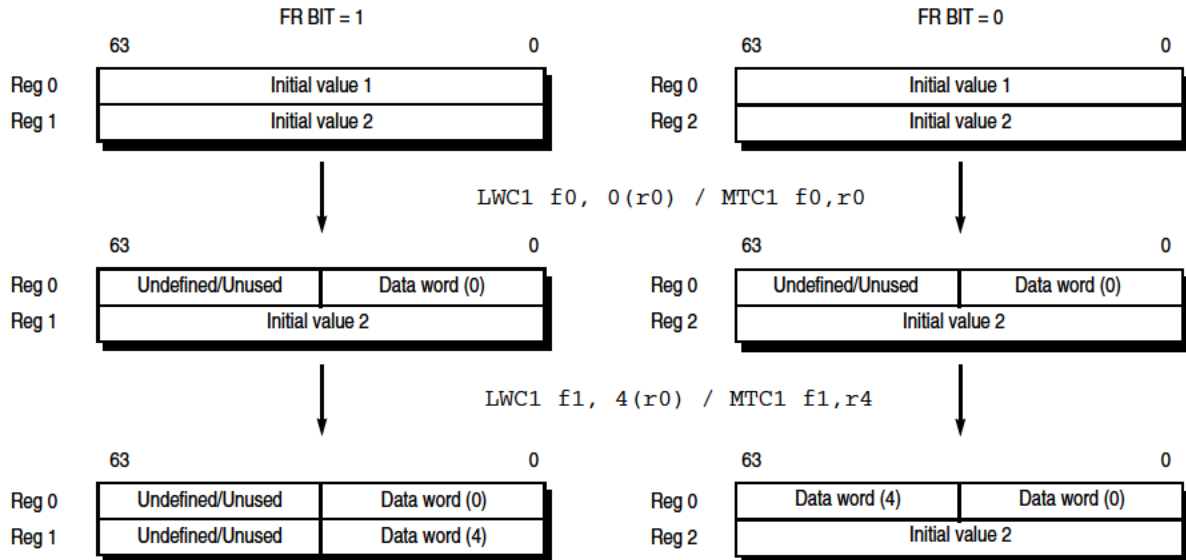
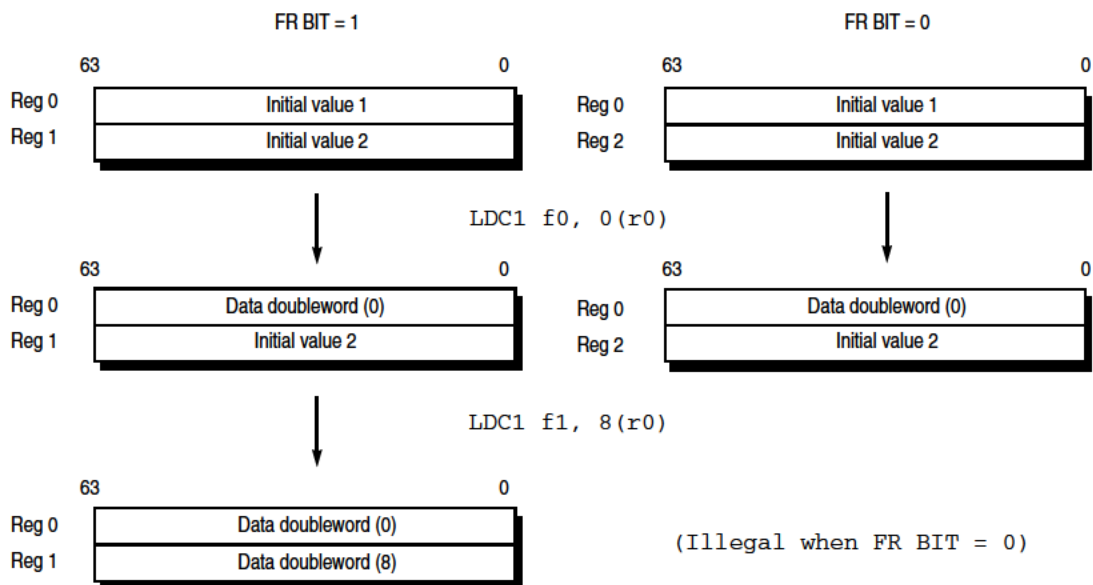


Figure 3.10 FPU Doubleword Load and Move-to Operations



3.5 Floating-Point Control Registers

The FPU Control Registers (FCRs) identify and control the FPU. The five FPU control registers are 32 bits wide: *FIR*, *FCCR*, *FEXR*, *FENR*, *FCSR*. Three of these registers, *FCCR*, *FEXR*, and *FENR*, select subsets of the floating-point Control/Status register, the *FCSR*. These registers are also denoted Coprocessor 1 (CP1) control registers.

CP1 control registers are summarized in Table 3.4 and are described individually in the following subsections of this chapter. Each register’s description includes the read/write properties and the reset state of each field.

Table 3.4 Coprocessor 1 Register Summary

Register Number	Register Name	Function
0	FIR	Floating-Point Implementation register. Contains information that identifies the FPU.
25	FCCR	Floating-Point Condition Codes register.
26	FEXR	Floating-Point Exceptions register.
28	FENR	Floating-Point Enables register.
31	FCSR	Floating-Point Control and Status register.

Table 3.5 defines the notation used for the read/write properties of the register bit fields.

Table 3.5 Read/Write Properties

Read/Write Notation	Hardware Interpretation	Software Interpretation
R/W	All bits in this field are readable and writable by software and potentially by hardware. Hardware updates of this field are visible by software reads. Software updates of this field are visible by hardware reads. If the reset state of this field is “Undefined,” either software or hardware must initialize the value before the first read returns a predictable value. This definition should not be confused with the formal definition of UNDEFINED behavior.	
R	This field 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.	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 UNPREDICTABLE value except after a hardware update done under the conditions specified in the description of the field.
0	Hardware does not update this field. Hardware can assume a zero value.	The value software writes to this field must be zero. Software writes of non-zero values to this field might result in UNDEFINED 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.

Table 3.6 FIR Bit Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
D	17	Indicates that the double-precision (D) floating-point data type and instructions are implemented: <ul style="list-style-type: none"> • 0: D floating-point not implemented • 1: D floating-point implemented This bit is always 1 to indicate that double-precision floating-point data types are implemented.	R	1
S	16	Indicates that the single-precision (S) floating-point data type and instructions are implemented: <ul style="list-style-type: none"> • 0: S floating-point not implemented • 1: S floating-point implemented This bit is always 1 to indicate that single-precision floating-point data types are implemented.	R	1
Processor ID	15:8	Identifies the floating-point processor. This value matches the corresponding field of the CP0 PRId register.	R	0x97
Revision	7:0	Specifies the revision number of the FPU. This field allows software to distinguish between different revisions of the same floating-point processor type. This value matches the corresponding field of the CP0 PRId register.	R	Hardwired
0	31:25, 23	These bits must be written as zeros; they return zeros on reads.	0	0

3.5.2 Floating-Point Condition Codes Register (FCCR, CP1 Control 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 the *FCSR*. Unlike the *FCSR*, all eight FCC bits are contiguous in the *FCCR*. [Figure 3.12](#) shows the format of the *FCCR*; [Table 3.7](#) describes the *FCCR* bit fields.

Figure 3.12 FCCR Format



Table 3.7 FCCR Bit Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
FCC	7:0	Floating-point condition code. Refer to the description of this field in Section 3.5.5 “Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)” .	R/W	Undefined
0	31:8	These bits must be written as zeros; they return zeros on reads.	0	0

3.5.3 Floating-Point Exceptions Register (FEXR, CP1 Control 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 the *FCSR*. [Figure 3.13](#) shows the format of the *FEXR*; [Table 3.8](#) describes the *FEXR* bit fields.

Figure 3.13 FEXR Format

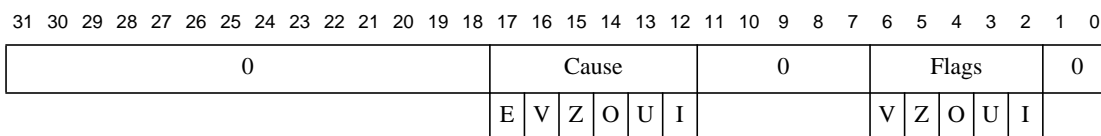


Table 3.8 FEXR Bit Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Cause	17:12	Cause bits. Refer to the description of this field in 3.5.5 “Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)”.	R/W	Undefined
Flags	6:2	Flag bits. Refer to the description of this field in 3.5.5 “Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)”.	R/W	Undefined
0	31:18, 11:7, 1:0	These bits must be written as zeros; they return zeros on reads.	0	0

3.5.4 Floating-Point Enables Register (FENR, CP1 Control 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 the *FCSR*. Figure 3.14 shows the format of the *FENR*; Table 3.9 describes the *FENR* bit fields.

Figure 3.14 FENR Format

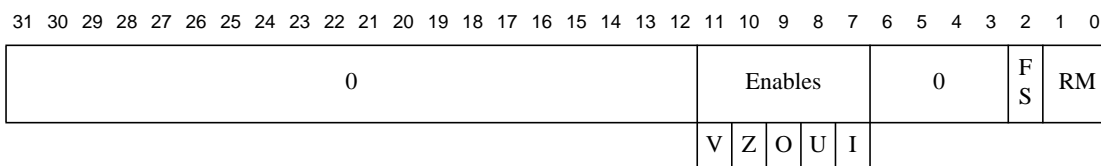


Table 3.9 FENR Bit Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Enables	11:7	Enable bits. Refer to the description of this field in 3.5.5 “Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)”.	R/W	Undefined
FS	2	Flush to Zero bit. Refer to the description of this field in 3.5.5 “Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)”.	R/W	Undefined
RM	1:0	Rounding mode. Refer to the description of this field in 3.5.5 “Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)”.	R/W	Undefined
0	31:12, 6:3	These bits must be written as zeros; they return zeros on reads.	0	0

3.5.5 Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)

The 32-bit Floating-Point Control and Status Register (*FCSR*) controls the operation of the FPU and shows the following status information:

- selects the default rounding mode for FPU arithmetic operations
- selectively enables traps of FPU exception conditions
- controls some denormalized number handling options
- reports any IEEE exceptions that arose during the most recently executed instruction
- reports any IEEE exceptions that cumulatively arose in completed instructions
- indicates the condition code result of FP compare instructions

Access to the *FCSR* is not privileged; it can be read or written by any program that has access to the FPU (via the coprocessor enables in the *Status* register). [Figure 3.15](#) shows the format of the *FCSR*; [Table 3.10](#) describes the *FCSR* bit fields.

Figure 3.15 FCSR Format

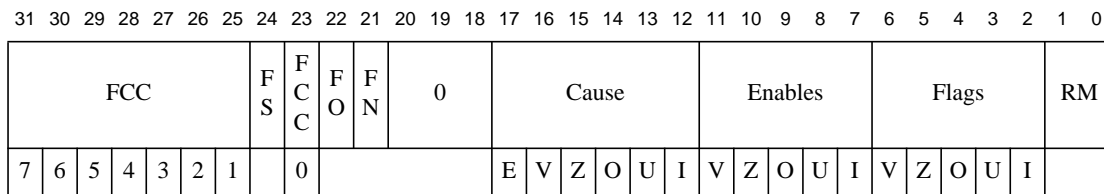


Table 3.10 FCSR Bit Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit			
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
FS	24	Flush to Zero (FS). Refer to 3.5.6 “Operation of the FS/FO/FN Bits” for more details on this bit.	R/W	Undefined
FO	22	Flush Override (FO). Refer to Section 3.5.6, “Operation of the FS/FO/FN Bits” for more details on this bit.	R/W	Undefined
FN	21	Flush to Nearest (FN). Refer to Section 3.5.6, “Operation of the FS/FO/FN Bits” for more details on this bit.	R/W	Undefined

Table 3.10 FCSR Bit Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bit			
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 when the corresponding exception condition arises during the execution of an instruction; otherwise, it is cleared to 0. By reading the registers, the exception condition caused by the preceding FPU arithmetic instruction can be determined. Refer to Table 3.11 for the meaning of each cause bit.	R/W	Undefined
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 its corresponding cause bit are set either during an FPU arithmetic operation or by moving a value to the <i>FCSR</i> or one of its alternative representations. Note that Cause bit E (CauseE) has no corresponding enable bit; the MIPS architecture defines non-IEEE Unimplemented Operation exceptions as always enabled. Refer to Table 3.11 for the meaning of each enable bit.	R/W	Undefined
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 an FPU arithmetic operation raises an IEEE exception condition that does not result in a Floating-Point Exception (the enable bit was off), the corresponding bit(s) in the Flags field are set, while the others remain unchanged. Arithmetic operations that result in a Floating-Point Exception (the enable bit was on) do not update the Flags field. Hardware never resets this field; software must explicitly reset this field. Refer to Table 3.11 for the meaning of each flag bit.	R/W	Undefined
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 3.12 for the encoding of this field.	R/W	Undefined
0	20:18	These bits must be written as zeros; they return zeros on reads.	0	0

Table 3.11 Cause, Enables, and Flags 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

Table 3.11 Cause, Enables, and Flags Definitions (Continued)

Bit Name	Bit Meaning
I	Inexact

Table 3.12 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 (even).
1	RZ - Round Toward Zero Rounds the result to the value closest to but not greater 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.
3	RM - Round Towards Minus Infinity Rounds the result to the value closest to but not greater than the result.

3.5.6 Operation of the FS/FO/FN Bits

The FS, FO, and FN bits in the CP1 *FCSR* register control handling of denormalized operands and *tiny* results (i.e. nonzero result between $\pm 2^{E_{min}}$), whereby the FPU can handle these cases right away instead of relying on the much slower software handler. The trade-off is a loss of IEEE compliance and accuracy (except for use of the FO bit), because a minimal normalized or zero result is provided by the FPU instead of the more accurate denormalized result that a software handler would give. The benefit is a significantly improved performance and precision.

Use of the FS, FO, and FN bits affects handling of denormalized floating-point numbers and tiny results for the instructions listed below:

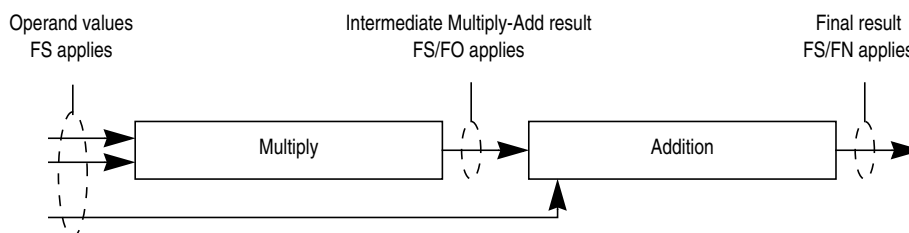
Table 3.13 Handling Denormalized Floating-point Numbers

FS and FN bit:	ADD, CEIL, CVT, DIV, FLOOR, MADD, MSUB, MUL, NMADD, NMSUB, RECIP, ROUND, RSQRT, SQRT, TRUNC, SUB, ABS, C.cond, and NEG ¹
FO bit:	MADD, MSUB, NMADD, and NMSUB

1. For ABS, C.cond, and NEG, denormal input operands or tiny results do not result in Unimplemented exceptions when FS = 0. Flushing to zero nonetheless is implemented when FS = 1 such that these operations return the same result as an equivalent sequence of arithmetic FPU operations.

Instructions not listed above do not cause Unimplemented Operation exceptions on denormalized numbers in operands or results.

Figure 3.16 depicts how the FS, FO, and FN bits control handling of denormalized numbers. For instructions that are not multiply or add types (such as DIV), only the FS and FN bits apply.

Figure 3.16 FS/FO/FN Bits Influence on Multiply and Addition Results

3.5.6.1 Flush To Zero Bit

When the Flush To Zero (FS) bit is set, denormal input operands are flushed to zero. Tiny results are flushed to either zero or the applied format's smallest normalized number (MinNorm) depending on the rounding mode settings. Table 3.14 lists the flushing behavior for tiny results..

Table 3.14 Zero Flushing for Tiny Results

Rounding Mode	Negative Tiny Result	Positive Tiny Result
RN (RM=0)	-0	+0
RZ(RM=1)	-0	+0
RP (RM=2)	-0	+MinNorm
RM (RM=3)	-MinNorm	+0

The flushing of results is based on an intermediate result computed by rounding the mantissa using an unbounded exponent range; that is, tiny numbers are not *normalized* into the supported exponent range by shifting in leading zeros prior to rounding.

Handling of denormalized operand values and tiny results depends on the FS bit setting as shown in Table 3.15.

Table 3.15 Handling of Denormalized Operand Values and Tiny Results Based on FS Bit Setting

FS Bit	Handling of Denormalized Operand Values
0	An Unimplemented Operation exception is taken.
1	Instead of causing an Unimplemented Operation exception, operands are flushed to zero, and tiny results are forced to zero or MinNorm.

3.5.6.2 Flush Override Bit

When the Flush Override (FO) bit is set, a tiny intermediate result of any multiply-add type instruction is not flushed according to the FS bit. The intermediate result is maintained in an internal normalized format to improve accuracy. FO only applies to the intermediate result of a multiply-add type instruction.

Handling of tiny intermediate results depends on the FO and FS bits as shown in Table 3.16.

Table 3.16 Handling of Tiny Intermediate Result Based on the FO and FS Bit Settings

FO Bit	FS Bit	Handling of Tiny Result Values
0	0	An Unimplemented Operation exception is taken.

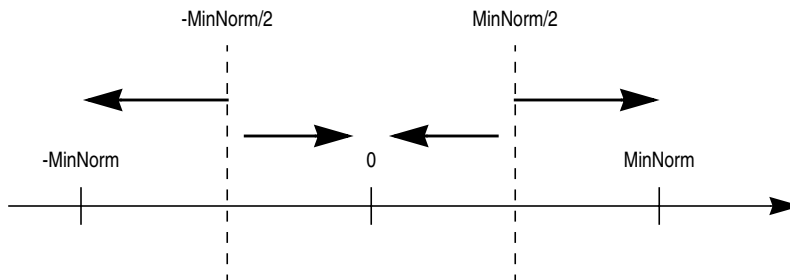
Table 3.16 Handling of Tiny Intermediate Result Based on the FO and FS Bit Settings (Continued)

FO Bit	FS Bit	Handling of Tiny Result Values
0	1	The intermediate result is forced to the value that would have been delivered for an untrapped underflow (see Table 3.34) instead of causing an Unimplemented Operation exception.
1	Don't care	The intermediate result is kept in an internal format, which can be perceived as having the usual mantissa precision but with unlimited exponent precision and without forcing to a specific value or taking an exception.

3.5.6.3 Flush to Nearest

When the Flush to Nearest (FN) bit is set and the rounding mode is Round to Nearest (RN), a tiny final result is flushed to zero or MinNorm. If a tiny number is strictly below MinNorm/2, the result is flushed to zero; otherwise, it is flushed to MinNorm (see Figure 3.17). The flushed result has the same sign as the result prior to flushing. Note that the FN bit takes precedence over the FS bit.

Figure 3.17 Flushing to Nearest when Rounding Mode is Round to Nearest



For all rounding modes other than Round to Nearest (RN), setting the FN bit causes final results to be flushed to zero or MinNorm as if the FS bit was set.

Handling of tiny final results depends on the FN and FS bits as shown in Table 3.17.

Table 3.17 Handling of Tiny Final Result Based on FN and FS Bit Settings

FN Bit	FS Bit	Handling of Tiny Result Values
0	0	An Unimplemented Operation exception is taken.
0	1	Final result is forced to the value that would have been delivered for an untrapped underflow (see Table 3.34) rather than causing an Unimplemented Operation exception.
1	Don't care	Final result is rounded to either zero or $2^{E_{min}}$ (MinNorm), whichever is closest when in Round to Nearest (RN) rounding mode. For other rounding modes, a final result is given as if FS was set to 1.

3.5.6.4 Recommended FS/FO/FN Settings

Table 3.18 summarizes the recommended FS/FO/FN settings.

Table 3.18 Recommended FS/FO/FN Settings

FS Bit	FO Bit	FN Bit	Remarks
0	0	0	IEEE-compliant mode. Low performance on denormal operands and tiny results.
1	0	0	Regular embedded applications. High performance on denormal operands and tiny results.
1	1	1	Highest accuracy and performance configuration. ¹

1. Note that in this mode, MADD might return a different result other than the equivalent MUL and ADD operation sequence.

3.5.7 FCSR Cause Bit Update Flow

3.5.7.1 Exceptions Triggered by CTC1

Regardless of the targeted control register, the CTC1 instruction causes the Enables and Cause fields of the *FCSR* to be inspected in order to determine if an exception is to be thrown.

3.5.7.2 Generic Flow

Computations are performed in two steps:

1. Compute rounded mantissa with unbound exponent range.
2. Flush to default result if the result from Step #1 above is overflow or tiny (no flushing happens on denorms for instructions supporting denorm results, such as MOV).

The Cause field is updated after each of these two steps. Any enabled exceptions detected in these two steps cause a trap, and no further updates to the Cause field are done by subsequent steps.

Step #1 can set cause bits I, U, O, Z, V, and E. E has priority over V; V has priority over Z; and Z has priority over U and O. Thus when E, V, or Z is set in Step #1, no other cause bits can be set. However, note that I and V both can be set if a denormal operand was flushed (FS = 1). I, U, and O can be set alone or in pairs (IU or IO). U and O never can be set simultaneously in Step #1. U and O are set if the computed unbounded exponent is outside the exponent range supported by the normalized IEEE format.

Step #2 can set I if a default result is generated.

3.5.7.3 Multiply-Add Flow

For multiply-add type instructions, the computation is extended with two more steps:

1. Compute rounded mantissa with unbound exponent range for the multiply.
2. Flush to default result if the result from Step #1 is overflow or tiny (no flushing happens on tiny results if FO = 1).
3. Compute rounded mantissa with unbounded exponent range for the add.

Floating-Point Unit of the 74Kf™ Core

4. Flush to default result if the result from Step #3 is overflow or tiny.

The Cause field is updated after each of these four steps. Any enabled exceptions detected in these four steps cause a trap, and no further updates to the Cause field are done by subsequent steps.

Step #1 and Step #3 can set a cause bit as described for Step #1 in 3.5.7.2 “Generic Flow”.

Step #2 and Step #4 can set I if a default result is generated.

Although U and O can never both be set in Step #1 or Step #3, both U and O might be set after the multiply-add has executed in Step #3 because U might be set in Step #1 and O might be set in Step #3.

3.5.7.4 Cause Update Flow for Input Operands

Denormal input operands to Step #1 or Step #3 always set Cause bit I when FS = 1. For example, SNaN+DeNorm set I (and V) provided that Step #3 was reached (in case of a multiply-add type instruction).

Conditions directly related to the input operand (for example, I/E set due to DeNorm, V set due to SNaN and QNaN propagation) are detected in the step where the operand is logically used. For example, for multiply-add type instructions, exceptional conditions caused by the input operand fr are detected in Step #3.

3.5.7.5 Cause Update Flow for Unimplemented Operations

Note that Cause bit E is special; it clears any Cause updates done in previous steps. For example, if Step #3 caused E to be set, any I, U, or O Cause update done in Step #1 or Step #2 is cleared. Only E is set in the Cause field when an Unimplemented Operation trap is taken.

3.6 Instruction Overview

The functional groups into which the FPU instructions are divided are described in the following subsections:

- Section 3.6.1 “Data Transfer Instructions”
- Section 3.6.2 “Arithmetic Instructions”
- Section 3.6.3 “Conversion Instructions”
- Section 3.6.4 “Formatted Operand-Value Move Instructions”
- Section 3.6.5 “Conditional Branch Instructions”
- Section 3.6.6 “Miscellaneous Instructions”

The instructions are described in detail in Chapter 13, “74K™ Processor Core Instructions”, including descriptions of supported formats (fmt).

3.6.1 Data Transfer Instructions

The FPU has two separate register sets: coprocessor general registers (FPRs) and coprocessor control registers (FCRs). The FPU has a load/store architecture; all computations are done on data held in coprocessor general registers. The control registers are used to control FPU operation. Data is transferred between registers and the rest of the

system with dedicated load, store, and move instructions. The transferred data is treated as unformatted binary data; no format conversions are performed, and therefore no IEEE floating-point exceptions can occur.

Table 3.19 lists the supported transfer operations.

Table 3.19 FPU Data Transfer Instructions

Transfer Direction			Data Transferred
FPU general register	↔	Memory	Word/doubleword load/store
FPU general register	↔	CPU general register	Word move
FPU control register	↔	CPU general register	Word move

3.6.1.1 Data Alignment in Loads, Stores, and Moves

All coprocessor loads and stores operate on naturally aligned data items. An attempt to load or store to an address that is not naturally aligned for the data item causes an Address Error exception. Regardless of byte ordering (the endianness), the address of a word or doubleword is the smallest byte address in the object. For a big-endian machine, this is the most-significant byte; for a little-endian machine, this is the least-significant byte.

3.6.1.2 Addressing Used in Data Transfer Instructions

The FPU has loads and stores using the same register+offset addressing as that used by the CPU. Moreover, for the FPU only, there are load and store instructions using *register+register* addressing.

Tables 3.20 through 3.22 list the FPU data transfer instructions.

Table 3.20 FPU Loads and Stores Using Register+Offset Address Mode

Mnemonic	Instruction
LDC1	Load Doubleword to Floating Point
LWC1	Load Word to Floating Point
SDC1	Store Doubleword to Floating Point
SWC1	Store Word to Floating Point

Table 3.21 FPU Loads and Stores Using Register+Register Address Mode

Mnemonic	Instruction
LDXC1	Load Doubleword Indexed to Floating Point
LUXC1	Load Doubleword Indexed Unaligned to Floating Point
LWXC1	Load Word Indexed to Floating Point
SDXC1	Store Doubleword Indexed to Floating Point
SUXC1	Store Doubleword Indexed Unaligned to Floating Point
SWXC1	Store Word Indexed to Floating Point

Table 3.22 FPU Move To and From Instructions

Mnemonic	Instruction
CFC1	Move Control Word From Floating Point
CTC1	Move Control Word To Floating Point

Table 3.22 FPU Move To and From Instructions (Continued)

Mnemonic	Instruction
MFC1	Move Word From Floating Point
MTC1	Move Word To Floating Point

3.6.2 Arithmetic Instructions

Arithmetic instructions operate on formatted data values. The results of most floating-point arithmetic operations meet IEEE Standard 754 for accuracy—a result is identical to an infinite-precision result that has been rounded to the specified format using the current rounding mode. The rounded result differs from the exact result by less than one Unit in the Least-significant Place (ULP).

In general, the arithmetic instructions take an Unimplemented Operation exception for denormalized numbers, except for the ABS, C, and NEG instructions, which can handle denormalized numbers. The FS, FO, and FN bits in the CP1 *FCSR* register can override this behavior as described in Section 3.5.6, "Operation of the FS/FO/FN Bits".

Table 3.23 lists the FPU IEEE compliant arithmetic operations.

Table 3.23 FPU IEEE Arithmetic Operations

Mnemonic	Instruction
ABS.fmt	Floating-Point Absolute Value
ADD.fmt	Floating-Point Add
C.cond.fmt	Floating-Point Compare
DIV.fmt	Floating-Point Divide
MUL.fmt	Floating-Point Multiply
NEG.fmt	Floating-Point Negate
SQRT.fmt	Floating-Point Square Root
SUB.fmt	Floating-Point Subtract

The two low latency operations, Reciprocal Approximation (RECIP) and Reciprocal Square Root Approximation (RSQRT), might be less accurate than the IEEE specification:

- The result of RECIP differs from the exact reciprocal by no more than one ULP.
- The result of RSQRT differs from the exact reciprocal square root by no more than two ULPs.

Table 3.24 lists the FPU-approximate arithmetic operations.

Table 3.24 FPU-Approximate Arithmetic Operations

Mnemonic	Instruction
RECIP.fmt	Floating-Point Reciprocal Approximation
RSQRT.fmt	Floating-Point Reciprocal Square Root Approximation

Four compound-operation instructions perform variations of multiply-accumulate operations; that is, multiply two operands, accumulate the result to a third operand, and produce a result. These instructions are listed in Table 3.25. The product is rounded according to the current rounding mode prior to the accumulation. This model meets the IEEE

accuracy specification; the result is numerically identical to an equivalent computation using multiply, add, subtract, or negate instructions.

Table 3.25 FPU Multiply-Accumulate Arithmetic Operations

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

3.6.3 Conversion Instructions

These instructions perform conversions between floating-point and fixed-point data types. Each instruction converts values from a number of operand formats to a particular result format. Some conversion instructions use the rounding mode specified in the Floating Control/Status register (*FCSR*), while others specify the rounding mode directly.

In general, the conversion instructions only take an Unimplemented Operation exception for denormalized numbers. The FS and FN bits in the CP1 *FCSR* register can override this behavior as described in [Section 3.5.6, "Operation of the FS/FO/FN Bits"](#).

[Table 3.26](#) and [Table 3.27](#) list the FPU conversion instructions according to their rounding mode.

Table 3.26 FPU Conversion Operations Using the FCSR Rounding Mode

Mnemonic	Instruction
CVT.D.fmt	Floating-Point Convert to Double Floating Point
CVT.L.fmt	Floating-Point Convert to Long Fixed Point
CVT.S.fmt	Floating-Point Convert to Single Floating Point
CVT.W.fmt	Floating-Point Convert to Word Fixed Point

Table 3.27 FPU Conversion Operations Using a Directed Rounding Mode

Mnemonic	Instruction
CEIL.L.fmt	Floating-Point Ceiling to Long Fixed Point
CEIL.W.fmt	Floating-Point Ceiling to Word Fixed Point
FLOOR.L.fmt	Floating-Point Floor to Long Fixed Point
FLOOR.W.fmt	Floating-Point Floor to Word Fixed Point
ROUND.L.fmt	Floating-Point Round to Long Fixed Point
ROUND.W.fmt	Floating-Point Round to Word Fixed Point
TRUNC.L.fmt	Floating-Point Truncate to Long Fixed Point
TRUNC.W.fmt	Floating-Point Truncate to Word Fixed Point

3.6.4 Formatted Operand-Value Move Instructions

These instructions move formatted operand values among FPU general registers. A particular operand type must be moved by the instruction that handles that type. There are three kinds of move instructions:

Floating-Point Unit of the 74Kf™ Core

- Unconditional move
- Conditional move that tests an FPU true/false condition code
- Conditional move that tests a CPU general-purpose register against zero

Conditional move instructions operate in a way that might be unexpected. They always force the value in the destination register to become a value of the format specified in the instruction. If the destination register does not contain an operand of the specified format before the conditional move is executed, the contents become undefined. (For more information, see the individual descriptions of the conditional move instructions in the *MIPS32 Architecture Reference Manual, Volume II*.)

Table 3.28 through Table 3.30 list the formatted operand-value move instructions.

Table 3.28 FPU Formatted Operand Move Instruction

Mnemonic	Instruction
MOV.fmt	Floating-Point Move

Table 3.29 FPU Conditional Move on True/False Instructions

Mnemonic	Instruction
MOV.fmt	Floating-Point Move Conditional on FP False
MOVT.fmt	Floating-Point Move Conditional on FP True

Table 3.30 FPU Conditional Move on Zero/Non-Zero Instructions

Mnemonic	Instruction
MOVN.fmt	Floating-Point Move Conditional on Nonzero
MOVZ.fmt	Floating-Point Move Conditional on Zero

3.6.5 Conditional Branch Instructions

The FPU has PC-relative conditional branch instructions that test condition codes set by FPU compare instructions (C.cond.fmt).

All branches have an architectural delay of one instruction. When a branch is taken, the instruction immediately following the branch instruction is said to be in the branch delay slot; it is executed before the branch to the target instruction takes place. Conditional branches come in two versions, depending upon how they handle an instruction in the delay slot when the branch is not taken and execution falls through:

- Branch instructions execute the instruction in the delay slot.
- Branch likely instructions do not execute the instruction in the delay slot if the branch is not taken (they are said to nullify the instruction in the delay slot).

Although the Branch Likely instructions are included, software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

The MIPS64 architecture defines eight condition codes for use in compare and branch instructions. For backward compatibility with previous revisions of the ISA, condition code bit 0 and condition code bits 1 through 7 are in discontinuous fields in the *FCSR*.

Table 3.31 lists the conditional branch (branch and branch likely) FPU instructions; Table 3.32 lists the deprecated conditional branch likely instructions.

Table 3.31 FPU Conditional Branch Instructions

Mnemonic	Instruction
BC1F	Branch on FP False
BC1T	Branch on FP True

Table 3.32 Deprecated FPU Conditional Branch Likely Instructions

Mnemonic	Instruction
BC1FL	Branch on FP False Likely
BC1TL	Branch on FP True Likely

3.6.6 Miscellaneous Instructions

The MIPS32 architecture defines various miscellaneous instructions that conditionally move one CPU general register to another, based on an FPU condition code.

Table 3.33 lists these conditional move instructions.

Table 3.33 CPU Conditional Move on FPU True/False Instructions

Mnemonic	Instruction
MOVN	Move Conditional on FP False
MOVZ	Move Conditional on FP True

3.7 Exceptions

FPU exceptions are implemented in the MIPS FPU architecture with the Cause, Enables, and Flags fields of the *FCSR*. The flag bits implement IEEE exception status flags, and the cause and enable bits control exception trapping. Each field has a bit for each of the five IEEE exception conditions. The Cause field has an additional exception bit, Unimplemented Operation, used to trap for software emulation assistance. If an exception type is enabled through the Enables field of the *FCSR*, then the FPU is operating in precise exception mode for this type of exception.

3.7.1 Precise Exception Mode

In precise exception mode, a trap occurs before the instruction that causes the trap or any following instruction can complete and write its results. If desired, the software trap handler can resume execution of the interrupted instruction stream after handling the exception.

The Cause field reports per-bit instruction exception conditions. The cause bits are written during each floating-point arithmetic operation to show any exception conditions that arise during the operation. A cause bit is set to 1 if its corresponding exception condition arises; otherwise, it is cleared to 0.

A floating-point trap is generated any time both a cause bit and its corresponding enable bit are set. This case occurs either during the execution of a floating-point operation or when moving a value into the *FCSR*. There is no enable bit for Unimplemented Operations; this exception always generates a trap.

In a trap handler, exception conditions that arise during any trapped floating-point operations are reported in the Cause field. Before returning from a floating-point interrupt or exception, or before setting cause bits with a move to the *FCSR*, software first must clear the enabled cause bits by executing a move to the *FCSR* to prevent the trap from being erroneously retaken.

If a floating-point operation sets only non-enabled cause bits, no trap occurs and the default result defined by IEEE Standard 754 is stored (see [Table 3.34](#)). When a floating-point operation does not trap, the program can monitor the exception conditions by reading the Cause field.

The Flags field is a cumulative report of IEEE exception conditions that arise as instructions complete; instructions that trap do not update the flag bits. The flag bits are set to 1 if the corresponding IEEE exception is raised, otherwise the bits are unchanged. There is no flag bit for the MIPS Unimplemented Operation exception. The flag bits are never cleared as a side effect of floating-point operations, but they can be set or cleared by moving a new value into the *FCSR*.

3.7.2 Exception Conditions

The subsections below describe the following five exception conditions defined by IEEE Standard 754:

- [Section 3.7.2.1 “Invalid Operation Exception”](#)
- [Section 3.7.2.2 “Division By Zero Exception”](#)
- [Section 3.7.2.3 “Underflow Exception”](#)
- [Section 3.7.2.4 “Overflow Exception”](#)
- [Section 3.7.2.5 “Inexact Exception”](#)

[3.7.2.6 “Unimplemented Operation Exception”](#) also describes a MIPS-specific exception condition, Unimplemented Operation Exception, that is used to signal a need for software emulation of an instruction. Normally an IEEE arithmetic operation can cause only one exception condition; the only case in which two exceptions can occur at the same time are Inexact With Overflow and Inexact With Underflow.

At the program’s direction, an IEEE exception condition can either cause a trap or not cause a trap. IEEE Standard 754 specifies the result to be delivered in case no trap is taken. The FPU supplies these results whenever the exception condition does not result in a trap. The default action taken depends on the type of exception condition and, in the case of the Overflow and Underflow, the current rounding mode. [Table 3.34](#) summarizes the default results.

Table 3.34 Result for Exceptions Not Trapped

Bit	Description	Default Action
V	Invalid Operation	Supplies a quiet NaN.
Z	Divide by zero	Supplies a properly signed infinity.

Table 3.34 Result for Exceptions Not Trapped (Continued)

Bit	Description	Default Action
U	Underflow	<p>Depends on the rounding mode as shown below:</p> <ul style="list-style-type: none"> • 0 (RN) and 1 (RZ): Supplies a zero with the sign of the exact result. • 2 (RP): For positive underflow values, supplies $2^{E_{\min}}$ (MinNorm). For negative underflow values, supplies a positive zero. • 3 (RM): For positive underflow values, supplies a negative zero. For negative underflow values, supplies a negative $2^{E_{\min}}$ (MinNorm). <p>Note that this behavior is only valid if the $FCSR_{FN}$ bit is cleared.</p>
I	Inexact	Supplies a rounded result. If caused by an overflow without the overflow trap enabled, supplies the overflowed result. If caused by an underflow without the underflow trap enabled, supplies the underflowed result.
O	Overflow	<p>Depends on the rounding mode, as shown below:</p> <ul style="list-style-type: none"> • 0 (RN): Supplies an infinity with the sign of the exact result. • 1 (RZ): Supplies the format's largest finite number with the sign of the exact result. • 2 (RP): For positive overflow values, supplies positive infinity. For negative overflow values, supplies the format's most negative finite number. • 3 (RM): For positive overflow values, supplies the format's largest finite number. For negative overflow values, supplies minus infinity.

3.7.2.1 Invalid Operation Exception

An Invalid Operation exception is signaled when one or both of the operands are invalid for the operation to be performed. When the exception condition occurs without a precise trap, the result is a quiet NaN.

The following operations are invalid:

- One or both operands are a signaling NaN (except for the non-arithmetic MOV fmt, MOVT fmt, MOVF.fmt, MOVN.fmt, and MOVZ.fmt instructions).
- Addition or subtraction: magnitude subtraction of infinities, such as $(+\infty) + (-\infty)$ or $(-\infty) - (-\infty)$.
- Multiplication: $0 \times \infty$, with any signs.
- Division: $0/0$ or ∞/∞ , with any signs.
- Square root: An operand of less than 0 (-0 is a valid operand value).
- Conversion of a floating-point number to a fixed-point format when either an overflow or an operand value of infinity or NaN precludes a faithful representation in that format.
- Some comparison operations in which one or both of the operands is a QNaN value.

3.7.2.2 Division By Zero Exception

The divide operation signals a Division By Zero exception if the divisor is zero and the dividend is a finite nonzero number. When no precise trap occurs, the result is a correctly signed infinity. Divisions ($0/0$ and $\infty/0$) do not cause the Division By Zero exception. The result of ($0/0$) is an Invalid Operation exception. The result of ($\infty/0$) is a correctly signed infinity.

3.7.2.3 Underflow Exception

Two related events contribute to underflow:

- **Tinness:** The creation of a tiny, nonzero result between $\pm 2^{E_{min}}$ which, because it is tiny, might cause some other exception later such as overflow on division. IEEE Standard 754 allows choices in detecting tininess events. The MIPS architecture specifies that tininess be detected after rounding, when a nonzero result computed as though the exponent range were unbounded would lie strictly between $\pm 2^{E_{min}}$.
- **Loss of accuracy:** The extraordinary loss of accuracy occurs during the approximation of such tiny numbers by denormalized numbers. IEEE Standard 754 allows choices in detecting loss of accuracy events. The MIPS architecture specifies that loss of accuracy be detected as inexact result, when the delivered result differs from what would have been computed if both the exponent range and precision were unbounded.

The way that an underflow is signaled depends on whether or not underflow traps are enabled:

- When an underflow trap is not enabled, underflow is signaled only when both tininess and loss of accuracy have been detected. The delivered result might be zero, denormalized, or $2^{E_{min}}$.
- When an underflow trap is enabled (through the *FCSR* Enables field), underflow is signaled when tininess is detected regardless of loss of accuracy.

3.7.2.4 Overflow Exception

An Overflow exception is signaled when the magnitude of a rounded floating-point result (if the exponent range is unbounded) is larger than the destination format's largest finite number.

When no precise trap occurs, the result is determined by the rounding mode and the sign of the intermediate result.

3.7.2.5 Inexact Exception

An Inexact exception is signaled when one of the following occurs:

- The rounded result of an operation is not exact.
- The rounded result of an operation overflows without an overflow trap.
- When a denormal operand is flushed to zero.

3.7.2.6 Unimplemented Operation Exception

The Unimplemented Operation exception is a MIPS-defined exception that provides software emulation support. This exception is not IEEE-compliant.

The MIPS architecture is designed so that a combination of hardware and software can implement the architecture. Operations not fully supported in hardware cause an Unimplemented Operation exception, allowing software to perform the operation.

There is no enable bit for this condition; it always causes a trap (but the condition is effectively masked for all operations when FS=1). After the appropriate emulation or other operation is done in a software exception handler, the original instruction stream can be continued.

An Unimplemented Operation exception is taken in the following situations:

- when denormalized operands or tiny results are encountered for instructions not supporting denormal numbers and where such are not handed by the FS/FO/FN bits.

3.8 Pipeline and Performance

This section describes the structure and operation of the FPU pipeline.

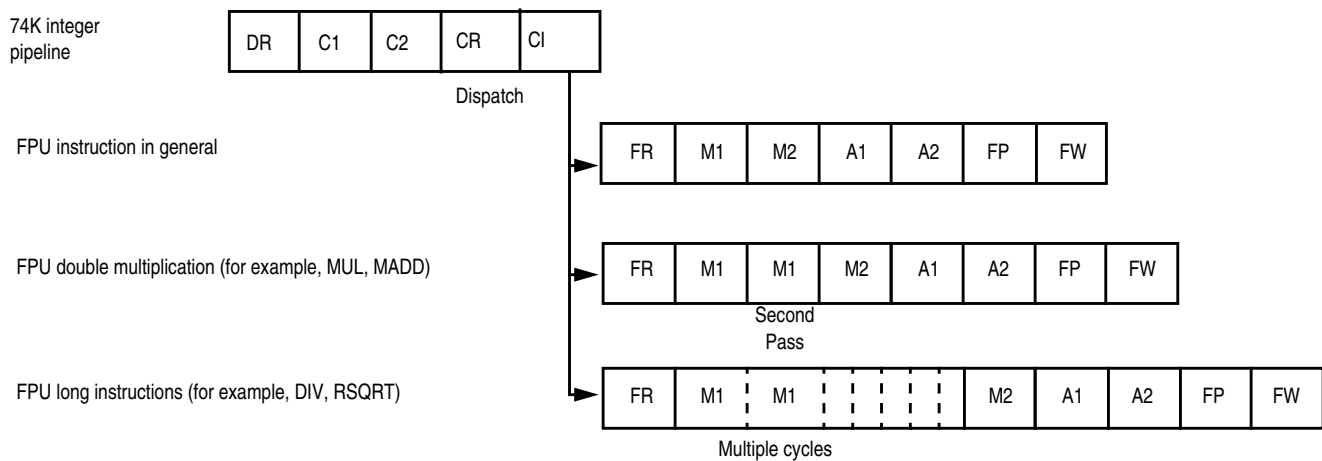
3.8.1 Pipeline Overview

The FPU has a seven stage pipeline to which the integer pipeline dispatches instructions. The FPU supports limited dual-issue. It can accept one “to/from” instruction and one arithmetic instruction every cycle. A “to/from” instruction is a data transfer instruction and covers the following instructions: MFC1, MFHC1, MTC1, MTHC1, CFC1, CTC1, LWC1, LWXC1, LUXC1, LDC1, LDXC1, SWC1, SWXC1, SDC1 and SDXC1. The arithmetic group refers to all other floating point instructions.

The FPU pipeline runs in parallel with the 74K integer pipeline. The FPU can be built to run at different frequencies compared to the integer core. The supported ratios of integer core clock to FPU clk are: 2:1, 3:2 and 1:1.

The FPU pipe is optimized for single-precision instructions, such that the basic multiply, ADD/SUB, and MADD/MSUB instructions can be performed with single-cycle throughput and low latency. Executing double-precision multiply and MADD/MSUB instructions requires a second pass through the M1 stage to generate all 64 bits of the product. Executing long latency instructions, such as DIV and RSQRT, extends the M1 stage. Figure 3.18 shows the FPU pipeline.

Figure 3.18 FPU Pipeline



3.8.1.1 DR Stage - Dispatch Rename

The DR stage is described in detail in [Chapter 2, “Pipeline of the 74K™ Core”](#). This stage is common to both the integer and floating point pipeline. The two pipelines fork off separately after this stage.

3.8.1.2 C1 - Coprocessor Interface Unit Stage 1

In this stage, the Coprocessor Interface Unit (CIU) receives two instructions from the IDU. It does some preliminary decoding and determines if there is space available in its internal queues for the received instructions.

3.8.1.3 CR Stage - Coprocessor Interface Unit Queue Read

The CIU internal instruction queues are read in this stage and up to two instructions are selected for dispatch to the FPU.

3.8.1.4 CI Stage - Coprocessor 1 Interface

This pipeline stage represents the interface stage, where instructions are sent to the FPU

3.8.1.5 FR Stage - Decode, Register Read, and Unpack

This pipeline is the second interface stage, where the instruction start is sent to the FPU.

The FR stage also has the following additional functionality:

- The dispatched instruction is decoded for register accesses.
- Data is read from the register file.
- The operands are unpacked into an internal format.

3.8.1.6 M1 Stage - Multiply Tree

The M1 stage has the following functionality:

- A single-cycle multiply array is provided for single-precision data format multiplication, and two cycles are provided for double-precision data format multiplication.
- The long instructions, such as divide and square root, iterate for several cycles in this stage.
- Sum of exponents is calculated.
- If an exception can be predicted, then it is sent out in this pipeline stage.

3.8.1.7 M2 Stage - Multiply Complete

The M2 stage has the following functionality:

- Multiplication is complete when the carry-save encoded product is compressed into binary.
- Rounding is performed.
- Exponent difference for addition path is calculated.

3.8.1.8 A1 Stage - Addition First Step

This stage performs the first step of the addition.

3.8.1.9 A2 Stage - Addition Second and Final Step

This stage performs the second and final step of the addition.

3.8.1.10 FP Stage - Result Pack

The FP stage has the following functionality:

- The result coming from the datapath is packed into IEEE 754 Standard format for the FPR register file.
- Overflow and underflow exceptional conditions are resolved.

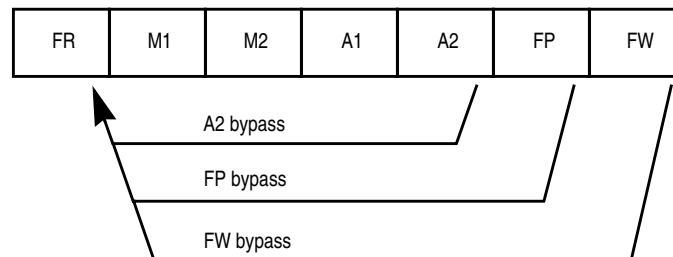
3.8.1.11 FW Stage - Register Write

The result is written to the FPR register file.

3.8.2 Bypassing

The FPU pipeline implements extensive bypassing, as shown in [Figure 3.19](#). Results do not need to be written into the register file and read back before they can be used, but can be forwarded directly to an instruction already in the pipe. Some bypassing is disabled when operating in 32-bit register file mode, the FP bit in the CP0 *Status* register is 0, due to the paired even-odd 32-bit registers that provide 64-bit registers.

Figure 3.19 Arithmetic Pipeline Bypass Paths



3.8.3 Repeat Rate and Latency

[Table 3.35](#) shows the repeat rate and latency for the FPU instructions. Note that cycles related to floating point operations are listed in terms of FPU clocks.

Table 3.35 74Kf Core FPU Latency and Repeat Rate

Opcode ¹	Latency (cycles)	Repeat Rate (cycles)
ABS.[S,D], NEG.[S,D], ADD.[S,D], SUB.[S,D], MUL.S, MADD.S, MSUB.S, NMADD.S, NMSUB.S	4	1
MUL.D, MADD.D, MSUB.D, NMADD.D, NMSUB.D	5	2
RECIP.S	13	10
RECIP.D	25	21
RSQRT.S	17	14
RSQRT.D	35	31
DIV.S, SQRT.S	17	14
DIV.D, SQRT.D	32	29

Table 3.35 74Kf Core FPU Latency and Repeat Rate (Continued)

Opcode ¹	Latency (cycles)	Repeat Rate (cycles)
C.cond.[S,D] to MOVF fmt and MOVT fmt instruction / MOVT, MOVN, BC1 instruction	1 / 2	1
CVT.D.S, CVT.[S,D].[W,L]	4	1
CVT.S.D	6	1
CVT.[W,L].[S,D], CEIL.[W,L].[S,D], FLOOR.[W,L].[S,D], ROUND.[W,L].[S,D], TRUNC.[W,L].[S,D]	5	1
MOV.[S,D], MOVF.[S,D], MOVN.[S,D], MOVT.[S,D], MOVZ.[S,D]	4	1
LWC1, LDC1, LDXC1, LUXC1, LWXC1	3	1
MTC1, MFC1	2	1

1. Format: S = Single, D = Double, W = Word, L = Longword.

The MIPS® DSP Application-Specific Extension to the MIPS32® Instruction Set

The 74K core includes support for the MIPS DSP Application-Specific Extension (ASE) Revision 2 that provides enhanced performance capabilities for a wide range of signal-processing applications, with computational support for fractional data types, SIMD, saturation, and other operations that are commonly used in these applications.

Refer to *MIPS32® Architecture For Programmers, Volume IV-e* for a general description of the DSP ASE and detailed descriptions of the DSP instructions. Additional programming information is contained in *Programming the MIPS 74K Family Cores for DSP* and in the DSP chapter of *Programming the MIPS32® 74K™ Coherent Processing System Family*.

4.1 Additional Register State for the DSP ASE

The DSP ASE defines three additional accumulator registers and one additional control/status register, as described below. These registers require the operating system to recognize the presence of the DSP ASE and to include these additional registers in the context save and restore operations.

4.1.1 HI-LO Registers

The DSP ASE includes three HI/LO accumulator register pairs (ac1, ac2, and ac3) in addition to the HI/LO register pair (ac0) in the standard MIPS32 architecture. These registers improve the parallelization of independent accumulation routines—for example, filter operations, convolutions, etc. DSP instructions that target the accumulators use two instruction bits to specify the destination accumulator, with the zero value referring to the original accumulator.

4.1.2 DSP Control Register

The *DSPControl* register contains control and status information used by DSP instructions. [Figure 4.1](#) illustrates the bits in this register, and [Table 4.1](#) describes their usage.

Figure 4.1 MIPS32® DSP ASE Control Register (DSPControl) Format

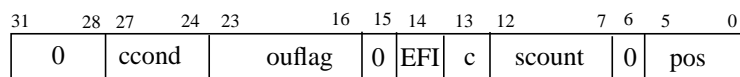


Table 4.1 MIPS® DSP ASE Control Register (DSPControl) Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
0	31:28	Reserved. Used in the MIPS64 architecture but not used in the MIPS32 architecture. Must be written as zero; returns zero on read.	0	0	Required
<i>ccond</i>	27:24	Condition code bits set by compare instructions. The compare instruction sets the right-most bits as required by the number of elements in the vector compare. Bits not set by the instruction remain unchanged.	R/W	0	Required
<i>ouflag</i>	23:16	This field is written by hardware when certain instructions overflow or underflow and may have been saturated. See Table 4.2 for a full list of which bits are set by what instructions.	R/W	0	Required
<i>EFI</i>	14	Extract Fail Indicator. This bit is set to 1 when an EXTP, EXTPV, EXTPDP, or EXTPDPV instruction fails. These instructions fail when there are insufficient bits to extract, that is, when the value of <i>pos</i> in <i>DSPControl</i> is less than the value of <i>size</i> specified in the instruction. This bit is not sticky, so each invocation of one of the four instructions will reset the bit depending on whether or not the instruction failed.	R/W	0	Required
<i>c</i>	13	Carry bit. This bit is set and used by special add instructions that implement a 64-bit add across two GPRs. The ADDSC instruction sets the bit and the ADDWC instruction uses this bit.	R/W	0	Required
<i>scount</i>	12:7	This field is for use by the INSV instruction. The value of this field is used to specify the size of the bit field to be inserted.	R/W	0	Required
<i>pos</i>	5:0	This field is used by the variable insert instructions INSV to specify the insert position. It is also used to indicate the extract position for the EXTP, EXTPV, EXTPDP, and EXTPDPV instructions. The decrement <i>pos</i> (DP) variants of these instructions on completion will have decremented the value of <i>pos</i> (by the size amount). The MTHLIP instruction will increment the <i>pos</i> value by 32 after copying the value of <i>LO</i> to <i>HI</i> .	R/W	0	Required
0	15:13	Must be written as zero; returns zero on read.	0	0	Reserved

The bits of the overflow flag `ouflag` field in the `DSPControl` register are set by a number of instructions, as described in Table 4.2. These bits are sticky and can be reset only by an explicit write to these bits in the register (using the `WRDSP` instruction).

Table 4.2 DSPControl ouflag Bits

Bit Number	Description
16	This bit is set when the destination is accumulator (<i>HI-LO</i> pair) zero, and an operation overflow or underflow occurs. These instructions are: <code>DPAQ_S</code> , <code>DPAQ_SA</code> , <code>DPSQ_S</code> , <code>DPSQ_SA</code> , <code>DPAQX_S</code> , <code>DPAQX_SA</code> , <code>DPSQX_S</code> , <code>DPSQX_SA</code> , <code>MAQ_S</code> , <code>MAQ_SA</code> and <code>MULSAQ_S</code> .
17	Same instructions as above, when the destination is accumulator (<i>HI-LO</i> pair) one.
18	Same instructions as above, when the destination is accumulator (<i>HI-LO</i> pair) two.
19	Same instructions as above, when the destination is accumulator (<i>HI-LO</i> pair) three.
20	Instructions that set this bit on an overflow/underflow: <code>ABSQ_S</code> , <code>ADDQ</code> , <code>ADDQ_S</code> , <code>ADDU</code> , <code>ADDU_S</code> , <code>ADDWC</code> , <code>SUBQ</code> , <code>SUBQ_S</code> , <code>SUBU</code> and <code>SUBU_S</code> .
21	Instructions that set this bit on an overflow/underflow: <code>MUL</code> , <code>MUL_S</code> , <code>MULEQ_S</code> , <code>MULEU_S</code> , <code>MULQ_RS</code> , and <code>MULQ_S</code> .
22	Instructions that set this bit on an overflow/underflow: <code>PRECRQ_RS</code> , <code>SHLL</code> , <code>SHLL_S</code> , <code>SHLLV</code> , and <code>SHLLV_S</code> .
23	Instructions that set this bit on an overflow/underflow: <code>EXTR</code> , <code>EXTR_S</code> , <code>EXTR_RS</code> , <code>EXTRV</code> , and <code>EXTRV_RS</code> .

4.2 Software Detection of the DSP ASE Revision 2

The presence of the MIPS DSP ASE in the 74K core is indicated by two static bits in the `Config3` register: the `DSPP` (*DSP Present*) bit indicates the presence of the DSP ASE, and the `DSP2P` (*DSP Rev2 Present*) bit indicates the presence of the MIPS DSP ASE Rev2. Because all members of the 74K core family support both ASEs, these bits are always set to 1.

The `MX` (*DSP ASE Enable*) read/write bit in the `CP0 Status` register must be set to enable access to the extra instructions defined by the DSP ASE, as well as to the `MTLO/HI`, `MFLO/HI` instructions that access accumulators `ac1`, `ac2`, and `ac3`. Executing a DSP ASE instruction or the `MTLO/HI`, `MFLO/HI` instructions with this bit set to zero causes a DSP State Disabled Exception (exception code 26 in the `CP0 Cause` register). This exception can be used by system software to do lazy context-switching.

Memory Management of the 74K™ Core

The 74K processor core includes a Memory Management Unit (MMU) that interfaces between the execution unit and the cache controller. The core contains either a Translation Lookaside Buffer (TLB) or a simpler Fixed Mapping (FM)-style MMU, specified as a build-time option when the core is implemented.

This chapter contains the following sections:

- [Section 5.1 “Introduction”](#)
- [Section 5.2 “Modes of Operation”](#)
- [Section 5.3 “Translation Lookaside Buffer”](#)
- [Section 5.4 “Virtual-to-Physical Address Translation”](#)
- [Section 5.5 “Fixed Mapping MMU”](#)

5.1 Introduction

The MMU in a 74K processor core translates a virtual address to a physical address before the request is sent to the cache controllers for tag comparison or to the bus interface unit for an external memory reference. Virtual-to-physical address translation is especially useful for operating systems that must manage physical memory to accommodate multiple tasks active in the same memory, and possibly in the same virtual address space (though, of course, in different locations in physical memory). The MMU also enforces the protection of memory areas and defines the cache protocols.

By default, the MMU is TLB-based. The TLB consists of two address-translation buffers: a dual-ported 16/32/48/64 dual-entry fully associative Joint TLB (JTLB) and a 4-entry instruction micro TLB (ITLB). When an instruction address is translated, the ITLB is accessed first, and if the translation is not found, the JTLB is accessed. If there is a miss in the JTLB, an exception is taken. When a data reference is translated, the JTLB is accessed directly. If the address is not present in the JTLB, an exception is taken. The JTLB is dual-ported to prevent contention between instruction and data accesses.

Optionally, the MMU can implement a Fixed Mapping (FM) mechanism, based on a simple algorithm that translates virtual addresses into physical addresses. These translations are different for various regions of the virtual address space.

[Figure 5.1](#) shows how the MMU with TLB interacts with cache accesses, and [Figure 5.2](#) shows the equivalent for the FM MMU.

Figure 5.1 Address Translation For Cache Access with TLB MMU

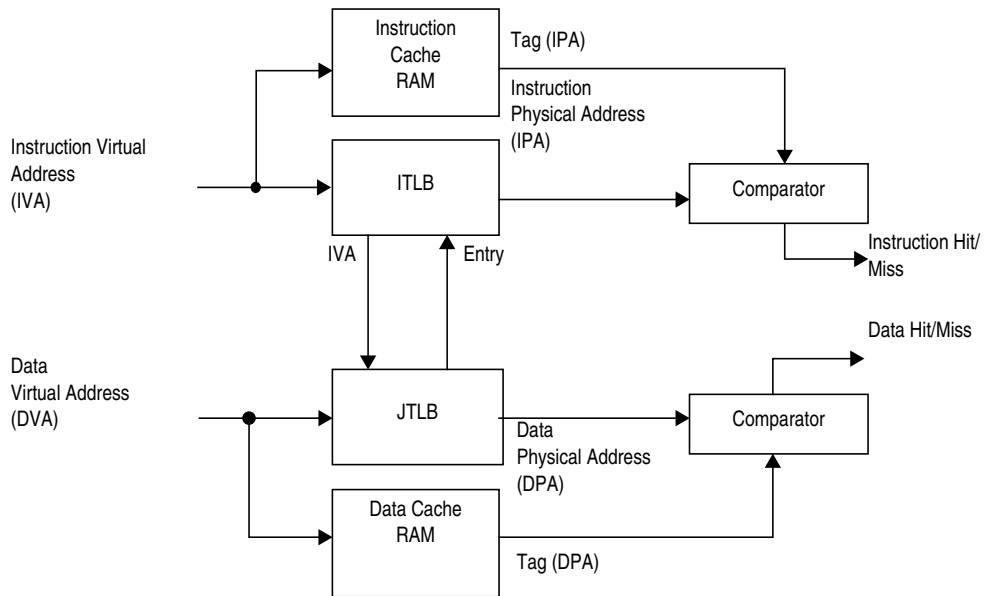
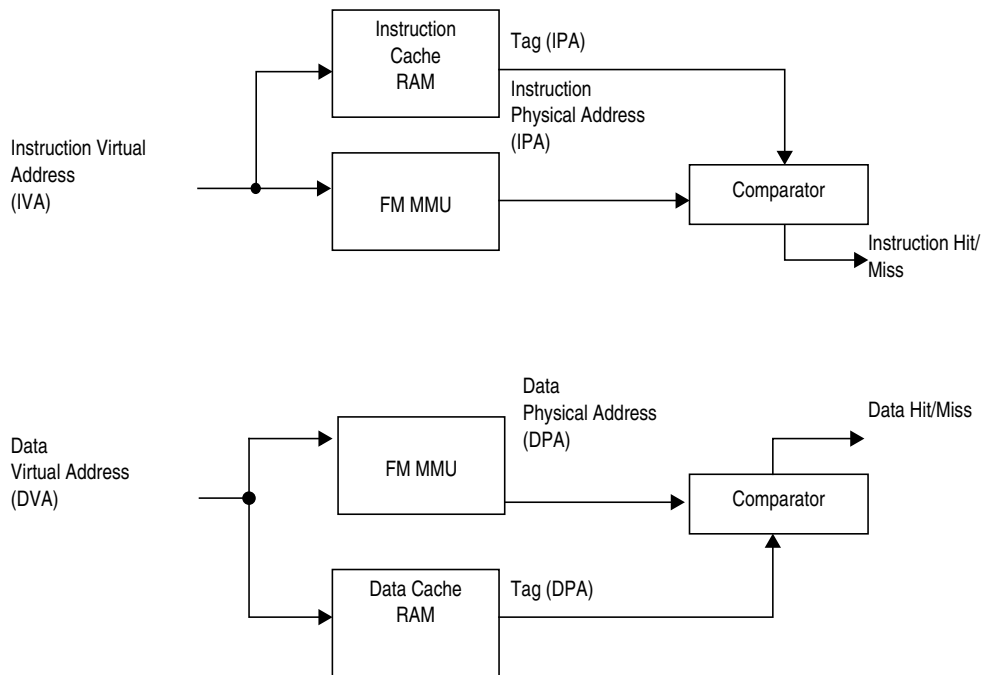


Figure 5.2 Address Translation For Cache Access with FM MMU



5.2 Modes of Operation

The MMU's virtual-to-physical address translation is determined by the mode in which the processor is operating. A 74K processor core operates in one of four modes:

- User mode
- Supervisor mode
- Kernel mode
- Debug mode

User mode is most often used for application programs. Supervisor mode is an intermediate privilege level with access to an additional region of memory and is only supported with the TLB-based MMU. Kernel mode is typically used for handling exceptions and privileged operating system functions, including CP0 management and I/O device accesses. Debug mode is used for software debugging and usually occurs within a software development tool.

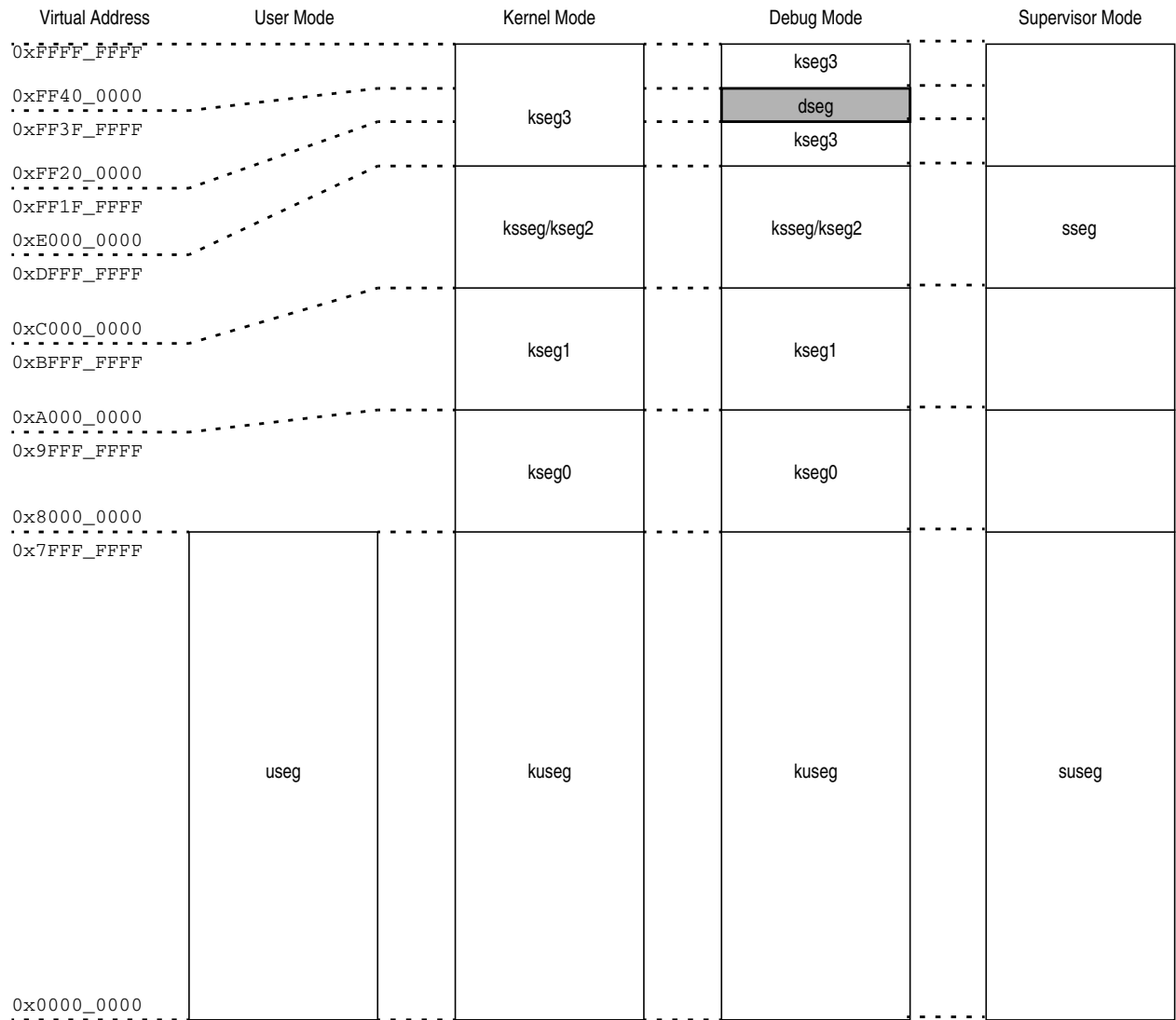
5.2.1 Virtual Memory Segments

The MIPS32 architecture supports a 4 GByte virtual address space that is partitioned into a number of segments, each characterized by a set of attributes defined by hardware and software. The virtual memory segments are different depending on the mode of operation. [Figure 5.3](#) shows the segmentation for the 4 GByte (2^{32} bytes) virtual memory space, addressed by a 32-bit virtual address, for each of the four modes.

User mode accesses are limited to a subset of the virtual address space (0x0000_0000 to 0x7FFF_FFFF) and can be inhibited from accessing CP0 functions. In User mode, virtual addresses 0x8000_0000 to 0xFFFF_FFFF are invalid and cause an exception if accessed. Supervisor mode adds access to sseg (0xC000_0000 to 0xDFFF_FFFF). kseg0, kseg1, and kseg3 will still cause exceptions if they are accessed. In Kernel mode, software has access to the entire address space, as well as all CP0 registers.

Debug mode is entered on a debug exception. While in Debug mode, the debug software has access to the same address space and CP0 registers as Kernel mode. In addition, while in Debug mode, the core has access to the debug segment (dseg). This area overlays part of the kernel segment kseg3. Access to dseg in Debug mode can be turned on or off, allowing full access to the entire kseg3 in Debug mode, if so desired.

Figure 5.3 74K™ Processor Core Virtual Memory Map



Segments can be mapped or unmapped, as described in the following subsections.

5.2.1.1 Unmapped Segments

An unmapped segment does not use the TLB or the FM to translate virtual to physical addresses. Especially after reset, it is important to have unmapped memory segments, because the TLB is not yet programmed to perform the translation.

Unmapped segments have a fixed simple translation from virtual to physical address. This is much like the translations the FM provides for the core, but we will still make the distinction.

Except for kseg0, unmapped segments are always uncached. The cacheability of kseg0 is set in the K0 field of the CP0 Config register (see Section 7.2.23 “Config (CP0 Register 16, Select 0): Legacy Configuration Register”).

5.2.1.2 Mapped Segments

A mapped segment uses the TLB or the FM to translate from virtual to physical addresses.

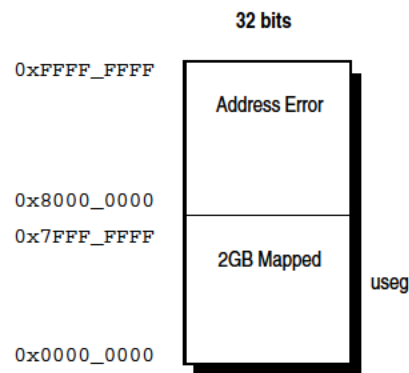
For the core with TLB, the translation of mapped segments is handled on a per-page basis. Included in this translation is information defining whether the page is cacheable or not, and the protection attributes that apply to the page.

For the core with the FM MMU, the mapped segments have a fixed translation from virtual to physical address. The cacheability of the segment is defined in the K23 and KU fields of the CP0 register *Config* (see [Section 7.2.23 “Config \(CP0 Register 16, Select 0\): Legacy Configuration Register”](#)). Write protection of segments is not possible during FM translation.

5.2.2 User Mode

In user mode, a single 2 GByte (2^{31} bytes) uniform virtual address space, called the user segment (useg), is available. [Figure 5.4](#) shows the location of user mode virtual address space.

Figure 5.4 User Mode Virtual Address Space



The user segment starts at address 0x0000_0000 and ends at address 0x7FFF_FFFF. Accesses to all other addresses cause an address error exception.

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

- *KSU* = 0b10
- *EXL* = 0
- *ERL* = 0

In addition to the above values, the *DM* bit in the *Debug* register must be 0.

Table 5.1 lists the characteristics of the User mode segment.

Table 5.1 User Mode Segments

Address-Bit Value	Status Register			Segment Name	Address Range	Segment Size
	Bit Value					
	EXL	ERL	KSU			
32-bit A(31) = 0	0	0	0b10	useg	0x0000_0000 --> 0x7FFF_FFFF	2 GByte (2 ³¹ bytes)

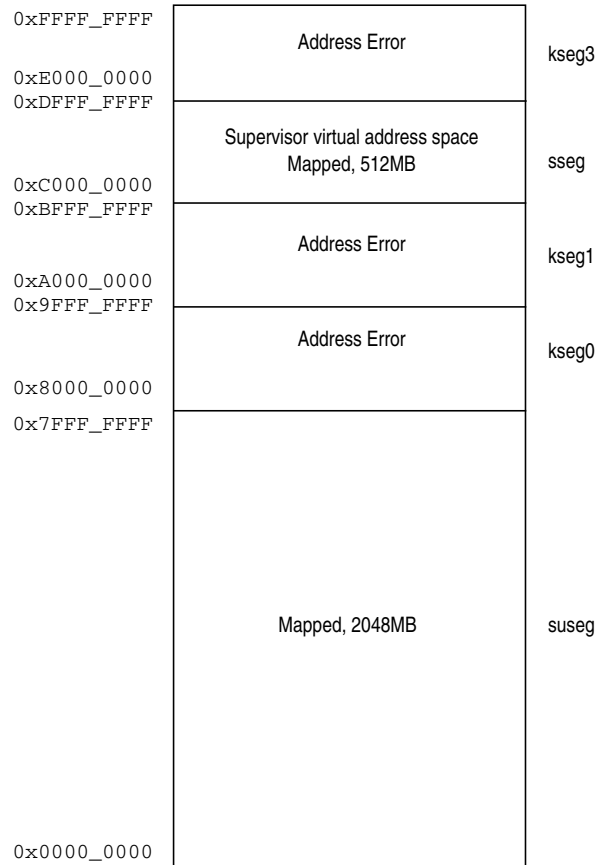
All valid user mode virtual addresses have their most-significant bit cleared to 0, indicating that user mode can only access the lower half of the virtual memory map. All attempts to reference an address with the most-significant bit set while in user mode causes an address error exception.

All references to useg are mapped through the TLB or FM. For cores with a TLB, the virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address before translation. Also, bit settings within the TLB entry for the page determine the cacheability of a reference. For FM MMU cores, the cacheability is set via the KU field of the CP0 *Config* register.

5.2.3 Supervisor Mode

In supervisor mode, two uniform virtual address spaces are available: a 2 GByte (2³¹ bytes) virtual address space called the supervisor user segment (suseg), and a 512 MByte virtual address space called the supervisor segment (sseg). The supervisor-mode virtual address space is shown in Figure 5.5.

Figure 5.5 Supervisor Mode Virtual Address Space



The supervisor user segment begins at address 0x0000_0000 and ends at address 0x7FFF_FFFF. The supervisor segment begins at 0xC000_0000 and ends at 0xDFFF_FFFF. Accesses to all other addresses in Supervisor mode cause an address error exception.

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

- $UM = 0$ and $SM = 1$
- $EXL = 0$
- $ERL = 0$

In addition to the above values, the *DM* bit in the *Debug* register must be 0.

Table 5.2 lists the characteristics of the Supervisor mode segments.

Table 5.2 Supervisor Mode Segments

Address-Bit Value	Status Register				Segment Name	Address Range	Segment Size
	Bit Value						
	EXL	ERL	UM	SM			
32-bit A(31) = 0	0	0	0	1	suseg	0x0000_0000 --> 0x7FFF_FFFF	2 GByte (2 ³¹ bytes)
32-bit A(31:29) = 110 ₂	0	0	0	1	sseg	0xC000_0000 -> 0xDFFF_FFFF	512MB (2 ²⁹ bytes)

The system maps all references to *suseg* and *sseg* through the TLB or FM. For cores with a TLB, the virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address before translation. Also, bit settings within the TLB entry for the page determine the cacheability of a reference. For FM MMU cores, the cacheability of *suseg* and *sseg* is set via the KU and K23 fields of the CP0 *Config* register respectively.

5.2.4 Kernel Mode

The processor operates in Kernel mode when the *DM* bit in the *Debug* register is 0 and the *Status* register contains one or more of the following values:

- *KSU* = 0b00
- *ERL* = 1
- *EXL* = 1

When a non-debug exception is detected, *EXL* or *ERL* will be set and the processor will enter Kernel mode. At the end of the exception handler routine, an Exception Return (ERET) instruction is generally executed. The ERET instruction jumps to the Exception PC, clears *ERL*, and clears *EXL* if *ERL*=0. This may return the processor to User mode.

In Kernel mode, a program has access to the entire virtual address space. Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 5.6. The characteristics of kernel-mode segments are listed in Table 5.3.

The core enters Kernel mode both at reset and when an exception is recognized.

Figure 5.6 Kernel Mode Virtual Address Space

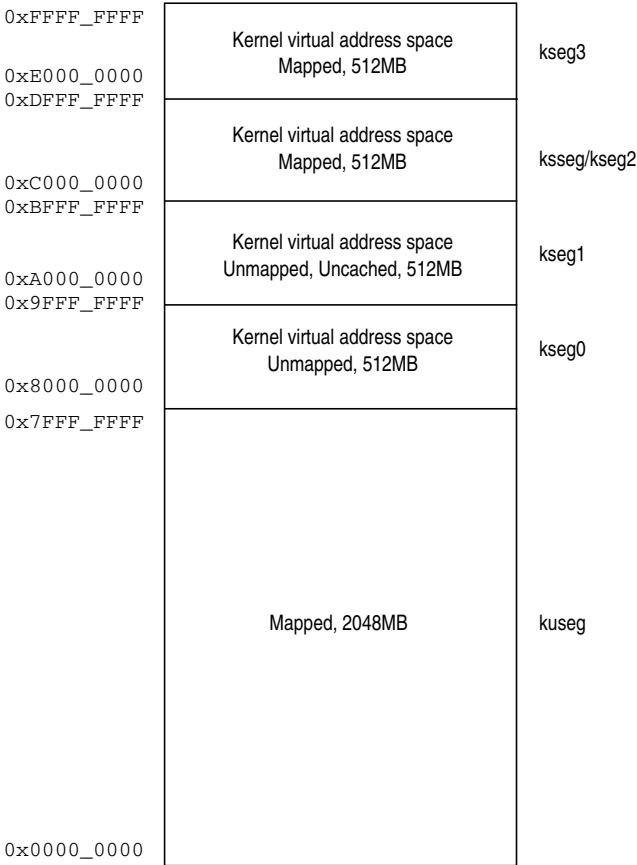


Table 5.3 Kernel Mode Segments

Address-Bit Values	Status Register Is One of These Values			Segment Name	Address Range	Segment Size
	KSU	EXL	ERL			
A(31) = 0	(KSU = 00 ₂ or EXL = 1 or ERL = 1) and DM = 0			kuseg	0x0000_0000 through 0x7FFF_FFFF	2 GBytes (2 ³¹ bytes)
A(31:29) = 100 ₂				kseg0	0x8000_0000 through 0x9FFF_FFFF	512 MBytes (2 ²⁹ bytes)
A(31:29) = 101 ₂				kseg1	0xA000_0000 through 0xBFFF_FFFF	512 MBytes (2 ²⁹ bytes)
A(31:29) = 110 ₂				ksseg/kseg2	0xC000_0000 through 0xDFFF_FFFF	512 MBytes (2 ²⁹ bytes)
A(31:29) = 111 ₂				kseg3	0xE000_0000 through 0xFFFF_FFFF	512 MBytes (2 ²⁹ bytes)

5.2.4.1 Kernel Mode, User Space (kuseg)

In Kernel mode, when the most-significant bit of the virtual address (A31) is cleared, the 32-bit kuseg virtual address space is selected and covers the full 2³¹ bytes (2 GBytes) of the current user address space mapped to addresses 0x0000_0000 - 0x7FFF_FFFF. For cores with TLBs, the virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

When ERL = 1 in the *Status* register, the user address region becomes a 2³¹-byte unmapped and uncached address space. While in this setting, the kuseg virtual address maps directly to the same physical address, and does not include the ASID field.

5.2.4.2 Kernel Mode, Kernel Space 0 (kseg0)

In Kernel mode, when the most-significant three bits of the virtual address are 100₂, 32-bit kseg0 virtual address space is selected; it is the 2²⁹-byte (512-MByte) kernel virtual space located at addresses 0x8000_0000 - 0x9FFF_FFFF. References to kseg0 are unmapped; the physical address selected is defined by subtracting 0x8000_0000 from the virtual address. The K0 field of the *Config* register controls cacheability.

5.2.4.3 Kernel Mode, Kernel Space 1 (kseg1)

In Kernel mode, when the most-significant three bits of the 32-bit virtual address are 101₂, kseg1 virtual address space is selected. kseg1 is the 2²⁹-byte (512-MByte) kernel virtual space located at addresses 0xA000_0000 - 0xBFFF_FFFF. References to kseg1 are unmapped; the physical address selected is defined by subtracting 0xA000_0000 from the virtual address. Caches are disabled for accesses to these addresses, and physical memory (or memory-mapped I/O device registers) are accessed directly.

5.2.4.4 Kernel Mode, Kernel/Supervisor Space 2 (ksseg/kseg2)

In Kernel mode, when $KSU = 00_2$, $ERL = 1$, or $EXL = 1$ in the *Status* register, and $DM = 0$ in the *Debug* register, and the most-significant three bits of the 32-bit virtual address are 110_2 , 32-bit kseg2 virtual address space is selected.

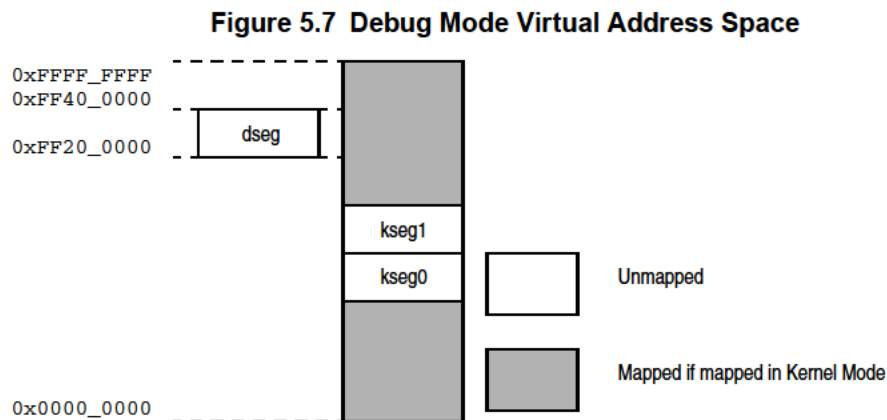
With the FM MMU, this 2^{29} -byte (512-MByte) kernel virtual space is located at physical addresses $0xC000_0000 - 0xDFFF_FFFF$. Otherwise, this space is mapped through the TLB.

5.2.4.5 Kernel Mode, Kernel Space 3 (kseg3)

In Kernel mode, when the most-significant three bits of the 32-bit virtual address are 111_2 , the kseg3 virtual address space is selected. With the FM MMU, this 2^{29} -byte (512-MByte) kernel virtual space is located at physical addresses $0xE000_0000 - 0xFFFF_FFFF$. Otherwise, this space is mapped through the TLB.

5.2.5 Debug Mode

Except for kseg3, debug-mode address space is identical to kernel-mode address space with respect to mapped and unmapped areas. In kseg3, a debug segment (dseg) coexists in the virtual address range $0xFF20_0000$ to $0xFF3F_FFFF$. The layout is shown in Figure 5.7.



dseg is subdivided into the dmseg segment at $0xFF20_0000$ to $0xFF2F_FFFF$, which is used when the debug probe services the memory segment, and the drseg segment at $0xFF30_0000$ to $0xFF3F_FFFF$, which is used when memory-mapped debug registers are accessed. The subdivision and attributes of the segments are shown in Table 5.4.

Accesses to memory that would normally cause an exception in kernel mode cause the core to re-enter debug mode via a debug-mode exception. This includes accesses usually causing a TLB exception, with the result that such accesses are not handled by the usual memory-management routines.

Memory Management of the 74K™ Core

The unmapped kseg0 and kseg1 segments from kernel-mode address space are available in debug mode, which allows the debug handler to be executed from uncached, unmapped memory.

Table 5.4 Physical Address and Cache Attributes for dseg, dmseg, and drseg

Segment Name	Sub-Segment Name	Virtual Address	Generates Physical Address	Cache Attribute
dseg	dmseg	0xFF20_0000 through 0xFF2F_FFFF	dmseg maps to addresses 0x0_0000 - 0xF_FFFF in EJTAG probe memory space.	Uncached
	drseg	0xFF30_0000 through 0xFF3F_FFFF	drseg maps to the breakpoint registers 0x0_0000 - 0xF_FFFF	

5.2.5.1 Debug Mode, Register (drseg)

The behavior of CPU access to the drseg address range at 0xFF30_0000 to 0xFF3F_FFFF is determined as shown in [Table 5.5](#)

Table 5.5 CPU Access to drseg

Transaction	LSNM Bit in Debug Register	Access
Load / Store	1	Kernel mode address space (kseg3)
Fetch	Don't care	drseg, see comments below
Load / Store	0	

Debug software is expected to read the *Debug Control* register (*DCR*) to determine which other memory-mapped registers exist in drseg. The value returned in response to a read of any unimplemented memory-mapped register is unpredictable, and writes are ignored to any unimplemented register in drseg. For more information about the *DCR*, refer to [Chapter 11, “EJTAG Debug Support in the 74K™ Core”](#).

The allowed access size is limited for the drseg. Only word-size transactions are allowed. Operation of the processor is undefined for other transaction sizes.

5.2.5.2 Debug Mode, Memory (dmseg)

The conditions for CPU accesses to the dmseg address range (0xFF20_0000 to 0xFF2F_FFFF) are shown in [Table 5.6](#).

Table 5.6 CPU Access to dmseg

Transaction	ProbEn Bit in DCR Register	LSNM Bit in Debug Register	Access
Load / Store	Don't care	1	Kernel mode address space (kseg3)
Fetch	1	Don't care	dmseg
Load / Store	1	0	dmseg
Fetch	0	Don't care	See comments below
Load / Store	0	0	See comments below

An attempt to access `dmseg` when the `ProbEn` bit in the DCR register is 0 should not happen, because debug software is expected to check the state of the `ProbEn` bit in DCR register before attempting to reference `dmseg`. If such a reference does occur, the reference hangs until it is satisfied by the probe. The probe must not assume that there will never be a reference to `dmseg` when the `ProbEn` bit in the DCR register is 0, because there is an inherent race between the debug software sampling the `ProbEn` bit as 1, and the probe clearing it to 0.

5.3 Translation Lookaside Buffer

The TLB memory-management scheme used in the 74Kc processor core includes two address-translation units:

- 16, 32, 48, or 64 dual-entry fully associative Joint TLB (JTLB)
- 4-entry fully associative Instruction micro TLB (ITLB)

5.3.1 Joint TLB

The 74K core implements a 16-64 dual-entry, fully associative Joint TLB that maps 32-128 virtual pages to their corresponding physical addresses. The purpose of the TLB is to translate virtual addresses and their corresponding ASID into a physical memory address. The translation is performed by comparing the upper bits of the virtual address (along with the ASID bits) against each of the entries in the *tag* portion of the JTLB structure. Because this structure is used to translate both instruction and data virtual addresses, it is referred to as a “joint” TLB.

The JTLB is organized as 16-64 pairs of even and odd entries containing descriptions of pages that range in size from 4-KBytes to 256MBytes into the 4-GByte physical address space.

The JTLB is organized in pairs of page entries to minimize its overall size. Each virtual *tag* entry corresponds to two physical data entries, an even page entry and an odd page entry. The highest order virtual address bit not participating in the tag comparison is used to determine which of the two data entries is used. Since page size can vary on a page-pair basis, the determination of which address bits participate in the comparison and which bit is used to make the even-odd selection must be done dynamically during the TLB lookup.

Figure 5.8 shows the contents of one of the dual-entries in the JTLB. The bit ranges shown in the figure serve to clarify which address bits are (or may be) affected during the translation process.

Figure 5.8 JTLB Entry (Tag and Data)

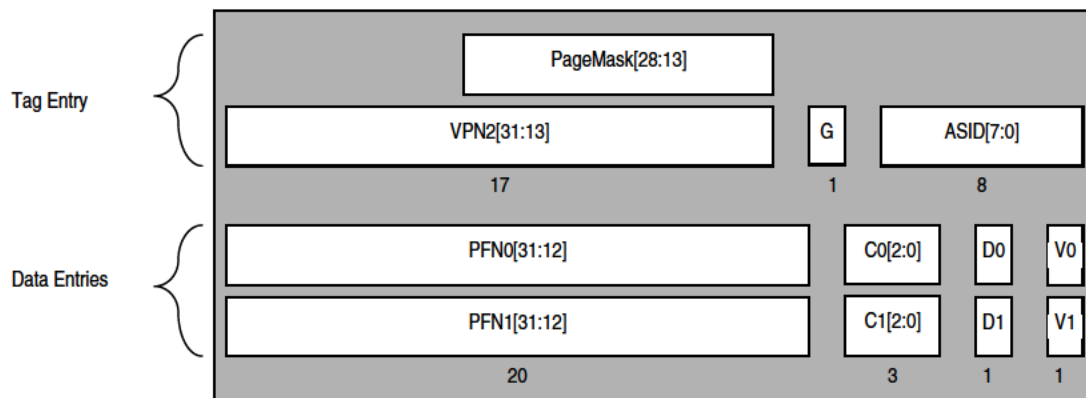


Table 5.7 and Table 5.8 explain each of the fields in a JTLB entry.

Table 5.7 TLB Tag Entry Fields

Field Name	Description																														
PageMask[28:13]	<p>Page Mask Value. The Page Mask defines the page size by masking the appropriate VPN2 bits from being involved in a comparison. It is also used to determine which address bit is used to make the even-odd page (PFN0-PFN1) determination. See the table below.</p> <table border="1"> <thead> <tr> <th>PageMask</th> <th>Page Size</th> <th>Even/Odd Bank Select Bit</th> </tr> </thead> <tbody> <tr> <td>00_0000_0000_0000_00</td> <td>4KB</td> <td>VAddr[12]</td> </tr> <tr> <td>00_0000_0000_0000_11</td> <td>16KB</td> <td>VAddr[14]</td> </tr> <tr> <td>00_0000_0000_0011_11</td> <td>64KB</td> <td>VAddr[16]</td> </tr> <tr> <td>00_0000_0000_1111_11</td> <td>256KB</td> <td>VAddr[18]</td> </tr> <tr> <td>00_0000_0011_1111_11</td> <td>1MB</td> <td>VAddr[20]</td> </tr> <tr> <td>00_0000_1111_1111_11</td> <td>4MB</td> <td>VAddr[22]</td> </tr> <tr> <td>00_0011_1111_1111_11</td> <td>16MB</td> <td>VAddr[24]</td> </tr> <tr> <td>00_1111_1111_1111_11</td> <td>64MB</td> <td>VAddr[26]</td> </tr> <tr> <td>11_1111_1111_1111_11</td> <td>256MB</td> <td>VAddr[28]</td> </tr> </tbody> </table> <p>The PageMask column above shows all the legal values for PageMask. Because each pair of bits can only have the same value, the physical entry in the JTLB will only save a compressed version of the <i>PageMask</i> using only 8 bits. This is however transparent to software, which will always work with a 16 bit field</p>	PageMask	Page Size	Even/Odd Bank Select Bit	00_0000_0000_0000_00	4KB	VAddr[12]	00_0000_0000_0000_11	16KB	VAddr[14]	00_0000_0000_0011_11	64KB	VAddr[16]	00_0000_0000_1111_11	256KB	VAddr[18]	00_0000_0011_1111_11	1MB	VAddr[20]	00_0000_1111_1111_11	4MB	VAddr[22]	00_0011_1111_1111_11	16MB	VAddr[24]	00_1111_1111_1111_11	64MB	VAddr[26]	11_1111_1111_1111_11	256MB	VAddr[28]
PageMask	Page Size	Even/Odd Bank Select Bit																													
00_0000_0000_0000_00	4KB	VAddr[12]																													
00_0000_0000_0000_11	16KB	VAddr[14]																													
00_0000_0000_0011_11	64KB	VAddr[16]																													
00_0000_0000_1111_11	256KB	VAddr[18]																													
00_0000_0011_1111_11	1MB	VAddr[20]																													
00_0000_1111_1111_11	4MB	VAddr[22]																													
00_0011_1111_1111_11	16MB	VAddr[24]																													
00_1111_1111_1111_11	64MB	VAddr[26]																													
11_1111_1111_1111_11	256MB	VAddr[28]																													
VPN2[31:13]	Virtual Page Number divided by 2. This field contains the upper bits of the virtual page number. Because it represents a pair of TLB pages, it is divided by 2. Bits 31:29 are always included in the TLB lookup comparison. Bits 28:13 are included depending on the page size, defined by PageMask																														
G	Global Bit. When set, indicates that this entry is global to all processes and/or threads and thus disables inclusion of the ASID in the comparison.																														
ASID[7:0]	Address Space Identifier. Identifies which process or thread this TLB entry is associated with.																														

