MIPS32® M4K™ Processor Core Software User's Manual

Document Number: MD00249
Revision 02.03
August 29, 2008

MIPS Technologies, Inc.
955 East Arques Avenue
Sunnyvale, CA 94085-4521

MIPS Technologies reserves the right to change the information contained in this document to improve function, design or otherwise. MIPS Technologies does 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 Technologies 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.

Template: nB1.01, Built with tags: 2B MIPS32 PROC

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction to the MIPS32® M4K™ Processor Core

The MIPS32® M4K™ core from MIPS Technologies is a high-performance, low-power, 32-bit MIPS RISC processor core intended for 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 M4K 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 M4K core is ideally positioned to support new products for emerging segments of the routing, network access, network storage, residential gateway, and smart mobile device markets. It is especially well-suited for applications where high performance density is critical, especially those requiring multiple processor cores on a single chip.

The M4K family has two members, distinguished by the range of build-time options available:

- MIPS32 M4K™ Core: Fully configurable cacheless core.
- MIPS32 M4K™ Lite Core: A subset of the full M4K core, with a reduced set of build-time configuration choices.

The term *M4K core* used throughout this document generally refers to all members of the M4K family. Since the M4K Lite core has fewer configuration options than the M4K core, certain features described in this document may not be available on the M4K Lite version.

The core implements the MIPS32 Release 2 Instruction Set Architecture (ISA), and may optionally support the MIPS16e Application Specific Extension (ASE) for code compression. The MMU consists of a simple Fixed Mapping Translation (FMT) mechanism, for applications that do not require the full capabilities of a Translation Lookaside Buffer- (TLB-) based MMU available on other MIPS cores.

The M4K core is cacheless; in lieu of caches, it includes a simple interface to SRAM-style devices. This interface may be configured for independent instruction and data devices or combined into a unified interface. The SRAM interface allows deterministic latency to memory, while still maintaining high performance.

The core includes one of two different Multiply/Divide Unit (MDU) implementations, selectable at build-time, allowing the user to trade off performance and area for integer multiply and divide operations. The high-performance MDU option implements single cycle multiply and multiply-accumulate (MAC) instructions, which enable DSP algorithms to be performed efficiently. It allows 32-bit x 16-bit MAC instructions to be issued every cycle, while a 32-bit x 32-bit MAC instruction can be issued every other cycle. The area-efficient MDU option handles multiplies with a one-bit-per-clock iterative algorithm.

The basic Enhanced JTAG (EJTAG) features provide CPU run control with stop, single stepping and re-start, and with software breakpoints through the SDBBP instruction. Additional EJTAG features - instruction and data virtual address hardware breakpoints, complex hardware breakpoints, connection to an external EJTAG probe through the Test Access Port (TAP), and PC/Data tracing, may optionally be included.

The rest of this chapter provides an overview of the MIPS32 M4K processor core and consists of the following sections:

- Section 1.1 “Features”
- Section 1.2 “M4K™ Core Block Diagram”

1.1 Features

- 5-stage pipeline
- 32-bit Address and Data Paths
- 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
  - Bit field manipulation instructions
- 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)
- Programmable Memory Management Unit

    - Simple Fixed Mapping Translation (FMT)
    - Address spaces mapped using register bits
- Simple SRAM-Style Interface
    - Cacheless operation enables deterministic response and reduces size
    - 32-bit address and data; input byte enables enable simple connection to narrower devices
    - Single or multi-cycle latencies
    - Configuration option for dual or unified instruction/data interfaces
    - Redirection mechanism on dual I/D interfaces permits D-side references to be handled by I-side
    - Transactions can be aborted to improve interrupt latency
- Multi-Core Support
    - External lock indication enables multi-processor semaphores based on LL/SC instructions
    - External sync indication allows memory ordering
    - Debug support includes cross-core triggers
- CorExtend™ User Defined Instruction capability (access to this feature is available in the M4K Pro™ cores and requires a separate license)
    - Optional support for the CorExtend feature allows users to define and add instructions to the core (as a build-time option)
    - Single or multi-cycle instructions
    - Source operations from register, immediate field, or local state
    - Destination to a register or local state
- Full featured Coprocessor 2 Interface
    - Almost all I/Os registered
    - Separate unidirectional 32-bit instruction and data buses
    - Support for branch on Coprocessor condition
    - Processor to/from Coprocessor register data transfers
    - Direct memory to/from Coprocessor register data transfers
- Multiply-Divide Unit (High performance build-time option)
    - Maximum issue rate of one 32x16 multiply per clock

  - Maximum issue rate of one 32x32 multiply every other clock
  - Early-in divide control. Minimum 11, maximum 34 clock latency on divide
- Multiply-Divide Unit (Area-efficient build-time option)
  - Iterative multiply and divide. 32 or more cycles for each instruction.
- 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
- EJTAG Debug Support
  - CPU control with start, stop and single stepping
  - Software breakpoints via the SDBBP instruction
  - Optional simple hardware breakpoints on virtual addresses; 4 instruction and 2 data breakpoints, 2 instruction and 1 data breakpoint, or no breakpoints
  - Optional complex hardware breakpoints with 6 instruction and 2 data simple breakpoints, plus ability to specify combinations of breakpoints for more specific break conditions
  - Optional Test Access Port (TAP) facilitates high speed download of application code
  - Optional trace hardware to enable real-time tracing of executed code

1.2 M4K™ Core Block Diagram

The M4K core contains both required and optional blocks, as shown in the block diagram in Figure 1.1. Required blocks are the lightly shaded areas of the block diagram and are always present in any core implementation. Optional blocks may be added to the base core, depending on the needs of a specific implementation. The required blocks are as follows:

- Execution Unit
- Multiply-Divide Unit (MDU)
- System Control Coprocessor (CP0)
- Memory Management Unit (MMU)
- Cache Controller
- SRAM Interface