Table 5.8 TLB Data Entry Fields

Field Name	Description
PFN0[31:12], PFN1[31:12]	Physical Frame Number. Defines the upper bits of the physical address.

Table 5.8 TLB Data Entry Fields (Continued)

Field Name	Description																		
C0[2:0], C1[2:0]	Cacheability. Contains an encoded value of the cacheability attributes and determines whether the page should be placed in the cache or not. The field is encoded as follows: <table border="1" data-bbox="560 367 1347 703"> <thead> <tr> <th>C[2:0]</th> <th>Coherency Attribute</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Cacheable, noncoherent, write-through, no write-allocate</td> </tr> <tr> <td>1</td> <td>Reserved</td> </tr> <tr> <td>2</td> <td>Uncached</td> </tr> <tr> <td>3</td> <td>Cacheable, noncoherent, write-back, write-allocate</td> </tr> <tr> <td>4</td> <td>Reserved</td> </tr> <tr> <td>5</td> <td>Reserved</td> </tr> <tr> <td>6</td> <td>Reserved</td> </tr> <tr> <td>7</td> <td>Uncached Accelerated</td> </tr> </tbody> </table>	C[2:0]	Coherency Attribute	0	Cacheable, noncoherent, write-through, no write-allocate	1	Reserved	2	Uncached	3	Cacheable, noncoherent, write-back, write-allocate	4	Reserved	5	Reserved	6	Reserved	7	Uncached Accelerated
C[2:0]	Coherency Attribute																		
0	Cacheable, noncoherent, write-through, no write-allocate																		
1	Reserved																		
2	Uncached																		
3	Cacheable, noncoherent, write-back, write-allocate																		
4	Reserved																		
5	Reserved																		
6	Reserved																		
7	Uncached Accelerated																		
D0, D1	“Dirty” or Write-enable Bit. Indicates that the page has been written and/or is writable. If this bit is set, stores to the page are permitted. If the bit is cleared, stores to the page cause a TLB Modified exception.																		
V0, V1	Valid Bit. Indicates that the TLB entry and, thus, the virtual page mapping are valid. If this bit is set, accesses to the page are permitted. If the bit is cleared, accesses to the page cause a TLB Invalid exception.																		

In order to fill an entry in the JTLB, software executes a TLBWI or TLBWR instruction (see [Section 5.4.3 “TLB Instructions”](#)). Prior to invoking one of these instructions, several CP0 registers must be updated with the information to be written to a TLB entry:

- PageMask is set in the CP0 *PageMask* register.
- VPN2, and ASID are set in the CP0 *EntryHi* register.
- PFN0, C0, D0, V0, and G bits are set in the CP0 *EntryLo0* register.
- PFN1, C1, D1, V1, and G bits are set in the CP0 *EntryLo1* register.

Note that the global bit “G” is part of both *EntryLo0* and *EntryLo1*. The resulting “G” bit in the JTLB entry is the logical AND between the two fields in *EntryLo0* and *EntryLo1*. Please refer to [Chapter 7, “CP0 Registers of the 74K™ Core”](#) for further details.

The address space identifier (ASID) helps to reduce the frequency of TLB flushing on a context switch. The existence of the ASID allows multiple processes to exist in both the TLB and instruction caches. The ASID value is stored in the *EntryHi* register and is compared to the ASID value of each entry.

5.3.2 Instruction TLB

The ITLB is a 4-entry, fully-associative TLB dedicated to performing translations for the instruction stream. The ITLB only maps 4-Kbyte pages/sub-pages or 1-Mbyte pages/sub-pages.

The ITLB is managed by hardware and is transparent to software. If a fetch address cannot be translated by the ITLB, the JTLB is accessed trying to translate it in the following clock cycles. If successful, the translation information is copied into the ITLB and bypassed to the tag comparators. This results in an ITLB miss penalty of at least 2 cycles. Depending on the JTLB implementation or if it is busy with other operations, it may take additional cycles.

5.4 Virtual-to-Physical Address Translation

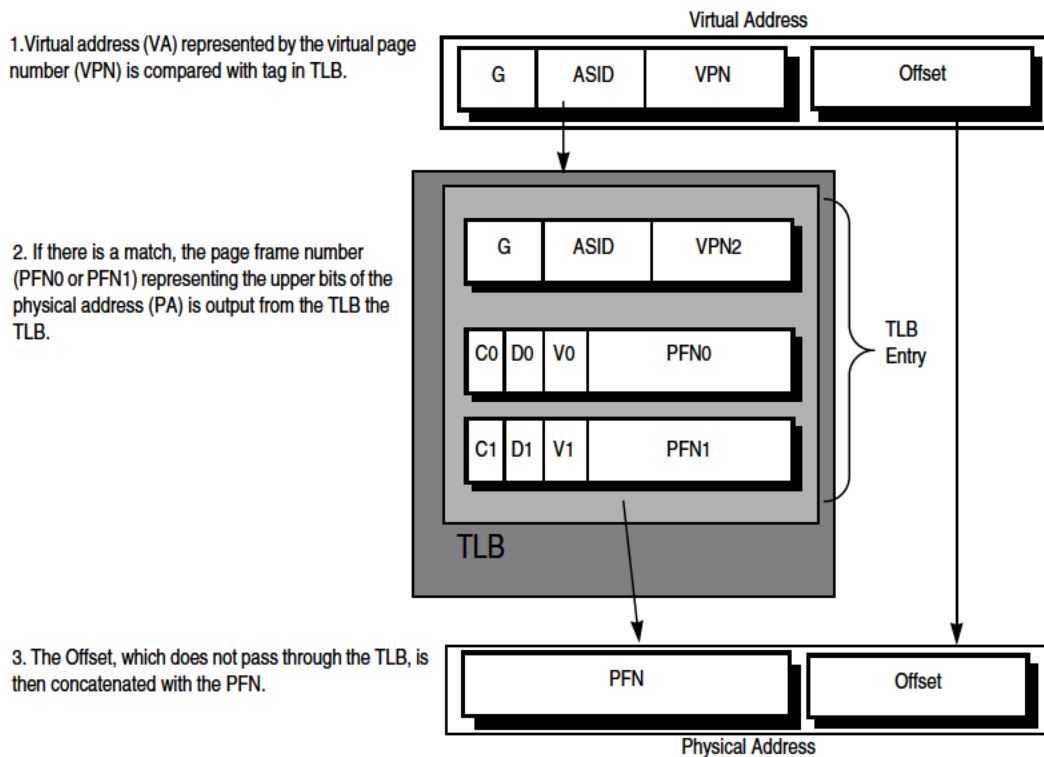
Converting a virtual address to a physical address begins by comparing the virtual address from the processor with the virtual addresses in the TLB. There is a match when the VPN of the address is the same as the VPN field of the entry, and either:

- The Global (G) bit of both the even and odd pages of the TLB entry are set, or
- The ASID field of the virtual address is the same as the ASID field of the TLB entry

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

Figure 5.9 shows the translation of a virtual address into a physical address. In this figure, the virtual address is extended with an 8-bit ASID, which reduces the frequency of TLB flushes during a context switch. This 8-bit ASID contains the number assigned to that process and is stored in the CP0 *EntryHi* register.

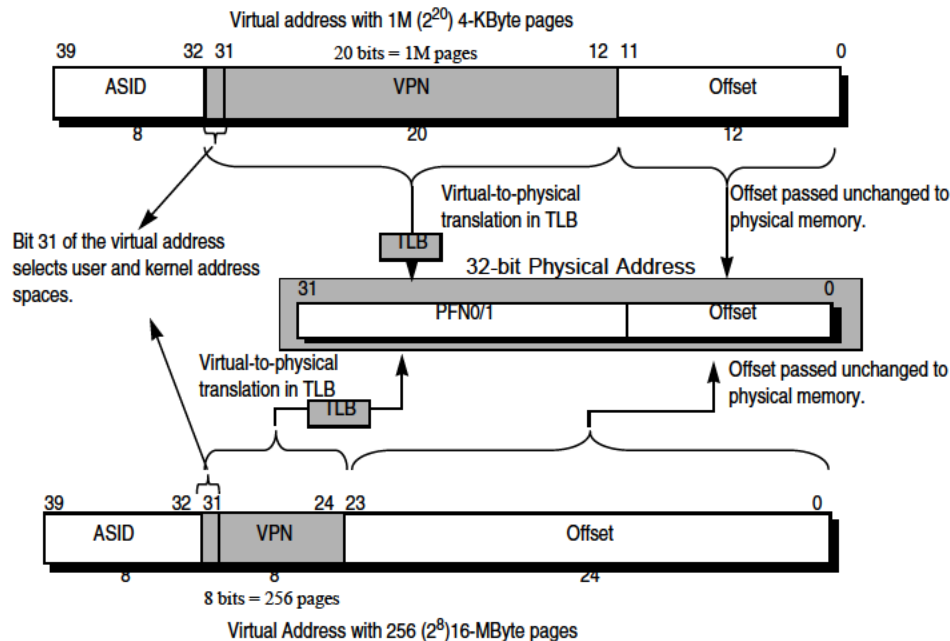
Figure 5.9 Overview of Virtual-to-Physical Address Translation



If there is a virtual address match in the TLB, the Physical Frame Number (PFN) is output from the TLB and concatenated with the *Offset* to form the physical address. The *Offset* represents an address within the page frame space. As shown in Figure 5.9, the *Offset* does not pass through the TLB.

Figure 5.10 shows a flow diagram of the address translation process for two page sizes. The top portion of the figure shows a virtual address for a 4 KByte page size. The width of the *Offset* is defined by the page size. The remaining 20 bits of the address represent the virtual page number (VPN). The bottom portion of Figure 5.10 shows the virtual address for a 16 MByte page size. The remaining 8 bits of the address represent the VPN.

Figure 5.10 32-bit Virtual Address Translation



5.4.1 Hits, Misses, and Multiple Matches

Each JTLB entry contains a tag and two data fields. If a match is found, the upper bits of the virtual address are replaced with the page frame number (PFN) stored in the corresponding entry in the data array of the JTLB. The granularity of JTLB mappings is defined in terms of TLB pages. The JTLB supports pages of different sizes ranging from 4 KB to 256 MB in powers of 4. If a match is found, but the entry is invalid (the V bit in the data field is 0), a TLB Invalid exception is taken.

If no match occurs (TLB miss), an exception is taken and software refills the TLB from the page table resident in memory. Figure 5.11 shows the translation and exception flow of the TLB.

Software can write over a selected TLB entry or use a hardware mechanism to write into a random entry. The *Random* register selects which TLB entry to use on a TLBWR. This register decrements almost every cycle, wrapping to the maximum once its value is equal to the *Wired* register. Thus, TLB entries below the *Wired* value cannot be replaced by a TLBWR allowing important mappings to be preserved. In order to reduce the possibility for a livelock situation, the *Random* register includes a 10-bit LFSR that introduces a pseudo-random perturbation into the decrement.

The core implements a TLB write-compare mechanism to ensure that multiple TLB matches do not occur. On the TLB write operation, the VPN2 field to be written is compared with all other entries in the TLB. If a match occurs, the entry in the TLB is valid, and the entry being written is valid, the core takes a machine-check exception, sets the TS bit in the CP0 *Status* register, and aborts the write operation. For further details on exceptions, please refer to Chapter 6, “Exceptions and Interrupts in the 74K™ Core”. There is a hidden bit in each TLB entry that is cleared on a Reset. This bit is set once the TLB entry is written and is included in the match detection. Therefore, uninitialized TLB entries will not cause a TLB shutdown.

Compared with previous cores from MIPS Technologies, the 74K core uses a more relaxed check for multiple matches in order to avoid machine check exceptions while flushing or initializing the TLB. On a write, all matching entries are disabled to prevent them from matching on future compares. A machine check is only signaled if the entry being written has its valid bit set, the matching entry in the TLB has its valid bit set, and the matching entry is not the entry being written. The cases for the signalling of the machine check exception are enumerated in [Table 5.9](#).

Table 5.9 Machine Check Exception

Existing Match	Matching Entry equals Written Entry	Existing Page Valid Bit	Written Page Valid Bit	Machine Check
No	X	X	X	No
Yes	Yes	X	X	No
Yes	No	0	0	No
Yes	No	0	1	No
Yes	No	1	0	No
Yes	No	1	1	Yes

5.4.2 Memory Space

To assist in controlling both the amount of mapped space and the replacement characteristics of various memory regions, the 74K core provides two mechanisms.

5.4.2.1 Page Sizes

First, the page size can be configured, on a per entry basis, to map different page sizes ranging from 4 KBytes to 256 MBytes, in multiples of 4. The *PageMask* register is loaded with the desired page size, which is then entered into the TLB when a new entry is written. Thus, operating systems can provide special-purpose maps. For example, a typical frame buffer can be memory-mapped with only one TLB entry.

The 74K core implements the following page sizes:

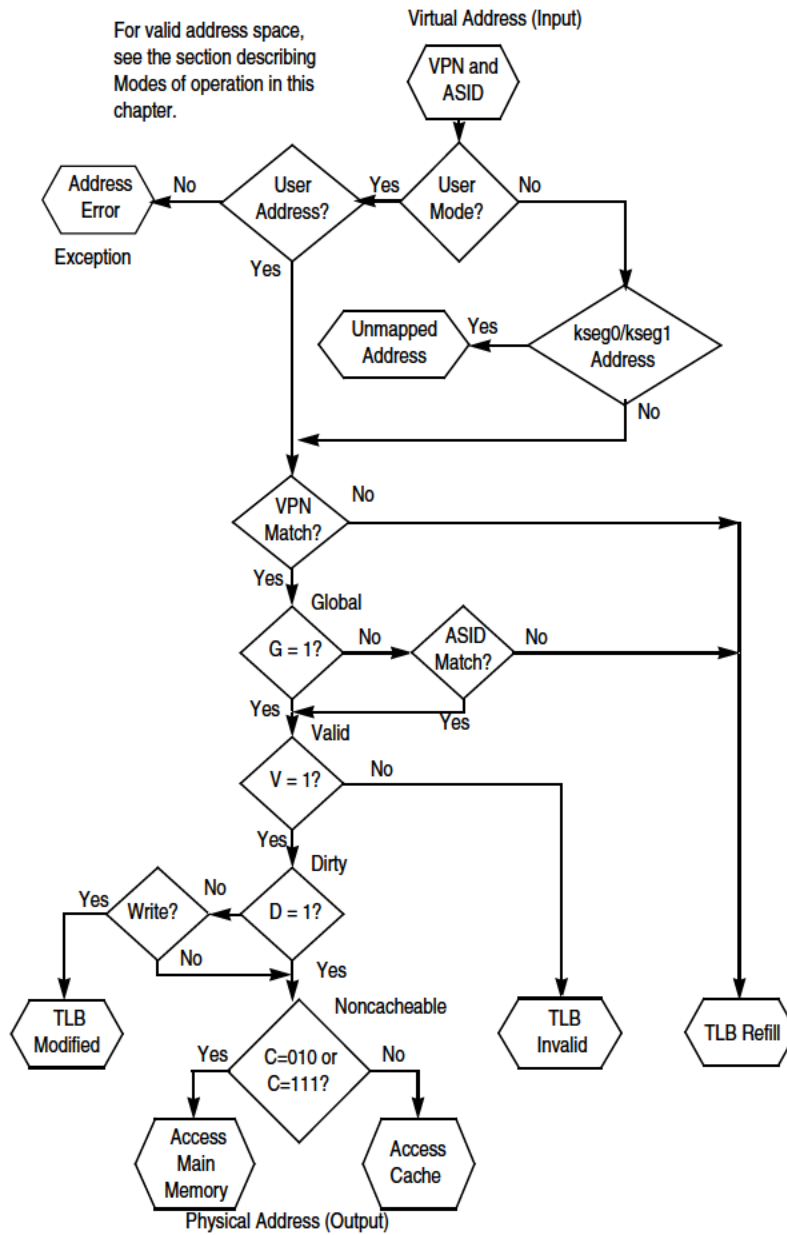
4K, 16K, 64K, 256K, 1M, 4M, 16M, 64M, and 256M

Software can determine which page sizes are supported by writing all ones to the *PageMask* register, then reading the value back. For additional information, see [Section 7.2.7 “PageMask \(CP0 Register 5, Select 0\): Control for Variable Page Size in TLB Entries”](#).

5.4.2.2 Replacement Algorithm

The second mechanism controls the replacement algorithm when a TLB miss occurs. TLB refill is often done using the TLBWR instruction to randomly replace one of the existing entries. The processor provides a mechanism whereby a programmable number of mappings can be locked into the TLB via the CP0 *Wired* register, thus avoiding random replacement. Refer to [Section 7.2.8 “Wired \(CP0 Register 6, Select 0\): Controls Number of Fixed \(“wired”\) TLB Entries”](#) for further details. For entries that are left available for random replacement, the core includes two algorithms for selecting the entry. One involves a counter that decrements almost every cycle (with some pseudo-random perturbation to minimize the likelihood of livelock conditions). A second scheme keeps a table of the TLB entries that have been most recently used and tries to avoid selecting one of them. Selection of the algorithm is done via the CP0 *Config6_{NMRUD}* bit.

Figure 5.11 TLB Address Translation Flow in the 74K™ Processor Core



5.4.3 TLB Instructions

Table 5.10 lists the TLB-related instructions. Refer to Chapter 13, “74K™ Processor Core Instructions” for more information on these instructions.

Table 5.10 TLB Instructions

Op Code	Description of Instruction
TLBP	Translation Lookaside Buffer Probe
TLBR	Translation Lookaside Buffer Read

Table 5.10 TLB Instructions (Continued)

Op Code	Description of Instruction
TLBWI	Translation Lookaside Buffer Write Index
TLBWR	Translation Lookaside Buffer Write Random

5.5 Fixed Mapping MMU

The 74K core optionally implements a simple Fixed Mapping (FM) memory management unit that is smaller than the a full translation lookaside buffer (TLB) and more easily synthesized. Like a TLB, the FM performs virtual-to-physical address translation and provides attributes for the different memory segments. Those memory segments which are unmapped in a TLB implementation (kseg0 and kseg1) are translated identically by the FM MMU.

The FM also determines the cacheability of each segment. These attributes are controlled via bits in the *Config* register. [Table 5.11](#) shows the encoding for the K23 (bits 30:28), KU (bits 27:25) and K0 (bits 2:0) fields of the *Config* register.

Table 5.11 Cache Coherency Attributes

Config Register Fields K23, KU, and K0	Cache Coherency Attribute
0	Cacheable, noncoherent, write-through, no write-allocate
1	Reserved
2	Uncached
3	Cacheable, noncoherent, write-back, write-allocate
4	Reserved
5	Reserved
6	Reserved
7	Uncached Accelerated

With the FM MMU, no translation exceptions can be taken, although address errors are still possible.

Table 5.12 Cacheability of Segments with Fixed Mapping Translation

Segment	Virtual Address Range	Cacheability
useg/kuseg	0x0000_0000- 0x7FFF_FFFF	Controlled by the KU field (bits 27:25) of the <i>Config</i> register. Refer to Table 5.11 for the encoding.
kseg0	0x8000_0000- 0x9FFF_FFFF	Controlled by the K0 field (bits 2:0) of the <i>Config</i> register. See Table 5.11 for the encoding.
kseg1	0xA000_0000- 0xBFFF_FFFF	Always uncacheable
kseg2	0xC000_0000- 0xDFFF_FFFF	Controlled by the K23 field (bits 30:28) of the <i>Config</i> register. Refer to Table 5.11 for the encoding.
kseg3	0xE000_0000- 0xFFFF_FFFF	Controlled by K23 field (bits 30:28) of the <i>Config</i> register. Refer to Table 5.11 for the encoding.

The FM performs a simple translation to map virtual addresses to physical addresses. This mapping is shown in Figure 5.12.

When the ERL bit in the *Status* register is set, useg and kuseg are unmapped and uncached, just as they are when there is a TLB. The mapping when ERL = 1 is shown in Figure 5.13. The ERL bit is usually not asserted by software, but is asserted by hardware after a Reset, NMI, or Cache Error. See Section 6.8 “Exception Descriptions” for further information on exceptions.

Figure 5.12 FM Memory Map (ERL=0) in the 74K™ Processor Core

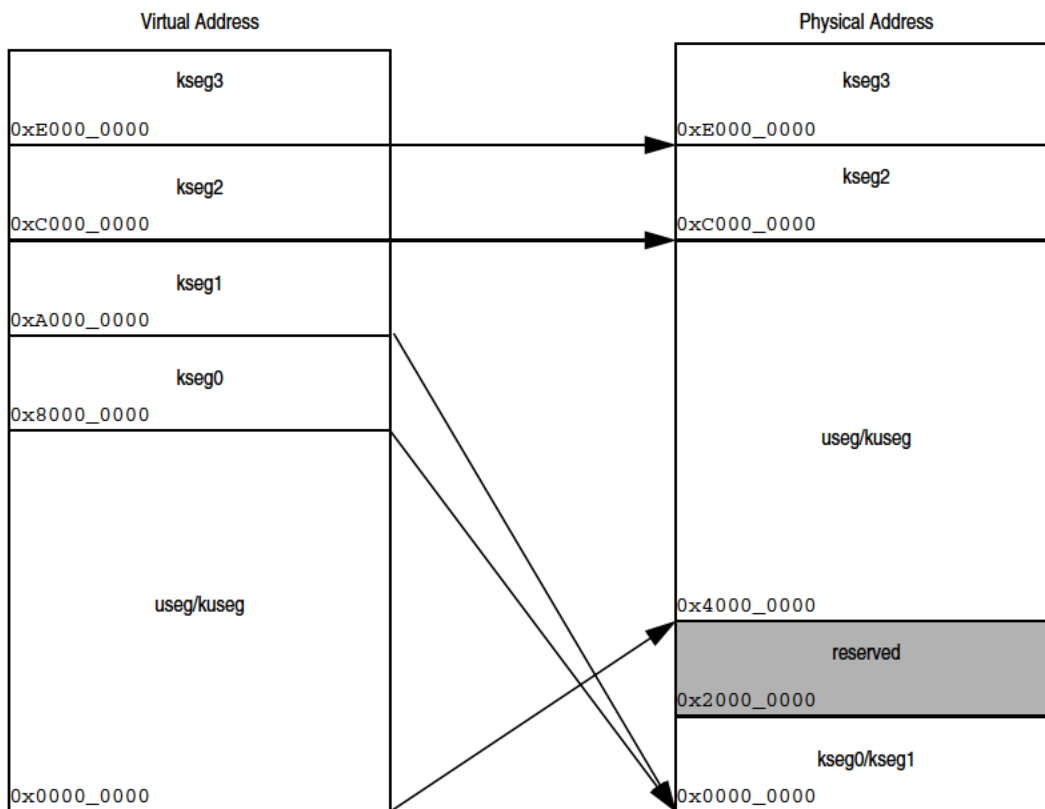
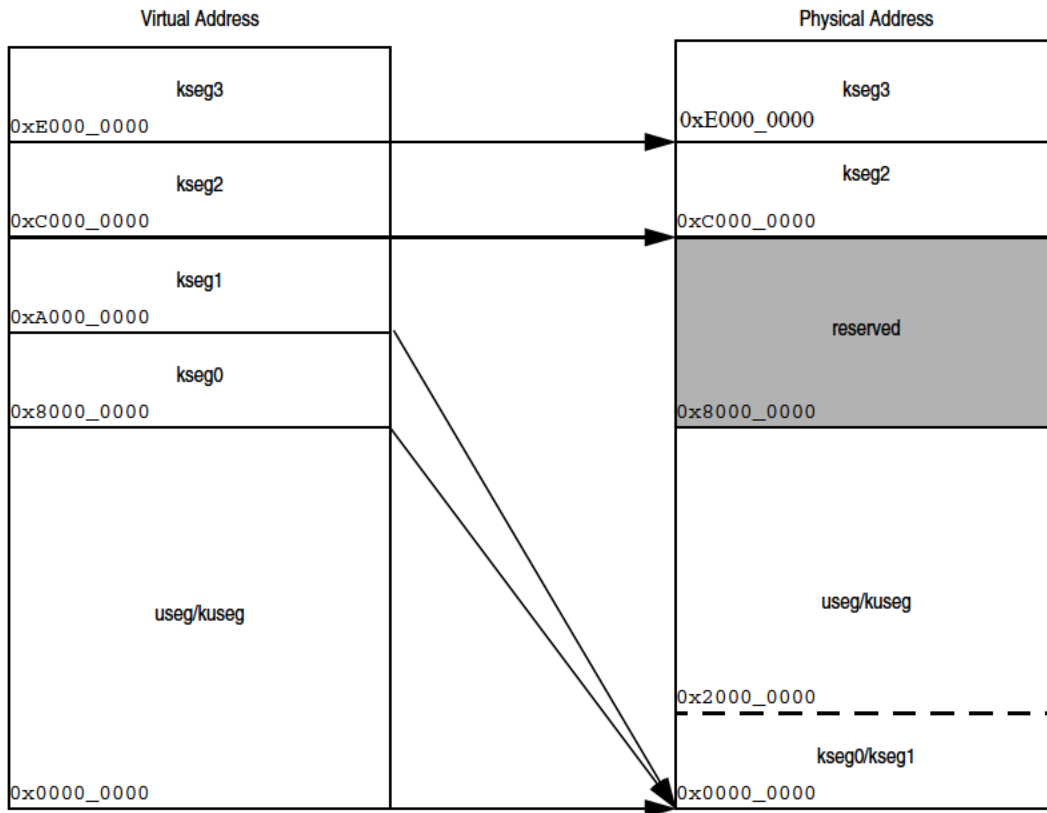


Figure 5.13 FM Memory Map (ERL=1) in the 74K™ Processor Core



Exceptions and Interrupts in the 74K™ Core

The 74K processor core receives exceptions from a number of sources, including arithmetic overflows, misses in the translation lookaside buffer (TLB), I/O interrupts, and system calls. When the CPU detects an exception, the normal sequence of instruction execution is suspended and the processor enters kernel mode, disables interrupts, loads the *Exception Program Counter (EPC)* register with the location where execution can restart after the exception has been serviced, and forces execution of a software exception handler located at a specific address.

The software exception handler saves the context of the processor, including the contents of the program counter, the current operating mode, and the status of the interrupts (enabled or disabled). This context is saved so it can be restored when the exception has been serviced.

Exceptions may be precise or imprecise. Precise exceptions are those for which the *EPC* can be used to identify the instruction that caused the exception. For precise exceptions, the restart location in the *EPC* register is the address of the instruction that caused the exception or, if the instruction was executing in a branch delay slot (as indicated by the *BD* bit in the *Cause* register), the address of the branch instruction immediately preceding the delay slot. Imprecise exceptions, on the other hand, are those for which no return address can be identified. Bus error exceptions and CP2 exceptions are examples of imprecise exceptions.

This chapter contains the following sections:

- [Section 6.1 “Exception Conditions”](#)
- [Section 6.2 “Exception Priority”](#)
- [Section 6.3 “Interrupts”](#)
- [Section 6.4 “GPR Shadow Registers”](#)
- [Section 6.5 “Exception Vector Locations”](#)
- [Section 6.6 “General Exception Processing”](#)
- [Section 6.7 “Debug Exception Processing”](#)
- [Section 6.8 “Exception Descriptions”](#)
- [Section 6.9 “Exception Handling and Servicing Flowcharts”](#)

6.1 Exception Conditions

When an exception condition occurs, the instruction causing the exception and all those that follow it in the pipeline are cancelled. Accordingly, any stall conditions and any later exception conditions that may have referenced this instruction are inhibited.

Exceptions and Interrupts in the 74K™ Core

When the exception condition is detected on an instruction fetch, the core aborts that instruction and all instructions that follow. When this instruction reaches the WB stage, the exception flag causes it to write various CP0 registers with the exception state, change the current program counter (PC) to the appropriate exception vector address, and clear the exception bits of earlier pipeline stages.

For most types of exceptions, this implementation allows all preceding instructions to complete execution and prevents all subsequent instructions from completing. Thus, the value in the *EPC* (or *ErrorEPC* for errors or *DEPC* for debug exceptions) is sufficient to restart execution. It also ensures that exceptions are taken in the order of execution—an instruction taking an exception may itself be killed by an instruction further down the pipeline that takes an exception in a later cycle.

Imprecise exceptions are taken after the instruction that caused them has completed and potentially after following instructions have completed.

6.2 Exception Priority

Table 6.1 contains a list and a brief description of all exception conditions. The exceptions are listed in the order of their relative priority, from highest priority (Reset) to lowest priority (Load/store bus error). When several exceptions occur simultaneously, the exception with the highest priority is taken.

Table 6.1 Priority of Exceptions

Exception	Description
Reset	Assertion of SI_Reset signal.
DSS	EJTAG Debug Single Step.
DINT	EJTAG Debug Interrupt. Caused by the assertion of the external <i>EJ_DINT</i> input, or by setting the <i>EjtagBrk</i> bit in the <i>ECR</i> register.
DDBLImpr/DDBSImpr	Debug Data Break Load/Store. Imprecise.
NMI	Asserting edge of <i>SI_NMI</i> signal.
Machine Check	TLB write that conflicts with an existing entry.
Interrupt	Assertion of unmasked hardware or software interrupt signal.
Deferred Watch	Deferred Watch (unmasked by K DM->!(K DM) transition).
DIB	EJTAG debug hardware instruction break matched.
WATCH	A reference to an address in one of the watch registers (fetch).
AdEL	Fetch address alignment error. Fetch reference to protected address.
TLBL	Fetch TLB miss. Fetch TLB hit to page with V=0.
I-cache Error	Parity error on I-cache access.
IBE	Instruction fetch bus error.
DBp	EJTAG Breakpoint (execution of SDBBP instruction).
Sys	Execution of SYSCALL instruction.
Bp	Execution of BREAK instruction.
CpU	Execution of a coprocessor instruction for a coprocessor that is not enabled.
CEU	Execution of a CorExtend instruction modifying local state when CorExtend is not enabled.

Table 6.1 Priority of Exceptions (Continued)

Exception	Description
DSPDis	DSP ASE State Disabled.
RI	Execution of a Reserved Instruction.
FPE	Floating Point exception.
Ov	Execution of an arithmetic instruction that overflowed.
Tr	Execution of a trap (when trap condition is true).
DDBL / DDBS	EJTAG Data Address Break (address only).
WATCH	A reference to an address in one of the watch registers (data).
AdEL	Load address alignment error. Load reference to protected address.
AdES	Store address alignment error. Store to protected address.
TLBL	Load TLB miss. Load TLB hit to page with V=0
TLBS	Store TLB miss. Store TLB hit to page with V=0.
TLB Mod	Store to TLB page with D=0.
D-cache Error	Cache parity error. Imprecise.
DBE	Load or store bus error. Imprecise.

6.3 Interrupts

In the MIPS32® Release 1 architecture, support for exceptions included two software interrupts, six hardware interrupts, and a special-purpose timer interrupt. The timer interrupt was provided external to the core and was typically combined with hardware interrupt 5 in a system-dependent manner. Interrupts were handled either through the general exception vector (offset 0x180) or the special interrupt vector (0x200), based on the value of *CauseIV*. Software was required to prioritize interrupts as a function of the *CauseIV* bits in the interrupt handler prologue.

Release 2 of the Architecture, implemented by the 74K core, adds a number of upward-compatible extensions to the Release 1 interrupt architecture, including support for vectored interrupts and the implementation of a new interrupt mode that permits the use of an external interrupt controller.

Additionally, internal performance counters have been added to the 74K core. These counters can be configured to count various events within the core. When the MSB of the counter is set, it can trigger a performance counter interrupt. This interrupt, like the timer interrupt, is an output from the core that can be brought back into the core's interrupt pins in a system-dependent manner.

The Fast Debug Channel feature in EJTAG provides a low overhead means for sending data between core software and the EJTAG probe. It includes a pair of FIFOs for transmit and receive data. Software can define FIFO thresholds for generating an interrupt. The fast debug channel interrupt is also routed similarly to the timer and performance counter interrupts. The interrupt status is made available on an output pin and can be brought back into the core's interrupt pins.

6.3.1 Interrupt Modes

The 74K core includes support for three interrupt modes, as defined by Release 2 of the Architecture:

- Interrupt Compatibility mode, in which the behavior of the 74K is identical to the behavior of an implementation of Release 1 of the Architecture.
- Vectored Interrupt (VI) mode, which adds the ability to prioritize and vector interrupts to a handler dedicated to that interrupt, and to assign a GPR shadow set for use during interrupt processing. The presence of this mode is denoted by the *VInt* bit in the *Config3* register. Although this mode is architecturally optional, it is always present on the 74K core, so the *VInt* bit will always read as a 1.
- External Interrupt Controller (EIC) mode, which redefines the way interrupts are handled to provide full support for an external interrupt controller that handles prioritization and vectoring of interrupts. As with VI mode, this mode is architecturally optional. The presence of this mode is denoted by the *VEIC* bit in the *Config3* register. On the 74K core, the *VEIC* bit is set externally by the static input, *SL_EICPresent*, to allow system logic to indicate the presence of an external interrupt controller.

Following reset, the 74K processor defaults to Compatibility mode, which is fully compatible with all implementations of Release 1 of the Architecture.

Table 6.2 shows the current interrupt mode of the processor as a function of the Coprocessor 0 register fields that can affect the mode.

Table 6.2 Interrupt Modes

<i>StatusBEV</i>	<i>CauseIV</i>	<i>IntCtlVS</i>	<i>Config3VINT</i>	<i>Config3VEIC</i>	Interrupt Mode
1	x	x	x	x	Compatibility
x	0	x	x	x	Compatibility
x	x	=0	x	x	Compatibility
0	1	≠0	1	0	Vectored Interrupt
0	1	≠0	x	1	External Interrupt Controller
0	1	≠0	0	0	Cannot occur because <i>IntCtl_{VS}</i> cannot be non-zero if neither Vectored Interrupt nor External Interrupt Controller mode is implemented.
"x" denotes don't care					

6.3.1.1 Interrupt Compatibility Mode

This is the default interrupt mode for the processor and is entered when a Reset exception occurs. In this mode, interrupts are non-vectored and dispatched through exception vector offset 0x180 (if *Cause_{IV}* = 0) or vector offset 0x200 (if *Cause_{IV}* = 1). This mode is in effect when any of the following conditions are true:

- *Cause_{IV}* = 0
- *StatusBEV* = 1

- $IntCtl_{VS} = 0$, which is the case if vectored interrupts are not implemented or have been disabled.

Here is a typical software handler for compatibility mode:

```

/*
 * Assumptions:
 * - CauseIV = 1 (if it were zero, the interrupt exception would have to
 *   be isolated from the general exception vector before arriving
 *   here)
 * - GPRs k0 and k1 are available (no shadow register switches invoked in
 *   compatibility mode)
 * - The software priority is IP7..IP0 (HW5..HW0, SW1..SW0)
 *
 * Location: Offset 0x200 from exception base
 */

IVexception:
    mfc0    k0, C0_Cause      /* Read Cause register for IP bits */
    mfc0    k1, C0_Status    /* and Status register for IM bits */
    andi   k0, k0, M_CauseIM /* Keep only IP bits from Cause */
    and    k0, k0, k1        /* and mask with IM bits */
    beq    k0, zero, Dismiss /* no bits set - spurious interrupt */
    clz    k0, k0            /* Find first bit set, IP7..IP0; k0 = 16..23 */
    xori   k0, k0, 0x17      /* 16..23 => 7..0 */
    sll   k0, k0, VS        /* Shift to emulate software IntCtlVS */
    la     k1, VectorBase    /* Get base of 8 interrupt vectors */
    addu   k0, k0, k1        /* Compute target from base and offset */
    jr     k0                /* Jump to specific exception routine */
    nop

/*
 * Each interrupt processing routine processes a specific interrupt, analogous
 * to those reached in VI or EIC interrupt mode. Since each processing routine
 * is dedicated to a particular interrupt line, it has the context to know
 * which line was asserted. Each processing routine may need to look further
 * to determine the actual source of the interrupt if multiple interrupt requests
 * are ORed together on a single IP line. Once that task is performed, the
 * interrupt may be processed in one of two ways:
 *
 * - Completely at interrupt level (e.g., a simple UART interrupt). The
 *   SimpleInterrupt routine below is an example of this type.
 * - By saving sufficient state and re-enabling other interrupts. In this
 *   case the software model determines which interrupts are disabled during
 *   the processing of this interrupt. Typically, this is either the single
 *   StatusIM bit that corresponds to the interrupt being processed, or some
 *   collection of other StatusIM bits so that "lower" priority interrupts are
 *   also disabled. The NestedInterrupt routine below is an example of this type.
 */

SimpleInterrupt:
/*
 * Process the device interrupt here and clear the interrupt request
 * at the device. In order to do this, some registers may need to be
 * saved and restored. The coprocessor 0 state is such that an ERET
 * will simply return to the interrupted code.
 */
    eret                    /* Return to interrupted code */

```

Exceptions and Interrupts in the 74K™ Core

```
NestedException:
/*
 * Nested exceptions typically require saving the EPC and Status registers,
 * saving any GPRs that may be modified by the nested exception routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below cannot cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

/* Save GPRs here, and setup software context */
mfc0 k0, C0_EPC /* Get restart address */
s k0, EPCSave /* Save in memory */
mfc0 k0, C0_Status /* Get Status value */
sw k0, StatusSave /* Save in memory */
li k1, ~IMbitsToClear /* Get IM bits to clear for this interrupt */
/* this must include at least the IM bit */
/* for the current interrupt, and may include */
/* others */
and k0, k0, k1 /* Clear bits in copy of Status */
ins k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
/* Clear KSU, ERL, EXL bits in k0 */
mtc0 k0, C0_Status /* Modify mask, switch to kernel mode, */
/* re-enable interrupts */

/*
 * Process interrupt here, including clearing device interrupt.
 * In some environments this may be done with a thread running in
 * kernel or user mode. Such an environment is well beyond the scope of
 * this example.
 */

/*
 * To complete interrupt processing, the saved values must be restored
 * and the original interrupted code restarted.
 */

di /* Disable interrupts - may not be required */
lw k0, StatusSave /* Get saved Status (including EXL set) */
l k1, EPCSave /* and EPC */
mtc0 k0, C0_Status /* Restore the original value */
mtc0 k1, C0_EPC /* and EPC */
/* Restore GPRs and software state */
eret /* Dismiss the interrupt */
```

6.3.1.2 Vectored Interrupt Mode

In Vectored Interrupt (VI) mode, a priority encoder prioritizes pending interrupts and generates a vector which can be used to direct each interrupt to a dedicated handler routine. This mode also allows each interrupt to be mapped to a GPR shadow register set for use by the interrupt handler. VI mode is in effect when all the following conditions are true:

- $Config3_{VInt} = 1$
- $Config3_{VEIC} = 0$
- $IntCtl_{VS} \neq 0$

- $Cause_{IV} = 1$
- $Status_{BEV} = 0$

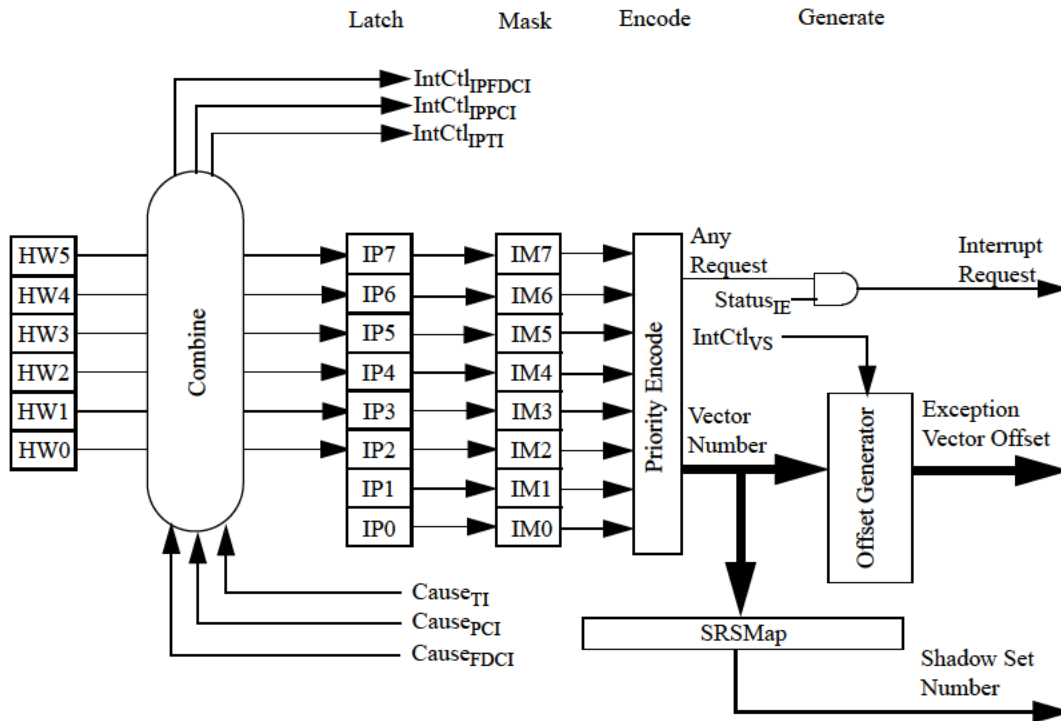
In VI interrupt mode, the six hardware interrupts are interpreted as individual hardware interrupt requests. The timer, performance counter, and fast debug channel interrupts are combined in a system-dependent way (external to the core) with the hardware interrupts (the interrupt with which they are combined is indicated by the $IntCtl_{IPTI/IPCII/IPFDCI}$ fields) to provide the appropriate relative priority of the those interrupts with that of the hardware interrupts. The processor interrupt logic ANDs each of the $Cause_{IP}$ bits with the corresponding $Status_{IM}$ bits. If any of these values is 1, and if interrupts are enabled ($Status_{IE} = 1$, $Status_{EXL} = 0$, and $Status_{ERL} = 0$), an interrupt is signaled and a priority encoder scans the values in the order shown in [Table 6.3](#).

Table 6.3 Relative Interrupt Priority for Vectored Interrupt Mode

Relative Priority	Interrupt Type	Interrupt Source	Interrupt Request Calculated From	Vector Number Generated by Priority Encoder
Highest Priority	Hardware	HW5	IP7 and IM7	7
		HW4	IP6 and IM6	6
		HW3	IP5 and IM5	5
		HW2	IP4 and IM4	4
		HW1	IP3 and IM3	3
		HW0	IP2 and IM2	2
Lowest Priority	Software	SW1	IP1 and IM1	1
		SW0	IP0 and IM0	0

The priority order places a relative priority on each hardware interrupt and places the software interrupts at a priority lower than all hardware interrupts. When the priority encoder finds the highest priority pending interrupt, it outputs an encoded vector number that is used in the calculation of the handler for that interrupt, as described below. This is shown pictorially in [Figure 6.1](#).

Figure 6.1 Interrupt Generation for Vectored Interrupt Mode



A typical software handler for Vectored Interrupt mode bypasses the entire sequence of code following the IVexception label shown for the compatibility mode handler above. Instead, the hardware performs the prioritization, dispatching directly to the interrupt processing routine. Unlike the compatibility mode examples, a vectored interrupt handler may take advantage of a dedicated GPR shadow set to avoid saving any registers. As such, the SimpleInterrupt code shown above need not save the GPRs.

A nested interrupt is similar to that shown for compatibility mode, but may also take advantage of running the nested exception routine in the GPR shadow set dedicated to the interrupt or in another shadow set. Such a routine might look as follows:

```

NestedException:
/*
 * Nested exceptions typically require saving the EPC, Status and SRSCtl registers,
 * setting up the appropriate GPR shadow set for the routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below cannot cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

/* Use the current GPR shadow set, and setup software context */
mfc0 k0, C0_EPC          /* Get restart address */
s k0, EPCSave           /* Save in memory */
mfc0 k0, C0_Status      /* Get Status value */
sw k0, StatusSave       /* Save in memory */
mfc0 k0, C0_SRSCtl      /* Save SRSCtl if changing shadow sets */
sw k0, SRSCtlSave
li k1, ~IMbitsToClear  /* Get IM bits to clear for this interrupt */
/* this must include at least the IM bit */
    
```

```

/* for the current interrupt, and may include */
/* others */
and k0, k0, k1 /* Clear bits in copy of Status */
/* If switching shadow sets, write new value to SRSCtl_PSS here */
ins k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
/* Clear KSU, ERL, EXL bits in k0 */
mtc0 k0, C0_Status /* Modify mask, switch to kernel mode, */
/* re-enable interrupts */

/*
 * If switching shadow sets, clear only KSU above, write target
 * address to EPC, and do execute an eret to clear EXL, switch
 * shadow sets, and jump to routine
 */

/* Process interrupt here, including clearing device interrupt */

/*
 * To complete interrupt processing, the saved values must be restored
 * and the original interrupted code restarted.
 */

di /* Disable interrupts - may not be required */
lw k0, StatusSave /* Get saved Status (including EXL set) */
l k1, EPCSave /* and EPC */
mtc0 k0, C0_Status /* Restore the original value */
lw k0, SRSCtlSave /* Get saved SRSCtl */
mtc0 k1, C0_EPC /* and EPC */
mtc0 k0, C0_SRSCtl /* Restore shadow sets */
ehb /* Clear hazard */
eret /* Dismiss the interrupt */

```

6.3.1.3 External Interrupt Controller Mode

External Interrupt Controller (EIC) mode redefines the way that the processor interrupt logic is configured to provide support for an external interrupt controller. The interrupt controller is responsible for prioritizing all interrupts, including hardware, software, timer, fast debug channel, and performance counter interrupts, and directly supplying to the processor the vector number of the highest priority interrupt.

EIC interrupt mode is in effect if all of the following conditions are true:

- $Config3_{VEIC} = 1$
- $IntCtl_{VS} \neq 0$
- $Cause_{IV} = 1$
- $Status_{BEV} = 0$

In EIC mode, the processor sends the state of the software interrupt requests ($Cause_{IP1..IP0}$) and the timer, performance counter, and fast debug channel interrupt requests ($Cause_{TI/PC/FDC}$) to the external interrupt controller, which prioritizes these interrupts in a system-dependent way with other hardware interrupts. The interrupt controller can be a hardwired logic block, or it can be configurable by control and status registers. This allows the interrupt controller to be more specific or more general as a function of the system environment and needs.

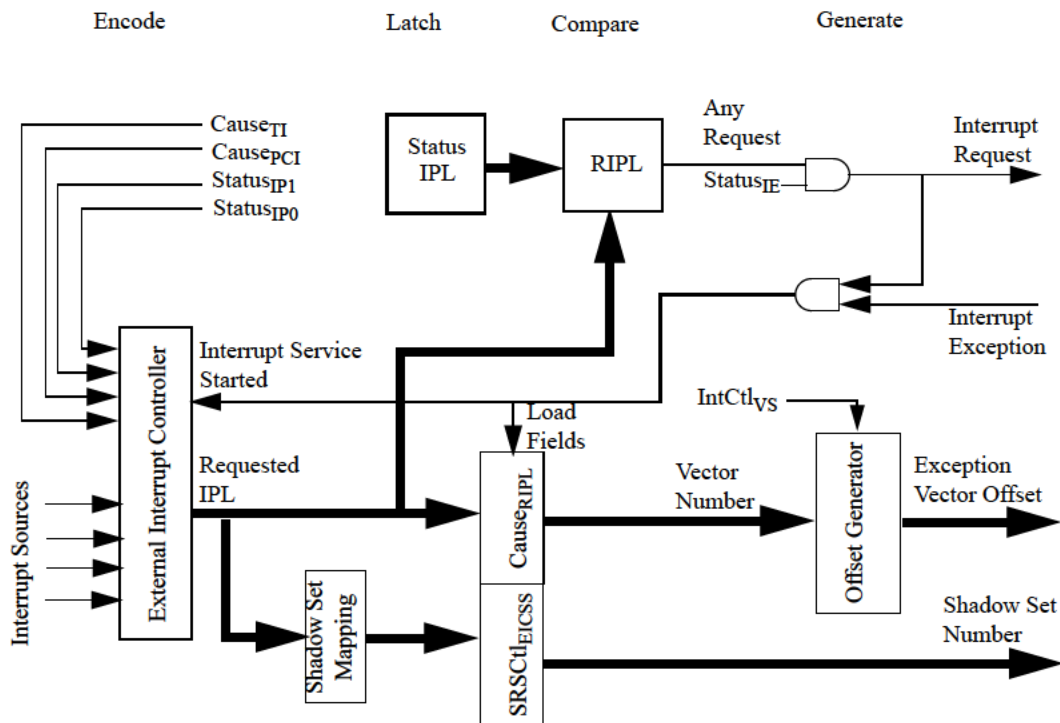
The external interrupt controller prioritizes its interrupt requests and produces the vector number of the highest priority interrupt to be serviced. The vector number, called the Requested Interrupt Priority Level (RIPL), is a 6-bit encoded value in the range 0..63, inclusive. The values 1..63 represent the lowest (1) to highest (63) RIPL for the interrupt to be serviced. A value of 0 indicates that no interrupt requests are pending. The interrupt controller inputs this value on the 6 hardware interrupt lines, which are treated as an encoded value in EIC mode.

$Status_{IPL}$ (which overlays $Status_{IM7..IM2}$) is interpreted as the Interrupt Priority Level (IPL) at which the processor is currently operating (a value of zero indicates that no interrupt is currently being serviced). When the interrupt controller requests service for an interrupt, the processor compares RIPL with $Status_{IPL}$ to determine if the requested interrupt has a higher priority than the current IPL. If RIPL is strictly greater than $Status_{IPL}$, and interrupts are enabled ($Status_{IE} = 1$, $Status_{EXL} = 0$, and $Status_{ERL} = 0$), an interrupt request is signaled to the pipeline. When the processor starts the interrupt exception, it loads RIPL into $Cause_{RIPL}$ (which overlays $Cause_{IP7..IP2}$) and signals the external interrupt controller to notify it that the request is being serviced. The interrupt exception uses the value of $Cause_{RIPL}$ as the vector number. Because $Cause_{RIPL}$ is only loaded by the processor when an interrupt exception is signaled, it is available to software during interrupt processing.

In EIC mode, the external interrupt controller is also responsible for supplying the GPR shadow register set number to use when servicing the interrupt. As such, the $SRSMap$ register is not used in this mode, and the mapping of the vectored interrupt to a GPR shadow set is done by programming (or designing) the interrupt controller to provide the correct GPR shadow set number when an interrupt is requested. When the processor loads an interrupt request into $Cause_{RIPL}$, it also loads the GPR shadow set number into $SRSctl_{EICSS}$, which is copied to $SRSctl_{CSS}$ when the interrupt is serviced.

The operation of EIC interrupt mode is shown pictorially in Figure 6.2.

Figure 6.2 Interrupt Generation for External Interrupt Controller Interrupt Mode



A typical software handler for EIC mode bypasses the entire sequence of code following the `IVexception` label shown for the compatibility-mode handler above. Instead, the hardware performs the prioritization, dispatching directly to the interrupt processing routine. Unlike the compatibility-mode examples, an EIC interrupt handler may take advantage of a dedicated GPR shadow set to avoid saving any registers. As such, the `SimpleInterrupt` code shown above need not save the GPRs.

A nested interrupt is similar to that shown for compatibility mode, but may also take advantage of running the nested exception routine in the GPR shadow set dedicated to the interrupt or in another shadow set. It also need only copy `CauseRIPL` to `StatusIPL` to prevent lower priority interrupts from interrupting the handler. Here is an example of such a routine:

```
NestedException:
/*
 * Nested exceptions typically require saving the EPC, Status, and SRSCtl registers,
 * setting up the appropriate GPR shadow set for the routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below can not cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

/* Use the current GPR shadow set, and setup software context */
mfc0 k1, C0_Cause /* Read Cause to get RIPL value */
mfc0 k0, C0_EPC /* Get restart address */
srl k1, k1, S_CauseRIPL /* Right justify RIPL field */
sw k0, EPCSave /* Save in memory */
mfc0 k0, C0_Status /* Get Status value */
sw k0, StatusSave /* Save in memory */
ins k0, k1, S_StatusIPL, 6 /* Set IPL to RIPL in copy of Status */
mfc0 k1, C0_SRSCtl /* Save SRSCtl if changing shadow sets */
sw k1, SRSCtlSave
/* If switching shadow sets, write new value to SRSCtlPSS here */
ins k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
/* Clear KSU, ERL, EXL bits in k0 */
mtc0 k0, C0_Status /* Modify IPL, switch to kernel mode, */
/* re-enable interrupts */

/*
 * If switching shadow sets, clear only KSU above, write target
 * address to EPC, and do execute an eret to clear EXL, switch
 * shadow sets, and jump to routine
 */

/* Process interrupt here, including clearing device interrupt */

/*
 * The interrupt completion code is identical to that shown for VI mode above.
 */
```

6.3.2 Generation of Exception Vector Offsets for Vectored Interrupts

For vectored interrupts (in either VI or EIC interrupt mode), a vector number is produced by the interrupt control logic. This number is combined with `IntCtlVS` to create the interrupt offset, which is added to `0x200` to create the exception vector offset. For VI mode, the vector number is in the range 0..7, inclusive. For EIC interrupt mode, the vector number is in the range 1..63, inclusive (0 being the encoding for “no interrupt”). The `IntCtlVS` field specifies the spacing between vector locations. If this value is zero (the default reset state), the vector spacing is zero and the pro-

cessor reverts to Interrupt Compatibility mode. A non-zero value enables vectored interrupts. Table 6.4 shows the exception vector offset for a representative subset of the vector numbers and values of the *IntCtl_{VS}* field.

Table 6.4 Exception Vector Offsets for Vectored Interrupts

Vector Number	Value of IntCtl _{VS} Field				
	0b00001	0b00010	0b00100	0b01000	0b10000
0	0x0200	0x0200	0x0200	0x0200	0x0200
1	0x0220	0x0240	0x0280	0x0300	0x0400
2	0x0240	0x0280	0x0300	0x0400	0x0600
3	0x0260	0x02C0	0x0380	0x0500	0x0800
4	0x0280	0x0300	0x0400	0x0600	0x0A00
5	0x02A0	0x0340	0x0480	0x0700	0x0C00
6	0x02C0	0x0380	0x0500	0x0800	0x0E00
7	0x02E0	0x03C0	0x0580	0x0900	0x1000
		• • •			
61	0x09A0	0x1140	0x2080	0x3F00	0x7C00
62	0x09C0	0x1180	0x2100	0x4000	0x7E00
63	0x09E0	0x11C0	0x2180	0x4100	0x8000

The general equation for the exception vector offset for a vectored interrupt is:

$$\text{vectorOffset} \leftarrow 0x200 + (\text{vectorNumber} \times (\text{IntCtl}_{VS} \parallel 0b00000))$$

6.4 GPR Shadow Registers

Release 2 of the Architecture optionally removes the need to save and restore GPRs on entry to high-priority interrupts and exceptions, and to provide specified processor modes with the same capability. This is done by introducing multiple copies of the GPRs, called *shadow sets*, and by allowing privileged software to associate a shadow set with entry to kernel mode via an interrupt vector or exception. The normal GPRs are designated as shadow set zero.

The number of GPR shadow sets is a build-time option on the 74K core. Although Release 2 of the Architecture defines a maximum of 16 shadow sets, the core allows one (the normal GPRs), two, or four shadow sets. The highest number actually implemented is indicated by the *SRSCtl_{HSS}* field. If this field is zero, only the normal GPRs are implemented.

Shadow sets are new copies of the GPRs that can be substituted for the normal GPRs on entry to kernel mode via an interrupt or exception. Once a shadow set is bound to a kernel mode entry condition, reference to GPRs operate exactly as one would expect, but they are redirected to registers that are dedicated to that condition. Privileged software may need to reference all GPRs in the register file, even specific shadow registers that are not visible in the current mode. The *RDPGPR* and *WRPGPR* instructions are used for this purpose. The *CSS* field of the *SRSCtl* register provides the number of the current shadow register set, and the *PSS* field of the *SRSCtl* register provides the number of the previous shadow register set (the set that was current before the last exception or interrupt occurred).

If the processor is operating in VI mode, binding of a vectored interrupt to a shadow set is done by writing to the *SRSMap* register. If the processor is operating in EIC interrupt mode, the binding of the interrupt to a specific shadow

set is provided by the external interrupt controller, and is configured in an implementation-dependent way. Binding of an exception or non-vectorized interrupt to a shadow set is done by writing to the *ESS* field of the *SRSCtl* register. When an exception or interrupt occurs, the value of *SRSCtl_{CSS}* is copied to *SRSCtl_{PSS}*, and *SRSCtl_{CSS}* is set to the value taken from the appropriate source. On an ERET, the value of *SRSCtl_{PSS}* is copied back into *SRSCtl_{CSS}* to restore the shadow set of the mode to which control returns. More precisely, the rules for updating the fields in the *SRSCtl* register on an interrupt or exception are as follows:

1. No field in the *SRSCtl* register is updated if any of the following conditions are true (in this case, steps 2 and 3 are skipped):
 - The exception is one that sets *Status_{ERL}*: Reset or NMI.
 - The exception causes entry into EJTAG Debug Mode
 - *Status_{BEV}* = 1
 - *Status_{EXL}* = 1
2. *SRSCtl_{CSS}* is copied to *SRSCtl_{PSS}*
3. *SRSCtl_{CSS}* is updated from one of the following sources:
 - The appropriate field of the *SRSMap* register, based on IPL, if the exception is an interrupt, *Cause_{IV}* = 1, *Config3_{VEIC}* = 0, and *Config3_{VInt}* = 1. These are the conditions for a vectored interrupt.
 - The *EICSS* field of the *SRSCtl* register if the exception is an interrupt, *Cause_{IV}* = 1, and *Config3_{VEIC}* = 1. These are the conditions for a vectored EIC interrupt.
 - The *ESS* field of the *SRSCtl* register in any other case. This is the condition for a non-interrupt exception, or a non-vectorized interrupt.

Similarly, the rules for updating the fields in the *SRSCtl* register at the end of an exception or interrupt are as follows:

1. No field in the *SRSCtl* register is updated if any of the following conditions are true (in this case, step 2 is skipped):
 - A DERET is executed
 - An ERET is executed with *Status_{ERL}* = 1
2. *SRSCtl_{PSS}* is copied to *SRSCtl_{CSS}*.

These rules have the effect of preserving the *SRSCtl* register in any case of a nested exception or one which occurs before the processor has been fully initialized (*Status_{BEV}* = 1).

Privileged software may switch the current shadow set by writing a new value into *SRSCtl_{PSS}*, loading *EPC* with a target address, and doing an ERET.

6.5 Exception Vector Locations

The Reset, Soft Reset, NMI, and EJTAG Debug exceptions are vectored to a specific location, as shown in Table 6.5 and Table 6.6. Addresses for all other exceptions are a combination of a vector offset and a vector base address. In Release 1 of the architecture, the vector base address was fixed. In Release 2, software is allowed to specify the vector base address via the *EBase* register for exceptions that occur when *Status_{BEV}* equals 0. Another degree of flexibility in the selection of the vector base address, for use when *Status_{BEV}* equals 1, is provided via a set of input pins, *SI_UseExceptionBase* and *SI_ExceptionBase*[29:12]. Table 6.5 shows the vector base address when *SI_UseExceptionBase* equals 0, as a function of the exception and whether the *BEV* bit is set in the *Status* register. Table 6.6 shows the vector base addresses when *SI_UseExceptionBase* = 1. As can be seen in Table 6.6, when *SI_UseExceptionBase* equals 1, the exception vectors for cases where *Status_{BEV}* = 0 are not affected.

Table 6.7 shows the offsets from the vector base address as a function of the exception. Note that the *IV* bit in the *Cause* register causes interrupts to use a dedicated exception vector offset, rather than the general exception vector. Table 6.4 (on page 118) shows the offset from the base address in the case where *Status_{BEV}* = 0 and *Cause_{IV}* = 1. Table 6.8 combines these two tables into one that contains all possible vector addresses as a function of the state that can affect the vector selection. To avoid complexity in the table, it is assumed that *IntCtl_{VS}* = 0.

Table 6.5 Exception Vector Base Addresses, *SI_UseExceptionBase* = 0

Exception	<i>Status_{BEV}</i>	
	0	1
Reset, NMI	0xBFC0.0000	
EJTAG Debug (with <i>ProbEn</i> = 0, in the EJTAG_Control_register and <i>DCR.RDVec</i> =0)	0xBFC0.0480	
EJTAG Debug (with <i>ProbEn</i> = 0, in the EJTAG_Control_register and <i>DCR.RDVec</i> =1)	0x DebugVectorAddr[31:7] 2b0000000	
EJTAG Debug (with <i>ProbEn</i> = 1 in the EJTAG_Control_register)	0xFF20.0200	
Cache Error	<i>EBase</i> _{31 30} 1 <i>EBase</i> _{28 12} 0x000 Note that <i>EBase</i> _{31 30} have the fixed value 0b10	0xBFC0.0300
Other	<i>EBase</i> _{31 12} 0x000 Note that <i>EBase</i> _{31 30} have the fixed value 0b10	0xBFC0.0200
‘ ’ denotes bit string concatenation		

Table 6.6 Exception Vector Base Addresses, *SI_UseExceptionBase* = 1

Exception	<i>Status_{BEV}</i>	
	0	1
Reset, NMI	0b10 <i>SI_ExceptionBase</i> [29:12] 0x000	
EJTAG Debug (with <i>ProbEn</i> = 0 in the EJTAG_Control_register and <i>DCR.RDVec</i> =0)	DebugVectorAddr[31:7] 2b0000000	

Table 6.6 Exception Vector Base Addresses, $SI_UseExceptionBase = 1$ (Continued)

Exception	Status _{BEV}	
	0	1
EJTAG Debug (with $ProbEn = 0$ in the EJTAG_Control_register and $DCR.RDVec=1$)	0b10 $SI_ExceptionBase[29:12]$ 0x480	
EJTAG Debug (with $ProbEn = 1$ in the EJTAG_Control_register)	0xFF20.0200	
Cache Error	$EBase_{31:30} 1 EBase_{28:12} 0x000$ Note that $EBase_{31:30}$ have the fixed value 0b10	0b101 $SI_ExceptionBase[28:12]$ 0x300
Other	$EBase_{31:12} 0x000$ Note that $EBase_{31:30}$ have the fixed value 0b10	0b10 $SI_ExceptionBase[29:12]$ 0x200
‘ ’ denotes bit string concatenation		

Table 6.7 Exception Vector Offsets

Exception	Vector Offset
TLB Refill, $EXL = 0$	0x000
General Exception	0x180
Interrupt, $Cause_{IV} = 1$	0x200 (In Release 2 implementations, this is the base of the vectored interrupt table when $Status_{BEV} = 0$)
Reset, NMI	None (uses reset base address)

Table 6.8 Exception Vectors

Exception	$SI_UseExceptionBase$	Status _{BEV}	Status _{EXL}	Cause _{IV}	EJTAG ProbEn	Vector (IntCtl _{VS} = 0)
Reset, NMI	0	x	x	x	x	0xBFC0.0000
Reset, NMI	1	x	x	x	x	0b10 $SI_ExceptionBase[29:12]$ 0x000
EJTAG Debug	0	x	x	x	0	0xBFC0.0480 (if $DCR.RDVec=0$) DebugVectorAddr[31:7] 2b0000000 (if $DCR.RDVec=1$)
EJTAG Debug	1	x	x	x	0	0b10 $SI_ExceptionBase[29:12]$ 0x480 (if $DCR.RDVec=0$) DebugVectorAddr[31:7] 2b0000000 (if $DCR.RDVec=1$)

Table 6.8 Exception Vectors (Continued)

Exception	SI_UseExceptionBase	StatusBEV	StatusEXL	CauseV	EJTAG Proben	Vector (IntCtlvs = 0)
EJTAG Debug	x	x	x	x	1	0xFF20.0200
TLB Refill	0	0	1	x	x	0xEBase[31:12] 0x180
TLB Refill	0	1	0	x	x	0xBFC0.0200
TLB Refill	1	1	0	x	x	0b10 SI_ExceptionBase[29:12] 0x200
TLB Refill	0	1	1	x	x	0xBFC0.0380
TLB Refill	1	1	1	x	x	0b10 SI_ExceptionBase[29:12] 0x380
Cache Error	0	0	x	x	x	0xEBase[31:30] 0b1 EBase[28:12] 0x100
Cache Error	0	1	x	x	x	0xBFC0.0300
Cache Error	1	1	x	x	x	0b101 SI_ExceptionBase[28:12] 0x300
Interrupt	x	0	0	0	x	0xEBase[31:12] 0x180
Interrupt	x	0	0	1	x	0xEBase[31:12] 0x200
Interrupt	0	1	0	0	x	0xBFC0.0380
Interrupt	1	1	0	0	x	0b10 SI_ExceptionBase[29:12] 0x380
Interrupt	0	1	0	1	x	0xBFC0.0400
Interrupt	1	1	0	1	x	0b10 SI_ExceptionBase[29:12] 0x400
All others	0	0	x	x	x	0xEBase[31:12] 0x180
All others	0	1	x	x	x	0xBFC0.0380
All others	1	1	x	x	x	0b10 SI_ExceptionBase[29:12] 0x380

'x' denotes don't care,
'||' denotes bit string concatenation

6.6 General Exception Processing

With the exception of Reset, NMI, cache error, and EJTAG Debug 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 (see Table 7.25). The value loaded into the *EPC* register is dependent on whether the processor implements the MIPS16 ASE, and whether the instruction is in the delay slot of a branch or jump which has delay slots. Table 6.9 shows the value stored in each of the CP0 PC registers, including *EPC*. For implementations of Release 2 of the Architecture if *Status_{BEV}* = 0, the *CSS* field in the *SRSCtl* register is copied to the *PSS* field, and the *CSS* value is loaded from the appropriate source.

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. For implementations of Release 2 of the Architecture, the *SRSCtl* register is not changed.

Table 6.9 Value Stored in EPC, ErrorEPC, or DEPC on Exception

MIPS16 Implemented?	In Branch/Jump Delay Slot?	Value stored in EPC/ErrorEPC/DEPC
No	No	Address of the instruction
No	Yes	Address of the branch or jump instruction (PC-4)
Yes	No	Upper 31 bits of the address of the instruction, combined with the ISA Mode bit
Yes	Yes	Upper 31 bits of the branch or jump instruction (PC-2 in the MIPS16 ISA Mode and PC-4 in the 32-bit ISA Mode), combined with the ISA Mode bit

- 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 begins executing 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 it 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 is 1, all exceptions go through the general exception vector */
/* and neither EPC nor Cause_BD nor SRSCtl are modified */
if Status_EXL = 1 then
    vectorOffset ← 0x180
else
    /* For implementations that include the MIPS16e ASE, calculate potential */
    /* PC adjustment for exceptions in the delay slot */
    if Config1_CA = 0 then
        restartPC ← PC
        branchAdjust ← 4      /* Possible adjustment for delay slot */
    else
        restartPC ← PC..1 || ISAMode
        if (ISAMode = 0) or ExtendedMIPS16Instruction
            branchAdjust ← 4  /* Possible adjustment for 32-bit MIPS delay slot */
        else
            branchAdjust ← 2  /* Possible adjustment for MIPS16 delay slot */
        endif
    endif
endif
if InstructionInBranchDelaySlot then
    EPC ← restartPC - branchAdjust /* PC of branch/jump */
    Cause_BD ← 1
else
    EPC ← restartPC                /* PC of instruction */
    Cause_BD ← 0

```

Exceptions and Interrupts in the 74K™ Core

```

endif

/* Compute vector offsets as a function of the type of exception */
NewShadowSet ← SRSCtlESS /* Assume exception, Release 2 only */
if ExceptionType = TLBRefill then
    vectorOffset ← 0x000
elseif (ExceptionType = Interrupt) then
    if (CauseIV = 0) then
        vectorOffset ← 0x180
    else
        if (StatusBEV = 1) or (IntCtlVS = 0) then
            vectorOffset ← 0x200
        else
            if Config3VEIC = 1 then
                VecNum ← CauseR IPL
                NewShadowSet ← SRSCtlEICSS
            else
                VecNum ← VIntPriorityEncoder()
                NewShadowSet ← SRSMaPIPLX4+3..IPLX4
            endif
            vectorOffset ← 0x200 + (VecNum × (IntCtlVS || 0b00000))
        endif /* if (StatusBEV = 1) or (IntCtlVS = 0) then */
    endif /* if (CauseIV = 0) then */
endif /* elseif (ExceptionType = Interrupt) then */

/* Update the shadow set information for an implementation of */
/* Release 2 of the architecture */
if ((ArchitectureRevision ≥ 2) and (SRSCtlHSS > 0) and (StatusBEV = 0) and
    (StatusERL = 0)) then
    SRSCtlPSS ← SRSCtlCSS
    SRSCtlCSS ← NewShadowSet
endif
endif /* if StatusEXL = 1 then */

CauseCE ← FaultingCoproprocessorNumber
CauseExcCode ← ExceptionType
StatusEXL ← 1

if Config1CA = 1 then
    ISAMode ← 0
endif

/* Calculate the vector base address */
if StatusBEV = 1 then
    vectorBase ← 0xBFC0.0200
else
    if ArchitectureRevision ≥ 2 then
        /* The fixed value of EBase31..30 forces the base to be in kseg0 or kseg1 */
        vectorBase ← EBase31..12 || 0x000
    else
        vectorBase ← 0x8000.0000
    endif
endif

/* Exception PC is the sum of vectorBase and vectorOffset */
PC ← vectorBase..30 || (vectorBase29..0 + vectorOffset29..0)
/* No carry between bits 29 and 30 */

```

6.7 Debug Exception Processing

All debug exceptions have the same basic processing flow:

- The *DEPC* register is loaded with the program counter (PC) value at which execution will be restarted and the *DBD* bit is set appropriately in the *Debug* register. The value loaded into the *DEPC* register is the current PC if the instruction is not in the delay slot of a branch, or the PC-4 of the branch if the instruction is in the delay slot of a branch.
- The *DSS*, *DBp*, *DDBL*, *DDBS*, *DIB*, and *DINT* bits (D* bits [5:0]) in the *Debug* register are updated appropriately, depending on the debug exception type.
- *Halt* and *Doze* bits in the *Debug* register are updated appropriately.
- The *DM* bit in the *Debug* register is set to 1.
- The processor is started at the debug exception vector.

The value loaded into *DEPC* represents the restart address for the debug exception and need not be modified by the debug exception handler software in the usual case. Debug software need not look at the *DBD* bit in the *Debug* register unless it wishes to identify the address of the instruction that actually caused the debug exception.

A unique debug exception is indicated through the *DSS*, *DBp*, *DDBL*, *DDBS*, *DIB* and *DINT* bits (D* bits [5:0]) in the *Debug* register.

No other CP0 registers or fields are changed due to the debug exception, and thus no additional state is saved.

Operation:

```

if InstructionInBranchDelaySlot then
    DEPC ← PC-4
    DebugDBD ← 1
else
    DEPC ← PC
    DebugDBD ← 0
endif
DebugD* bits at [5:0] ← DebugExceptionType
DebugHalt ← HaltStatusAtDebugException
DebugDoze ← DozeStatusAtDebugException
DebugDM ← 1
if EJTAGControlRegisterProbTrap = 1 then
    PC ← 0xFF20_0200
else
    if DebugControlRegisterRDVec = 1 then
        if CacheErr then
            PC ← 2#101 || DebugVectorAddr28..7 || 2#0000000
        else
            PC ← 2#10 || DebugVectorAddr29..7 || 2#0000000
        else
            if SI_UseExceptionBase
                if CacheErr then
                    PC ← 2#101 || SI_ExceptionBase[28:12] || 0x000
                else
                    PC ← 2#10 || SI_ExceptionBase[29:12] || 0x000
            else

```

Exceptions and Interrupts in the 74K™ Core

```

                PC ← 0xBFC0_0480
    endif

```

The location of the debug exception vector is determined by the *ProbTrap* bit in the *EJTAG Control* register (*ECR*) and the *RDVec* bit in the *Debug Control* register (*DCR*), as shown in [Table 6.10](#).

Table 6.10 Debug Exception Vector Addresses

ProbTrap bit in ECR Register	RDVec bit in DCR Register	Debug Exception Vector Address
0	0	0xBFC0 0480
0	1	DebugVectorAddr _{31:7} 0000000
1	0	0xFF20 0200 in dmseg
1	1	

The value in the optional drseg register *DebugVectorAddr* (offset 0x00020) is used as the debug exception vector when the *ECR ProbTrap* bit is 0 and when enabled through the optional *RDVec* control bit in the *Debug Control Register* (*DCR*). Bit 0 of *DebugVectorAddr* determines the ISA mode used to execute the handler. [Figure 6.3](#) shows the format of the *DebugVectorAddr* register; [Table 6.11](#) describes the *DebugVectorAddr* register fields.

Figure 6.3 DebugVectorAddr Register Format

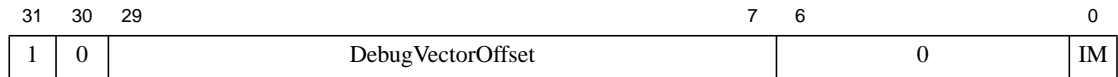


Table 6.11 DebugVectorAddr Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
1	31	Ignored on write; returns one on read.	R	1
DebugVectorOffset	29:7	Programmable Debug Exception Vector Offset	R/W	Preset to 0x7F8009
IM	0	ISA mode to be used for exception handler	R	0
0	30,6:1	Ignored on write; returns zero on read.	R	0

Bits 31..30 of the *DebugVectorAddr* register are fixed with the value 0b10, and the addition of the base address and the exception offset is done inhibiting a carry between bit 29 and bit 30 of the final exception address. The combination of these two restrictions forces the final exception address to be in the kseg0 or kseg1 unmapped virtual address segments. For cache error exceptions, bit 29 is forced to a 1 in the ultimate exception base address, so that this exception always runs in the kseg1 unmapped, uncached virtual address segment.

When MIPS16 is implemented, the power-up state of *IM* is zero. If the implementation does not include MIPS16, the *IM* field is read-only, should be written with zero and will return 0 on a read.

If the TAP is not implemented, the debug exception vector location is as if *ProbTrap*=0.

6.8 Exception Descriptions

The following subsections describe each of the exceptions listed in the same sequence as shown in [Table 6.1](#).

6.8.1 Reset Exception

A reset exception occurs when the *SI_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 is 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* register is initialized with its boot state.
- The *RP*, *BEV*, *TS*, *SR*, *NMI*, and *ERL* fields of the *Status* register are initialized to a specified state.
- The *I*, *R*, and *W* fields of the *WatchLo* register are initialized to 0.
- 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 0xBFC0_0000.

Cause Register ExcCode Value:

None

Additional State Saved:

None

Entry Vector Used:

Reset (0xBFC0_0000)

Operation:

```

Random ← TLBEntries - 1
Wired ← 0
Config ← ConfigurationState
StatusRP ← 0
StatusBEV ← 1
StatusTS ← 0
StatusSR ← 0
StatusNMI ← 0
StatusERL ← 1
WatchLoI ← 0
WatchLoR ← 0
WatchLoW ← 0
if InstructionInBranchDelaySlot then
    ErrorEPC ← PC - 4
else
    ErrorEPC ← PC
endif
PC ← 0xBFC0_0000

```

6.8.2 Debug Single Step Exception

A debug single step exception occurs after the CPU has executed one/two instructions in non-debug mode, when returning to non-debug mode after debug mode. One instruction is allowed to execute when returning to a non-jump/branch instruction, otherwise two instructions are allowed to execute since the jump/branch and the instruction in the delay slot are executed as one step. Debug single step exceptions are enabled by the *SSt* bit in the *Debug* register, and are always disabled for the first one/two instructions after a DERET.

The *DEPC* register points to the instruction on which the debug single step exception occurred, which is also the next instruction to single step or execute when returning from debug mode. So the *DEPC* register will not point to the instruction which has just been single stepped, but rather the following instruction. The *DBD* bit in the *Debug* register is never set for a debug single step exception, since the jump/branch and the instruction in the delay slot is executed in one step.

Exceptions occurring on the instruction(s) executed with debug single step exception enabled are taken even though debug single step was enabled. For a normal exception (other than reset), a debug single step exception is then taken on the first instruction in the normal exception handler. Debug exceptions are unaffected by single step mode, e.g. returning to a SDBBP instruction with debug single step exceptions enabled causes a debug software breakpoint exception, and *DEPC* will point to the SDBBP instruction. However, returning to an instruction (not jump/branch) just before the SDBBP instruction, causes a debug single step exception with *DEPC* pointing to the SDBBP instruction.

To ensure proper functionality of single step, the debug single step exception has priority over all other exceptions, except reset and soft reset.

Debug Register Debug Status Bit Set

DSS

Additional State Saved

None

Entry Vector Used

Debug exception vector

6.8.3 Debug Interrupt Exception

A debug interrupt exception is either caused by the *EjtagBrk* bit in the *EJTAG Control* register (controlled through the TAP), or caused by the debug interrupt request signal to the CPU.

The debug interrupt exception is an asynchronous debug exception which is taken as soon as possible, but with no specific relation to the executed instructions. The *DEPC* register is set to the instruction where execution should continue after the debug handler is through. The *DBD* bit is set based on whether the interrupted instruction was executing in the delay slot of a branch.

Debug Register Debug Status Bit Set

DINT

Additional State Saved

None

Entry Vector Used

Debug exception vector

6.8.4 Non-Maskable Interrupt (NMI) Exception

A non maskable interrupt exception occurs when the *SI_NMI* signal is asserted to the processor. *SI_NMI* is an edge sensitive signal - only one NMI exception will be taken each time it is asserted. An NMI exception occurs only at instruction boundaries, so it does not cause any reset or other hardware initialization. The state of the cache, memory, and other processor states are 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 0xBFC0_0000.

Cause Register ExcCode Value:

None

Additional State Saved:

None

Entry Vector Used:

Reset (0xBFC0_0000)

Operation:

```

StatusBEV ← 1
StatusTS ← 0
StatusSR ← 0
StatusNMI ← 1
StatusERL ← 1
if InstructionInBranchDelaySlot then
    ErrorEPC ← PC - 4
else
    ErrorEPC ← PC
endif
PC ← 0xBFC0_0000

```

6.8.5 Machine Check Exception

A machine check exception occurs when the processor detects an internal inconsistency. The following condition causes a machine check exception:

- The detection of multiple matching entries in the TLB. The core detects this condition on a TLB write and prevents the write from being completed. The *TS* bit in the *Status* register is set to indicate this condition. This bit is only a status flag and does not affect the operation of the device. Software clears this bit at the appropriate time. This condition is resolved by flushing the conflicting TLB entries. The TLB write can then be completed.

Cause Register ExcCode Value:

MCheck

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

6.8.6 Interrupt Exception

The interrupt exception occurs when one or more of the six hardware, two software, or timer interrupt requests is enabled by the *Status* register and the interrupt input is asserted. See 6.3 “Interrupts” on page 109 for more details about the processing of interrupts.

Register ExcCode Value:

Int

Additional State Saved:

Table 6.12 Register States an Interrupt Exception

Register State	Value
<i>CauseIP</i>	Indicates the interrupts that are pending.

Entry Vector Used:

See 6.3.2 “Generation of Exception Vector Offsets for Vectored Interrupts” on page 117 for the entry vector used, depending on the interrupt mode the processor is operating in.

6.8.7 Debug Instruction Break Exception

A debug instruction break exception occurs when an instruction hardware breakpoint matches an executed instruction. The *DEPC* register and *DBD* bit in the *Debug* register indicate the instruction that caused the instruction hardware breakpoint to match. This exception can only occur if instruction hardware breakpoints are implemented.

Debug Register Debug Status Bit Set:

DIB

Additional State Saved:

None

Entry Vector Used:

Debug exception vector

6.8.8 Watch Exception — Instruction Fetch or Data Access

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 and the *DM* bit of the *Debug* register is also zero. If any of those bits is a one at the time that a watch exception would normally be taken, then the *WP* bit in the *Cause* register is set, and the exception is deferred until all three bits 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.

The Watch exception can occur on either an instruction fetch or a data access. Watch exceptions that occur on an instruction fetch have a higher priority than watch exceptions that occur on a data access.

Register ExcCode Value:

WATCH

Additional State Saved:

Table 6.13 Register States on Watch Exception

Register State	Value
<i>Cause_{WP}</i>	Indicates that the watch exception was deferred until after <i>Status_{EXL}</i> , <i>Status_{ERL}</i> , and <i>Debug_{DM}</i> 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.
<i>WatchHi_{I,R,W}</i>	Set for the watch channel that matched, and indicates which type of match there was.

Entry Vector Used:

General exception vector (offset 0x180)

6.8.9 Address Error Exception — Instruction Fetch/Data Access

An address error exception occurs on an instruction or data access when an attempt is made to execute one of the following:

- Fetch an instruction, load a word, or store a word that is not aligned on a word boundary
- Load or store a halfword that is not aligned on a halfword boundary
- Reference the kernel address space from user mode

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 to the unaligned instruction address. In the case of a data access the exception is taken if either an unaligned address or an address that was inaccessible in the current processor mode was referenced by a load or store instruction.

Cause Register ExcCode Value:

ADEL: Reference was a load or an instruction fetch

ADES: Reference was a store

Additional State Saved:

Table 6.14 CP0 Register States on Address Exception Error

Register State	Value
<i>BadVAddr</i>	Failing address
<i>Context_{VPN2}</i>	UNPREDICTABLE
<i>EntryHi_{VPN2}</i>	UNPREDICTABLE
<i>EntryLo0</i>	UNPREDICTABLE
<i>EntryLo1</i>	UNPREDICTABLE

Entry Vector Used:

General exception vector (offset 0x180)

6.8.10 TLB Refill Exception — Instruction Fetch or Data Access

During an instruction fetch or data access, a TLB refill exception occurs when no TLB entry matches a reference to a mapped address space and the *EXL* bit is 0 in the *Status* register. Note that this is distinct from the case in which an entry matches but has the valid bit off. In that case, a TLB Invalid exception occurs.

Cause Register ExcCode Value:

TLBL: Reference was a load or an instruction fetch

TLBS: Reference was a store

Additional State Saved:

Table 6.15 CP0 Register States on TLB Refill Exception

Register State	Value
<i>BadVAddr</i>	Failing address.
<i>Context</i>	The <i>BadVPN2</i> field contains VA _{31:13} of the failing address.
<i>EntryHi</i>	The <i>VPN2</i> field contains VA _{31:13} of the failing address; the <i>ASID</i> field contains the ASID of the reference that missed.
<i>EntryLo0</i>	UNPREDICTABLE
<i>EntryLo1</i>	UNPREDICTABLE

Entry Vector Used:

TLB refill vector (offset 0x000) if *Status_{EXL}* = 0 at the time of exception;

General exception vector (offset 0x180) if *Status_{EXL}* = 1 at the time of exception

6.8.11 TLB Invalid Exception — Instruction Fetch or Data Access

During an instruction fetch or data access, a TLB invalid exception occurs in one of the following cases:

- No TLB entry matches a reference to a mapped address space; and the *EXL* bit is 1 in the *Status* register.

- A TLB entry matches a reference to a mapped address space, but the matched entry has the valid bit off.

Cause Register ExcCode Value:

TLBL: Reference was a load or an instruction fetch

TLBS: Reference was a store

Additional State Saved:**Table 6.16 CP0 Register States on TLB Invalid Exception**

Register State	Value
<i>BadVAddr</i>	Failing address
<i>Context</i>	The BadVPN2 field contains VA _{31:13} of the failing address.
<i>EntryHi</i>	The VPN2 field contains VA _{31:13} of the failing address; the ASID field contains the ASID of the reference that missed.
<i>EntryLo0</i>	UNPREDICTABLE
<i>EntryLo1</i>	UNPREDICTABLE

Entry Vector Used:

General exception vector (offset 0x180)

6.8.12 Cache Error Exception

A cache error exception occurs when an instruction or data reference detects a cache tag or data error. This exception is not maskable. Because the error was in a cache, the exception vector is to an unmapped, uncached address. This exception can be imprecise and the *ErrorEPC* may not point to the instruction that saw the error. L2 cache errors are considered to be imprecise. An L2 cache error on a data load operation can potentially corrupt the target GPR.

Cause Register ExcCode Value

N/A

Additional State Saved**Table 6.17 CP0 Register States on Cache Error Exception**

Register State	Value
<i>CacheErr</i>	Error state
<i>ErrorEPC</i>	Restart PC

Entry Vector Used

Cache error vector (offset 0x100)

6.8.13 Bus Error Exception — Instruction Fetch or Data Access

A bus error exception occurs when an instruction or data access makes a bus request (due to a cache miss or an uncacheable reference) and that request terminates in an error. The bus error exception can occur on either an instruc-

Exceptions and Interrupts in the 74K™ Core

tion fetch or a data read. Bus error exceptions cannot be generated on data writes. Bus error exceptions that occur on an instruction fetch have a higher priority than bus error exceptions that occur on a data access.

Instruction errors are precise, will Data bus errors can be imprecise. These errors are taken when the ERR code is returned on the *OC_SResp* input.

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)

6.8.14 Debug Software Breakpoint Exception

A debug software breakpoint exception occurs when an SDBBP instruction is executed. The *DEPC* register and DBD bit in the *Debug* register will indicate the SDBBP instruction that caused the debug exception.

Debug Register Debug Status Bit Set:

DBp

Additional State Saved:

None

Entry Vector Used:

Debug exception vector

6.8.15 Execution Exception — System Call

The system call exception is one of the execution exceptions. All of these exceptions have the same priority. 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)

6.8.16 Execution Exception — Breakpoint

The breakpoint exception is one of the execution exceptions. All of these exceptions have the same priority. 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)

6.8.17 Execution Exception — Reserved Instruction

The reserved instruction exception is one of the execution exceptions. All of these exceptions have the same priority. A reserved instruction exception occurs when a reserved or undefined major opcode or function field is executed. This includes Coprocessor 2 instructions which are decoded reserved in the Coprocessor 2.

Cause Register ExcCode Value:

RI

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

6.8.18 Execution Exception — Coprocessor Unusable

The coprocessor unusable exception is one of the execution exceptions. All of these exceptions have the same priority. A coprocessor unusable exception occurs when an attempt is made to execute a coprocessor instruction for one of the following:

- a corresponding coprocessor unit that has not been marked usable by setting its CU bit in the *Status* register
- CP0 instructions, when the unit has not been marked usable, and the processor is executing in user mode

Cause Register ExcCode Value:

CpU

Additional State Saved:**Table 6.18 Register States on Coprocessor Unusable Exception**

Register State	Value
<i>Cause_{CE}</i>	Unit number of the coprocessor being referenced

Entry Vector Used:

General exception vector (offset 0x180)

6.8.19 Execution Exception — CorExtend block Unusable

The CorExtend block unusable exception is one of the execution exceptions. All of these exceptions have the same priority. A CEU exception occurs when an attempt is made to execute a CorExtend instruction when the CEE bit in the *Status* register is not set. It is dependent on the implementation of the CorExtend block, but this exception should be taken on any CorExtend instruction that modifies local state within the CorExtend block and can optionally be taken on other CorExtend instructions.

Cause Register ExcCode Value:

CEU

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

6.8.20 Execution Exception — DSP ASE State Disabled

The DSP ASE State Disabled exception an execution exception. It occurs when an attempt is made to execute a DSP ASE instruction when the MX bit in the *Status* register is not set. This allows an OS to do “lazy” context switching.

Cause Register ExcCode Value:

DSPDis

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

6.8.21 Execution Exception — Floating Point Exception

A floating point exception is initiated by the floating point coprocessor.

Cause Register ExcCode Value:

FPE

Additional State Saved:

Table 6.19 Register States on Floating Point Exception

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

Entry Vector Used:

General exception vector (offset 0x180)

6.8.22 Execution Exception — Integer Overflow

The integer overflow exception is one of the execution exceptions. All of these exceptions have the same priority. 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)

6.8.23 Execution Exception — Trap

The trap exception is one of the execution exceptions. All of these exceptions have the same priority. 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)

6.8.24 Debug Data Break Exception

A debug data break exception occurs when a data hardware breakpoint matches the load/store transaction of an executed load/store instruction. The *DEPC* register and *DBD* bit in the *Debug* register will indicate the load/store instruction that caused the data hardware breakpoint to match. The load/store instruction that caused the debug exception has not completed e.g. not updated the register file, and the instruction can be re-executed after returning from the debug handler.

Debug Register Debug Status Bit Set:

DDBL for a load instruction or *DDBS* for a store instruction

Additional State Saved:

None

Entry Vector Used:

Debug exception vector

6.8.25 TLB Modified Exception — Data Access

During a data access, a TLB modified exception occurs on a store reference to a mapped address if the following condition is true:

Exceptions and Interrupts in the 74K™ Core

- The matching TLB entry is valid, but not dirty.

Cause Register ExcCode Value:

Mod

Additional State Saved:

Table 6.20 Register States on TLB Modified Exception

Register State	Value
<i>BadVAddr</i>	Failing address
<i>Context</i>	The BadVPN2 field contains VA _{31:13} of the failing address.
<i>EntryHi</i>	The VPN2 field contains VA _{31:13} of the failing address; the ASID field contains the ASID of the reference that missed.
<i>EntryLo0</i>	UNPREDICTABLE
<i>EntryLo1</i>	UNPREDICTABLE

Entry Vector Used:

General exception vector (offset 0x180)

6.9 Exception Handling and Servicing Flowcharts

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

- General exceptions
- TLB miss exceptions
- Reset and NMI exceptions
- Debug exceptions

Generally speaking, exceptions are handled by hardware and then serviced by software. Note that unexpected debug exceptions to the debug exception vector at 0xBFC0_0200 may be viewed as a reserved instruction since uncontrolled execution of an SDBBP instruction caused the exception. The DERET instruction must be used at return from the debug exception handler, in order to leave debug mode and return to non-debug mode. The DERET instruction returns to the address in the *DEPC* register.

Figure 6.4 General Exception Handler (HW)

Exceptions other than Reset, NMI, or first-level TLB miss. Note: Interrupts can be masked by IE or IMs, and Watch is masked if EXL = 1.

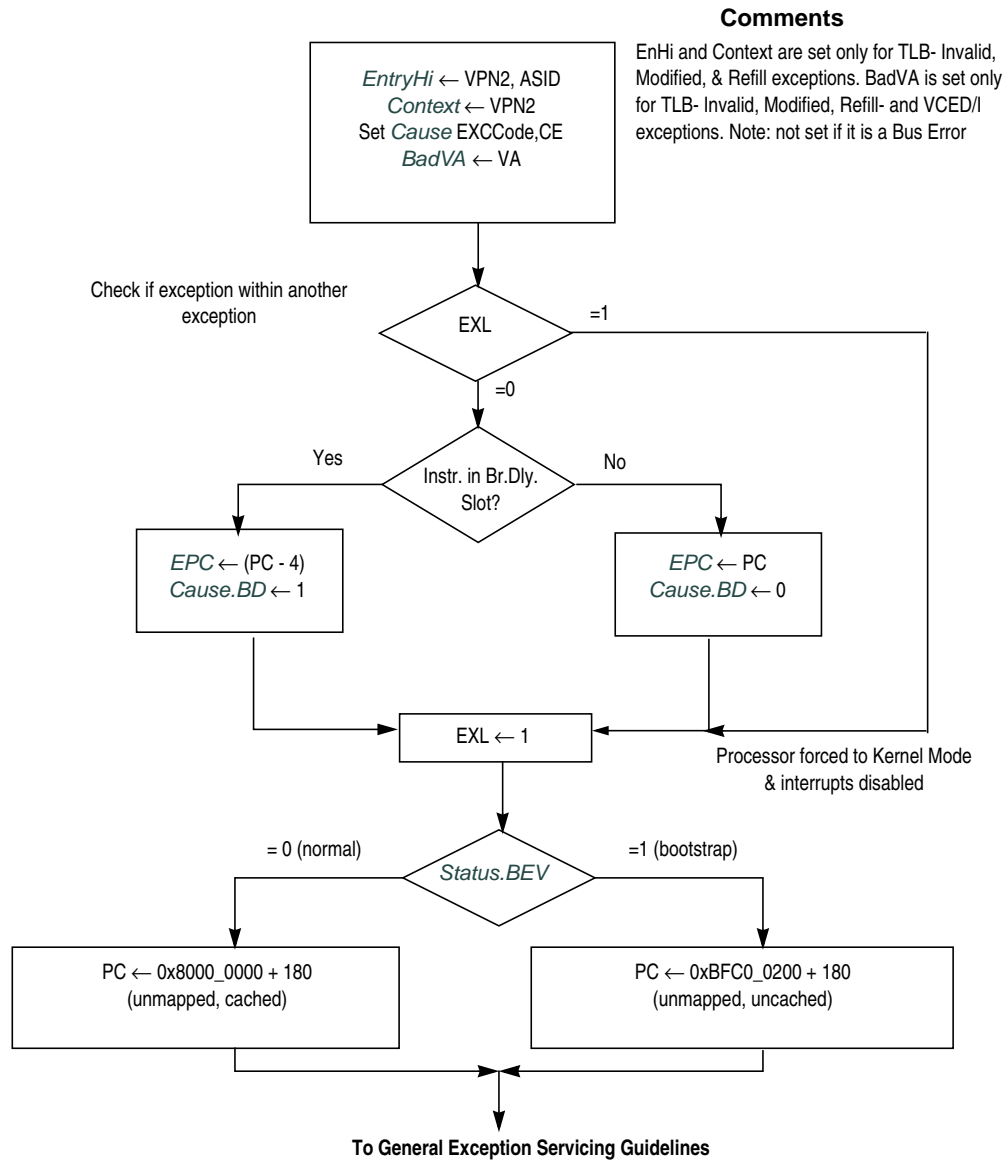


Figure 6.5 General Exception Servicing Guidelines (SW)

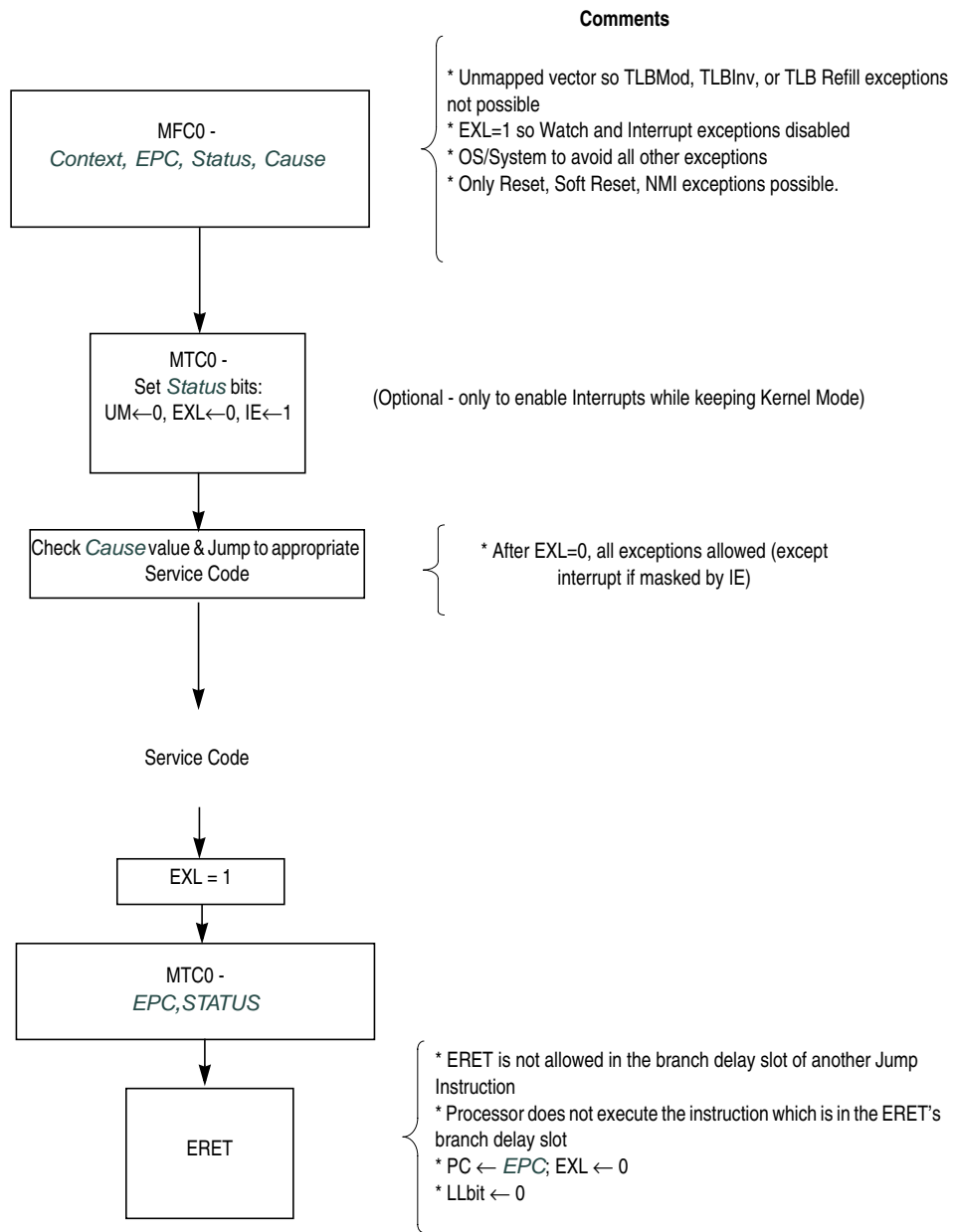


Figure 6.6 TLB Miss Exception Handler (HW)

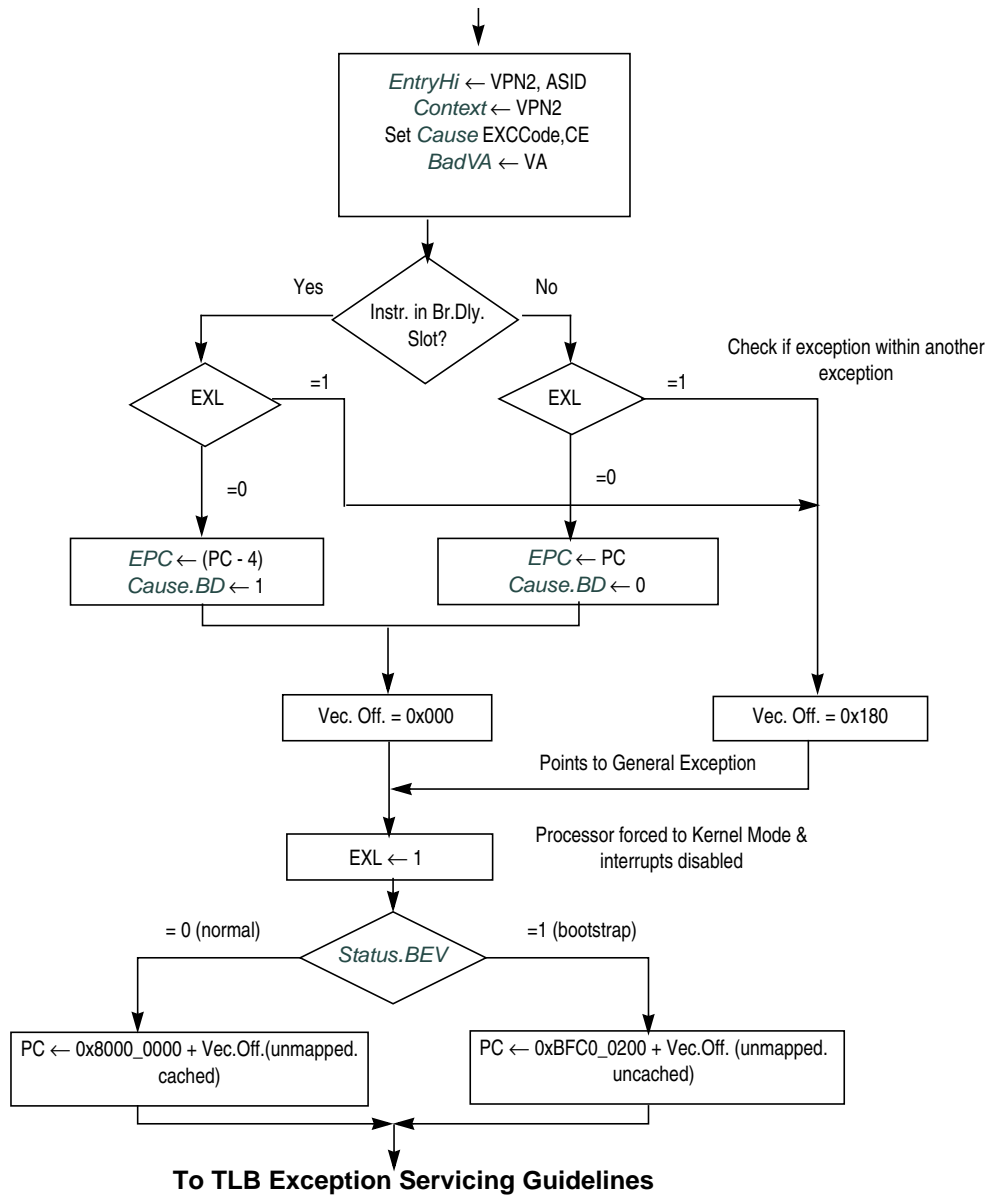


Figure 6.7 TLB Exception Servicing Guidelines (SW)

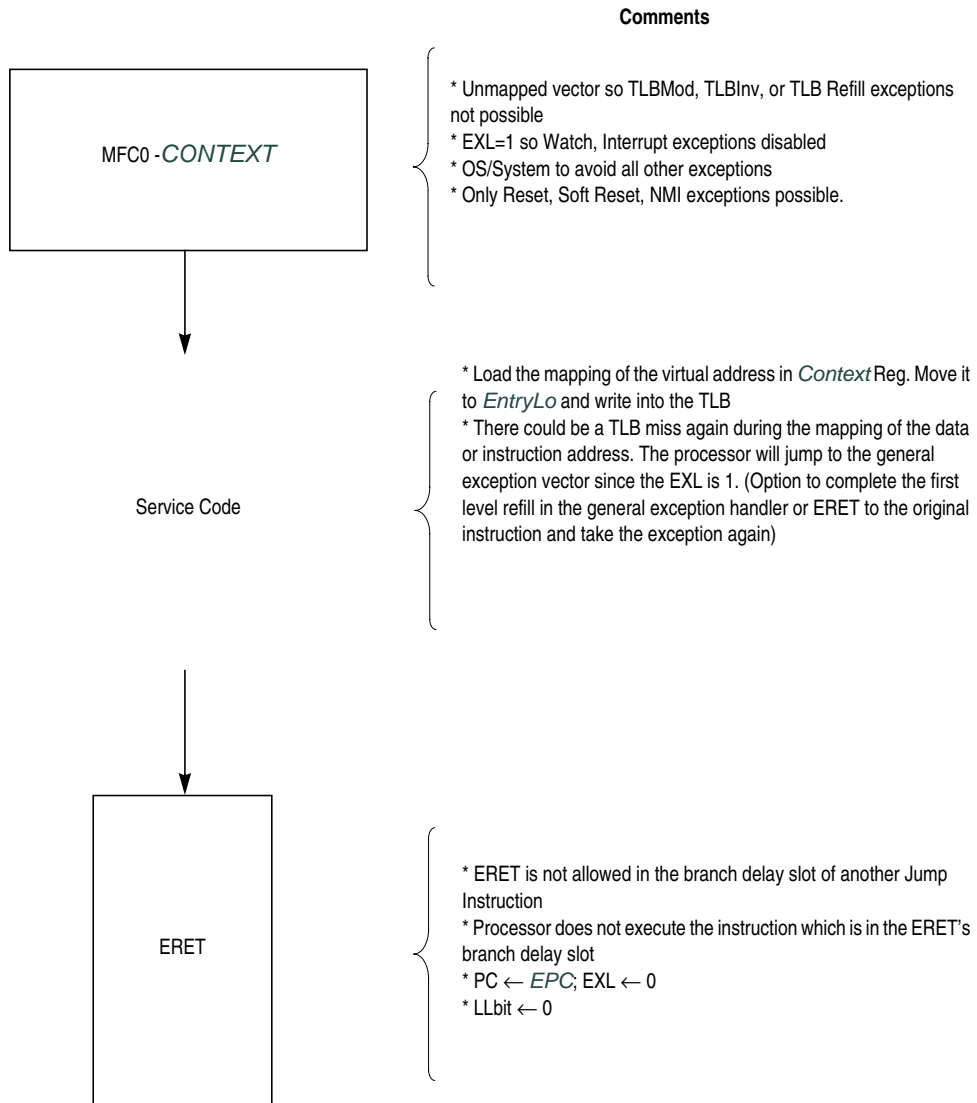
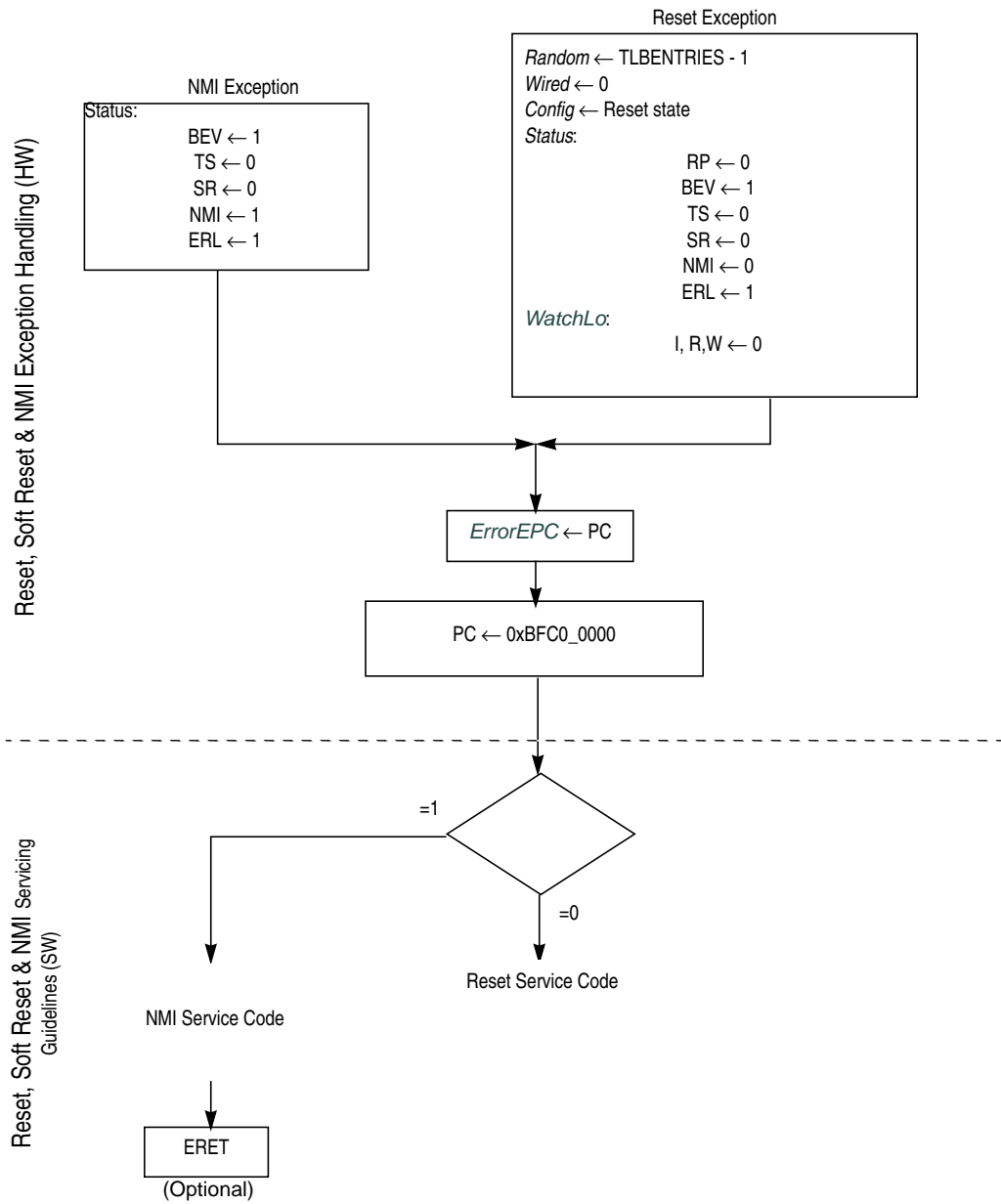


Figure 6.8 Reset and NMI Exception Handling and Servicing Guidelines



CP0 Registers of the 74K™ Core

The System Control Coprocessor (CP0) provides the register interface to the 74K processor core and supports memory management, address translation, exception handling, and other privileged operations. Each CP0 register has a unique number that identifies it, referred to as its *register number*. CP0 register numbers are denoted by *n.s*, where *n* is the register number (between 0-31) and *s* is the "select" field (0-7). If the select field is omitted, it is zero. A select field of *x* denotes all eight potential select numbers.

This chapter contains the following sections:

- [Section 7.1 “CP0 Register Summary”](#)
- [Section 7.2 “CP0 Register Descriptions”](#)

The CP0 EJTAG registers are described in [Chapter 11, “EJTAG Debug Support in the 74K™ Core.”](#)

7.1 CP0 Register Summary

The CP0 registers are described in three tables. [Table 7.1](#) lists the registers in alphabetic order, [Table 7.2](#) lists the registers in numerical order, and [Table 7.3](#) groups the registers according to their function. The CP0 registers are described individually in [Section 7.2, “CP0 Register Descriptions.”](#)

Table 7.1 CP0 Registers in Alphabetical Order

Name	Number	Name	Number	Name	Number	Name	Number
<i>BadVAddr</i>	8.0	<i>DDataLo</i>	28.3	<i>IDataLo</i>	28.1	<i>SRSMap</i>	12.3
<i>CacheErr</i>	27.0	<i>Debug</i>	23.0	<i>Index</i>	0.0	<i>Status</i>	12.0
<i>Cause</i>	13.0	<i>DEPC</i>	24.0	<i>IntCtl</i>	12.1	<i>TraceControl</i>	23.1
<i>CDMMBase</i>	15.2	<i>DESAVE</i>	31.0	<i>ITagHi</i>	29.0	<i>TraceControl2</i>	23.2
		<i>DTagHi</i>	29.2	<i>ITagLo</i>	28.0	<i>TraceControl3</i>	24.2
<i>Compare</i>	11.0	<i>DTagLo</i>	28.2	<i>L23DataHi</i>	29.5	<i>TraceIPBC</i>	23.4
<i>Config</i>	16.0	<i>EBase</i>	15.1	<i>L23DataLo</i>	28.5	<i>TraceDPBC</i>	23.5
<i>Config1-2</i>	16.1-2	<i>EntryHi</i>	10.0	<i>L23TagLo</i>	28.4	<i>UserLocal</i>	4.2
<i>Config3</i>	16.3	<i>EntryLo0-1</i>	2.0 3.0	<i>PageMask</i>	5.0	<i>UserTraceData1</i>	23.3
<i>Config6</i>	16.6	<i>EPC</i>	14.0	<i>PerfCnt0-3</i>	25.1 25.3 25.5 25.7	<i>UserTraceData2</i>	24.3

Table 7.1 CP0 Registers in Alphabetical Order (Continued)

Name	Number	Name	Number	Name	Number	Name	Number
<i>Config7</i>	16.7	<i>ErrCtl</i>	26.0	<i>PerfCtl0-3</i>	25.0 25.2 25.4 25.6	<i>WatchHi0-3</i>	19.0-3
<i>Context</i>	4.0	<i>ErrorEPC</i>	30.0	<i>PRId</i>	15.0	<i>WatchLo0-3</i>	18.0-3
<i>ContextConfig</i>	4.1	<i>HWREna</i>	7.0	<i>Random</i>	1.0	<i>Wired</i>	6.0
<i>Count</i>	9.0	<i>IDataHi</i>	29.1	<i>SRSCtl</i>	12.2		

Table 7.2 CP0 Registers in Numerical Order

Number	Register	Description	Page
0.0	<i>Index</i>	Index into the TLB array	7.2.1, p.151
1.0	<i>Random</i>	Randomly generated index into the TLB array	7.2.2, p.152
2.0 3.0	<i>EntryLo0-1</i>	Output (physical) side of TLB entry (even-/odd-numbered virtual pages)	7.2.3, p.152
4.0	<i>Context</i>	Mixture of pre-programmed and <i>BadVAddr</i> bits which can act as an OS page table pointer.	7.2.4, p.154
4.1	<i>ContextConfig</i>	Defines the bits of the <i>Context</i> register into which the high order bits of the virtual address causing a TLB exception will be written.	7.2.5, p.155
4.2	<i>UserLocal</i>	Kernel-writable but user-readable software-defined thread ID	7.2.6, p.156
5.0	<i>PageMask</i>	Control for variable page size in TLB entries	7.2.7, p.157
6.0	<i>Wired</i>	Controls the number of fixed ("wired") TLB entries	7.2.8, p.158
7.0	<i>HWREna</i>	Bitmask limiting user-mode access to rdhwr registers	7.2.9, p.158
8.0	<i>BadVAddr</i>	Address causing the last TLB-related exception	7.2.10, p.160
9.0	<i>Count</i>	Free-running counter at half pipeline speed	7.11, p.160
10.0	<i>EntryHi</i>	High-order portion of the TLB entry	7.2.12, p.161
11.0	<i>Compare</i>	Timer interrupt control	7.2.13, p.162
12.0	<i>Status</i>	Processor status and control	7.2.14, p.162
12.1	<i>IntCtl</i>	Setup for interrupt vector and interrupt priority features.	7.2.15, p.168
12.2	<i>SRSCtl</i>	Shadow register set selectors	7.2.16, p.169
12.3	<i>SRSMap</i>	Shadow set choice for each interrupt level in VI mode	7.2.17, p.171
13.0	<i>Cause</i>	Cause of last general exception	7.2.18, p.172
14.0	<i>EPC</i>	Restart address from exception	7.2.19, p.176
15.0	<i>PRId</i>	Processor identification and revision	7.2.20, p.177
15.1	<i>EBase</i>	Exception entry point base address and CPU/VPE ID	7.2.21, p.178
15.2	<i>CDMMBase</i>	36-bit physical base address for the Common Device Memory Map facility	7.2.22, p.179
16.0	<i>Config</i>	Legacy configuration register	7.2.23, p.180

Table 7.2 CP0 Registers in Numerical Order (Continued)

Number	Register	Description	Page
16.1-2	<i>Config1-2</i>	MIPS32/64 configuration registers (caches etc)	7.2.24, p.181
16.3	<i>Config3</i>	Configuration register showing ASEs etc	7.2.25, p.183
16.6	<i>Config6</i>	Additional information about the presence of optional extensions to the base MIPS32 architecture	7.2.26, p.184
16.7	<i>Config7</i>	CPU-specific configuration	7.2.27, p.187
18.0-3	<i>WatchLo0-3</i>	Watchpoint address and qualifiers	7.2.28, p.190
19.0-3	<i>WatchHi0-3</i>	Watchpoint control/status	7.2.29, p.190
23.0	<i>Debug</i>	EJTAG Debug status/control register	7.2.30, p.191
23.1	<i>TraceControl</i>	EJTAG Trace Control register	7.2.31, p.195
23.2	<i>TraceControl2</i>	EJTAG Trace Control2 register	7.2.32, p.197
23.3	<i>UserTraceData1</i>	EJTAG User Trace Data1 register	7.2.33, p.199
23.4	<i>TraceIBPC</i>	EJTAG Trace Instruction breakpoint control register	7.2.34, p.200
23.5	<i>TraceDBPC</i>	EJTAG Trace Data breakpoint control register	7.2.35, p.201
24.0	<i>DEPC</i>	Restart address from last EJTAG debug exception	7.2.36, p.202
24.2	<i>TraceControl3</i>	EJTAG Trace Control3 register	7.2.37, p.203
24.3	<i>UserTraceData2</i>	EJTAG User Trace Data2 register	7.2.33, p.199
25.0 25.2 25.4 25.6	<i>PerfCtl0-3</i>	Performance counter control	7.2.38, p.204
25.1 25.3 25.5 25.7	<i>PerfCnt0-3</i>	Performance counters	7.2.39, p.209
26.0	<i>ErrCtl</i>	Software parity control and test modes for cache RAM arrays	7.2.40, p.209
27.0	<i>CacheErr</i>	Cache parity exception status	7.2.41, p.210
28.0	<i>ITagLo</i>	Read/write interface for I-cache tag cacheops	7.2.42, p.212
28.1	<i>IDataLo</i>	Low-order data read/write interface for I-cache special cacheops	7.2.43, p.213
28.2	<i>DTagLo</i>	Read/write interface for load/store tag cacheops	7.2.44, p.214
28.3	<i>DDataLo</i>	Low-order data read/write interface for D-cache	7.2.45, p.216
28.4	<i>L23TagLo</i>	Level 2/3 cache Tag information	7.2.46, p.216
28.5	<i>L23DataLo</i>	Low-order data read/write interface for Level 2/3 cache	7.2.47, p.216
29.0	<i>ITagHi</i>	I-cache predecode bits	7.2.48, p.217
29.1	<i>IDataHi</i>	High-order data read/write interface for I-cache special cacheops	7.2.49, p.217
29.2	<i>DTagHi</i>	D-cache virtual index (including ASID)	7.2.50, p.218
29.5	<i>L23DataHi</i>	High-order data read/write interface for Level 2/3 cache	7.2.51, p.218
30.0	<i>ErrorEPC</i>	Restart location from a reset or a cache error exception	7.2.52, p.219
31.0	<i>DESAVE</i>	Scratch read/write register for EJTAG debug exception handler	7.2.53, p.219

Table 7.3 CP0 Registers Grouped by Function

Basic modes	<i>Status</i>	12.0	TLB Management	<i>BadVAddr</i>	8.0	EJTAG Debug	<i>DEPC</i>	24.0		
OS/userland thread ID	<i>UserLocal</i>	4.2		<i>Context</i>	4.0		<i>DESAVE</i>	31.0		
Exception Control	<i>Cause</i>	13.0		<i>ContextConfig</i>	4.1		<i>Debug</i>	23.0		
	<i>EPC</i>	14.0		<i>EntryHi</i>	10.0	PDtrace	<i>TraceControl</i>	23.1		
Timer	<i>Compare</i>	11.0		<i>EntryLo0-1</i>	2.0 3.0		<i>TraceControl2</i>	23.2		
	<i>Count</i>	9.0		<i>Index</i>	0.0		<i>TraceControl3</i>	24.2		
CPU Configuration	<i>Config</i>	16.0		<i>PageMask</i>	5.0		<i>TraceIPBC</i>	23.4		
	<i>Config1-2</i>	16.1-2		<i>Random</i>	1.0		<i>TraceIDBC</i>	23.5		
	<i>Config3</i>	16.3		<i>Wired</i>	6.0		<i>UserTraceData1</i>	23.3		
	<i>Config6</i>	16.6		Cache Management	<i>DDataLo</i>		28.3	<i>UserTraceData2</i>	24.3	
	<i>Config7</i>	16.7	<i>DTagHi</i>		29.2	<i>PerfCnt0-3</i>	25.1 25.3 25.5 25.7			
	Debug/Analysis	<i>EBase</i>	15.1		<i>DTagLo</i>	28.2	<i>PerfCtl0-3</i>	25.0 25.2 25.4 25.6		
		Control rdhwr Access	<i>CDMMBase</i>		15.2	<i>ErrCtl</i>	28.2	<i>PerfCnt0-3</i>	25.1 25.3 25.5 25.7	
Parity/ECC control			<i>IntCtl</i>		12.1	Debug/Analysis	<i>ErrorEPC</i>	26.0	<i>WatchHi0-3</i>	19.0-3
			<i>PRId</i>		15.0		<i>IDataHi</i>	29.1	<i>WatchLo0-3</i>	18.0-3
<i>SRSCtl</i>		12.2	<i>IDataLo</i>		28.1		Control rdhwr Access	<i>HWREna</i>	7.0	
Parity/ECC control	<i>SRSMap</i>	12.3	<i>ITagHi</i>	29.0	Parity/ECC control			<i>CacheErr</i>	27.0	
			<i>ITagLo</i>	28.0						
			<i>L23DataHi</i>	29.5						
		<i>L23TagLo</i>	28.4							

Note that after a CP0 register has been updated, there is a hazard period of zero or more instructions from the update instruction (**mtc0**) until the update has taken effect in the core. **There is no automatic handling of hazard by the hardware in the 74K core.** Release 2 requires that an **ehb** (Execution Hazard Barrier) instruction be placed between these two instructions when they address the same CPO register. For more information about the MIPS32® Release 2 Architecture guidelines on hazard barriers, refer to [Section 2.3, "Hazards."](#)

7.2 CP0 Register Descriptions

This section contains descriptions of each CP0 register. The registers are listed in numerical order, first by register number, then by select field number.

R/W Access Types

For each register described below, field descriptions include the read/write access properties of the field and the reset state of the field. The read/write access properties are described in [Table 7.4](#).

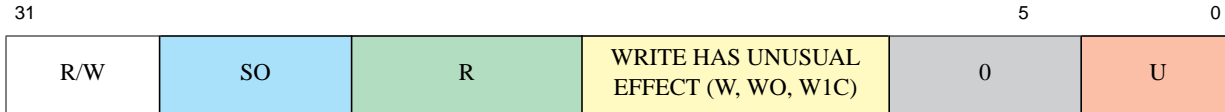
Table 7.4 CP0 Register Field R/W Access Types

Notation	Hardware Interpretation	Software Interpretation
R/W	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 reads. Software updates of this field are visible by hardware reads. 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 UNDEFINED behavior.	
SO	Software Only. A field that is read and written by software but has no hardware effect. An example is the <i>DESAVE</i> register.	
R	A field that 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.	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 UNPREDICTABLE value except after a hardware update done under the conditions specified in the description of the field.
W	A field that can be written by software but which cannot be read by software. Software reads of this field will return an UNDEFINED value.	
W0	Hardware can write 1’s or 0’s to this field.	Software writes will only cause the bit to be cleared. Software can never set this bit. An example is the <i>NMI</i> bit field in the <i>Status</i> register.
W1C	Hardware can write 1’s or 0’s to this field.	Software should write “1” to this bit to clear it. An example is the <i>I</i> , <i>R</i> , and <i>W</i> bit fields in the <i>WatchHi0-3</i> register.
0	A field that hardware does not update, and for which hardware can assume a zero value.	A field to which the value written by software must be zero. Software writes of non-zero values to this field may result in UNDEFINED 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.
U	A field that is not read or written by hardware.	Software writes to this field will be ignored. Software reads of this field will return an UNDEFINED value.

Color Coding of Register Descriptions

The color codes used in the register descriptions to indicate the access types are summarized in Figure 7.1. A field with two access types (for example, R/W0) is uncolored,

Figure 7.1 Register Format Color Coding of Access Field Types



Power-up State of CP0 Registers

The traditions of the MIPS architecture regard it as software's job to initialize CP0 registers. As a rule, only fields where a wrong setting could prevent the CPU from booting are specified to be brought to a particular state by reset; other fields—perhaps other fields in the same register—are undefined. This manual documents where a field has a forced-from-reset value; conversely, when no reset-time value is documented, that means the register comes up in an undefined state.

To ensure robust programs, you should initialize all CP0 register fields, except those in which a random value is known to be harmless.

A Note on Unused Fields in CP0 Registers

Unused fields in registers are marked either with the digit 0, an "X", or occasionally a "U". A field marked zero is expected to read zero; a field marked "U" is expected to read back whatever you last wrote to it; and if the field is marked "X", the value is unpredictable.

But again, for robustness, you should write unused fields **either** to a value you previously read from the same field or (if no such value is available) to zero.

7.2.1 Index (CP0 Register 0, Select 0): Index into TLB array

Index is used as the TLB index when reading or writing the TLB with **tlbr/tlbw** respectively. It is also set by a TLB probe (**tlbp**) instruction to return the location of an address match in the TLB.

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.

This register is only valid when the TLB is implemented; it is reserved if the FM is implemented.

Figure 7.2 Index Register Format



Table 7.5 Field Descriptions for Index Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>P</i>	31	Probe Failure. This bit is automatically set when a tlbp search of the TLB fails to find a matching entry.	R/W	Undefined
<i>Index</i>	5:0	An index into the TLB used for tlbwi and tlbr . It's set by tlbp when it finds a matching entry.	R/W	Undefined

7.2.2 Random (CP0 Register 1, Select 0): Randomly Generated Index into the TLB Array

The *Random* register is a read-only register whose value is used to index the TLB during a **tlbwr** instruction. It provides a quick way of replacing a TLB entry at random. *Random* is a free counter cycling through the range of valid TLB indexes. The *Random* register is decremented by one almost every clock, wrapping after the value in the *Wired* register is reached. As a result, it will not take values less than the value programmed in *Wired*.

To reduce the possibility of a live lock condition, an *LFSR* register is used which prevents the decrement pseudo-randomly.

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

Figure 7.3 Random Register Format



Table 7.6 Field Descriptions for Random Register

Name	Bit(s)	Description	Read/Write	Reset State
0	31:6	Must be written as zero; returns zero on reads.	0	0
<i>Random</i>	5:0	This field cycles "randomly" through the potential indices of the TLB, so its length varies with the TLB size (the diagram shows a max TLB size of 64 entries). It's usually a down counter, and starts off at the largest plausible index.	R	#TLB Entries — 1

7.2.3 EntryLo0-1 (CP0 Registers 2 and 3, Select 0): Output (physical) side of TLB entry

These registers hold and represent the contents of the physical (output) side of a TLB entry — each entry maps a pair of pages and *EntryLo0* and *EntryLo1* are for even-/odd-numbered virtual pages respectively. They're read during a **tlbwr** or **tlbw** instruction, and written by a **tlbr**, and are not used for anything else.

Figure 7.4 EntryLo0, EntryLo1 Register Format

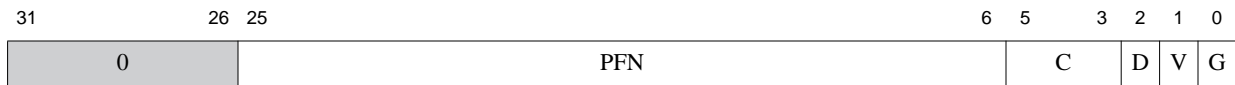


Table 7.7 Field Descriptions for EntryLo0-1 Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>PFN</i>	25:6	The "physical frame number" — traditional OS name for the high-order bits of the physical address. 20 bits of <i>PFN</i> together with 12 bits of in-page address make up a 32-bit physical address. The MIPS32® Architecture permits the <i>PFN</i> to be as large as 24 bits, but the 74K core has a 32-bit physical address bus.	R/W	Undefined
<i>C</i>	5:3	Coherency attribute of the page. See Table 7.8.	R/W	Undefined
<i>D</i>	2	The "dirty" flag. Indicates that the page has been written, and/or is writable. If this bit is a one, then stores to the page are permitted. If this bit is a zero, then stores to the page cause a TLB Modified exception. Software can use this bit to track pages that have been written to; when you first map a page, you leave this bit clear, and then the first write causes an exception which you note somewhere in the OS' memory management tables (and, of course, remember to set the bit).	R/W	Undefined
<i>V</i>	1	The "valid" flag. Indicates that the TLB entry, and thus the virtual page mapping are valid. If this bit is a one, then accesses to the page are permitted. If this bit is a zero, then accesses to the page cause a <i>TLB Invalid</i> exception. This bit can be used to make just one of a pair of pages valid.	R/W	Undefined
<i>G</i>	0	The "global" bit. On a TLB write, the logical AND of the G bits in both the <i>EntryLo0</i> and <i>EntryLo1</i> registers become the G bit in the TLB entry. If the TLB entry G bit is a one, then the ASID comparisons are ignored during TLB matches. On a read from a TLB entry, the G bits of both <i>EntryLo0</i> and <i>EntryLo1</i> reflect the state of the TLB G bit.	R/W	Undefined

Table 7.8 Cache Coherency Attributes encoding of C field of EntryLo0-1 and K0 field of Config Register

C[5:3] K0[2:0]	Cache Coherency Attribute
0	Cacheable, noncoherent, write-through, no write allocate
1	Reserved
2	uncached
3	Cacheable, noncoherent, write-back, write allocate
4	Reserved
5	Reserved
6	Reserved
7	Uncached Accelerated

7.2.4 Context (CP0 Register 4, Select 0): Mixture of Pre-programmed and BadVAddr Bits which can act as an OS Page Table Pointer.

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 duplicates some of the information provided in the *BadVAddr* register.

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.

If *Config3CTXTC*=1, the pointer implemented by the *Context* register can point to any power-of-two-sized PTE structure within memory. This allows the TLB refill handler to use the pointer without additional shifting and masking steps. Depending on the value in the *ContextConfig* register, it may point to an 8-byte pair of 32-bit PTEs within a single-level page table scheme, or to a first level page directory entry in a two-level lookup scheme.

A TLB exception (Refill, Invalid, or Modified) causes bits $VA_{31:32-X+Y}$ to be written to a variable range of bits “(X-1):Y” of the *Context* register, where this range corresponds to the contiguous range of set bits in the *ContextConfig* register. Bits 31:X are R/W to software, and are unaffected by the exception. Bits Y-1:0 will always read as 0. If X = 23 and Y = 4, i.e. bits 22:4 are set in *ContextConfig*, the behavior is identical to the standard MIPS32 *Context* register (bits 22:4 are filled with $VA_{31:13}$). Although the fields have been made variable in size and interpretation, the MIPS32 nomenclature is retained. Bits 31:X are referred to as the *PTEBase* field, and bits X-1:Y are referred to as *BadVPN2*. When *Config3CTXTC*=0, X = 23, Y = 4.

The value of the *Context* register is **UNPREDICTABLE** following a modification of the contents of the *ContextConfig* register.

Figure 7-5 shows the format of the *Context* Register; Table 7.9 describes the *Context* register fields.

Figure 7-5 Context Register Format



Table 7.9 Context Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
<i>PTEBase</i>	Variable, 31:X where X in {31..0}. May be null. If <i>Config3CTXTC</i> =0, X = 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 <i>Context</i> Register as a pointer to an array of data structures in memory corresponding to the address region that contains the virtual address which caused the exception.	R/W	Undefined

Table 7.9 Context Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
<i>BadVPN2</i>	Variable, (X-1):Y where X in {32..1} and Y in {31..0}. May be null. If <i>Config3CTXTC</i> =0, X = 23, Y = 4	This field is written by hardware on a TLB exception. It contains bits VA _{31:32-X+Y} of the virtual address that caused the exception.	R	Undefined
0	Variable, (Y-1):0 where Y in {31:1}. May be null. If <i>Config3CTXTC</i> =0, Y = 4	Must be written as zero; returns zero on read.	0	0

7.2.5 ContextConfig Register (CP0 Register 4, Select 1)

The *ContextConfig* register defines the bits of the *Context* register into which the high order bits of the virtual address causing a TLB exception will be written, and how many bits of that virtual address will be extracted. Bits above the selected of the *Context* register are R/W to software and serve as the *PTEBase* field. Bits below the selected field of the *Context* register will read as zeroes.

The field to contain the virtual address index is defined by a single block of contiguous non-zero bits within the *ContextConfig* register's *VirtualIndex* field. Any zero bits to the right of the least significant one bit cause the corresponding *Context* register bits to read as zero. Any zero bits to the left of the most significant one bit cause the corresponding *Context* register bits to be R/W to software and unaffected by TLB exceptions.

A value of all ones in the *ContextConfig* register means that the full 32 bits of the faulting virtual address will be copied into the context register, making it duplicate the *BadVAddr* register. A value of all zeroes means that the full 32 bits of the *Context* register are R/W for software and unaffected by TLB exceptions.

The *ContextConfig* register is optional and its existence is denoted by the *Config3CTXTC* or *Config3SM* register fields.

Figure 7.6 shows the formats of the *ContextConfig* Register; Table 7.10 describes the *ContextConfig* register fields.

Figure 7.6 ContextConfig Register Format

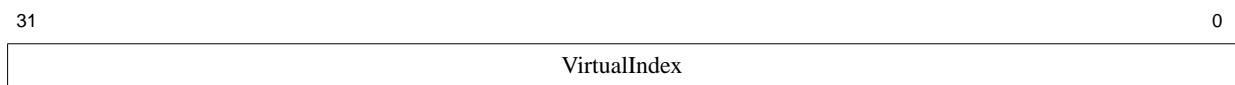


Table 7.10 ContextConfig Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
VirtualIndex	31:0	A mask of 0 to 32 contiguous 1 bits in this field causes the corresponding bits of the <i>Context</i> register to be written with the high-order bits of the virtual address causing a TLB exception. Behavior of the processor is UNDEFINED if non-contiguous 1 bits are written into the register field.	R/W	0x007ffff0

It is permissible to implement a subset of the *ContextConfig* register, in which some number of bits are read-only and set to one or zero as appropriate. It is possible for software to determine which bits are implemented by alternately writing all zeroes and all ones to the register, and reading back the resulting values. Table 7.11 describes some useful *ContextConfig* values.

Table 7.11 Recommended ContextConfig Values

Value	Page Table Organization	Page Size	PTE Size	Compliance
0x00000000007ffff0	Single Level	4K	64 bits/page	REQUIRED
0x00000000003ffff8	Single Level	4K	32 bits/page	RECOMMENDED
0x00000000007ffff8	Single Level	2K	32 bits/page	RECOMMENDED
0x0000000000ffff8	Single Level	1K	32 bits/page	RECOMMENDED

7.2.6 UserLocal (CP0 Register 4, Select 2): Address Causing the Last TLB-related Exception

UserLocal is a read-write 32-bit register that is not interpreted by the hardware and conditionally readable by software. This register is suitable for a kernel-maintained thread ID whose value can be read by user-level code with **rdhwr 29**, as long as *HWRENA_{UL}* is set.

The presence of the *UserLocal* register is indicated by *Config3_{ULRF}*=1.

Figure 7.7 iUserLocal Register Format

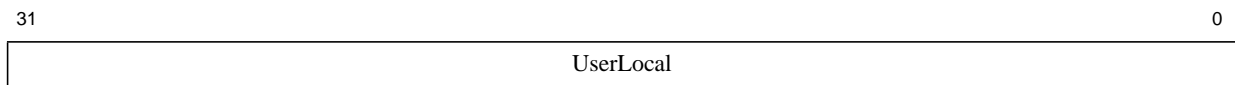


Table 7.12 UserLocal Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bits			
<i>UserLocal</i>	31:0	Software information that is not interpreted by hardware.	R/W	Undefined

7.2.7 PageMask (CP0 Register 5, Select 0): Control for Variable Page Size in TLB Entries

Every TLB entry has an independent virtual-address mask that allows it to ignore some address bits when deciding to match. By selectively ignoring lower page addresses, the entry can be made to match all the addresses in a "page" larger than 4KB.

Software can determine which page sizes are supported by writing all ones to the *PageMask* register, then reading the value back. 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 *Mask* field with a value other than one of those listed in Table 7.13, even if the hardware returns a different value on read. Hardware may depend on this requirement in implementing hardware structures.

Figure 7.8 PageMask Register Format



Table 7.13 Field Descriptions for PageMask Register

Name	Bit(s)	Description	Read/Write	Reset State																				
0	31:29, 12:0	Ignored on write; returns zero on read.	R	0																				
<i>Mask</i>	28:13	Acts as a kind of backward mask, in that a 1 bit means "don't compare this address bit when matching this address". However, only a restricted range of <i>PageMask</i> values are legal (i.e., with "1"s filling the <i>PageMaskMask</i> field from low bits upward, two at a time): <table border="1" style="margin: 10px auto;"> <thead> <tr> <th>PageMask Value</th> <th>Size of Each Output Page</th> </tr> </thead> <tbody> <tr><td>0x0000 . 0000</td><td>4 Kbytes</td></tr> <tr><td>0x0000 . 6000</td><td>16 Kbytes</td></tr> <tr><td>0x0001 . E000</td><td>64 Kbytes</td></tr> <tr><td>0x0007 . E000</td><td>256 Kbytes</td></tr> <tr><td>0x001F . E000</td><td>1 Mbyte</td></tr> <tr><td>0x007F . E000</td><td>4 Mbytes</td></tr> <tr><td>0x01FF . E000</td><td>16 Mbytes</td></tr> <tr><td>0x07FF . E000</td><td>64 Mbytes</td></tr> <tr><td>0x1FFF . E000</td><td>256 Mbytes</td></tr> </tbody> </table> <p>Note that the uTLBs handle only 4Kbyte and 1Mbyte page sizes; other page sizes are down-converted to 4Kbyte or 1Mbyte as they are referenced. For other page sizes, this may cause an unexpectedly high rate of uTLB misses, which could lead to a noticeable performance loss.</p>	PageMask Value	Size of Each Output Page	0x0000 . 0000	4 Kbytes	0x0000 . 6000	16 Kbytes	0x0001 . E000	64 Kbytes	0x0007 . E000	256 Kbytes	0x001F . E000	1 Mbyte	0x007F . E000	4 Mbytes	0x01FF . E000	16 Mbytes	0x07FF . E000	64 Mbytes	0x1FFF . E000	256 Mbytes	R/W	Undefined
PageMask Value	Size of Each Output Page																							
0x0000 . 0000	4 Kbytes																							
0x0000 . 6000	16 Kbytes																							
0x0001 . E000	64 Kbytes																							
0x0007 . E000	256 Kbytes																							
0x001F . E000	1 Mbyte																							
0x007F . E000	4 Mbytes																							
0x01FF . E000	16 Mbytes																							
0x07FF . E000	64 Mbytes																							
0x1FFF . E000	256 Mbytes																							

7.2.8 Wired (CP0 Register 6, Select 0): Controls Number of Fixed ("wired") TLB Entries

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 7.14. 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 that are not overwritten by a TLBWR instruction. Wired entries can be overwritten by a TLBWI instruction.

The *Wired* register is reset 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.

This register is only valid with a TLB. It is reserved when the FM is implemented.

Wired can be set to a non-zero value to prevent the random replacement of that many TLB pages. It does this by preventing the *Random* register from taking values between 0 and the value of wired minus one: in turn that's done by arranging that the *Random* downcounter bounces back to its maximum value when it was previously equal to *Wired*.

Figure 7.9 Wired Register Format



Table 7.14 Field Descriptions for Wired Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>Wired</i>	5:0	The oddly-named <i>Wired</i> controls <i>Random</i> 's behavior. <i>Random</i> is implemented as a full CPU clock-rate downcounter. It won't decrement below the current value of <i>Wired</i> (when it gets there it bounces off and starts again at the highest legal index). So in practice, when used inside the TLB refill exception handler, <i>Random</i> delivers a random index into the TLB somewhere between the value of <i>Wired</i> and the top. <i>Wired</i> can therefore be set to reserve some TLB entries from random replacement — a good place for an OS to keep translations which must never cause a TLB translation-not-present exception.	R/W	0

7.2.9 HWREna (CP0 Register 7, Select 0): Bitmask Limiting User-mode Access to rdhwr Registers

The *HWREna* register contains a bit mask that determines which hardware registers are accessible via the **rdhwr** instruction when that instruction is executed in user mode.

The low-order four bits [3:0] control access to the four registers required by the MIPS32® architecture standard. The two high-order bits [31:30] are available for implementation-dependent use.

Using the *HWREna* register, privileged software may select which of the hardware registers are accessible via the RDHWR instruction. In doing so, a register may be virtualized at the cost of handling a Reserved Instruction Exception, interpreting the instruction, and returning the virtualized value. For example, if it is not desirable to provide

direct access to the *Count* register, access to that register may be individually disabled and the return value can be virtualized by the operating system.

Software may determine which registers are implemented by writing all ones to the *HWREna* register, then reading the value back. If a bit reads back as a one, the processor implements that hardware register.

Figure 7.10 HWREna Register Format



Table 7.15 Field Descriptions for HWREna Register

Name	Bit(s)	Description	Read/Write	Reset State										
<i>Impl</i>	31:30	These bits control access to implementation-dependent hardware registers. These registers are not currently implemented in any 74K family processor. Attempts to access these bits results in a Reserved Instruction Exception.	R	0										
<i>UL</i>	29	User Local Register. This register provides read access to the coprocessor 0 <i>UserLocal</i> register. Set this bit to 1 to permit user programs to obtain the value of the <i>UserLocal</i> CP0 register using rdhwr 29 .	R/W	0										
<i>CCRes</i>	3	Resolution of the CC register. This value denotes the number of cycles between update of the register. For example: <table border="1" style="margin: 10px auto;"> <thead> <tr> <th>CCRes Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>CC register increments every cycle</td> </tr> <tr> <td>2</td> <td>CC register increments every second cycle</td> </tr> <tr> <td>3</td> <td>CC register increments every third cycle</td> </tr> <tr> <td colspan="2" style="text-align: center;">etc.</td> </tr> </tbody> </table> <p>Set this bit to 1 so a user-mode rdhwr 3 can read the <i>CCRes</i> value. The <i>CCRes</i> register will read a value of 2 when the <i>Count</i> register runs at half the pipeline speed (as it always does in 74K family CPUs).</p>	CCRes Value	Meaning	1	CC register increments every cycle	2	CC register increments every second cycle	3	CC register increments every third cycle	etc.		R/W	0
CCRes Value	Meaning													
1	CC register increments every cycle													
2	CC register increments every second cycle													
3	CC register increments every third cycle													
etc.														
<i>CC</i>	2	High-resolution Cycle Counter. This register provides read access to the coprocessor 0 <i>Count</i> Register. Set this bit to 1 so a user-mode rdhwr 2 can read out the value of the <i>Count</i> register.	R/W	0										
<i>SYNCL_Step</i>	1	Address step size to be used with the SYNCI instruction. See that instruction's description for the use of this value. In the typical implementation, this value should be zero if there are no caches in the system that must be synchronized (either because there are no caches, or because the instruction cache tracks writes to the data cache). In other cases, the return value should be the smallest line size of the caches that must be synchronized. For the 74K <i>corename-lowercase</i> , the SYNCL_Step value is 32 since the line size is 32 bytes. <p>Set this bit to 1 so a user-mode rdhwr 1 can read the cache line size (actually, the smaller of the L1 I-cache line size and D-cache line size). That line size determines the step between successive uses of the synci instruction, which does the cache manipulation necessary to ensure that the CPU can correctly execute instructions which you just wrote.</p>	R/W	0										

Table 7.15 Field Descriptions for HWREna Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>CPUNum</i>	0	This register provides read access to the coprocessor 0 <i>EBaseCPUNum</i> field. Set this bit 1 so a user-mode rdhwr 0 reads out the CPU ID number.	R/W	0

7.2.10 BadVAddr (CP0 Register 8, Select 0): Address Causing the Last TLB-related Exception

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) and TLB Modified.

The *BadVAddr* register does not capture address information for cache or bus errors, since they are not addressing errors.

There is more information about this register in the notes to the *CauseExcCode* field.

Figure 7.11 BadVAddr Register Format

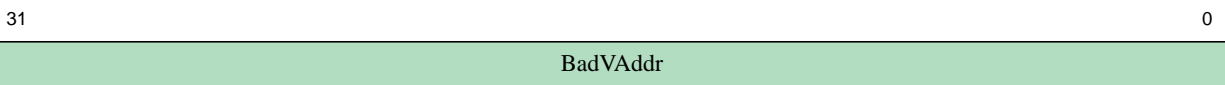


Table 7.16 BadVAddr Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bits			
Bad-VAddr	31:0	Bad virtual address.	R	Undefined

7.2.11 Count (CP0 Register 9, Select 0): Free-running Counter at Half the Pipeline Speed

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. When enabled by clearing the *DC* bit in the *Cause* register, the counter increments every other clock. By writing the *CountDM* bit in the *Debug* register, it is possible to control whether the *Count* register continues incrementing while the processor is in debug mode.

Count may stop in only two circumstances. First, some implementations may stop *Count* in the low-power mode, for example, through the **wait** instruction, but only if the *CauseDC* flag is set to 1. Secondly, you can arrange to stop *Count* in debug mode by setting *DebugCountDM*.

Count will carry on counting from whatever value is loaded into it. However, OS timers are usually implemented by leaving *Count* free-running and writing *Compare* as necessary.

Figure 7.12 Count Register Format

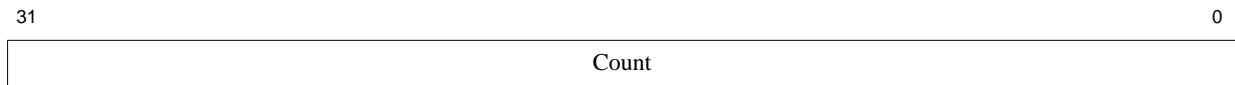


Table 7.17 Count Register Field Description

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

7.2.12 EntryHi (CP0 Register10, Select 0): High-order Portion of TLB Entry

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

A TLB exception (TLB Refill, TLB Invalid, or TLB Modified) causes bits $VA_{31..13}$ of the virtual address to be written into the *VPN2* field of the *EntryHi* register. A TLBR instruction writes the *EntryHi* register with the corresponding fields from the selected TLB entry. 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.

Because the *ASID* field is overwritten by a TLBR instruction, software must save and restore the value of *ASID* around use of the TLBR. This is especially important in TLB Invalid and TLB Modified exceptions, and in other memory management software.

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

This register is only valid with the TLB. It is reserved if the FM is implemented.

Figure 7.13 EntryHi Register Format



Table 7.18 Field Descriptions for EntryHi Register

Name	Bit(s)	Description	Read/ Write	Reset State
<i>VPN2</i>	31:13	<i>EntryHi_{VPN2}</i> is the virtual address to be matched on a tlbp . It is also the virtual address to be written into the TLB on a tlbw and the destination of the virtual address on a tlbr . On a TLB-related exception, the field <i>VPN2</i> is automatically set to the virtual address we were trying to translate when we got the exception. If — as is most often the case — the outcome of the exception handler is to find and install a translation to that address, <i>VPN2</i> (and generally the whole of <i>EntryHi</i>) will turn out to already have the right values in it. It is written by software before a tlbp or tlbw and written by hardware in all other cases.	R/W	Undefined

Table 7.18 Field Descriptions for EntryHi Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>ASID</i>	7:0	This field does double-duty. It is used to stage data to and from the TLB, but in normal running software it's also the source of the current "ASID" value, used to extend the virtual address to make sure you only get translations for the current process.	R/W	0

7.2.13 Compare (CP0 Register 11, Select 0): Timer Interrupt Control

The *Compare* register acts in conjunction with the *Count* register to implement a timer and timer interrupt function—when the value of the *Count* register equals the value of the *Compare* register, the *SI_TimerInt* output pin is asserted. *SI_TimerInt* remains asserted until the *Compare* register is written.

The *SI_TimerInt* output can be fed back into the core on one of the interrupt pins to generate an interrupt. Traditionally, this has been done by multiplexing it with hardware interrupt 5 in order to set interrupt bit *IP(7)* in the *Cause* register.

For diagnostic purposes, the *Compare* register is a read/write register. In normal use, however, the *Compare* register is write-only. As a side effect, writing a value to this register clears the timer interrupt.

Figure 7.14 Compare Register Format

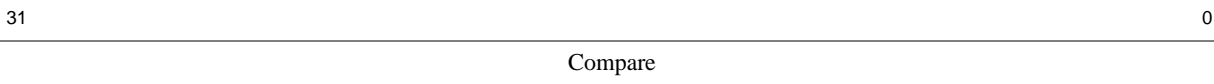


Table 7.19 Compare Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
Compare	31:0	Interval count compare value.	R/W	Undefined

7.2.14 Status (CP0 Register 12, Select 0): Processor Status and Control

The *Status* register is a read/write register that contains the operating mode, interrupt enabling, and diagnostic states of the processor. Fields in this register combine to create operating modes for the processor. Refer to [Section 5.2 “Modes of Operation”](#) for a discussion of operating modes, and to [Section 6.3 “Interrupts”](#) for a discussion of interrupt modes. A brief summary is provided below.

7.2.14.1 Interruptibility

Interrupts are enabled when:

Status_{IE} == 1, *Status_{EXL}* == 0, *Status_{ERL}* == 0, and *Debug_{DM}* == 0.

When these conditions are met, individual interrupts can be disabled/enabled using the *Status_{IM7-0}* mask bits.

7.2.14.2 Operating Modes

User Mode

The CPU is in user mode when:

$Status_{EXL} == 0$, $Status_{ERL} == 0$, $Debug_{DM} == 0$, and $Status_{UM,SM} == 2$.

In user mode, the CPU has access only to the mapped kuseg address region.

Refer to [Section 5.2.2 “User Mode”](#).

Supervisor Mode

The CPU is in supervisor mode when:

$Status_{EXL} == 0$, $Status_{ERL} == 0$, $Debug_{DM} == 0$, and $Status_{UM,SM} == 1$.

In supervisor mode, the CPU has access to the top half of the kseg2 region (sometimes known as kseg3), but no access to CP0 registers or most kernel memory. Supervisor mode is not compatible with the "fixed mapping" MMU option. This mode is not used by any MIPS OS code.

Refer to [Section 5.2.3 “Supervisor Mode”](#).

Kernel Mode

In kernel mode, the CPU has unrestricted access to all memory spaces (including, importantly, the "unmapped" regions kseg0 and kseg1), and to all the privileged (CP0) registers documented in this chapter, but it is unable to access some debug resources.

The CPU is in kernel mode when $Debug_{DM}$ is 0 and any of the following conditions are true:

$Status_{EXL} == 1$, $Status_{ERL} == 1$, or $Status_{UM,SM} == 0$.

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.

Refer to [Section 5.2.4 “Kernel Mode”](#).

Debug Mode

The processor is operating in Debug Mode when the DM bit in the CP0 *Debug* register is set to 1. In debug mode, the processor has full access to all resources that are available in Kernel Mode operation, in addition to those provided by EJTAG.

Refer to [Section 5.2.5 “Debug Mode”](#).

7.2.14.3 Coprocessor Accessibility

The *Status* register CU bits control coprocessor accessibility. If any coprocessor is unusable, then an instruction that accesses it generates an exception.

Figure 7.15 shows the format of the *Status* Register; Table 7.20 describes the *Status* register fields.

Figure 7.15 Status Register Format

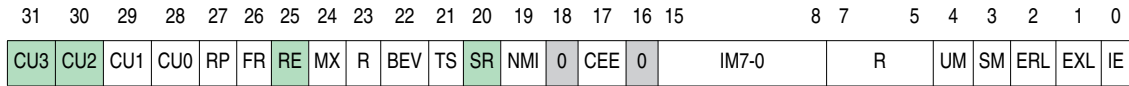


Table 7.20 Field Descriptions for Status Register

Name	Bit(s)	Description	Read/Write	Reset State						
<i>CU3</i>	31	Coprocessor 3 Usable. Because no 74K family core has a coprocessor 3, <i>Status_{CU3}</i> is hardwired zero..	R	0						
<i>CU2</i>	30	<p>Coprocessor 2 Usable. Controls access to coprocessor 2:</p> <table border="1" style="margin-left: 20px; border-collapse: collapse;"> <thead> <tr> <th style="width: 15%;">Encoding</th> <th style="width: 85%;">Meaning</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">0</td> <td>Access not allowed</td> </tr> <tr> <td style="text-align: center;">1</td> <td>Access allowed</td> </tr> </tbody> </table> <p><i>CU2</i> is reserved for a customer’s coprocessor. Currently the 74K family of cores does not support Coprocessor 2, so this bit is read-only and reads zero.</p>	Encoding	Meaning	0	Access not allowed	1	Access allowed	R	0
Encoding	Meaning									
0	Access not allowed									
1	Access allowed									
<i>CU1</i>	29	<p>Coprocessor 1 Usable. Controls access to coprocessor 1:</p> <table border="1" style="margin-left: 20px; border-collapse: collapse;"> <thead> <tr> <th style="width: 15%;">Encoding</th> <th style="width: 85%;">Meaning</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">0</td> <td>Access not allowed</td> </tr> <tr> <td style="text-align: center;">1</td> <td>Access allowed</td> </tr> </tbody> </table> <p><i>CU1</i> is most often used for a floating-point unit. When no coprocessor 1 is present, this bit is read-only and reads zero.</p>	Encoding	Meaning	0	Access not allowed	1	Access allowed	R/W	Undefined
Encoding	Meaning									
0	Access not allowed									
1	Access allowed									
<i>CU0</i>	28	<p>Coprocessor 0 Usable. Controls access to coprocessor 0 :</p> <table border="1" style="margin-left: 20px; border-collapse: collapse;"> <thead> <tr> <th style="width: 15%;">Encoding</th> <th style="width: 85%;">Meaning</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">0</td> <td>Access not allowed</td> </tr> <tr> <td style="text-align: center;">1</td> <td>Access allowed</td> </tr> </tbody> </table> <p>Coprocessor 0 is always usable when the processor is running in Kernel or Debug Mode, regardless of the state of the <i>CU0</i> bit. Setting <i>Status_{CU0}</i> to 1 has the peculiar effect of allowing privileged instructions to execute in user mode, though this is not something a secure OS is likely to allow.</p>	Encoding	Meaning	0	Access not allowed	1	Access allowed	R/W	Undefined
Encoding	Meaning									
0	Access not allowed									
1	Access allowed									
<i>MX</i>	24	<p>MIPS Extension. Enables access to DSP ASE resources:</p> <table border="1" style="margin-left: 20px; border-collapse: collapse;"> <thead> <tr> <th style="width: 15%;">Encoding</th> <th style="width: 85%;">Meaning</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">0</td> <td>Access not allowed</td> </tr> <tr> <td style="text-align: center;">1</td> <td>Access allowed</td> </tr> </tbody> </table> <p>An attempt to execute any DSP ASE instruction before this bit has been set to 1 will cause a DSP State Disabled exception. The state of this bit is reflected in <i>Config3_{DSP}</i>.</p>	Encoding	Meaning	0	Access not allowed	1	Access allowed	R/W	0
Encoding	Meaning									
0	Access not allowed									
1	Access allowed									

Table 7.20 Field Descriptions for Status Register (Continued)

Name	Bit(s)	Description	Read/Write	Reset State						
<i>CEE</i>	17	<p>CorExtend Enable. Enable/disable CorExtend User Defined Instructions (UDIs):</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Disable CorExtend block</td> </tr> <tr> <td>1</td> <td>Enable CorExtend block</td> </tr> </tbody> </table> <p>This signal's usage by a CorExtend block is implementation-dependent. The presence of the CorExtend extension is indicated in <i>ConfigUDI</i>, which is set when the core is configured. This bit is reserved if CorExtend is not present</p>	Encoding	Meaning	0	Disable CorExtend block	1	Enable CorExtend block	R/W	Undefined
Encoding	Meaning									
0	Disable CorExtend block									
1	Enable CorExtend block									
<i>RP</i>	27	<p>Reduced Power. Enable/disable reduced power mode:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Disable reduced power mode</td> </tr> <tr> <td>1</td> <td>Enable reduced power mode</td> </tr> </tbody> </table> <p>The state of the <i>RP</i> bit is visible on the core's external interface signal <i>SL_RP</i>. The 74K core uses clocks that are generated outside the core, and this could be used in your design to slow the input clock(s).</p>	Encoding	Meaning	0	Disable reduced power mode	1	Enable reduced power mode	R/W	0
Encoding	Meaning									
0	Disable reduced power mode									
1	Enable reduced power mode									
<i>FR</i>	26	<p>Floating Register. This bit is used to control the floating-point register mode for 64-bit floating point units:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Floating point registers can contain any 32-bit datatype. 64-bit data types are stored in even-odd pairs of registers</td> </tr> <tr> <td>1</td> <td>Floating point registers can contain any datatype</td> </tr> </tbody> </table> <p>This bit must be ignored on write and read as zero under the following conditions</p> <ul style="list-style-type: none"> • No floating point unit is implemented • 64-bit floating point unit is not implemented <p>If your processor has a floating point unit, set 0 for MIPS I compatibility mode, which means you have only 16 real FP registers, with 16 odd FP register numbers reserved for access to the high-order bits of double-precision values.</p>	Encoding	Meaning	0	Floating point registers can contain any 32-bit datatype. 64-bit data types are stored in even-odd pairs of registers	1	Floating point registers can contain any datatype	R/W	0
Encoding	Meaning									
0	Floating point registers can contain any 32-bit datatype. 64-bit data types are stored in even-odd pairs of registers									
1	Floating point registers can contain any datatype									
<i>RE</i>	25	Reverse Endian. Enables Reverse endianness for instructions that execute in User mode. This feature is not supported in the 74K core and reads 0.	R	0						
R	23	Reserved. Returns zeros on reads.	R	0						
<i>BEV</i>	22	<p>Boot Exception Vector. Controls the location of exception vectors:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Normal</td> </tr> <tr> <td>1</td> <td>Bootstrap</td> </tr> </tbody> </table> <p>When set to 1, all exception entry points are relocated to near the reset start address. Refer to Section 6.5 "Exception Vector Locations".</p>	Encoding	Meaning	0	Normal	1	Bootstrap	R/W	1
Encoding	Meaning									
0	Normal									
1	Bootstrap									

Table 7.20 Field Descriptions for Status Register (Continued)

Name	Bit(s)	Description	Read/Write	Reset State												
<i>TS</i>	21	TLB Shutdown. Set if software attempts to create a duplicate TLB entry (which will also produce a "machine check" exception). Can be written back to zero, but never written to 1. The name of the field originated as a "TLB Shutdown"—historical MIPS CPUs quietly stopped translating addresses when they detected TLB abuse.	R/W	0												
<i>SR</i>	20	Soft Reset. The 74K core's interface only supports a full external reset, so this bit always reads zero.	R	0												
<i>NMI</i>	19	Indicates that the entry through the reset exception vector was due to an NMI: <table border="1" style="margin: 10px auto;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Not NMI (Reset)</td> </tr> <tr> <td>1</td> <td>NMI</td> </tr> </tbody> </table> Software can only write a 0 to this bit to clear it and cannot force a 0 to 1 transition.	Encoding	Meaning	0	Not NMI (Reset)	1	NMI	R/W0	1 for NMI, 0 otherwise						
Encoding	Meaning															
0	Not NMI (Reset)															
1	NMI															
<i>IM7-0</i>	15:8	Interrupt Mask. Bitwise interrupt enables for the eight interrupt conditions. The state of these bits is visible in <i>CauseIP7-0</i> , except in EIC Mode, which is activated when <i>Config3VEIC</i> reads 1, you set <i>CauseIVx</i> and write a non-zero "vector spacing" in <i>IntCtlVS</i> . In EIC mode, <i>IM7-2</i> is recycled to become a 6-bit <i>StatusIPL</i> (Interrupt Priority Level) field. An interrupt is only triggered when the interrupt controller presents an interrupt code which is numerically higher than the current value of <i>StatusIPL</i> . <i>StatusIM1-0</i> always act as bitwise masks for the two software interrupt bits programmable in <i>CauseIP1-0</i> .	R/W	Undefined												
R	7:5	Reserved. Returns zeros on reads.	R	0												
<i>UM</i>	4	These bits denote the processor's operating mode. See Section 5.2 "Modes of Operation" .	R/W	Undefined												
<i>SM</i>	3		R/W	Undefined												
		<table border="1" style="margin: 10px auto;"> <thead> <tr> <th>UM</th> <th>SM</th> <th>Mode</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>Kernel</td> </tr> <tr> <td>0</td> <td>1</td> <td>Supervisor</td> </tr> <tr> <td>1</td> <td>0</td> <td>User</td> </tr> </tbody> </table> Note that the processor can also be in Kernel mode if <i>ERL</i> or <i>EXL</i> is set, regardless of the state of these bits.	UM	SM	Mode	0	0	Kernel	0	1	Supervisor	1	0	User		
UM	SM	Mode														
0	0	Kernel														
0	1	Supervisor														
1	0	User														

Table 7.20 Field Descriptions for Status Register (Continued)

Name	Bit(s)	Description	Read/Write	Reset State						
<i>ERL</i>	2	<p>Error Level; Set by the processor when a Reset, Soft Reset, NMI, or Cache Error exception is taken.</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Normal level</td> </tr> <tr> <td>1</td> <td>Error level</td> </tr> </tbody> </table> <p>When <i>ERL</i> is set:</p> <ul style="list-style-type: none"> The processor is running in kernel mode Interrupts are disabled The ERET instruction will use the return address held in <i>ErrorEPC</i> instead of <i>EPC</i> The lower 2^{29} bytes of kuseg are treated as an unmapped and uncached region. See Chapter 5, “Memory Management of the 74K™ Core”. This allows main memory to be accessed for cache errors. The operation of the processor is UNDEFINED if the ERL bit is set while the processor is executing instructions from kuseg. 	Encoding	Meaning	0	Normal level	1	Error level	R/W	1
Encoding	Meaning									
0	Normal level									
1	Error level									
<i>EXL</i>	1	<p>Exception Level; Set by the processor when any exception other than Reset, Soft Reset, Cache Error, or NMI exception is taken.</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Normal level</td> </tr> <tr> <td>1</td> <td>Exception level</td> </tr> </tbody> </table> <p>When EXL is set:</p> <ul style="list-style-type: none"> The processor is running in Kernel Mode Hardware and software interrupts are disabled. TLB Refill exceptions use the general exception vector instead of the TLB Refill vector. EPC, Cause_D and SRSCtl are not be updated if another exception is taken. <p>When an exception occurs and <i>EXL</i> is set, a nested TLB Refill exception is sent to the general exception handler (rather than to its dedicated handler) and the values in <i>EPC</i>, <i>Cause_{BD}</i> and <i>SRSCtl</i> are not overwritten. The result is that when you return from the second exception, you skip straight back to the code that was executing before the first exception occurred.</p>	Encoding	Meaning	0	Normal level	1	Exception level	R/W	Undefined
Encoding	Meaning									
0	Normal level									
1	Exception level									
<i>IE</i>	0	<p>Interrupt Enable, Acts as the master enable for software and hardware interrupts:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Interrupts are disabled</td> </tr> <tr> <td>1</td> <td>Interrupts are enabled</td> </tr> </tbody> </table> <p>This bit can be written using the di/ei instructions.</p>	Encoding	Meaning	0	Interrupts are disabled	1	Interrupts are enabled	R/W	Undefined
Encoding	Meaning									
0	Interrupts are disabled									
1	Interrupts are enabled									

7.2.15 IntCtl (CP0 Register 12, Select 1): Setup for Interrupt Vector and Interrupt Priority Features

The *IntCtl* register controls the interrupt capabilities of the 74K core, including vectored interrupts and support for an external interrupt controller.

Figure 7.16 IntCtl Register Format

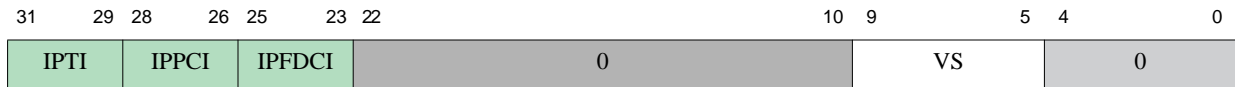


Table 7.21 Field Descriptions for IntCtl Register

Name	Bit(s)	Description	Read/Write	Reset State																					
<i>IPTI</i>	31:29	<p>For Interrupt Compatibility and Vectored Interrupt modes, this field specifies the IP number to which the Timer Interrupt request is merged, and allows software to determine whether to consider <i>CauseTI</i> for a potential interrupt.</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>IP bit</th> <th>Hardware Interrupt Source</th> </tr> </thead> <tbody> <tr> <td>2</td> <td>2</td> <td>HW0</td> </tr> <tr> <td>3</td> <td>3</td> <td>HW1</td> </tr> <tr> <td>4</td> <td>4</td> <td>HW2</td> </tr> <tr> <td>5</td> <td>5</td> <td>HW3</td> </tr> <tr> <td>6</td> <td>6</td> <td>HW4</td> </tr> <tr> <td>7</td> <td>7</td> <td>HW5</td> </tr> </tbody> </table> <p>The value of this bit is set by the static input, <i>SI_IPTI</i>[2:0]. This allows external logic to communicate the specific <i>SI_Int</i> hardware interrupt pin to which the <i>SI_TimerInt</i> signal is attached. The value of this field is not meaningful if External Interrupt Controller Mode is enabled. The external interrupt controller is expected to provide this information for that interrupt mode.</p>	Encoding	IP bit	Hardware Interrupt Source	2	2	HW0	3	3	HW1	4	4	HW2	5	5	HW3	6	6	HW4	7	7	HW5	R	Externally Set
Encoding	IP bit	Hardware Interrupt Source																							
2	2	HW0																							
3	3	HW1																							
4	4	HW2																							
5	5	HW3																							
6	6	HW4																							
7	7	HW5																							
<i>IPPCI</i>	28:26	<p>For Interrupt Compatibility and Vectored Interrupt modes, this field specifies the IP number to which the Performance Counter Interrupt request is merged, and allows software to determine whether to consider <i>CausePCI</i> for a potential interrupt.</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>IP bit</th> <th>Hardware Interrupt Source</th> </tr> </thead> <tbody> <tr> <td>2</td> <td>2</td> <td>HW0</td> </tr> <tr> <td>3</td> <td>3</td> <td>HW1</td> </tr> <tr> <td>4</td> <td>4</td> <td>HW2</td> </tr> <tr> <td>5</td> <td>5</td> <td>HW3</td> </tr> <tr> <td>6</td> <td>6</td> <td>HW4</td> </tr> <tr> <td>7</td> <td>7</td> <td>HW5</td> </tr> </tbody> </table> <p>The value of this bit is set by the static input <i>SI_IPPCI</i>[2:0]. This allows external logic to communicate the specific <i>SI_Int</i> hardware interrupt pin to which the <i>SI_PCInt</i> signal is attached. The value of this field is not meaningful if External Interrupt Controller Mode is enabled. The external interrupt controller is expected to provide this information for that interrupt mode.</p>	Encoding	IP bit	Hardware Interrupt Source	2	2	HW0	3	3	HW1	4	4	HW2	5	5	HW3	6	6	HW4	7	7	HW5	R	Externally Set
Encoding	IP bit	Hardware Interrupt Source																							
2	2	HW0																							
3	3	HW1																							
4	4	HW2																							
5	5	HW3																							
6	6	HW4																							
7	7	HW5																							

Table 7.21 Field Descriptions for IntCtl Register

Name	Bit(s)	Description	Read/Write	Reset State																					
<i>IPFDCI</i>	25:23	<p>For Interrupt Compatibility and Vectored Interrupt modes, this field specifies the IP number to which the Fast Debug Channel Interrupt request is merged, and allows software to determine whether to consider <i>CauseFDCI</i> for a potential interrupt.</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>IP bit</th> <th>Hardware Interrupt Source</th> </tr> </thead> <tbody> <tr> <td>2</td> <td>2</td> <td>HW0</td> </tr> <tr> <td>3</td> <td>3</td> <td>HW1</td> </tr> <tr> <td>4</td> <td>4</td> <td>HW2</td> </tr> <tr> <td>5</td> <td>5</td> <td>HW3</td> </tr> <tr> <td>6</td> <td>6</td> <td>HW4</td> </tr> <tr> <td>7</td> <td>7</td> <td>HW5</td> </tr> </tbody> </table> <p>The value of this bit is set by the static input, <i>SI_IPFDCI</i>[2:0]. This allows external logic to communicate the specific <i>SI_Int</i> hardware interrupt pin to which the <i>SI_FDCInt</i> signal is attached.</p> <p>The value of this field is not meaningful if External Interrupt Controller Mode is enabled. The external interrupt controller is expected to provide this information for that interrupt mode.</p>	Encoding	IP bit	Hardware Interrupt Source	2	2	HW0	3	3	HW1	4	4	HW2	5	5	HW3	6	6	HW4	7	7	HW5	R	Externally Set
Encoding	IP bit	Hardware Interrupt Source																							
2	2	HW0																							
3	3	HW1																							
4	4	HW2																							
5	5	HW3																							
6	6	HW4																							
7	7	HW5																							
<i>VS</i>	9:5	<p>If vectored interrupts are implemented (as denoted by <i>Config3VInt</i> or <i>Config3VEIC</i>), this field specifies the spacing between vectored interrupts.</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Spacing Between Vectors (hex)</th> <th>Spacing Between Vectors (decimal)</th> </tr> </thead> <tbody> <tr> <td>16#00</td> <td>16#000</td> <td>0</td> </tr> <tr> <td>16#01</td> <td>16#020</td> <td>32</td> </tr> <tr> <td>16#02</td> <td>16#040</td> <td>64</td> </tr> <tr> <td>16#04</td> <td>16#080</td> <td>128</td> </tr> <tr> <td>16#08</td> <td>16#100</td> <td>256</td> </tr> <tr> <td>16#10</td> <td>16#200</td> <td>512</td> </tr> </tbody> </table> <p>All other values are reserved. The operation of the processor is UNDEFINED if a reserved value is written to this field.</p> <p>Vector Spacing. Is writable to give you software control of the vector spacing; if the value in <i>VS</i> is VS, you will get a spacing of $32 \times 2^{(VS-1)}$ bytes.</p> <p>Only values of 1, 2, 4, 8 and 16 are permitted, for spacings of 32, 64, 128, 256, and 512 bytes respectively.</p>	Encoding	Spacing Between Vectors (hex)	Spacing Between Vectors (decimal)	16#00	16#000	0	16#01	16#020	32	16#02	16#040	64	16#04	16#080	128	16#08	16#100	256	16#10	16#200	512	R/W	0
Encoding	Spacing Between Vectors (hex)	Spacing Between Vectors (decimal)																							
16#00	16#000	0																							
16#01	16#020	32																							
16#02	16#040	64																							
16#04	16#080	128																							
16#08	16#100	256																							
16#10	16#200	512																							

7.2.16 SRSCtl (CP0 Register12, Select 2): Shadow Register Set Selectors

The *SRSCtl* register controls the operation of GPR shadow sets in the processor. Refer to [Section 6.4 “GPR Shadow Registers”](#).

The presence and number of shadow sets is configurable by the SoC designer. If your CPU has shadow register sets, *SRSCtl_{HSS}* will be non-zero. If no shadow sets are implemented, a read of this register returns all zeroes.

Figure 7.17 SRSCtl Register Format

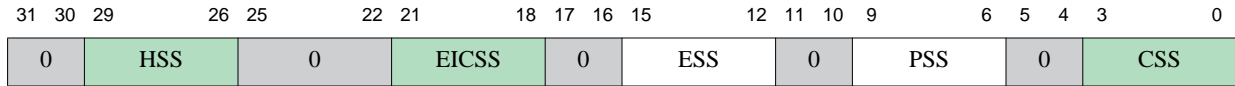


Table 7.22 Field Descriptions for SRSCtl Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>HSS</i>	29:26	Highest Shadow Set. This field contains the highest shadow set number that is implemented by this processor (i.e., the number of available register sets minus 1). A value of zero in this field indicates that only the regular GPRs are implemented. Possible values of this field for the 74K processor are: The value in this field also represents the highest value that can be written to the <i>ESS</i> , <i>EICSS</i> , <i>PSS</i> , and <i>CSS</i> fields of this register, or to any of the fields of the <i>SRSMap</i> register. The operation of the processor is UNDEFINED if a value larger than the one in this field is written to any of those other fields	R	Preset
<i>EICSS</i>	21:18	External Interrupt Controller Shadow Set. If <i>Config3VEIC</i> is 1 (EIC interrupt mode is enabled), this field is loaded from the external interrupt controller for each interrupt request and is used in place of the <i>SRSMap</i> register to select the current shadow set for the interrupt. See Section 6.3.1.3 “External Interrupt Controller Mode” for a discussion of EIC interrupt mode. If <i>Config3VEIC</i> is 0, this field returns zero on reads.	R	Undefined
<i>ESS</i>	15:12	Exception Shadow Set. This field specifies the shadow set to use on entry to Kernel Mode caused by any exception other than a vectored interrupt. The operation of the processor is UNDEFINED if software writes a value in this field that is greater than the value in the <i>HSS</i> field.	R/W	0
<i>PSS</i>	9:6	Previous Shadow Set. If GPR shadow registers are implemented, with the exclusions noted in the next paragraph, this field is copied from the <i>CSS</i> field when an exception or interrupt occurs. An eret instruction copies this value back into the <i>CSS</i> field if <i>StatusBEV</i> = 0. This field is not updated on any exception which sets <i>StatusERL</i> to 1 (i.e., Reset, Soft Reset, NMI, cache error), an entry into EJTAG Debug mode, or any exception or interrupt that occurs with <i>StatusEXL</i> = 1 or <i>StatusBEV</i> = 1, or <i>StatusERL</i> = 1. The operation of the processor is UNDEFINED if software writes a value into this field that is greater than the value in the <i>HSS</i> field.	R/W	0

Table 7.22 Field Descriptions for SRSCtl Register (Continued)

Name	Bit(s)	Description	Read/Write	Reset State
CSS	3:0	<p>Current Shadow Set. If GPR shadow registers are implemented, this field is the number of the current GPR set. With the exclusions noted in the next paragraph, this field is updated with a new value on any interrupt or exception, and restored from the <i>PSS</i> field on an eret. Table 7.23 describes the various sources from which the <i>CSS</i> field is updated on an exception or interrupt.</p> <p>This field is not updated on any exception which sets <i>StatusERL</i> to 1 (i.e., Reset, Soft Reset, NMI, cache error), an entry into EJTAG Debug mode, or any exception or interrupt that occurs with <i>StatusEXL</i> = 1, or <i>StatusBEV</i> = 1. Neither is it updated on an eret with <i>StatusERL</i> = 1 or <i>StatusBEV</i> = 1. This field is not updated on an exception that occurs while <i>StatusERL</i> = 1.</p> <p>The value of <i>CSS</i> can be changed directly by software only by writing the <i>PSS</i> field and executing an eret instruction.</p>	R	0

Table 7.23 Sources for SRSCtl_{CSS} on an Exception or Interrupt

Exception Type	Condition	SRSCtl _{CSS} Source	Comment
Exception	All	<i>SRSCtl_{ESS}</i>	
Non-Vectored Interrupt	<i>Cause_{IV}</i> = 0	<i>SRSCtl_{ESS}</i>	Treat as exception
Vectored Interrupt	<i>Cause_{IV}</i> = 1 and <i>Config3_{VEIC}</i> = 0 and <i>Config3_{VInt}</i> = 1	<i>SRSSMap_{VECTNUM}</i>	Source is internal map register. (for <i>VECTNUM</i> see Table 6.4).
Vectored EIC Interrupt	<i>Cause_{IV}</i> = 1 and <i>Config3_{VEIC}</i> = 1	<i>SRSCtl_{EICSS}</i>	Source is external interrupt controller.

7.2.17 SRSSMap (CP0 Register 12, Select 3): Shadow Set Choice for Each Interrupt Level in VI Mode

The *SRSSMap* register specifies the mapping of a vector number to a shadow register set number for use when servicing an interrupt in Vectored Interrupt (VI) mode (*Config3_{VInt}* = 1, *Config3_{VEIC}* = 0, and *Cause_{IV}* = 1). The values in this register are not used for non-interrupt exceptions or non-vectored interrupts (*Cause_{IV}* = 0 or *IntCtl_{VS}* = 0); in these cases, the shadow set number is the value in *SRSCtl_{ESS}*.

If *SRSCtl_{HSS}* is zero (indicating that no shadow sets are implemented), the result of a software read or write of this register is **UNPREDICTABLE**.

The operation of the processor is **UNDEFINED** if a value is written to any field in this register that is greater than the value of *SRSCtl_{HSS}*.

The same shadow set number can be used for multiple interrupt vectors, creating a many-to-one mapping from a vector to a single shadow register set number. In Interrupt Compatibility mode, one shadow set can be used for all exception handlers, including interrupt handlers, by setting *SRSCtl_{ESS}* to a non-zero value.

CP0 Registers of the 74K™ Core

In EIC interrupt mode, this register has no effect, and the shadow set number to be used is determined by an input bus from the external interrupt controller.

Figure 7.18 SRSSMap Register Format

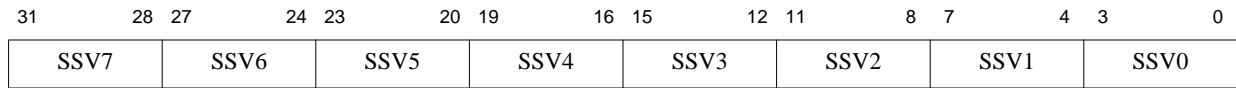


Table 7.24 Field Descriptions for SRSSMap Register

Name	Bit(s)	Description	Read/Write	Reset State
SSV7	31:28	Shadow register set number for Vector Number 7	R/W	0
SSV6	27:24	Shadow register set number for Vector Number 6	R/W	0
SSV5	23:20	Shadow register set number for Vector Number 5	R/W	0
SSV4	19:16	Shadow register set number for Vector Number 4	R/W	0
SSV3	15:12	Shadow register set number for Vector Number 3	R/W	0
SSV2	11:8	Shadow register set number for Vector Number 2	R/W	0
SSV1	7:4	Shadow register set number for Vector Number 1	R/W	0
SSV0	3:0	Shadow register set number for Vector Number 0	R/W	0

7.2.18 Cause (CP0 Register 13, Select 0): Cause of Last General Exception

The *Cause* register describes the cause of the most recent exception and controls software interrupt requests and the vector through which interrupts are dispatched. With the exception of the *IP1..0*, *DC*, *IV*, and *WP* fields, all fields in the *Cause* register are read-only. *IP7..2* are interpreted as the Requested Interrupt Priority Level (RIPL) in External Interrupt Controller (EIC) interrupt mode.

Figure 7.19 Cause Register Format

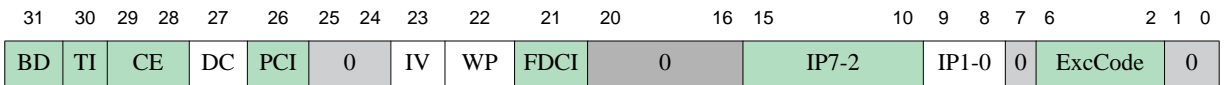


Table 7.25 Field Descriptions for Cause Register

Name	Bit(s)	Description	Read/Write	Reset State						
<i>BD</i>	31	<p>Indicates whether the last exception taken occurred in a branch delay slot:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Not in delay slot</td> </tr> <tr> <td>1</td> <td>In delay slot</td> </tr> </tbody> </table> <p>The processor updates <i>BD</i> only if the <i>EXL</i> bit in the <i>Status</i> register was zero when the exception occurred.</p> <p>If the exception occurred in a branch delay slot, <i>EPC</i> is set to restart execution at the branch, which is usually the correct thing to do. You need to consult <i>CAUSEBD</i> only when you need to look at the instruction which caused the exception (perhaps to emulate it).</p>	Encoding	Meaning	0	Not in delay slot	1	In delay slot	R	Undefined
Encoding	Meaning									
0	Not in delay slot									
1	In delay slot									
<i>TI</i>	30	<p>Timer Interrupt. Denotes whether a timer interrupt is pending (analogous to the IP bits for other interrupt types):</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No timer interrupt is pending</td> </tr> <tr> <td>1</td> <td>Timer interrupt is pending</td> </tr> </tbody> </table> <p>The state of this bit is available on the external Core interface as the <i>SI_TimerInt</i> signal.</p> <p>See also the descriptions of the <i>Count</i> and <i>Compare</i> registers.</p>	Encoding	Meaning	0	No timer interrupt is pending	1	Timer interrupt is pending	R	Undefined
Encoding	Meaning									
0	No timer interrupt is pending									
1	Timer interrupt is pending									
<i>CE</i>	29:28	<p>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 Coprocessor Unusable.</p>	R	Undefined						
<i>DC</i>	27	<p>Disable <i>Count</i> register. In some power-sensitive applications, the <i>Count</i> register is not used but may still be the source of some noticeable power dissipation. This bit allows the <i>Count</i> register to be stopped in such situations, for example, during low-power operation following a wait instruction.</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Enable counting of <i>Count</i> register</td> </tr> <tr> <td>1</td> <td>Disable counting of <i>Count</i> register</td> </tr> </tbody> </table>	Encoding	Meaning	0	Enable counting of <i>Count</i> register	1	Disable counting of <i>Count</i> register	R/W	0
Encoding	Meaning									
0	Enable counting of <i>Count</i> register									
1	Disable counting of <i>Count</i> register									

Table 7.25 Field Descriptions for Cause Register (Continued)

Name	Bit(s)	Description	Read/Write	Reset State						
<i>PCI</i>	26	<p>Performance Counter Interrupt. Indicates whether a performance counter interrupt is pending (analogous to the <i>IP</i> bits for other interrupt types):</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No performance counter interrupt is pending</td> </tr> <tr> <td>1</td> <td>Performance counter interrupt is pending</td> </tr> </tbody> </table> <p>See also the description of the <i>PerfCnt</i> registers.</p>	Encoding	Meaning	0	No performance counter interrupt is pending	1	Performance counter interrupt is pending	R	Undefined
Encoding	Meaning									
0	No performance counter interrupt is pending									
1	Performance counter interrupt is pending									
<i>IV</i>	23	<p>Indicates whether an interrupt exception uses the general exception vector or a special interrupt vector:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Use the general exception vector (0x180)</td> </tr> <tr> <td>1</td> <td>Use the special interrupt vector (0x200)</td> </tr> </tbody> </table> <p>When the <i>IV</i> bit in the <i>Cause</i> register is 1 and the <i>BEV</i> bit in the <i>Status</i> register is 0, the special interrupt vector represents the base of the vector interrupt table.</p>	Encoding	Meaning	0	Use the general exception vector (0x180)	1	Use the special interrupt vector (0x200)	R/W	Undefined
Encoding	Meaning									
0	Use the general exception vector (0x180)									
1	Use the special interrupt vector (0x200)									
<i>WP</i>	22	<p>Indicates that a watch exception was deferred because <i>StatusEXL</i> or <i>StatusERL</i> was 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 when <i>StatusEXL</i> and <i>StatusERL</i> are both zero. As such, software must clear this bit as part of the watch exception handler to prevent a watch exception loop.</p> <p>Software should not write a 1 to this bit when its value is a 0, thereby causing a 0-to-1 transition. If such a transition is caused by software, it is UNPREDICTABLE whether hardware ignores the write, accepts the write with no side effects, or accepts the write and initiates a watch exception once <i>StatusEXL</i> and <i>StatusERL</i> are both zero.</p>	R/W	Undefined						
<i>FDCI</i>	21	<p>Fast Debug Channel Interrupt: This bit denotes whether an FDC interrupt is pending (analogous to the <i>IP</i> bits for other interrupt types):</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No FDC interrupt is pending</td> </tr> <tr> <td>1</td> <td>FDC interrupt is pending</td> </tr> </tbody> </table> <p>The state of the <i>FDCI</i> bit is available on the external core interface as the <i>SI_FDCInt</i> signal.</p>	Encoding	Meaning	0	No FDC interrupt is pending	1	FDC interrupt is pending	R	Undefined
Encoding	Meaning									
0	No FDC interrupt is pending									
1	FDC interrupt is pending									

Table 7.25 Field Descriptions for Cause Register (Continued)

Name	Bit(s)	Description	Read/Write	Reset State																					
<i>IP7-2</i>	15:10	<p>Indicates an interrupt is pending:</p> <table border="1"> <thead> <tr> <th>Bit</th> <th>Name</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>15</td> <td>IP7</td> <td>Hardware interrupt 5</td> </tr> <tr> <td>14</td> <td>IP6</td> <td>Hardware interrupt 4</td> </tr> <tr> <td>13</td> <td>IP5</td> <td>Hardware interrupt 3</td> </tr> <tr> <td>12</td> <td>IP4</td> <td>Hardware interrupt 2</td> </tr> <tr> <td>11</td> <td>IP3</td> <td>Hardware interrupt 1</td> </tr> <tr> <td>10</td> <td>IP2</td> <td>Hardware interrupt 0</td> </tr> </tbody> </table> <p>If EIC interrupt mode is not enabled ($Config3_{VEIC} = 0$), timer interrupts are combined in a system-dependent way with any hardware interrupt. If EIC interrupt mode is enabled ($Config3_{VEIC} = 1$), these bits take on a different meaning and are interpreted as the <i>R IPL</i> field, described below. See Section 6.3 “Interrupts” for a general description of interrupt processing.</p>	Bit	Name	Meaning	15	IP7	Hardware interrupt 5	14	IP6	Hardware interrupt 4	13	IP5	Hardware interrupt 3	12	IP4	Hardware interrupt 2	11	IP3	Hardware interrupt 1	10	IP2	Hardware interrupt 0	R	Undefined
Bit	Name	Meaning																							
15	IP7	Hardware interrupt 5																							
14	IP6	Hardware interrupt 4																							
13	IP5	Hardware interrupt 3																							
12	IP4	Hardware interrupt 2																							
11	IP3	Hardware interrupt 1																							
10	IP2	Hardware interrupt 0																							
<i>IP1-0</i>	9:8	<p>Controls the request for software interrupts:</p> <table border="1"> <thead> <tr> <th>Bit</th> <th>Name</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>9</td> <td>IP1</td> <td>Request software interrupt 1</td> </tr> <tr> <td>8</td> <td>IP0</td> <td>Request software interrupt 0</td> </tr> </tbody> </table> <p>These bits are exported to an external interrupt controller for prioritization in EIC interrupt mode with other interrupt sources. The state of these bits is available on the external core interface as the <i>SL_SWInt[1:0]</i> bus.</p>	Bit	Name	Meaning	9	IP1	Request software interrupt 1	8	IP0	Request software interrupt 0	R/W	Undefined												
Bit	Name	Meaning																							
9	IP1	Request software interrupt 1																							
8	IP0	Request software interrupt 0																							
<i>ExcCode</i>	6:2	Encodes the cause of the last exception as described in Table 7.26.	R	Undefined																					

Table 7.26 Exception Code values in ExcCode Field of Cause Register

Value	Code	What just happened?
0	Int	Interrupt
1	Mod	Store, but page marked as read-only in the TLB
2	TLBL	Load or fetch, but page marked as invalid in the TLB
3	TLBS	Store, but page marked as invalid in the TLB
4	AdEL	Address error on load/fetch or store respectively. Address is either wrongly aligned, or a privilege violation.
5	AdES	
6	IBE	Bus error signaled on instruction fetch
7	DBE	Bus error signaled on load/store (imprecise)
8	Sys	System call, i.e. syscall instruction executed.
9	Bp	Breakpoint, i.e. break instruction executed.
10	RI	Instruction code not recognized (or not legal)
11	CpU	Instruction code was for a co-processor which is not enabled in <i>Status_{CU3-0}</i> .

Table 7.26 Exception Code values in ExcCode Field of Cause Register

Value	Code	What just happened?
12	Ov	Overflow from a trapping variant of integer arithmetic instructions.
13	Tr	Condition met on one of the conditional trap instructions teq etc.
14	-	Reserved
15	FPE	Floating point unit exception — more details in the FPU control/status registers.
16-17	-	Available for implementation-dependent use.
19-22	-	Reserved
23	WATCH	Instruction or data reference matched a watchpoint.
24	MCheck	"Machine check"
25	Thread	Thread-related exception, only for CPUs supporting the MIPS MT ASE. In that case the cause is further detailed in <i>VPEControlEXCPT</i> .
26	DSP	Tried to run an instruction from the MIPS DSP ASE, but it's either not enabled or not available. In particular, <i>StatusMX</i> is zero).
27-29	-	Reserved
30	CacheErr	Parity/ECC error somewhere in the core, on either instruction fetch, load or cache refill. In fact you never see this value in <i>CauseExcCode</i> ; but some of the codes in this table including this one can be visible in the "debug mode" of the EJTAG debug unit — see and in particular the notes on the <i>Debug</i> register.
31	-	Reserved

7.2.19 EPC (CP0 Register 14, Select 0): Restart Address from Exception

Following an exception other than a debug or error or debug exception, the *Exception Program Counter (EPC)* contains the address at which processing resumes after the exception has been serviced. (The corresponding debug and error exception use *DEPC* and *ErrorEPC* respectively.)

Unless the *EXL* bit in the *Status* register is set (indicating, among other things, that interrupts are disabled), the processor writes the *EPC* register when an exception occurs.

- 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 reads the *EPC* register as the result of execution of the **eret** instruction.

Figure 7.20 EPC Register Format

31

0



Table 7.27 EPC Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
EPC	31:0	Exception Program Counter.	R/W	Undefined

7.2.20 PRId (CP0 Register 15, Select 0): Processor Identification and Revision

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 7.21 PRId Register Format

31

24 23

16 15

8 7

0

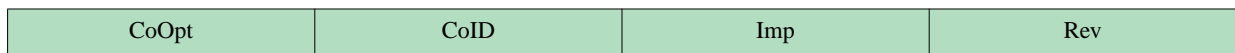


Table 7.28 Field Descriptions for PRId Register

Name	Bit(s)	Description	Read/ Write	Reset State												
<i>CoOpt</i>	31:24	Company Option. Whatever is specified by the SoC builder who synthesizes the core— refer to your SoC manual. It should be a number between 0 and 127— higher values are reserved by MIPS Technologies.	R	Preset												
<i>CoID</i>	23:16	Company ID. Identifies the company that designed or manufactured the processor. In the 74K, this field contains a value of 1 to indicate MIPS Technologies, Inc.	R	1												
<i>Imp</i>	15:8	Processor ID. Identifies the type of processor. This field allows software to distinguish between the various types of processors from MIPS Technologies. The value of this field is 0x97 for the 74K core.	R	0x97												
<i>Rev</i>	7:0	The revision number of the core design. <table border="1" style="margin: 10px auto; width: 80%;"> <thead> <tr> <th>Bit(s)</th> <th>Name</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>7:5</td> <td>Major Revision</td> <td>This number is increased on major revisions of the processor core</td> </tr> <tr> <td>4:2</td> <td>Minor Revision</td> <td>This number is increased on each incremental revision of the processor and reset on each new major revision</td> </tr> <tr> <td>1:0</td> <td>Patch Level</td> <td>If a patch is made to modify an older revision of the processor, this field will be incremented</td> </tr> </tbody> </table>	Bit(s)	Name	Meaning	7:5	Major Revision	This number is increased on major revisions of the processor core	4:2	Minor Revision	This number is increased on each incremental revision of the processor and reset on each new major revision	1:0	Patch Level	If a patch is made to modify an older revision of the processor, this field will be incremented	R	Preset
Bit(s)	Name	Meaning														
7:5	Major Revision	This number is increased on major revisions of the processor core														
4:2	Minor Revision	This number is increased on each incremental revision of the processor and reset on each new major revision														
1:0	Patch Level	If a patch is made to modify an older revision of the processor, this field will be incremented														

7.2.21 EBase (CP0 Register 15, Select 1): Exception entry point base address and CPU/VPE ID

The *EBase* register is a read/write register containing the base address of the exception vectors used when *StatusBEV* equals 0, and a read-only CPU number value that may be used by software to distinguish different processors in a multi-processor system.

The *EBase* register provides the ability for software to identify the specific processor within a multi-processor system, and allows the exception vectors for each processor to be different, especially in systems composed of heterogeneous processors. Bits 31:12 of the *EBase* register are concatenated with zeros to form the base of the exception vectors when *StatusBEV* is 0. The exception vector base address comes from the fixed defaults (see Section 6.5 “Exception Vector Locations”) when *StatusBEV* is 1, or for any EJTAG Debug exception. The reset state of bits 31:12 of the *EBase* register initialize the exception base register to 16#8000.0000, providing backward compatibility with Release 1 implementations.

Bits 31:30 of the *EBase* Register are fixed with the value 2#10 to force the exception base address to be in the kseg0 or kseg1 unmapped virtual address segments. Bit 29 of exception base address will be forced to 1 on Cache Error exceptions so the exception handler will be executed from the uncached kseg1 segment.

If the value of the exception base register is to be changed, this must be done with *StatusBEV* equal 1. The operation of the processor is **UNDEFINED** if the Exception Base field is written with a different value when *StatusBEV* is 0.

Combining bits 31:12 with the *Exception Base* field allows the base address of the exception vectors to be placed at any 4Kbbyte page boundary.

Figure 7.22 EBase Register Format

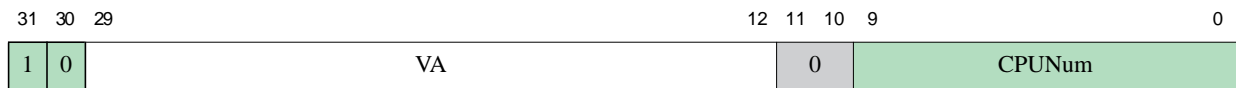


Table 7.29 Field Descriptions for EBase Register

Name	Bit(s)	Description	Read/Write	Reset State
VA	29:12	The base address for the exception vectors, adjustable to a resolution of 4Kbytes. See the <i>exception entry points table</i> for how that moves all the exception entry points. The top two bits of this register must be 10 to make sure the exception vector ends up in kseg0, conventionally used for OS code. By setting <i>EBase</i> in any CPU to a unique value, that CPU can have its own unique exception handlers. Write this field only when <i>StatusBEV</i> is set so that any exception will be handled through the ROM entry points (otherwise you would be changing the exception address under your own feet, and the results of that are undefined).	R/W	0x0000.0
CPUNum	9:0	This field contains an identifier that will be unique among the CPUs in a multi-processor system. The value in this field is set by the <i>SI_CPUNum[9:0]</i> static input pins to the core.	R	Externally Set

7.2.22 CDMMBase Register (CP0 Register 15, Select 2)

The 36-bit physical base address for the Common Device Memory Map facility is defined by this register. This register only exists if *Config3_{CDMM}* is set to one.

Figure 7.23 shows the format of the *CDMMBase* register, and Table 7.30 describes the register fields.

Figure 7.23 CDMMBase Register



Table 7.30 CDMMBase Register Field Descriptions

Fields		Description	Read / Write	Reset State												
Name	Bits															
<i>CDMM_UPPER_ADDR</i>	27:11	Bits 31:15 of the base physical address of the common device memory-mapped registers.	R/W	Undefined												
<i>EN</i>	10	Enables the CDMM region. If this bit is cleared, memory requests to this address region go to regular system memory. If this bit is set, memory requests to this region go to the CDMM logic	R/W	0												
		<table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>CDMM region is disabled.</td> </tr> <tr> <td>1</td> <td>CDMM region is enabled.</td> </tr> </tbody> </table>	Encoding	Meaning	0	CDMM region is disabled.	1	CDMM region is enabled.								
Encoding	Meaning															
0	CDMM region is disabled.															
1	CDMM region is enabled.															
<i>CI</i>	9	If set to 1, this indicates that the first 64-byte Device Register Block of the CDMM is reserved for additional registers which manage CDMM region behavior and are not IO device registers. This feature is not implemented and this field will read as 0.	R	0												
<i>CDMMSize</i>	8:0	This field represents the number of 64-byte Device Register Blocks instantiated in the core.	R	2												
		<table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>1 DRB</td> </tr> <tr> <td>1</td> <td>2 DRBs</td> </tr> <tr> <td>2</td> <td>3 DRBs</td> </tr> <tr> <td>...</td> <td>...</td> </tr> <tr> <td>511</td> <td>512 DRBs</td> </tr> </tbody> </table>	Encoding	Meaning	0	1 DRB	1	2 DRBs	2	3 DRBs	511	512 DRBs		
Encoding	Meaning															
0	1 DRB															
1	2 DRBs															
2	3 DRBs															
...	...															
511	512 DRBs															

7.2.23 Config (CP0 Register 16, Select 0): Legacy Configuration Register

The main role of the (several) configuration registers is to be a read-only repository of information about the core’s resources, encoded so as to be useful to operating system initialization code.

But this original *Config* register acquired some writable fields. Typically, these select the sort of options you’d write once in initialization software and then never touch again.

Figure 7.24 Config Register Format

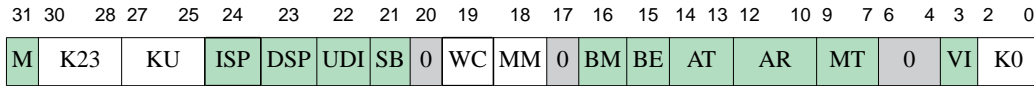


Table 7.31 Field Descriptions for Config Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>M</i>	31	Reads 1 if <i>Config1</i> is available.	R	1
<i>K23</i>	30:28	If your CPU uses fixed mapping instead of having a TLB, you set the cacheability attributes of chunks of the memory map by writing these fields. They’re encoded like <i>EntryLo0-1C</i> .	FMT: R/W TLB: R	FMT:2 TLB:0
<i>KU</i>	27:25	If you have a TLB, these fields are unused (but please write only zero to them). <i>ConfigK23</i> is for program addresses 0xC000.0000-0xFFFF.FFFF (the "kseg2" and "kseg3" areas), while <i>ConfigKU</i> is for program addresses 0x0000.0000-0x7FFF.FFFF (the "kuseg" area) From reset, both are "uncached" (code 2).	FMT: R/W TLB: R	FMT:2 TLB:0
<i>ISP</i>	24	Reads 1 if I-side scratchpad (ISPRAM) is fitted.	R	Preset
<i>DSP</i>	23	Reads 1 if D-side scratchpad (SPRAM) is fitted. (Don’t confuse this with the MIPS DSP ASE, whose presence is indicated by <i>Config3DSPP</i>)	R	Preset
<i>UDI</i>	22	Reads 1 if your core implements user-defined "CorExtend" instructions.	R	Preset
<i>SB</i>	21	Read-only "SimpleBE" bus mode indicator, which reflects the core input signal <i>SI_SimpleBE</i> . If set, means that this core will only do simple partial-word transfers on its OCP interface; that is, the only partial-word transfers will be byte, aligned half-word, and aligned word. If zero, it may generate partial-word transfers with an arbitrary set of bytes enabled (which some memory controllers may not like).	R	Externally Set
<i>WC</i>	19	Write Control. Enable write control of cache size and special function bits in the <i>Config1</i> register. 0: Write control disabled 1: Write control enabled	R/W	0
<i>MM</i>	18	Writable: set 1 if you want writes resulting from separate store instructions in write-through mode merged into a single (possibly burst) transaction at the interface. This doesn’t affect cache writebacks (which are always whole blocks together) or uncached writes (which are never merged).	R/W	1
<i>BM</i>	16	Read-only. Reads 0 when bus uses sequential burst order and reads 1 when it uses sub-block burst order; set by the core input signal <i>SI_SBlock</i> signal to match your system controller.	R	Externally Set

Table 7.31 Field Descriptions for Config Register (Continued)

Name	Bit(s)	Description	Read/Write	Reset State								
<i>BE</i>	15	Reads 1 for big-endian, 0 for little-endian, as selected by the core input <i>SI_Endian</i> .	R	Externally Set								
<i>AT</i>	14:13	Reads 0 for MIPS32. <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td> <td>MIPS32</td> </tr> <tr> <td>1</td> <td>MIPS64 instruction set but MIPS32 address map</td> </tr> <tr> <td>2</td> <td>MIPS64 instruction set with full address map</td> </tr> </table>	0	MIPS32	1	MIPS64 instruction set but MIPS32 address map	2	MIPS64 instruction set with full address map	R	0		
0	MIPS32											
1	MIPS64 instruction set but MIPS32 address map											
2	MIPS64 instruction set with full address map											
<i>AR</i>	12:10	Reads 2 to reflect Release 2 of the MIPS32 architecture. Zero is for the original release.	R	1								
<i>MT</i>	9:7	MMU type: <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td> <td>None</td> </tr> <tr> <td>1</td> <td>MIPS32/64 compliant TLB</td> </tr> <tr> <td>2</td> <td>"BAT" type</td> </tr> <tr> <td>3</td> <td>MIPS-standard fixed mapping</td> </tr> </table> <p>All MIPS Technologies cores are type 1 or 3, as selected by your SoC builder.</p>	0	None	1	MIPS32/64 compliant TLB	2	"BAT" type	3	MIPS-standard fixed mapping	R	Preset
0	None											
1	MIPS32/64 compliant TLB											
2	"BAT" type											
3	MIPS-standard fixed mapping											
<i>VI</i>	3	Reads 0 to indicate L1 I-cache is physically tagged. It would read 1 if the L1 I-cache were virtual (both indexed and tagged using virtual address).	R	0								
<i>KO</i>	2:0	Kseg0 coherency attribute of the page. See Table 7.8 for the field encoding.	R/W	2								

7.2.24 Config1-2 (CP0 Register 16, Select 1-2): MIPS32/64 Configuration Registers

These two registers tell you the size of the TLB, and the size and organization of L1, L2, and L3 caches (a zero "line size" is used to indicate a cache which isn't there).

Config2 also has fields which tell you about the presence of some extensions to the base MIPS32 architecture that are implemented on this core.

7.2.24.1 Config1

This register displays the size and configuration of the TLB and primary caches, and the availability of some optional CPU features.

Figure 7.25 Config1 Register Format

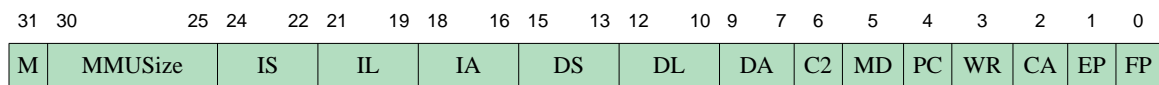


Table 7.32 Field Descriptions for Config1 Register

Name	Bit(s)	Description	Read/ Write	Reset State						
<i>M</i>	31	Continuation bit, set to 1 to indicate that <i>Config2</i> is implemented.	R	1						
<i>MMUSize</i>	30:25	The size of the TLB array (the array has <i>MMUSize</i> +1 entries).	R	Preset						
<i>IS</i>	24:22	Fields for the L1 I-cache. All caches have the same triplet of fields, which report: <table border="1" style="margin-left: 20px;"> <tr> <td>S</td> <td>Number of sets per way. Calculate as: 64×2^S</td> </tr> <tr> <td>L</td> <td>Line size. Zero means no cache at all, otherwise calculate as: 2×2^L</td> </tr> <tr> <td>A</td> <td>Associativity/number of ways. Calculate as $A + 1$</td> </tr> </table>	S	Number of sets per way. Calculate as: 64×2^S	L	Line size. Zero means no cache at all, otherwise calculate as: 2×2^L	A	Associativity/number of ways. Calculate as $A + 1$	R R/W when config.WC is set	Preset
S	Number of sets per way. Calculate as: 64×2^S									
L	Line size. Zero means no cache at all, otherwise calculate as: 2×2^L									
A	Associativity/number of ways. Calculate as $A + 1$									
<i>IL</i>	21:19		R R/W when config.WC is set	4						
<i>IA</i>	18:16	So if (<i>IS</i> , <i>IL</i> , <i>IA</i>) is (2,4,3), you have 256 sets/way, 32 bytes per line, and 4-way set associativity, which is a 32Kbyte cache. 74K family cores always have 32-byte cache lines. The L1 caches are 4-way set associative and are 16KB, 32KB, or 64KB. L1 I-cache of 0KB is supported.	R	3						
<i>DS</i>	15:13	For L1 D-cache: same encoding as <i>Config1_{IS,IL,IA}</i> . The cache line size is fixed at 32 bytes when a D-cache is present. This field reads 0 when a D-cache is not present. L1-D-cache of 0KB, 16KB, 32Kb, and 64KB are supported.	R R/W when config.WC is set	Preset						
<i>DL</i>	12:10		R R/W when config.WC is set	Preset						
<i>DA</i>	9:7		R	3						
<i>C2</i>	6	the absence of a coprocessor 2 (that would be a customer-designed coprocessor).	R	0						
<i>MD</i>	5	0 to indicate that the MDMX ASE is not implemented in the floating point unit of the 74K core	R	0						
<i>PC</i>	4	There is at least one performance counter implemented, see <i>PerfCnt0-3</i> .	R	1						
<i>WR</i>	3	Reads 1 because the 74K core always has watchpoint registers, see <i>WatchLo0-3/WatchHi0-3</i> .	R	1						
<i>CA</i>	2	Reads 1 because the MIPS16e compressed-code instruction set is available (as it is on most MIPS Technologies cores).	R	1						
<i>EP</i>	1	Reads 1 because an <i>EJTAG debug unit</i> is always provided on MIPS Technologies cores.	R	1						
<i>FP</i>	0	A floating point unit is attached.	R	Preset						

7.2.24.2 Config2

Figure 7.26 Config2 Register Format

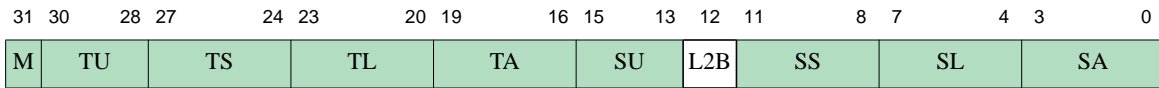


Table 7.33 Field Descriptions for Config2 Register

Name	Bit(s)	Description	Read/ Write	Reset State
<i>M</i>	31	Continuation bit, 1 if <i>Config3</i> is implemented.	R	1
<i>TU</i>	30:28	Reserved for extra control/status bits for an L3 cache, if fitted.	R	0
<i>TS</i>	27:24	If <i>Config2_{TL}</i> is non-zero, your CPU has an L3 (tertiary) cache. Its size and shape are encoded as <i>Config1_{IS,IL,IA}</i> described above. However, no 74K family core is equipped for an L3 cache.	R	0
<i>TL</i>	23:20		R	0
<i>TA</i>	19:16		R	0
<i>SU</i>	15:13	Reserved for more secondary cache control/status bits, when required. Not used on the 74K family cores.	R	0
<i>L2B</i>	12	L2_Bypass/L2_Bypassed. In systems which include an L2 cache, writing a 1 to this bit, will set the L2_Bypass output from the core. Setting the L2_Bypass output, directs the L2 cache to go into bypass mode, L2 responds by assertion its L2_Bypassed output pin. The value of L2_Bypassed is returned when L2B is read. When this bit is set through a write operation, a subsequent read of this bit will not indicate a 1, until the L2 has asserted the signal L2_Bypassed indicating that it has been bypassed.	R/W	0
<i>SS</i>	11:8	If <i>Config2_{SL}</i> is non-zero, your CPU has an external L2 (secondary) cache. Its size and shape are encoded as in <i>Config1_{IS,IL,IA}</i> above.	R R/W when config.WC is set	Preset
<i>SL</i>	7:4		R R/W when config.WC is set	Preset
<i>SA</i>	3:0		R R/W when config.WC is set	Preset

7.2.25 Config3 (CP0 Register 16, Select 3): Configuration register showing ASEs

Config3 provides information about the presence of optional extensions to the base MIPS32 architecture in addition to those specified in *Config2*.

Figure 7.27 Config3 Register Format

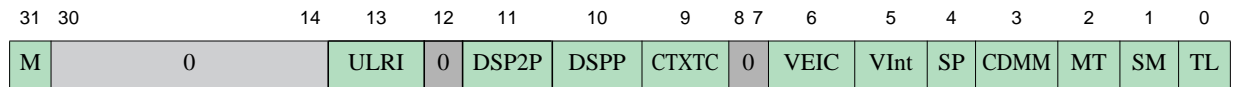


Table 7.34 Field Descriptions for Config3 Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>M</i>	31	Continuation bit, zero because there is no <i>Config4</i> .	R	0
<i>ULRI</i>	13	Reads 1 to indicate that the <i>UserLocal</i> Register is implemented	R	1
<i>DSP2P</i>	11	Reads 1 to indicate that Revision 2 of the MIPS DSP ASE is implemented	R	1
<i>DSPP</i>	10	Reads 1 to indicate that the MIPS DSP ASE extension is implemented.	R	1
<i>CTXTC</i>	9	Reads 1 to indicate the <i>ContextConfig</i> register is . The width of the <i>BadVPN2</i> field in the <i>Context</i> register depends on the contents of this register.	R	1
<i>VEIC</i>	6	Read-only bit from the core input signal <i>SI_EICPresent</i> , which should be set in the SoC to alert software to the availability of an <i>EIC</i> -compatible interrupt controller.	R	Externally Set
<i>VInt</i>	5	Reads 1 to indicate the CPU can handle vectored interrupts.	R	1
<i>SP</i>	4	Reads 0 to indicate the CPU does not support small (1Kbyte) pages.	R	0
<i>CDMM</i>	3	Reads 1 to indicate the Common Device Memory Map is implemented.	R	1
<i>MT</i>	2	Reads 0 to indicate the CPU does not include the MIPS MT (multithreading) ASE.	R	0
<i>SM</i>	1	Reads 0 to indicate the CPU does not include the instructions of the SmartMIPS ASE.	R	0
<i>TL</i>	0	Reads 1 to indicate instruction trace is supported.	R	0

7.2.26 Config6 (CP0 Register 16, Select 6)

Config3 provides information about the presence of optional extensions to the base MIPS32 architecture in addition to those specified in *Config2* and *Config3*.

Figure 7.28 Config6 Register Format

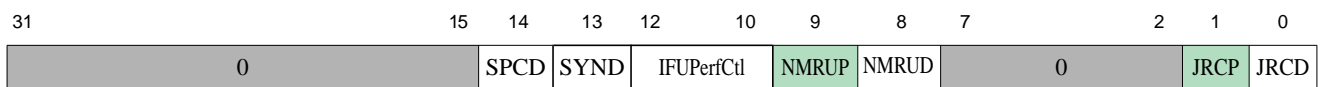


Table 7.35 Field Descriptions for Config6 Register

Name	Bit(s)	Description	Read/ Write	Reset State						
<i>SPCD</i>	14	<p>Sleep state Performance Counter Disable. When this bit is set, the performance counter core clocks are prevented from shutting down. The primary use of this bit is to keep performance counters alive when the core is in sleep mode.</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Performance counter operation is enabled in Sleep state.</td> </tr> <tr> <td>1</td> <td>Performance counter operation is disabled in Sleep state.</td> </tr> </tbody> </table>	Encoding	Meaning	0	Performance counter operation is enabled in Sleep state.	1	Performance counter operation is disabled in Sleep state.	R/W	0
Encoding	Meaning									
0	Performance counter operation is enabled in Sleep state.									
1	Performance counter operation is disabled in Sleep state.									
<i>SYND</i>	13	<p>Synonym tag update Disable. This bit controls the tag update behavior for loads with a Virtual Address miss but a Physical Address hit during a D-cache look-up.</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Synonym load misses will opportunistically update the tag so that subsequent loads will hit (virtual address hit) at lookup.</td> </tr> <tr> <td>1</td> <td>Synonym load misses at lookup will not update the tag with new information.</td> </tr> </tbody> </table>	Encoding	Meaning	0	Synonym load misses will opportunistically update the tag so that subsequent loads will hit (virtual address hit) at lookup.	1	Synonym load misses at lookup will not update the tag with new information.	R/W	0
Encoding	Meaning									
0	Synonym load misses will opportunistically update the tag so that subsequent loads will hit (virtual address hit) at lookup.									
1	Synonym load misses at lookup will not update the tag with new information.									

Table 7.35 Field Descriptions for Config6 Register (Continued)

Name	Bit(s)	Description	Read/Write	Reset State																		
<i>IFUPerfCtl</i>	12:10	<p>IFU Performance Control. This field encodes IFU events that provide debug and performance information for the IFU pipeline.</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>000</td> <td>IDU is accepting instructions, but IFU is not providing any.</td> </tr> <tr> <td>001</td> <td>A control transfer instruction such as a branch or jump causes lost IDU bandwidth.</td> </tr> <tr> <td>010</td> <td>A stalled instruction such as an unpredicted jump must wait for an address and thus causes lost IDU bandwidth.</td> </tr> <tr> <td>011</td> <td>Cache prediction was correct.</td> </tr> <tr> <td>100</td> <td>Cache prediction was incorrect.</td> </tr> <tr> <td>101</td> <td>Cache did not predict due to invalid JR cache entry, or the instruction tag miscompared with tag in JR cache.</td> </tr> <tr> <td>110</td> <td>Unimplemented.</td> </tr> <tr> <td>111</td> <td>Condition branch was taken.</td> </tr> </tbody> </table> <p>Lost IDU bandwidth occurs when the IDU is accepting instructions, but instructions are not being provided by the IFU. The count of these events can be seen via Performance Counters 0 or 3, and the event number 11. In order to view the <i>IFU PerfCtl</i> events, the Performance Counter Control needs to be programmed accordingly Table 7.49, "Performance Counter Events and Codes" for general information on event number 11.</p>	Encoding	Meaning	000	IDU is accepting instructions, but IFU is not providing any.	001	A control transfer instruction such as a branch or jump causes lost IDU bandwidth.	010	A stalled instruction such as an unpredicted jump must wait for an address and thus causes lost IDU bandwidth.	011	Cache prediction was correct.	100	Cache prediction was incorrect.	101	Cache did not predict due to invalid JR cache entry, or the instruction tag miscompared with tag in JR cache.	110	Unimplemented.	111	Condition branch was taken.	R/W	0
Encoding	Meaning																					
000	IDU is accepting instructions, but IFU is not providing any.																					
001	A control transfer instruction such as a branch or jump causes lost IDU bandwidth.																					
010	A stalled instruction such as an unpredicted jump must wait for an address and thus causes lost IDU bandwidth.																					
011	Cache prediction was correct.																					
100	Cache prediction was incorrect.																					
101	Cache did not predict due to invalid JR cache entry, or the instruction tag miscompared with tag in JR cache.																					
110	Unimplemented.																					
111	Condition branch was taken.																					
<i>NMRUP</i>	9	<p>NMRU Present. Rather than a fully random replacement on TLBWR, a table of the most recently used JTLB entries is maintained, whose entries are not replaced whenever possible.</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Most Recently Used JTLB replacement scheme not present.</td> </tr> <tr> <td>1</td> <td>Most Recently Used JTLB replacement scheme present.</td> </tr> </tbody> </table>	Encoding	Meaning	0	Most Recently Used JTLB replacement scheme not present.	1	Most Recently Used JTLB replacement scheme present.	R	1												
Encoding	Meaning																					
0	Most Recently Used JTLB replacement scheme not present.																					
1	Most Recently Used JTLB replacement scheme present.																					
<i>NMRUD</i>	8	<p>NMRU Disable. Disable bit for NMRU JTLB replacement scheme.</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>TLBWR instruction uses NMRU scheme.</td> </tr> <tr> <td>1</td> <td>TLBWR instruction uses random replacement.</td> </tr> </tbody> </table>	Encoding	Meaning	0	TLBWR instruction uses NMRU scheme.	1	TLBWR instruction uses random replacement.	R/W	0												
Encoding	Meaning																					
0	TLBWR instruction uses NMRU scheme.																					
1	TLBWR instruction uses random replacement.																					

Table 7.35 Field Descriptions for Config6 Register (Continued)

Name	Bit(s)	Description	Read/Write	Reset State						
<i>JRCP</i>	1	JR Cache Present. JR cache learns the target address of "Jump Register" type instructions and subsequently predicts that target address. <table border="1" data-bbox="500 390 1065 522"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>JR cache is not implemented.</td> </tr> <tr> <td>1</td> <td>JR cache is implemented.</td> </tr> </tbody> </table>	Encoding	Meaning	0	JR cache is not implemented.	1	JR cache is implemented.	R	1
Encoding	Meaning									
0	JR cache is not implemented.									
1	JR cache is implemented.									
<i>JRCD</i>	0	JR Cache Prediction Disable. Disables JR target address prediction.. <table border="1" data-bbox="506 590 1071 779"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>JR cache target address prediction is enabled.</td> </tr> <tr> <td>1</td> <td>JR cache target address prediction is not enabled.</td> </tr> </tbody> </table>	Encoding	Meaning	0	JR cache target address prediction is enabled.	1	JR cache target address prediction is not enabled.	R/W	0
Encoding	Meaning									
0	JR cache target address prediction is enabled.									
1	JR cache target address prediction is not enabled.									

7.2.27 Config7 (CP0 Register 16, Select 7): CPU-specific Configuration

This register controls machine-specific features of the 74K core. A few of them are for hardware interface adaptation, but most are for chip or system test only. They default into a "safe" value, and most software—even bootstrap software—can and should ignore these registers.

Figure 7.29 Config7 Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	13	12	11	10	9	8	7	6	5	4	3	2	1	0
WII	FPFS	IHB	FPR1	SEHB	CP2IO	IAGN	IALU	DGHR	SG	SUI	0	HCI	FPR0	AR	0	PREF	IAR	IVA	ES	0	CP1IO	0	ULB	BP	RPS	BHT	SL			

Table 7.36 Field Descriptions for Config7 Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>WII</i>	31	Wait IE Ignore. When this bit is set, an interrupt will unblock a wait instruction, even if <i>Status_{IE}</i> is preventing the interrupt from being taken. If <i>WII</i> reads 0, the 74K core remains in the wait condition forever if entered with interrupts disabled. If set to 1, it allows OS code to avoid tricky race conditions.	R	1
<i>FPFS</i>	30	Fast Prepare for Store. When this bit is set, pref 31 will behave as specified, i.e., the prefetch instruction will only validate the data tag but not write 0's into the data cache. By default, this bit will be 0 and pref 31 will behave like pref 30 . This means that pref 31 will validate the data tag and write 0's into the data cache array for the specified line,	R/W	0

Table 7.36 Field Descriptions for Config7 Register (Continued)

Name	Bit(s)	Description	Read/Write	Reset State
<i>IHB</i>	29	<p>If <i>IHB</i>=1, the following behavior will be true:</p> <ul style="list-style-type: none"> When the core sees any explicit/implicit mtc0(cache, ll, mtc0, tlbop, eret, deret, sync-in-debug-mode, di, ei) followed by any implicit mfc0(ehb, mfc0, eret, deret, di, ei), the pipeline will behave as if an ehb is introduced implicitly prior to executing the mfc0. This ensures all state modification by mtc0 is completely seen by mfc0. Any jalr r31, jr r31 instruction seen by the core when CP0 is usable (i.e CU0=1 or Kernel or Debug mode as defined in the PRA) will automatically treat those instructions as jalr.hb and jr.hb. <p>If <i>IHB</i>=0, the following behavior will be true:</p> <ul style="list-style-type: none"> Programmer is responsible for resolving hazards and put ehb or .hb where appropriate. Prior cores may have used some number of nops or ssnops to ensure that the effect of a CP0 modifying instruction is seen by a CP0 read instruction. 74K cannot guarantee such behavior with a small number of nops/ssnops. <p>Per Release2, the programmer is expected to put in an explicit ehb or .hb where needed. If there is reason to believe that the programmer has not done this, then this bit can be enabled to get correct operation.</p>	R/W	0
<i>SEHB</i>	27	<p>"Slow EHB": experimental mode to accelerate CP0 sequences using ehb</p> <p>If this bit is set, ehb will block issue of instructions from the instruction buffer until all older instructions have graduated and the pipe is empty. By default, ehb will block issue of instructions from the instruction buffer only if there are pending explicit CP0-modifying instructions in the pipe.</p>	R/W	0
<i>CP2IO</i>	26	<p>core</p> <p>Reserved for future use.</p>	R/W	0
<i>CP1IO</i>	6	By default data sent from the core to a coprocessor block may be sent in an order reflecting the internal pipeline execution sequence. Set this bit to arrange that data will be sent only in instruction order to the FPU	R/W	0
<i>IAGN</i>	25	Selective control of out-of-order behavior: issue ALU-side or load/store-side instructions (respectively) in program order.	R/W	0
<i>IALU</i>	24		R/W	0
<i>DGHR</i>	23	Disables the use of any global history in the branch predictor.	R/W	0
<i>SG</i>	22	Set 1 to allow only one instruction to graduate per cycle. This has a negative impact on performance and should only be used for test purposes.	R/W	0
<i>SUI</i>	21	<p>Strict Uncached Instruction (SUI) policy control.</p> <p>Run uncached instruction strictly in order and (as far as possible) unpipelined. This will be quite slow (the policy itself will introduce a 15-cycle bubble between each instruction), but you'll hardly notice, because running uncached is already so slow. Only the branch-delay-slot instruction of a branch is fetched without this bubble.</p> <p>The advantage is that the CPU will not wander off speculatively fetching unwanted instructions from a (perhaps slow) boot memory.</p>	R/W	0

Table 7.36 Field Descriptions for Config7 Register (Continued)

Name	Bit(s)	Description	Read/Write	Reset State
<i>HCI</i>	18	Hardware Cache Initialization: 1 indicates that a cache does not require initialization by software. This bit will most likely only be set on simulation-only cache models and not on real hardware.	R	Based on Hardware Present
<i>FPR1, FPR0</i>	28,17	Read-only fields. Indicates frequency of the core relative to FPU. <ul style="list-style-type: none"> • 2'b00: core:FPU = 1:1 • 2'b01: core:FPU = 2:1 • 2'b10: core:FPU = 3:2 • 2'b11: Reserved 	R	Based on Hardware Present
<i>AR</i>	16	Read-only field, indicating that the D-cache is configured to avoid <i>cache aliases</i> . All the remaining fields are read/write, and control various functions. Only one of them is likely to find real system use:	R	Based on Hardware Present
<i>PREF</i>	12:11	These two bits control the extent of prefetching of Instructions into the Instruction Cache as indicated. <ul style="list-style-type: none"> • 2'b00: Prefetch 0 cache lines on an I-cache miss in addition to fetching the missing cache line. i.e. Disable I-cache prefetching. • 2'b01: Prefetch 1 cache line (sequential next line) on an I-cache miss in addition to fetching the missing cache line. • 2'b10: Reserved • 2'b11: Prefetch 2 cache lines (sequential next 2 lines) on an I-cache miss in addition to fetching the missing cache line. 	R/W	01
<i>IAR</i>	10	Instruction Alias Removed. Indicates that this processor has hardware support to remove instruction cache aliasing. This hardware is only present when the core is configured with a TLB and cache size of 32KB and larger. The hardware is disabled via the <i>IVA</i> bit.	R	Based on Hardware Present.
<i>IVA</i>	9	Instruction Virtual Aliasing disabled. Setting this bit will disable the HW alias removal on the I-cache. If this bit is cleared, CACHE Hit Invalidate and SYNCI instructions will look up all possible aliased locations and invalidate the given cache line in all of them. This bit is Read-only if <i>IAR</i> =0.	R/W or R	0
<i>ES</i>	8	Externalize sync . If this bit is set, and if the downstream device is capable of accepting SYNCs (indicated by the pin <i>SI_SyncTxEn</i>), the sync instruction will cause a SYNC-specific transaction to go out on the external bus. If this bit is cleared or if <i>SI_SyncTxEn</i> is deasserted, no transaction will go out, but all SYNC handling internal to the core will nevertheless be performed. The sync instruction will be signalled on the core's OCP interface as an "ordering barrier" transaction. The transaction is an extension to the OCP standards, and system controllers which don't support it will typically under-decode it as a read from the boot ROM area. But that's going to be quite slow, so set this bit only if your system understands the synchronizing transaction. When this bit is read, the value returned depends on the state of the <i>SI_SyncTxEn</i> pin. If <i>SI_SyncTxEn</i> is 0, a value of 0 is returned. If <i>SI_SyncTxEn</i> is 1, the value returned is the last value that was written to this bit.	R/W	0

Table 7.36 Field Descriptions for Config7 Register (Continued)

Name	Bit(s)	Description	Read/Write	Reset State
<i>ULB</i>	4	Set to 1 to make all uncached loads blocking (a program usually only blocks when it uses the data which is loaded). You should only do this when nothing else works.	R/W	0
<i>BP</i>	3	When set, no branch prediction is done, and all branches and jump stall as above.	R/W	0
<i>RPS</i>	2	When set, the return address branch predictor is disabled, so jr \$31 is treated just like any other jump register. Instruction fetch stalls after the branch delay slot, until the jump instruction reaches the "EC" stage in the pipeline and can provide the right address.	R/W	0
<i>BHT</i>	1	When set, the branch history table is disabled and all branches are predicted taken. This bit is don't care if <i>Config7BP</i> is set.	R/W	0
<i>SL</i>	0	When set, disables non-blocking loads. Normally the 74K core will keep running after a load instruction, even if it misses in the D-cache, until the data is used. With this disable bit set, the CPU will stall on any load D-cache miss.	R/W	0

7.2.28 WatchLo0-3 (CP0 Register 18, Select 0-3): Watchpoint Address and Qualifiers

Used in conjunction with *WatchHi0-3* respectively, each of these registers carries the virtual address and what-to-match fields for a CP0 watchpoint. *WatchLo0-1* are used for instruction side accesses and *WatchLo2-3* are used for data side accesses.

Figure 7.30 WatchLo Register Format

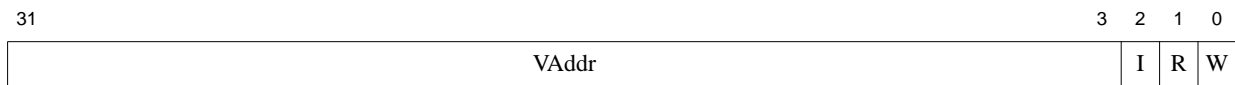


Table 7.37 Field Descriptions for WatchLo0-3 Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>VAddr</i>	31:3	The address to match on, with a resolution of a doubleword.	R/W	Undefined
<i>I</i>	2	Accesses to match: I-fetches, Reads (loads), Writes (stores). <i>WatchLo0-1_R</i> and <i>WatchLo0-1_W</i> are fixed to zero, while <i>WatchLo2-3_I</i> will be zero.	R/W	0
<i>R</i>	1		R/W	0
<i>W</i>	0		R/W	0

7.2.29 WatchHi0-3 (CP0 Register 19, Select 0-3): Watchpoint Control/Status

These registers provide the interface to a debug facility that causes an exception if an instruction or data access matches the address specified in the registers. Watch exceptions are not taken if the CPU is already in exception mode (that is if *StatusEXL* or *StatusERL* is already set).

Watch events which trigger in exception mode are remembered, and result in a "deferred" exception, taken as soon as the CPU leaves exception mode.

WatchHi0-1 are used for instruction side accesses and *WatchHi2-3* are used for data side accesses.

This CP0 watchpoint system is independent of the EJTAG debug system (which provides more sophisticated hardware breakpoints).

The *WatchLo0-3* registers hold the address to match, while *WatchHi0-3* hold a bundle of control fields.

Figure 7.31 WatchHi Register Format

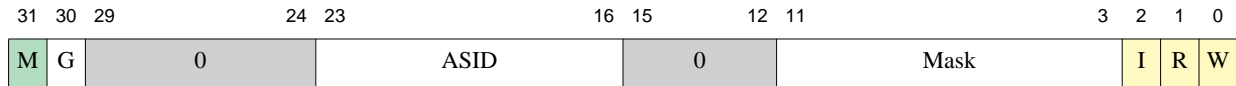


Table 7.38 Field Descriptions for WatchHi0-3 Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>M</i>	31	The <i>WatchHi0-3_M</i> bit is set whenever there is one more watchpoint register pair to find; your software should use it (starting with <i>WatchHi0</i>) to figure out how many watchpoints there are. This field is set for <i>WatchHi0-2</i> and cleared on <i>WatchHi3</i> .	R	X
<i>G</i>	30	<i>WatchHi0-3_{ASID}</i> matches addresses from a particular address space (the "ASID" is like that in TLB entries) — except that you can set <i>WatchHi0-3_G</i> ("global") to match the address in any address space.	R/W	Undefined
<i>ASID</i>	23:16		R/W	Undefined
<i>Mask</i>	11:3	Implements address ranges. Set bits in <i>WatchHi0-3_{Mask}</i> to mark corresponding <i>WatchLo0-3_{VAddr}</i> address bits to be ignored when deciding whether this is a match.	R/W	Undefined
<i>I</i>	2	Read your <i>WatchHi0-3</i> after a watch exception, and these fields tell you what type of access (if any) matched. Write a 1 to any of these bits in order to <i>clear</i> it (and therefore prevent the exception from immediately happening again). This behaviour is unusual among CP0 registers, but it is quite convenient: to clear a watchpoint of all the exception causes you've seen, just read the value of <i>WatchHi0-3</i> and write it back again. <i>WatchHi0-1_R</i> and <i>WatchHi0-1_W</i> should always read 0 and <i>WatchHi2-3_I</i> should always read 0	W1C	Undefined
<i>R</i>	1		W1C	Undefined
<i>W</i>	0		W1C	Undefined

7.2.30 Debug (CP0 Register 23, Select 0): EJTAG Debug Status/Control Register

Very little can be accessed outside of debug mode. In non-debug mode, *Debug* may not be written at all, and only the *DM* bit and the *EJTAGver* field return valid data.

The read-only information bits are updated every time the debug exception is taken, or when a normal exception is taken when already in debug mode (a "nested exception"). Not all fields are valid in both circumstances: *Halt* and *Doze* are not defined after a nested exception, and the nested-exception-type field *DExcCode* is undefined from a debug exception.

Figure 7.32 Debug Register Format

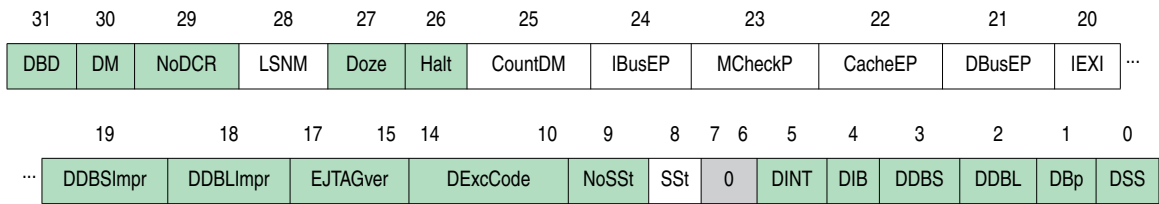


Table 7.39 Field Descriptions for Debug Register

Name	Bit(s)	Description	Read/Write	Reset State						
<i>DBD</i>	31	<p>Indicates if the last debug exception or exception in debug mode occurred in a branch delay slot:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Not in delay slot</td> </tr> <tr> <td>1</td> <td>In delay slot</td> </tr> </tbody> </table> <p>When set to 1, <i>DEPC</i> points to the branch instruction, which is usually the correct place to restart.</p>	Encoding	Description	0	Not in delay slot	1	In delay slot	R	Undefined
Encoding	Description									
0	Not in delay slot									
1	In delay slot									
<i>DM</i>	30	<p>Indicates if the processor is operating in debug mode:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Processor is operating in non-debug mode</td> </tr> <tr> <td>1</td> <td>Processor is operating in debug mode</td> </tr> </tbody> </table> <p>In debug mode, this bit is set on any debug exception and is cleared by deret.</p>	Encoding	Description	0	Processor is operating in non-debug mode	1	Processor is operating in debug mode	R	0
Encoding	Description									
0	Processor is operating in non-debug mode									
1	Processor is operating in debug mode									
<i>NoDCR</i>	29	<p>Indicates if the dseg memory segment and a memory-mapped <i>DCR</i> register is present:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>dseg is present</td> </tr> <tr> <td>1</td> <td>No dseg present</td> </tr> </tbody> </table>	Encoding	Description	0	dseg is present	1	No dseg present	R	0
Encoding	Description									
0	dseg is present									
1	No dseg present									
<i>LSNM</i>	28	<p>Controls access of load/store between dseg and main memory:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Load/stores in dseg address range goes to dseg</td> </tr> <tr> <td>1</td> <td>Load/stores in dseg address range goes to main memory</td> </tr> </tbody> </table> <p>Set this to 1 if you want debug-mode accesses to dseg addresses to be sent to system memory. This makes most of the EJTAG unit's control systems unavailable, so will probably only be done around a particular load/store.</p>	Encoding	Description	0	Load/stores in dseg address range goes to dseg	1	Load/stores in dseg address range goes to main memory	R/W	0
Encoding	Description									
0	Load/stores in dseg address range goes to dseg									
1	Load/stores in dseg address range goes to main memory									

Table 7.39 Field Descriptions for Debug Register (Continued)

Name	Bit(s)	Description	Read/Write	Reset State						
<i>Doze</i>	27	<p>Indicates that the processor was in any kind of low power mode when a debug exception occurred:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Processor not in low power mode when debug exception occurred</td> </tr> <tr> <td>1</td> <td>Processor in low power mode when debug exception occurred</td> </tr> </tbody> </table> <p>Before the debug exception, CPU was in some kind of reduced power mode.</p>	Encoding	Description	0	Processor not in low power mode when debug exception occurred	1	Processor in low power mode when debug exception occurred	R	Undefined
Encoding	Description									
0	Processor not in low power mode when debug exception occurred									
1	Processor in low power mode when debug exception occurred									
<i>Halt</i>	26	<p>Indicates that the internal system bus clock was stopped when the debug exception occurred.:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Internal system bus clock running</td> </tr> <tr> <td>1</td> <td>Internal system bus clock stopped</td> </tr> </tbody> </table> <p>Before the debug exception, the CPU was stopped — probably asleep following a wait instruction.</p>	Encoding	Description	0	Internal system bus clock running	1	Internal system bus clock stopped	R	Undefined
Encoding	Description									
0	Internal system bus clock running									
1	Internal system bus clock stopped									
<i>CountDM</i>	25	<p>Controls or indicates the Count register behavior in debug mode:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Count register stopped in debug mode</td> </tr> <tr> <td>1</td> <td>Count register is running in debug mode</td> </tr> </tbody> </table>	Encoding	Description	0	Count register stopped in debug mode	1	Count register is running in debug mode	R/W	1
Encoding	Description									
0	Count register stopped in debug mode									
1	Count register is running in debug mode									
<i>IBusEP</i>	24	<p>These "pending exception" flags remember exception events caused by instructions run in debug mode, but which have not yet occurred because they are imprecise and <i>DebugIEXI</i> is set. Note that you can write a 1 to any of these at any time, so they survive writes to the whole <i>Debug</i> register; but a write of zero to a field is ignored.</p> <p>They remain set until <i>DebugIEXI</i> is cleared explicitly, or implicitly by a deret. If the deret clears the bit, the exception is taken and the pending bit cleared.</p> <p><i>IBusEP</i> remembers a bus error on an instruction fetch. This exception is precise on the 74K core, so it can't occur, and the field is always zero.</p> <p><i>MCheckP</i> machine check condition (usually an illegal TLB update). As above, the machine check is always precise on the 74K core, so this is always zero.</p> <p><i>CacheEP</i> remembers a cache parity error.</p> <p><i>DBusEP</i> remembers a bus error on a data access.</p>	R/W	0						
<i>MCheckP</i>	23		R/W	0						
<i>CacheEP</i>	22		R/W	0						
<i>DBusEP</i>	21		R/W	0						
<i>IEXI</i>	20	<p>Set to 1 to defer imprecise exceptions. By default, this bit is set on entry to debug mode and cleared on exit. The deferred exception will return when and if this bit is cleared, and until then you can observe the occurrence of the imprecise exception in a "pending exception" flag (<i>DebugIBusEP</i><i>MCheckP</i><i>CacheEP</i><i>DBusEP</i>).</p>	R/W	0						
<i>DDBSImpr</i>	19	<p>Imprecise store breakpoint. <i>DEPC</i> probably points to an instruction some time later in the sequence than the store which triggered the breakpoint. The debugger or user (or both) have to cope as best they can.</p>	R	0						

Table 7.39 Field Descriptions for Debug Register (Continued)

Name	Bit(s)	Description	Read/Write	Reset State								
<i>DDBLImpr</i>	18	Imprecise load breakpoint. See description above of imprecise store breakpoint.	R	0								
<i>EJTAGver</i>	17:15	These read-only bits encode the revision of the EJTAG specification to which this implementation conforms. On the 74K core, the value is 3 for version 3.1. The legal values are: <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td> <td>Version 2.0 and earlier</td> </tr> <tr> <td>1</td> <td>Version 2.5</td> </tr> <tr> <td>2</td> <td>Version 2.6</td> </tr> <tr> <td>3</td> <td>Version 3.1</td> </tr> </table>	0	Version 2.0 and earlier	1	Version 2.5	2	Version 2.6	3	Version 3.1	R	3
0	Version 2.0 and earlier											
1	Version 2.5											
2	Version 2.6											
3	Version 3.1											
<i>DExcCode</i>	14:10	Indicates the cause of the latest exception in debug mode. Following initial entry to debug mode, this field is undefined. The subsequent value will be one of those defined in <i>CAUSEExcCode</i> .	R	Undefined								
<i>NoSSt</i>	9	Indicates whether the single-step feature controllable by the <i>SSt</i> bit is available in this implementation. This read-only bit is always zero on MIPS Technologies' cores because single-step is implemented.	R	0								
<i>SSt</i>	8	Controls if debug single step exception is enabled.	R/W	0								
R	7..6	Reserved. Must be written as zeros; returns zeros on reads.	R	0								
<i>DINT</i>	5	Indicates that a debug interrupt exception (from EJTAG pin) occurred. Cleared on exception in debug mode. <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Encoding</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No debug interrupt exception</td> </tr> <tr> <td>1</td> <td>Debug interrupt exception</td> </tr> </tbody> </table>	Encoding	Description	0	No debug interrupt exception	1	Debug interrupt exception	R	Undefined		
Encoding	Description											
0	No debug interrupt exception											
1	Debug interrupt exception											
<i>DIB</i>	4	Instruction breakpoint.	R	Undefined								
<i>DDBS</i>	3	Indicates that a debug data break exception occurred on a store. Cleared on exception in debug mode. <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Encoding</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No debug data exception on a store</td> </tr> <tr> <td>1</td> <td>Debug instruction exception on a store</td> </tr> </tbody> </table>	Encoding	Description	0	No debug data exception on a store	1	Debug instruction exception on a store	R	Undefined		
Encoding	Description											
0	No debug data exception on a store											
1	Debug instruction exception on a store											
<i>DDBL</i>	2	Indicates that a debug data break exception occurred on a load. Cleared on exception in debug mode. <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Encoding</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No debug data exception on a load</td> </tr> <tr> <td>1</td> <td>Debug instruction exception on a load</td> </tr> </tbody> </table>	Encoding	Description	0	No debug data exception on a load	1	Debug instruction exception on a load	R	Undefined		
Encoding	Description											
0	No debug data exception on a load											
1	Debug instruction exception on a load											

Table 7.39 Field Descriptions for Debug Register (Continued)

Name	Bit(s)	Description	Read/Write	Reset State						
<i>DBp</i>	1	Indicates that a debug software breakpoint exception occurred. Cleared on exception in debug mode. <table border="1" data-bbox="472 453 1092 569"> <thead> <tr> <th>Encoding</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No debug software breakpoint exception</td> </tr> <tr> <td>1</td> <td>Debug software breakpoint exception</td> </tr> </tbody> </table>	Encoding	Description	0	No debug software breakpoint exception	1	Debug software breakpoint exception	R	Undefined
Encoding	Description									
0	No debug software breakpoint exception									
1	Debug software breakpoint exception									
<i>DSS</i>	0	Indicates that a debug single-step exception occurred. Cleared on exception in debug mode. <table border="1" data-bbox="472 667 1092 783"> <thead> <tr> <th>Encoding</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No debug single-step exception</td> </tr> <tr> <td>1</td> <td>Debug single-step exception</td> </tr> </tbody> </table>	Encoding	Description	0	No debug single-step exception	1	Debug single-step exception	R	Undefined
Encoding	Description									
0	No debug single-step exception									
1	Debug single-step exception									

7.2.31 Trace Control Register (CP0 Register 23, Select 1)

The *TraceControl* register configuration is shown below.

Figure 7.33 TraceControl Register Format

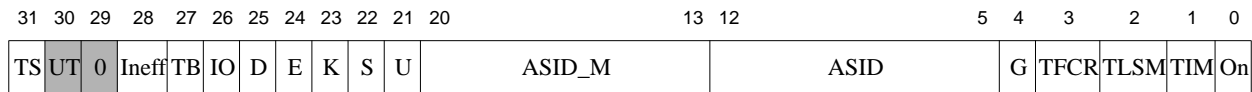


Table 7.40 TraceControl Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
<i>TS</i>	31	The trace select bit is used to select between the hardware and the software trace control bits. A value of zero selects the external hardware trace block signals, and a value of one selects the trace control bits in the <i>TraceControl</i> register.	R/W	0
<i>UT</i>	30	This bit is deprecated since there are now two explicit trace registers, <i>UserTraceData1</i> and <i>UserTraceData2</i> . Previously this bit indicated the type of user-triggered trace record. A value of zero implies a user type 1, and a value of one implies a user type 2.	0	Undefined
<i>R</i>	29	Reserved for future use; Must be written as zero; returns zero on read.	0	0
<i>Ineff</i>	28	When set to 1, core-specific inefficiency tracing is enabled, and core-specific trace information is included in the trace stream. The inefficiency code replaces an “NI” and is interpreted in the trace stream with an expanded InsComp (Instruction Completion Indicator). The InsComp is expanded from 3b to 4b for all trace formats.	R/W	0

Table 7.40 TraceControl Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
<i>TB</i>	27	Trace All Branch. When set to 1, this tells the processor to trace the PC value for all branches taken, not just the ones whose branch target address is statically unpredictable.	R/W	Undefined
<i>IO</i>	26	Inhibit Overflow. This signal is used to indicate to the core trace logic that slow but complete tracing is desired. Hence, the core tracing logic must not allow a FIFO overflow and discard trace data. This is achieved by stalling the pipeline when the FIFO is nearly full, so that no trace records are ever lost.	R/W	Undefined
<i>D</i>	25	When set to one, this enables tracing in Debug Mode. For trace to be enabled in Debug mode, the <i>On</i> bit must also be one, and either the <i>G</i> bit must be one, or the current process ASID must match the <i>ASID</i> field in this register. When set to zero, trace is disabled in Debug Mode, regardless of the setting other bits.	R/W	Undefined
<i>E</i>	24	When set to one, enables tracing in Exception Mode. For trace to be enabled in Exception mode, the <i>On</i> bit must be one, and either the <i>G</i> bit must be one, or the current process ASID must match the <i>ASID</i> field in this register. When set to zero, trace is disabled in Exception Mode, regardless of the setting of other bits.	R/W	Undefined
<i>K</i>	23	When set to one, enables tracing in Kernel Mode. For trace to be enabled in Kernel mode, the <i>On</i> bit must be one, and either the <i>G</i> bit must be one, or the current process ASID must match the <i>ASID</i> field in this register. When set to zero, trace is disabled in Kernel Mode, regardless of the setting other bits.	R/W	Undefined
<i>S</i>	22	When set to one, this enables tracing in Supervisor Mode. For trace to be enabled in Supervisor mode, the <i>On</i> bit must be one, and either the <i>G</i> bit must be one, or the current process ASID must match the <i>ASID</i> field in this register. When set to zero, trace is disabled in Supervisor Mode, regardless of other bits. If the processor does not implement Supervisor Mode, this bit is ignored on write and returns zero on read.	R/W	Undefined
<i>U</i>	21	When set to one, enables tracing in User Mode. For trace to be enabled in User mode, the <i>On</i> bit must be one, and either the <i>G</i> bit must be one, or the current process ASID must match the <i>ASID</i> field in this register. When set to zero, trace is disabled in User Mode, regardless of the setting of other bits.	R/W	Undefined
<i>ASID_M</i>	20:13	This is a mask value applied to the ASID comparison (done when the <i>G</i> bit is zero). A “1” in any bit in this field inhibits the corresponding <i>ASID</i> bit from participating in the match. As such, a value of zero in this field compares all bits of ASID. Note that the ability to mask the <i>ASID</i> value is not available in the hardware signal bit; it is only available via the software control register. If the processor does not implement the standard TLB-based MMU, this field is ignored on writes and returns zero on reads.	R/W	Undefined

Table 7.40 TraceControl Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
<i>ASID</i>	12:5	The <i>ASID</i> field to match when the <i>G</i> bit is zero. When the <i>G</i> bit is one, this field is ignored. If the processor does not implement the standard TLB-based MMU, this field is ignored on writes and returns zero on reads.	R/W	Undefined
<i>G</i>	4	When set, this implies that tracing is to be enabled for all processes, provided that other enabling functions (like <i>U</i> , <i>S</i> , etc..) are also true. If the processor does not implement the standard TLB-based MMU, this field is ignored on writes and returns 1 on reads. This causes all match equations to work correctly in the absence of an <i>ASID</i> .	R/W	Undefined
<i>TFCR</i>	3	When set, indicates to the PDtrace interface that the optional <i>Fcr</i> bit must be traced in the appropriate trace formats. If PC tracing is disabled, the full PC of the function call (or return) instruction must also be traced. Note that function call/return information is only traced if tracing is actually enabled for the current mode.	R/W	Undefined
<i>TLSM</i>	2	When set, this indicates to the PDtrace interface that information about data cache misses should be traced. If PC, load/store address, and data tracing are disabled (see the <i>TraceControl2Mode</i> field), the full PC and load/store address are traced for data cache misses. If load/store data tracing is enabled, the <i>LSm</i> bit must be traced in the appropriate trace format. Note that data cache miss information is only traced if tracing is actually enabled for the current mode.	R/W	Undefined
<i>TIM</i>	1	When set, this indicates to the PDtrace interface that the optional <i>Im</i> bit must be traced in the appropriate trace formats. If PC tracing is disabled, the full PC of the instruction that missed in the I-cache must be traced. Note that instruction cache miss information is only traced if tracing is actually enabled in the current mode.	R/W	Undefined
<i>On</i>	0	This is the master trace enable switch in software control. When zero, tracing is always disabled. When set to one, tracing is enabled whenever the other enabling functions are also true.	R/W	0

7.2.32 Trace Control2 Register (CP0 Register 23, Select 2)

The *TraceControl2* register provides additional control and status information. Note that some fields in the *TraceControl2* register are read-only, but have a reset state of “Undefined”. This is because these values are loaded from the Trace Control Block (TCB) (see [Section 11.10 “Trace Control Block \(TCB\) Registers \(Hardware Control\)”](#)). As such, these fields in the *TraceControl2* register will not have valid values until the TCB asserts these values.

This register is only implemented if the MIPS Trace capability is present.

Figure 7.34 TraceControl2 Register Format

31	30	29	28	21	20	19	12	11	7	6	5	4	3	2	1	0
SyPExt	CPUIdV	CPUId		TCV	TCNum		Mode	ValidModes	TBI	TBU	SyP					

Table 7.41 TraceControl2 Register Field Descriptions

Fields		Description	Read / Write	Reset State												
Name	Bits															
<i>SyPExt</i>	31:30	Extension to the <i>SyP</i> (sync period) field for implementations that need higher numbers of cycles between synchronization events. The value of <i>SyP</i> is extended by assuming that these two bits are juxtaposed to the left of the three bits of <i>SyP</i> (<i>SyPExtSyP</i>). When only <i>SyP</i> was used to specify the synchronization period, the value was $2x$, where x was computed from <i>SyP</i> by adding 5 to the actual value represented by the bits. A similar formula is applied to the 5 bits just obtained by the juxtaposition of <i>SyPExt</i> and <i>SyP</i> . Sync period values greater than 2^{31} are UNPREDICTABLE. That is all values greater than 11010 ($26+5=31$) are UNPREDICTABLE. With <i>SyPExt</i> bits, a sync period range of 25 to 2^{31} cycles can be obtained.	R/W	0												
<i>CPUIdV</i>	29	When set, this bit specifies the VPE defined in <i>CPUId</i> must be traced. Otherwise, instructions from all VPEs are traced when other conditions for tracing are valid. This bit is ignored if <i>TCV</i> is asserted.	R/W	0												
<i>CPUId</i>	28:21	This field specifies the number of the VPE to trace when <i>CPUIdV</i> is set.	R/W	0												
<i>TCV</i>	20	When set, the <i>TCNum</i> field specifies the number of the TC that must be traced. Otherwise, instructions from all TCs are traced when other conditions for tracing are valid.	R/W	0												
<i>TCNum</i>	19:12	Specifies the TC to trace when <i>TCV</i> is set. The right-most bits only are used.	R/W	0												
<i>Mode</i>	11:7	When tracing is turned on, this signal specifies what information is to be traced by the core. It uses 5 bits, where each bit turns on tracing of a specific tracing mode when that bit value is a 1. If the corresponding bit is 0, then the Trace Value shown in column two is not traced by the processor. The table shows what trace value is turned on: <table border="1" style="margin: 10px auto;"> <thead> <tr> <th>Bit</th> <th>Trace the Following</th> </tr> </thead> <tbody> <tr> <td>7</td> <td>PC</td> </tr> <tr> <td>8</td> <td>Load address</td> </tr> <tr> <td>9</td> <td>Store address</td> </tr> <tr> <td>10</td> <td>Load data</td> </tr> <tr> <td>11</td> <td>Store data</td> </tr> </tbody> </table>	Bit	Trace the Following	7	PC	8	Load address	9	Store address	10	Load data	11	Store data	R/W	Undefined
Bit	Trace the Following															
7	PC															
8	Load address															
9	Store address															
10	Load data															
11	Store data															
<i>ValidModes</i>	6:5	This field specifies the subset of tracing that is supported by the processor. <table border="1" style="margin: 10px auto;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>PC tracing only</td> </tr> <tr> <td>01</td> <td>PC and load and store address tracing only</td> </tr> <tr> <td>10</td> <td>PC, load and store address, and load and store data</td> </tr> <tr> <td>11</td> <td>Reserved</td> </tr> </tbody> </table>	Encoding	Meaning	00	PC tracing only	01	PC and load and store address tracing only	10	PC, load and store address, and load and store data	11	Reserved	R	Preset		
Encoding	Meaning															
00	PC tracing only															
01	PC and load and store address tracing only															
10	PC, load and store address, and load and store data															
11	Reserved															

Table 7.41 TraceControl2 Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State																		
Name	Bits																					
<i>TBI</i>	4	<p>This bit indicates how many trace buffers are implemented by the TCB, as follows:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Only one trace buffer is implemented, and the TBU bit of this register indicates which trace buffer is implemented</td> </tr> <tr> <td>1</td> <td>Both on-chip and off-chip trace buffers are implemented by the TCB and the TBU bit of this register indicates to which trace buffer the traces is currently written.</td> </tr> </tbody> </table>	Encoding	Meaning	0	Only one trace buffer is implemented, and the TBU bit of this register indicates which trace buffer is implemented	1	Both on-chip and off-chip trace buffers are implemented by the TCB and the TBU bit of this register indicates to which trace buffer the traces is currently written.	R	Undefined												
Encoding	Meaning																					
0	Only one trace buffer is implemented, and the TBU bit of this register indicates which trace buffer is implemented																					
1	Both on-chip and off-chip trace buffers are implemented by the TCB and the TBU bit of this register indicates to which trace buffer the traces is currently written.																					
<i>TBU</i>	3	<p>This bit denotes to which trace buffer the trace is currently being written and is used to select the appropriate interpretation of the <i>TraceControl2</i>_{SyP} field.</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Trace data is being sent to an on-chip trace buffer</td> </tr> <tr> <td>1</td> <td>Trace Data is being sent to an off-chip trace buffer</td> </tr> </tbody> </table> <p>This bit is loaded from <i>TCBCONTROLB</i>_{OfC}.</p>	Encoding	Meaning	0	Trace data is being sent to an on-chip trace buffer	1	Trace Data is being sent to an off-chip trace buffer	R	Undefined												
Encoding	Meaning																					
0	Trace data is being sent to an on-chip trace buffer																					
1	Trace Data is being sent to an off-chip trace buffer																					
<i>SyP</i>	2:0	<p>The period (in cycles) to which the internal synchronization counter is reset when tracing is started, or when the synchronization counter has overflowed.</p> <table border="1"> <thead> <tr> <th>SyP</th> <th>Sync Period</th> </tr> </thead> <tbody> <tr> <td>000</td> <td>2⁵</td> </tr> <tr> <td>001</td> <td>2⁶</td> </tr> <tr> <td>010</td> <td>2⁷</td> </tr> <tr> <td>011</td> <td>2⁸</td> </tr> <tr> <td>100</td> <td>2⁹</td> </tr> <tr> <td>101</td> <td>2¹⁰</td> </tr> <tr> <td>110</td> <td>2¹¹</td> </tr> <tr> <td>111</td> <td>2¹²</td> </tr> </tbody> </table> <p>This field is loaded from <i>TCBCONTROLA</i>_{SyP}.</p>	SyP	Sync Period	000	2 ⁵	001	2 ⁶	010	2 ⁷	011	2 ⁸	100	2 ⁹	101	2 ¹⁰	110	2 ¹¹	111	2 ¹²	R	Undefined
SyP	Sync Period																					
000	2 ⁵																					
001	2 ⁶																					
010	2 ⁷																					
011	2 ⁸																					
100	2 ⁹																					
101	2 ¹⁰																					
110	2 ¹¹																					
111	2 ¹²																					

7.2.33 User Trace Data1 Register (CP0 Register 23, Select 3) and User Trace Data2 Register (CP0 Register 24, Select 3)

A software write to any bits in the *UserTraceData1* register or *UserTraceData2* register will trigger a trace record to be written with a type indicator TU1 or TU2 respectively.

These register are only implemented if the MIPS Trace capability is present.

Figure 7.35 User Trace Data1 / User Trace Data2 Register Format

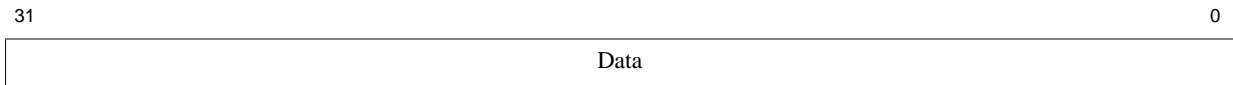


Table 7.42 UserTraceData1 / UserTraceData2 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
<i>Data</i>	31:0	Software readable/writable data. When written, this triggers a user format trace record out of the PDtrace interface that transmits the Data field to trace memory.	R/W	0

7.2.34 TracelBPC Register (CP0 Register 23, Select 4)

The *TracelBPC* register is used to control start and stop of tracing using an EJTAG Instruction Hardware breakpoint. The Instruction Hardware breakpoint would then be set as a trigger source and optionally also as a Debug exception breakpoint.

This register is only implemented if both Hardware breakpoints and the MIPS Trace capability are present.

Figure 7.36 TracelBPC Register Format



Table 7.43 TracelBPC Register Field Descriptions

Fields		Description	Read / Write	Reset State						
Name	Bits									
0	31:30, 27:12	Reserved for future implementations.	R	0						
<i>PCT</i>	29	Used to specify whether a performance counter trigger signal is generated when an EJTAG instruction breakpoint match occurs: <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 15%;">Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Disables performance counter trigger signal from instruction breakpoints</td> </tr> <tr> <td>1</td> <td>Enables performance trigger signals from instruction breakpoints</td> </tr> </tbody> </table>	Encoding	Meaning	0	Disables performance counter trigger signal from instruction breakpoints	1	Enables performance trigger signals from instruction breakpoints	R/W	0
Encoding	Meaning									
0	Disables performance counter trigger signal from instruction breakpoints									
1	Enables performance trigger signals from instruction breakpoints									
<i>IE</i>	28	Used to specify whether or not the trigger signal from EJTAG instruction breakpoint should trigger tracing functions: <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 15%;">Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Disables trigger signals from instruction breakpoints</td> </tr> <tr> <td>1</td> <td>Enables trigger signals from instruction breakpoints</td> </tr> </tbody> </table>	Encoding	Meaning	0	Disables trigger signals from instruction breakpoints	1	Enables trigger signals from instruction breakpoints	R/W	0
Encoding	Meaning									
0	Disables trigger signals from instruction breakpoints									
1	Enables trigger signals from instruction breakpoints									

Table 7.43 TraceIBPC Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
<i>IBPC_n</i>	3n+2:3n	The three bits are decoded to enable different tracing modes. Table 7.45 shows the possible interpretations. Each set of 3 bits represents the encoding for the instruction breakpoint <i>n</i> in the EJTAG implementation, if it exists. If the breakpoint does not exist, then the bits are reserved, read as zero, and writes are ignored.	R/W	0

7.2.35 TraceDBPC Register (CP0 Register 23, Select 5)

The *TraceDBPC* register is used to control start and stop of tracing using an EJTAG Data Hardware breakpoint. The Data Hardware breakpoint would then be set as a trigger source and optionally also as a Debug exception breakpoint.

This register is only implemented if both Hardware breakpoints and the MIPS Trace capability are present.

Figure 7.37 TraceDBPC Register Format

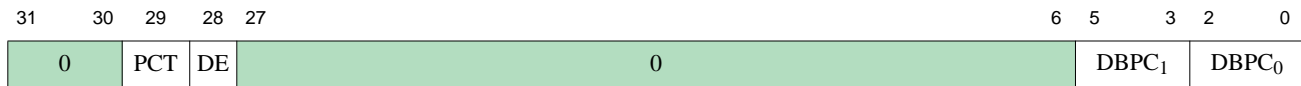


Table 7.44 TraceDBPC Register Field Descriptions

Fields		Description	Read / Write	Reset State						
Name	Bits									
0	31:30, 27:6	Reserved for future implementations.	R	0/1						
<i>PCT</i>	29	Used to specify whether a performance counter trigger signal is generated when an EJTAG data breakpoint match occurs: <table border="1" style="margin: 10px auto;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">0</td> <td>Disables performance counter trigger signal from data breakpoints</td> </tr> <tr> <td style="text-align: center;">1</td> <td>Enables performance trigger signals from data breakpoints</td> </tr> </tbody> </table>	Encoding	Meaning	0	Disables performance counter trigger signal from data breakpoints	1	Enables performance trigger signals from data breakpoints	R/W	0
Encoding	Meaning									
0	Disables performance counter trigger signal from data breakpoints									
1	Enables performance trigger signals from data breakpoints									
<i>DE</i>	28	Used to specify whether the trigger signal from EJTAG data breakpoint should trigger tracing functions: <table border="1" style="margin: 10px auto;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">0</td> <td>Disables trigger signals from data breakpoints</td> </tr> <tr> <td style="text-align: center;">1</td> <td>Enables trigger signals from data breakpoints</td> </tr> </tbody> </table>	Encoding	Meaning	0	Disables trigger signals from data breakpoints	1	Enables trigger signals from data breakpoints	R/W	0
Encoding	Meaning									
0	Disables trigger signals from data breakpoints									
1	Enables trigger signals from data breakpoints									
<i>DBPC_n</i>	3n+2:3n	The three bits are decoded to enable different tracing modes. Table 7.45 shows the possible interpretations. Each set of 3 bits represents the encoding for the data breakpoint <i>n</i> in the EJTAG implementation, if it exists. If the breakpoint does not exist then the bits are reserved, read as zero and writes are ignored.	R/W	0						

Table 7.45 BreakPoint Control Modes: IBPC and DBPC

Value	Trigger Action	Description
000	Unconditional Trace Stop	Unconditionally stop tracing if tracing was turned on. If tracing is already off, then there is no effect.
001	Unconditional Trace Start	Unconditionally start tracing if tracing was turned off. If tracing is already turned on, then there is no effect.
010	None	Reserved for future implementations.
100	Identical to trigger condition 000, and in addition, dump the full performance counter values into the trace stream	If tracing is currently on, dump the full values of all the implemented performance counters into the trace stream, and turn tracing off. If tracing is already off, then there is no effect.
101	Identical to trigger condition 001, and in addition, also dump the full performance counter values into the trace stream	Unconditionally start tracing if tracing was turned off. If tracing is already turned on, then there is no effect. In both cases, dump the full values of all the implemented performance counters into the trace stream.
110	Not used	Reserved for future implementations.

7.2.36 DEPC (CP0 Register 24, Select 0): Restart Address from Last EJTAG Debug Exception

Points to the instruction to restart when you run an **deret** to leave debug mode. When *DebugDBD* is set, it means that the "real" return address is in a branch delay slot, and *DEPC* points to the preceding branch.

Figure 7.38 DEPC Register Format



Table 7.46 DEPC Register Formats

Field		Description	Read / Write	Reset
Name	Bit(s)			
<i>DEPC</i>	31:0	The <i>DEPC</i> register is updated with the virtual address of the instruction that caused the debug exception. If the instruction is in the branch delay slot, then the virtual address of the immediately preceding branch or jump instruction is placed in this register. Execution of the DERET instruction causes a jump to the address in the <i>DEPC</i> .	R/W	Undefined

7.2.37 TraceControl3 Register (CP0 Register 24, Select 2)

The *TraceControl3* register provides additional control and status information. Note that some fields in the *TraceControl3* register are read-only, but have a reset state of “Undefined”. This is because these values are loaded from the Trace Control Block (TCB) (see Section 11.10 “Trace Control Block (TCB) Registers (Hardware Control)”). As such, these fields in the *TraceControl3* register will not have valid values until the TCB asserts these values.

This register is only implemented if the MIPS Trace capability is present.

Figure 7.39 TraceControl3 Register Format

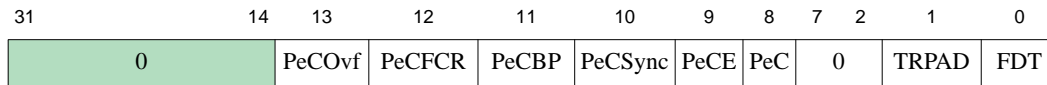


Table 7.47 TraceControl3 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:14	Reserved for future implementations.	R	0
<i>PeCOvf</i>	13	Trace performance counters when one of the performance counters overflows its count value. Enabled when set to 1.	R/W	0
<i>PeCFCR</i>	12	Trace performance counters on function call/return or on an exception handler entry. Enabled when set to 1.	R/W	0
<i>PeCBP</i>	11	Trace performance counters on hardware breakpoint match trigger. Enabled when set to 1.	R/W	0
<i>PeCSync</i>	10	Trace performance counters on synchronization counter expiration. Enabled when set to 1.	R/W	0
<i>PeCE</i>	9	Performance counter tracing enable. When set to 0, the tracing out of performance counter values as specified is disabled. To enable, this bit must be set to 1. This bit is used under software control. When trace is controlled by an external probe, this enabling is done via <i>TraceControl3_{PeCE}</i> .	R/W	0
<i>PeC</i>	8	Specifies whether or not Performance Control Tracing is implemented. This is an optional feature that may be omitted by implementation choice. Implemented when set to 1.	R/W	0
<i>TrIDLE</i>	2	Trace Unit Idle. This bit indicates if the trace hardware is currently idle (not processing any data). This can be useful when switching control of trace from hardware to software and vice versa. The bit is read-only and updated by the trace hardware.	R/W	0
<i>TRPAD</i>	1	Trace RAM Access Disable. Disables program software access to the on-chip trace RAM using load/store instructions. This bit is loaded from <i>TCBCONTROLB_{TRPAD}</i> .	R/W	0
<i>FDT</i>	0	Filtered Data Trace Mode Enable. When the bit is 0, this mode is disabled. When set to 1, this mode is enabled.	R/W	0

7.2.38 PerfCtl0-3 (CP0 Register 25, Select 0, 2, 4, 6): Performance Counter Control

Cores in the 74K family provide four performance counters that provide the capability to count events or cycles for use in performance analysis. Each performance counter consists of a pair of registers: a 32-bit control register (*PerfCtl*) and a 32-bit counter register (*PerfCnt*).

Performance counters can be configured to count implementation-dependent events or cycles under a specified set of conditions that are determined by the performance counter’s control register. The counter register increments once for each enabled event; when the most-significant bit of the counter register is a one (the counter overflows), and the counter is enabled, the performance counter optionally requests an interrupt.

The *IE* flag in the performance counter control register is used to enable an interrupt to be signalled when bit 31 of the corresponding counter overflows. The OR of all the performance counter register interrupts becomes the core output *SI_PCI*, which is typically fed back into an interrupt input, conventionally identified by *IntCtl_IPCI*. However, systems using more sophisticated interrupt controllers may feed the performance counter interrupt into the interrupt controller (refer to Section 6.3.1.3, "External Interrupt Controller Mode").

Figure 7.40 PerfCtl0-3 Register Format

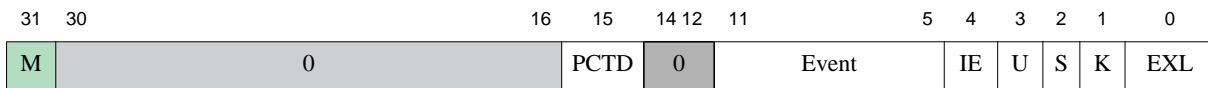


Table 7.48 Field Descriptions for PerfCtl0-3 Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>M</i>	31	Set to 1 if there is another <i>PerfCtl</i> register after this one. This field is set for <i>PerfCtl0-2</i> and cleared on <i>PerfCtl3</i> .	R	X
<i>PCTD</i>	15	Performance Counter Trace Disable. Setting this bit will prevent the tracing of data from this performance counter when performance counter trace mode in PDTrace is enabled.	R/W	Undefined
<i>Event</i>	11:5	Determines which event to count. Available events are listed in Table 7.49, "Performance Counter Events and Codes".	R/W	Undefined
<i>IE</i>	4	Set to cause an interrupt when the counter overflows into bit 31. This can either be used to implement an extended count or (by presetting the counter appropriately) to notify software after a certain number of events have occurred.	R/W	0
<i>U</i>	3	Count events in User mode, Supervisor mode, Kernel mode, and Exception mode (i.e., when <i>StatusEXL</i> is set) respectively. Set multiple bits to count in all cases.	R/W	Undefined
<i>S</i>	2		R/W	Undefined
<i>K</i>	1		R/W	Undefined
<i>EXL</i>	0		R/W	Undefined

Table 7.49 Performance Counter Events and Codes

Event No	Counter0/2	Counter1/3
0	Cycles	
1	Instructions graduated	

Table 7.49 Performance Counter Events and Codes (Continued)

Event No	Counter0/2	Counter1/3
2	jr \$31 (return) instructions whose target is predicted	jr \$31 (return) predicted but guessed wrong
3	Cycles where no instruction is fetched because it has no “next address” candidate. This includes stalls due to register indirect jumps such as jr , stalls following a wait or eret and stalls due to exceptions from instruction fetch	jr \$31 (return) instructions fetched and not predicted using RPS
4	ITLB accesses.	ITLB misses, which result in a JTLB access.
5	Reserved	JTLB instruction access misses (will lead to an exception)
6	Instruction Cache accesses. 74K cores have a 128-bit connection to the I-cache and fetch 4 instructions every access. This counts every such access, including accesses for instructions which are eventually discarded. For example, following a branch which is incorrectly predicted, the 74K core will continue to fetch instructions, which will eventually get thrown away.	Instruction cache misses. Includes misses resulting from fetch-ahead and speculation.
7	Cycles where no instruction is fetched because we missed in the I-cache.	Reserved
8	Cycles where no instruction is fetched because we are waiting for an I-fetch from uncached memory.	PDTrace back stalls
9	Number of times the instruction fetch pipeline is flushed and replayed because the IFU buffers are full and unable to accept any instructions.	Valid fetch slots killed due to taken branches/jumps or stalling instructions
10	Reserved	Reserved
11		Table 7.35, "Field Descriptions for Config6 Register"
12	•	Reserved
13	Cycles where no instructions are brought into the IDU because the ALU instruction candidate pool is full.	Cycles where no instructions are brought into the IDU because the AGEN instruction candidate pool is full.
14	Cycles where no instructions can be added to the issue pool because we have run out of ALU completion buffers (CB's).	Cycles where no instructions can be added to the issue pool because we have run out of AGEN completion buffers (CB's).
15	Cycles where no instructions can be added to the issue pool, because we have used all the FIFO entries in the CLDQ, which keep track of data coming back from the FPU.	Cycles where no instructions can be added to the issue pool, because we have filled the “in order” FIFO used for coprocessor 1 instructions (IOIQ).

Table 7.49 Performance Counter Events and Codes (Continued)

Event No	Counter0/2	Counter1/3
16	Cycles with no ALU-pipe issue: no instructions available.	Cycles with no AGEN-pipe issue: no instructions available.
17	Cycles with no ALU-pipe issue: we have instructions, but operands not ready.	Cycles with no AGEN-pipe issue: we have instructions, but operands not ready.
18	Cycles with no ALU-pipe issue: we have instructions, but some resource is unavailable. This includes: <ul style="list-style-type: none"> • Operands are not ready (same as event 17) • div in progress inhibits MDU instructions • CorExtend resource limitation. 	Cycles with no AGEN-pipe issue: we have instructions, but some resource is unavailable. This includes: <ul style="list-style-type: none"> • Operands not ready (same as event 17) • Non-issued stores blocking ready to issue loads
19	ALU-pipe bubble issued. This resulting empty pipe stage guarantees that some resource will be unused for a cycle, sometime soon. Used, for example, to guarantee an opportunity to write mfc1 data into a CB. <ul style="list-style-type: none"> • 	AGEN-pipe bubble issued. This resulting empty pipe stage guarantees that some resource will be unused for a cycle, sometime soon. Used, for example, to allow access to the data cache for refill or eviction. <ul style="list-style-type: none"> •
20	Cycles when only one instruction is issued.	Cycles when two instructions are issued (one ALU, one AGEN).
21	Cycles when instructions are issued out of order into the ALU pipe. i.e. instruction issued is not the oldest in the pool.	Cycles when instructions are issued out of order into the AGEN pipe. i.e. instruction issued is not the oldest in the pool.
22	Graduated JAR/JALR.HB	D-cache line refill (not LD/ST misses)
23	Cacheable loads - Counts all accesses to the D-cache caused by load instructions. This count includes instructions that do not graduate.	All D-cache accesses (loads, stores, prefetch, cacheop etc.). This count includes instructions that do not graduate.
24	D-cache writebacks	D-cache misses. This count is per instruction at graduation and includes load, store, prefetch, synci and address based cacheops.
25	JTLB d-side (data side as opposed to instruction side) accesses	JTLB translation fails on d-side (data side as opposed to instruction side) accesses. This count includes instructions that do not graduate.
26	Load/store instruction redirects, which happen when the load/store follows too closely on a possibly matching cacheop.	The 74K core's D-cache has an auxiliary virtual tag, used to pick the right line early. When (occasionally) the physical tag match and virtual tag match do not line up, it is treated as a cache miss - in processing the "miss" the virtual tag is corrected for future accesses. This event counts those bogus misses.
27		•
28	L2 cache writebacks	L2 cache accesses
29	L2 cache misses	L2 cache miss cycles
30	Cycles Fill Store Buffer(FSB) are full and cause a pipe stall	Cycles Fill Store Buffer(FSB) > 1/2 full
31	Cycles Load Data Queue (LDQ) are full and cause a pipe stall	Cycles Load Data Queue(LDQ) > 1/2 full

Table 7.49 Performance Counter Events and Codes (Continued)

Event No	Counter0/2	Counter1/3
32	Cycles Writeback Buffer(WBB) are full and cause a pipe stall	Cycles Writeback Buffer(WBB) > 1/2 full
33	Reserved	Reserved
34	Reserved	Reserved
35	Replays following optimistic issue of instruction dependent on load which missed. Counted only when the dependent instruction graduates.	Floating Point Load instructions graduated.
36	jr (not \$31) instructions graduated.	jr \$31 mispredicted at graduation
37	Integer Branch instructions graduated	Floating Point Branch instructions graduated
38	Branch likely instructions graduated	Mispredicted Branch likely instructions graduated
39	Conditional branches graduated	Mispredicted Conditional branches graduated
40	Integer instructions graduated (includes nop , ssnop , ehb as well as all arithmetic, logic, shift and extract type operations).	Floating Point instructions graduated (but not counting Floating Point load/store)
41	Loads graduated (includes Floating Point)	Stores graduated (includes Floating Point). Of sc instructions, only successful ones are counted.
42	j/jal graduated	MIPS16e instructions graduated
43	no-ops graduated - included (sll , nop , ssnop , ehb).	integer multiply/divides graduated
44	DSP instructions graduated	ALU-DSP instructions graduated, result was saturated
45	DSP branch instructions graduated	MDU-DSP instructions graduated, result was saturated.
46	Uncached loads graduated.	Uncached stores graduated.
47	Reserved	Reserved
48	Reserved	Reserved
49	EJTAG instruction triggers	EJTAG data triggers
50	CP1 branches mispredicted.	Reserved
51	sc instructions graduated.	sc instructions failed.
52	prefetch instructions graduated at the top of LSGB.	prefetch instructions which did nothing, because they hit in the cache.
53	Cycles where no instructions graduated	Load misses graduated. Includes Floating Point Loads.
54	Cycles where one instruction graduated	Cycles where two instructions graduated
55	GFifo blocked cycles	Floating point stores graduated

Table 7.49 Performance Counter Events and Codes (Continued)

Event No	Counter0/2	Counter1/3
56	GFifo blocked due to TLB or Cacheop	Number of cycles no instructions graduated from the time the pipe was flushed because of a replay until the first new instruction graduates. This is an indicator graduation bandwidth loss due to replay. Often times this replay is a result of event 25 and therefore an indicator of bandwidth lost due to cache miss.
57	Slot 0 mispredicted branch instruction graduation cycles without the delayslo	Cycles waiting for delayslot to graduate on a mispredicted branch
58	Exceptions taken	Replays initiated from graduation
59	Implementation-specific CorExtend event. The integrator of this core may connect the core pin <i>UDI_perfcnt_event</i> to an event to be counted. This is intended for use with the CorExtend interface.	Reserved
60	Reserved	Reserved
61	Reserved	Reserved
62		Implementation-specific DSPRAM event. The integrator of this core may connect the pin <i>SP_prf_c13_e62_xx</i> to the event to be counted
63	L2 single-bit errors detected	Reserved
64	SI_Event[0] - Implementation-specific system event. The integrator of this core may connect the core pin <i>SI_PCEvent[0]</i> to an event to be counted	SI_Event[1] - Implementation-specific system event. The integrator of this core may connect the core pin <i>SI_PCEvent[1]</i> to an event to be counted
65	SI_Event[2] - Implementation-specific system event. The integrator of this core may connect the core pin <i>SI_PCEvent[2]</i> to an event to be counted	SI_Event[3] - Implementation-specific system event. The integrator of this core may connect the core pin <i>SI_PCEvent[3]</i> to an event to be counted
66	SI_Event[4] - Implementation-specific system event. The integrator of this core may connect the core pin <i>SI_PCEvent[4]</i> to an event to be counted	SI_Event[5] - Implementation-specific system event. The integrator of this core may connect the core pin <i>SI_PCEvent[5]</i> to an event to be counted
67	SI_Event[6] - Implementation-specific system event. The integrator of this core may connect the core pin <i>SI_PCEvent[6]</i> to an event to be counted	SI_Event[7] - Implementation-specific system event. The integrator of this core may connect the core pin <i>SI_PCEvent[7]</i> to an event to be counted
68	All OCP requests accepted	All OCP cacheable requests accepted
69	OCP read requests accepted	OCP cacheable read requests accepted
70	OCP write requests accepted	OCP cacheable write requests accepted
71	Reserved	OCP write data sent
72	Reserved	OCP read data received
73	Reserved	Reserved
74	Cycles Fill Store Buffer(FSB) < 1/4 full	Cycles Fill Store Buffer(FSB) 1/4 to 1/2 full
75	Cycles Load Data Queue (LDQ) < 1/4 full	Cycles Load Data Queue (LDQ) 1/4 to 1/2 full
76	Cycles Writeback Buffer(WBB) < 1/4 full	Cycles Writeback Buffer(WBB) 1/4 to 1/2 full
77-127	Reserved	Reserved

7.2.39 PerfCnt0-3 (CP0 Register 25, Select 1, 3, 5, 7): Performance Counters

General purpose event counters, which operate as directed by *PerfCnt0-3*.

Figure 7.41 Performance Counter Count Register

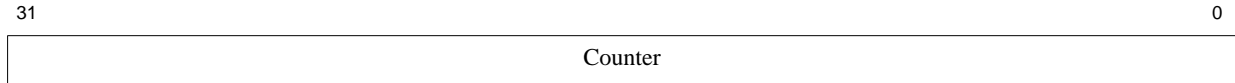
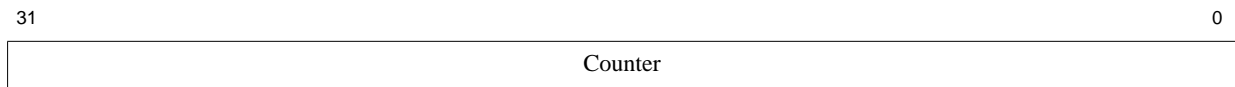


Table 7.50 Performance Counter Count Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Counter	31:0	Counter	R/W	Undefined



7.2.40 ErrCtl (CP0 Register 26, Select 0): Software Parity Control and Test Modes for Cache RAM Arrays

Most of the fields of this register are for test software only. The MIPS32 Architecture defines this register as implementation-dependent, but most CPUs put the parity-enable control in the top bit. So running OS software is well advised to set this register to 0×8000.0000 to enable cache parity checking, or to zero to disable parity checking.

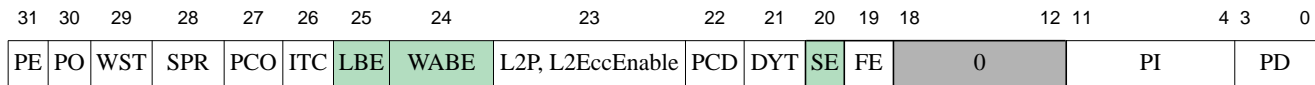


Table 7.51 Field Descriptions for ErrCtl Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>PE</i>	31	1 to enable cache parity checking. Hard-wired to zero if parity isn't implemented.	R/W	0
<i>PO</i>	30	Parity Overwrite. Set 1 to set the parity bit regardless of parity computation, which is only for diagnostic/test purposes. After setting this bit you can use cache IndexStoreTag to set the cache data parity to the value currently in <i>PI</i> (for I-cache) or <i>PD</i> (for D-cache), while the tag parity is forcefully set from <i>ITagLoP/DTagLoP</i> .	R/W	0
<i>WST</i>	29	Write to 1 for test mode for cache IndexLoadTag/cache IndexStoreTag instructions, which then read/write the cache's internal <i>way-selection RAM</i> instead of the cache tags.	R/W	0
<i>SPR</i>	28	Scratchpad RAM. When set, index-type cache instructions work on the <i>scratchpad/DSPRAM/ISPRAM</i> , if fitted.	R/W	0
<i>PCO</i>	27	Precode override. Used for diagnostic/test of the I-cache. When this bit is set, then the precode values in the <i>ITagHi</i> register are used instead of the hardware generated precode values. This applies to index store data cacheop operation.	R/W	0
<i>ITC</i>	26	Reserved	R/W	Undefined

Table 7.52 Field Descriptions for CacheErr Register (Continued)

Name	Bit(s)	Description	Read/Write	Reset State
<i>ED</i>	29	Set for errors in data field and tag field respectively.	R	Undefined
<i>ET</i>	28		R	Undefined
<i>ES</i>	27	Error source. Not Supported.	R	Undefined
<i>EE</i>	26	Error external: Not supported.	R	Undefined
<i>EB/EM</i>	25	<ul style="list-style-type: none"> If <i>EC</i> equals 0 indicating an error in the L1 cache, this bit is <i>EB</i>, indicating Error in Both caches. If data and instruction-fetch error are reported on the same instruction, it is unrecoverable. If so, the rest of the register reports on the instruction-fetch error. If <i>EC</i> equals 1 indicating an error in the L2 or higher cache, this bit is <i>EM</i>, indicating Error in Multiple locations. 	R	Undefined
<i>EF</i>	24	<p>Unrecoverable (fatal) error (other than the <i>EB</i> type above). Some parity errors can be fixed by invalidating the cache line and relying on good data from memory. But if this bit is set, all is lost... It's one of the following:</p> <p>7.52.1 Dirty parity error in dirty victim</p> <p>7.52.1 Line being displaced from cache ("victim") has a tag parity error, so we don't know whether to write it back, or whether the writeback location (which needs a correct tag) would be correct.</p> <p>7.52.2 The victim's tag indicates it has been written by the CPU since it was obtained from memory (the line is "dirty" and needs a write-back), but it has a data parity error.</p> <p>7.52.3 Writeback store miss and <i>CacheErr_{EW}</i> error.</p> <p>7.52.4 At least one more cache error happened concurrently with or after this one, but before we reached the relative safety of the cache error exception handler.</p> <p>7.52.5 If <i>EC</i> equals 0, and a second L2 error occurs when an earlier L2 error is pending.</p>	R	Undefined
<i>SP</i>	23	Error affecting a <i>scratchpad RAM</i> access.	R	Undefined
<i>EW</i>	22	Parity error on way-selection RAM array.	R	Undefined
<i>Way</i>	21:19	<ul style="list-style-type: none"> If <i>EC</i> equals 0, bit 19 is unused. Bits 21:20 indicate the way-number of the cache entry where the error occurred. It is not valid if a Scratchpad RAM error is detected (<i>SP</i>=1). If <i>EC</i> equals 1, indicating an L2 or higher-level cache error, bits 21:19 indicate the way-number of the cache entry where the error occurred. 	R	Undefined
<i>DR</i>	18	A 1 bit indicates that the reported error affected the cache-line "dirty" bits. This bit is only meaningful in case of an L1 data cache access.	R	Undefined
<i>Index</i>	16:0	<p>The cache index or Scratchpad RAM index of the double word entry where the error occurred. The way of the faulty cache is written by hardware in the Way field. The CacheErr bits [16:0] represents the Address index bits [19:3].</p> <p>The index-type cache instruction will need an "index" with the way bits glued on top of this cache-entry field; you know how to put that together, because the shape of the cache is defined in the <i>Config1-2</i> registers.</p>	R	Undefined

7.2.42 ITagLo (CP0 Register 28, Select 0): Read/write Interface for Load/Store Tag Cacheops

These registers are a staging location for cache tag information being read/written with **cache** load-tag/store-tag operations.

The interpretation of this register changes depending on the settings of *ErrCtlWST* and *ErrCtlSPR*.

- Default cache interface mode (*ErrCtlWST* = 0, *ErrCtlSPR* = 0)
- Diagnostic "way select test mode" (*ErrCtlWST* = 1, *ErrCtlSPR* = 0)
- For scratchpad memory setup (*ErrCtlWST* = 0, *ErrCtlSPR* = 1)

See the diagrams below for a description.

7.2.42.1 ITagLo (*ErrCtlWST* = 0, *ErrCtlSPR* = 0)

In this mode, this register is a staging location for cache tag information being read/written with **cache** load-tag/store-tag operations—routinely used in cache initialization.

Figure 7.43 ITagLo Register Format (*ErrCtlWST* = 0, *ErrCtlSPR* = 0)



Table 7.53 Field Descriptions for ITagLo Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>PTagLo</i>	31:12	The cache address tag, which is a physical address because the 74K core's caches are physically tagged. It holds bits 31:12 of the physical address, i.e., the low-order 12 bits of the address are implied by the position of the data in the cache.	R/W	Undefined
<i>V</i>	7	Set to 1 if this cache entry is valid (set to zero to initialize the cache).	R/W	Undefined
<i>L</i>	5	Set to 1 to lock this cache entry, preventing it from being replaced by another line when a cache miss occurs. Used when you have data so critical that it must be in the cache; however, it's quite costly, reducing the efficiency of the cache for memory data competing for space at this index.	R/W	Undefined
<i>P</i>	0	Parity bit over the cache tag entries (excluding the <i>D</i> bit).	R/W	Undefined

7.2.42.2 ITagLo-WST (*ErrCtlWST* = 1, *ErrCtlSPR* = 0)

The way-select RAM is an independent slice of the cache memory (distinct from the tag and data arrays). Test software can access the data in these fields either by **cache** load-tag or store-tag operations when *ErrCtlWST* is set.

Figure 7.44 ITagLo Register Format (*ErrCtlWST* = 1, *ErrCtlSPR* = 0)

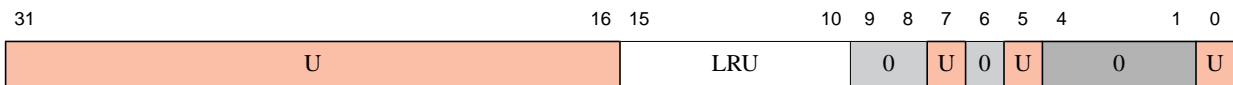


Table 7.54 Field Descriptions for ITagLo-WST Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>LRU</i>	15:10	When you read or write the tag in way-select test mode (that is, with <i>ErrCtlWST</i> set), this field reads or writes the LRU ("least recently used") state bits, held in the way-select RAM.	R/W	Undefined

7.2.42.3 ITagLo-WST (*ErrCtlWST* = 0, *ErrCtlSPR* = 1)

In this mode, the *ITagLo* register becomes the interface to the instruction scratchpad RAM.

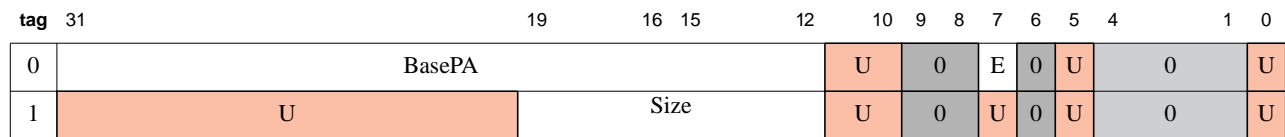
Figure 7.45 ITagLo Register Format (*ErrCtlWST* = 0, *ErrCtlSPR* = 1)

Table 7.55 Field Descriptions for ITagLo-SPR Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>BasePA</i>	31:12	When reading pseudo-tag 0 of a scratchpad RAM, this field will contain bits [31:12] of the base address of the scratchpad region	R/W	Undefined
<i>E</i>	7	When reading pseudo-tag 0 of a scratchpad RAM, this bit will indicate whether the scratchpad is enabled	R/W	Undefined
<i>Size</i>	19:12	When reading pseudo-tag 1 of a scratchpad RAM, this field indicates the size of the scratchpad array. This field is the number of 4KB sections it contains.	R/W	Undefined

7.2.43 IDataLo (CP0 Register 28, Select 1): Read/write Interface for I-cache Special Cacheops

Staging registers for special **cache** instruction which loads or stores data from or to the cache line. Two registers (*IDataHi*, *IDataLo*) are needed, because the 74K core loads I-cache data at least 64 bits at a time.

Figure 7.46 IDataLo Register Format



Table 7.56 IDataLo Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
<i>DATA</i>	31:0	Low-order data read from the cache data array.	R/W	Undefined

7.2.44 DTagLo (CP0 Register 28, Select 2): Read/Write Interface for Load/Store Tag Cacheops

These registers are a staging location for cache tag information being read/written with **cache** load-tag/store-tag operations.

The interpretation of this register changes depending on the settings of *ErrCtlWST* and *ErrCtlSPR*.

- Default cache interface mode (*ErrCtlWST* = 0, *ErrCtlDYT* = 0, *ErrCtlSPR* = 0)
- Diagnostic "way select test mode" (*ErrCtlWST* = 1, *ErrCtlDYT* = 0, *ErrCtlSPR* = 0)
- Diagnostic "dirty array test mode" (*ErrCtlWST* = 0, *ErrCtlDYT* = 1, *ErrCtlSPR* = 0)
- For scratchpad memory setup (*ErrCtlWST* = 0, *ErrCtlDYT* = 0, *ErrCtlSPR* = 1)

7.2.44.1 DTagLo (*ErrCtlWST* = 0, *ErrCtlDYT* = 0, *ErrCtlSPR* = 0)

In this mode, this register is a staging location for cache tag information being read/written with **cache** load-tag/store-tag operations—routinely used in cache initialization.

Figure 7.47 DTagLo Register Format (*ErrCtlWST* = 0, *ErrCtlDYT* = 0, *ErrCtlSPR* = 0)

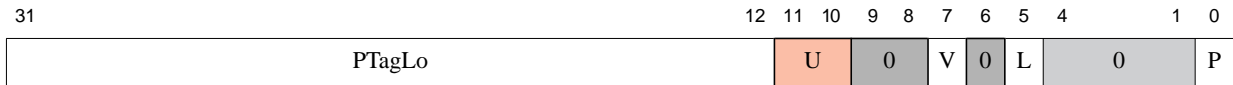


Table 7.57 Field Descriptions for DTagLo Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>PTagLo</i>	31:12	The cache address tag — a physical address because the 74K core’s caches are physically tagged. It holds bits 31–12 of the physical address — the low 12 bits of the address are implied by the position of the data in the cache.	R/W	Undefined
<i>V</i>	7	1 if this cache entry is valid (set zero to initialize the cache).	R/W	Undefined
<i>L</i>	5	1 to lock this cache entry, preventing it from being replaced by another line when there’s a cache miss. Done when you have data so critical that it must be in the cache: it’s quite costly, reducing the efficiency of the cache for memory data competing for space at this index.	R/W	Undefined
<i>P</i>	0	Parity bit over the cache tag entries (excluding the <i>D</i> bit).	R/W	Undefined

7.2.44.2 DTagLo-WST(*ErrCtlWST* = 1, *ErrCtlDYT* = 0, *ErrCtlSPR* = 0)

The way-select RAM is an independent slice of the cache memory (distinct from the tag and data arrays). Test software can access either by **cache** load-tag/store-tag operations when *ErrCtlWST* is set: then you get the data in these fields.

Figure 7.48 DTagLo Register Format ($ErrCtl_{WST} = 1$, $ErrCtl_{DYT} = 0$, $ErrCtl_{SPR} = 0$)

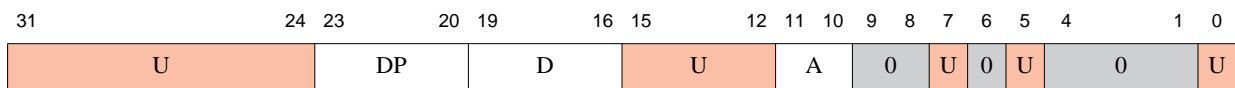
Table 7.58 Field Descriptions for DTagLo-WST Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>LP</i>	23:20	Cache-line locking control bits, held in the way select RAM.	R/W	Undefined
<i>L</i>	19:16		R/W	Undefined
<i>LRU</i>	15:10	When you read or write the tag in way select test mode (that is, with $ErrCtl_{WST}$ set) this field reads or writes the LRU ("least recently used") state bits, held in the way select RAM.	R/W	Undefined

7.2.44.3 DTagLo-DYT ($ErrCtl_{WST} = 0$, $ErrCtl_{DYT} = 1$, $ErrCtl_{SPR} = 0$)

The dirty RAM is another slice of the cache memory (distinct from the tag and data arrays). Test software can access either by **cache** load-tag/store-tag operations when $ErrCtl_{DYT}$ is set: then you get the data in these fields.

Figure 7.49 Field Descriptions for DTagLo-DYT Register



r

Table 7.59 Field Descriptions for DTagLo-DYT Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>DP</i>	23:20	Cache line "dirty" bits (and parity across them).	R/W	Undefined
<i>D</i>	19:16		R/W	Undefined
<i>A</i>	11:10	Cache line "alias" bits.	R/W	Undefined

Figure 7.50 DTagLo-SPT ($ErrCtl_{WST} = 0$, $ErrCtl_{DYT} = 0$, $ErrCtl_{SPR} = 1$)

If your CPU has *scratchpad* RAM, you will need to initialize and manage it using **cache** load/store operations while $ErrCtl_{SPR}$ is set. The tag load/store operations are used to read and write control registers: and then you see these fields.

Figure 7.51 DTagLo Register Format (ErrCtl_{WST} = 0, ErrCtl_{DYT} = 0, ErrCtl_{SPR} = 1)

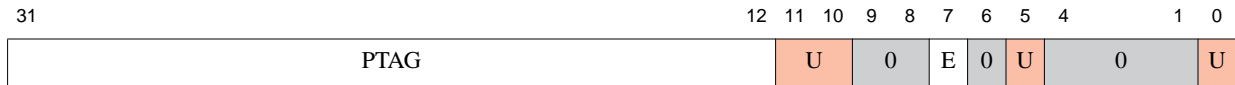


Table 7.60 Field Descriptions for DTagLo-SPT Register

Name	Bit(s)	Description	Read/Write	Reset State
<i>PTAG</i>	31:12	Scratchpad control. Sets base address.	R/W	Undefined
<i>E</i>	7	Scratchpad control enable.	R/W	Undefined

7.2.45 DDataLo (CP0 Register 28, Select 3): Low-order Data Read/Write Interface for D-cache

On 74K family cores, test software can read or write cache data using a **cache** index load tag/index store data instruction. Which word of the cache line is transferred depends on the low address fed to the **cache** instruction.

Figure 7.52 DDataLo Register Format



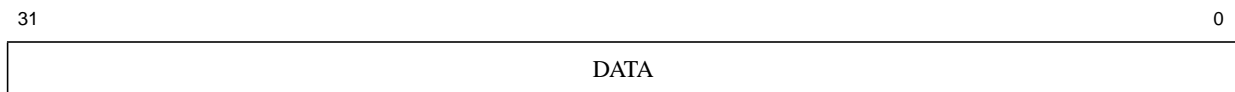
Table 7.61 DDataLo Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
<i>DATA</i>	31:0	Low-order data read from the cache data array.	R/W	Undefined

7.2.46 L23TagLo (CP0 Register 28, Select 4): L2 and L3 Cacheop Tag Use

This register in the 74K core is implemented to support access to external L2 cache tags via **cache** instructions. The definition of the fields of this 32 bit register are defined by the SoC designer. Refer to the section on L2 Transactions in the document ““MIPS32@ 74K™ Processor Core Family Integrator’s Guide, MD00499” for further information on using this register.

Figure 7.53 L23TagLo Register Format



7.2.47 L23DataLo (CP0 Register 28, Select 5): Low-order Data Read/Write Interface for L2 or L3 cache

On 74K family cores, test software can read or write cache data using a **cache** index load/store data instruction. Which word of the cache line is transferred depends on the low address fed to the **cache** instruction.

Figure 7.54 L23DataLo Register Format



Table 7.62 L23DataLo Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
<i>DATA</i>	31:0	Low-order data read from the cache data array.	R/W	Undefined

7.2.48 ITagHi (CP0 Register 29, Select 0): I-cache Predecode Bits

This register represents the I-cache Predecode bits and is intended for diagnostic use only

Figure 7.55 ITagHi Register Format

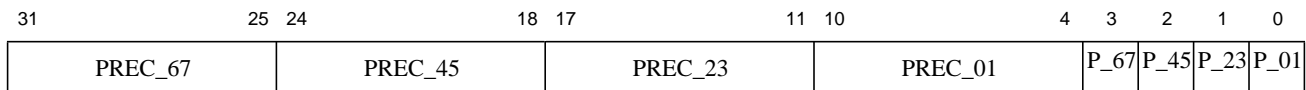


Table 7.63 Field Descriptions for ITagHi Register

Name	Bit(s)	Description	Read/ Write	Reset State
<i>PREC_67</i>	31:25	74K family cores do some decoding of instructions when they're loaded into the I-cache, which helps speed instruction dispatch. When you use cache tag load/store instructions, you see that information here. The individual <i>PREC</i> fields hold precode information for pairs of adjacent instructions in the I-cache line, and the <i>P</i> fields hold parity over them.	R/W	Undefined
<i>PREC_45</i>	24:18		R/W	Undefined
<i>PREC_23</i>	17:11		R/W	Undefined
<i>PREC_01</i>	10:4		R/W	Undefined
<i>P_67</i>	3		R/W	Undefined
<i>P_45</i>	2		R/W	Undefined
<i>P_23</i>	1		R/W	Undefined
<i>P_01</i>	0		R/W	Undefined

7.2.49 IDataHi (CP0 Register 29, Select 1): High-order Data Read/write Interface for I-cache Special Cacheops

Staging registers for special **cache** which load or store data from or to the cache line. Two registers (*IDataHi*, *IDataLo*) are needed because the 74K core loads I-cache data at least 64-bits at a time.

Figure 7.56 IDataHi Register Format

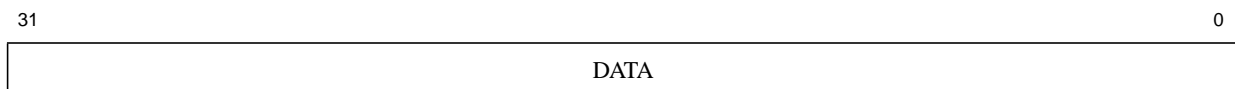


Table 7.64 IDataHi Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
<i>DATA</i>	31:0	High-order data read from the cache data array.	R/W	Undefined

7.2.50 DTagHi (CP0 Register 29, Select 2): D-cache Virtual Index (including ASID)

More cache tag bits for the 74K core’s dual-tagged L1 D-cache. For diagnostics only.

Figure 7.57 DTagHi Register Format



Table 7.65 Field Descriptions for DTagHi Register

Name	Bit(s)	Description	Read/ Write	Reset State
<i>VTAG</i>	31:12	74K family cores have a dual-tagged D-cache, combining a virtual tag for fast lookup with a physical tag to avoid <i>aliases</i> . Bit[11] always gets virtual address[11] of the tag when index load tag cache instruction is executed.	R/W	Undefined
<i>U</i>	11		R/W	Undefined
<i>G</i>	8		R/W	Undefined
<i>ASID</i>	7:0	These fields store the information required to match a virtual address: the virtual address itself, the ASID (tracking the "address space identifier" maintained in <i>EntryHiASID</i>) and a global ("G") bit which can be set to make it not necessary to match the ASID.	R/W	Undefined

7.2.51 L23DataHi (CP0 Register 29, Select 5): High-order Data Read/Write Interface for L2 or L3 cache

On 74K family cores, test software can read or write cache data using a **cache** index load/store data instruction. Which word of the cache line is transferred depends on the low address fed to the **cache** instruction.

Figure 7.58 L23DataHi Register Format

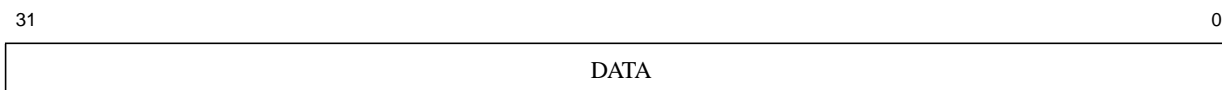


Table 7.66 L23DataHi Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
<i>DATA</i>	31:0	High-order data read from the cache data array.	R/W	Undefined

7.2.52 ErrorEPC (CP0 Register 30, Select 0): Restart Location from Reset or Cache Error Exception

This full 32-bit register is filled with the restart address on a cache error exception or any kind of CPU reset — in fact, any exception which sets *StatusEPC* and leaves the CPU in "error mode".

Figure 7.59 ErrorEPC Register Format

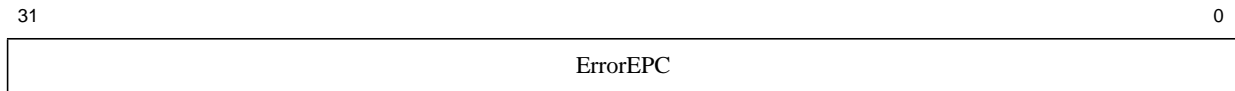


Table 7.67 ErrorEPC Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
<i>ErrorEPC</i>	31:0	Error Exception Program Counter.	R/W	Undefined

7.2.53 DESAVE (CP0 Register 31, Select 0): Scratch Read/Write Register for EJTAG Debug Exception Handler

Software-only register, with no hardware effect. Provided because the debug exception handler can't use the *k0-1* GP registers, used by ordinary exception handlers to bootstrap themselves: but a debug handler can save a GPR into *DESAVE*, and then use that GPR register in code which saves everything else.

Figure 7.60 DeSave Register Format



Table 7.68 DeSave Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
<i>DESAVE</i>	31:0	Debug exception save contents.	SO	Undefined

Hardware and Software Initialization of the 74K™ Core

A 74K processor core contains only a minimal amount of hardware initialization and relies on software to fully initialize the device.

This chapter contains the following sections:

- Section 8.1 “Hardware-Initialized Processor State”
- Section 8.2 “Software-Initialized Processor State”

8.1 Hardware-Initialized Processor State

A 74K processor core, like most other MIPS processors, is not fully initialized by hardware reset. Only a minimal subset of the processor state is cleared. This is enough to bring the core up while running in unmapped and uncached code space. All other processor state can then be initialized by software. Unlike previous MIPS processors, there is no distinction between cold and warm resets (or hard and soft resets). *SI_Reset* is used for both power-up reset and soft reset.

8.1.1 Coprocessor 0 State

Much of the hardware initialization occurs in Coprocessor 0:

- *Random* - cleared to maximum value on Reset (TLB/MMU only)
- *Wired* - cleared to 0 on Reset (TLB/MMU only)
- *Status_BEV* - set to 1 on Reset
- *Status_TS* - cleared to 0 on Reset
- *Status_NMI* - cleared to 0 on Reset
- *Status_ERL* - set to 1 on Reset
- *Status_RP* - cleared to 0 on Reset
- *CDMMBase_EN* - cleared to 0 on Reset
- *WatchLo_{I,R,W}* - cleared to 0 on Reset
- *Config* fields related to static inputs - set to input value by Reset
- *Config_K0* - set to 010 (uncached) on Reset

Hardware and Software Initialization of the 74K™ Core

- *Config_{KU}* - set to 010 (uncached) on Reset (FMT/MMU only)
- *Config_{K23}* - set to 010 (uncached) on Reset (FMT/MMU only)
- *Debug_{DM}* - cleared to 0 on Reset (unless EJTAGBOOT option is used to boot into Debug Mode, as described in Chapter 11, “EJTAG Debug Support in the 74K™ Core”).
- *Debug_{LSNM}* - cleared to 0 on Reset
- *Debug_{IBusEP}* - cleared to 0 on Reset
- *Debug_{DBusEP}* - cleared to 0 on Reset
- *Debug_{IEXI}* - cleared to 0 on Reset
- *Debug_{SSi}* - cleared to 0 on Reset

8.1.2 TLB Initialization

Each TLB entry has a “hidden” state bit, which is set by Reset and is cleared when the TLB entry is written. This bit disables matches and prevents “TLB Shutdown” conditions from being generated by the power-up values in the TLB array (when two or more TLB entries match a single address). This bit is not visible to software.

8.1.3 Bus State Machines

All pending bus transactions are aborted and the state machines in the bus interface unit are reset when a Reset exception is taken.

8.1.4 Static Configuration Inputs

All static configuration inputs (for example, those defining the bus mode and cache size) should only be changed during Reset.

8.1.5 Fetch Address

Upon Reset, unless the EJTAGBOOT option is used, the fetch is directed to VA 0xBFC00000 (PA 0x1FC00000). This address is in kseg1, which is unmapped and uncached, so that the TLB and caches do not require hardware initialization.

8.2 Software-Initialized Processor State

Software is required to initialize parts of the device, as described below.

8.2.1 Register File

The register file powers up in an unknown state with the exception of r0, which is always 0. Initializing the rest of the register file is not required for proper operation. Good code will generally not read a register before writing to it, but the boot code can initialize the register file for added safety.

8.2.2 TLB

Because of the hidden bit indicating initialization, the core does not initialize the TLB upon Reset. This is an implementation-specific feature of the 74K core and cannot be relied upon if writing generic code for MIPS32/64 processors.

8.2.3 Caches

The cache tag and data arrays power up to an unknown state and are not affected by reset. Every tag in the cache arrays should be initialized to an invalid state using the CACHE instruction (typically the Index Invalidate function). This can be a long process, especially because the instruction cache initialization must run in an uncached address region.

8.2.4 Coprocessor 0 State

Miscellaneous COP0 states need to be initialized before exiting the boot code. There are various exceptions which are blocked by *ERL*=1 or *EXL*=1, and which are not cleared by Reset. These can be cleared to avoid taking spurious exceptions when leaving the boot code.

- *Cause*: *WP* (Watch Pending), and *SW0* and *SW1* (Software Interrupts) should be cleared.
- *Config*: *K0* should be set to the desired Cache Coherency Algorithm (CCA) prior to accessing *kseg0*.
- *Config*: (FM MMU only) *KU* and *K23* should be set to the desired CCA for *useg/kugeg* and *kseg2/3* respectively prior to accessing those regions.
- *Count*: Should be set to a known value if timer interrupts are used.
- *Compare*: Should be set to a known value if timer interrupts are used. Note that the write to *Compare* will also clear any pending timer interrupts, so *Count* should be set before *Compare* to avoid any unexpected interrupts.
- *Status*: Desired state of the device should be set.
- Other COP0 state: Other registers should be written before they are read. Some registers are not explicitly writable, and are only updated as a by-product of instruction execution or a taken exception. Uninitialized bits should be masked off after reading these registers.

Caches of the 74K™ Core

This chapter describes the caches present in a 74K processor core. It contains the following sections:

- [Section 9.1 “Cache Configurations”](#)
- [Section 9.2 “Instruction Cache”](#)
- [Section 9.3 “Data Cache”](#)
- [Section 9.4 “Write Back Buffer”](#)
- [Section 9.5 “Cache Protocols”](#)
- [Section 9.6 “CACHE Instruction”](#)
- [Section 9.7 “Software Cache Testing”](#)
- [Section 9.8 “Memory Coherence Issues”](#)

9.1 Cache Configurations

A 74K processor core has separate instruction and data caches, which allows instruction and data references to proceed simultaneously. Each of the caches is 4-way set associative and can be configured at build time to be 0, 16, 32, or 64KB. Both caches use a 32B line size and support locking on a per line basis. Parity protection of the cache arrays is an optional feature.

9.2 Instruction Cache

[Table 9.1](#) shows the key characteristics of the instruction cache. [Figure 9.1](#) shows the format of an entry in the three arrays comprising the instruction cache: data, tag, and way-select.

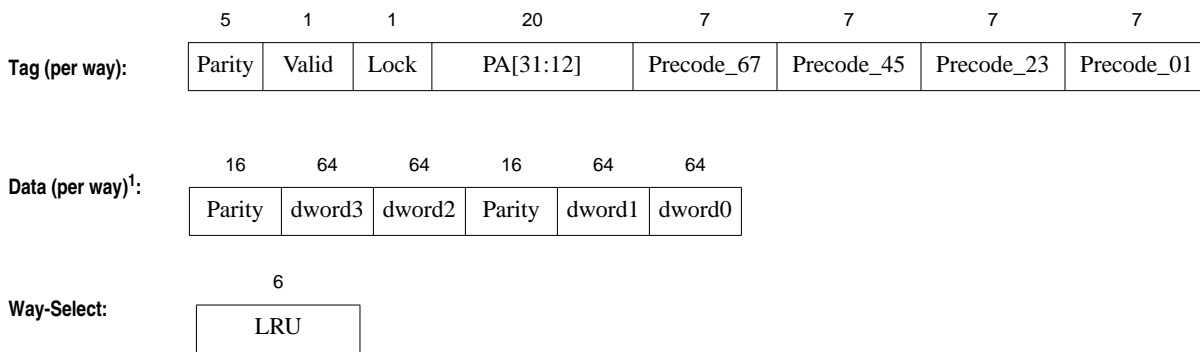
Table 9.1 Instruction Cache Attributes

Attribute	With Parity	Without Parity
Size	0, 16, 32, 64KB	
Line Size	32B	
Number of Cache Sets	128, 256, 512	
Associativity	4 way	
Replacement	LRU	
Cache Locking	per line	
Data Array		

Table 9.1 Instruction Cache Attributes (Continued)

Attribute	With Parity	Without Parity
Read Unit	144b x 4	128b x 4
Write Unit	144b	128b
Tag Array		
Read Unit	55b x 4	50b x 4
Write Unit	55b	50b
Way-Select Array		
Read Unit	6b	
Write Unit	1-6b	

Figure 9.1 Instruction Cache Organization



1. Parity bits in data array will be interleaved with precode and data bytes.

9.2.1 Virtual Aliasing

The instruction cache on the 74K processor core is virtually indexed and physically tagged. The lower bits of the virtual address are used to access the cache arrays and the physical address is used in the tags. Because the way size can be larger than the minimum TLB page size, there is a potential for virtual aliasing. This means that one physical address can exist in multiple indices within the cache, if it is accessed with different virtual addresses. Virtual aliasing comes into effect only for cache sizes that are larger than 16KB. The 16KB cache size does not suffer from virtual aliasing, because the way size equals the minimum page size.

This reduces the cache efficiency somewhat, but is generally not a problem unless the instruction stream is being written to. When instructions are written, software must ensure that the store data is written out to memory and the old data is invalidated in the instruction cache (via the CACHE or SYNCI instruction). Because one physical address can exist in multiple locations, the cache should be invalidated using all of the virtual addresses used to access that physical address. The hardware implementation takes care of this automatically when *Config7.IVA* bit is cleared. In the absence of the hardware feature, alternatively, all of the relevant cache indices or the entire cache can be invalidated.

9.2.2 Precode bits

In order for the fetch unit to quickly detect branches and jumps when executing code, the instruction cache array contains some additional precode bits. These bits indicate the type and location of branch or jump instructions within a 64b fetch bundle. These precode bits are not used when executing MIPS16e code.

9.2.3 Parity

Parity protection of the instruction cache arrays can optionally be included. The data array has 16 parity bits—one for each byte of the 128b data. The tag array has 5 parity bits for each tag—one for each of the 4 precode fields and one for the physical tag, lock, and valid bits. The LRU array does not have any parity.

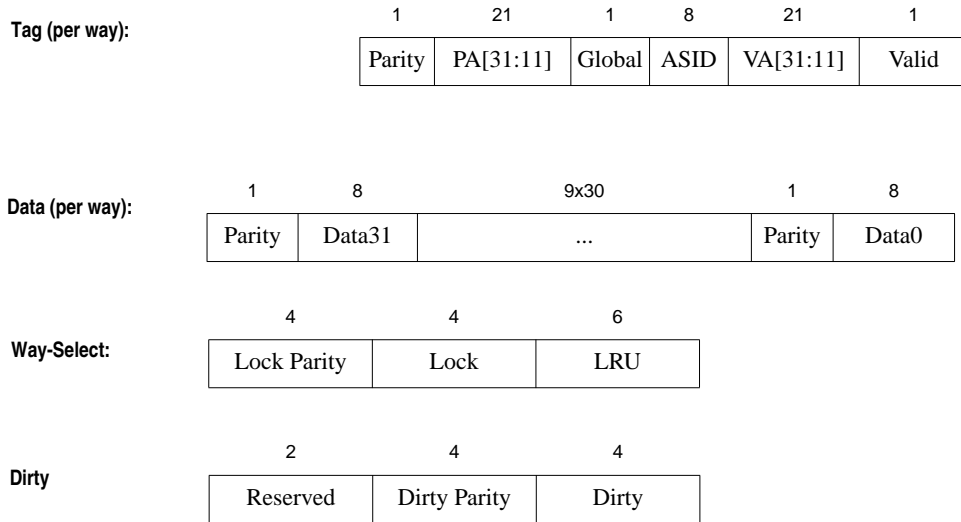
9.3 Data Cache

The data cache is similar to the instruction cache, with a few key differences. The data cache does not contain any precode information. To handle store bytes, the data array is byte-accessible, and the optional data parity is 1 bit per byte. The way-select array for the data cache holds the lock bits (and optional lock parity bits) for each cache line, in addition to the LRU information. The lock bits indicate the cache lines that have been locked using the CACHE instruction. There is a separate dirty array to hold the dirty bits of cache lines. [Table 9.2](#) shows the key characteristics of the data cache. [Figure 9.2](#) shows the format of an entry in the arrays comprising the data cache: tag, data, way-select, and dirty.

Table 9.2 Data Cache Attributes

Attribute	With Parity	Without Parity
Size	0, 16, 32, 64KB	
Line Size	32B	
Number of Cache Sets	128,256,512	
Associativity	4 way	
Replacement	LRU	
Cache Locking	per line	
Data Array		
Read Unit	72b x 4	64b x 4
Write Unit	9b	8b
Tag Array		
Read Unit	53b x 4	52b x 4
Write Unit	53b	52b
Way-Select Array		
Read Unit	14b	10b
Write Unit	1-14b	
Dirty Array		
Read Unit	10b	6b
Write Unit	1-10b	

Figure 9.2 Data Cache Organization



9.3.1 Virtual Aliasing

Since the caches are virtually indexed and physically tagged, a phenomenon known as *virtual aliasing* can occur for some cache sizes. Virtual aliasing occurs if the virtual bits used to index a cache array are not consistent with the overlapping physical bits, after the virtual address has been translated to a physical address. Virtual aliasing can only occur in address regions which are mapped through a TLB-based memory management unit.

In TLB-mapped address regions, virtual aliasing may occur if the cache size per way is greater than the page size. For example, consider a 32KB cache organized as 4-way set associative. The size per way is then 8 KB, so virtual address bits [12:0] are used to index the array. If the address is in a translated region with a page size of 4 KB, then address bits [11:0] are untranslated but address bits [31:12] will be mapped and for these bits the virtual and physical addresses may be different. In this example, bit [12] could pose a potential problem due to virtual aliasing. Imagine two virtual addresses, VA0 and VA1, whose only difference is the value of bit [12], which map to the same physical address. These two virtual addresses would be indexed to two different lines by the cache, even though they were intended to represent the same physical address. Then if a program does a load using VA0 and a store using VA1, or vice-versa, the cache may not return the expected data.

Table 9.3 shows the overlapped virtual/physical address bits which could potentially be involved in virtual aliasing, given the possible minimum page sizes and cache way sizes supported by a 74K core. Because there are no direct writes to the I-cache in the MIPS architecture, aliasing is usually an issue only for the D-cache. A special hardware mechanism is available to prevent the possibility of virtual aliasing in 32KB and 64KB data caches. In cores not configured with this mechanism, virtual aliasing must be handled by software. The software solution must ensure that the mapping of virtual address bits which overlap with physical address bits be handled consistently. The simplest approach is to ensure that the overlapping bits are unity-mapped (VA equals PA).

Table 9.3 Potential Virtual Aliasing Bits

Minimum Page Size (KB)	Cache Way Size (KB)	Overlapped address bits with possible aliasing
4	8	[12]
	16	[13:12]

Table 9.3 Potential Virtual Aliasing Bits (Continued)

Minimum Page Size (KB)	Cache Way Size (KB)	Overlapped address bits with possible aliasing
8	16	[13]

A related issue can occur in virtually indexed, physically tagged caches if the number of physical bits stored in the tag array does not fully overlap the physically translated bits for the smallest page size. For a 74K core, there are always at least 20 address bits stored in the cache tag, representing bits [31:12] of the physical address. Since the minimum page size is 4KB with bits [31:12] physically translated by the TLB, the cache tag size does overlap the translated bits and this issue will not occur.

9.3.2 Parity

Parity protection of the data cache arrays can optionally be included. The data array requires a parity bit for each byte, corresponding to the minimum write quantum for a store. The tag array has a single parity bit for each tag. The way-select array has separate parity bits to cover each lock bit, but the LRU bits are not covered by parity. The dirty array also has a parity bit for each dirty bit.

9.4 Write Back Buffer

The BIU includes a Write Back Buffer (WBB) that holds writes going to memory. This includes evictions from the data cache, as well as write-through stores, uncached stores, and uncached accelerated stores. The WBB consists of 4 entries, each of which is capable of holding 32B of data. The WBB also holds L2 CACHE instructions that are to be sent out on the bus.

The WBB will attempt to gather uncached accelerated (UCA) stores to allow full line burst writes. UCA behavior is described in [Section 9.4.1 “Uncached Accelerated Stores”](#). Write through stores can also be gathered in a WBB entry if ConfigMM= 1.

WBB entries are ‘flushed’ under a variety of conditions. When a buffer is flushed, the write command is queued in the BIU and the WBB entry will not accept any more activity until the data has been written to the bus and the buffer is freed up. UCA flush conditions are described in the next section. Flush conditions for other types are shown here:

- Uncached (non-accelerated) stores flush immediately
- L2 CACHE instruction commands are also flushed immediately
- Entries for D\$ evictions are flushed when all 4 dwords (32B) of data have been gathered
- Write-through entries are flushed under the following conditions:
 - A full 32B line has been gathered
 - A read request matches the address of the WT line. The write command will be ordered ahead of the read command. There is no direct bypass of the WBB data to the read—the read gets the data from memory.
 - A WT request to a different 32B line is seen. Only 1 WT merge can be active at any time.

9.4.1 Uncached Accelerated Stores

Uncached Accelerated gathering is supported for word and doubleword stores only.

Gathering of uncached accelerated stores will start on cache-line aligned addresses, i.e. 32 byte aligned addresses. Uncached accelerated word or doubleword stores that do not meet the conditions required to start gathering will be treated like regular uncached stores.

When an uncached accelerated store meets the requirements needed to start gathering, a gather buffer is reserved for this store. All subsequent uncached accelerated word or doubleword stores to the same cache line will write sequentially into this buffer, regardless of the word address associated with these stores. The uncached accelerated buffer is tagged with the address of the first store.

An uncached accelerated buffer is written to memory (flushed) if:

1. The last word in the entry being gathered is written (implicit flush).
2. A PREF Nudge which match the address associated with the gather buffer (explicit flush).
3. A SYNC instruction is executed (Explicit flush).
4. Bits <31:5> of the address of a Load instruction match the address associated with the gather buffer (implicit flush).
5. Uncached Accelerated store to a different 32B line (implicit flush).
6. An exception occurs (implicit flush).

When an uncached accelerated buffer is flushed, the address sent out on the system interface is the address associated with the gather buffer.

Caveats:

- Any uncached stores and any uncached loads to unrelated addresses that occur between uncached accelerated stores that are part of a gather sequence will go out-of-order. They will not enforce ordering.
- The only constraint imposed on the gathering is that doubleword stores are only allowed to write to doubleword-aligned locations in the buffer. For example, if uncached accelerated gathering starts with a Store Word (SW), it may not be followed by a Store Double (SDC1).
- Uncached accelerated stores of the following types are not intended to be used by software and may generate unpredictable results:
 1. Halfword Stores
 2. Unaligned Stores
 3. Store conditionals
- In order for software to execute correctly on implementations without uncached accelerated stores, software should always generate accesses starting on a cache-line aligned address, proceed to generate correctly incremented sequential addresses, and observe the restrictions for uncached accelerated stores.

9.5 Cache Protocols

This section describes cache organization, attributes, and cache-line replacement for the instruction and data caches. This section also discusses issues relating to virtual aliasing.

9.5.1 Cache Organization

The instruction and data caches each consist of three arrays: tag, data, and way-select. In addition, the data cache has a dirty array. The caches are virtually indexed, since a virtual address is used to select the appropriate line within each of the three arrays. The caches are physically tagged, as the tag array contains a physical, not virtual, address.

The tag and data arrays hold 4 ways of information per set, corresponding to the 4-way set associativity of the cache. The way-select array holds information to choose the way to be filled, as well as dirty bits in the case of the data cache.

Figure 9.1 (instruction cache) and Figure 9.2 (data cache) show the format of each line in the tag, data, and way-select arrays.

A tag entry consists of the upper bits of the physical address (bits [31:12] for instruction cache, bits[31:11] for data cache), one valid bit for the line, and a lock bit. A data entry contains the four, 64-bit doublewords in the line, for a total of 32 bytes. All four words in the line are present or not in the data array together, hence the single valid bit stored with the tag. Once a valid line is resident in the cache, byte, halfword, triple-byte or full word stores can update all or a portion of the words in that line. The tag and data entries are repeated for each of the 4 lines in the set.

A way-select entry holds bits choosing the way to be replaced according to a Least Recently Used (LRU) algorithm. The LRU information applies to all the ways and there is one way-select entry for all the ways in the set. The array with way-select entries for the data cache also holds dirty bits for the lines. One dirty bit is required per line, as shown in Figure 9.2. The instruction cache only supports reads, hence only LRU entries are stored in the instruction way-select array.

9.5.2 Cacheability Attributes

A 74K core supports the following cacheability attributes:

- *Uncached*: Addresses in a memory area indicated as uncached are not read from the cache. Stores to such addresses are written directly to main memory, without changing cache contents.
- *Writeback With Write Allocation*: Loads and instruction fetches first search the cache, reading main memory only if the desired data does not reside in the cache. On data store operations, the cache is first searched to see if the target address is in the cache. If it is, the cache contents are updated, but main memory is not written. If the cache lookup misses on a store, main memory is read to bring the line into the cache and merge it with the new store data. Hence, the allocation policy on a cache miss is read- or write-allocate. Data stores will update the appropriate dirty bit in the way-select array to indicate that the line contains modified data. When a line with dirty data is displaced from the cache, it is written back to memory.
- *Write-through With No Write Allocation*: Loads and instruction fetches first search the cache, reading main memory only if the desired data does not reside in the cache. On data store operations, the cache is first searched to see if the target address is cache resident. If it is resident, the cache contents are updated, and main memory is also written. If the cache lookup misses on a store, only main memory is written. Hence, the allocation policy on a cache miss is read-allocate only.

- *Uncached Accelerated*: Uncached stores are gathered together for more efficient bus utilization. See [Section 9.4.1 “Uncached Accelerated Stores”](#) for more details

Some segments of memory employ a fixed caching policy; for example, `kseg1` is always uncacheable. Other segments of memory allow the caching policy to be selected by software. Generally, the cache policy for these programmable regions is defined by a cacheability attribute field associated with that region of memory. See [Chapter 5, “Memory Management of the 74K™ Core”](#) for further details.

9.5.3 Replacement Policy

The replacement policy refers to how a way is chosen to hold an incoming cache line on a miss which will result in a cache fill. The replacement policy is least-recently used (LRU), but excluding any locked ways. The LRU bit(s) in the way-select array encode the order in which ways on that line have been accessed.

On a cache miss, the lock and LRU bits for the tag and way-select entries of the selected line may be used to determine the way which will be chosen.

The LRU field in the way select array is updated as follows:

- On a cache hit, the associated way is updated to be the most recently used. The order of the other ways relative to each another is unchanged.
- On a cache refill, the filled way is updated to be the most recently used.
- On CACHE instructions, the update of the LRU bits depends on the type of operation to be performed:
 - **Index (Writeback) Invalidate**: Least-recently used.
 - **Index Load Tag**: No update.
 - **Index Store Tag, *WST=0***: Most-recently used if valid bit is set in *TagLo* CP0 register. Least-recently used if valid bit is cleared in *TagLo* CP0 register.
 - **Index Store Tag, *WST=1***: Update the field with the contents of the *TagLo* CP0 register (refer to Table 7.58 for the valid values of this field).
 - **Index Store Data**: No update.
 - **Hit Invalidate**: Least-recently used if a hit is generated, otherwise unchanged.
 - **Fill**: Most-recently used.
 - **Hit (Writeback) Invalidate**: Least-recently used if a hit is generated, otherwise unchanged.
 - **Hit Writeback**: No update.
 - **Fetch and Lock**: For instruction cache, no update. For data cache, most-recently used.

If all ways are valid, then any locked ways will be excluded from consideration for replacement. For the unlocked ways, the LRU bits are used to identify the way which has been used least-recently, and that way is selected for replacement.

If the way selected for replacement has its dirty bit asserted in the way-select array, then that 32-byte line will be written back to memory before the new fill can occur.

9.5.4 Line Locking

The 74K core does not support the locking of all 4 ways of either cache at a particular index. If all 4 ways of the cache at a given index are locked by either Fetch and Lock or Index Store Tag CACHE instructions, subsequent cache misses at that cache index will displace one of the locked lines.

- The core does not support the locking of all 4 ways of either cache at a particular index. If all 4 ways of the cache at a given index are locked, subsequent cache misses at that cache index will displace one of the locked lines.

9.6 CACHE Instruction

Both caches support the CACHE instructions, which allow users to manipulate the contents of the Data and Tag arrays, including the locking of individual cache lines. These instructions are described in detail in [Chapter 13, “74K™ Processor Core Instructions”](#).

The CACHE Index Load Tag and Index Store Tag instructions can be used to read and write the Way Select (WS) RAM by setting the *WST* bit in the *ErrCtl* register. (The *ErrCtl* register is described in [Section 7.2.40 “ErrCtl \(CP0 Register 26, Select 0\): Software Parity Control and Test Modes for Cache RAM Arrays”](#).) Similarly, the *SPR* bit in the *ErrCtl* register will cause Index Load Tag and Index Store Tag instructions to read the pseudo-tags associated with the scratchpad RAM array. Note that when the *WST* and *SPR* bits are zero, the CACHE index instructions access the cache Tag array.

Not all values of the WS field are valid for defining the order in which the ways are selected. This is only an issue, however, if the WS RAM is written after the initialization (invalidation) of the Tag array. Valid WS field encodings for way selection order is shown in [Table 9.4](#).

Table 9.4 Way Selection Encoding, 4 Ways

Selection Order ¹	WS[5:0]	Selection Order	WS[5:0]
0123	000000	2013	100010
0132	000001	2031	110010
0213	000010	2103	100110
0231	010010	2130	101110
0312	010001	2301	111010
0321	010011	2310	111110
1023	000100	3012	011001
1032	000101	3021	011011
1203	100100	3102	011101
1230	101100	3120	111101
1302	001101	3201	111011
1320	101101	3210	111111

1. The order is indicated by listing the least-recently used way to the left and the most-recently used way to the right, etc.

9.7 Software Cache Testing

Typically, the cache RAM arrays will be tested using BIST. It is, however, possible for software running on the processor to test all of the arrays. Of course, testing of the I-cache arrays should be done from an uncacheable space with interrupts disabled in order to maintain the cache contents. There are multiple methods for testing these arrays in software, some of which are described in the following subsections.

9.7.1 I-cache and D-cache Tag Arrays

These arrays can be tested via the Index Load Tag and Index Store Tag varieties of the CACHE instruction. Index Store Tag will write the contents of the *ITagLo* and *ITagHi* registers into the selected tag entry. Index Load Tag will read the selected tag entry into the *ITagLo* and *ITagHi* registers.

If parity is implemented, the parity bits can be tested as normal bits by setting the *PO* (parity override) bit in the *ErrCtl* register. This will override the parity calculation and use the parity bits in *ITagLo* and *ITagHi* as the parity values.

9.7.2 I-cache Data Array

This array can be tested using the Index Store Data and Index Load Tag varieties of the CACHE instruction. The Index Store Data variety is enabled by setting the *WST* bit in the *ErrCtl* register.

The Index Store Data instruction can optionally update the corresponding precode field in the tag array. The precode bits in the array are updated if the *PCD* bit in the *ErrCtl* register is zero when we execute the Index Store Data instruction. The precode value is generated by the hardware automatically if the *PCO* bit in the *ErrCtl* register is zero. Otherwise, the corresponding precode value (*PREC_01*/*PREC_23*/*PREC_45*/*PREC_67*) from the *ITagHi* register is used in updating the tag array.

The parity bits in the array can be tested by setting the *PO* bit in the *ErrCtl* register. This will use the *PI* field in *ErrCtl* instead of calculating the parity on a write.

The rest of the data bits are read/written to/from the *IDataLo* and *IDataHi* registers.

9.7.3 I-cache WS Array

The testing of this array is done with via Index Load Tag and Index Store Tag CACHE instructions. By setting the *WST* bit in the *ErrCtl* register, these operations will read and write the WS array instead of the tag array.

9.7.4 D-cache Data Array

This array can be tested using the Index Store Tag CACHE, SW, and LW instructions. First, use Index Store Tag to set the initial state of the tags to valid with a known physical address (PA). Write the array using SW instructions to the PAs that are resident in the cache. The value can then be read using LW instructions and compared to the expected data.

The parity bits can be implicitly tested using this mechanism. The parity bits can be explicitly tested by setting the *PO* bit in *ErrCtl* and using Index Store Data and Index Load Tag CACHE operations. The parity bits (one bit per byte) are read/written to/from the *PD* field in *ErrCtl*. Unlike the I-cache, the *DataHi* register is not used, and only 32b of data is read/written per operation.

9.7.5 D-cache WS Array

The lock and LRU bits can be tested using the same mechanism as the I-cache WS array.

9.7.6 D-cache DirtyArray

The testing of this array is also done through Index Load Tag and Index Store Tag CACHE instructions. By setting the *DYT* bit in the *ErrCtl* register, these operations will read and write the dirty array instead of the tag array.

9.8 Memory Coherence Issues

A cache presents coherency issues within the memory hierarchy which must be considered in the system design. Since a cache holds a copy of memory data, it is possible for another memory master to modify a memory location, thus making other copies of that location stale. A detailed discussion of memory coherence is beyond the scope of this document, but following are a few related comments.

A 74K processor contains no direct hardware support for managing coherency with respect to its caches, so it must be handled via the system design or software. The data cache supports either write-back or write-through protocols.

In write-through mode, all data writes will eventually be sent to memory. However, because of the presence of write buffers, there could be a delay in the actual write to memory. So if another memory master updates cacheable memory that could also be in the core's caches, those locations may need to be flushed from the cache. The only way to accomplish this invalidation is by use of the CACHE instruction.

In write-back mode, data writes only go to the cache and not to memory (until explicitly evicted). So the processor cache may contain the *only* copy of data in the system, until that data is written to main memory. Dirty lines are only written to memory when displaced from the cache as a new line is filled, or if they are explicitly forced by certain flavors of the CACHE or PREF instructions.

The SYNC instruction may also be useful to software in enforcing memory coherence, because it flushes the core's write buffers.

Power Management in the 74K™ Core

A 74K processor core offers a number of power management features, including low-power design, active power management, and power-down modes of operation. The core is a static design that supports a WAIT instruction designed to signal the rest of the device that execution and clocking should be halted, reducing system power consumption during idle periods.

The core provides two mechanisms for system-level low-power support, which are discussed in the following sections:

- [Section 10.1 “Register-Controlled Power Management”](#)
- [Section 10.2 “Instruction-Controlled Power Management”](#)

10.1 Register-Controlled Power Management

The *RP* (Reduced Power) bit in the CP0 *Status* register enables a standard software mechanism for placing the system into a low-power state. The state of the *RP* bit is available externally on the *SI_RP* output signal. Three additional pins—*SI_EXL*, *SI_ERL*, and *EJ_DebugM*—support the power-management functions by allowing the user to change the power state if an exception or error occurs while the core is in a low-power state.

Setting the *RP* bit of the CP0 *Status* register causes the core to assert the *SI_RP* signal. The external agent can then decide to reduce the clock frequency and place the core into power-down mode.

If an interrupt occurs while the device is in power-down mode, that interrupt may need to be serviced, depending on the needs of the application. The interrupt causes an exception, which in turn causes the *EXL* bit to be set. Setting the *EXL* bit causes the assertion of the *SI_EXL* signal on the external bus, indicating to the external agent that an interrupt has occurred. When *SI_EXL* is asserted, the external agent can choose to either speed-up the clocks and service the interrupt or let it be serviced at the lower clock speed.

The setting of the *ERL* bit causes the assertion of the *SI_ERL* signal on the external bus, indicating to the external agent that an error has occurred. The external agent can then choose to either speed up the clocks and service the error or let it be serviced at the lower clock speed.

Similarly, the *EJ_DebugM* signal indicates that the processor is in debug mode. Debug mode is entered when the processor takes a debug exception. If fast handling of this is desired, the external agent can speed up the clocks.

The core provides four power-down signals that are part of the system interface. Three of the pins change state as the corresponding bits in the CP0 *Status* register are set or cleared, and the fourth pin indicates that the processor is in debug mode:

- The *SI_RP* signal represents the state of the *RP* bit (27) in the CP0 *Status* register.
- The *SI_EXL* signal represents the state of the *EXL* bit (1) in the CP0 *Status* register.

Power Management in the 74K™ Core

- The *SI_ERL* signal represents the state of the *ERL* bit (2) in the CP0 *Status* register.
- The *EJ_DebugM* signal indicates that the processor has entered debug mode.

10.2 Instruction-Controlled Power Management

The second mechanism for invoking power-down mode is through execution of the WAIT instruction. The WAIT instruction brings the processor into a low-power state, where the internal clocks are suspended and the pipeline is frozen. However, the internal timer and some of the input pins (*SI_Int*[5:0], *SI_NMI*, *SI_Reset*, and *EJ_DINT*) continue to run. The clocks are not shut down until all bus and coprocessor transactions have completed. When the CPU is in instruction-controlled power management mode, any enabled interrupt, NMI, debug interrupt, or reset condition causes the CPU to exit this mode and resume normal operation. While the core is in this low-power mode, the *SI_SLEEP* signal is asserted to indicate to external agents the state of the chip.

EJTAG Debug Support in the 74K™ Core

The EJTAG debug logic in the 74K processor core is compliant with MIPS® EJTAG Specification Version 4.12 and includes:

1. Standard core debug features
2. Optional hardware breakpoints
3. Standard Test Access Port (TAP) for a dedicated connection to a debug host
4. Optional MIPS trace capability for program counter/data address/data value trace to on-chip memory or to trace probe

This chapter contains the following sections:

- [Section 11.1 “Debug Control Register”](#)
- [Section 11.2 “Hardware Breakpoints”](#)
- [Section 11.3 “Test Access Port \(TAP\)”](#)
- [Section 11.4 “EJTAG TAP Registers”](#)
- [Section 11.5 “TAP Processor Accesses”](#)
- [Section 11.6 “PC Sampling”](#)
- [Section 11.7 “Fast Debug Channel”](#)
- [Section 11.8 “MIPS® Trace”](#)
- [Section 11.9 “PDtrace™ Registers \(Software Control\)”](#)
- [Section 11.10 “Trace Control Block \(TCB\) Registers \(Hardware Control\)”](#)
- [Section 11.11 “Enabling MIPS Trace”](#)
- [Section 11.12 “TCB Trigger Logic”](#)
- [Section 11.13 “MIPS Trace Cycle-by-Cycle Behavior”](#)
- [Section 11.14 “TCB On-Chip Trace Memory”](#)

11.1 Debug Control Register

The *Debug Control* register (*DCR*) controls and provides information about debug issues, and is always provided with the CPU core. The register is memory-mapped in *drseg* at offset 0x0.

The *DataBrk* and *InstBrk* bits indicate if hardware breakpoints are included in the implementation, and debug software is expected to read hardware breakpoint registers for additional information.

Hardware and software interrupts are maskable for non-debug mode with the *INTE* bit, which works in addition to the other mechanisms for interrupt masking and enabling. NMI is maskable in non-debug mode with the *NMIE* bit, and a pending NMI is indicated through the *NMIP* bit.

The *SRE* bit allows implementation dependent masking of some sources for reset. The 74K core does not distinguish between soft and hard reset, but typically only soft reset sources in the system would be maskable, and hard sources such as the reset switch would not be. The soft reset masking should only be applied to a soft reset source if that source can be efficiently masked in the system, thus resulting in no reset at all. If that is not possible, then that soft reset source should not be masked, since a partial soft reset may cause the system to fail or hang. There is no automatic indication of whether the *SRE* is effective, so the user must consult system documentation.

The *PE* bit reflects the *ProbEn* bit from the *EJTAG Control* register (*ECR*), whereby the probe can indicate to the debug software running on the CPU if the probe expects to service dmseg accesses. The reset value in the table below takes effect on any CPU reset.

Figure 11.1 Debug Control Register Format

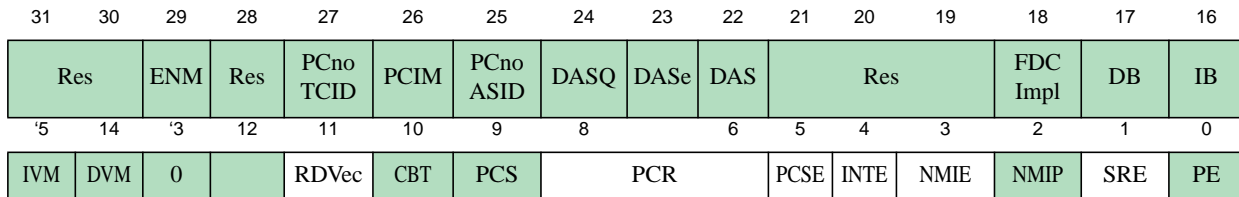


Table 11.1 Debug Control Register Field Descriptions

Fields		Description	Read / Write	Reset State						
Name	Bit(s)									
Res	31:30	Reserved	R	0						
<i>ENM</i>	29	Endianness in which the processor is running in kernel and Debug Mode: <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Little endian</td> </tr> <tr> <td>1</td> <td>Big endian</td> </tr> </tbody> </table>	Encoding	Meaning	0	Little endian	1	Big endian	R	Preset
Encoding	Meaning									
0	Little endian									
1	Big endian									
Res	28	Reserved	R	0						

Table 11.1 Debug Control Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State						
Name	Bit(s)									
<i>PCnoTCID</i>	27	Controls whether PC Sampling includes or omits the TC identity field when the MT ASE is implemented: <table border="1" data-bbox="521 415 1045 533"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>TC field included in PCSAMPLE scan</td> </tr> <tr> <td>1</td> <td>TC field omitted from PCSAMPLE scan</td> </tr> </tbody> </table>	Encoding	Meaning	0	TC field included in PCSAMPLE scan	1	TC field omitted from PCSAMPLE scan	Read, Optional write	Preset
Encoding	Meaning									
0	TC field included in PCSAMPLE scan									
1	TC field omitted from PCSAMPLE scan									
<i>PCIM</i>	26	Configures PC Sampling to capture all executed addresses or only those that miss in the instruction cache: <table border="1" data-bbox="514 653 1045 800"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>All PCs are captured</td> </tr> <tr> <td>1</td> <td>Only PCs that miss in the instruction cache are captured</td> </tr> </tbody> </table>	Encoding	Meaning	0	All PCs are captured	1	Only PCs that miss in the instruction cache are captured	Read, Optional write	Undefined
Encoding	Meaning									
0	All PCs are captured									
1	Only PCs that miss in the instruction cache are captured									
<i>PCnoASID</i>	25	Controls whether the PCSAMPLE scan chain includes or omits the ASID field: <table border="1" data-bbox="532 890 1034 1005"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>ASID included in PCSAMPLE scan</td> </tr> <tr> <td>1</td> <td>ASID omitted from PCSAMPLE scan</td> </tr> </tbody> </table>	Encoding	Meaning	0	ASID included in PCSAMPLE scan	1	ASID omitted from PCSAMPLE scan	R	Undefined
Encoding	Meaning									
0	ASID included in PCSAMPLE scan									
1	ASID omitted from PCSAMPLE scan									
<i>DASQ</i>	24	Qualifies Data Address Sampling using a data breakpoint: <table border="1" data-bbox="532 1073 1034 1188"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>All data addresses are sampled</td> </tr> <tr> <td>1</td> <td>Sample matches of data breakpoint 0</td> </tr> </tbody> </table>	Encoding	Meaning	0	All data addresses are sampled	1	Sample matches of data breakpoint 0	R	0
Encoding	Meaning									
0	All data addresses are sampled									
1	Sample matches of data breakpoint 0									
<i>DASe</i>	23	Enables Data Address Sampling: <table border="1" data-bbox="532 1251 1034 1367"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Data Address sampling disabled</td> </tr> <tr> <td>1</td> <td>Data Address sampling enabled</td> </tr> </tbody> </table>	Encoding	Meaning	0	Data Address sampling disabled	1	Data Address sampling enabled	R	0
Encoding	Meaning									
0	Data Address sampling disabled									
1	Data Address sampling enabled									
<i>DAS</i>	22	Indicates if the Data Address Sampling feature is implemented: <table border="1" data-bbox="532 1430 1034 1570"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No Data Address sampling implemented</td> </tr> <tr> <td>1</td> <td>Data Address sampling implemented</td> </tr> </tbody> </table>	Encoding	Meaning	0	No Data Address sampling implemented	1	Data Address sampling implemented	R	Preset
Encoding	Meaning									
0	No Data Address sampling implemented									
1	Data Address sampling implemented									
Res	21:19	Reserved	R	0						
<i>FDCImpl</i>	18	Indicates if the fast debug channel is implemented: <table border="1" data-bbox="532 1692 1034 1808"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No fast debug channel implemented</td> </tr> <tr> <td>1</td> <td>Fast debug channel implemented</td> </tr> </tbody> </table>	Encoding	Meaning	0	No fast debug channel implemented	1	Fast debug channel implemented	R	1
Encoding	Meaning									
0	No fast debug channel implemented									
1	Fast debug channel implemented									

Table 11.1 Debug Control Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State						
Name	Bit(s)									
<i>DB</i>	17	Indicates if data hardware breakpoint is implemented: <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No data hardware breakpoint implemented</td> </tr> <tr> <td>1</td> <td>Data hardware breakpoint implemented</td> </tr> </tbody> </table>	Encoding	Meaning	0	No data hardware breakpoint implemented	1	Data hardware breakpoint implemented	R	Preset
Encoding	Meaning									
0	No data hardware breakpoint implemented									
1	Data hardware breakpoint implemented									
<i>IB</i>	16	Indicates if instruction hardware breakpoint is implemented: <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No instruction hardware breakpoint implemented</td> </tr> <tr> <td>1</td> <td>Instruction hardware breakpoint implemented</td> </tr> </tbody> </table>	Encoding	Meaning	0	No instruction hardware breakpoint implemented	1	Instruction hardware breakpoint implemented	R	Preset
Encoding	Meaning									
0	No instruction hardware breakpoint implemented									
1	Instruction hardware breakpoint implemented									
Res	13:11	Reserved	R	0						
<i>IVM</i>	15	Indicates if inverted data value match on data hardware breakpoints is implemented: <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No inverted data value match on data hardware breakpoints implemented</td> </tr> <tr> <td>1</td> <td>Inverted data value match on data hardware breakpoints implemented</td> </tr> </tbody> </table>	Encoding	Meaning	0	No inverted data value match on data hardware breakpoints implemented	1	Inverted data value match on data hardware breakpoints implemented	R	0
Encoding	Meaning									
0	No inverted data value match on data hardware breakpoints implemented									
1	Inverted data value match on data hardware breakpoints implemented									
<i>DVM</i>	14	Indicates if a data value store on a data value breakpoint match is implemented: <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No data value store on a data value breakpoint match implemented</td> </tr> <tr> <td>1</td> <td>Data value store on a data value breakpoint match implemented</td> </tr> </tbody> </table>	Encoding	Meaning	0	No data value store on a data value breakpoint match implemented	1	Data value store on a data value breakpoint match implemented	R	0
Encoding	Meaning									
0	No data value store on a data value breakpoint match implemented									
1	Data value store on a data value breakpoint match implemented									
<i>RDVec</i>	11	Enables relocation of the debug exception vector. The value in the DebugVectorAddr register is used for EJTAG exceptions when ProbTrap=0 and RDVec=1.	R/W	0						
<i>CBT</i>	10	Indicates if complex breakpoint block is implemented: <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No complex breakpoint block implemented</td> </tr> <tr> <td>1</td> <td>Complex breakpoint block implemented</td> </tr> </tbody> </table>	Encoding	Meaning	0	No complex breakpoint block implemented	1	Complex breakpoint block implemented	R	Preset
Encoding	Meaning									
0	No complex breakpoint block implemented									
1	Complex breakpoint block implemented									

Table 11.1 Debug Control Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State						
Name	Bit(s)									
<i>PCS</i>	9	Indicates if the PC Sampling feature is implemented.: <table border="1" data-bbox="532 384 1036 499"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No PC Sampling implemented</td> </tr> <tr> <td>1</td> <td>PC Sampling implemented</td> </tr> </tbody> </table>	Encoding	Meaning	0	No PC Sampling implemented	1	PC Sampling implemented	R	Preset
Encoding	Meaning									
0	No PC Sampling implemented									
1	PC Sampling implemented									
<i>PCR</i>	8:6	PC Sampling Rate: Values from 0 to 7 map to 2^5 to 2^{12} cycles respectively. That is, a PC sample is written out every 32, 64, 128, 256, 512, 1024, 2048, or 4096 cycles. The external probe or software is allowed to set this value to the desired sample rate	R/W	7						
<i>PCSE</i>	5	Indicates if PC Sampling is enabled: <table border="1" data-bbox="532 705 1036 821"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>PC Sampling not enabled</td> </tr> <tr> <td>1</td> <td>PC Sampling enabled</td> </tr> </tbody> </table> This bit is set to 0 following Reset. It must be set by software to enable PC sampling.	Encoding	Meaning	0	PC Sampling not enabled	1	PC Sampling enabled	R/W	0
Encoding	Meaning									
0	PC Sampling not enabled									
1	PC Sampling enabled									
<i>INTE</i>	4	Interrupt Enable in Normal Mode. This bit provides the hardware and software interrupt enable for non-debug mode, in addition to other masking mechanisms in conjunction with other disable mechanisms: <table border="1" data-bbox="532 1031 1036 1171"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Interrupt disabled</td> </tr> <tr> <td>1</td> <td>Interrupt enabled depending on other enabling mechanisms</td> </tr> </tbody> </table>	Encoding	Meaning	0	Interrupt disabled	1	Interrupt enabled depending on other enabling mechanisms	R/W	1
Encoding	Meaning									
0	Interrupt disabled									
1	Interrupt enabled depending on other enabling mechanisms									
<i>NMIE</i>	3	Non-Maskable Interrupt (NMI) enable for Non-Debug Mode: <table border="1" data-bbox="532 1247 1036 1362"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>NMI disabled</td> </tr> <tr> <td>1</td> <td>NMI enabled</td> </tr> </tbody> </table>	Encoding	Meaning	0	NMI disabled	1	NMI enabled	R/W	1
Encoding	Meaning									
0	NMI disabled									
1	NMI enabled									
<i>NMIP</i>	2	Indication for pending NMI: <table border="1" data-bbox="532 1434 1036 1549"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No NMI pending</td> </tr> <tr> <td>1</td> <td>NMI pending</td> </tr> </tbody> </table>	Encoding	Meaning	0	No NMI pending	1	NMI pending	R	0
Encoding	Meaning									
0	No NMI pending									
1	NMI pending									

Table 11.1 Debug Control Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State						
Name	Bit(s)									
<i>SRE</i>	1	<p>Soft Reset Enable</p> <p>This bit allows the system to mask soft resets. The core does not internally mask resets. Rather the state of this bit appears on the <i>EJ_SRstE</i> external output signal, allowing the system to mask soft resets if desired.</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Soft reset masked for soft reset sources dependent on implementation</td> </tr> <tr> <td>1</td> <td>Soft reset is fully enabled</td> </tr> </tbody> </table> <p>Bit is read-only (R) and reads as zero if not implemented.</p>	Encoding	Meaning	0	Soft reset masked for soft reset sources dependent on implementation	1	Soft reset is fully enabled	R/W	1
Encoding	Meaning									
0	Soft reset masked for soft reset sources dependent on implementation									
1	Soft reset is fully enabled									
<i>PE</i>	0	<p>Probe Enable</p> <p>This bit reflects the value of the <i>ProbEn</i> bit in the EJTAG Control register:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No access should occur to the dmseg segment</td> </tr> <tr> <td>1</td> <td>Probe services accesses to the dmseg segment</td> </tr> </tbody> </table> <p>Bit is read-only (R) and reads as zero if not implemented.</p>	Encoding	Meaning	0	No access should occur to the dmseg segment	1	Probe services accesses to the dmseg segment	R	Same value as <i>ProbEn</i> in <i>ECR</i> (see Table 11.25)
Encoding	Meaning									
0	No access should occur to the dmseg segment									
1	Probe services accesses to the dmseg segment									

11.2 Hardware Breakpoints

Hardware breakpoints provide for the comparison by hardware of executed instructions and data load/store transactions. It is possible to set instruction breakpoints on addresses even in ROM area. Data breakpoints can be set to cause a debug exception on a specific data transaction. Instruction and data hardware breakpoints are alike in many aspects, and are thus described in parallel in the following. The term hardware is not applied to breakpoint, unless required to distinguish it from software breakpoint.

There are two types of simple hardware breakpoints implemented in the 74K core: instruction breakpoints and data breakpoints.

A core may be configured with the following breakpoint options:

- No breakpoints
- Four instruction and two data breakpoints

11.2.1 Features of Instruction Breakpoint

Instruction breaks occur on instruction fetch operations and the break is set on the virtual address on the bus between the CPU and the instruction cache. Instruction breaks can also be made on the ASID value used by the TLB-based MMU. Finally, a mask can be applied to the virtual address to set breakpoints on a range of instructions.

Instruction breakpoints compare the virtual address of the executed instructions (PC) and the ASID with the registers for each instruction breakpoint including masking of address and ASID. When an instruction breakpoint matches, a debug exception and/or a trigger is generated. An internal bit in the instruction breakpoint registers is set to indicate that the match occurred.

11.2.2 Features of Data Breakpoint

Data breakpoints occur on load/store transactions. Breakpoints are set on virtual address and ASID values, similar to the Instruction breakpoint. Data breakpoints can be set on a load, a store or both. Data breakpoints can also be set based on the value of the load/store operation. Finally, masks can be applied to both the virtual address and the load/store value.

Data breakpoints compare the transaction type (TYPE), which may be load or store, the virtual address of the transaction (ADDR), the ASID, accessed bytes (BYTELANE) and data value (DATA), with the registers for each data breakpoint including masking or qualification on the transaction properties. When a data breakpoint matches, a debug exception and/or a trigger is generated, and an internal bit in the data breakpoint registers is set to indicate that the match occurred. The match is precise in that the debug exception or trigger occurs on the instruction that caused the breakpoint to match.

11.2.3 Instruction Breakpoint Registers Overview

The register with implementation indication and status for instruction breakpoints in general is shown in [Table 11.2](#).

Table 11.2 Overview of Status Register for Instruction Breakpoints

Register Mnemonic	Register Name and Description
<i>IBS</i>	Instruction Breakpoint Status

The four instruction breakpoints are numbered 0 to 3 for registers and breakpoints, and the number is indicated by n. The registers for each breakpoint are shown in [Table 11.3](#).

Table 11.3 Overview of Registers for Each Instruction Breakpoint

Register Mnemonic	Register Name and Description
<i>IBAn</i>	Instruction Breakpoint Address n
<i>IBMn</i>	Instruction Breakpoint Address Mask n
<i>IBASIDn</i>	Instruction Breakpoint ASID n
<i>IBCn</i>	Instruction Breakpoint Control n

11.2.4 Data Breakpoint Registers Overview

The register with implementation indication and status for data breakpoints in general is shown in [Table 11.4](#).

Table 11.4 Overview of Status Register for Data Breakpoints

Register Mnemonic	Register Name and Description
<i>DBS</i>	Data Breakpoint Status

The two data breakpoints are numbered 0 and 1 for registers and breakpoints, and the number is indicated by n. The registers for each breakpoint are shown in [Table 11.5](#).

Table 11.5 Overview of Registers for Each Data Breakpoint

Register Mnemonic	Register Name and Description
<i>DBAn</i>	Data Breakpoint Address n
<i>DBMn</i>	Data Breakpoint Address Mask n
<i>DBASIDn</i>	Data Breakpoint ASID n
<i>DBCn</i>	Data Breakpoint Control n
<i>DBVn</i>	Data Breakpoint Value n

11.2.5 Conditions for Matching Breakpoints

A number of conditions must be fulfilled in order for a breakpoint to match on an executed instruction or a data transaction, and the conditions for matching instruction and data breakpoints are described below. The breakpoints only match for instructions executed in non-debug mode, thus never on instructions executed in debug mode.

The match of an enabled breakpoint can either generate a debug exception or a trigger indication. The BE and/or TE bits in the *IBCn* or *DBCn* registers are used to enable the breakpoints.

Debug software should not configure breakpoints to compare on an ASID value unless a TLB is present in the implementation.

11.2.5.1 Conditions for Matching Instruction Breakpoints

When an instruction breakpoint is enabled, that breakpoint is evaluated for the address of every executed instruction in non-debug mode, including execution of instructions at an address causing an address error on an instruction fetch. The breakpoint is not evaluated on instructions from a speculative fetch or execution, nor for addresses which are unaligned with an executed instruction.

A breakpoint match depends on the virtual address of the executed instruction (PC) which can be masked at bit level, and match also can include an optional compare of ASID. The registers for each instruction breakpoint have the values and mask used in the compare, and the equation that determines the match is shown below in C-like notation.

```

IB_match =
    ( ! IBCn_ASIDuse || ( ASID == IBASIDn_ASID ) ) &&
    ( <all 1's> == ( IBMn_IBM | ~ ( PC ^ IBAn_IBA ) ) &&
      ( ( IBMn_ISAM | ~(ISAMode ^ IBAn_ISA) ) )

```

The match indication for instruction breakpoints is always precise, i.e. indicated on the instruction causing the *IB_match* to be true.

11.2.5.2 Conditions for Matching Data Breakpoints

When a data breakpoint is enabled, that breakpoint is evaluated for every data transaction due to a load/store instruction executed in non-debug mode, including load/store for coprocessor, and transactions causing an address error on data access. The breakpoint is not evaluated due to a PREF instruction or other transactions which are not part of explicit load/store transactions in the execution flow, nor for addresses which are not the explicit load/store source or destination address.

A breakpoint match depends on the transaction type (TYPE) as load or store, the address, and optionally the data value of a transaction. The registers for each data breakpoint have the values and mask used in the compare, and the equation that determines the match is shown below in C-like notation.

The overall match equation is the DB_match.

```
DB_match =
    ( ( ( TYPE == load ) && ! DBCnNoLB ) ||
      ( ( TYPE == store ) && ! DBCnNoSB ) ) &&
    DB_addr_match && ( DB_no_value_compare || DB_value_match )
```

The match on the address part, DB_addr_match, depends on the virtual address of the transaction (ADDR), the ASID value, and the accessed bytes (BYTELANE) where BYTELANE[0] is 1 only if the byte at bits [7:0] on the bus is accessed, and BYTELANE[1] is 1 only if the byte at bits [15:8] is accessed, etc. The DB_addr_match is shown below.

```
DB_addr_match =
    ( ! DBCnASIDuse || ( ASID == DBASIDnASID ) ) &&
    ( <all 1's> == ( DBMnDBM | ~ ( ADDR ^ DBAnDBA ) ) ) &&
    ( <all 0's> != ( ~ BAI & BYTELANE ) )
```

The size of *DBCn*_{BAI} and BYTELANE is 8 bits. They are 8 bits to allow for data value matching on doubleword floating point loads and stores. For non-doubleword loads and stores, only the lower 4 bits will be used.

Data value compare is included in the match condition for the data breakpoint depending on the bytes (BYTELANE as described above) accessed by the transaction, and the contents of breakpoint registers. The DB_no_value_compare is shown below.

```
DB_no_value_compare =
    ( <all 1's> == ( DBCnBLM | DBCnBAI | ~ BYTELANE ) )
```

The size of *DBCn*_{BLM}, *DBCn*_{BAI} and BYTELANE is 8 bits.

In case a data value compare is required, DB_no_value_compare is false, then the data value from the data bus (DATA) is compared and masked with the registers for the data breakpoint. The endianness is not considered in these match equations for value, as the compare uses the data bus value directly, thus debug software is responsible for setup of the breakpoint corresponding with endianness.

```
DB_value_match =
    ( ( DATA[7:0] == DBVnDBV[7:0] ) || !BYTELANE[0] || DBCnBLM[0] || DBCnBAI[0] ) &&
    ( ( DATA[15:8] == DBVnDBV[15:8] ) || !BYTELANE[1] || DBCnBLM[1] || DBCnBAI[1] ) &&
    ( ( DATA[23:16] == DBVnDBV[23:16] ) || !BYTELANE[2] || DBCnBLM[2] || DBCnBAI[2] ) &&
    ( ( DATA[31:24] == DBVnDBV[31:24] ) || !BYTELANE[3] || DBCnBLM[3] || DBCnBAI[3] ) &&
    ( ( DATA[39:32] == DBVnDBV[39:32] ) || !BYTELANE[4] || DBCnBLM[4] || DBCnBAI[4] ) &&
    ( ( DATA[47:40] == DBVnDBV[47:40] ) || !BYTELANE[5] || DBCnBLM[5] || DBCnBAI[5] ) &&
    ( ( DATA[55:48] == DBVnDBV[55:48] ) || !BYTELANE[6] || DBCnBLM[6] || DBCnBAI[6] ) &&
    ( ( DATA[63:56] == DBVnDBV[63:56] ) || !BYTELANE[7] || DBCnBLM[7] || DBCnBAI[7] ) )
```

The match for a data breakpoint without value compare is always precise, since the match expression is fully evaluated at the time the load/store instruction is executed. A true DB_match can thereby be indicated on the very same instruction causing the DB_match to be true. The match for data breakpoints with value compare is always imprecise.

11.2.6 Debug Exceptions from Breakpoints

Instruction and data breakpoints may be set up to generate a debug exception when the match condition is true, as described below.

11.2.6.1 Debug Exception by Instruction Breakpoint

If the breakpoint is enabled by BE bit in the *IBCn* register, then a debug instruction break exception occurs if the IB_match equation is true. The corresponding BS[n] bit in the *IBS* register is set when the breakpoint generates the debug exception.

The debug instruction break exception is always precise, so the *DEPC* register and *DBD* bit in the *Debug* register point to the instruction that caused the IB_match equation to be true.

The instruction receiving the debug exception does not update any registers due to the instruction, nor does any load or store by that instruction occur. Thus a debug exception from a data breakpoint can not occur for instructions receiving a debug instruction break exception.

The debug handler usually returns to the instruction causing the debug instruction break exception, whereby the instruction is executed. Debug software is responsible for disabling the breakpoint when returning to the instruction, otherwise the debug instruction break exception reoccurs.

11.2.6.2 Debug Exception by Data Breakpoint

If the breakpoint is enabled by BE bit in the *DBCn* register, then a debug exception occurs when the DB_match condition is true. The corresponding BS[n] bit in the *DBS* register is set when the breakpoint generates the debug exception. A matching data breakpoint generates either a precise or imprecise debug exception

Debug Data Break Load/Store Exception as a Precise Debug Exception

A precise debug data break exception occurs when a data breakpoint without value compare indicates a match. In this case the *DEPC* register and *DBD* bit in the *Debug* register points to the instruction that caused the DB_match equation to be true.

The instruction causing the debug data break exception does not update any registers due to the instruction, and the following applies to the load or store transaction causing the debug exception:

- A store transaction is not allowed to complete the store to the memory system.
- A load transaction with no data value compare, i.e. where the DB_no_value_compare is true for the match, is not allowed to complete the load.

The result of this is that the load or store instruction causing the debug data break exception appears as not executed.

If both data breakpoints without and with data value compare would match the same transaction and generate a debug exception, then the rules shown in Table 11.6 apply with respect to updating the BS[n] bits.

Table 11.6 Rules for Update of BS Bits on Data Breakpoint Exceptions

Instruction	Breakpoints that Match		Update of BS Bits for Matching Data Breakpoints	
	Without Value Compare	With Value Compare	Without Value Compare	With Value Compare
Load/Store	One or more	None	BS bits set for all	(No matching breakpoints)
Load	One or more	One or more	BS bits set for all	Unchanged BS bits since load of data value does not occur so match of the breakpoint cannot be determined
Load	None	One or more	(No matching breakpoints)	BS bits set for all
Store	One or more	One or more	BS bits set for all	BS bits set for all
Store	None	One or more	(No matching breakpoints)	BS bits set for all

Any BS[n] bit set prior to the match and debug exception are kept set, since BS[n] bits are only cleared by debug software.

The debug handler usually returns to the instruction causing the debug data break exception, whereby the instruction is re-executed. Debug software is responsible for disabling breakpoints when returning to the instruction; otherwise the debug data break exception will reoccur.

Debug Data Break Load/Store Exception as an Imprecise Debug Exception

An Debug Data Break Load/Store Imprecise exception occurs when a data breakpoint indicates an imprecise match. Imprecise matches are generated when data value compare is used. In this case, the DEPC register and DBD bit in the Debug register point to an instruction later in the execution flow rather than at the load/store instruction that caused the DB_match equation to be true.

The load/store instruction causing the Debug Data Break Load/Store Imprecise exception always updates the destination register and completes the access to the external memory system. Therefore this load/store instruction is not re-executed on return from the debug handler, because the DEPC register and DDBSImpr bit do not point to that instruction.

Several imprecise data breakpoints can be pending at a given time, if the bus system supports multiple outstanding data accesses. The breakpoints are evaluated as the accesses finalize, and a Debug Data Break Load/Store Imprecise exception is generated only for the first one matching. Both the first and succeeding matches cause corresponding DDBSImpr bits and DDBLImpr/DDBSImpr in the Debug register to be set, but no debug exception is generated for succeeding matches because the processor is already in Debug Mode. Similarly, if a debug exception had already occurred at the time of the first match (for example, due to a precise debug exception), then all matches cause the corresponding BS bits and DDBLImpr/DDBSImpr to be set, but no debug exception is generated because the processor is already in Debug Mode.

The SYNC instruction, followed by appropriate spacing must be executed before the *DDBSImpr* bits and *DDBSImpr/DDBSImpr* bits are accessed for read or write. This delay ensures that these bits are fully updated.

Any *DDBSImpr* bit set prior to the match and debug exception are kept set, because only debug software can clear the *DDBSImpr* bits.

11.2.7 Breakpoint used as TriggerPoint

Both instruction and data hardware breakpoints can be setup by software so a matching breakpoint does not generate a debug exception, but only an indication through the *BS[n]* bit. The *TE* bit in the *IBCn* or *DBCn* register controls if an instruction or data breakpoint is used as a so-called triggerpoint. The triggerpoints are, like breakpoints, only compared for instructions executed in non-debug mode.

The *BS[n]* bit in the *IBS* or *DBS* register is set when the respective *IB_match* or *DB_match* bit is true.

11.2.8 Instruction Breakpoint Registers

The registers for instruction breakpoints are described below. These registers have implementation information and are used to set up the instruction breakpoints. All registers are in *drseg*, and the addresses are shown in [Table 11.7](#).

Table 11.7 Addresses for Instruction Breakpoint Registers

Offset in <i>drseg</i>	Register Mnemonic	Register Name and Description
0x1000	<i>IBS</i>	Instruction Breakpoint Status
0x1100 + n * 0x100	<i>IBAn</i>	Instruction Breakpoint Address n
0x1108 + n * 0x100	<i>IBMn</i>	Instruction Breakpoint Address Mask n
0x1110 + n * 0x100	<i>IBASIDn</i>	Instruction Breakpoint ASID n
0x1118 + n * 0x100	<i>IBCn</i>	Instruction Breakpoint Control n
n is breakpoint number in range 0 to 3		

An example of some of the registers; *IBA0* is at offset 0x1100 and *IBC2* is at offset 0x1318.

11.2.8.1 Instruction Breakpoint Status (IBS) Register

Compliance Level: Implemented only if instruction breakpoints are implemented.

The *Instruction Breakpoint Status (IBS)* register holds implementation and status information about the instruction breakpoints.

The ASID applies to all the instruction breakpoints.

Figure 11.2 IBS Register Format



Table 11.8 IBS Register Field Descriptions

Fields		Description	Read / Write	Reset State						
Name	Bit(s)									
Res	31	Must be written as zero; returns zero on read.	R	0						
<i>ASIDsup</i>	30	Indicates that ASID compare is supported in instruction breakpoints. <table border="1" data-bbox="527 504 1031 651"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No ASID compare.</td> </tr> <tr> <td>1</td> <td>ASID compare (<i>IBASIDn</i> register implemented).</td> </tr> </tbody> </table>	Encoding	Meaning	0	No ASID compare.	1	ASID compare (<i>IBASIDn</i> register implemented).	R	Fixed MMU - 0 TLB - 1
Encoding	Meaning									
0	No ASID compare.									
1	ASID compare (<i>IBASIDn</i> register implemented).									
Res	29:28	Must be written as zero; returns zero on read.	R	0						
<i>BCN</i>	27:24	Number of instruction breakpoints implemented.	R	4						
Res	23:4	Must be written as zero; returns zero on read.	R	0						

11.2.8.2 Instruction Breakpoint Address *n* (IBAn) Register

Compliance Level: Implemented only for implemented instruction breakpoints.

The *Instruction Breakpoint Address n (IBAn)* register has the address used in the condition for instruction breakpoint *n*.

Figure 11.3 IBAn Register Format

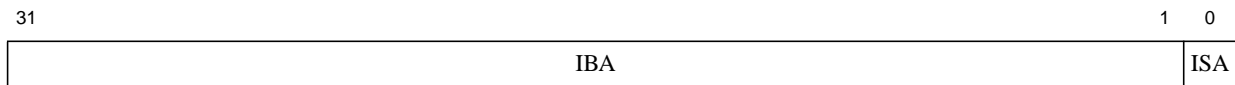


Table 11.9 IBAn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
<i>IBAn</i>	31:1	Instruction breakpoint address for condition.	R/W	Undefined
<i>ISA</i>	0	Instruction breakpoint ISA mode for condition	R/W	Undefined

11.2.8.3 Instruction Breakpoint Address Mask *n* (IBMn) Register

Compliance Level: Implemented only for implemented instruction breakpoints.

The *Instruction Breakpoint Address Mask n (IBMn)* register has the mask for the address compare used in the condition for instruction breakpoint *n*.

Figure 11.4 IBMn Register Format

31

1 0



Table 11.10 IBMn Register Field Descriptions

Fields		Description	Read / Write	Reset State						
Name	Bit(s)									
<i>IBM</i>	31:1	Instruction breakpoint address mask for condition: <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Corresponding address bit not masked.</td> </tr> <tr> <td>1</td> <td>Corresponding address bit masked.</td> </tr> </tbody> </table>	Encoding	Meaning	0	Corresponding address bit not masked.	1	Corresponding address bit masked.	R/W	Undefined
Encoding	Meaning									
0	Corresponding address bit not masked.									
1	Corresponding address bit masked.									
<i>ISAM</i>	0	Instruction breakpoint ISA mode mask for condition: <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>ISA mode considered for match condition</td> </tr> <tr> <td>1</td> <td>ISA mode masked</td> </tr> </tbody> </table>	Encoding	Meaning	0	ISA mode considered for match condition	1	ISA mode masked	R/W	Undefined
Encoding	Meaning									
0	ISA mode considered for match condition									
1	ISA mode masked									

11.2.8.4 Instruction Breakpoint ASID n (IBASIDn) Register

Compliance Level: Implemented only for implemented instruction breakpoints.

For processors with a TLB based MMU, this register is used to define an ASID value to be used in the match expression. For cores with an FM MMU, this register is reserved and reads as 0.

Figure 11.5 IBASIDn Register Format

31

8 7

0



Table 11.11 IBASIDn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
Res	31:8	Must be written as zero; returns zero on read.	R	0
<i>ASID</i>	7:0	Instruction breakpoint ASID value for a compare.	R/W	Undefined

11.2.8.5 Instruction Breakpoint Control n (IBCN) Register

Compliance Level: Implemented only for implemented instruction breakpoints.

The *Instruction Breakpoint Control n (IBCN)* register controls the setup of instruction breakpoint *n*.

Figure 11.6 IBCn Register Format



Table 11.12 BCn Register Field Descriptions

Fields		Description	Read / Write	Reset State						
Name	Bits									
Res	31:24	Must be written as zero; returns zero on read.	R	0						
<i>ASIDuse</i>	23	Use ASID value in compare for instruction breakpoint <i>n</i> : <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Don't use ASID value in compare</td> </tr> <tr> <td>1</td> <td>Use ASID value in compare</td> </tr> </tbody> </table>	Encoding	Meaning	0	Don't use ASID value in compare	1	Use ASID value in compare	R/W	Undefined
Encoding	Meaning									
0	Don't use ASID value in compare									
1	Use ASID value in compare									
Res	22	Must be written as zero; returns zero on read.	R	0						
Res	21:3	Must be written as zero; returns zero on read.	R	0						
<i>TE</i>	2	Use instruction breakpoint <i>n</i> as triggerpoint: <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Don't use it as triggerpoint</td> </tr> <tr> <td>1</td> <td>Use it as triggerpoint</td> </tr> </tbody> </table>	Encoding	Meaning	0	Don't use it as triggerpoint	1	Use it as triggerpoint	R/W	0
Encoding	Meaning									
0	Don't use it as triggerpoint									
1	Use it as triggerpoint									
Res	1	Must be written as zero; returns zero on read.	R	0						
<i>BE</i>	0	Use instruction breakpoint <i>n</i> as breakpoint: <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Don't use it as breakpoint</td> </tr> <tr> <td>1</td> <td>Use it as breakpoint</td> </tr> </tbody> </table>	Encoding	Meaning	0	Don't use it as breakpoint	1	Use it as breakpoint	R/W	0
Encoding	Meaning									
0	Don't use it as breakpoint									
1	Use it as breakpoint									

11.2.9 Data Breakpoint Registers

The registers for data breakpoints are described below. These registers have implementation information and are used the setup the data breakpoints. All registers are in drseg, and the addresses are shown in [Table 11.13](#).

Table 11.13 Addresses for Data Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
0x2000	<i>DBS</i>	Data Breakpoint Status
0x2100 + 0x100 * n	<i>DBAn</i>	Data Breakpoint Address n
0x2108 + 0x100 * n	<i>DBMn</i>	Data Breakpoint Address Mask n
0x2110 + 0x100 * n	<i>DBASIDn</i>	Data Breakpoint ASID n
0x2118 + 0x100 * n	<i>DBCn</i>	Data Breakpoint Control n
0x2120 + 0x100 * n	<i>DBVn</i>	Data Breakpoint Value n
0x2124 + 0x100*n	<i>DBVHn</i>	Data Breakpoint Value High n

n is breakpoint number as 0 or 1

An example of some of the registers; *DBM0* is at offset 0x2108 and *DBV1* is at offset 0x2220.

11.2.9.1 Data Breakpoint Status (DBS) Register

Compliance Level: Implemented if data breakpoints are implemented.

The *Data Breakpoint Status (DBS)* register contains implementation and status information about the data breakpoints.

The *ASIDsup* field indicates whether ASID compares are supported.

Figure 11.7 DBS Register Format



Table 11.14 DBS Register Field Descriptions

Fields		Description	Read / Write	Reset State						
Name	Bit(s)									
Res	31	Must be written as zero; returns zero on read.	R	0						
<i>ASIDsup</i>	30	Indicates that ASID compares are supported in data breakpoints. <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">0</td> <td>Not supported</td> </tr> <tr> <td style="text-align: center;">1</td> <td>Supported</td> </tr> </tbody> </table>	Encoding	Meaning	0	Not supported	1	Supported	R	TLB MMU - 1 FM MMU - 0
Encoding	Meaning									
0	Not supported									
1	Supported									
Res	29:28	Must be written as zero; returns zero on read.	R	0						
<i>BCN</i>	27:24	Number of data breakpoints implemented.	R	2						
Res	23:2	Must be written as zero; returns zero on read.	R	0						

Table 11.14 DBS Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
<i>BS</i>	1:0	Break status for breakpoint n is at BS[n], with n from 0 to 1. The bit is set to 1 when the condition for the corresponding breakpoint has matched.	R/W0	Undefined

11.2.9.2 Data Breakpoint Address n (DBAn) Register

Compliance Level: Implemented only for implemented data breakpoints.

The *Data Breakpoint Address n (DBAn)* register has the address used in the condition for data breakpoint n.

Figure 11.8 DBAn Register Format

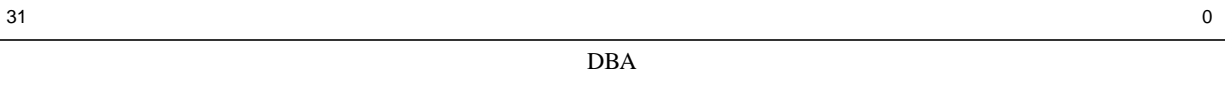


Table 11.15 DBAn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
<i>DBA</i>	31:0	Data breakpoint address for condition.	R/W	Undefined

11.2.9.3 Data Breakpoint Address Mask n (DBMn) Register

Compliance Level: Implemented only for implemented data breakpoints.

The *Data Breakpoint Address Mask n (DBMn)* register has the mask for the address compare used in the condition for data breakpoint n.

Figure 11.9 DBMn Register Format



Table 11.16 DBMn Register Field Descriptions

Fields		Description	Read / Write	Reset State						
Name	Bit(s)									
<i>DBM</i>	31:0	Data breakpoint address mask for condition: <table border="1" style="margin-left: 20px; width: 100%;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">0</td> <td>Corresponding address bit not masked</td> </tr> <tr> <td style="text-align: center;">1</td> <td>Corresponding address bit masked</td> </tr> </tbody> </table>	Encoding	Meaning	0	Corresponding address bit not masked	1	Corresponding address bit masked	R/W	Undefined
Encoding	Meaning									
0	Corresponding address bit not masked									
1	Corresponding address bit masked									

11.2.9.4 Data Breakpoint ASID n (DBASIDn) Register

Compliance Level: Implemented only for implemented data breakpoints.

For processors with a TLB based MMU, this register is used to define an ASID value to be used in the match expression. For cores with the FM MMU, this register is reserved and reads as 0.

Figure 11.10 DBASIDn Register Format

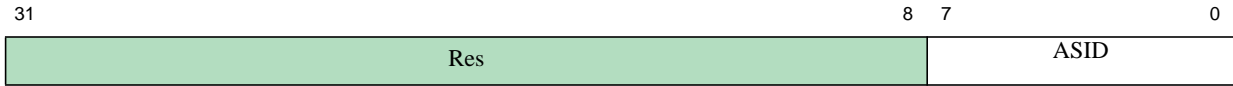


Table 11.17 DBASIDn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
Res	31:8	Must be written as zero; returns zero on read.	R	0
ASID	7:0	Data breakpoint ASID value for compares.	R/W	Undefined

11.2.9.5 Data Breakpoint Control n (DBCn) Register

Compliance Level: Implemented only for implemented data breakpoints.

The *Data Breakpoint Control n (DBCn)* register controls the setup of data breakpoint *n*.

Figure 11.11 DBCn Register Format

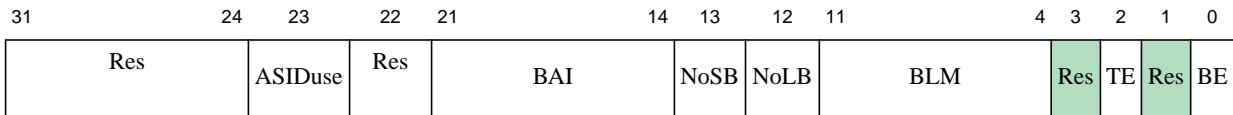


Table 11.18 DBCn Register Field Descriptions

Fields		Description	Read / Write	Reset State						
Name	Bits									
Res	31:24	Must be written as zero; returns zero on reads.	R	0						
ASIDuse	23	Use ASID value in compare for data breakpoint n: <table border="1" style="margin: 5px auto; border-collapse: collapse;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Don't use ASID value in compare</td> </tr> <tr> <td>1</td> <td>Use ASID value in compare</td> </tr> </tbody> </table>	Encoding	Meaning	0	Don't use ASID value in compare	1	Use ASID value in compare	R/W	Undefined
Encoding	Meaning									
0	Don't use ASID value in compare									
1	Use ASID value in compare									

Table 11.18 DBCn Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State						
Name	Bits									
Res	22	Must be written as zero; returns zero on reads	R	0						
<i>BAI</i>	21:14	Byte access ignore controls ignore of access to a specific byte. <i>BAI[0]</i> ignores access to byte at bits [7:0] of the data bus, <i>BAI[1]</i> ignores access to byte at bits [15:8], etc. <table border="1" data-bbox="542 646 1044 825"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Condition depends on access to corresponding byte</td> </tr> <tr> <td>1</td> <td>Access for corresponding byte is ignored</td> </tr> </tbody> </table>	Encoding	Meaning	0	Condition depends on access to corresponding byte	1	Access for corresponding byte is ignored	R/W	Undefined
Encoding	Meaning									
0	Condition depends on access to corresponding byte									
1	Access for corresponding byte is ignored									
<i>NoSB</i>	13	Controls if condition for data breakpoint is not fulfilled on a store transaction: <table border="1" data-bbox="542 909 1044 1087"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Condition may be fulfilled on store transaction</td> </tr> <tr> <td>1</td> <td>Condition is never fulfilled on store transaction</td> </tr> </tbody> </table>	Encoding	Meaning	0	Condition may be fulfilled on store transaction	1	Condition is never fulfilled on store transaction	R/W	Undefined
Encoding	Meaning									
0	Condition may be fulfilled on store transaction									
1	Condition is never fulfilled on store transaction									
<i>NoLB</i>	12	Controls if condition for data breakpoint is not fulfilled on a load transaction: <table border="1" data-bbox="542 1171 1044 1350"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Condition may be fulfilled on load transaction</td> </tr> <tr> <td>1</td> <td>Condition is never fulfilled on load transaction</td> </tr> </tbody> </table>	Encoding	Meaning	0	Condition may be fulfilled on load transaction	1	Condition is never fulfilled on load transaction	R/W	Undefined
Encoding	Meaning									
0	Condition may be fulfilled on load transaction									
1	Condition is never fulfilled on load transaction									
<i>BLM</i>	11:4	Byte lane mask for value compare on data breakpoint. <i>BLM[0]</i> masks byte at bits [7:0] of the data bus, <i>BLM[1]</i> masks byte at bits [15:8], etc.: <table border="1" data-bbox="542 1465 1044 1581"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Compare corresponding byte lane</td> </tr> <tr> <td>1</td> <td>Mask corresponding byte lane</td> </tr> </tbody> </table>	Encoding	Meaning	0	Compare corresponding byte lane	1	Mask corresponding byte lane	R/W	Undefined
Encoding	Meaning									
0	Compare corresponding byte lane									
1	Mask corresponding byte lane									
Res	3	Must be written as zero; returns zero on reads.	R	0						
<i>TE</i>	2	Use data breakpoint n as triggerpoint: <table border="1" data-bbox="542 1686 1044 1801"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Don't use it as triggerpoint</td> </tr> <tr> <td>1</td> <td>Use it as triggerpoint</td> </tr> </tbody> </table>	Encoding	Meaning	0	Don't use it as triggerpoint	1	Use it as triggerpoint	R/W	0
Encoding	Meaning									
0	Don't use it as triggerpoint									
1	Use it as triggerpoint									
Res	1	Must be written as zero; returns zero on reads.	R	0						

Table 11.18 DBCn Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State						
Name	Bits									
<i>BE</i>	0	Use data breakpoint n as breakpoint:	R/W	0						
		<table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Don't use it as breakpoint</td> </tr> <tr> <td>1</td> <td>Use it as breakpoint</td> </tr> </tbody> </table>	Encoding	Meaning	0	Don't use it as breakpoint	1	Use it as breakpoint		
Encoding	Meaning									
0	Don't use it as breakpoint									
1	Use it as breakpoint									

11.2.9.6 Data Breakpoint Value n (DBVn) Register

Compliance Level: Implemented only for implemented data breakpoints.

The *Data Breakpoint Value n (DBVn)* register has the value used in the condition for data breakpoint n.

Figure 11.12 DBVn Register Format

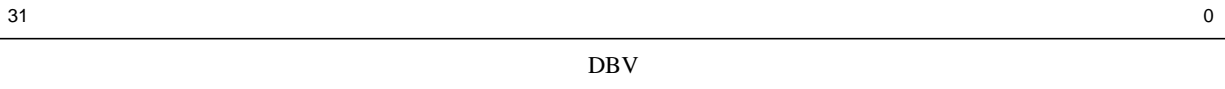


Table 11.19 DBVn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
<i>DBV</i>	31:0	Data breakpoint value for condition.	R/W	Undefined

11.2.9.7 Data Breakpoint Value High n (DBVHn) Register

Compliance Level: Implemented only for implemented data breakpoints.

The *Data Breakpoint Value High n (DBVHn)* register has the value used in the condition for data breakpoint n.

Figure 11.13 DBVHn Register Format



Table 11.20 DBVHn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
<i>DBVH</i>	31:0	Data breakpoint value high for condition. This register provides the high order bits [63:32] for data value on double-word floating point loads and stores.	R/W	Undefined

11.3 Test Access Port (TAP)

The following main features are supported by the TAP module:

- 5-pin industry standard JTAG Test Access Port (*TCK*, *TMS*, *TDI*, *TDO*, *TRST_N*) interface which is compatible with IEEE Std. 1149.1.
- Target chip and EJTAG feature identification available through the Test Access Port (TAP) controller.
- The processor can access external memory on the EJTAG Probe serially through the EJTAG pins. This is achieved through Processor Access (PA), and is used to eliminate the use of the system memory for debug routines.
- Support for both ROM based debugger and debugging both through TAP.

11.3.1 EJTAG Internal and External Interfaces

The external interface of the EJTAG module consists of the 5 signals defined by the IEEE standard.

Table 11.21 EJTAG Interface Pins

Pin	Type	Description
<i>TCK</i>	I	Test Clock Input Input clock used to shift data into or out of the Instruction or data registers. The <i>TCK</i> clock is independent of the processor clock, so the EJTAG probe can drive <i>TCK</i> independently of the processor clock frequency. The core signal for this is called <i>EJ_TCK</i>
<i>TMS</i>	I	Test Mode Select Input The <i>TMS</i> input signal is decoded by the TAP controller to control test operation. <i>TMS</i> is sampled on the rising edge of <i>TCK</i> . The core signal for this is called <i>EJ_TMS</i>
<i>TDI</i>	I	Test Data Input Serial input data (<i>TDI</i>) is shifted into the Instruction register or data registers on the rising edge of the <i>TCK</i> clock, depending on the TAP controller state. The core signal for this is called <i>EJ_TDI</i>
<i>TDO</i>	O	Test Data Output Serial output data is shifted from the Instruction or data register to the <i>TDO</i> pin on the falling edge of the <i>TCK</i> clock. When no data is shifted out, the <i>TDO</i> is 3-stated. The core signal for this is called <i>EJ_TDO</i> with output enable controlled by <i>EJ_TDOzstate</i> .

Table 11.21 EJTAG Interface Pins (Continued)

Pin	Type	Description
<i>TRST_N</i>	I	<p>Test Reset Input (Optional pin)</p> <p>The <i>TRST_N</i> pin is an active-low signal for asynchronous reset of the TAP controller and instruction in the TAP module, independent of the processor logic. The processor is not reset by the assertion of <i>TRST_N</i>.</p> <p>The core signal for this is called <i>EJ_TRST_N</i></p> <p>This signal is optional, but power-on reset must apply a low pulse on this signal at power-on and then leave it high, in case the signal is not available as a pin on the chip. If available on the chip, then it must be low on the board when the EJTAG debug features are unused by the probe.</p>

11.3.2 Test Access Port Operation

The TAP controller is controlled by the Test Clock (*TCK*) and Test Mode Select (*TMS*) inputs. These two inputs determine whether an instruction register scan or data register scan is performed. The TAP consists of a small controller, driven by the *TCK* input, which responds to the *TMS* input as shown in the state diagram in Figure 11.14. The TAP uses both clock edges of *TCK*. *TMS* and *TDI* are sampled on the rising edge of *TCK*, while *TDO* changes on the falling edge of *TCK*.

At power-up, the TAP is forced into the *Test-Logic-Reset* state by a low value on *TRST_N*. The TAP instruction register is thereby reset to IDCODE. No other parts of the EJTAG hardware are reset through the *Test-Logic-Reset* state.

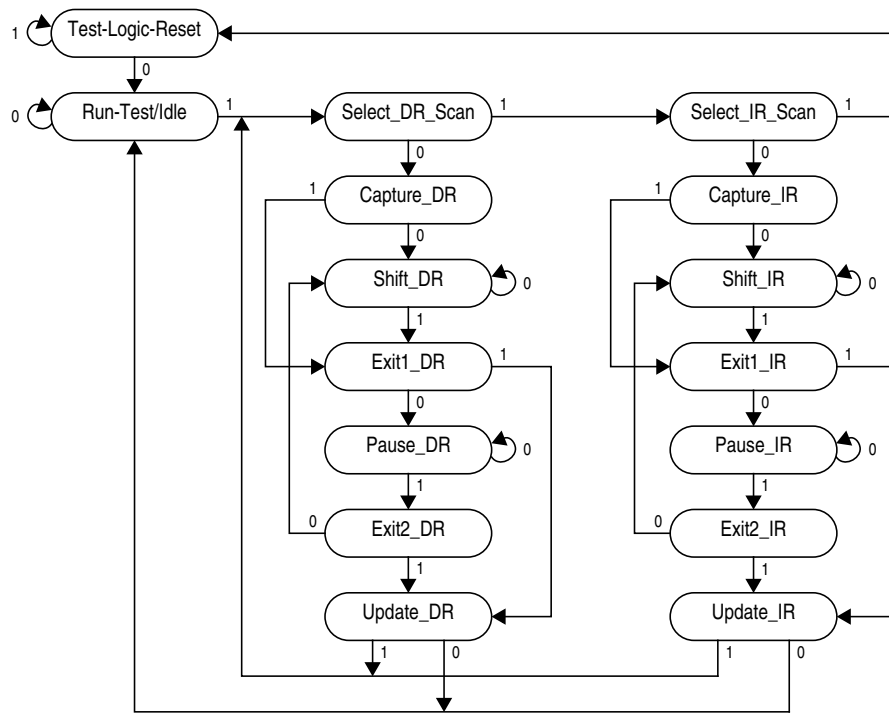
When test access is required, a protocol is applied via the *TMS* and *TCK* inputs, causing the TAP to exit the *Test-Logic-Reset* state and move through the appropriate states. From the *Run-Test/Idle* state, an Instruction register scan or a data register scan can be issued to transition the TAP through the appropriate states shown in Figure 11.14.

The states of the data and instruction register scan blocks are mirror images of each other, adding symmetry to the protocol sequences. The first action that occurs when either block is entered is a capture operation. For the data registers, the *Capture-DR* state is used to capture (or parallel load) the data into the selected serial data path. In the Instruction register, the *Capture-IR* state is used to capture status information into the Instruction register.

From the *Capture* states, the TAP transitions to either the *Shift* or *Exit1* states. Normally the *Shift* state follows the *Capture* state so that test data or status information can be shifted out for inspection and new data shifted in. Following the *Shift* state, the TAP either returns to the *Run-Test/Idle* state via the *Exit1* and *Update* states or enters the *Pause* state via *Exit1*. The reason for entering the *Pause* state is to temporarily suspend the shifting of data through either the Data or Instruction Register while a required operation, such as refilling a host memory buffer, is performed. From the *Pause* state shifting can resume by re-entering the *Shift* state via the *Exit2* state or terminate by entering the *Run-Test/Idle* state via the *Exit2* and *Update* states.

Upon entering the data or Instruction register scan blocks, shadow latches in the selected scan path are forced to hold their present state during the Capture and Shift operations. The data being shifted into the selected scan path is not output through the shadow latch until the TAP enters the *Update-DR* or *Update-IR* state. The *Update* state causes the shadow latches to update (or parallel load) with the new data that has been shifted into the selected scan path.

Figure 11.14 TAP Controller State Diagram



11.3.2.1 Test-Logic-Reset State

In the *Test-Logic-Reset* state the boundary scan test logic is disabled. The test logic enters the *Test-Logic-Reset* state when the *TMS* input is held HIGH for at least five rising edges of *TCK*. The BYPASS instruction is forced into the instruction register output latches during this state. The controller remains in the *Test-Logic-Reset* state as long as *TMS* is HIGH.

11.3.2.2 Run-Test/Idle State

The controller enters the *Run-Test/Idle* state between scan operations. The controller remains in this state as long as *TMS* is held LOW. The instruction register and all test data registers retain their previous state. The instruction cannot change when the TAP controller is in this state.

When *TMS* is sampled HIGH on the rising edge of *TCK*, the controller transitions to the *Select_DR* state.

11.3.2.3 Select_DR_Scan State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, then the controller transitions to the *Capture_DR* state. A HIGH on *TMS* causes the controller to transition to the *Select_IR* state. The instruction cannot change while the TAP controller is in this state.

11.3.2.4 Select_IR_Scan State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW on the rising edge of *TCK*, the controller transitions to the *Capture_IR* state. A

HIGH on *TMS* causes the controller to transition to the *Test-Reset-Logic* state. The instruction cannot change while the TAP controller is in this state.

11.3.2.5 Capture_DR State

In this state the boundary scan register captures the value of the register addressed by the Instruction register, and the value is then shifted out in the *Shift_DR*. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Shift_DR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1_DR* state. The instruction cannot change while the TAP controller is in this state.

11.3.2.6 Shift_DR State

In this state the test data register connected between *TDI* and *TDO* as a result of the current instruction shifts data one stage toward its serial output on the rising edge of *TCK*. If *TMS* is sampled LOW on the rising edge of *TCK*, the controller remains in the *Shift_DR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1_DR* state. The instruction cannot change while the TAP controller is in this state.

11.3.2.7 Exit1_DR State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Pause_DR* state. A HIGH on *TMS* causes the controller to transition to the *Update_DR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

11.3.2.8 Pause_DR State

The *Pause_DR* state allows the controller to temporarily halt the shifting of data through the test data register in the serial path between *TDI* and *TDO*. All test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW on the rising edge of *TCK*, the controller remains in the *Pause_DR* state. A HIGH on *TMS* causes the controller to transition to the *Exit2_DR* state. The instruction cannot change while the TAP controller is in this state.

11.3.2.9 Exit2_DR State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Shift_DR* state to allow another serial shift of data. A HIGH on *TMS* causes the controller to transition to the *Update_DR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

11.3.2.10 Update_DR State

When the TAP controller is in this state the value shifted in during the *Shift_DR* state takes effect on the rising edge of the *TCK* for the register indicated by the Instruction register.

If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Run-Test/Idle* state. A HIGH on *TMS* causes the controller to transition to the *Select_DR_Scan* state. The instruction cannot change while the TAP controller is in this state and all shift register stages in the test data registers selected by the current instruction retain their previous state.

11.3.2.11 Capture_IR State

In this state the shift register contained in the Instruction register loads a fixed pattern (00001₂) on the rising edge of *TCK*. The data registers selected by the current instruction retain their previous state.

If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Shift_IR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1_IR* state. The instruction cannot change while the TAP controller is in this state.

11.3.2.12 Shift_IR State

In this state the instruction register is connected between *TDI* and *TDO* and shifts data one stage toward its serial output on the rising edge of *TCK*. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller remains in the *Shift_IR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1_IR* state.

11.3.2.13 Exit1_IR State

This is a temporary controller state in which all registers retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Pause_IR* state. A HIGH on *TMS* causes the controller to transition to the *Update_IR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state and the instruction register retains its previous state.

11.3.2.14 Pause_IR State

The *Pause_IR* state allows the controller to temporarily halt the shifting of data through the instruction register in the serial path between *TDI* and *TDO*. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller remains in the *Pause_IR* state. A HIGH on *TMS* causes the controller to transition to the *Exit2_IR* state. The instruction cannot change while the TAP controller is in this state.

11.3.2.15 Exit2_IR State

This is a temporary controller state in which the instruction register retains its previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, then the controller transitions to the *Shift_IR* state to allow another serial shift of data. A HIGH on *TMS* causes the controller to transition to the *Update_IR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

11.3.2.16 Update_IR State

The instruction shifted into the instruction register takes effect on the rising edge of *TCK*.

If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Run-Test/Idle* state. A HIGH on *TMS* causes the controller to transition to the *Select_DR_Scan* state.

11.3.3 Test Access Port (TAP) Instructions

The TAP Instruction register allows instructions to be serially input into the device when TAP controller is in the *Shift-IR* state. Instructions are decoded and define the serial test data register path that is used to shift data between *TDI* and *TDO* during data register scanning.

The Instruction register is a 5-bit register. In the current EJTAG implementation only some instructions have been decoded; the unused instructions default to the BYPASS instruction.

Table 11.22 Implemented EJTAG Instructions

Value	Instruction	Function
0x01	IDCODE	Select Chip Identification data register
0x03	IMPCODE	Select Implementation register
0x08	ADDRESS	Select Address register

Table 11.22 Implemented EJTAG Instructions (Continued)

Value	Instruction	Function
0x09	DATA	Select Data register
0x0A	CONTROL	Select EJTAG Control register
0x0B	ALL	Select the Address, Data and EJTAG Control registers
0x0C	EJTAGBOOT	Set <i>EjtagBrk</i> , <i>ProbEn</i> and <i>ProbTrap</i> to 1 as reset value
0x0D	NORMALBOOT	Set <i>EjtagBrk</i> , <i>ProbEn</i> and <i>ProbTrap</i> to 0 as reset value
0x0E	FASTDATA	Selects the <i>Data</i> and <i>Fastdata</i> registers
0x10	TCBCONTROLA	Selects the <i>TCBTCONTROLA</i> register in the Trace Control Block
0x11	TCBCONTROLB	Selects the <i>TCBTCONTROLB</i> register in the Trace Control Block
0x12	TCBDATA	Selects the <i>TCBDATA</i> register in the Trace Control Block
0x13	TCBCONTROLC	Selects the <i>TCBTCONTROLC</i> register in the Trace Control Block
0x14	PCSAMPLE	Selects the <i>PCSAMPLE</i> register
0x16	TCBCONTROLE	Selects the <i>TCBTCONTROLE</i> register in the Trace Control Block
0x17	FDC	Select Fast Debug Channel
0x1F	BYPASS	Bypass mode

11.3.3.1 BYPASS Instruction

The required BYPASS instruction allows the processor to remain in a functional mode and selects the *Bypass* register to be connected between *TDI* and *TDO*. The BYPASS instruction allows serial data to be transferred through the processor from *TDI* to *TDO* without affecting its operation. The bit code of this instruction is defined to be all ones by the IEEE 1149.1 standard. Any unused instruction is defaulted to the BYPASS instruction.

11.3.3.2 IDCODE Instruction

The IDCODE instruction allows the processor to remain in its functional mode and selects the Device Identification (ID) register to be connected between *TDI* and *TDO*. The *Device ID* register is a 32-bit shift register containing information regarding the IC manufacturer, device type, and version code. Accessing the Identification Register does not interfere with the operation of the processor. Also, access to the *Identification* Register is immediately available, via a TAP data scan operation, after power-up when the TAP has been reset with on-chip power-on or through the optional *TRST_N* pin.

11.3.3.3 IMPCODE Instruction

This instruction selects the Implementation register for output, which is always 32 bits.

11.3.3.4 ADDRESS Instruction

This instruction is used to select the *Address* register to be connected between *TDI* and *TDO*. The EJTAG Probe shifts 32 bits through the *TDI* pin into the *Address* register and shifts out the captured address via the *TDO* pin.

11.3.3.5 DATA Instruction

This instruction is used to select the *Data* register to be connected between *TDI* and *TDO*. The EJTAG Probe shifts 32 bits of *TDI* data into the *Data* register and shifts out the captured data via the *TDO* pin.

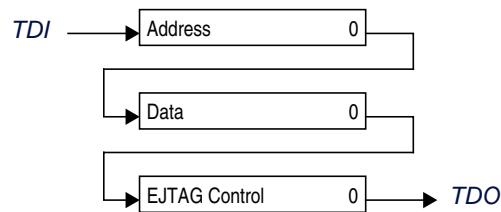
11.3.3.6 CONTROL Instruction

This instruction is used to select the *EJTAG Control* register to be connected between *TDI* and *TDO*. The EJTAG Probe shifts 32 bits of *TDI* data into the *EJTAG Control* register and shifts out the EJTAG Control register bits via *TDO*.

11.3.3.7 ALL Instruction

This instruction is used to select the concatenation of the Address and Data register, and the EJTAG Control register between *TDI* and *TDO*. It can be used in particular if switching instructions in the instruction register takes too many *TCK* cycles. The first bit shifted out is bit 0.

Figure 11.15 Concatenation of the EJTAG Address, Data and Control Registers



11.3.3.8 EJTAGBOOT Instruction

When the EJTAGBOOT instruction is given and the Update-IR state is left, then the reset values of the *ProbTrap*, *ProbEn* and *EjtagBrk* bits in the *EJTAG Control* register are set to 1 after a reset.

This EJTAGBOOT indication is effective until a NORMALBOOT instruction is given, *TRST_N* is asserted or a rising edge of *TCK* occurs when the TAP controller is in Test-Logic-Reset state.

It is possible to make the CPU go into debug mode just after a reset, without fetching or executing any instructions from the normal memory area. This can be used for download of code to a system which has no code in ROM.

The *Bypass* register is selected when the EJTAGBOOT instruction is given.

11.3.3.9 NORMALBOOT Instruction

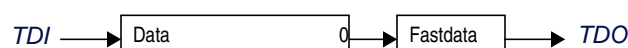
When the NORMALBOOT instruction is given and the Update-IR state is left, then the reset value of the *ProbTrap*, *ProbEn* and *EjtagBrk* bits in the *EJTAG Control* register are set to 0 after reset.

The *Bypass* register is selected when the NORMALBOOT instruction is given.

11.3.3.10 FASTDATA Instruction

This selects the Data and the Fastdata registers at once, as shown in [Figure 11.16](#).

Figure 11.16 TDI to TDO Path When in Shift-DR State and FASTDATA Instruction is Selected



11.3.3.11 TCBCONTROLA Instruction

This instruction is used to select the *TCBCONTROLA* register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace unit is present. If no TRU is present, this instruction will select the *Bypass* register.

11.3.3.12 TCBCONTROLB Instruction

This instruction is used to select the *TCBCONTROLB* register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace unit is present. If no TRU is present, then this instruction will select the *Bypass* register.

11.3.3.13 TCBCONTROLC Instruction

This instruction is used to select the *TCBCONTROLC* register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace unit is present. If no TRU is present, then this instruction will select the *Bypass* register.

11.3.3.14 TCBCONTROLE Instruction

This instruction is used to select the *TCBCONTROLE* register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace unit is present. If no TRU is present, then this instruction will select the *Bypass* register.

11.3.3.15 TCBDATA Instruction

This instruction is used to select the *TCBDATA* register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace unit is present. If no TRU is present, then this instruction will select the *Bypass* register. It should be noted that the *TCBDATA* register is only an access register to other TCB registers. The width of the *TCBDATA* register is dependent on the specific TCB register.

11.3.3.16 PCSAMPLE Instruction

This instruction is used to select the *PCSAMPLE* register to be connected between *TDI* and *TDO*. This register is always implemented.

11.3.3.17 FDC Instruction

This instruction is used to select the Fast Debug Channel register to be connected between *TDI* and *TDO*. This register is always implemented.

11.4 EJTAG TAP Registers

The EJTAG TAP Module has one Instruction register and a number of Data registers, all of which are accessible through the TAP:

11.4.1 Instruction Register

The Instruction register is accessed when the TAP receives an Instruction register scan protocol. During an Instruction register scan operation, the TAP controller selects the output of the Instruction register to drive the *TDO* pin. The shift register consists of a series of bits arranged to form a single scan path between *TDI* and *TDO*. During an Instruction register scan operations, the TAP controls the register to capture status information and shift data from *TDI* to *TDO*. Both the capture and shift operations occur on the rising edge of *TCK*. However, the data shifted out from the *TDO* occurs on the falling edge of *TCK*. In the Test-Logic-Reset and *Capture-IR* state, the instruction shift register is set to

00001₂, as for the IDCODE instruction. This forces the device into the functional mode and selects the Device ID register. The Instruction register is 5 bits wide. The instruction shifted in takes effect for the following data register scan operation. A list of the implemented instructions are listed in [Table 11.22](#).

11.4.2 Data Registers Overview

The EJTAG uses several data registers, which are arranged in parallel from the primary *TDI* input to the primary *TDO* output. The Instruction register supplies the address that allows one of the data registers to be accessed during a data register scan operation. During a data register scan operation, the addressed scan register receives TAP control signals to capture the register and shift data from *TDI* to *TDO*. During a data register scan operation, the TAP selects the output of the data register to drive the *TDO* pin. The register is updated in the *Update-DR* state with respect to the write bits.

The description above applies in general to the following EJTAG data registers:

- *Bypass*
- *Device Identification*
- *Implementation*
- *EJTAG Control (ECR)*
- *Processor Access Address*
- *Processor Access Data*
- *FastData*

11.4.2.1 Bypass Register

The *Bypass* register consists of a single scan register bit. When selected, the *Bypass* register provides a single bit scan path between *TDI* and *TDO*. The *Bypass* register allows abbreviating the scan path through devices that are not involved in the test. The *Bypass* register is selected when the *Instruction* register is loaded with a pattern of all ones to satisfy the IEEE 1149.1 Bypass instruction requirement.

11.4.2.2 Device Identification (ID) Register

The *Device Identification* register is defined by IEEE 1149.1, to identify the device's manufacturer, part number, revision, and other device-specific information. [Table 11.23](#) shows the bit assignments defined for the read-only *ID* register; inputs to the core determine the value of these bits. These bits can be scanned out of the *ID* register after being selected. The register is selected when the Instruction register is loaded with the IDCODE instruction.

Figure 11.17 Device Identification Register Format



Table 11.23 Device Identification Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
<i>Version</i>	31:28	Version (4 bits) This field identifies the version number of the processor derivative.	R	<i>EJ_Version[3:0]</i>
<i>PartNumber</i>	27:12	Part Number (16 bits) This field identifies the part number of the processor derivative.	R	<i>EJ_PartNumber[15:0]</i>
<i>ManufID</i>	11:1	Manufacturer Identity (11 bits) Accordingly to IEEE 1149.1-1990, the manufacturer identity code shall be a compressed form of the JEDEC Publications 106-A.	R	<i>EJ_ManufID[10:0]</i>
R	0	Reserved	R	1

11.4.2.3 Implementation Register

This 32-bit read-only register is used to identify the features of the EJTAG implementation. Some of the reset values are set by inputs to the core. The register is selected when the Instruction register is loaded with the IMPCODE instruction.

Figure 11.18 Implementation Register Format

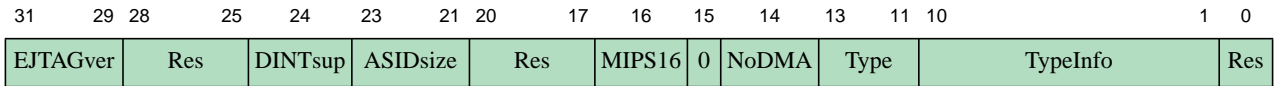


Table 11.24 Implementation Register Descriptions

Fields		Description	Read / Write	Reset State								
Name	Bit(s)											
<i>EJTAGver</i>	31:29	EJTAG Version 4.14	R	3								
Res	28:25	Reserved	R	0								
<i>DINTsup</i>	24	DINT Signal Supported from Probe This bit indicates if the DINT signal from the probe is supported: <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>DINT signal from the probe is not supported</td> </tr> <tr> <td>1</td> <td>Probe can use DINT signal to make debug interrupt.</td> </tr> </tbody> </table>	Encoding	Meaning	0	DINT signal from the probe is not supported	1	Probe can use DINT signal to make debug interrupt.	R	<i>EJ_DINTsup</i>		
Encoding	Meaning											
0	DINT signal from the probe is not supported											
1	Probe can use DINT signal to make debug interrupt.											
<i>ASIDsize</i>	23:21	Size of ASID field in implementation: <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No ASID in implementation</td> </tr> <tr> <td>1</td> <td>8-bit ASID</td> </tr> <tr> <td>1,3</td> <td>Reserved</td> </tr> </tbody> </table>	Encoding	Meaning	0	No ASID in implementation	1	8-bit ASID	1,3	Reserved	R	TLB MMU- 2 FM MMU- 0
Encoding	Meaning											
0	No ASID in implementation											
1	8-bit ASID											
1,3	Reserved											

Table 11.24 Implementation Register Descriptions (Continued)

Fields		Description	Read / Write	Reset State										
Name	Bit(s)													
Res	20:17	Reserved	R	0										
<i>MIPS16</i>	16	Indicates whether MIPS16 is implemented: <table border="1" data-bbox="542 426 1044 543"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No MIPS16 support</td> </tr> <tr> <td>1</td> <td>MIPS16 implemented</td> </tr> </tbody> </table>	Encoding	Meaning	0	No MIPS16 support	1	MIPS16 implemented	R	1				
Encoding	Meaning													
0	No MIPS16 support													
1	MIPS16 implemented													
Res	15	Reserved	R	0										
<i>NoDMA</i>	14	No EJTAG DMA Support	R	1										
Type	13:11	Indicates what type of entity is associated with this TAP and whether the TypeInfo field exists. <table border="1" data-bbox="542 741 1044 1079"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>TypeInfo field not implemented. Legacy value - probably attached to a CPU.</td> </tr> <tr> <td>1</td> <td>This TAP is attached to a CPU and the TypeInfo field reflects <i>EBase_{CPUNUM}</i>.</td> </tr> <tr> <td>2</td> <td>This TAP is attached to a Trace-Master and the TypeInfo field is not used.</td> </tr> <tr> <td>Others</td> <td>Reserved</td> </tr> </tbody> </table> Tied to b'01 for this core.	Encoding	Meaning	0	TypeInfo field not implemented. Legacy value - probably attached to a CPU.	1	This TAP is attached to a CPU and the TypeInfo field reflects <i>EBase_{CPUNUM}</i> .	2	This TAP is attached to a Trace-Master and the TypeInfo field is not used.	Others	Reserved	R	Preset
Encoding	Meaning													
0	TypeInfo field not implemented. Legacy value - probably attached to a CPU.													
1	This TAP is attached to a CPU and the TypeInfo field reflects <i>EBase_{CPUNUM}</i> .													
2	This TAP is attached to a Trace-Master and the TypeInfo field is not used.													
Others	Reserved													
TypeInfo	10:1	Identifier information specific to the type of entity associated with this TAP. The attached entity is specified by the Type field. <table border="1" data-bbox="542 1260 1044 1436"> <thead> <tr> <th>Attached Entity</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>CPU</td> <td>Reflects <i>EBase_{CPUNUM}</i> of the associated CPU.</td> </tr> <tr> <td>others</td> <td>Reserved</td> </tr> </tbody> </table>	Attached Entity	Meaning	CPU	Reflects <i>EBase_{CPUNUM}</i> of the associated CPU.	others	Reserved	R	Preset				
Attached Entity	Meaning													
CPU	Reflects <i>EBase_{CPUNUM}</i> of the associated CPU.													
others	Reserved													
Res	0	Reserved	R	0										

11.4.2.4 EJTAG Control Register

This 32-bit register controls the various operations of the TAP modules. This register is selected by executing the CONTROL instruction. Bits in the *EJTAG Control* register can be set/cleared by shifting in data; status is read by shifting out the contents of this register. This *EJTAG Control* register can only be accessed by the TAP interface.

The *EJTAG Control* register is not updated in the *Update-DR* state unless the Reset occurred (*Rocc* bit is either 0 or written to 0). This is in order to ensure proper handling of processor accesses.

EJTAG Debug Support in the 74K™ Core

The value used for reset indicated in the table below takes effect on CPU resets, but not on TAP controller resets by e.g., *TRST_N*. *TCK* clock is not required when the CPU reset occurs, but the bits are still updated to the reset value when the *TCK* applies. The first 5 *TCK* clocks after CPU resets may result in reset of the bits, due to synchronization between clock domains.

Figure 11.19 EJTAG Control Register Format

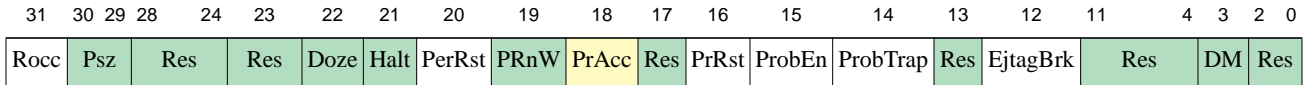


Table 11.25 EJTAG Control Register Descriptions

Fields		Description	Read / Write	Reset State						
Name	Bit(s)									
<i>Rocc</i>	31	<p>Reset Occurred</p> <p>The bit indicates if a CPU reset has occurred:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No reset occurred since bit last cleared.</td> </tr> <tr> <td>1</td> <td>Reset occurred since bit last cleared.</td> </tr> </tbody> </table> <p>The <i>Rocc</i> bit will keep the 1 value as long as reset is applied. This bit must be cleared by the probe, to acknowledge that the incident was detected.</p> <p>The <i>EJTAG Control</i> register is not updated in the <i>Update-DR</i> state unless <i>Rocc</i> is 0, or written to 0. This is in order to ensure proper handling of processor access.</p>	Encoding	Meaning	0	No reset occurred since bit last cleared.	1	Reset occurred since bit last cleared.	R/W	1
Encoding	Meaning									
0	No reset occurred since bit last cleared.									
1	Reset occurred since bit last cleared.									

Table 11.25 EJTAG Control Register Descriptions (Continued)

Fields		Description	Read / Write	Reset State																																	
Name	Bit(s)																																				
<i>Psz[1:0]</i>	30:29	<p>Processor Access Transfer Size</p> <p>These bits are used in combination with the lower two address bits of the Address register to determine the size of a processor access transaction. The bits are only valid when processor access is pending.</p> <table border="1"> <thead> <tr> <th>PAA[1:0]</th> <th>Psz[1:0]</th> <th>Transfer Size</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>00</td> <td>Byte (LE, byte 0; BE, byte 3)</td> </tr> <tr> <td>01</td> <td>00</td> <td>Byte (LE, byte 1; BE, byte 2)</td> </tr> <tr> <td>10</td> <td>00</td> <td>Byte (LE, byte 2; BE, byte 1)</td> </tr> <tr> <td>11</td> <td>00</td> <td>Byte (LE, byte 3; BE, byte 0)</td> </tr> <tr> <td>00</td> <td>01</td> <td>Halfword (LE, bytes 1:0; BE, bytes 3:2)</td> </tr> <tr> <td>10</td> <td>01</td> <td>Halfword (LE, bytes 3:2; BE, bytes 1:0)</td> </tr> <tr> <td>00</td> <td>10</td> <td>Word (LE, BE; bytes 3, 2, 1, 0)</td> </tr> <tr> <td>00</td> <td>11</td> <td>Triple (LE, bytes 2, 1, 0; BE, bytes 3, 2, 1)</td> </tr> <tr> <td>01</td> <td>11</td> <td>Triple (LE, bytes 3, 2, 1; BE, bytes 2, 1, 0)</td> </tr> <tr> <td colspan="2">All others</td> <td>Reserved</td> </tr> </tbody> </table> <p>Note: LE=little endian, BE=big endian, the byte# refers to the byte number in a 32-bit register, where byte 3 = bits 31:24; byte 2 = bits 23:16; byte 1 = bits 15:8; byte 0=bits 7:0, independently of the endianness.</p>	PAA[1:0]	Psz[1:0]	Transfer Size	00	00	Byte (LE, byte 0; BE, byte 3)	01	00	Byte (LE, byte 1; BE, byte 2)	10	00	Byte (LE, byte 2; BE, byte 1)	11	00	Byte (LE, byte 3; BE, byte 0)	00	01	Halfword (LE, bytes 1:0; BE, bytes 3:2)	10	01	Halfword (LE, bytes 3:2; BE, bytes 1:0)	00	10	Word (LE, BE; bytes 3, 2, 1, 0)	00	11	Triple (LE, bytes 2, 1, 0; BE, bytes 3, 2, 1)	01	11	Triple (LE, bytes 3, 2, 1; BE, bytes 2, 1, 0)	All others		Reserved	R	Undefined
PAA[1:0]	Psz[1:0]	Transfer Size																																			
00	00	Byte (LE, byte 0; BE, byte 3)																																			
01	00	Byte (LE, byte 1; BE, byte 2)																																			
10	00	Byte (LE, byte 2; BE, byte 1)																																			
11	00	Byte (LE, byte 3; BE, byte 0)																																			
00	01	Halfword (LE, bytes 1:0; BE, bytes 3:2)																																			
10	01	Halfword (LE, bytes 3:2; BE, bytes 1:0)																																			
00	10	Word (LE, BE; bytes 3, 2, 1, 0)																																			
00	11	Triple (LE, bytes 2, 1, 0; BE, bytes 3, 2, 1)																																			
01	11	Triple (LE, bytes 3, 2, 1; BE, bytes 2, 1, 0)																																			
All others		Reserved																																			
Res	28:24	Reserved	R	0																																	
Res	23	Reserved	R	0																																	
<i>Doze</i>	22	<p>Doze state</p> <p>The Doze bit indicates any kind of low-power mode. The value is sampled in the Capture-DR state of the TAP controller:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>CPU not in low-power mode.</td> </tr> <tr> <td>1</td> <td>CPU is in low-power mode.</td> </tr> </tbody> </table> <p>Doze includes the Reduced Power (RP) and WAIT power-reduction modes.</p>	Encoding	Meaning	0	CPU not in low-power mode.	1	CPU is in low-power mode.	R	0																											
Encoding	Meaning																																				
0	CPU not in low-power mode.																																				
1	CPU is in low-power mode.																																				
<i>Halt</i>	21	<p>Halt state</p> <p>The Halt bit indicates if the internal system bus clock is running or stopped. The value is sampled in the Capture-DR state of the TAP controller:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Internal system clock is running</td> </tr> <tr> <td>1</td> <td>Internal system clock is stopped</td> </tr> </tbody> </table>	Encoding	Meaning	0	Internal system clock is running	1	Internal system clock is stopped	R	0																											
Encoding	Meaning																																				
0	Internal system clock is running																																				
1	Internal system clock is stopped																																				

Table 11.25 EJTAG Control Register Descriptions (Continued)

Fields		Description	Read / Write	Reset State						
Name	Bit(s)									
<i>PerRst</i>	20	<p>Peripheral Reset</p> <p>When the bit is set to 1, it is only guaranteed that the peripheral reset has occurred in the system when the read value of this bit is also 1. This is to ensure that the setting from the <i>TCK</i> clock domain takes effect in the CPU clock domain and in peripherals.</p> <p>When the bit is written to 0, then the bit must also be read as 0 before it is guaranteed that the indication is also cleared in the CPU clock domain.</p> <p>This bit controls the <i>EJ_PerRst</i> signal on the core.</p>	R/W	0						
<i>PRnW</i>	19	<p>Processor Access Read and Write</p> <p>This bit indicates if the pending processor access is for a read or write transaction, and the bit is only valid while PrAcc is set.</p> <table border="1" data-bbox="522 720 1024 837"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Read transaction</td> </tr> <tr> <td>1</td> <td>Write transaction</td> </tr> </tbody> </table>	Encoding	Meaning	0	Read transaction	1	Write transaction	R	Undefined
Encoding	Meaning									
0	Read transaction									
1	Write transaction									
<i>PrAcc</i>	18	<p>Processor Access (PA)</p> <p>Read value of this bit indicates if a Processor Access (PA) to the EJTAG memory is pending:</p> <table border="1" data-bbox="522 989 1024 1106"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No pending processor access</td> </tr> <tr> <td>1</td> <td>Pending processor access</td> </tr> </tbody> </table> <p>The probe's software must clear this bit to 0 to indicate the end of the PA. Write of 1 is ignored.</p> <p>A pending Processor Access is cleared when <i>Rocc</i> is set, but another PA may occur just after the reset if a debug exception occurs.</p> <p>Finishing a Processor Access is not accepted while the <i>Rocc</i> bit is set. This is to avoid that a Processor Access occurring after the reset is finished due to indication of a Processor Access that occurred before the reset.</p> <p>The FASTDATA access can clear this bit.</p>	Encoding	Meaning	0	No pending processor access	1	Pending processor access	R/W0	0
Encoding	Meaning									
0	No pending processor access									
1	Pending processor access									
Res	17	Reserved	R	0						
<i>PrRst</i>	16	<p>Processor Reset (Implementation dependent behavior)</p> <p>When the bit is set to 1, then it is only guaranteed that this setting has taken effect in the system when the read value of this bit is also 1. This is to ensure that the setting from the <i>TCK</i> clock domain gets effect in the CPU clock domain, and in peripherals.</p> <p>When the bit is written to 0, then the bit must also be read as 0 before it is guaranteed that the indication is cleared in the CPU clock domain also.</p> <p>This bit controls the <i>EJ_PrRst</i> signal. If the signal is used in the system, then it must be ensured that both the processor and all devices required for a reset are properly reset. Otherwise the system may fail or hang. The bit resets itself, since the <i>EJTAG Control</i> register is reset by a reset.</p>	R/W	0						

Table 11.25 EJTAG Control Register Descriptions (Continued)

Fields		Description	Read / Write	Reset State												
Name	Bit(s)															
<i>ProbEn</i>	15	<p>Probe Enable</p> <p>This bit indicates to the CPU if the EJTAG memory is handled by the probe so processor accesses are answered:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>The probe does not handle EJTAG memory transactions</td> </tr> <tr> <td>1</td> <td>The probe does handle EJTAG memory transactions</td> </tr> </tbody> </table> <p>It is an error by the software controlling the probe if it sets the ProbTrap bit to 1, but resets the <i>ProbEn</i> to 0. The operation of the processor is UNDEFINED in this case.</p> <p>The <i>ProbEn</i> bit is reflected as a read-only bit in the <i>ProbEn</i> bit, bit 0, in the <i>Debug Control Register (DCR)</i>.</p> <p>The read value indicates the effective value in the DCR, due to synchronization issues between <i>TCK</i> and CPU clock domains; however, it is ensured that change of the <i>ProbEn</i> prior to setting the <i>EjtagBrk</i> bit will have effect for the debug handler executed due to the debug exception.</p> <p>The reset value of the bit depends on whether the EJTAGBOOT indication is given or not:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Processor is in non-debug mode (No EJTAGBOOT indication given)</td> </tr> <tr> <td>1</td> <td>Processor is in debug mode (EJTAGBOOT indication given)</td> </tr> </tbody> </table>	Encoding	Meaning	0	The probe does not handle EJTAG memory transactions	1	The probe does handle EJTAG memory transactions	Encoding	Meaning	0	Processor is in non-debug mode (No EJTAGBOOT indication given)	1	Processor is in debug mode (EJTAGBOOT indication given)	R/W	0 or 1 from EJTAGBOOT
Encoding	Meaning															
0	The probe does not handle EJTAG memory transactions															
1	The probe does handle EJTAG memory transactions															
Encoding	Meaning															
0	Processor is in non-debug mode (No EJTAGBOOT indication given)															
1	Processor is in debug mode (EJTAGBOOT indication given)															

Table 11.25 EJTAG Control Register Descriptions (Continued)

Fields		Description	Read / Write	Reset State												
Name	Bit(s)															
<i>ProbTrap</i>	14	<p>Probe Trap This bit controls the location of the debug exception vector:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>In normal memory 0xBFC0.0480</td> </tr> <tr> <td>1</td> <td>In EJTAG memory at 0xFF20.0200 in dmseg</td> </tr> </tbody> </table> <p>Valid setting of the <i>ProbTrap</i> bit depends on the setting of the <i>ProbEn</i> bit, as described for the <i>ProbEn</i> bit. The <i>ProbTrap</i> should not be set to 1, for debug exception vector in EJTAG memory, unless the <i>ProbEn</i> bit is also set to 1 to indicate that the EJTAG memory may be accessed. The read value indicates the effective value to the CPU, due to synchronization issues between <i>TCK</i> and CPU clock domains; however, it is ensured that change of the <i>ProbTrap</i> bit prior to setting the <i>EjtagBrk</i> bit will have effect for the <i>EjtagBrk</i>. The reset value of the bit depends on whether the EJTAGBOOT indication is given or not:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Processor is in non-debug mode (No EJTAGBOOT indication given)</td> </tr> <tr> <td>1</td> <td>Processor is in debug mode (EJTAG-BOOT indication given)</td> </tr> </tbody> </table>	Encoding	Meaning	0	In normal memory 0xBFC0.0480	1	In EJTAG memory at 0xFF20.0200 in dmseg	Encoding	Meaning	0	Processor is in non-debug mode (No EJTAGBOOT indication given)	1	Processor is in debug mode (EJTAG-BOOT indication given)	R/W	0 or 1 from EJTAGBOOT
Encoding	Meaning															
0	In normal memory 0xBFC0.0480															
1	In EJTAG memory at 0xFF20.0200 in dmseg															
Encoding	Meaning															
0	Processor is in non-debug mode (No EJTAGBOOT indication given)															
1	Processor is in debug mode (EJTAG-BOOT indication given)															
Res	13	Reserved	R	0												
<i>EjtagBrk</i>	12	<p>EJTAG Break Setting this bit to 1 causes a debug exception to the processor, unless the CPU was in debug mode or another debug exception occurred. When the debug exception occurs, the processor core clock is restarted if the CPU was in low-power mode. This bit is cleared by hardware when the debug exception is taken. The reset value of the bit depends on whether the EJTAGBOOT indication is given or not:</p> <table border="1"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Processor is in non-debug mode (No EJTAGBOOT indication given)</td> </tr> <tr> <td>1</td> <td>Processor is in debug mode (EJTAG-BOOT indication given)</td> </tr> </tbody> </table>	Encoding	Meaning	0	Processor is in non-debug mode (No EJTAGBOOT indication given)	1	Processor is in debug mode (EJTAG-BOOT indication given)	R/W1	0 or 1 from EJTAGBOOT						
Encoding	Meaning															
0	Processor is in non-debug mode (No EJTAGBOOT indication given)															
1	Processor is in debug mode (EJTAG-BOOT indication given)															
Res	11:4	Reserved	R	0												

Table 11.25 EJTAG Control Register Descriptions (Continued)

Fields		Description	Read / Write	Reset State						
Name	Bit(s)									
<i>DM</i>	3	Debug Mode This bit indicates the debug or non-debug mode: <table border="1" data-bbox="522 415 1024 531"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Processor is in non-debug mode</td> </tr> <tr> <td>1</td> <td>Processor is in debug mode</td> </tr> </tbody> </table> The bit is sampled in the <i>Capture-DR</i> state of the TAP controller.	Encoding	Meaning	0	Processor is in non-debug mode	1	Processor is in debug mode	R	0
Encoding	Meaning									
0	Processor is in non-debug mode									
1	Processor is in debug mode									
<i>Res</i>	2:0	Reserved	R	0						

11.4.3 Processor Access Address Register

The *Processor Access Address (PAA)* register is used to provide the address of the processor access in the dmseg, and the register is only valid when a processor access is pending. The length of the Address register is 32 bits, and this register is selected by shifting in the ADDRESS instruction.

11.4.3.1 Processor Access Data Register

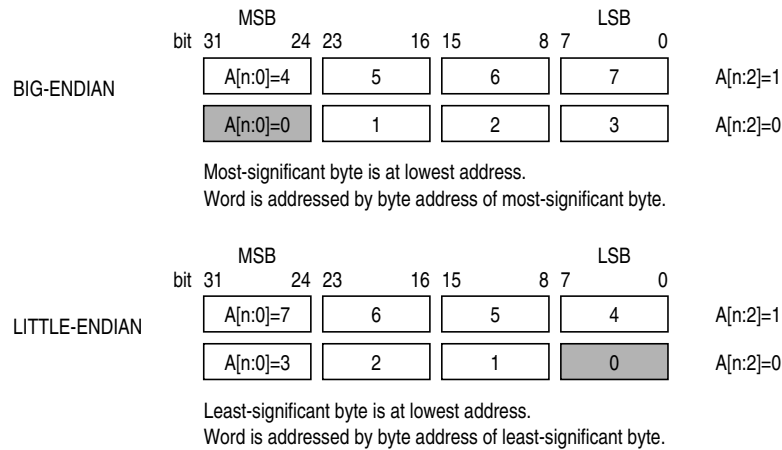
The *Processor Access Data (PAD)* register is used to provide data value to and from a processor access. The length of the Data register is 32 bits, and this register is selected by shifting in the DATA instruction.

The register has the written value for a processor access write due to a CPU store to the dmseg, and the output from this register is only valid when a processor access write is pending. The register is used to provide the data value for a processor access read due to a CPU load or fetch from the dmseg, and the register should only be updated with a new value when a processor access write is pending.

The *PAD* register is 32 bits wide. Data alignment is not used for this register, so the value in the *PAD* register matches data on the internal bus. The undefined bytes for a PA write are undefined, and for a *PAD* read then 0 (zero) must be shifted in for the unused bytes.

The organization of bytes in the *PAD* register depends on the endianness of the core, as shown in [Figure 11.20](#). The endian mode for debug/kernel mode is determined by the state of the *SI_Endian* input at power-up.

Figure 11.20 Endian Formats for PAD Register



The size of the transaction and thus the number of bytes available/required for the *PAD* register is determined by the *Psz* field in the *ECR*.

11.4.4 Fastdata Register (TAP Instruction FASTDATA)

The width of the *Fastdata* register is 1 bit. During a *Fastdata* access, the *Fastdata* register is written and read, i.e., a bit is shifted in and a bit is shifted out. During a *Fastdata* access, the *Fastdata* register value shifted in specifies whether the *Fastdata* access should be completed or not. The value shifted out is a flag that indicates whether the *Fastdata* access was successful or not (if completion was requested).

Figure 11.21 Fastdata Register Format

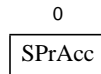


Table 11.26 Fastdata Register Field Description

Fields		Description	Read / Write	Power-up State
Name	Bits			
<i>SPrAcc</i>	0	Shifting in a zero value requests completion of the <i>Fastdata</i> access. The <i>PrAcc</i> bit in the <i>EJTAG Control</i> register is overwritten with zero when the access succeeds. (The access succeeds if <i>PrAcc</i> is one, and the operation address is in the legal <i>dmseg</i> <i>Fastdata</i> area.) When successful, a one is shifted out. Shifting out a zero indicates a <i>Fastdata</i> access failure. Shifting in a one does not complete the <i>Fastdata</i> access, and the <i>PrAcc</i> bit is unchanged. Shifting out a one indicates that the access would have been successful if allowed to complete, and a zero indicates the access would not have successfully completed.	R/W	Undefined

The *FASTDATA* access is used for efficient block transfers between *dmseg* (on the probe) and target memory (on the processor). An “upload” is defined as a sequence of processor loads from target memory and stores to *dmseg*. A “download” is a sequence of processor loads from *dmseg* and stores to target memory. The “*Fastdata* area” specifies the legal range of *dmseg* addresses (0xFF20.0000 - 0xFF20.000F) that can be used for uploads and downloads. The

Data plus *Fastdata* registers (selected with the FASTDATA instruction) allow efficient completion of pending Fastdata area accesses.

During Fastdata uploads and downloads, the processor will stall on accesses to the Fastdata area. The *PrAcc* (processor access pending bit) will be 1, indicating the probe is required to complete the access. Both upload and download accesses are attempted by shifting in a zero *SPrAcc* value (to request access completion) and shifting out *SPrAcc* to see if the attempt will be successful (i.e., there was an access pending and a legal Fastdata area address was used). Downloads will also shift in the data to be used to satisfy the load from dmseg's Fastdata area, while uploads will shift out the data being stored to dmseg's Fastdata area.

As noted above, two conditions must be true for the Fastdata access to succeed:

- *PrAcc* must be 1, i.e., there must be a pending processor access.
- The Fastdata operation must use a valid Fastdata area address in dmseg (0xFF20.0000 to 0xFF20.000F).

Table 11.27 shows the values of the *PrAcc* and *SPrAcc* bits and the results of a Fastdata access.

Table 11.27 Operation of the FASTDATA Access

Probe Operation	Address Match Check	PrAcc in the Control Register	LSB (SPrAcc) Shifted In	Action in the Data Register	PrAcc Changes to	LSB Shifted Out	Data Shifted Out
Download using FAST-DATA	Fails	x	x	none	unchanged	0	invalid
	Passes	1	1	none	unchanged	1	invalid
		1	0	write data	0 (SPrAcc)	1	valid (previous) data
		0	x	none	unchanged	0	invalid
Upload using FASTDATA	Fails	x	x	none	unchanged	0	invalid
	Passes	1	1	none	unchanged	1	invalid
		1	0	read data	0 (SPrAcc)	1	valid data
		0	x	none	unchanged	0	invalid

There are no restrictions on the contents of the Data register. It is expected that the transfer size is negotiated between the download/upload transfer code and the probe software. Note that the most efficient transfer size is a 32-bit word.

The *Rocc* bit of the *ECR* register is not used for the FASTDATA operation.

11.5 TAP Processor Accesses

The TAP modules support handling of fetches, loads and stores from the CPU through the dmseg segment, whereby the TAP module can operate like a *slave unit* connected to the on-chip bus. The core can then execute code taken from the EJTAG Probe and it can access data (via a load or store) which is located on the EJTAG Probe. This occurs in a serial way through the EJTAG interface: the core can thus execute instructions e.g. debug monitor code, without occupying the memory.

Accessing the dmseg segment (EJTAG memory) can only occur when the processor accesses an address in the range from 0xFF20.0000 to 0xFF2F.FFFF, the *ProbEn* bit is set, and the processor is in debug mode ($DM=1$). In addition the LSNM bit in the CP0 Debug register controls transactions to/from the dmseg.

When a debug exception is taken, while the *ProbTrap* bit is set, the processor will start fetching instructions from address 0xFF20.0200.

A pending processor access can only finish if the probe writes 0 to *PrAcc* or by a reset.

11.5.1 Fetch/Load and Store From/To the EJTAG Probe Through dmseg

1. The internal hardware latches the requested address into the *PA Address* register (in case of the Debug exception: 0xFF20.0200).
2. The internal hardware sets the following bits in the *EJTAG Control* register:
PrAcc = 1 (selects Processor Access operation)
PRnW = 0 (selects processor read operation)
Psz[1:0] = value depending on the transfer size
3. The EJTAG Probe selects the *EJTAG Control* register, shifts out this control register's data and tests the *PrAcc* status bit (Processor Access): when the *PrAcc* bit is 1, the requested address is available and can be shifted out.
4. The EJTAG Probe checks the *PRnW* bit to determine the required access.
5. The EJTAG Probe selects the *PA Address* register and shifts out the requested address.
6. The EJTAG Probe selects the *PA Data* register and shifts in the instruction corresponding to this address.
7. The EJTAG Probe selects the *EJTAG Control* register and shifts a *PrAcc* = 0 bit into this register to indicate to the processor that the instruction is available.
8. The instruction becomes available in the instruction register and the processor starts executing.
9. The processor increments the program counter and outputs an instruction read request for the next instruction. This starts the whole sequence again.

Using the same protocol, the processor can also execute a load instruction to access the EJTAG Probe's memory. For this to happen, the processor must execute a load instruction (e.g. a LW, LH, LB) with the target address in the appropriate range.

Almost the same protocol is used to execute a store instruction to the EJTAG Probe's memory through dmseg. The store address must be in the range: 0xFF20.0000 to 0xFF2F.FFFF, the *ProbEn* bit must be set and the processor has to be in debug mode ($DM=1$). The sequence of actions is found below:

1. The internal hardware latches the requested address into the *PA Address* register.
2. The internal hardware latches the data to be written into the *PA Data* register.
3. The internal hardware sets the following bits in the *EJTAG Control* register:
PrAcc = 1 (selects Processor Access operation)
PRnW = 1 (selects processor write operation)
Psz[1:0] = value depending on the transfer size

4. The EJTAG Probe selects the *EJTAG Control* register, shifts out this control register's data and tests the *PrAcc* status bit (Processor Access): when the *PrAcc* bit is found 1, it means that the requested address is available and can be shifted out.
5. The EJTAG Probe checks the *PRnW* bit to determine the required access.
6. The EJTAG Probe selects the *PA Address* register and shifts out the requested address.
7. The EJTAG Probe selects the *PA Data* register and shifts out the data to be written.
8. The EJTAG Probe selects the *EJTAG Control* register and shifts a *PrAcc* = 0 bit into this register to indicate to the processor that the write access is finished.
9. The EJTAG Probe writes the data to the requested address in its memory.
10. The processor detects that *PrAcc* bit = 0, which means that it is ready to handle a new access.

The above examples imply that no reset occurs during the operations, and that *Rocc* is cleared.

11.6 PC Sampling

The PC sampling feature enables periodic sampling of the PC value. This information can be used for statistical profiling akin to `gprof` and is also very useful for detecting hot-spots in the code. PC sampling cannot be turned on or off, that is, the PC value is continually sampled.

The presence or absence of the PC Sampling feature is specified in *Debug Control*_{PCS} (bit 9). The sampled PC values are written into a TAP register. The old value in the TAP register is overwritten by a new value, even if this register has not been read out by the debug probe. The sample rate is specified in a manner similar to the PDtrace synchronization period, with three bits. These bits in the *Debug Control* register are 8:6 and called *PCSR* (PC SampleRate). These three bits take the value 2^5 to 2^{12} similar to *SyncPeriod*. Note that the processor samples PC even when it is asleep (in a WAIT state). This permits an analysis of the amount of time spent by a processor in WAIT state which may be used for example to revert to a low-power mode during the non-execution phase of a real-time application.

The sampled values includes a New data bit, the PC, the ASID of the sampled PC, and the Thread Context ID if the processor implements the MIPS MT ASE. Figure 11.22 shows the format of the sampled values in the TAP register PCsample. The new data bit is used by the probe to determine if the PCsample register data just read out is new or already been read and must be discarded.

Figure 11.22 TAP Register PCsample Format

48	41	40	33	32	1	0
TC (for MIPS MT processors only)	ASID		PC			New

The sampled PC value is the PC of the graduating instruction in the current cycle. If the processor is stalled when the PC sample counter overflows, then the sampled PC is the PC of the next graduating instruction. The processor continues to sample the PC value even when it is in Debug mode.

11.6.1 PC Sampling in Wait State

When the processor is in a WAIT state (to save power, for example), an external agent might want to know how long it stays in the WAIT state. But counting cycles to update the PC sample value is a waste of power. Hence, when in a WAIT state, the processor must simply switch the New bit to 1 every time it is set to 0 by the probe hardware. Hence, the external agent or probe reading the PC value will detect a WAIT instruction for as long as the processor remains in the WAIT state. When the processor leaves the WAIT state, counting is resumed as before.

11.7 Fast Debug Channel

The Fast Debug Channel (FDC) mechanism provides an efficient means to transfer data between the Core and an external device using the EJTAG TAP pins. The external device would typically be an EJTAG probe and that is the term used here, but it could be something else. FDC utilizes two First In First Out (FIFO) structures to buffer data between the Core and probe. The probe uses the FDC TAP instruction to access these FIFOs, while the Core itself accesses them using memory accesses. To transfer data out of the Core, the Core writes one or more pieces of data to the transmit FIFO. At this time, the Core can resume doing other work. An external probe would examine the status of the transmit FIFO periodically. If there is data to be read, the probe starts to receive data from the FIFO, one entry at a time. When all data from the FIFO has been drained, the probe goes back to waiting for more data. The Core can either choose to be informed of the empty transmit FIFO via an interrupt, or it can choose to periodically check the status. Receiving data works in a similar manner - the probe writes to the receive FIFO. At that time, the Core is either interrupted, or finds out via polling a status bit. The Core can then do load accesses to the receive FIFO and receive data being sent to it by the probe. The TAP transfer is bidirectional - a single shift can be pulling transmit data and putting receive data at the same time.

The primary advantage of FDC over normal processor accesses or fastdata accesses is that it does not require the Core to be blocked when the probe is reading or writing to the data transfer FIFOs. This significantly reduces the Core overhead and makes the data transfer far less intrusive to the code executing on the Core.

Refer to the EJTAG Specification [11] for the general details on FDC. The remainder of this section describes implementation specific behavior and register values.

The FDC memory mapped registers are located in the common device memory map (CDMM) region. FDC has a device ID of 0xFD.

11.7.1 Common Device Memory Map

Software on the Core accesses FDC through memory mapped registers. These memory mapped registers are located within the Common Device Memory Map (CDMM). The CDMM is a region of physical address space that is reserved for mapping IO device configuration registers within a MIPS processor. The base address and enabling of this region is controlled by the CDMMBase CP0 register, see [Section 7.2.22 “CDMMBase Register \(CP0 Register 15, Select 2\)”](#).

Refer to Volume III of the Architecture Reference Manuals [14] for full details on CDMM.

11.7.2 Fast Debug Channel Interrupt

The FDC block can generate an interrupt to inform software of incoming data being available or space being available in the outgoing FIFO. This interrupt is handled similarly to the timer or performance counter interrupts. The *Cause_{FDCI}* bit indicates that the interrupt is pending. The interrupt is also sent to the core output *SI_{FDCI}* where it is combined with one of the *SI_{Int}* pins. For non-EIC mode, the *SI_{IPFDCI}* input indicates which interrupt pin is has

been combined with and this information is reflected in the *IntCtlIPFDCI* field. Note that this interrupt is a regular interrupt and not a debug interrupt.

The FDC Configuration Register (see [Section 11.7.6.2 “FDC Configuration \(FDCFG\) Register \(Offset 0x8\)”](#)) includes fields for enabling and setting the threshold for generating each interrupt. Receive and transmit interrupt thresholds are specified independently, but they are ORed together to form a single interrupt .

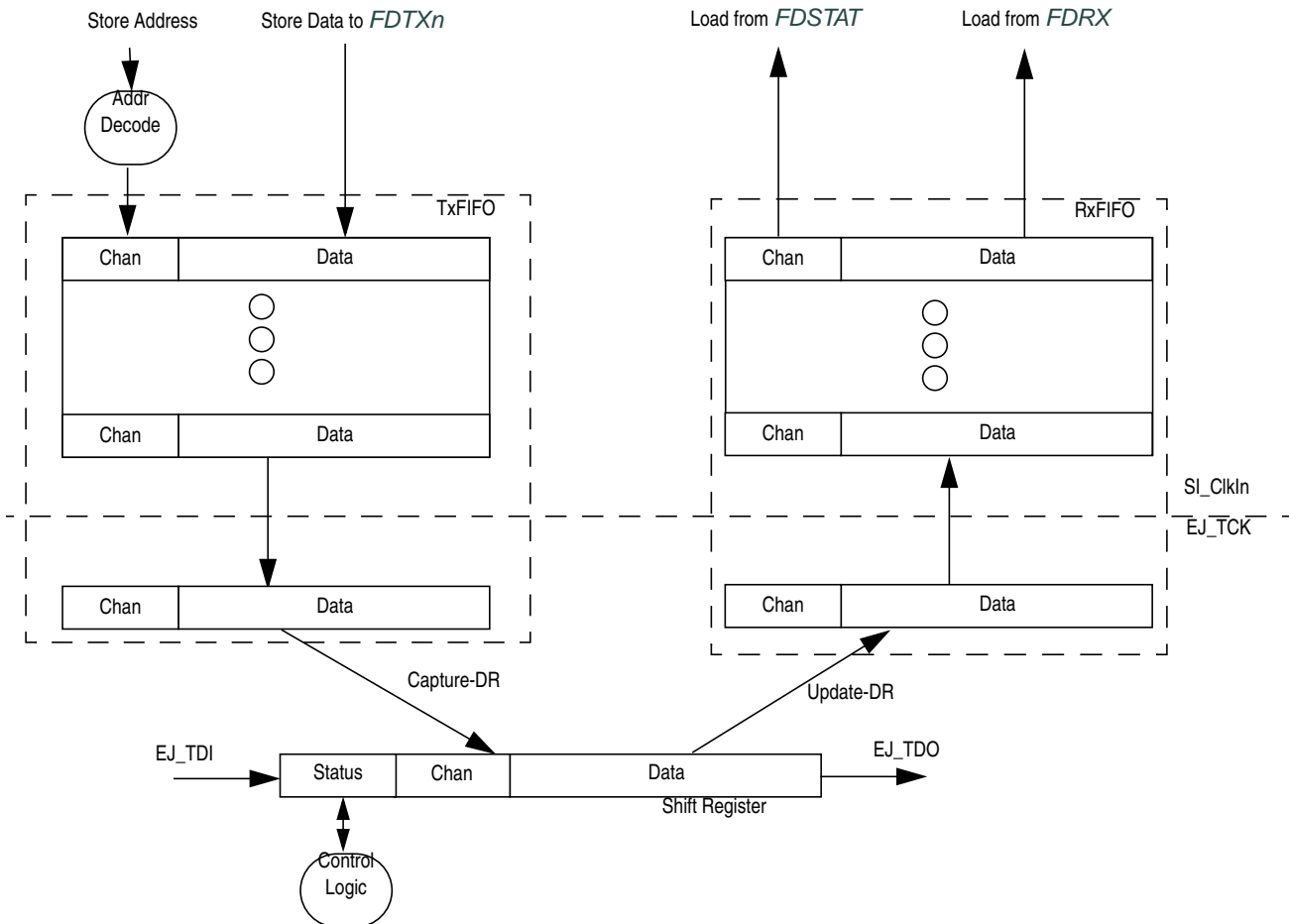
The following interrupt thresholds are supported:

- **Interrupts Disabled:** No interrupt will be generated and software must poll the status registers to determine if incoming data is available or if there is space for outgoing data.
- **Minimum Core Overhead:** This setting minimizes the Core overhead by not generating an interrupt until the receive FIFO (RxFIFO) is completely full or the transmit FIFO (TxFIFO) is completely empty.
- **Minimum latency:** To have the Core take data as soon as it is available, the receive interrupt can be fired whenever the RxFIFO is not empty. There is a complimentary TxFIFO not full setting although that may not be quite as useful.
- **Maximum bandwidth:** When configured for minimum Core overhead, bandwidth between the probe and Core can be wasted if the Core does not service the interrupt before the next transfer occurs. To reduce the chances of this happening, the interrupt threshold can be set to almost full or almost empty to generate an interrupt earlier. This setting causes receive interrupts to be generated when there are 0 or 1 unused RxFIFO entries. Transmit interrupts are generated when there are 0 or 1 used TxFIFO entries (see note in following section about this condition)

11.7.3 74K™Core FDC Buffers

[Figure 11.23](#) shows the general organization of the transmit and receive buffers on the 74K Core.

Figure 11.23 Fast Debug Channel Buffer Organization



One particular thing to note is the asynchronous crossings between the EJ_TCK and SI_ClkIn clock domains. This crossing is handled with a handshake interface that safely transfers data between the domains. Two data registers are included in this interface, one in the source domain and one in the destination domain. The control logic actively manages these registers so that they can be used as FIFO entries. The fact that one FIFO entry is in the EJ_TCK clock domain is normally transparent, but it can create some unexpected behavior:

- **TxFIFO availability:** Data is first written into the SI_ClkIn FIFO entries, then it will move into the EJ_TCK FIFO entry. But, it takes several EJ_TCK cycles to complete the handshake and move the data. EJ_TCK is generally much slower than SI_ClkIn and may even be stopped (although that would be uncommon when this feature is in use). This can result in there not being space for new data, even though there are only $N-1$ data values queued up. To prevent the loss of data, the $FDSTAT_{TxF}$ bit is set when all of the SI_ClkIn FIFO entries are full. Software writing to the FIFO should always check the $FDSTAT_{TxF}$ bit prior to attempting a write and should not make any assumptions about being able to arbitrarily use all entries. i.e., software seeing the $FDSTAT_{FxE}$ bit set should not assume that it can write $FDCFG_{TxCnt}$ data words without checking for full.
- **TxFIFO Almost Empty Interrupt:** As transmit data moves from SI_ClkIn to EJ_TCK , both of the flops will temporarily look full. This makes it difficult to determine when just 1 FIFO entry is in use. To enable a simpler condition, the almost empty TxInterrupt condition is set when all of the SI_ClkIn FIFO entries are empty. When this

condition is met, there will be 0 or 1 valid entries. However, the interrupt will not be asserted when there is only one valid entry if it is an *Sl_ClkIn* entry

- The RxFIFO has similar characteristics but these are even less visible to software since *Sl_ClkIn* must be running to access the FDC registers.

11.7.4 Sleep mode

FDC data transfers do not prevent the core from entering sleep mode and will proceed normally in sleep mode. The FDC block monitors the TAP interface signals with a free-running clock. When new receive data is available or transmit data can be sent, the gated clock will be enabled for a few cycles to transfer the data and then allowed to stop again. If FDC interrupts are enabled, transferring data may cause an interrupt to be generated which can wake the core up.

11.7.5 FDC TAP Register

The FDC TAP instruction performs a 38 bit bidirectional transfer of the FDC TAP register. The register format is shown in [Figure 11.24](#) and the fields are described in [Figure 11.28](#)

Figure 11.24 FDC TAP Register Format

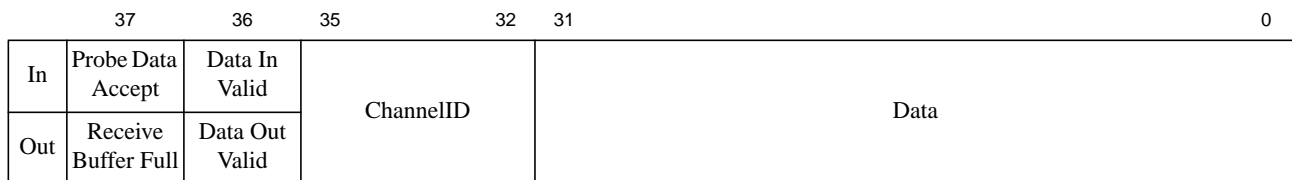


Table 11.28 FDC TAP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Probe Data Accept	37	Indicates to core that the probe is accepting the data that was scanned out.	W	Undefined
Data In Valid	36	Indicates to core that the probe is sending new data to the receive FIFO.	W	Undefined
Receive Buffer Full	37	Indicates to probe that the receive buffer is full and the core will not accept the data being scanned in. Analogous to ProbeDataAccept, but opposite polarity	R	0x0
Data Out Valid	36	Indicates to probe that the core is sending new data from the transmit FIFO	R	0
ChannelID	35:32	Channel number associated with the data being scanned in or out. This field can be used to indicate the type of data that is being sent and allow independent communication channels Scanning in a value with ChannelID=0xd and Data In Valid = 0 will generate a receive interrupt. This can be used when the probe has completed sending data to the core.	R/W	Undefined

Table 11.28 FDC TAP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Data	31:0	Data value being scanned in or out	R/W	Undefined

11.7.6 Fast Debug Channel Registers

This section describes the Fast Debug Channel registers. CPU access to FDC is via loads and stores to the FDC device in the Common Device Memory Map (CDMM) region. These registers provide access control, configuration and status information, as well as access to the transmit and receive FIFOs. The registers and their respective offsets are shown in [Table 11.29](#)

Table 11.29 FDC Register Mapping

Offset in CDMM Device Block	Register Mnemonic	Register Name and Description
0x0	FDACSR	FDC Access Control and Status Register
0x8	FDCFG	FDC Configuration Register
0x10	FDSTAT	FDC Status Register
0x18	FDRX	FDC Receive Register
0x20 + 0x8* n	FDTXn	FDC Transmit Register n (0 ≤ n ≤ 15)

11.7.6.1 FDC Access Control and Status (FDACSR) Register (Offset 0x0)

This is the general CDMM Access Control and Status register which defines the device type and size and controls user and supervisor access to the remaining FDC registers. The Access Control and Status register itself is only accessible in kernel mode. [Figure 11.25](#) has the format of an Access Control and Status register (shown as a 64-bit register), and [Table 11.30](#) describes the register fields.

Figure 11.25 FDC Access Control and Status Register Format



Table 11.30 FDC Access Control and Status Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
DevID	31:24	This field specifies the type of device.	R	0xfd
DevSize	21:16	This field specifies the number of extra 64-byte blocks allocated to this device. The value 0x2 indicates that this device uses 2 extra, or 3 total blocks.	R	0x2
DevRev	15:12	This field specifies the revision number of the device. The value 0x0 indicates that this is the initial version of FDC.	R	0x0

Table 11.30 FDC Access Control and Status Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
Uw	3	This bit indicates if user-mode write access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to write to the device while in user mode with access disabled is ignored.	R/W	0
Ur	2	This bit indicates if user-mode read access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to read from the device while in user mode with access disabled will return 0 and not change any state.	R/W	0
Sw	1	This bit indicates if supervisor-mode write access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to write to the device while in supervisor mode with access disabled is ignored.	R/W	0
Sr	0	This bit indicates if supervisor-mode read access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to read from the device while in supervisor mode with access disabled will return 0 and not change any state..	R/W	0
0	11:4	Reserved for future use. Ignored on write; returns zero on read.	R	0

11.7.6.2 FDC Configuration (FDCFG) Register (Offset 0x8)

The FDC configuration register holds information about the current configuration of the Fast Debug Channel mechanism. Figure 11.26 has the format of the FDC Configuration register, and Table 11.31 describes the register fields.

Figure 11.26 FDC Configuration Register Format

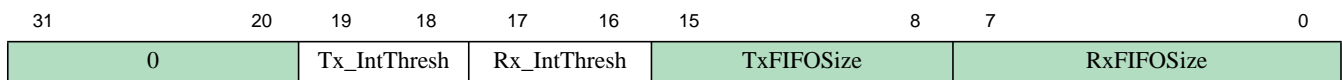


Table 11.31 FDC Configuration Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:20	Reserved for future use. Read as zeros, must be written as zeros.	R	0

Table 11.31 FDC Configuration Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State										
Name	Bits													
TxIntThresh	19:18	Controls whether transmit interrupts are enabled and the state of the TxFIFO needed to generate an interrupt. <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Transmit Interrupt Disabled</td> </tr> <tr> <td>1</td> <td>Empty</td> </tr> <tr> <td>2</td> <td>Not Full</td> </tr> <tr> <td>3</td> <td>Almost Empty - zero or one entry in use*(see 11.7.2 for specifics)</td> </tr> </tbody> </table>	Encoding	Meaning	0	Transmit Interrupt Disabled	1	Empty	2	Not Full	3	Almost Empty - zero or one entry in use*(see 11.7.2 for specifics)	R/W	0
Encoding	Meaning													
0	Transmit Interrupt Disabled													
1	Empty													
2	Not Full													
3	Almost Empty - zero or one entry in use*(see 11.7.2 for specifics)													
RxIntThresh	17:16	Controls whether receive interrupts are enabled and the state of the RxFIFO needed to generate an interrupt. <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Receive Interrupt Disabled</td> </tr> <tr> <td>1</td> <td>Full</td> </tr> <tr> <td>2</td> <td>Not empty</td> </tr> <tr> <td>3</td> <td>Almost Full - zero or one entry free</td> </tr> </tbody> </table>	Encoding	Meaning	0	Receive Interrupt Disabled	1	Full	2	Not empty	3	Almost Full - zero or one entry free	R/W	0
Encoding	Meaning													
0	Receive Interrupt Disabled													
1	Full													
2	Not empty													
3	Almost Full - zero or one entry free													
TxFIFOSize	15:8	This field holds the total number of entries in the transmit FIFO.	R	Preset										
RxFIFOSize	7:0	This field holds the total number of entries in the receive FIFO.	R	Preset										

11.7.6.3 FDC Status (FDSTAT) Register (Offset 0x10)

The FDC Status register holds up to date state information for the FDC mechanism. [Figure 11.27](#) has the format of the FDC Status register, and [Table 11.32](#) describes the register fields.

Figure 11.27 FDC Status Register Format

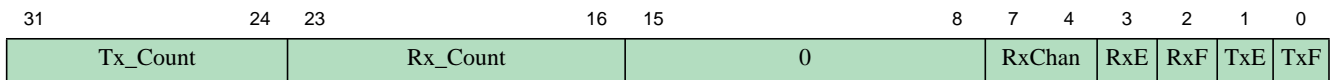


Table 11.32 FDC Status Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Tx_Count	31:24	This optional field is not implemented and will read as 0	R	0
Rx_Count	23:16	This optional field is not implemented and will read as 0	R	0

Table 11.32 FDC Status Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
0	15:8	Reserved for future use. Must be written as zeros and read as zeros.	R	0
RxChan	7:4	This field indicates the channel number used by the top item in the receive FIFO. This field is only valid if RxE=0.	R	Undefined
RxE	3	If RxE is set, the receive FIFO is empty. If RxE is not set, the FIFO is not empty.	R	1
RxF	2	If RxF is set, the receive FIFO is full. If RxF is not set, the FIFO is not full.	R	0
TxE	1	If TxE is set, the transmit FIFO is empty. If TxE is not set, the FIFO is not empty.	R	1
TxF	0	If TxF is set, the transmit FIFO is full. If TxF is not set, the FIFO is not full.	R	0

11.7.6.4 FDC Receive (FDRX) Register (Offset 0x18)

This register exposes the top entry in the receive FIFO. A read from this register returns the top item in the FIFO and removes it from the FIFO itself. The result of a write to this register is **UNDEFINED**. The result of a read when the FIFO is empty is also **UNDEFINED** so software must check the $FDSTAT_{RxE}$ flag prior to reading. Figure 11.28 has the format of the FDC Receive register, and Table 11.33 describes the register fields.

Figure 11.28 FDC Receive Register Format

31

0

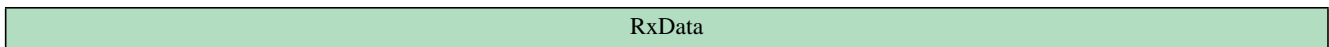


Table 11.33 FDC Receive Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
RxData	31:0	This register holds the top entry in the receive FIFO	R	Undefined

11.7.6.5 FDC Transmit n (FDTXn) Registers (Offset 0x20 + 0x8*n)

These sixteen registers all access the bottom entry in the transmit FIFO. The different addresses are used to generate a 4b channel identifier that is attached to the data value. This allows software to track different event types without needing to reserve a portion of the 32b data as a tag. A write to one of these registers results in a write to the transmit FIFO of the data value and channel ID corresponding to the register being written. Reads from these registers are **UNDEFINED**. Attempting to write to the transmit FIFO if it is full has **UNDEFINED** results. Hence, the software running on the core must check the $FDSTAT_{TxF}$ flag to ensure that there is space for the write. Figure 11.29 has the format of the FDC Transmit register, and Table 11.34 describes the register fields.

Figure 11.29 FDC Transmit Register Format

31

0

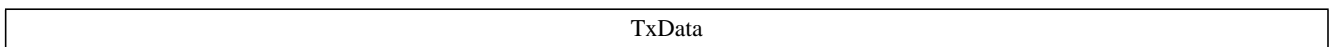


Table 11.34 FDC Transmit Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
TxData	31:0	This register holds the bottom entry in the transmit FIFO	W, Undefined value on read	Undefined

Table 11.35 FDTXn Address Decode

Addr	Chan		Addr	Chan		Addr	Chan		Addr	Chan
0x20	0x0		0x40	0x4		0x60	0x8		0x80	0xc
0x28	0x1		0x48	0x5		0x68	0x9		0x88	0xd
0x30	0x2		0x50	0x6		0x70	0xa		0x90	0xe
0x38	0x3		0x58	0x7		0x78	0xb		0x98	0xf

11.8 MIPS® Trace

The optional MIPS Trace block provides the user with an informative method of tracing program flow, load/store addresses and data, performance counters, and core-specific inefficiencies. The level of information which is traced, for example, tracing only when in specific processor modes (e.g., User Mode or Kernel Mode) is controlled by run-time options.

When MIPS Trace is implemented, the *CP0 Config3_{TL}* bit is set by hardware when the core is configured.

The pipeline-specific architecture of MIPS Trace is specified in the *PDtrace™ Interface and Trace Control Block Specification* [12].

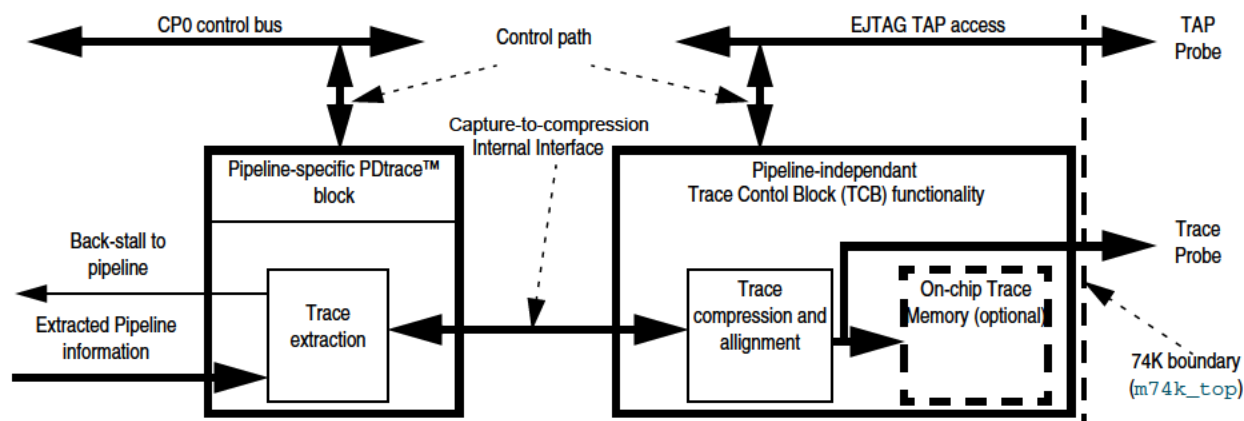
There are two primary functional blocks: the PDtrace capture block and the blocks that implement the functionality of the Trace Control Block (TCB). The PDtrace capture block extracts the trace information from the end of the processor pipeline from the in-order graduation stage and stores the information in an internal FIFO called the Unified FIFO. The capture block then presents the data from the Unified FIFO to the PDtrace compression block.

The functionality of the TCB is specified in

The compression block and the TCB Registers implement the Trace Control Block (TCB). Though there is not an explicit module called TCB, the functionality of the TCB, as specified in [12], has been completely implemented and integrated into the PDtrace unit. Thus, it is no longer a customer option to implement a custom TCB.

Note that the generic pin interface, as defined in the retired document *PDtrace™ Interface Specification*, that was used to “communicate” between the capture and the TCB functionality is deprecated. The interface is replaced by an internal interface that is called the capture-to-compression interface. This interface is embedded inside the 74K core, and will not be discussed in detail here. Suffice it to say that the internal interface embodies all the functionality described in [12]. While working closely together, the two parts of MIPS Trace are controlled separately by software. Figure 11.30 shows an overview of the MIPS Trace modules within the core.

Figure 11.30 MIPS® Trace Functional Blocks in the 74K™ Core



To some extent, the two modules provide similar trace control features, but the access to these features is quite different. The PDtrace software controls can only be reached through access to CP0 registers. The PDtrace hardware controls can only be reached through EJTAG TAP access. The selection of one of these controls determines what is traced from the core pipeline and the information presented in the internal capture-to-compression interface.

Before describing the MIPS Trace implemented in the 74K core, some common terminology and basic features are explained. The remaining sections of this chapter will then provide a more thorough explanation.

11.8.1 Processor Modes

Tracing can be enabled or disabled based on various processor modes. This section precisely describes these modes. The terminology is then used elsewhere in the document.

```

DebugMode ← (DebugDM = 1)
ExceptionMode ← (not DebugMode) and ((StatusEXL = 1) or (StatusERL = 1))
KernelMode ← (not (DebugMode or ExceptionMode)) and (StatusKSU = 2#00)
SupervisorMode ← (not (DebugMode or ExceptionMode)) and (StatusKSU = 2#01)
UserMode ← (not (DebugMode or ExceptionMode)) and (StatusKSU = 2#10)

```

11.8.2 Software Versus Hardware Control

In some of the specifications and in this text, the terms “software control” and “hardware control” are used to refer to the method used to control for the trace. Software control is when the CP0 register *TraceControl* is used to select the modes to trace, etc. Hardware control is when the EJTAG register *TCBCONTROLA* in the TCBRegs is used to select the trace modes. The *TraceControl_{TS}* bit determines whether software or hardware control is active. Even in Software control mode, trace logic will need to toggle *TCK* at least once before it is turned on. It is assumed that the EJTAG probe will be connected while using trace, and the probe’s reset sequence would toggle *TCK*. Note that to extract trace data from the trace compression block, *TCBCONTROLB_{EN}* should be set to 1, even in “software control” mode.

11.8.3 Trace Information

The main object of trace is to show the exact program flow from a specific program execution or just a small window of the execution. In MIPS Trace this is done by providing the minimal cycle-by-cycle information necessary for trace regeneration software to reproduce the trace. The following is a summary of the type of information traced:

- Only instructions which complete at the end of the pipeline are traced, and indicated with a completion-flag. The PC is implicitly pointing to the next instruction.
- Load instructions are indicated with a load-flag.
- Store instructions are indicated with a store-flag¹.
- Taken branches are indicated with a branch-taken-flag on the target instruction.
- New PC information for a branch is only traced if the branch target is unpredictable from the static program image.
- When branch targets are unpredictable, only the delta value from the current PC is traced, if it is dynamically determined to reduce the number of bits necessary to indicate the new PC. Otherwise the full PC value is traced.
- When a completing instruction is executed in a different processor mode from the previous one, the new processor mode is traced.
- The first instruction is always traced as a branch target, with processor mode and full PC.
- Periodic synchronization instructions are identified with a sync-flag, and traced with the processor mode and full PC. The sync instruction is not a load and not a store.

All the instruction flags above are combined into one, 3-bit value, called the “instruction completion” to minimize the bit information to trace. The possible processor modes are explained in [Section 11.8.1 “Processor Modes”](#).

The target address is statically predictable for all branch and all jump-immediate instructions. If the branch is taken, then the branch-taken-flag will indicate this. All jump-register instructions and ERET/DERET are instructions which have an unpredictable target address. These will have full/delta PC values included in the trace information. Also treated as unpredictable are PC changes which occur due to exceptions, such as an interrupt, reset, etc.

Trace regeneration software must know the static program image in memory, in order to reproduce the dynamic flow with the above information. But this is usually not a problem. Only the virtual value of the PC is used. Physical memory location will typically differ.

It is possible to turn on PC delta/full information for all branches, but this should not normally be necessary. As a safety check for trace regeneration software, a periodic synchronization with a full PC is sent. The period of this synchronization is cycle-based and programmable.

11.8.4 Load/Store Address and Data Trace Information

In addition to PC flow, it is possible to get information on the load/store addresses, as well as the data read/written. When enabled, the following information is optionally added to the trace:

- When load-address tracing is on, the full load address of the first load instruction is traced (indicated by the load-flag). For subsequent loads, a dynamically-determined delta to the previous load address is traced to compress the information which must be sent.

1. An SC (Store Conditional) instruction is flagged as a store instruction, even if the load-locked bit prevented the actual store. Thus the SC does not have special handling and is treated as any other store; it is up to the reconstruction software to determine if the SC succeeded or failed.

- When store-address tracing is on, the full store address of the first store instruction is traced (indicated by the store-flag). For subsequent stores, a dynamically-determined delta to the previous store address is traced.
- When load-data tracing is on, the full load data read by each load instruction is traced (indicated by the load-flag). Only actual read bytes are traced.
- When store-data tracing is on, the full store data written by each store instruction is traced (indicated by the store-flag). Only written bytes are traced.

After each synchronization instruction, the first load address and the first store address following this are both traced with the full address if load/store address tracing is enabled.

11.8.5 Programmable Processor Trace Mode Options

To enable tracing, a global Trace On signal must be set. When trace is on, it is possible to enable tracing in any combination of the processor modes described in [Section 11.8.1 “Processor Modes”](#). In addition to this, trace can be turned on globally for all processes, or only for specific processes, by tracing only specific masked values of the ASID found in *EntryHiASID*.

Additionally, an EJTAG Simple Break trigger point can override the processor mode and ASID selection and turn them all on. Another trigger point can disable this override.

11.8.6 Programmable Trace Information Options

The processor mode changes are always traced:

- On the first instruction
- On any synchronization instruction
- When the mode changes, and either the previous or the current processor mode is selected for trace

The amount of extra information traced is programmable to include:

- PC information only
- PC and cross product of load/store address/data
- If the optional performance counter trace is enabled, when the specific events defined in [Section 11.8.11 “Performance Counter Tracing”](#) occur, up to four performance counter registers are traced.

If the full internal state of the processor is known prior to trace start, PC and load data are the only information needed to recreate all register values on an instruction-by-instruction basis.

11.8.6.1 User Data Trace

Two special CP0 registers, *UserTraceData1* and *UserTraceData2*, can generate a data trace. When either of these registers is written, and the global Trace On is set, the 32-bit data written is put in the trace as special User Data information. Since writing these registers is performed via an MTC0 operation, only one register is updated in any given cycle. Thus in the same cycle, only one of the UserTraceData registers is traced. However, in back to back cycles, the tracing of the two registers can alternate and is handled correctly.

Note: The User Data is sent even if the processor is operating in an un-traced processor mode.

11.8.7 Enable Trace to Probe On-chip Memory

When trace is On, based on the options listed in [Section 11.8.5 “Programmable Processor Trace Mode Options”](#), the trace information is continuously sent to the Trace Compression and TCB Control Block. However, the TCB must be enabled to transmit the trace information to the Trace probe or to on-chip trace memory by setting the *TCBCONTROLB_{EN}* bit. This bit can be set in three ways:

- Set/clear the *TCBCONTROLB_{EN}* bit via an EJTAG TAP operation.
- Initialize a TCB trigger to set/clear the *TCBCONTROLB_{EN}* bit.
- Use the drseg mapping of *TCBCONTROLB* to clear *TCBCONTROLB_{EN}* via a load to drseg space. See [Section 11.8.15 “Memory-mapped Access to On-Chip Trace RAM”](#) for special access rules.

11.8.8 TCB Trigger

The TCB can optionally include 0 to 8 triggers. A TCB trigger can be programmed to fire from any combination of:

- Probe Trigger Input to the TCB
- Chip-level Trigger Input to the TCB
- Processor entry into DebugMode

When a trigger fires, it can be programmed to have any combination of actions:

- Create Probe Trigger Output from TCB
- Create Chip-level Trigger Output from TCB
- Set, clear, or start countdown to clear the *TCBCONTROLB_{EN}* bit (start/end/about trigger)
- Put an information byte into the trace stream, that is, a TF6 is inserted into the trace stream

11.8.9 Cycle-by-Cycle Information

The PDtrace capture block collects all of the trace information listed in [Section 11.8.3 “Trace Information”](#) and [Section 11.8.4 “Load/Store Address and Data Trace Information”](#). The trace is then compressed and aligned to fit in 64-bit trace words, with no loss of information. It is possible to exclude/include the exact cycle-by-cycle relationship between each instruction. If excluded, the number of bits required in the trace information from the TCB is reduced, and each trace word will only contain information from completing instructions.

11.8.10 Instruction and Data Cache Miss Tracing

It is possible to embed information about Instruction and/or Data cache misses into the trace information, with some limitations, as described below.

For the instruction cache miss indicator:

- The instruction cache miss indicator is based on whether the instruction is pulled from the cache or the fill buffer. On a cache miss, the fetch is restarted when the data comes back from the BIU and the instructions will come from the Fill Buffer (FB). The miss flag is only set for the first fetch that hits out of the FB to avoid marking every fetch from the line a miss. However, two instructions can be fetched per cycle and both will be marked as a miss, though if branching to the middle of a dword, only 1 miss will be seen.
- The IFU can prefetch a speculative path which might not be immediately executed. These speculative fetches are filled into the cache; subsequently, when the code accesses the same address, it is possible that the instruction will hit in the cache even if that instruction was being executed for the first time.

For the data cache miss indicator:

- PDtrace instruction capture is done at the end of the GRU (graduation) pipe. However, at this point the cache miss info is not accurate. Hit indication is accurate, but the miss indication is not. The miss could change to a hit after it enters the LSU graduation buffer. Thus, this miss indicator is instead sent with the data value.
 - For loads, this allows an accurate miss indication as the miss state must be resolved before we have the data.
 - For stores, the miss indicator is also sent with the data value. The store data value is captured when the store instruction exist the LSU graduation buffer.

11.8.11 Performance Counter Tracing

The optional feature of including performance counter values in the trace stream allows performance counter events to be correlated with the specific instruction execution path. *TraceControl3PeC* indicates if this optional feature is implemented, and the feature is enabled via *TraceControl3PeCE* and *TCBCONTROLEPeCE*.

Performance counters are traced out based on four specific events. When an enabled event occurs, up to four performance counters are traced and a fifth value is traced by the core. The fifth value is the cycle count register. Tracing the cycle count register is unique to the core.

Control over which particular performance counter is traced is specified by bit *PCTD* in each *Performance Counter Control* register. If set to zero (default setting), tracing is enabled for this performance counter, and if set to one, tracing is disabled for this performance counter. The cycle counter is traced out if at least one of the performance counter control *PCTD* bits is set, and at least one of the enabled performance counter tracing events occurs. In the case where more than one event occurs in the same cycle, the performance counter values are traced only once for that cycle.

1. Synchronization counter expiration will trigger tracing of the performance counter values. This is controlled by *TraceControl3PeCSync* and *TCBCONTROLEPeCSync*.
2. Hardware trace breakpoint will trigger tracing of the performance counter values. This is contingent on several control bit settings. The *TE* bit in the breakpoint control register should be set. This allows a trigger signal to be sent to the Trace Unit. When set, *TraceControl3PeCBP* / *TCBCONTROLEPeCBP* act as the enable for performance counter tracing. Additionally the generation of a performance counter trigger is controlled by setting active both *TraceIBPCPCT* and *TraceIBPCIE*, and, or setting active hi both *TraceDBPCPCT* and *TraceDBPCIE*. Furthermore, the *BreakPointControl* field for the specific hardware breakpoint in *TraceIBPC* or *TraceDBPC* must be encoded as 3'b100 or 3'b101 to allow performance counter values into the trace stream.
3. Function call, function return, or the occurrence of an exception will trigger tracing of the performance counter values. This is controlled by *TraceControl3PeCFCR* / *TCBCONTROLEPeCFCR*.

4. An overflow of an active performance counter will trigger tracing of the performance counter values. This is controlled by *TraceControl3PeCOvf* / *TCBCONTROLEPeCOvf*.

The Performance counter data will always use a TF3 with the *PCV* bit set to one. If the traced data is not Performance counter data, and performance counter tracing is enabled, then the *PCV* bit will be zero.

11.8.12 Filtered Data Trace Mode

This mode is used to support tracing of events in application code on a Linux system. This type of instrumented code tracing is primarily used for performance analysis, although it can also be used for event logging and debug. Filtered data tracing mode provides a mechanism to do low-overhead event tracing from user application code, since the *UserTraceData* registers require a kernel call from user mode.

In this mode, data load and store addresses are compared to the hardware data breakpoint address. If the addresses match, the data value and address associated with that match are traced out.

This mode works even when data address and/or value tracing is turned on. However, the general usage model is when both PC and Data trace are turned off since it may not always be possible to identify data that was traced due to a match vs. the regular data stream. This mode is used to shadow one or more static (fixed-address) variables. When there is a store to the variable, the store value is captured into the trace. Since there are generally two or more data triggers/watchpoints, the trace will need to uniquely identify the shadowed variable by also tracing out the associated address.

Filtered Data Trace mode is controlled by *TraceControl2FDT* / *TCBCONTROLBFDT*.

11.8.13 PC Tracing Off

PC tracing turned off is simply to disable PC tracing which is controlled by *TraceControl2Mode.PC* / *TCBCONTROLMode.PC*. Turning off this bit has more implications than simply not tracing the PC. There is some special behavior which is contingent on the setting of other mode bits.

1. PC tracing off, *TLSM=1* (*TraceControlTLSM* / *TCBCONTROLATLSM*), Address tracing=0, Data tracing=0. For data cache misses, trace out full PC and full address and the associated instruction completions. Instruction completion information not associated with a data cache miss will not appear in trace memory.
2. PC tracing off, *TLSM=1*, One or both of these modes is enabled {Address tracing, Data tracing}: PC is not traced out, but only what is enabled—for example if address tracing is enabled, the full address is traced out. Instruction completion information not associated with a traced address or a traced data will not appear in trace memory.
3. PC tracing off, *TIM=1* (*TraceControlTIM* / *TCBCONTROLATIM*), or *TFCR=1* (*TraceControlTFCR* / *TCBCONTROLATFCR*): If an instruction cache miss or function call/return occurs, the full PC is traced along with the corresponding instruction completion information.
4. PC tracing off, *TLSM=0*, *TIM=0*, *TFCR=0*: All trace messages related to instructions are disabled. TF6 with no-trace counts can still be generated if Cycle Accurate mode is enabled. TF2 should never be generated.

In addition, when PC tracing is turned off, PC-sync messages are globally disabled, except if Performance Monitoring is enabled. The reconstruction software would need a PC-sync in the case of *TLSM=1* if the PCs traced out were delta PCs. However, given that the full PC is traced, there is no need for the PC-Sync message.

When Performance Monitoring is enabled and PC tracing is off then PC-sync messages are traced. The sync messages provide periodic sampling and along with the Performance Counter data, some conditions like average IPC over fixed intervals, or cache miss ratio can be inferred from the trace.

With PC tracing disabled, there is a significant decrease in the instruction completion information that is traced. Only if the PC, address, or data have been traced out will the corresponding instruction completion also be traced; otherwise, the instruction completion is dropped.

11.8.14 TMOAS Handling

The *MIPS PDtrace™ Specification* requires a TMOAS transaction to be inserted into the trace stream. TMOAS transactions are used to record processor mode change, start or end of the tracing activity, overflow of the internal buffers in the PDtrace unit, and periodic synchronization. The following is a summary of the cases where a TMOAS transaction is generated:

- **Start of Tracing,** When tracing is first started, or when it is re-started after a break, some basic information is needed to allow external software to identify the trace start point in the static program image, and to make some reasonable conclusions about the processor mode. At the start of the tracing, a TMOAS record is sent out at the same time as the first completed instruction. This trace record type shows the processor mode and the ASID value of the currently executing processor. This record is followed by a trace of the full PC value for the first instruction traced.
- **Trace Synchronization,** The synchronization tracing function is triggered when the internal synchronization counter overflows, based on the synchronization period bits as set in *TraceControl2_{SYR}* and *TCBCONTROLA_{SYR}*. As with the start of tracing, when the synchronization period is reached, a TMOAS record is sent, followed by a full PC value. Note that the TMOAS associated with synchronization is sent only when the IPC instruction has been identified, in order to prevent other TType records between the TMOAS and the full PC trace for the synchronization.
- **Trace Overflow and Restart.** The trace unit's internal FIFO or buffers are used to hold address and data values waiting to be compressed, formatted, and traced out of the processor. It is possible to have a program sequence that overflows one or more of these FIFOs. In this situation, the core is essentially losing trace data and thus the output is no longer a true representation of the program execution sequence. In this situation, the abandon tracing in the current cycle, discard all entries in the FIFO, and restart tracing from the next completed instruction in the following cycle. In this situation, a TMOAS record is first sent after the overflow.
- **Tracing During Processor Mode Changes.** During normal execution, the processor will change its operation mode frequently. For example, when executing user-level code, an interrupt may cause the processor to jump to kernel mode to service the interrupt. When the interrupt has been serviced, the processor will return to user mode. A mode change is indicated in the tracing logic by tracing out a TMOAS for TType. In the situation that the mode change affects tracing—for example, the tracing system has been set up to trace only in user mode and not in kernel mode—then the interrupt service routine should not be traced. Upon jumping to kernel mode, the core tracing logic will add a TMOAS as the last record. When jumping from a non-tracing mode to a tracing mode, the first record output is TMOAS to indicate the mode change. This is followed by a full PC value of the first instruction in the tracing mode. This will enable the external trace reconstruction software to re-synchronize itself and track program execution in the desired mode.

Figure 11.31 and Table 11.36 describe the bit fields for a TMOAS record. A TMOAS record is usually associated with an instruction, except for the case of a trace end TMOAS, where a TMOAS is sent out because the processor enters a non-tracing mode. In this case, the TMOAS is not associated with any instruction because the processor is not tracing, and some of the fields in the TMOAS record can be invalid data, for example, the ISAM field can be indeterminate. This should not present an issue for the software, because this TMOAS is only used as an indication that the trace has ended.

Figure 11.31 A TMOAS Trace Record

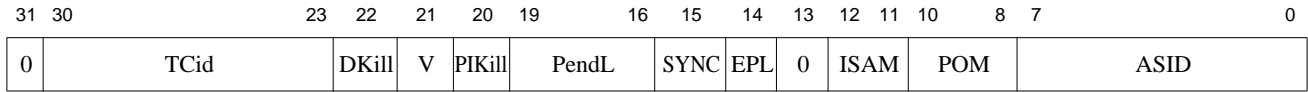


Table 11.36 TMOAS Trace Record Field Descriptions

Fields		Description
Name	Bits	
<i>TCid</i>	30:23	Id of the TC that corresponds to the <i>DKill</i> signal assertion described below. Only required if the processor implements MT; otherwise reserved.
<i>DKill</i>	22	When a ITC data instruction is killed for a given TC, this is indicated by asserting a TMOAS record with this bit set and a <i>TCid</i> value. When this bit is not set to one, it indicates that no Data kill information is valid in this TMOAS record. Only required if the processor implements MT; otherwise reserved.
<i>V</i>	21	This bit determines whether or not only the <i>DKill</i> bits are valid in this TMOAS record, or the entire TMOAS record is valid. That is, if <i>V</i> is 0, then all defined TMOAS bits are valid, and if <i>V</i> is 1, then only bits 30:22 are valid. Only required if the processor implements MT; otherwise ignored.
<i>PIKill</i>	20	This bit indicates that the instruction just previously traced was actually killed after it was traced. This scenario is possible in some situations where for example, an exception is taken after the ER stage of the ALU pipe. There are at least two cases to consider: <ul style="list-style-type: none"> • If an exception happens after ER when tracing a LW/SW accessing ITC memory in a core implementing MT. • If in an MT core, a TC is halted while executing Wait, Yield, or an instruction accessing ITC memory. Only required if the processor implements MT; otherwise ignored.
<i>PendL</i>	19:16	This field is valid only when <i>SYNC</i> is 1. When <i>SYNC</i> is 1, this field indicates the number of outstanding loads and stores at the IPC cycle. If the number of load/stores is zero, then all data transmissions' TDs thereafter are ignored until the next load/store instruction, at which point counting is restarted. Such TD transmissions are from store instructions which could not complete before the IPC signal was sent. Note that a sync occurs with an <i>InsComp</i> value of IPC (indicating that the instruction completed this cycle was a PC SYNC). Depending on whether or not there is data internally buffered and waiting to be sent, the accompanying TMOAS may not be sent until several cycles later. In the meantime, any data sent in between the IPC and the TMOAS record may be ignored (at trace start or after an overflow) since this belongs to load and store instructions that happened before the sync. Now, if there are any load or store instructions between the IPC and the TMOAS, then the data for this will only be seen after the TMOAS is transmitted, since they would get buffered behind the TMOAS.

Fields		Description																		
Name	Bits																			
<i>SYNC</i>	15	When 0, this record was sent when the <i>ASID</i> , <i>POM</i> , or <i>ISAM</i> changed. When 1, this record was sent for a synchronization event.																		
<i>EPL</i>	14	When 1, the <i>PendL</i> field is to be interpreted as (<i>PendL</i> + 16). When 0, the <i>PendL</i> field is interpreted by itself. This is introduced in PDtrace rev. 6.00																		
<i>ISAM</i>	12:11	Indicates ISA mode: <table border="1" data-bbox="690 506 1216 695"> <thead> <tr> <th>Value</th> <th>In Architecture Mode</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>MIPS32</td> </tr> <tr> <td>01</td> <td>MIPS64</td> </tr> <tr> <td>10</td> <td>MIPS16e from MIPS32 mode</td> </tr> <tr> <td>11</td> <td>MIPS16e from MIPS64 mode</td> </tr> </tbody> </table>	Value	In Architecture Mode	00	MIPS32	01	MIPS64	10	MIPS16e from MIPS32 mode	11	MIPS16e from MIPS64 mode								
Value	In Architecture Mode																			
00	MIPS32																			
01	MIPS64																			
10	MIPS16e from MIPS32 mode																			
11	MIPS16e from MIPS64 mode																			
<i>POM</i>	10:8	Indicates processor mode: <table border="1" data-bbox="607 764 1299 1100"> <thead> <tr> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>000</td> <td>Kernel Mode (<i>EXL</i> = 0, <i>ERL</i> = 0)</td> </tr> <tr> <td>001</td> <td>Exception Mode (<i>EXL</i> = 1, <i>ERL</i> = 0)</td> </tr> <tr> <td>010</td> <td>Exception Mode (<i>EXL</i> = don't care, <i>ERL</i> = 1)</td> </tr> <tr> <td>011</td> <td>Debug Mode</td> </tr> <tr> <td>100</td> <td>Supervisor Mode</td> </tr> <tr> <td>101</td> <td>User Mode</td> </tr> <tr> <td>110</td> <td>Reserved</td> </tr> <tr> <td>111</td> <td>Reserved</td> </tr> </tbody> </table>	Value	Description	000	Kernel Mode (<i>EXL</i> = 0, <i>ERL</i> = 0)	001	Exception Mode (<i>EXL</i> = 1, <i>ERL</i> = 0)	010	Exception Mode (<i>EXL</i> = don't care, <i>ERL</i> = 1)	011	Debug Mode	100	Supervisor Mode	101	User Mode	110	Reserved	111	Reserved
Value	Description																			
000	Kernel Mode (<i>EXL</i> = 0, <i>ERL</i> = 0)																			
001	Exception Mode (<i>EXL</i> = 1, <i>ERL</i> = 0)																			
010	Exception Mode (<i>EXL</i> = don't care, <i>ERL</i> = 1)																			
011	Debug Mode																			
100	Supervisor Mode																			
101	User Mode																			
110	Reserved																			
111	Reserved																			
<i>ASID</i>	7:0	The <i>ASID</i> of the current process. If the processor does not implement the standard TLB-based MMU, this field is always traced as a zero because the <i>EntryHi</i> register, and hence the <i>ASID</i> , is not defined.																		
R	31,13	Reserved for future use.																		

11.8.15 Memory-mapped Access to On-Chip Trace RAM

The main access mode to the on-chip trace memory is provided by the TAP Probe using the EJTAG Tap access port to the Trace unit. The on-chip trace memory can also be accessed directly by software using load and store instructions. Access is provided by mapping the TCB registers to drseg address space, which allows them to be accessed by software in debug mode. Because the TCB registers that are accessed indirectly via *TCBData* by the TAP Probe are mapped directly to drseg, the *TCBData* register does not need to be mapped.

The mapped drseg registers are shown in [Table 11.37](#). These mappings are “active” only when an external probe is either not present or not enabled (i.e., the *ProbEN* bit in the *EJTAG Control Register* or *ECR* is set to zero). If the mappings are active, writes to the TCB registers via drseg are enabled (so long as these writes are otherwise permitted). If the mappings are inactive, writes to the TCB registers via drseg are ignored. Note that a hardware probe could set the *ProbEN* bit to zero and still access the TCBControl registers. Writing the TCB registers via the probe and drseg simultaneously will result in unpredictable behavior. Software should not rely on reads from the TCB registers via drseg to return reliable data when the mappings are inactive. If the mappings are active on reset (i.e., *ProbEN*=0), software is responsible for initializing all control register fields, except for *TCBCONTROLA_{On}* and *TCBCONTROLB_{En}*; those control bits are set to zero on a core reset if the drseg mappings are active.

Table 11.37 Mapping TCB Registers in drseg

Offset in drseg	Register Name	Description
0x3000	<i>TCBControlA</i>	The TCBControlA register. See Section 11.10.1 “TCBCONTROLA Register” for more details about register contents.
0x3008	<i>TCBControlB</i>	The TCBControlB register. See Section 11.10.2 “TCBCONTROLB Register” for more details about register contents.
0x3010	<i>TCBControlC</i>	The TCBControlC register. See Section 11.10.4 “TCBCONTROLC Register” for more details about register contents.
0x3020	<i>TCBControlE</i>	The TCBControlE register. See Section 11.10.5 “TCBCONTROLE Register” for more details about register contents.
0x3028	<i>TCBConfig</i>	The TCBConfig register. See Section 11.10.6 “TCBCONFIG Register (Reg 0)” for more details about register contents.
0x3100	<i>TCBTW</i>	Trace Word read register. This register holds the Trace Word just read from on-line trace memory. See Section 11.10.7 “TCBTW Register (Reg 4)” for more details about register contents.
0x3108	<i>TCBRDP</i>	Trace Word Read pointer. It points to the location in the on-line trace memory where the next Trace Word will be read. A TW read has the side-effect of post-incrementing this register value to point to the next TW location. (A maximum value wraps the address around to the beginning of the trace memory). See Section 11.10.8 “TCBRDP Register (Reg 5)” for more details about register contents.
0x3110	<i>TCBWRP</i>	Trace Word Write pointer. It points to the location in the on-line trace memory where the next new Trace Word will be written. See Section 11.10.9 “TCBWRP Register (Reg 6)” for more details about register contents.
0x3118	<i>TCBSTP</i>	Trace Word Start Pointer. It points to the location of the oldest TW in the on-chip trace memory. See Section 11.10.10 “TCBSTP Register (Reg 7)” for more details about register contents.
0x3120	<i>BKUPRDP</i>	This is not a TCB register, but is needed on a reset to save the <i>TCBRDP</i> value before that register is reset to 0. This allows the software that comes up after a (hard or soft) reset to know the last-known good value of <i>TCBRDP</i> before system crash, and potentially read the trace memory from or to the appropriate trace memory location.
0x3128	<i>BKUPWRP</i>	This is not a TCB register, but needed on a reset to save the <i>TCBWRP</i> value before that register is reset to 0. This allows the software that comes up after a (hard or soft) reset to know the last known good value of <i>TCBWRP</i> before system crash, and potentially read the trace memory from or to the appropriate trace memory location.
0x3130	<i>BKUPSTP</i>	This is not a TCB register, but is needed on a reset to save the <i>TCBSTP</i> value before that register is reset to 0. This allows the software that comes up after a (hard or soft) reset to know the last known good value of <i>TCBSTP</i> before system crash, and potentially read the trace memory from or to the appropriate trace memory location.
0x3200-0x3238	<i>TCBTrigX</i>	The <i>TCBTrigX</i> set of registers. The number of implemented registers is determined by the value in <i>TCBCONFIG_{TRIG}</i> . See Section 11.10.11 “TCBTRIGx Register (Reg 16-23)” for more details about register contents.

On-chip trace memory can be read by doing a load instruction to *TCBTW*. Accessing the *TCBTW* has the side effect of automatically incrementing the value of *TCBRDP* to the next trace word. The trace memory cannot be written to via this mechanism. Software can also do direct loads and stores to *TCBRDP* and *TCBWRP* at the beginning of the

trace memory dump function. Note that writing to these registers in the middle of the trace logic writing into this memory can result in UNPREDICTABLE results and junked values in the trace memory.

Whether or not software has access to on-chip trace memory is controlled by *TCBCONTROLB_{TRPAD}*. This is a control disable bit. The bit in *TCBCONTROLB* is mirrored in *TraceControl3*. To access the on-chip memory control registers, namely the memory pointers, the *TCBTW*, and both of the backup pointer bits, *TRPAD* and *ProbEN*, must be zero. To access the other registers, it is sufficient to set the *ProbEN* bit to zero. Regardless of the setting of *ProbEN* and *TRPAD*, all the registers listed in Table 11.37 can be read out by software.

Tracing is stopped when the system crashes and an exception handler is invoked. The last known valid values of *TCBRDP*, *TCBWRP*, and *TCBSTP* are saved in the backup registers shown in the table. Software should not rely on *TCBRDP*, *TCBWRP*, and *TCBSTP* holding their last known good values across a reset, and should use the backup registers for this purpose.

11.8.16 Core-Specific Event Inefficiency Tracing

It is possible the trace can relay some hints as to the reason for loss of execution performance. This is done by enabling the core-specific inefficiency tracing via *TraceControl_{bit28} / TCBCONTROLA_{bit28}*. The inefficiency is determined at the same point that the PC, address, data, etc. is captured from the core pipeline. That is, the inefficiency is determined in the last pipestage of the graduation. An inefficiency code applies only when there is no instruction graduating. In other words, the inefficiency replaces an “NI”.

When inefficiency tracing is enabled, the instruction completion indicator in all trace formats that have an instruction completion field will increase from three bits to four bits. Reconstruction software will look for the extra bit, and when the msb of the instruction completion is set to 1'b1, which indicates that the inefficiency code is valid. If the msb of the instruction completion is set to 1'b0, then a valid instruction has graduated and there is no inefficiency.

The inefficiency events defined are the following:

1. Load/store cache miss is the reason for the “NI”.
2. Branch/return misprediction is the reason for the “NI”.
3. Replay of a load consumer, or a branch likely, or a cacheop is the reason for the “NI”.
4. Graduation stall, due to the backpressure of the Load Store Graduation Buffer (LSGB) full or some other stall from the core is the reason for the “NI”.

11.8.17 Trace Message Format

The TCB collects trace information every cycle from the PDtrace interface. This information is collected into six different Trace Formats (TF1 to TF6). All Trace Formats have at least one non-zero bit, which prevents the reconstruction software from incorrectly detecting an end of trace.

11.8.18 Trace Word Format

After the PDtrace data has been encoded in Trace Formats, the trace information must be streamed to either on-chip trace memory or to the trace probe. Each of the major Trace Formats are of different size, which complicates their efficient storage in an on-chip memory of fixed width and their transmission through a fixed-width trace probe interface to off-chip memory. To minimize memory overhead and/or bandwidth-loss, the Trace Formats are collected into Trace Words of fixed width.

A Trace Word (TW) is defined to be 64 bits wide. An empty/invalid TW contains all zeros. A TW which contains one or more valid TFs is guaranteed to have a non-zero value in one of the four least-significant bits [3:0]. During operation of the TCB, each TW is built from the TFs generated each clock cycle. When all 64 bits are used, the TW is full and can be sent to either on-chip trace memory or to the trace probe.

11.9 PDtrace™ Registers (Software Control)

The CP0 registers associated with PDtrace are listed in [Table 11.38](#) and described in [Chapter 7, “CP0 Registers of the 74K™ Core”](#).

Table 11.38 A List of Coprocessor 0 Trace Registers

Register Number	Sel	Register Name	Reference
23	1	<i>TraceControl</i>	Section 7.2.31 “Trace Control Register (CP0 Register 23, Select 1)”
23	2	<i>TraceControl2</i>	Section 7.2.32 “Trace Control2 Register (CP0 Register 23, Select 2)”
24	2	<i>TraceControl3</i>	Section 7.2.37 “Trace Control3 Register (CP0 Register 24, Select 2)”
23	3	<i>UserTraceData1</i>	Section 7.2.33 “User Trace Data1 Register (CP0 Register 23, Select 3) and User Trace Data2 Register (CP0 Register 24, Select 3)”
24	3	<i>UserTraceData2</i>	Section 7.2.33 “User Trace Data1 Register (CP0 Register 23, Select 3) and User Trace Data2 Register (CP0 Register 24, Select 3)”
23	4	<i>TraceIBPC</i>	Section 7.2.34 “TraceIBPC Register (CP0 Register 23, Select 4)”
23	5	<i>TraceDBPC</i>	Section 7.2.35 “TraceDBPC Register (CP0 Register 23, Select 5)” Section 7.2.34 “TraceIBPC Register (CP0 Register 23, Select 4)”

11.10 Trace Control Block (TCB) Registers (Hardware Control)

The registers used to control TCB operations are described in [Table 11.39](#) and [Table 11.40](#). These registers are accessed via the EJTAG TAP interface.

Table 11.39 TCB EJTAG Registers

EJTAG Register	Name	Description	Implemented
0x10	<i>TCBCTRLOLA</i>	Control register in the TCB mainly used for controlling the trace input signals to the core on the PDtrace interface. See Section 11.10.1 “TCBCTRLOLA Register” .	Yes
0x11	<i>TCBCTRLORB</i>	Control register in the TCB that is mainly used to specify what to do with the trace information. The <i>REG</i> [25:21] field in this register specifies the number of the TCB internal register accessed by the <i>TCBDATA</i> register. A list of all the registers that can be accessed by the <i>TCBDATA</i> register is shown in Table 11.40 . See Section 11.10.2 “TCBCTRLORB Register” .	Yes
0x12	<i>TCBDATA</i>	This is used to access registers specified by the <i>REG</i> field in the <i>TCBCTRLORB</i> register. See Section 11.10.3 “TCBDATA Register” .	Yes

Table 11.39 TCB EJTAG Registers

EJTAG Register	Name	Description	Implemented
0x13	<i>TCBCONTROLC</i>	Control Register in the TCB used to control and hold tracing information. See Section 11.10.4 “TCBCONTROLC Register”.	Yes
0x16	<i>TCBCONTROLE</i>	Control Register in the TCB used to control tracing for the performance counter tracing feature. See Section 11.10.5 “TCBCONTROLE Register”.	Yes

Table 11.40 Registers Selected by TCBCONTROLB

TCBCONTROLB _{REG} Field	Name	Reference	Implemented
0	<i>TCBCONFIG</i>	Section 11.10.6 “TCBCONFIG Register (Reg 0)”	Yes
4	<i>TCBTW</i>	Section 11.10.7 “TCBTW Register (Reg 4)”	Yes if on-chip memory exists. Otherwise No
5	<i>TCBRDP</i>	Section 11.10.8 “TCBRDP Register (Reg 5)”	
6	<i>TCBWRP</i>	Section 11.10.9 “TCBWRP Register (Reg 6)”	
7	<i>TCBSTP</i>	Section 11.10.10 “TCBSTP Register (Reg 7)”	
16-23	<i>TCBTRIG_x</i>	Section 11.10.11 “TCBTRIG _x Register (Reg 16-23)”	Only the number indicated by <i>TCBCONFIG_{TRIG}</i> are implemented.

11.10.1 TCBCONTROLA Register

The TCB is responsible for asserting or de-asserting the trace input control signals on the PDtrace interface to the core’s tracing logic. Most of the control is done using the *TCBCONTROLA* register.

The *TCBCONTROLA* register is written by an EJTAG TAP controller instruction, TCBCONTROLA (0x10).

The format of the *TCBCONTROLA* register is shown below, and the fields are described in Table 11.41.

Figure 11.32 TCBCONTROLA Register Format

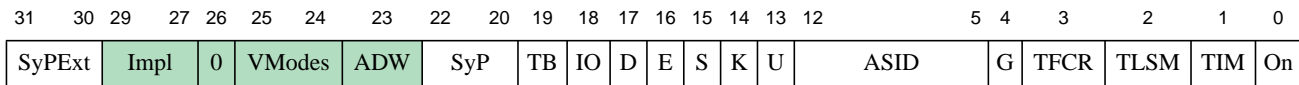


Table 11.41 TCBCONTROLA Register Field Descriptions

Fields		Description	Read / Write	Reset State										
Name	Bits													
<i>SyPExt</i>	31:30	Extension to the <i>SyP</i> (sync period) field for implementations that need higher numbers of cycles between synchronization events. The value of <i>SyP</i> is extended by assuming that these two bits are juxtaposed to the left of the three bits of <i>SyP</i> (<i>SyPExt.SyP</i>). When only <i>SyP</i> was used to specify the synchronization period, the value was 2^x , where x was computed from <i>SyP</i> by adding 5 to the actual value represented by the bits. A similar formula is applied to the 5 bits obtained by the juxtaposition of <i>SyPExt</i> and <i>SyP</i> . Sync period values greater than 2^{31} are UNPREDICTABLE. That is all values greater than 11010 ($26+5=31$) are UNPREDICTABLE. With <i>SyPExt</i> bits, a sync period range of 2^5 to 2^{31} cycles can be obtained.	R/W	0										
<i>Impl</i>	29	Reserved for implementation-specific use.	R	0										
<i>Ineff</i>	28	Core-specific inefficiency tracing is enabled. If enabled core-specific trace information is included in the trace stream. The inefficiency code replaces an “NI” and is interpreted in the trace stream with an expanded InsComp. The InsComp is expanded from 3b to 4b for all trace formats.	R/W	0										
<i>Impl</i>	27	Reserved for implementation-specific use.	R	0										
0	26	Reserved. Must be written as zero; returns zero on read.	R	0										
<i>VModes</i>	25:24	This field specifies the type of tracing that is supported by the processor: <table border="1" data-bbox="464 1052 1084 1243"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>PC tracing only</td> </tr> <tr> <td>01</td> <td>PC and Load and store address tracing only</td> </tr> <tr> <td>10</td> <td>PC, load and store address, and load and store data.</td> </tr> <tr> <td>11</td> <td>Reserved</td> </tr> </tbody> </table> This field is preset to the value of <i>ValidModes</i> .	Encoding	Meaning	00	PC tracing only	01	PC and Load and store address tracing only	10	PC, load and store address, and load and store data.	11	Reserved	R	10
Encoding	Meaning													
00	PC tracing only													
01	PC and Load and store address tracing only													
10	PC, load and store address, and load and store data.													
11	Reserved													
<i>ADW</i>	23	The data value width used in the trace formats: <table border="1" data-bbox="464 1339 1084 1455"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>width is 16 bits</td> </tr> <tr> <td>1</td> <td>width is is 32 bits</td> </tr> </tbody> </table>	Encoding	Meaning	0	width is 16 bits	1	width is is 32 bits	R	1				
Encoding	Meaning													
0	width is 16 bits													
1	width is is 32 bits													

Table 11.41 TCBCONTROLA Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State																		
Name	Bits																					
<i>SyP</i>	22:20	Used to indicate the synchronization period. The period (in cycles) between which the periodic synchronization information is to be sent is defined as shown in the table below. <table border="1" data-bbox="618 443 930 833" style="margin: 10px auto;"> <thead> <tr> <th>SyP</th> <th>Sync Period</th> </tr> </thead> <tbody> <tr> <td>000</td> <td>2^5</td> </tr> <tr> <td>001</td> <td>2^6</td> </tr> <tr> <td>010</td> <td>2^7</td> </tr> <tr> <td>011</td> <td>2^8</td> </tr> <tr> <td>100</td> <td>2^9</td> </tr> <tr> <td>101</td> <td>2^{10}</td> </tr> <tr> <td>110</td> <td>2^{11}</td> </tr> <tr> <td>111</td> <td>2^{12}</td> </tr> </tbody> </table>	SyP	Sync Period	000	2^5	001	2^6	010	2^7	011	2^8	100	2^9	101	2^{10}	110	2^{11}	111	2^{12}	R/W	000
SyP	Sync Period																					
000	2^5																					
001	2^6																					
010	2^7																					
011	2^8																					
100	2^9																					
101	2^{10}																					
110	2^{11}																					
111	2^{12}																					
<i>TB</i>	19	Trace All Branches. When set to one, this field indicates that the core must trace either full or incremental PC values for all branches. When set to zero, only the unpredictable branches are traced.	R/W	Undefined																		
<i>IO</i>	18	Inhibit Overflow. This bit is used to indicate to the core trace logic that slow but complete tracing is desired. Hence, the core tracing logic must not allow a FIFO overflow and discard trace data. This is achieved by stalling the pipeline when the FIFO is nearly full so that no trace records are ever lost.	R/W	Undefined																		
<i>D</i>	17	When set to one, this enables tracing in Debug mode, i.e., when the <i>DM</i> bit is one in the <i>Debug</i> register. For trace to be enabled in Debug mode, the <i>On</i> bit must be one and either the <i>G</i> bit must be one, or the current process must match the <i>ASID</i> field in this register. When set to zero, trace is disabled in Debug mode, irrespective of other bits.	R/W	Undefined																		
<i>E</i>	16	This controls when tracing is enabled. When set, tracing is enabled when either of the <i>EXL</i> or <i>ERL</i> bits in the <i>Status</i> register is one, provided that the <i>On</i> bit (bit 0) is also set, and either the <i>G</i> bit is set, or the current process ASID matches the <i>ASID</i> field in this register.	R/W	Undefined																		
<i>S</i>	15	When set, this enables tracing when the core is in Supervisor mode as defined in the MIPS32 or MIPS64 architecture specification. This is provided the <i>On</i> bit (bit 0) is also set, and either the <i>G</i> bit is set, or the current process ASID matches the <i>ASID</i> field in this register.	R/W	Undefined																		
<i>K</i>	14	When set, this enables tracing when the <i>On</i> bit is set and the core is in Kernel mode. Unlike the usual definition of Kernel Mode, this bit enables tracing only when the <i>ERL</i> and <i>EXL</i> bits in the <i>Status</i> register are zero. This is provided the <i>On</i> bit (bit 0) is also set, and either the <i>G</i> bit is set, or the current process ASID matches the <i>ASID</i> field in this register.	R/W	Undefined																		

Table 11.41 TCBCONTROLA Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
<i>U</i>	13	When set, this enables tracing when the core is in User mode as defined in the MIPS32 or MIPS64 architecture specification. This is provided the <i>On</i> bit (bit 0) is also set, and either the <i>G</i> bit is set, or the current process ASID matches the <i>ASID</i> field in this register.	R/W	Undefined
<i>ASID</i>	12:5	The ASID field to match when the <i>G</i> bit is zero. When the <i>G</i> bit is one, this field is ignored.	R/W	Undefined
<i>G</i>	4	When set, this implies that tracing is to be enabled for all processes, provided that other enabling functions (like <i>U</i> , <i>S</i> , etc..) are also true.	R/W	Undefined
<i>TFCR</i>	3	When set, this indicates to the PDtrace interface that the optional <i>Fcr</i> bit must be traced in the appropriate trace formats. If PC tracing is disabled, the full PC of the function call (or return) instruction must also be traced. Note that function call/return information is only traced if tracing is actually enabled for the current mode.	R/W	Undefined
<i>TLSM</i>	2	When set, this indicates to the PDtrace interface that information about data cache misses should be traced. If PC, load/store addresses and data tracing are disabled (see <i>TraceControlMode</i> field), the full PC and load/store address are traced for data cache misses. If load/store data tracing is enabled, the <i>LSm</i> bit must be traced in the appropriate trace format. Note that data cache miss information is only traced if tracing is actually enabled for the current mode.	R/W	Undefined
<i>TIM</i>	1	When set, this indicates to the PDtrace interface that the optional <i>Im</i> bit must be traced in the appropriate trace formats. If PC tracing is disabled, the full PC of the instruction that missed in the I-cache must be traced. Note that instruction cache miss information is only traced if tracing is actually enabled in the current mode.	R/W	Undefined
<i>On</i>	0	This is the global trace enable switch to the core. When zero, tracing from the core is always disabled, unless enabled by core internal software override. When set to one, tracing is enabled whenever the other enabling functions are also true.	R/W	0

11.10.2 TCBCONTROLB Register

The TCB includes a second control register, *TCBCONTROLB* (0x11). This register generally controls what to do with the trace information received.

The format of the *TCBCONTROLB* register is shown below, and the fields are described in [Table 11.42](#).

Figure 11.33 TCBCONTROLB Register Format

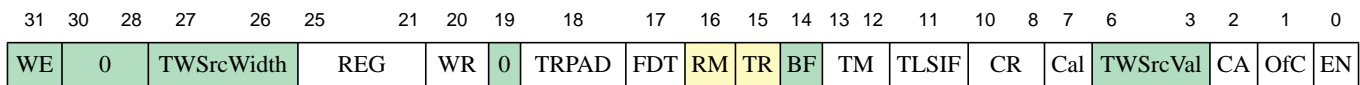


Table 11.42 TCBCONTROLB Register Field Descriptions

Fields		Description	Read / Write	Reset State										
Name	Bits													
<i>WE</i>	31	Write Enable. Only when set to 1 will the other bits be written in <i>TCBCONTROLB</i> . This bit will always read 0.	R	0										
0	30:28	Reserved. Must be written as zero; returns zero on read.	R	0										
<i>TWSrcWidth</i>	27:26	Used to indicate the number of bits used in the source field of the Trace Word, this is a configuration option of the core that cannot be modified by software. <table border="1" data-bbox="581 596 1008 785"> <thead> <tr> <th>Encoding</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>Zero source field width</td> </tr> <tr> <td>01</td> <td>2- bit source field width</td> </tr> <tr> <td>10</td> <td>4- bit source field width</td> </tr> <tr> <td>11</td> <td>Reserved for future use</td> </tr> </tbody> </table> <p>This field can either be 00, 01, or 10 for the 74K core.</p>	Encoding	Meaning	00	Zero source field width	01	2- bit source field width	10	4- bit source field width	11	Reserved for future use	R	Preset
Encoding	Meaning													
00	Zero source field width													
01	2- bit source field width													
10	4- bit source field width													
11	Reserved for future use													
<i>REG</i>	25:21	Register select: This field select the registers accessible through the <i>TCBDATA</i> register. Legal values are shown in Table 11.40 .	R/W	0										
<i>WR</i>	20	Write Registers: When set, the register selected by <i>REG</i> field is read and written when <i>TCBDATA</i> is accessed. Otherwise the selected register is only read.	R/W	0										
0	19	Reserved. Must be written as zero; returns zero on read.	R	0										
<i>TRPAD</i>	18	Trace RAM access disable bit, disables program software access to the on-chip trace RAM using load/store instructions. When this bit is set, that is, the access is disabled, then software access to the on-chip memory is disabled. If probe access is not provided in the implementation, then this register bit must be tied to zero value to allow software to control access.	R/W	0										
<i>FDT</i>	17	Filtered Data Trace Mode enable bit. When the bit is 0, this mode is disabled, reset value is disable. When set to 1, this mode is enabled. This mode is described in Section 11.8.12 “Filtered Data Trace Mode”	R/W	0										
<i>RM</i>	16	Read on-chip trace memory. When written to 1, the read address-pointer of the on-chip memory is set to point to the oldest memory location written since the last reset of pointers. Subsequent access to the <i>TCBTW</i> register (through the <i>TCBDATA</i> register), will automatically increment the read pointer (<i>TCBRDP</i> register) after each read. Note: The read pointer does not auto-increment if the <i>WR</i> field is one. When the write pointer is reached, this bit is automatically reset to 0, and the <i>TCBTW</i> register will read all zeros. Once set to 1, writing 1 again will have no effect. The bit is reset by setting the <i>TR</i> bit or by reading the last Trace word in <i>TCBTW</i> . This bit is reserved if on-chip memory is not implemented	R/W1	0										

Table 11.42 TCBCONTROLB Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State										
Name	Bits													
<i>TR</i>	15	Trace memory reset. When written to one, the address pointers for the on-chip trace memory are reset to zero. Also the <i>RM</i> bit is reset to 0. This bit is automatically de-asserted back to 0 when the reset is completed. This bit is reserved if on-chip memory is not implemented.	R/W1	0										
<i>BF</i>	14	Buffer Full. Indicator used by the TCB to communicate to external software that the on-chip trace memory is being deployed in the trace-from and trace-to mode. (See Section 11.14 “TCB On-Chip Trace Memory”) This bit is cleared when writing 1 to the <i>TR</i> bit This bit is reserved if on-chip memory is not implemented.	R	0										
<i>TM</i>	13:12	Trace Mode. This field determines how the trace memory is filled when using the simple-break control in the PDtrace interface to start or stop trace. <table border="1" data-bbox="591 877 997 1066"> <thead> <tr> <th>TM</th> <th>Trace Mode</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>Trace-To</td> </tr> <tr> <td>01</td> <td>Trace-From</td> </tr> <tr> <td>10</td> <td>Reserved</td> </tr> <tr> <td>11</td> <td>Reserved</td> </tr> </tbody> </table> In Trace-To mode, the on-chip trace memory is filled, continuously wrapping around and overwriting older Trace Words, as long as there is trace data coming from the core. In Trace-From mode, the on-chip trace memory is filled from the point that the core starts tracing until the on-chip trace memory is full. In both cases, de-asserting the <i>EN</i> bit in this register will also stop fill to the trace memory. If a <i>TCBTRIGx</i> trigger control register is used to start/stop tracing, then this field should be set to Trace-To mode. This bit is reserved if on-chip memory is not implemented.	TM	Trace Mode	00	Trace-To	01	Trace-From	10	Reserved	11	Reserved	R/W	0
TM	Trace Mode													
00	Trace-To													
01	Trace-From													
10	Reserved													
11	Reserved													
<i>TLSIF</i>	11	When set, this indicates to the TCB that information about Load and Store data cache miss, instruction cache miss, and function call are to be taken from the PDtrace interface and trace them out in the appropriate trace formats as the three optional bits <i>LSm</i> , <i>Im</i> , and <i>Fcr</i> .	R/W	0										
<i>CR</i>	10:8	Off-chip Clock Ratio. Writing this field, sets the ratio of the core clock to the off-chip trace memory interface clock. The clock-ratio encoding is shown in Table 11.43. Remark: As the Probe interface works in double data rate (DDR) mode, a 1:2 ratio indicates one data packet sent per core clock rising edge. This bit is reserved if off-chip trace option is not implemented.	R/W	100										

Table 11.42 TCBCONTROLB Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State																																																												
Name	Bits																																																															
<i>Cal</i>	7	<p>Calibrate off-chip trace interface.</p> <p>If set to one, the off-chip trace pins will produce the following pattern in consecutive trace clock cycles. If more than 4 data pins exist, the pattern is replicated for each set of 4 pins. The pattern repeats from top to bottom until the Cal bit is de-asserted.</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th colspan="4">Calibrations pattern</th> </tr> <tr> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td><td>1</td></tr> </tbody> </table> <p style="text-align: center; margin-left: 100px;">This pattern is replicated for every 4 bits of TR_DATA pins.</p> <p>Note: The clock source of the TCB and PIB must be running. This bit is reserved if off-chip trace option is not implemented.</p>	Calibrations pattern				3	2	1	0	0	0	0	0	1	1	1	1	0	0	0	0	0	1	0	1	1	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	1	1	1	0	1	1	0	1	1	0	1	1	0	1	1	1	R/W	0
Calibrations pattern																																																																
3	2	1	0																																																													
0	0	0	0																																																													
1	1	1	1																																																													
0	0	0	0																																																													
0	1	0	1																																																													
1	0	1	0																																																													
1	0	0	0																																																													
0	1	0	0																																																													
0	0	1	0																																																													
0	0	0	1																																																													
1	1	1	0																																																													
1	1	0	1																																																													
1	0	1	1																																																													
0	1	1	1																																																													
<i>TWSrcVal</i>	6:3	<p>These bits are used to indicate the value of the TW source field that will be traced if <i>TWSrcWidth</i> indicates a source bit-field width of 2 or 4 bits. Note that if the field is 2 bits, then only bits 4:3 of this field will be used in the TW.</p>	R	0																																																												
<i>CA</i>	2	<p>Cycle accurate trace.</p> <p>When set to 1, the trace will include stall information.</p> <p>When set to 0, the trace will exclude stall information, and remove bit zero from all transmitted TF's.</p> <p>The stall information included/excluded is:</p> <ul style="list-style-type: none"> • TF6 formats with TCBcode 0001 and 0101. • All TF1 formats. 	R/W	0																																																												
<i>OfC</i>	1	<p>If set to 1, trace is sent to off-chip memory using <i>TR_DATA</i> pins.</p> <p>If set to 0, trace info is sent to on-chip memory.</p> <p>This bit is read only if a single memory option exists (either off-chip or on-chip only).</p>	R/W	Preset																																																												

Table 11.42 TCBCONTROLB Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
<i>EN</i>	0	Enable trace. This is the master enable for trace to be generated from the TCB. This bit can be set or cleared, either by writing this register or from a start/stop/about trigger. When set to 1, trace information is sampled on the output pins (of the capture-to-compression interface) or written in the on-chip trace memory. Trace Words are generated and sent to either on-chip memory or to the Trace Probe. The target of the trace is selected by the OfC bit. When set to 0, trace information on the output pins are ignored. A potential TF6-stop (from a stop trigger) is generated as the last information, the TCB pipeline is flushed, and trace output is stopped.	R/W	0

Table 11.43 Clock Ratio encoding of the CR field

CR/CRMin/CRMax	Clock Ratio
000	8:1 (Trace clock is eight times that of core clock)
001	4:1 (Trace clock is four times that of core clock)
010	2:1 (Trace clock is double that of core clock)
011	1:1 (Trace clock is same as core clock)
100	1:2 (Trace clock is one half of core clock)
101	1:4 (Trace clock is one fourth of core clock)
110	1:6 (Trace clock is one sixth of core clock)
111	1:8 (Trace clock is one eighth of core clock)

11.10.3 TCBDATA Register

The *TCBDATA* register (0x12) is used to access the registers defined by the *TCBCONTROLB_{REG}* field; see [Table 11.40](#). Regardless of which register or data entry is accessed through *TCBDATA*, the register is only written if the *TCBCONTROLB_{WR}* bit is set. For read-only registers, the *TCBCONTROLB_{WR}* is a don't care. If software is accessing the on-chip trace memory to read out the trace words, then *TCBDATA* is not used for the indirect read of the *TCBTW*. Instead software can read from *TCBTW* directly.

The format of the *TCBDATA* register is shown below, and the field is described in [Table 11.44](#). The width of *TCBDATA* is 64 bits when on-chip trace words (TWs) are accessed (*TCBTW* access).

Figure 11.34 TCBDATA Register Format

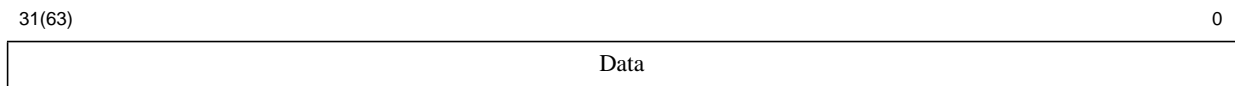


Table 11.44 TCBDATA Register Field Descriptions

Fields		Description	Read/Write	Reset State
Names	Bits			
<i>Data</i>	31:0 63:0	Register fields or data as defined by the <i>TCBCONTROLB_{REG}</i> field	Only writable if <i>TCBCONTROLB_{WR}</i> is set	0

11.10.4 TCBCONTROLC Register

The trace output from the processor on the PDtrace interface can be controlled by the trace input signals to the processor from the TCB. The TCB uses a control register, *TCBCONTROLC*, whose values are used to change the signal values on the PDtrace input interface. External software (i.e., debugger), can therefore manipulate the trace output by writing the *TCBCONTROLC* register.

The *TCBCONTROLC* register is written by an EJTAG TAP controller instruction, *TCBCONTROLC* (0x13).

The format of the *TCBCONTROLC* register is shown below, and the fields are described in [Table 11.45](#).

Figure 11.35 TCBCONTROLC Register Format

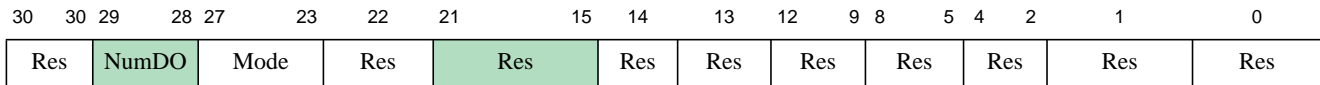


Table 11.45 TCBCONTROLC Register Field Descriptions

Fields		Description	Read / Write	Reset State												
Name	Bits															
Reserved	31:30	Reserved for future use. Must be written as zero; returns zero on read.	0	0												
<i>NumDO</i>	29:28	Specifies the number of bits needed by this implementation to specify the DataOrder: 00 - Four bits 01 - Five bits 10 - Six bits 11 - Eight bits	R	Preset												
<i>Mode</i>	27:23	When tracing is turned on, this signal specifies what information is to be traced by the core. It uses 5 bits, where each bit turns on a tracing of a specific tracing mode. <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Bit # Set</th> <th>Trace The Following</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>PC</td> </tr> <tr> <td>1</td> <td>Load address</td> </tr> <tr> <td>2</td> <td>Store address</td> </tr> <tr> <td>3</td> <td>Load data</td> </tr> <tr> <td>4</td> <td>Store data</td> </tr> </tbody> </table> <p>The table shows what trace value is turned on when that bit value is a 1. If the corresponding bit is 0, then the Trace Value shown in column two is not traced by the processor.</p>	Bit # Set	Trace The Following	0	PC	1	Load address	2	Store address	3	Load data	4	Store data	R/W	0
Bit # Set	Trace The Following															
0	PC															
1	Load address															
2	Store address															
3	Load data															
4	Store data															

Table 11.45 TCBCONTROLC Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
Res	22:0	Reserved for future use. Must be written as zero; returns zero on read.	0	0

11.10.5 TCBCONTROLE Register

The trace output from the processor on the PDtrace interface can be controlled by the trace input signals to the processor from the TCB. The TCB uses a control register, *TCBCONTROLE*, whose values are used to change the signal values on the PDtrace input interface. External software (i.e., debugger), can therefore manipulate the trace output by writing the *TCBCONTROLE* register.

The *TCBCONTROLE* register is written by an EJTAG TAP controller instruction, *TCBCONTROLE* (0x16).

The format of the *TCBCONTROLE* register is shown below, and the fields are described in [Table 11.46](#).

Figure 11.36 TCBCONTROLE Register Format

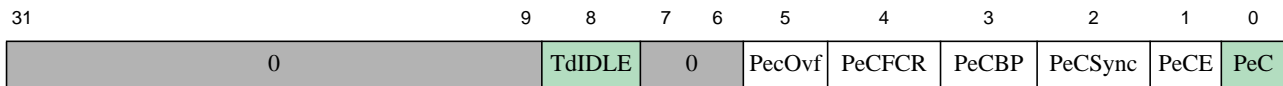


Table 11.46 TCBCONTROLE Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:9	Reserved for future use. Must be written as zero; returns zero on read.	0	0
<i>TrIDLE</i>	8	Trace Unit Idle. This bit indicates if the trace hardware is currently idle (not processing any data). This can be useful when switching control of trace from hardware to software and vice versa. The bit is read-only and updated by the trace hardware.	R	1
0	7:6	Reserved for future use; Must be written as zero; returns zero on read. (Hint to architect, Reserved for future expansion of performance counter trace events).	0	0
<i>PeCOvf</i>	5	Trace performance counters when one of the performance counters overflows its count value. Enabled when set to 1.	R/W	0
<i>PeCFCR</i>	4	Trace performance counters on function call/return or on an exception handler entry. Enabled when set to 1.	R/W	0
<i>PeCBP</i>	3	Trace performance counters on hardware breakpoint match trigger. Enabled when set to 1.	R/W	0
<i>PeCSync</i>	2	Trace performance counters on synchronization counter expiration. Enabled when set to 1.	R/W	0

Table 11.46 TCBCONTROLE Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
<i>PeCE</i>	1	Performance counter tracing enable. When set to 0, the tracing out of performance counter values as specified is disabled. To enable, this bit must be set to 1. This bit is used under software control. When trace is controlled by an external probe, this enabling is done via the <i>TCB Control</i> register.	R/W	0
<i>PeC</i>	0	Specifies whether or not Performance Control Tracing is implemented. This is an optional feature that may be omitted by implementation choice. See Section 11.8.11 “Performance Counter Tracing” for details.	R	Preset

11.10.6 TCBCONFIG Register (Reg 0)

The *TCBCONFIG* register holds information about the hardware configuration of the TCB. The format of the *TCBCONFIG* register is shown below, and the field is described in [Table 11.47](#).

Figure 11.37 TCBCONFIG Register Format

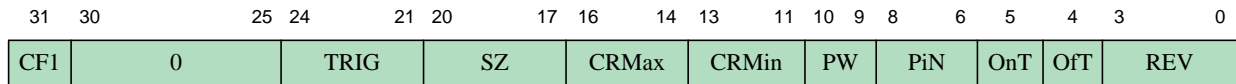


Table 11.47 TCBCONFIG Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
<i>CF1</i>	31	This bit is set if a <i>TCBCONFIG1</i> register exists. In this revision, <i>TCBCONFIG1</i> does not exist and this bit always reads zero.	R	0
0	30:25	Reserved. Must be written as zero; returns zero on read.	R	0
<i>TRIG</i>	24:21	Number of triggers implemented. This also indicates the number of <i>TCBTRIGx</i> registers that exist.	R	Preset Legal values are 0 - 8
<i>SZ</i>	20:17	On-chip trace memory size. This field holds the encoded size of the on-chip trace memory. The size in bytes is given by $2^{(SZ+8)}$, implying that the minimum size is 256 bytes and the largest is 8Mb. This bit is reserved if on-chip memory is not implemented.	R	Preset
<i>CRMax</i>	16:14	Off-chip Maximum Clock Ratio. This field indicates the maximum ratio of the core clock to the off-chip trace memory interface clock. The clock-ratio encoding is shown in Table 11.43 . This bit is reserved if off-chip trace option is not implemented.	R	Preset

Table 11.47 TCBCONFIG Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State										
Name	Bits													
<i>CRMin</i>	13:11	Off-chip Minimum Clock Ratio. This field indicates the minimum ratio of the core clock to the off-chip trace memory interface clock. The clock-ratio encoding is shown in Table 11.43. This bit is reserved if off-chip trace option is not implemented.	R	Preset										
<i>PW</i>	10:9	Probe Width: Number of bits available on the off-chip trace interface <i>TR_DATA</i> pins. The number of <i>TR_DATA</i> pins is encoded, as shown in the table. <table border="1" data-bbox="479 598 1026 787"> <thead> <tr> <th>PW</th> <th>Number of bits used on <i>TR_DATA</i></th> </tr> </thead> <tbody> <tr> <td>00</td> <td>4 bits</td> </tr> <tr> <td>01</td> <td>8 bits</td> </tr> <tr> <td>10</td> <td>16 bits</td> </tr> <tr> <td>11</td> <td>Reserved</td> </tr> </tbody> </table> This field is preset based on input signals to the TCB and the actual capability of the TCB. This bit is reserved if off-chip trace option is not implemented.	PW	Number of bits used on <i>TR_DATA</i>	00	4 bits	01	8 bits	10	16 bits	11	Reserved	R	Preset
PW	Number of bits used on <i>TR_DATA</i>													
00	4 bits													
01	8 bits													
10	16 bits													
11	Reserved													
<i>PiN</i>	8:6	Pipe number. Indicates the number of execution pipelines.	R	0										
<i>OnT</i>	5	When set, this bit indicates that on-chip trace memory is present. This bit is preset based on the selected option when the TCB is implemented.	R	Preset										
<i>OfT</i>	4	When set, this bit indicates that off-chip trace interface is present. This bit is preset based on the selected option when the TCB is implemented, and on the existence of a PIB module (<i>TC_PibPresent</i> asserted).	R	Preset										
<i>REV</i>	3:0	Revision of TCB. An implementation that conforms to PDtrace version 4.1 must have a value of 1 for this field.	R	1										

11.10.7 TCBTW Register (Reg 4)

The *TCBTW* register is used to read Trace Words from the on-chip trace memory. The Trace Word (TW) read is the one pointed to by the *TCBRDP* register. A side effect of reading the *TCBTW* register is that the *TCBRDP* register increments to the next TW in the on-chip trace memory. If *TCBRDP* is at the max size of the on-chip trace memory, the increment wraps back to address zero. The *TCBTW* register is mapped to offset 0x3100 in *drseg*. An access to offset 0x3100 automatically causes the read pointer to be incremented. The use of load half-word or load byte instructions can lead to unpredictable results, and is not recommended. The results of attempting to write to trace memory by an explicit store instruction targeting *TCBTW* are unpredictable.

This register is reserved if on-chip trace memory is not implemented.

The format of the *TCBTW* register is shown below, and the field is described in Table 11.48.

Figure 11.38 TCBTW Register Format

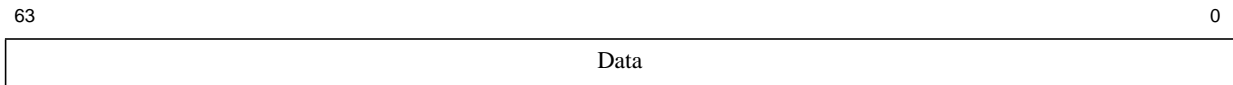


Table 11.48 TCBTW Register Field Descriptions

Fields		Description	Read / Write	Reset State
Names	Bits			
<i>Data</i>	63:0	Trace Word (TW)	R/W	0

11.10.8 TCBRDP Register (Reg 5)

The *TCBRDP* register is the address pointer to on-chip trace memory. It points to the TW read when reading the *TCBTW* register. When writing the *TCBCONTROLB_{RM}* bit to 1, this pointer is reset to the current value of *TCBSTP*.

This register is reserved if on-chip trace memory is not implemented.

The format of the *TCBRDP* register is shown below, and the field is described in [Table 11.49](#). The value of *n* depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, lower three bits are always zero.

Figure 11.39 TCBRDP Register Format

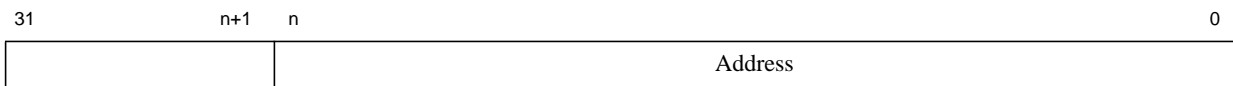


Table 11.49 TCBRDP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Names	Bits			
<i>Data</i>	31:(n+1)	Reserved. Must be written zero, reads back zero.	0	0
<i>Address</i>	n:0	Byte address of on-chip trace memory word.	R/W	0

11.10.9 TCBWRP Register (Reg 6)

The *TCBWRP* register is the address pointer to on-chip trace memory. It points to the location where the next new TW for on-chip trace will be written.

This register is reserved if on-chip trace memory is not implemented.

The format of the *TCBWRP* register is shown below, and the fields are described in [Table 11.50](#). The value of *n* depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, the lower three bits are always zero.

Figure 11.40 TCBWRP Register Format

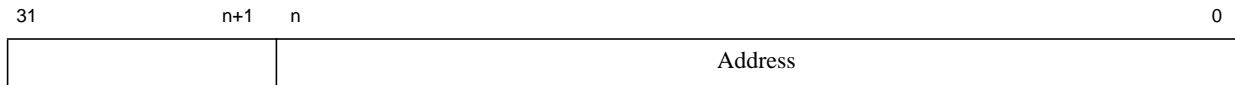


Table 11.50 TCBWRP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Names	Bits			
<i>Data</i>	31:(n+1)	Reserved. Must be written zero, reads back zero.	0	0
<i>Address</i>	n:0	Byte address of on-chip trace memory word.	R/W	0

11.10.10 TCBSTP Register (Reg 7)

The *TCBSTP* register is the start pointer register. This register points to the on-chip trace memory address at which the oldest TW is located. This pointer is reset to zero when the *TCBCONTROLB_{TR}* bit is written to 1. If a continuous trace to on-chip memory wraps around the on-chip memory, *TCBSTP* will have the same value as *TCBWRP*.

This register is reserved if on-chip trace memory is not implemented.

The format of the *TCBSTP* register is shown below, and the fields are described in Table 11.51. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, lower three bits are always zero.

Figure 11.41 TCBSTP Register Format

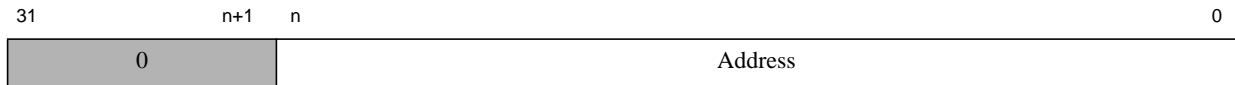


Table 11.51 TCBSTP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Names	Bits			
<i>Data</i>	31:(n+1)	Reserved. Must be written zero, reads back zero.	0	0
<i>Address</i>	n:0	Byte address of on-chip trace memory word.	R/W	0

11.10.11 TCBTRIGx Register (Reg 16-23)

Up to eight Trigger Control registers are possible. Each register is named *TCBTRIG_x*, where *x* is a single digit number from 0 to 7 (*TCBTRIG0* is Reg 16). The actual number of trigger registers implemented is defined in the *TCBCONFIG_{TRIG}* field. An unimplemented register will read all zeros and writes are ignored.

Each Trigger Control register controls when an associated trigger is fired, and the action to be taken when the trigger occurs. Please also read Section 11.12 “TCB Trigger Logic”, for detailed description of trigger logic issues.

The format of the *TCBTRIG_x* register is shown below, and the fields are described in Table 11.52.

Figure 11.42 TCBTRIGx Register Format

31	24	23	22	16	15	14	13	7	6	5	4	3	2	1	0
TCBinfo	Trace	0	CHTro	PDTro	0	DM	CHTri	PDTri	Type	FO	TR				

Table 11.52 TCBTRIGx Register Field Descriptions

Fields		Description	Read / Write	Reset State
Names	Bits			
<i>TCBinfo</i>	31:24	TCBinfo to be used in a possible TF6 trace format when this trigger fires.	R/W	0
<i>Trace</i>	23	When set, generate TF6 trace information when this trigger fires. Use <i>TCBinfo</i> field for the <i>TCBinfo</i> of TF6 and use Type field for the two MSB of the <i>TCBtype</i> of TF6. The two LSB of <i>TCBtype</i> are 00. The write value of this bit always controls the behavior of this trigger. When this trigger fires, the read value will change to indicate if the TF6 format was ever suppressed by a simultaneous trigger. If so, the read value will be 0. If the write value was 0, the read value is always 0. This special read value is valid until the <i>TCBTRIGx</i> register is written.	R/W	0
0	22:16	Reserved. Must be written as zero; returns zero on read.	R	0
<i>CHTro</i>	15	When set, generate a single cycle strobe on <i>TC_ChipTrigOut</i> when this trigger fires.	R/W	0
<i>PDTro</i>	14	When set, generate a single cycle strobe on <i>TC_ProbeTrigOut</i> when this trigger fires.	R/W	0
0	13:7	Reserved. Must be written as zero; returns zero on read.	R	0
<i>DM</i>	6	When set, this Trigger will fire when a rising edge on the Debug mode indication from the core is detected. The write value of this bit always controls the behavior of this trigger. When this trigger fires, the read value will change to indicate if this source was ever the cause of a trigger action (even if the action was suppressed). If so the read value will be 1. If the write value was 0 the read value is always 0. This special read value is valid until the <i>TCBTRIGx</i> register is written.	R/W	0
<i>CHTri</i>	5	When set, this Trigger will fire when a rising edge on <i>TC_ChipTrigIn</i> is detected. The write value of this bit always controls the behavior of this trigger. When this trigger fires, the read value will change to indicate if this source was ever the cause of a trigger action (even if the action was suppressed). If so the read value will be 1. If the write value was 0 the read value is always 0. This special read value is valid until the <i>TCBTRIGx</i> register is written.	R/W	0

Table 11.52 TCBTRIGx Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State										
Names	Bits													
<i>PDTri</i>	4	<p>When set, this Trigger will fire when a rising edge on <i>TC_ProbeTrigIn</i> is detected.</p> <p>The write value of this bit always controls the behavior of this trigger.</p> <p>When this trigger fires, the read value will change to indicate if this source was ever the cause of a trigger action (even if the action was suppressed). If so the read value will be 1. If the write value was 0 the read value is always 0. This special read value is valid until the <i>TCBTRIGx</i> register is written.</p>	R/W	0										
<i>Type</i>	3:2	<p>Trigger Type: The Type indicates the action to take when this trigger fires. The table below show the Type values and the Trigger action.</p> <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Type</th> <th>Trigger action</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>Trigger Start: Trigger start-point of trace.</td> </tr> <tr> <td>01</td> <td>Trigger End: Trigger end-point of trace.</td> </tr> <tr> <td>10</td> <td>Trigger About: Trigger center-point of trace.</td> </tr> <tr> <td>11</td> <td>Trigger Info: No action trigger, only for trace info.</td> </tr> </tbody> </table> <p>The actual action is to set or clear the <i>TCBCONTROLB_{EN}</i> bit. A Start trigger will set <i>TCBCONTROLB_{EN}</i>, a End trigger will clear <i>TCBCONTROLB_{EN}</i>. The About trigger will clear <i>TCBCONTROLB_{EN}</i> half way through the trace memory, from the trigger. The size determined by the <i>TCBCONFIG_{SZ}</i> field for on-chip memory. Or from the <i>TCBCONTROLA_{SYP}</i> field for off-chip trace.</p> <p>If Trace is set, then a TF6 format is added to the trace words. For Start and Info triggers this is done before any other TF's in that same cycle. For End and About triggers, the TF6 format is added after any other TF's in that same cycle.</p> <p>If the <i>TCBCONTROLB_{TM}</i> field is implemented, it must be set to Trace-To mode (00), for the Type field to control on-chip trace fill.</p> <p>The write value of this bit always controls the behavior of this trigger.</p> <p>When this trigger fires, the read value will change to indicate if the trigger action was ever suppressed. If so the read value will be 11. If the write value was 11 the read value is always 11. This special read value is valid until the <i>TCBTRIGx</i> register is written.</p>	Type	Trigger action	00	Trigger Start: Trigger start-point of trace.	01	Trigger End: Trigger end-point of trace.	10	Trigger About: Trigger center-point of trace.	11	Trigger Info: No action trigger, only for trace info.	R/W	0
Type	Trigger action													
00	Trigger Start: Trigger start-point of trace.													
01	Trigger End: Trigger end-point of trace.													
10	Trigger About: Trigger center-point of trace.													
11	Trigger Info: No action trigger, only for trace info.													
<i>FO</i>	1	<p>Fire Once. When set, this trigger will not re-fire until the <i>TR</i> bit is de-asserted. When de-asserted, this trigger will fire each time one of the trigger sources indicates trigger.</p>	R/W	0										

Table 11.52 TCBTRIGx Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Names	Bits			
<i>TR</i>	0	<p>Trigger happened. When set, this trigger fired since the <i>TR</i> bit was last written 0.</p> <p>This bit is used to inspect whether the trigger fired since this bit was last written zero.</p> <p>When set, all the trigger source bits (bit 4 to 13) will change their read value to indicate if the particular bit was the source to fire this trigger. Only enabled trigger sources can set the read value, but more than one is possible.</p> <p>Also when set the <i>Type</i> field and the <i>Trace</i> field will have read values which indicate if the trigger action was ever suppressed by a higher priority trigger.</p>	R/W0	0

11.10.12 Register Reset State

Reset state for all register fields is entered when either of the following occur:

1. TAP controller enters/is in Test-Logic-Reset state.
2. *EJ_TRST_N* input is asserted low.

11.11 Enabling MIPS Trace

As there are several ways to enable tracing, it can be quite confusing to figure out how to turn tracing on and off. This section should help clarify the enabling of trace.

11.11.1 Trace Trigger from EJTAG Hardware Instruction/Data Breakpoints

If hardware instruction/data simple breakpoints are implemented in the 74K core, then these breakpoint can be used as triggers to start/stop trace. When used for this, the breakpoints need not also generate a debug exception, but are capable of only generating an internal trigger to the trace logic. This is done by only setting the *TE* bit and not the *BE* bit in the *Breakpoint Control* register. Please see [Section 11.2.8.5 “Instruction Breakpoint Control n \(IBCn\) Register”](#) and [Section 11.2.9.5 “Data Breakpoint Control n \(DBCn\) Register”](#) for details on breakpoint control.

In connection with the breakpoints, the *Trace BreakPoint Control (TraceBPC)* register is used to define the trace action when a trigger happens. When a breakpoint is enabled as a trigger (*TE* = 1), it can be selected to be either a start or a stop trigger to the trace logic. Please see sections [Section 7.2.34 “TraceIBPC Register \(CP0 Register 23, Select 4\)”](#) and [Section 7.2.35 “TraceDBPC Register \(CP0 Register 23, Select 5\)”](#) for details on how to define a start/stop trigger.

The trace triggers are best used for fine grain tracing. First the mode in which the fine grain tracing is desired should be disabled in the *TraceControl*. For example if fine grain tracing is to be done in User Mode then *TraceControl_U* is set to zero. This disables general tracing while the program is in User Mode. Then a breakpoint start trigger turns on the trace at a particular instruction, and a breakpoint off trigger turns off the trace at the desired ending instruction. The breakpoint trigger can be caused by a breakpoint address match or a breakpoint data match. Both precise and imprecise data match is supported. The imprecise data match will generate the trace trigger after the expected start/end point, as the imprecise data match is dependent on system latencies.

11.11.2 Turning On PDtrace™ Trace

Trace enabling and disabling from software is similar to the hardware method, except that the bits in the control register are used instead of the input enable signals from the TCB. The *TraceControl_{TS}* bit controls whether hardware (via the TCB), or software (via the *TraceControl* register) controls tracing functionality.

Trace is turned on when the following expression evaluates true:

```
(
  (
    (TraceControlTS and TraceControlOn) or
    ((not TraceControlTS) and TCBCONTROLAOn)
  )
  and
  (MatchEnable or TriggerEnable)
)
```

where,

```
MatchEnable ←
(
  TraceControlTS
  and
  (
    TraceControlG or
    (((TraceControlASID xor EntryHiASID) and (not TraceControlASID_M)) = 0)
  )
  and
  (
    (TraceControlU and UserMode) or
    (TraceControlS and SupervisorMode) or
    (TraceControlK and KernelMode) or
    (TraceControlE and ExceptionMode) or
    (TraceControlD and DebugMode)
  )
)
or
(
  (not TraceControlTS)
  and
  (TCBCONTROLAG or (TCBCONTROLAASID = EntryHiASID))
  and
  (
    (TCBCONTROLAU and UserMode) or
    (TCBCONTROLAS and SupervisorMode) or
    (TCBCONTROLAK and KernelMode) or
    (TCBCONTROLAE and ExceptionMode) or
    (TCBCONTROLADM and DebugMode)
  )
)
```

and where,

```
TriggerEnable ←
(
  DBCiTE and
  DBSBS[i] and
```

```

    TraceBPCDE    and
    (TraceBPCDBPON[i] = 1)
)
or
(
    IBCiTE        and
    IBSBS[i]      and
    TraceBPCIE    and
    (TraceBPCIBPON[i] = 1)
)

```

As seen in the expression above, trace can be turned on only if the master switch *TraceControl_{On}* or *TCBCONTROLA_{On}* is first asserted.

Once this is asserted, there are two ways to turn on tracing. The first way, the *MatchEnable* expression, uses the input enable signals from the TCB or the bits in the *TraceControl* register. This tracing is done over general program areas. For example, all of the user-level code for a particular process (if ASID is specified), and so on.

The second way to turn on tracing, the *TriggerEnable* expression, is from the processor side using the EJTAG hardware breakpoint triggers. If EJTAG is implemented, and hardware breakpoints can be set, then using this method enables finer grain tracing control. It is possible to send a trigger signal that turns on tracing at a particular instruction. For example, it would be possible to trace a single procedure in a program by triggering on trace at the first instruction, and triggering off trace at the last instruction.

The easiest way to unconditionally turn on trace is to assert either hardware or software tracing and the corresponding trace on signal with other enables. For example, with *TraceControl_{TS}*=0, i.e., hardware controlled tracing, assert *TCBCONTROLA_{On}*, *TCBCONTROLA_G*, and all the other signals in the second part of expression *MatchEnable*. To only trace when a particular process with a particular ASID is executing, assert *TCBCONTROLA_{On}*, the correct *TCBCONTROLA_{ASID}* value, and the *TCBCONTROLA_U*, *TCBCONTROLA_K*, *TCBCONTROLA_E*, and *TCBCONTROLA_{DM}* registers. (If it is known that the particular process is a user-level process, then it would be sufficient to only assert *TCBCONTROLA_U* for example). When using the EJTAG hardware triggers to turn trace on and off, it is best if *TCBCONTROLA_{On}* is asserted with all the other processor mode selection bits in *TCBCONTROLA* are turned off. This would be the least confusing way to control tracing with the trigger signals. In a similar manner, tracing can be controlled via software with the *TraceControl* register.

11.11.3 Turning Off PDtrace™ Trace

Trace is turned off when the following expression evaluates true:

```

(
    (TraceControlTS and (not TraceControlOn)) or
    ((not TraceControlTS) and (not TCBCONTROLAOn))
)
or
(
    (not MatchEnable)    and
    (not TriggerEnable)  and
    TriggerDisable
)

```

where,

```

TriggerDisable ←
(
    DBCiTE        and

```

```
    DBSBS[i]      and
    TraceBPCDE    and
    (TraceBPCDBPON[i] = 0)
)
or
(
    IBCiTE      and
    IBSBS[i]      and
    TraceBPCIE    and
    (TraceBPCIBPON[i] = 0)
)
```

Tracing can be unconditionally turned off by de-asserting the *TraceControl_{On}* bit or the *TCBCONTROLA_{On}* signal. When either of these are asserted, tracing can be turned off if all of the enables are de-asserted, regardless of the *TraceControl_G* bit (*TCBCONTROLA_G*) and *TraceControl_{ASID}* (*TCBCONTROLA_{ASID}*) values. EJTAG hardware break-points can be used to trigger trace off as well.

Note that if simultaneous triggers are generated, and even one of them turns on tracing, then even if all of the others attempt to trigger trace off, tracing will still be turned on. This condition is reflected in presence of the “(not Trigger-Enable)” term in the expression above.

11.11.4 TCB Trace Enabling

The TCB must be enabled in order to produce a trace on the probe or to on-chip memory, when trace information is sent on the PDtrace interface. The main switch for this is the *TCBCONTROLB_{EN}* bit. When set, the TCB will send trace information to either on-chip trace memory or to the Trace Probe, as controlled by the setting of the *TCBCONTROLB_{OfC}* bit.

The TCB can optionally include trigger logic, which can control the *TCBCONTROLB_{EN}* bit. Please see [Section 11.12 “TCB Trigger Logic”](#) for details.

11.11.5 Tracing a Reset Exception

Tracing a reset exception is possible. However, the *TraceControl_{TS}* bit is reset to 0 at core reset, so all the trace control must be from the TCB (using *TCBCONTROLA* and *TCBCONTROLB*). The PDtrace FIFO and the entire TCB are reset based on an EJTAG reset. It is thus possible to set up the trace modes, etc., using the TAP controller, and then reset the processor core.

11.12 TCB Trigger Logic

The TCB is optionally implemented with a trigger unit, which is indicated by a non-zero value in the *TCBCONFIG_{TRIG}*. This subsection will explain some of the issues around triggers in the TCB.

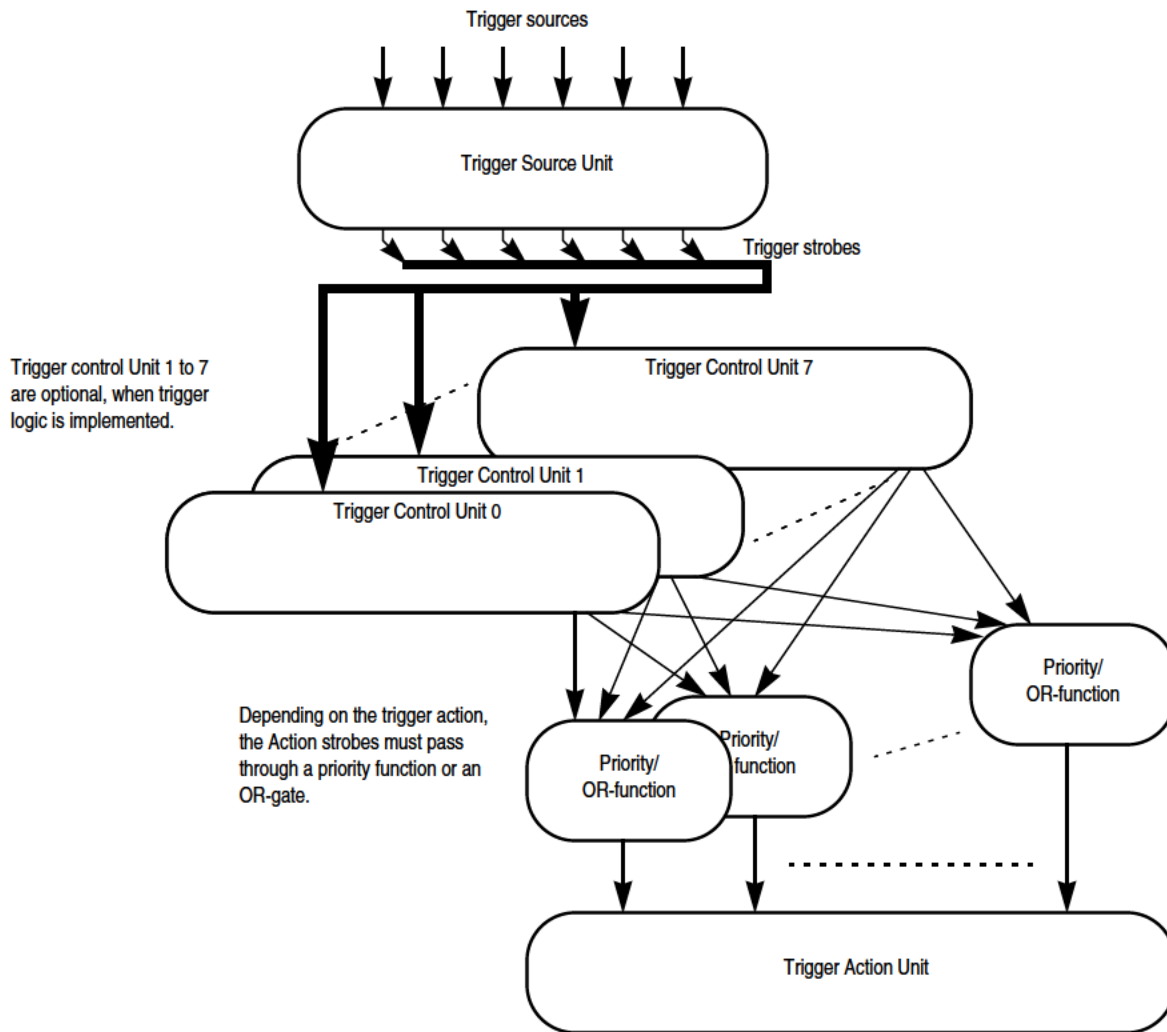
11.12.1 Trigger Units Overview

TCB trigger logic features three main parts:

1. Common Trigger Source detection unit
2. 1 to 8 separate Trigger Control units
3. Common Trigger Action unit

Figure 11.43 shows the functional overview of the trigger flow in the TCB.

Figure 11.43 TCB Trigger Processing Overview



11.12.2 Trigger Source Unit

The TCB has three trigger sources:

1. Chip-level trigger input (*TC_ChipTrigIn*)
2. Probe trigger input (*TR_TRIGIN*)
3. Debug Mode (DM) entry indication from the processor core

The input triggers are all rising-edge triggers, and the Trigger Source Units convert the edge into a single-cycle strobe to the Trigger Control Units.

11.12.3 Trigger Control Units

Up to eight Trigger Control Units are possible. Each of them has its own Trigger Control Register (*TCBTRIG_x*, $x=\{0..7\}$). Each of these registers controls the trigger fire mechanism for the unit. Each unit has all of the Trigger Sources as possible trigger events, and they can fire one or more of the Trigger Actions, as explained in the description of the Trigger Control register *TCBTRIG_x* (Section 11.10.11 “TCBTRIG_x Register (Reg 16-23)”).

11.12.4 Trigger Action Unit

The TCB has four possible trigger actions:

1. Chip-level trigger output (*TC_ChipTrigOut*)
2. Probe trigger output (*TR_TRIGOUT*)
3. Trace information. Put a programmable byte into the trace stream from the TCB.
4. Start, End, or About (delayed end) control of the *TCBCONTROLB_{EN}* bit

The basic function of the trigger actions is explained in Section 11.10.11 “TCBTRIG_x Register (Reg 16-23)”. See also Section 11.12.5 “Simultaneous Triggers”.

11.12.5 Simultaneous Triggers

Two or more triggers can fire simultaneously. The resulting behavior depends on the trigger action set for each, and whether or not they should produce a TF6 trace information output. There are two groups of trigger actions: Prioritized and OR'ed.

11.12.5.1 Prioritized Trigger Actions

For simultaneous prioritized trigger actions, the trigger control unit which has the lowest number takes precedence over the higher numbered units. The *x* in *TCBTRIG_x* registers defines the number. The oldest trigger takes precedence over all others.

The following trigger actions are prioritized when two or more units fire simultaneously:

- Trigger Start, End and About type triggers (*TCBTRIG_{xType}* field set to 00, 01 or 10), which will assert/de-assert the *TCBCONTROLB_{EN}* bit. The About trigger is delayed and will always change *TCBCONTROLB_{EN}* because it is the oldest trigger when it de-asserts *TCBCONTROLB_{EN}*. An About trigger will not start the countdown if an even older About trigger is using the Trace Word counter.
- Triggers which produce TF6 trace information in the trace flow (Trace bit is set).

Regardless of priority, the *TCBTRIG_{xTR}* bit is set when the trigger fires, even if a trigger action is suppressed by a higher priority trigger action. If the trigger is set to only fire once (the *TCBTRIG_{xFO}* bit is set), then the suppressed trigger action will not happen until after the *TCBTRIG_{xTR}* is cleared.

If a Trigger action is suppressed by a higher priority trigger, then the read value, when the *TCBTRIG_{xTR}* bit is set, for the *TCBTRIG_{xTrace}* field will be 0 for suppressed TF6 trace information actions. The read value in the *TCBTRIG_{xType}* field for suppressed Start/End/About triggers will be 11. This indication of a suppressed action is sticky. If any of the two actions (Trace and Type) are ever suppressed for a multi-fire trigger (the *TCBTRIG_{xFO}* bit is zero), then the read values in Trace and/or Type are set to indicate any suppressed action.

About Trigger

The About triggers delayed de-assertion of the *TCBCONTROLB_{EN}* bit is always executed, regardless of priority from another Start trigger at the time of the *TCBCONTROLB_{EN}* change. This means that if a simultaneous About trigger action on the *TCBCONTROLB_{EN}* bit ($n/2$ Trace Words after the trigger) and a Start trigger hit the same cycle, then the About trigger wins, regardless of which trigger number it is. The oldest trigger takes precedence.

However, if an About trigger has started the count down from $n/2$, but not yet reached zero, then a new About trigger will NOT be executed. Only one About trigger can have the cycle counter. This second About trigger will store 11 in the *TCBTRIGxType* field. But, if the *TCBTRIGxTrace* bit is set, a TF6 trace information will still go in the trace.

11.12.5.2 OR'ed Trigger Actions

The simple trigger actions *CHTro* and *PDTro* from each trigger unit, are OR'ed together to produce the final trigger. One or more expected trigger strobes on *TC_ChipTrigOut* can thus disappear. External logic should not rely on the counting of strobes to predict a specific event, unless simultaneous triggers are known not to occur.

11.13 MIPS Trace Cycle-by-Cycle Behavior

A key reason for using trace to debug a software problem, rather than using single stepping, is to get an accurate picture of real-time behavior. However, the trace logic itself can, when enabled, affect the exact cycle-by-cycle behavior.

11.13.1 FIFO Logic in PDtrace and TCB Modules

All the information that needs to be captured from the core pipeline is stored in a structure called the Unified FIFO (UFIFO). The Unified FIFO holds PC, load/store address values (delta or full), load/store data, processor mode changes (which are in the form of a TMOAS message), and Performance Counter data. The capture module reads out two UFIFO entries per cycle and sends them to the TCB where they are translated into two PDtrace-defined trace formats. These trace messages are then fed to the compression datapath which generates a PDtrace-defined trace word. To maintain throughput, the compression datapath is wide enough to sustain the generation of up to two trace words per cycle. The trace words are stored in a buffer called the TraceWord FIFO (TWFIFO). One trace word is read out from the TWFIFO and sent to the off-chip memory interface, or up to 2 trace words are read out per cycle and sent to the on-chip trace memory. The buffer will advance to the next trace word when a read acknowledge is received from the on-chip or off-chip memory interface.

In the TCB, the on-chip trace memory is defined as a 128-bit wide synchronous memory running at core-clock speed. In this case, the TWFIFO needs only four entries to guarantee it will not overflow. The TWFIFO could be filled in such a way that only 64 bits of the 128 bits is written to memory, which accounts for the four-entry requirement. For off-chip trace going through the Trace Probe, the FIFO is much more important because of the limited number of pins (4, 8, or 16) between the Probe and memory. Also, the speed of the Trace Probe interface can be different (either faster or slower) from that of the 74K core. So for off-chip tracing, a deeper TCB TWFIFO is desirable.

11.13.2 Handling of FIFO Overflow in the PDtrace Module

Depending on the amount of trace information selected for trace, and the frequency with which the 32-bit data interface is needed, it is possible for the PDtrace FIFO to overflow from time to time. There are two ways to handle this case:

1. Allow the overflow to happen, and thereby lose some information from the trace data.
2. Prevent the overflow by back-stalling the core until the FIFO has enough empty slots to accept new trace data.

The PDtrace FIFO option is controlled by either the *TraceControl_{IO}* or the *TCBCONTROL_{IO}* bit, depending on the setting of the *TraceControl_{TS}* bit.

The first option is free of any cycle-by-cycle change whether trace is turned on or not. This is achieved at the cost of potentially losing trace information. After an overflow, the FIFO is completely emptied, and the next instruction is traced as if it were the start of the trace (processor mode and full PC are traced). This guarantees that only the un-traced FIFO information is lost.

The second option guarantees that all the trace information is traced to the TCB. In some cases this is then achieved by back-stalling the core pipeline, giving the PDtrace FIFO time to empty enough room in the FIFO to accept new trace information from a new instruction. This option can obviously change the real-time behavior of the core when tracing is turned on. In 74K core, the UFIFO has 64 entries, while TWFIPO has 11 entries. The TWFIPO will almost always fill up first and be the cause of the back-stall to the core pipeline.

If PC trace information is the only thing enabled (in *TraceControl_{2MODE}* or *TCBCONTROL_{CMODE}*, depending on the setting of *TraceControl_{TS}*), and tracing of all branches is turned off (via *TraceControl_{TB}* or *TCBCONTROL_{ATB}*, depending on the setting of *TraceControl_{TS}*), then the FIFO is unlikely to overflow. Of course, this depends on the code executed and the frequency of exception handler jumps, but with this setting there is very little information overhead.

11.13.3 Handling of FIFO Overflow in the TCB

As mentioned earlier, the buffer in the TCB (TWFIPO) is used to buffer the TWs which are sent off-chip through the Trace Probe. The data width of the probe can be either 4, 8, or 16 pins, and the speed of these data pins can be from 16 times the core clock to 1/4 of the core clock (the trace probe clock always runs at a double data rate multiple of the core-clock). See [Section 11.13.3.1 “Probe Width and Clock-ratio Settings”](#) for a description of probe width and clock-ratio options. The combination of the probe width (4, 8, or 16) and the data speed, allows for data rates through the trace probe from 256 bits to only 1 bit per core-clock cycle. The high extreme is not likely to be supported in any implementation, but the low one might be.

The data rate is an important figure when the likelihood of a TCB TWFIPO overflow is considered. The TCB will at maximum produce one full 64-bit TW per core-clock cycle. This is true for any selection of trace mode in *TraceControl_{2MODE}* or *TCBCONTROL_{CMODE}*. The PDtrace module will guarantee the limited amount of data. If the TCB data rate cannot be matched by the off-chip probe width and data speed, then the TCB FIFO can possibly overflow. There are two options:

1. Allow the overflow to happen, and thereby lose some information from the trace data.
2. Prevent the overflow by asserting a stall-signal back to the core. This will in turn stall the core pipeline.

There is no way to guarantee that this back-stall from the TCB is never asserted, unless the effective data rate of the Trace Probe interface is at least 64-bits per core-clock cycle.

As a practical matter, the amount of data to the TCB can be minimized by only tracing PC information and excluding any cycle-accurate information. This is explained in [Section 11.13.2 “Handling of FIFO Overflow in the PDtrace Module”](#) and [Section 11.13.4 “Adding Cycle Accurate Information to the Trace”](#). With this setting, a data rate of 8-bits per core-clock cycle is usually sufficient. No guarantees can be given here, however, as heavy interrupt activity can increase the number of unpredictable jumps considerably.

11.13.3.1 Probe Width and Clock-ratio Settings

The actual number of data pins (4, 8, or 16) is defined by the *TCBCONFIG_{PW}* field. Furthermore, the frequency of the Trace Probe can be different from the core-clock frequency. The trace clock (*TR_CLK*) is a double data rate clock, which means that the data pins (*TR_DATA*) change their value on both edges of the trace clock. When the trace clock

is running at a clock ratio of 1:2 (one half) of the core clock, the data output registers are running at core-clock frequency. The clock ratio is set in the *TCBCONTROLB_{CR}* field. The legal range for the clock ratio is defined in *TCBCONFIG_{CRM_{Max}}* and *TCBCONFIG_{CRM_{Min}}* (both values inclusive). If *TCBCONTROLB_{CR}* is set to an unsupported value, the result is UNPREDICABLE. The maximum possible value for *TCBCONFIG_{CRM_{Max}}* is 8:1 (*TR_CLK* is running 8 times faster than the core-clock). The minimum possible value for *TCBCONFIG_{CRM_{Min}}* is 1:8 (*TR_CLK* is running at one eighth of the core-clock). See Table 11.43 for a description of the encoding of the clock ratio fields.

11.13.4 Adding Cycle Accurate Information to the Trace

Depending on the trace regeneration software, it is possible to obtain the exact cycle time relationship between each instruction in the trace. This information is added to the trace when the *TCBCONTROLB_{CA}* bit is set. The overhead on the trace information is a little more than one extra bit per core-clock cycle.

This setting only affects the TCB TWFIFO and not the PDtrace UFIFO. The extra bit therefore only affects the likelihood of the TCB TWFIFO overflowing.

11.14 TCB On-Chip Trace Memory

When on-chip trace memory is available (*TCBCONFIG_{OnT}* is set), the on-chip memory is typically smaller than external trace-probe memory. The assumption is that it is of some value to trace a smaller piece of the program.

With on-chip trace memory, the TCB can work in three possible modes:

1. Trace-From mode
2. Trace-To mode
3. Under Trigger unit control

Software can select this mode using the *TCBCONTROLB_{TM}* field. If one or more trigger control registers (*TCBTRIG_X*) are implemented, and they are using Start, End or About triggers, then the trace mode in *TCBCONTROLB_{TM}* should be set to Trace-To mode.

11.14.1 On-Chip Trace Memory Size

The supported on-chip trace memory size can range from 256 bytes to 8 Mbytes, in powers of 2. The actual size is shown in the *TCBCONFIG_{SZ}* field.

11.14.2 Trace-From Mode

In the Trace-From mode, tracing begins when the processor enters into a processor mode/ASID value which is defined to be traced or when an EJTAG hardware breakpoint trace trigger turns on tracing. Trace collection is stopped when the buffer is full. The TCB then signals buffer full using *TCBCONTROLB_{BF}*. When external software polling this register finds the *TCBCONTROLB_{BF}* bit set, it can then read out the internal trace memory. Saving the trace into the internal buffer will re-commence again only when the *TCBCONTROLB_{BF}* bit is reset and if the core is sending valid trace data.

11.14.3 Trace-To Mode

In the Trace-To mode, the TCB continues writing to the internal trace memory, overwriting the oldest information, until the processor reaches an end of trace condition. End of trace is reached by leaving the processor mode/ASID value which is traced, or when an EJTAG hardware breakpoint trace trigger turns tracing off. At this point, the on-chip trace buffer is dumped as described in [Section 11.14.2 “Trace-From Mode”](#).

Instruction Set Overview

This chapter provides an overview of the 74K™ core instruction set, including the instruction formats and the basic instruction types.

This chapter discusses the following topics:

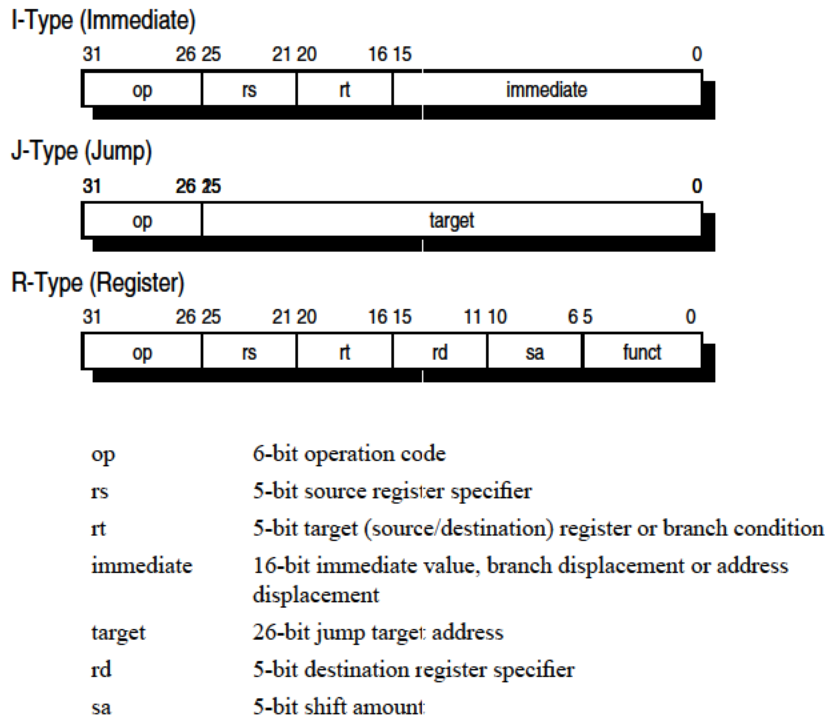
- Section 12.1 “CPU Instruction Formats”
- Section 12.2 “Load and Store Instructions”
- Section 12.3 “Computational Instructions”
- Section 12.4 “Jump and Branch Instructions”
- Section 12.5 “Control Instructions”
- Section 12.6 “Coprocesor Instructions”

Refer to Chapter 13, “74K™ Processor Core Instructions” for a complete listing and description of those instructions whose behavior differs in the 74K processor core. The complete MIPS32 instruction set is described in Volume II of the MIPS32® Architecture For Programmers.

12.1 CPU Instruction Formats

A CPU instruction consists of a single 32-bit word, aligned on a word boundary. There are three instruction formats: *immediate* (*I*-type), *jump* (*J*-type), and *register* (*R*-type). The use of a small number of instruction formats simplifies instruction decoding, allowing the compiler to synthesize more complicated (and less frequently used) operations and addressing modes from these three formats as needed. The instruction formats are shown in [Figure 12.1](#).

Figure 12.1 Instruction Formats



12.2 Load and Store Instructions

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

12.2.1 Scheduling a Load Delay Slot

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

In a 74K core, the instruction immediately following a load instruction can use the contents of the loaded register; however in such cases hardware interlocks insert additional real cycles. Although not required, the scheduling of load delay slots can be desirable, both for performance and R-Series processor compatibility.

12.2.2 Defining Access Types

Access type indicates the size of a core data item to be loaded or stored, set by the load or store instruction opcode.

Regardless of access type or byte ordering (endianness), the address given specifies the low-order byte in the addressed field. For a big-endian configuration, the low-order byte is the most-significant byte; for a little-endian configuration, the low-order byte is the least-significant byte.

The access type, together with the three low-order bits of the address, defines the bytes accessed within the addressed word, as shown in Table 12.1. Only the combinations shown in Table 12.1 are permissible; other combinations cause address-error exceptions.

Instruction fetches are either halfword accesses (MIPS16e™ code) or word accesses (32b code). These references will be impacted by endianness in the same way as load/store references of those sizes.

Table 12.1 Byte Access Within a Doubleword

Access Type	Low-Order Address Bits			Bytes Accessed															
				Big Endian (63-----31-----0)								Little Endian (63-----31-----0)							
	2	1	0	Byte								Byte							
Doubleword	0	0	0	0	1	2	3	4	5	6	7	7	6	5	4	3	2	1	0
Word	0	0	0	0	1	2	3									3	2	1	0
	1	0	0					4	5	6	7	7	6	5	4				
Triplebyte	0	0	0	0	1	2											2	1	0
	0	0	1		1	2	3									3	2	1	
	1	0	0					4	5	6			6	5	4				
	1	0	1						5	6	7	7	6	5					
Halfword	0	0	0	0	1													1	0
	0	1	0			2	3									3	2		
	1	0	0					4	5				5	4					
	1	1	0							6	7	7	6						
Byte	0	0	0	0															0
	0	0	1		1													1	
	0	1	0			2											2		
	0	1	1				3								3				
	1	0	0					4						4					
	1	0	1						5						5				
	1	1	0							6			6						
	1	1	1								7	7							

12.3 Computational Instructions

Computational instructions can be either in register (R-type) format, in which both operands are registers, or in immediate (I-type) format, in which one operand is a 16-bit immediate.

Instruction Set Overview

Computational instructions perform the following operations on register values:

- Arithmetic
- Logical
- Shift
- Count Leading Zeros/Ones
- Multiply
- Divide

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

- ALU Immediate instructions
- Three-operand Register-type Instructions
- Shift Instructions
- Multiply And Divide Instructions

12.3.1 Cycle Timing for Multiply and Divide Instructions

Multiply instruction in the integer pipeline are transferred to the multiplier while remaining instructions continue through the pipeline; the product of the multiply instruction is saved in the HI and LO registers. If the multiply instruction is followed by an MFHI or MFLO before the product is available, the pipeline interlocks until this product does become available. Refer to [Chapter 2, “Pipeline of the 74K™ Core”](#) for more information on instruction latency and repeat rates.

12.4 Jump and Branch Instructions

Jump and branch instructions change the control flow of a program. All jump and branch instructions occur with a delay of one instruction: that is, the instruction immediately following the jump or branch (the instruction in the so-called *delay slot*) always executes while the target instruction is being fetched from storage.

12.4.1 Overview of Jump Instructions

Subroutine calls in high-level languages are usually implemented with Jump or Jump and Link instructions, both of which are J-type instructions. In J-type format, the 26-bit target address shifts left 2 bits and combines with the high-order 4 bits of the current program counter to form an absolute address.

Returns, dispatches, and large cross-page jumps are usually implemented with the Jump Register or Jump and Link Register instructions. Both are R-type instructions that use the 32-bit byte address contained in one of the general purpose registers.

For more information about jump instructions, refer to the individual instructions in *MIPS32® Architecture Reference Manual, Volume II: The MIPS32® Instruction Set*.

12.4.2 Overview of Branch Instructions

All branch instruction target addresses are computed by adding the address of the instruction in the delay slot to the 16-bit *offset* (shifted left 2 bits and sign-extended to 32 bits). All branches occur with a delay of one instruction.

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

Branches, jumps, ERET, and DERET instructions should not be placed in the delay slot of a branch or jump.

12.5 Control Instructions

Control instructions allow the software to initiate traps; they are always R-type.

12.6 Coprocessor Instructions

CP0 instructions perform operations on the System Control Coprocessor registers to manipulate the memory management and exception handling facilities of the processor. Refer to [Chapter 13, “74K™ Processor Core Instructions”](#) for a listing of CP0 instructions.

74K™ Processor Core Instructions

This chapter supplements the *MIPS32® Architecture Reference Manual* by describing instruction behavior that is specific to the 74K processor core. The chapter contains the following sections:

- Section 13.1 “Understanding the Instruction Descriptions”
- Section 13.2 “74K™ Opcode Map”
- Section 13.3 “Floating Point Unit Instruction Format Encodings”
- Section 13.4 “MIPS32™ Instruction Set for the 74K™ Core”

The 74K processor core also supports the instructions in the MIPS DSP ASE Revision 2 and the MIPS16e ASE. The MIPS DSP ASE Revision 2 instruction set is described in Chapter 4, “The MIPS® DSP Application-Specific Extension to the MIPS32® Instruction Set”. The MIPS16e ASE instruction set is described in Chapter 14, “MIPS16e™ Application-Specific Extension to the MIPS32® Instruction Set”.

13.1 Understanding the Instruction Descriptions

Refer to *Volume II* of the *MIPS32® Architecture Reference Manual* for more information about the instruction descriptions. That document contains a description of the instruction fields, a definition of terms, and a description function notation.

13.2 74K™ Opcode Map

Table 13.1 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
*	Operation or field codes marked with this symbol are reserved for future use, are valid encodings for a higher-order MIPS ISA level, or are part of an application specific extension not implemented on this core. Executing such an instruction will cause a Reserved Instruction Exception.
δ	(Also <i>italic</i> field name.) Operation or field codes marked with this symbol are a field class. The instruction word must be further decoded by examining additional tables that show values for another instruction field.
∇	Operation or field codes marked with this symbol represent instructions which are only legal if 64-bit floating point operations are enabled. In other cases, executing such an instruction will 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).
ϕ	Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS32 ISA. Software should avoid using these operation or field codes.

Table 13.2 MIPS32 Encoding of the Opcode Field

opcode		<i>bits 28..26</i>							
		0	1	2	3	4	5	6	7
<i>bits 31..29</i>		000	001	010	011	100	101	110	111
0	000	<i>SPECIAL</i> δ	<i>REGIMM</i> δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	<i>COP0</i> δ	<i>COP1</i> δ	<i>COP2</i> δ	<i>COP1X</i>	BEQL ϕ	BNEL ϕ	BLEZL ϕ	BGTZL ϕ
3	011	*	*	*	*	<i>SPECIAL2</i> δ	JALX	*	<i>SPECIAL3</i> δ
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	*
5	101	SB	SH	SWL	SW	*	*	SWR	CACHE
6	110	LL	LWC1	LWC2	PREF	*	LDC1	LDC2	*
7	111	SC	SWC1	SWC2	*	*	SDC1	SDC2	*

Table 13.3 MIPS32 SPECIAL Opcode Encoding of Function Field

function		<i>bits 2..0</i>							
		0	1	2	3	4	5	6	7
<i>bits 5..3</i>		000	001	010	011	100	101	110	111
0	000	SLL ¹	<i>MOVCI</i> δ	<i>SRL</i> δ	SRA	SLLV	*	<i>SRLV</i> δ	SRAV
1	001	JR ²	JALR ²	MOVZ	MOVN	SYSCALL	BREAK	*	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	*	*	*	*
3	011	MULT	MULTU	DIV	DIVU	*	*	*	*
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	*	*	SLT	SLTU	*	*	*	*
6	110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	111	*	*	*	*	*	*	*	*

1. Specific encodings of the *rt*, *rd*, and *sa* fields are used to distinguish among the SLL, NOP, SSNOP, and EHB functions.
2. Specific encodings of the hint field are used to distinguish JR from JR.HB and JALR from JALR.HB

Table 13.4 MIPS32 REGIMM Encoding of *rt* Field

rt		<i>bits 18..16</i>							
		0	1	2	3	4	5	6	7
<i>bits 20..19</i>		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	*	*	*	*	*	*	*	SYNCI

Table 13.5 MIPS32 SPECIAL2 Encoding of Function Field

function		<i>bits 2..0</i>							
<i>bits 5..3</i>		0	1	2	3	4	5	6	7
<i>bits 5..3</i>		000	001	010	011	100	101	110	111
0	000	MADD	MADDU	MUL	*	MSUB	MSUBU	*	*
1	001	*	*	*	*	*	*	*	*
2	010	CorExtend							
3	011								
4	100	CLZ	CLO	*	*	*	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	SDBBP

Table 13.6 MIPS32 Special3 Encoding of Function Field for Release 2 of the Architecture

function		<i>bits 2..0</i>							
<i>bits 5..3</i>		0	1	2	3	4	5	6	7
<i>bits 5..3</i>		000	001	010	011	100	101	110	111
0	000	EXT	*	*	*	INS	*	*	*
1	001	*	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	<i>BSHFL</i> δ	*	*	*	*	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	RDHWR	*	*	*	*

Table 13.7 MIPS32 MOVCI Encoding of tf Bit

tf	<i>bit 16</i>	
	0	1
	MOVF	MOVT

Table 13.8 MIPS32 SRL Encoding of Shift/Rotate

tf	<i>bit 21</i>	
	0	1
	SRL	ROTR

Table 13.9 MIPS32 SRLV Encoding of Shift/Rotate

tf	<i>bit 6</i>	
	0	1
	SRLV	ROTRV

Table 13.10 MIPS32 BSHFLEncoding of sa Field¹

sa		bits 8..6							
		0	1	2	3	4	5	6	7
bits 10..9		000	001	010	011	100	101	110	111
0	00			WSBH					
1	01								
2	10	SEB							
3	11	SEH							

1. The sa field is sparsely decoded to identify the final instructions. Entries in this table with no mnemonic are reserved for future use by MIPS Technologies and may or may not cause a Reserved Instruction exception.

Table 13.11 MIPS32 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	*	*	*	MTC0	*	*	*
1	01	*	*	RDPGPR	MFMC0 ¹ δ	*	*	WRPGPR	*
2	10	C0 δ							
3	11	C0 δ							

1. Release 2 of the Architecture added the MFMC0 function, which is further decoded as the DI and EI instructions.

Table 13.12 MIPS32COP0 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
4	100	WAIT	*	*	*	*	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

Table 13.13 MIPS32 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	*	CFC1	MFHC1	MTC1	*	CTC1	MTHC1
1	01	BC1 δ	*	*	*	*	*	*	*
2	10	S δ	D δ	*	*	W δ	L δ	*	*
3	11	*	*	*	*	*	*	*	*

Table 13.14 MIPS32 COP1 Encoding of Function Field When rs=S

function		bits 2..0							
bits 5..3		0	1	2	3	4	5	6	7
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	*	*	*	*	*	*	*	*
4	100	*	CVT.D	*	*	CVT.W	CVT.L ∇	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

Table 13.15 MIPS32 COP1 Encoding of Function Field When rs=D

function		bits 2..0							
bits 5..3		0	1	2	3	4	5	6	7
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	*	*	*	*	*	*	*	*
4	100	CVT.S	*	*	*	CVT.W	CVT.L ∇	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

Table 13.16 MIPS32 COP1 Encoding of Function Field When rs=W or L¹

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

1. Format type L is legal only if 64-bit floating point operations are enabled.

Table 13.17 MIPS32 COP1 Encoding of tf Bit When rs=S or D, Function=MOVCF

tf	bit 16	
	0	1
	MOV.fmt	MOVT.fmt

Table 13.18 COP1X Encoding of Function Field¹

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	*	*	*	*	*	*	*	*
4	100	MADD.S ▽	MADD.D ▽	*	*	*	*	*	*
5	101	MSUB.S ▽	MSUB.D ▽	*	*	*	*	*	*
6	110	NMADD.S ▽	NMADD.D ▽	*	*	*	*	*	*
7	111	NMSUB.S ▽	NMSUB.D ▽	*	*	*	*	*	*

1. COP1X instructions are legal only if 64-bit floating point operations are enabled.

Table 13.19 MIPS32 COP2 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	MFC2	*	CFC2	MFHC2	MTC2	*	CTC2	MTHC2
1	01	BC2δ	*	*	*	*	*	*	*
2	10	C2							
3	11								

13.3 Floating Point Unit Instruction Format Encodings

Instruction format encodings for the floating point unit are presented in this section. This information is a tabular presentation of the encodings described in tables Table 13.13 and Table 13.18 above.

Table 13.20 Floating Point Unit Instruction Format Encodings

fmt field (bits 25..21 of COP1 opcode)		fmt3 field (bits 2..0 of COP1X opcode)		Mnemonic	Name	Bit Width	Data Type
Decimal	Hex	Decimal	Hex				
0..15	00..0F	—	—	Used to encode Coprocessor 1 interface instructions (MFC1, CTC1, etc.). Not used for format encoding.			
16	10	0	0	S	Single	32	Floating Point
17	11	1	1	D	Double	64	Floating Point
0..15	00..0F	—	—	Used to encode Coprocessor 1 interface instructions (MFC1, CTC1, etc.). Not used for format encoding.			
20	14	4	4	W	Word	32	Fixed Point
21	15	5	5	L	Long	64	Fixed Point
22	16	6	6	PS	Paired Single	2 × 32	Floating Point

Table 13.20 Floating Point Unit Instruction Format Encodings (Continued)

<i>fmt</i> field (bits 25..21 of COP1 opcode)		<i>fmt3</i> field (bits 2..0 of COP1X opcode)		Mnemonic	Name	Bit Width	Data Type
Decimal	Hex	Decimal	Hex				
23	17	7	7	Reserved for future use by the architecture.			
24..31	18..1F	—	—	Reserved for future use by the architecture. Not available for <i>fmt3</i> encoding.			

13.4 MIPS32™ Instruction Set for the 74K™ Core

This section describes the MIPS32 instructions for the 74K cores. Table 13.21 lists the instructions in alphabetical order. Following the table, the instructions that have implementation-dependent behavior in the 74K core are described individually. The descriptions of other instructions that exist in the *MIPS32® Architecture Reference Manual* are not duplicated here.

Table 13.21 74K™ Core Instruction Set

Instruction	Description	Function
ADD	Integer Add	$Rd = Rs + Rt$
ADDI	Integer Add Immediate	$Rt = Rs + \text{Immed}$
ADDIU	Unsigned Integer Add Immediate	$Rt = Rs +_U \text{Immed}$
ADDIU_PC	Unsigned Integer Add Immediate to PC (MIPS16 only)	$Rt = PC +_U \text{Immed}$
ADDU	Unsigned Integer Add	$Rd = Rs +_U Rt$
AND	Logical AND	$Rd = Rs \& Rt$
ANDI	Logical AND Immediate	$Rt = Rs \& (0_{16} \ \ \text{Immed})$
B	Unconditional Branch (Assembler idiom for: BEQ r0, r0, offset)	$PC += (\text{int})\text{offset}$
BAL	Branch and Link (Assembler idiom for: BGEZAL r0, offset)	$GPR[31] = PC + 8$ $PC += (\text{int})\text{offset}$
BC2F	Branch On Cp2 False	if (cc[i] == 0) then $PC += (\text{int})\text{offset}$
BC2FL	Branch On Cp2 False Likely	if (cc[i] == 0) then $PC += (\text{int})\text{offset}$ else Ignore Next Instruction
BC2T	Branch On Cp2 True	if (cc[i] == 1) then $PC += (\text{int})\text{offset}$
BC2TL	Branch On Cp2 True Likely	if (cc[i] == 1) then $PC += (\text{int})\text{offset}$ else Ignore Next Instruction

Table 13.21 74K™ Core Instruction Set (Continued)

Instruction	Description	Function
BEQ	Branch On Equal	if Rs == Rt PC += (int)offset
BEQL	Branch On Equal Likely	if Rs == Rt PC += (int)offset else Ignore Next Instruction
BGEZ	Branch on Greater Than or Equal To Zero	if !Rs[31] PC += (int)offset
BGEZAL	Branch on Greater Than or Equal To Zero And Link	GPR[31] = PC + 8 if !Rs[31] PC += (int)offset
BGEZALL	Branch on Greater Than or Equal To Zero And Link Likely	GPR[31] = PC + 8 if !Rs[31] PC += (int)offset else Ignore Next Instruction
BGEZL	Branch on Greater Than or Equal To Zero Likely	if !Rs[31] PC += (int)offset else Ignore Next Instruction
BGTZ	Branch on Greater Than Zero	if !Rs[31] && Rs != 0 PC += (int)offset
BGTZL	Branch on Greater Than Zero Likely	if !Rs[31] && Rs != 0 PC += (int)offset else Ignore Next Instruction
BLEZ	Branch on Less Than or Equal to Zero	if Rs[31] Rs == 0 PC += (int)offset
BLEZL	Branch on Less Than or Equal to Zero Likely	if Rs[31] Rs == 0 PC += (int)offset else Ignore Next Instruction
BLTZ	Branch on Less Than Zero	if Rs[31] PC += (int)offset
BLTZAL	Branch on Less Than Zero And Link	GPR[31] = PC + 8 if Rs[31] PC += (int)offset
BLTZALL	Branch on Less Than Zero And Link Likely	GPR[31] = PC + 8 if Rs[31] PC += (int)offset else Ignore Next Instruction
BLTZL	Branch on Less Than Zero Likely	if Rs[31] PC += (int)offset else Ignore Next Instruction
BNE	Branch on Not Equal	if Rs != Rt PC += (int)offset

Table 13.21 74K™ Core Instruction Set (Continued)

Instruction	Description	Function
BNEL	Branch on Not Equal Likely	if Rs != Rt PC += (int)offset else Ignore Next Instruction
BREAK	Breakpoint	Break Exception
CACHE	Cache Operation	See the description of the CACHE instruction on page 347 .
CFC2	Move Control Word From Cp2	Rt = CP2_Control[Fs]
CLO	Count Leading Ones	Rd = NumLeadingOnes(Rs)
CLZ	Count Leading Zeroes	Rd = NumLeadingZeroes(Rs)
COP0	Coprocessor 0 Operation	See Software User's Manual
COP2	Coprocessor 2 Operation	Implementation-dependent
CTC2	Move Control Word to Cp2	Cp2_Control[Fs] = Rt
DERET	Return from Debug Exception	PC = DEPC Exit Debug Mode
DI	Atomically Disable Interrupts	Rt = Status; Status _{IE} = 0
DIV	Divide	LO = (int)Rs / (int)Rt HI = (int)Rs % (int)Rt
DIVU	Unsigned Divide	LO = (uns)Rs / (uns)Rt HI = (uns)Rs % (uns)Rt
EHB	Execution Hazard Barrier	Stop instruction execution until execution hazards are cleared
EI	Atomically Enable Interrupts	Rt = Status; Status _{IE} = 1
ERET	Return from Exception	if SR[2] PC = ErrorEPC else PC = EPC SR[1] = 0 SR[2] = 0 LL = 0
EXT	Extract Bit Field	Rt = ExtractField(Rs, pos, size)
INS	Insert Bit Field	Rt = InsertField(Rs, Rt, pos, size)
J	Unconditional Jump	PC = PC[31:28] offset<<2
JAL	Jump and Link	GPR[31] = PC + 8 PC = PC[31:28] offset<<2
JALR	Jump and Link Register	Rd = PC + 8 PC = Rs

Table 13.21 74K™ Core Instruction Set (Continued)

Instruction	Description	Function
JALR.HB	Jump and Link Register with Hazard Barrier	Like JALR, but also clears execution and instruction hazards
JALRC	Jump and Link Register Compact - do not execute instruction in jump delay slot(MIPS16 only)	Rd = PC + 2 PC = Rs
JR	Jump Register	PC = Rs
JR.HB	Jump Register with Hazard Barrier	Like JR, but also clears execution and instruction hazards
JRC	Jump Register Compact - do not execute instruction in jump delay slot (MIPS16 only)	PC = Rs
LB	Load Byte	Rt = (byte)Mem[base+offset]
LBU	Unsigned Load Byte	Rt = (ubyte)Mem[base+offset]
LDC2	Load Doubleword to Cp2	Ft = memory[base+offset]
LH	Load Halfword	Rt = (half)Mem[base+offset]
LHU	Unsigned Load Halfword	Rt = (uhalf)Mem[base+offset]
LL	Load Linked Word	Rt = Mem[base+offset] LL = 1 See also the description of the LL instruction on page 355 .
LUI	Load Upper Immediate	Rt = immediate << 16
LW	Load Word	Rt = Mem[Rs+offset]
LWC2	Load Word to Cp2	Ft = memory[base+offset]
LWPC	Load Word, PC relative	Rt = Mem[PC+offset]
LWL	Load Word Left	See Architecture Reference Manual
LWR	Load Word Right	See Architecture Reference Manual
MADD	Multiply-Add	HI LO += (int)Rs * (int)Rt
MADDU	Multiply-Add Unsigned	HI LO += (uns)Rs * (uns)Rt
MFC0	Move From Coprocessor 0	Rt = CPR[0, Rd, sel]
MFC2	Move From Cp2 Register	Rt = FS _{31..0}
MFHC2	Move From High Half of Cp2 Register	Rt = FS _{63..32}
MFHI	Move From HI	Rd = HI
MFLO	Move From LO	Rd = LO
MOVN	GPR Conditional Move on Not Zero	if Rt ≠ 0 then Rd = Rs
MOVZ	GPR Conditional Move on Zero	if Rt = 0 then Rd = Rs

Table 13.21 74K™ Core Instruction Set (Continued)

Instruction	Description	Function
MSUB	Multiply-Subtract	HI LO $-=$ (int)Rs * (int)Rt
MSUBU	Multiply-Subtract Unsigned	HI LO $-=$ (uns)Rs * (uns)Rt
MTC0	Move To Coprocessor 0	CPR[0, n, Sel] = Rt
MTC2	Move to Cp2 register	Fs = Rt
MTHC2	Move to High Half of Cp2 register	Fd = Rt Fs _{31..0}
MTHI	Move To HI	HI = Rs
MTLO	Move To LO	LO = Rs
MUL	Multiply with register write	HI LO =Unpredictable Rd = ((int)Rs * (int)Rt) _{31..0}
MULT	Integer Multiply	HI LO = (int)Rs * (int)Rd
MULTU	Unsigned Multiply	HI LO = (uns)Rs * (uns)Rd
NOP	No Operation (Assembler idiom for: SLL r0, r0, r0)	
NOR	Logical NOR	Rd = ~(Rs Rt)
OR	Logical OR	Rd = Rs Rt
ORI	Logical OR Immediate	Rt = Rs Immed
PREF	Prefetch	Load Specified Line into Cache. See also the description of the PREF instruction on page 357.
RDHWR	Read Hardware Register	Allows unprivileged access to registers enabled by HWREna register
RDPGPR	Read GPR from Previous Shadow Set	Rt = SGPR[SRSCtl _{PSS} , Rd]
RESTORE	Restore registers and deallocate stack frame (MIPS16 only)	See Architecture Reference Manual
ROTR	Rotate Word Right	Rd = Rt _{sa-1..0} Rt _{31..sa}
ROTRV	Rotate Word Right Variable	Rd = Rt _{Rs-1..0} Rt _{31..Rs}
SAVE	Save registers and allocate stack frame (MIPS16 only)	See Architecture Reference Manual
SB	Store Byte	(byte)Mem[base+offset] = Rt
SC	Store Conditional Word	if LL = 1 mem[base+offset] = Rt Rt = LL See also the description of the SC instruction on page 361.
SDBBP	Software Debug Break Point	Trap to SW Debug Handler
SDC2	Store Doubleword from Cp2	memory[base+offset] = Ft

Table 13.21 74K™ Core Instruction Set (Continued)

Instruction	Description	Function
SEB	Sign Extend Byte	$Rd = (\text{byte})Rs$
SEH	Sign Extend Half	$Rd = (\text{half})Rs$
SH	Store Half	$(\text{half})\text{Mem}[\text{base}+\text{offset}] = Rt$
SLL	Shift Left Logical	$Rd = Rt \ll sa$
SLLV	Shift Left Logical Variable	$Rd = Rt \ll Rs[4:0]$
SLT	Set on Less Than	if $(\text{int})Rs < (\text{int})Rt$ $Rd = 1$ else $Rd = 0$
SLTI	Set on Less Than Immediate	if $(\text{int})Rs < (\text{int})\text{Immed}$ $Rt = 1$ else $Rt = 0$
SLTIU	Set on Less Than Immediate Unsigned	if $(\text{uns})Rs < (\text{uns})\text{Immed}$ $Rt = 1$ else $Rt = 0$
SLTU	Set on Less Than Unsigned	if $(\text{uns})Rs < (\text{uns})\text{Immed}$ $Rd = 1$ else $Rd = 0$
SRA	Shift Right Arithmetic	$Rd = (\text{int})Rt \gg sa$
SRAV	Shift Right Arithmetic Variable	$Rd = (\text{int})Rt \gg Rs[4:0]$
SRL	Shift Right Logical	$Rd = (\text{uns})Rt \gg sa$
SRLV	Shift Right Logical Variable	$Rd = (\text{uns})Rt \gg Rs[4:0]$
SSNOP	Superscalar Inhibit No Operation	NOP
SUB	Integer Subtract	$Rt = (\text{int})Rs - (\text{int})Rd$
SUBU	Unsigned Subtract	$Rt = (\text{uns})Rs - (\text{uns})Rd$
SW	Store Word	$\text{Mem}[\text{base}+\text{offset}] = Rt$
SWC2	Store Word From Cp2 Register	$\text{Mem}[\text{base}+\text{offset}] = Fs$
SWL	Store Word Left	See Architecture Reference Manual
SWR	Store Word Right	See Architecture Reference Manual
SYNC	Synchronize	See the description of the SYNC instruction on page 363 .
SYNCHI	Synchronize Caches to Make Instruction Writes Effective	For D-cache writeback and I-cache invalidate on specified address
SYSCALL	System Call	SystemCallException

Table 13.21 74K™ Core Instruction Set (Continued)

Instruction	Description	Function
TEQ	Trap if Equal	if Rs == Rt TrapException
TEQI	Trap if Equal Immediate	if Rs == (int)Immed TrapException
TGE	Trap if Greater Than or Equal	if (int)Rs >= (int)Rt TrapException
TGEI	Trap if Greater Than or Equal Immediate	if (int)Rs >= (int)Immed TrapException
TGEIU	Trap if Greater Than or Equal Immediate Unsigned	if (uns)Rs >= (uns)Immed TrapException
TGEU	Trap if Greater Than or Equal Unsigned	if (uns)Rs >= (uns)Rt TrapException
TLBWI	Write Indexed TLB Entry	See the description of the TLBWI instruction on page 369 .
TLBWR	Write Random TLB Entry	See the description of the TLBWR instruction on page 373 .
TLBP	Probe TLB for Matching Entry	See Software Users Manual
TLBR	Read Index for TLB Entry	See the description of the TLBR instruction on page 367 .
TLT	Trap if Less Than	if (int)Rs < (int)Rt TrapException
TLTI	Trap if Less Than Immediate	if (int)Rs < (int)Immed TrapException
TLTIU	Trap if Less Than Immediate Unsigned	if (uns)Rs < (uns)Immed TrapException
TLTU	Trap if Less Than Unsigned	if (uns)Rs < (uns)Rt TrapException
TNE	Trap if Not Equal	if Rs != Rt TrapException
TNEI	Trap if Not Equal Immediate	if Rs != (int)Immed TrapException
WAIT	Wait for Interrupts	Stall until interrupt occurs. See the description of the WAIT instruction on page 371 .
WRPGPR	Write to GPR in Previous Shadow Set	SGPR[SRSCtl _{PSS} , Rd] = Rt
WSBH	Word Swap Bytes Within HalfWords	Rd = Rt _{23..16} Rt _{31..24} Rt _{7..0} Rt _{15..8}
XOR	Exclusive OR	Rd = Rs ^ Rt
XORI	Exclusive OR Immediate	Rt = Rs ^ (uns)Immed
ZEB	Zero extend byte (MIPS16 only)	Rt = (ubyte) Rs

Table 13.21 74K™ Core Instruction Set (Continued)

Instruction	Description	Function
ZEH	Zero extend half (MIPS16 only)	$Rt = (\text{uhalf}) Rs$

tag operation, as these operations are used for initialization and diagnostic purposes.

An address Error Exception (with cause code equal AdEL) occurs if the effective address references a portion of the kernel address space which would normally result in such an exception.

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

Table 13.23 Encoding of Bits[17:16] of CACHE Instruction

Code	Name	Cache	Cop0 Registers Used
2#00	I	Primary Instruction	<i>I</i> TagLo, <i>I</i> TagHi, <i>I</i> DataLo, <i>I</i> DataHi, <i>ErrCtl</i>
2#01	D	Primary Data	<i>D</i> TagLo, <i>D</i> TagHi, <i>D</i> DataLo, <i>ErrCtl</i>
2#10	T	Tertiary - Not supported	
2#11	S	Secondary	<i>L2</i> TagLo

Some of the operations use coprocessor0 registers as either sources or destinations. Each of the caches has a separate set of Tag and Data registers. The last column in [Table 13.23](#) lists which registers are used by operations to each cache.

Bits [20:18] of the instruction specify the operation to perform. On Index Load Tag and Index Store Data operations, the specific word (primary D) or double-word (primary I, secondary) that is addressed is loaded into or read from the *D*DataLo (primary D), *L23*DataLo, and *L23*DataHi (secondary), or *I*DataLo and *I*DataHi (primary I) registers. All other cache instructions are line-based, and the word and byte indexes will not affect their operation

Table 13.24 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST, DYT, SPR] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation	
2#000	I	Index Invalidate	Index	Set the state of the cache line at the specified index to invalid. This encoding may be used by software to invalidate the entire instruction cache by stepping through all valid indices.	Yes
	D, S	Index Writeback Invalidate	Index	If the state of the cache line at the specified index is valid and dirty, write the line back to the memory address specified by the cache tag. After that operation is completed, set the state of the cache line to invalid. If the line is valid but not dirty, set the state of the line to invalid. This encoding may be used by software to invalidate the entire data cache by stepping through all valid indices. Note that Index Store Tag should be used to initialize the cache at powerup.	Yes

Table 13.24 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST, DYT, SPR] Cleared (Continued)

Code	Caches	Name	Effective Address Operand Type	Operation	
2#001	I	Index Load Tag	Index	<ul style="list-style-type: none"> Read the tag for the cache line at the specified index into the <i>ITagLo</i> and <i>ITagHi</i> registers. Read the data corresponding to the dword index into the <i>IDataLo</i> and <i>IDataHi</i> registers. If parity is implemented, read the parity bits corresponding to the data into <i>ErrCtl_{P1}</i>. 	Yes
2#001	D	Index Load Tag	Index	<ul style="list-style-type: none"> Read the tag for the cache line at the specified index into the <i>DTagLo</i> Coprocessor 0 register. Read the data corresponding to the word index into the <i>DDataLo</i> register. Data array parity bits are also read into the <i>ErrCtl</i> register. 	Yes
2#001	S	Index Load Tag	Index	<ul style="list-style-type: none"> Read the tag for the cache line at the specified index into the <i>L23TagLo</i> Coprocessor 0 register. Read the data corresponding to the dword index into the <i>L23DataLo</i> and <i>L23DataHi</i> registers. 	Yes
2#010	I	Index Store Tag	Index	<ul style="list-style-type: none"> Write the tag for the cache block at the specified index from the <i>ITagLo</i> and <i>ITagHi</i> registers. If parity is implemented, the parity written into the cache is generated by the hardware if <i>ErrCtl_{P0}</i> = 0, or it is obtained from <i>ITagLo</i> and <i>ITagHi</i> if <i>ErrCtl_{P0}</i> = 1. 	Yes
2#010	D,S	Index Store Tag	Index	<p>Write the tag for the cache line at the specified index from the associated <i>TagLo</i> Coprocessor 0 register.</p> <p>By default, the tag parity value will be automatically calculated. For test purposes, the parity/ECC bits from the associated <i>TagLo</i> register will be used if <i>ErrCtl_{P0}</i> is set.</p> <p>This encoding may be used by software to initialize the entire instruction or data caches by stepping through all valid indices. Doing so requires that the <i>TagLo</i> register associated with the cache be initialized first.</p>	Yes
2#011	I	Reserved	Unspecified	Executed as a no-op	No

Table 13.24 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST, DYT, SPR] Cleared (Continued)

Code	Caches	Name	Effective Address Operand Type	Operation	
2#011	S	Index Store Data	Index	Write the <i>L23DataHi</i> and <i>L23DataLo</i> Coprocessor 0 register contents at the way and dword index specified. The ECC bits are always generated by the hardware (if present)	Yes
2#100	All	Hit Invalidate	Address	If the cache line contains the specified address, set the state of the cache line to invalid. This 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.	Yes
2#101	I	Fill	Address	Fill the cache from the specified address. The cache line is refetched even if it is already in the cache. In that case, the existing copy in the cache is invalidated	Yes
	D, S	Hit WriteBack Invalidate	Address	If the cache line 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 line to invalid. If the line is valid but not dirty, set the state of the line to invalid. This 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.	Yes
2#110	D, S	Hit WriteBack	Address	If the cache line 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.	Yes
2#111	All	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. The way selected on fill from memory is the least recently used. The lock state is 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 with the lock bit reset in the associated <i>TagLo</i> register. It is illegal to lock all ways at a given cache index. If all ways are locked, subsequent references to that index will displace one of the locked lines.	Yes

Table 13.25 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST] Set, ErrCtl[DYT, SPR] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation	
2#001	I	Index Load WS	Index	Read the WS RAM at the specified index into the <i>ITagLo</i> Coprocessor 0 register.	Yes
2#001	D,S	Index Load WS	Index	Read the WS RAM at the specified index into the <i>associated TagLo</i> Coprocessor 0 register.	Yes
2#010	I	Index Store WS	Index	Update the WS RAM at the specified index from the <i>ITagLo</i> Coprocessor 0 register.	Yes
2#010	D	Index Store WS	Index	Update the WS RAM at the specified index from the <i>DTagLo</i> Coprocessor 0 register.	Yes
2#010	S	Index Store WS	Index	Update the WS RAM at the specified index from the <i>L23TagLo</i> Coprocessor 0 register. If <i>ErrCtl_{PO}</i> is set, the dirty parity values in the <i>L23TagLo</i> register will be written to the WS RAM. Otherwise, the parity will be calculated for the write data.	Yes
2#011	I	Index Store Data	Index	Write the <i>IDataLo</i> and <i>IDataHi</i> Coprocessor 0 register contents at the way and dword index specified. If <i>ErrCtl_{PO}</i> is set, <i>ErrCtl_{PI}</i> is used for the parity value. Otherwise, the parity value is calculated for the write data. In addition, the precode value for the write data is also updated in the tag if <i>ErrCtl_{PCD}</i> is not set. If <i>ErrCtl_{PCO}</i> is set, <i>ITagHi</i> is used for the precode value and its corresponding parity bit. Otherwise, the precode value and its corresponding parity bit are calculated based on the write data.	Yes
2#011	D	Index Store Data	Index	Write the <i>DDataLo</i> Coprocessor 0 register contents at the way and word index specified. If <i>ErrCtl_{PO}</i> is set, <i>ErrCtl_{PD}</i> is used for the parity value. Otherwise, the parity value is calculated for the write data.	Yes
2#011	S	Index Store ECC	Index	Write the <i>L23DataLo</i> Coprocessor 0 register contents to the <i>ECC</i> bits at the way and dword index specified.	Yes
All Others	All			Other codes should not be used while <i>ErrCtl_{WST}</i> is set.	

Table 13.26 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[SPR] Set, ErrCtl[DYT, WST] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation	
2#001	D	Index Load Tag	Index	Read the SPRAM tag at the specified index into the <i>TagLo1</i> Coprocessor 0 register.	Yes
2#010	D	Index Store Tag	Index	Update the SPRAM tag at the specified index from the <i>TagLo</i> Coprocessor 0 register.	Yes
2#011	D	Index Store Data	Index	Write the <i>DDataLo</i> Coprocessor 0 register contents into the SPRAM at the word index specified. The data parity is always calculated in this case.	Yes
All Others	D			Other codes should not be used while <i>ErrCtl_{SPR}</i> is set.	
All	S,T			Secondary and Tertiary operations should not be performed while <i>ErrCtl_{SPR}</i> is set.	

Note: *ErrCtl_{SPR}* is a don't care for cache operations to I-cache.

Table 13.27 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[DYT] Set, ErrCtl[SPR, WST] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation	
2#001	D	Index Load Tag	Index	Read the dirty RAM at the specified index into the <i>TagLo1</i> Coprocessor 0 register.	Yes
2#010	D	Index Store Tag	Index	Update the dirty RAM at the specified index from the <i>TagLo1</i> Coprocessor 0 register.	Yes
All Others	D			Other codes should not be used while <i>ErrCtl_{DYT}</i> is set.	

Note: *ErrCtl_{DYT}* is a don't care for cache operations to I-cache.

Restrictions:

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

The operation of this instruction is **UNDEFINED** if the operation requires an address, and that address is uncacheable.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```
vAddr ← GPR[base] + sign_extend(offset)
(pAddr, uncached) ← AddressTranslation(vAddr, DataReadReference)
```



```
CacheOp(op, vAddr, pAddr)
```

Exceptions:

TLB Refill Exception.

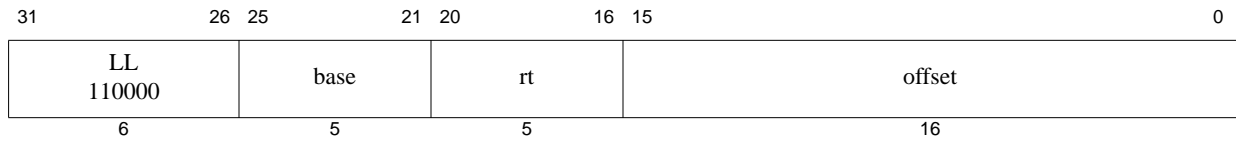
TLB Invalid Exception

Coprocessor Unusable Exception

Programming Notes:

For cache operations that require an index, it is implementation dependent whether the effective address or the translated physical address is used as the cache index. Therefore, the index value should always be converted to a kseg0 address by ORing the index with 0x80000000 before being used by the cache instruction. For example, the following code sequence performs a data cache Index Store Tag operation using the index passed in GPR a0:

```
li    a1, 0x80000000    /* Base of kseg0 segment */
or    a0, a0, a1        /* Convert index to kseg0 address */
cache DCIndexStTag, 0(a1) /* Perform the index store tag operation */
```

Format: LL *rt*, *offset*(*base*)

MIPS32

Purpose: Load Linked Word

To load a word from memory for an atomic read-modify-write

Description: $GPR[rt] \leftarrow memory[GPR[base] + offset]$

The LL and SC instructions provide the primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and written into GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

This begins a RMW sequence on the current processor. There can be only one active RMW sequence per processor. When an LL is executed, it starts an active RMW sequence replacing any other sequence that was active. The RMW sequence is completed by a subsequent SC instruction that either completes the RMW sequence atomically and succeeds, or does not and fails.

Executing LL on one processor does not cause an action that, by itself, causes an SC for the same block to fail on another processor.

An execution of LL does not have to be followed by execution of SC; a program is free to abandon the RMW sequence without attempting a write.

Restrictions:

The addressed location must be synchronizable by all processors and I/O devices sharing the location; if it is not, the result is **UNPREDICTABLE**. Which storage is synchronizable is a function of both CPU and system implementations. See the documentation of the SC instruction for the formal definition.

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

Operation:

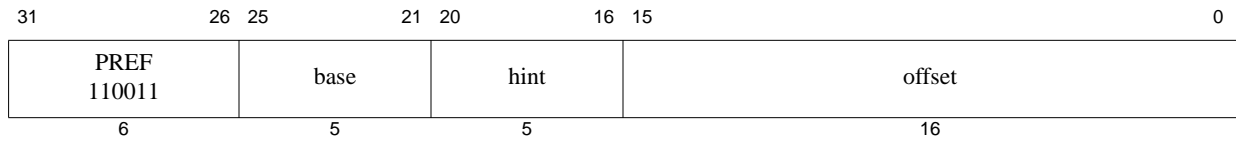
```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
memword ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
LLbit ← 1
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Watch

Programming Notes:

There is no Load Linked Word Unsigned operation corresponding to Load Word Unsigned.



Format: `PREF hint, offset(base)`

MIPS32

Purpose: Prefetch

To move data between memory and cache

Description: `prefetch_memory(GPR[base] + offset)`

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 the way that the data is expected to be used.

PREF does not cause addressing-related exceptions, including TLB exceptions. If the address specified would cause an addressing exception, the exception condition is ignored and no data movement occurs. However, even if no data is moved, some action that is not architecturally visible, such as writeback of a dirty cache line, can take place.

It is implementation-dependent whether a Bus Error or Cache Error exception is reported, if such an error is detected as a by-product of the action taken by the PREF instruction.

PREF neither generates a memory operation nor modifies the state of a cache line for a location with an *uncached* memory access type, whether this type is specified by the address segment (e.g., *kseg1*), the programmed coherency attribute of a segment (e.g., the use of the K0, KU, or K23 fields in the *Config* register), or the per-page coherency attribute provided by the TLB.

If PREF results in a memory operation, the memory access type and coherency attribute used for the operation are determined by the memory access type and coherency attribute of the effective address, just as it would be if the memory operation had been caused by a load or store to the effective address.

Table 13.28 Values of *hint* Field for PREF Instruction

Value	Name	Data Use and Desired Prefetch 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 - treated as a NOP.
4	load_streamed	Use: Prefetched data is expected to be read (not modified) but not reused extensively; it “streams” through cache. Action: Fetch data as if for a load. LRU replacement information is ignored and data is placed in way 0 of the cache, so it will be displaced by other streamed prefetches and not displace retained prefetches. If way 0 is locked, data will be placed in the way pointed to by the LRU.

Table 13.28 Values of *hint* Field for PREF Instruction

5	store_streamed	Use: Prefetched data is expected to be stored or modified but not reused extensively; it “streams” through cache. Action: Fetch data as if for a store. LRU replacement information is ignored and data is placed in way 0 of the cache, so it will be displaced by other streamed prefetches and not displace retained prefetches. If way 0 is locked, data will be placed in the way pointed to by the LRU.
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. LRU replacement information is used, but way 0 of the cache is specifically excluded. This prevents streamed prefetches from displacing the line.
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. LRU replacement information is used, but way 0 of the cache is specifically excluded. This prevents streamed prefetches from displacing the line.
8-24	Reserved	Reserved - treated as a NOP.
25	writeback_invalidate (also known as “nudge”)	Action: Schedule a writeback of any dirty data. The cache line is marked as invalid upon completion of the writeback. If cache line is locked, no action is taken. If a line is clean, it will be marked invalid and there will be no writeback scheduled.
26-29	Reserved	Reserved - treated as a NOP.
30	PrepareForStore	Use: Prepare the cache for writing an entire line, without the overhead involved in filling the line from memory. Action: If the reference hits in the cache, no action is taken. If the reference misses in the cache, a line is selected for replacement, any valid and dirty victim is written back to memory, the entire line is filled with zero data, and the state of the line is marked as valid and dirty. Programming Note: Because the cache line is filled with zero data on a cache miss, software must not assume that this action, in and of itself, can be used as a fast bzero-type function.
31	Fast Prepare For Store	Use: Prepare the cache for writing an entire line. No data is filled into the line. Action: If reference hits in the cache, no action is taken. If reference misses in the cache, a line is selected for replacement, any valid and dirty victim is written back to memory, the line is validated by writing the tag of the line while leaving the data as is. Programming Note: If the prefetch is not followed by real writes, it is possible that prevailing data and data parity may indicate a parity error.

Restrictions:

None

Operation:

```

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

```

Exceptions:

Bus Error, Cache Error

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

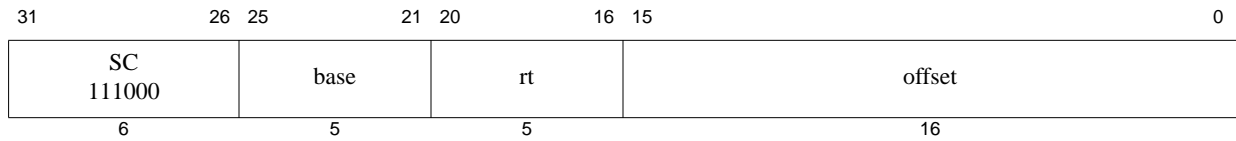
Programming Notes:

Prefetch cannot move data to or from 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. A prefetch may be used using an address pointer before the validity of the pointer is determined without worrying about an addressing exception.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a by-product of the action taken by the PREF instruction. Typically, this only occurs in systems which have high-reliability requirements.

Prefetch operations have no effect on cache lines that were previously locked with the CACHE instruction.



Format: SC *rt*, *offset*(*base*)

MIPS32

Purpose: Store Conditional Word

To store a word to memory to complete an atomic read-modify-write

Description: if *atomic_update* then $\text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{GPR}[\text{rt}]$, $\text{GPR}[\text{rt}] \leftarrow 1$
else $\text{GPR}[\text{rt}] \leftarrow 0$

The LL and SC instructions provide primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The 32-bit word in GPR *rt* is conditionally stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

The SC completes the RMW sequence begun by the preceding LL instruction executed on the processor. To complete the RMW sequence atomically, the following occur:

- The 32-bit word of GPR *rt* is stored into memory at the location specified by the aligned effective address.
- A 1, indicating success, is written into GPR *rt*.

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR *rt*.

If the following event occurs between the execution of LL and SC, the SC fails:

- An ERET instruction is executed.

If either of the following events occurs between the execution of LL and SC, the SC may succeed or it may fail; the success or failure is not predictable. Portable programs should not cause one of these events.

- A memory access instruction (load, store, or prefetch) is executed on the processor executing the LL/SC.
- The instructions executed starting with the LL and ending with the SC do not lie in a 2048-byte contiguous region of virtual memory. (The region does not have to be aligned, other than the alignment required for instruction words.)

The following conditions must be true or the result of the SC is **UNPREDICTABLE**:

- Execution of SC must have been preceded by execution of an LL instruction.
- An RMW sequence executed without intervening events that would cause the SC to fail must use the same address in the LL and SC. The address is the same if the virtual address, physical address, and cache-coherence algorithm are identical.

Restrictions:

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

$$\text{vAddr} \leftarrow \text{sign_extend}(\text{offset}) + \text{GPR}[\text{base}]$$

```

if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
if LLbit then
    StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
endif
GPR[rt] ← 0 || LLbit

```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

Programming Notes:

LL and SC are used to atomically update memory locations, as shown below.

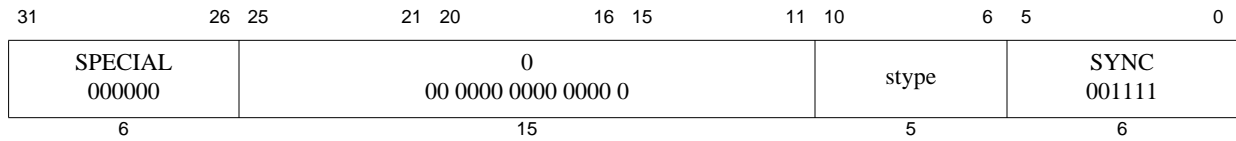
```

L1:
    LL      T1, (T0) # load counter
    ADDI   T2, T1, 1 # increment
    SC     T2, (T0) # try to store, checking for atomicity
    BEQ    T2, 0, L1 # if not atomic (0), try again
    NOP                    # branch-delay slot

```

Exceptions between the LL and SC cause SC to fail, so persistent exceptions must be avoided. Some examples of these are arithmetic operations that trap, system calls, and floating point operations that trap or require software emulation assistance.

LL and SC function on a single processor for *cached noncoherent* memory so that parallel programs can be run on uniprocessor systems that do not support *cached coherent* memory access types.



Format: SYNC (stype = 0 implied)

MIPS32

Purpose: Synchronize Shared Memory

To order loads and stores.

Description:

These types of ordering guarantees are available through the SYNC instruction:

- Completion Barriers
- Ordering Barriers

Simple Description of Completion Barrier:

- SYNC affects only *uncached* and *cached coherent* loads and stores. The loads and stores that occur before the SYNC must be completed before the loads and stores after the SYNC are allowed to start.
- Loads are completed when the destination register is written. Stores are completed when the stored value is visible to every other processor in the system.
- SYNC is required, potentially in conjunction with SSNOP (in Release 1 of the Architecture) or EHB (in Release 2 of the Architecture), to guarantee that memory reference results are visible across operating mode changes. For example, a SYNC is required on entry to and exit from Debug Mode to guarantee that memory affects are handled correctly.

Detailed Description of Completion Barrier:

- SYNC does not guarantee the order in which instruction fetches are performed. A stype value of zero will always be defined such that it performs the most complete set of synchronization operations that are defined. This means stype zero always does a completion barrier that affects both loads and stores preceding the SYNC instruction and both loads and stores that are subsequent to the SYNC instruction. Non-zero values of stype may be defined by the architecture or specific implementations to perform synchronization behaviors that are less complete than that of stype zero. If an implementation does not use one of these non-zero values to define a different synchronization behavior, then that non-zero value of stype must act the same as stype zero completion barrier. This allows software written for an implementation with a lighter-weight barrier to work on another implementation which only implements the stype zero completion barrier.

Simple Description of Ordering Barrier:

- The specified memory instructions (loads or stores or both) that occur before the SYNC instruction must always be ordered before the specified memory instructions after the SYNC.
- Memory instructions which are ordered before other memory instructions are processed by the load/store datapath first before the other memory instructions.

Detailed Description of Ordering Barrier:

- Every synchronizable specified memory instruction (loads or stores or both) that occurs in the instruction stream before the SYNC instruction must reach a stage in the load/store datapath after which no instruction re-ordering is possible before any synchronizable specified memory instruction which occurs after the SYNC instruction in the instruction stream reaches the same stage in the load/store datapath.
- If any memory instruction before the SYNC instruction in program order, generates a memory request to the external memory and any memory instruction after the SYNC instruction in program order also generates a memory request to external memory, the memory request belonging to the older instruction must be globally performed before the time the memory request belonging to the younger instruction is globally performed.
- The barrier does not guarantee the order in which instruction fetches are performed.

As compared to the completion barrier, the ordering barrier is a lighter-weight operation as it does not require the specified instructions before the SYNC to be already completed. Instead it only requires that those specified instructions which are subsequent to the SYNC in the instruction stream are never re-ordered for processing ahead of the specified instructions which are before the SYNC in the instruction stream. This potentially reduces how many cycles the barrier instruction must stall before it completes.

The Acquire and Release barrier types are used to minimize the memory orderings that must be maintained and still have software synchronization work.

Because the core processes loads and stores in order, ordering barriers are much lighter weight. The LSU does not issue the RMB and Acquire barrier types to the BIU. For the MB, WMB, Release barrier types, the LSU will complete any store hits and hit-type cacheops before issuing them to the BIU. The BIU will stop merging on all WBB entries. No external request will be generated and the core will not wait for pending transactions to complete.

For the purposes of this description, the CACHE, PREF and PREFX instructions are treated as loads and stores. That is, these instructions and the memory transactions sourced by these instructions obey the ordering and completion rules of the SYNC instruction.

Table 13.29 lists the available completion barrier and ordering barriers behaviors that can be specified using the stype field.

Table 13.29 Encodings of the Bits[10:6] of the SYNC instruction; the SType Field

Code	Name	Older instructions which must reach the load/store ordering point before the SYNC instruction completes.	Younger instructions which must reach the load/store ordering point only after the SYNC instruction completes.	Older instructions which must be globally performed when the SYNC instruction completes
0x0	SYNC or SYNC(0)	Loads, Stores	Loads, Stores	Loads, Stores
0x4	SYNC_WMB or SYNC(4)	Stores	Stores	

Table 13.29 Encodings of the Bits[10:6] of the SYNC instruction; the SType Field

Code	Name	Older instructions which must reach the load/store ordering point before the SYNC instruction completes.	Younger instructions which must reach the load/store ordering point only after the SYNC instruction completes.	Older instructions which must be globally performed when the SYNC instruction completes
0x10	SYNC_MB or SYNC(16)	Loads, Stores	Loads, Stores	
0x11	SYNC_ACQUIRE or SYNC(17)	Loads	Loads, Stores	
0x12	SYNC_RELEASE or SYNC(18)	Loads, Stores	Stores	
0x13	SYNC_RMB or SYNC(19)	Loads	Loads	
0x1, 0x5-0xF, 0x14 - 0x1F	RESERVED			

Restrictions:

The effect of SYNC on the global order of loads and stores for memory access types other than *uncached* and *cached coherent* is **UNPREDICTABLE**.

Operation:

`SyncOperation(stype)`

Exceptions:

None

31	26	25	24	6	5	0
COP0 010000	CO 1	Implementation-Dependent Code			WAIT 100000	
6	1	19			6	

Format: WAIT

MIPS32

Purpose: Enter Standby Mode

Wait for Event

Description:

The WAIT instruction forces the core into low-power mode. The pipeline is stalled, and when all external requests are completed, the processor's main clock is stopped. The processor will restart when reset (*SI_Reset* or *SI_ColdReset*) is signaled or when an interrupt is signalled, irrespective of whether the interrupt is enabled or not. (*SI_NMI*, *SI_Int*, or *EJ_DINT*). Note that the core does not use the code field in this instruction. If the pipeline restarts as the result of an interrupt, that interrupt is taken between the WAIT instruction and the following instruction (*EPC* for the interrupt points to the instruction following the WAIT instruction).

Restrictions:

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

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```
I: Enter lower power mode
I+1: /* Potential interrupt taken here */
```

Exceptions:

Coprocessor Unusable Exception

MIPS16e™ Application-Specific Extension to the MIPS32® Instruction Set

This chapter describes the MIPS16e™ ASE as implemented in the 74K core. Refer to *Volume IV-a* of the *MIPS32 Architecture Reference Manual* for a general description of the MIPS16e ASE and detailed descriptions of the instructions.

This chapter covers the following topics:

- [Section 14.1 “Instruction Bit Encoding”](#)
- [Section 14.2 “Instruction Listing”](#)

14.1 Instruction Bit Encoding

[Table 14.2](#) through [Table 14.9](#) describe the encoding used for the MIPS16e ASE. [Table 14.1](#) describes the meaning of the symbols used in the tables.

Table 14.1 Symbols Used in the Instruction Encoding Tables

*	Operation or field codes marked with this symbol are reserved for future use. Executing such an instruction causes a Reserved Instruction Exception.
δ	(Also <i>italic</i> field name.) Operation or field codes marked with this symbol denote 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 causes a Reserved Instruction Exception.
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 (<i>SPECIAL2</i> 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).
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 14.2 MIPS16e Encoding of the Opcode Field

opcode		bits 13..11							
		0	1	2	3	4	5	6	7
bits 15..14		000	001	010	011	100	101	110	111
0	00	ADDIUSP ¹	ADDIUPC ²	B	JAL(X) δ	BEQZ	BNEZ	SHIFT δ	β
1	01	RRI-A δ	ADDIU8 ³	SLTI	SLTIU	I8 δ	LI	CMPI	β
2	10	LB	LH	LWSP ⁴	LW	LBU	LHU	LWPC ⁵	β
3	11	SB	SH	SWSP ⁶	SW	RRR δ	RR δ	EXTEND δ	β

1. The ADDIUSP opcode is used by the ADDIU rx, sp, immediate instruction
2. The ADDIUPC opcode is used by the ADDIU rx, pc, immediate instruction
3. The ADDIU8 opcode is used by the ADDIU rx, immediate instruction
4. The LWSP opcode is used by the LW rx, offset(sp) instruction
5. The LWPC opcode is used by the LW rx, offset(pc) instruction
6. The SWSP opcode is used by the SW rx, offset(sp) instruction

Table 14.3 MIPS16e JAL(X) Encoding of the x Field

x	bit 26	
	0	1
	JAL	JALX

Table 14.4 MIPS16e SHIFT Encoding of the f Field

f	bits 1..0			
	0	1	2	3
	00	01	10	11
	SLL	β	SRL	SRA

Table 14.5 MIPS16e RRI-A Encoding of the f Field

f	bit 4	
	0	1
	ADDIU ¹	β

1. The ADDIU function is used by the ADDIU ry, rx, immediate instruction

Table 14.6 MIPS16e I8 Encoding of the funct Field

funct	bits 10..8							
	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
	BTEQZ	BTNEZ	SWRASP ¹	ADJSP ²	SVRS δ	MOV32R ³	*	MOVR32 ⁴

1. The SWRASP function is used by the SW ra, offset(sp) instruction
2. The ADJSP function is used by the ADDIU sp, immediate instruction
3. The MOV32R function is used by the MOVE r32, rz instruction

4. The MOVR32 function is used by the MOVE ry, r32 instruction

Table 14.7 MIPS16e RRR Encoding of the f Field

f	bits 1..0			
	0	1	2	3
	00	01	10	11
	β	ADDU	β	SUBU

Table 14.8 MIPS16e RR Encoding of the Funct Field

funct		bits 2..0							
		0	1	2	3	4	5	6	7
bits 4..3		000	001	010	011	100	101	110	111
0	00	<i>J(AL)R(C) δ</i>	SDBBP	SLT	SLTU	SLLV	BREAK	SRLV	SRAV
1	01	β	*	CMP	NEG	AND	OR	XOR	NOT
2	10	MFHI	<i>CNVT δ</i>	MFLO	β	β	*	β	β
3	11	MULT	MULTU	DIV	DIVU	β	β	β	β

Table 14.9 MIPS16e I8 Encoding of the s Field when funct=SVRS

s	bit 7	
	0	1
	RESTORE	SAVE

Table 14.10 MIPS16e RR Encoding of the ry Field when funct=J(AL)R(C)

ry	bits 7..5							
	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
	JR rx	JR ra	JALR	*	JRC rx	JRC ra	JALRC	*

Table 14.11 MIPS16e RR Encoding of the ry Field when funct=CNVT

ry	bits 7..5							
	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
	ZEB	ZEH	β	*	SEB	SEH	β	*

14.2 Instruction Listing

The MIPS16e instructions are listed by instruction type in [Table 14.12](#) through [Table 14.19](#).

Table 14.12 MIPS16e Load and Store Instructions

Mnemonic	Instruction	Extensible Instruction
LB	Load Byte	Yes
LBU	Load Byte Unsigned	Yes
LH	Load Halfword	Yes
LHU	Load Halfword Unsigned	Yes
LW	Load Word	Yes
SB	Store Byte	Yes
SH	Store Halfword	Yes
SW	Store Word	Yes

Table 14.13 MIPS16e Save and Restore Instructions

Mnemonic	Instruction	Extensible Instruction
RESTORE	Restore Registers and Deallocate Stack Frame	Yes
SAVE	Save Registers and Setup Stack Frame	Yes

Table 14.14 MIPS16e ALU Immediate Instructions

Mnemonic	Instruction	Extensible Instruction
ADDIU	Add Immediate Unsigned	Yes
CMPI	Compare Immediate	Yes
LI	Load Immediate	Yes
SLTI	Set on Less Than Immediate	Yes
SLTIU	Set on Less Than Immediate Unsigned	Yes

Table 14.15 MIPS16e Arithmetic Two or Three Operand Register Instructions

Mnemonic	Instruction	Extensible Instruction
ADDU	Add Unsigned	No
AND	AND	No
CMP	Compare	No
MOVE	Move	No
NEG	Negate	No

Table 14.15 MIPS16e Arithmetic Two or Three Operand Register Instructions

Mnemonic	Instruction	Extensible Instruction
NOT	Not	No
OR	OR	No
SEB	Sign-Extend Byte	No
SEH	Sign-Extend Halfword	No
SLT	Set on Less Than	No
SLTU	Set on Less Than Unsigned	No
SUBU	Subtract Unsigned	No
XOR	Exclusive OR	No
ZEB	Zero-Extend Byte	No
ZEH	Zero-Extend Halfword	No

Table 14.16 MIPS16e Special Instructions

Mnemonic	Instruction	Extensible Instruction
BREAK	Breakpoint	No
SDBBP	Software Debug Breakpoint	No
EXTEND	Extend	No

Table 14.17 MIPS16e Multiply and Divide Instructions

Mnemonic	Instruction	Extensible Instruction
DIV	Divide	No
DIVU	Divide Unsigned	No
MFHI	Move From HI	No
MFLO	Move From LO	No
MULT	Multiply	No
MULTU	Multiply Unsigned	No

Table 14.18 MIPS16e Jump and Branch Instructions

Mnemonic	Instruction	Extensible Instruction
B	Branch Unconditional	Yes
BEQZ	Branch on Equal to Zero	Yes
BNEZ	Branch on Not Equal to Zero	Yes

Table 14.18 MIPS16e Jump and Branch Instructions

Mnemonic	Instruction	Extensible Instruction
BTEQZ	Branch on T Equal to Zero	Yes
BTNEZ	Branch on T Not Equal to Zero	Yes
JAL	Jump and Link	No
JALR	Jump and Link Register	No
JALRC	Jump and Link Register Compact	No
JALX	Jump and Link Exchange	No
JR	Jump Register	No
JRC	Jump Register Compact	No

Table 14.19 MIPS16e Shift Instructions

Mnemonic	Instruction	Extensible Instruction
SRA	Shift Right Arithmetic	Yes
SRAV	Shift Right Arithmetic Variable	No
SLL	Shift Left Logical	Yes
SLLV	Shift Left Logical Variable	No
SRL	Shift Right Logical	Yes
SRLV	Shift Right Logical Variable	No

References

This appendix lists other publications available from MIPS Technologies, Inc. that are referenced in this document. These documents may be included in the `$MIPS_PROJECT/doc` area of a typical 74K soft or hard core release, or in some cases may be available on the MIPS web site <http://www.mips.com>.

1. MIPS32® Architecture For Programmers, Volume I: Introduction to the MIPS32 Architecture
MIPS document: MD0082
2. MIPS32® Architecture For Programmers, Volume II: The MIPS32 Instruction Set
MIPS document: MD0082
3. MIPS32® Architecture For Programmers, Volume IV-e: The MIPS® DSP Application-Specific Extension to the MIPS32 Architecture
MIPS document: MD00374
4. Programming the MIPS32® 74K™ Processor Core Family
MIPS document: MD00541
5. Programming the MIPS 74K™ Family Cores for DSP
MIPS document: MD00544
6. MIPS32® 74K™ Processor Core Family Integrator's Guide
MIPS document: MD00499
7. MIPS32® 74K™ Processor Core Family Implementor's Guide
MIPS document: MD00498
8. MIPS32® 74K™ Processor Core System User's Manual
MIPS document: MD00647
9. CorExtend™ Instruction Integrator's Guide for MIPS32® 74K™ Cores
MIPS document: MD00523
10. Getting Started with CorExtend® Instructions for MIPS32® 74K™ Cores
MIPS document: MD00524
11. MIPS® EJTAG Specification
MIPS document: MD00047
12. MIPS® PDtrace™ Interface and Trace Control Block Specification
MIPS document: MD00439
13. PDtrace™ and TCB Usage Guidelines
MIPS document: MD00365

References

14. MIPS32® Architecture For Programmers, Volume III: Privileged Resource Architecture
MIPS document: MD00088

Revision History

MIPS documents include change bars (vertical bars in the page margin) that mark significant changes to the document since its last release. Change bars are removed for changes which are more than one revision old.

This document may refer to Architecture specifications (for example, instruction set descriptions and EJTAG register definitions), and change bars in these sections indicate changes since the previous version of the relevant Architecture document.

Date	Revision	Description
January 31, 2007	01.00	Initial external release
May 25, 2007	01.01	Updates to Pipeline description, CP0 registers, FPU pipeline, CACHE and WAIT instructions
Dec 14, 2007	01.02	Updated to reflect: <ul style="list-style-type: none"> • Reduction in effective pipeline length by 2 stages (removal of AP, EF and GR stages) • Instruction buffer size increase to 12 entries • Addition of Instruction cache Prefetching and associated controls in <i>Config7_{PREF}</i> • Addition of <i>UserLocal</i> Coprocessor 0 register and associated controls in <i>Config3_{ULRj}</i>, <i>HWREn_{al}</i> • Addition of <i>Config7_{Wll}</i> to enable unblocking of wait even if <i>Status_{IE}</i> is cleared • Inclusion of <i>L23DataLo</i> and <i>L23DataHi</i> and renaming of <i>STagLo</i> to <i>L23TagLo</i> • Corrections to descriptions of <i>ITagLo</i> and <i>DTagLo</i> • Enhanced description of <i>PerfCtl</i> register • Removed <i>Config6</i> register • Revised description of <i>CacheErr</i> register
November 14, 2008	01.03	<ul style="list-style-type: none"> • Updated Fig.7.37 to mark bit[10] as unused • Add section on PDtrace, including new registers • Modify document to reflect support for Instruction Scratchpad RAM (ISPRAM) and optional I-cache size of 0KB
June 04, 2010	01.04	<ul style="list-style-type: none"> • Add IAR and IVA bits, updated ES bit description in <i>Config7</i> Register fields in CP0 Registers chapter. Changed UX, SX, KX, and PX bits in Status Register to R (Reserved. reads as 0). • Add <i>FastDebugChannel</i> and <i>Common Device Memory Map</i> descriptions • Add new CP0 registers <i>Config6</i>, <i>CDMMBase</i>, and <i>ContextConfig</i> • Add new drseg register <i>DebugVectorAddr</i>

Revision History

Date	Revision	Description
March 30, 2011	01.05	<ul style="list-style-type: none">• Add description of 6 new bits in DCR.• Add description of 2 new bits in Implementation register.• Updated hazard tables.• Updated description of DC bit in Cause register.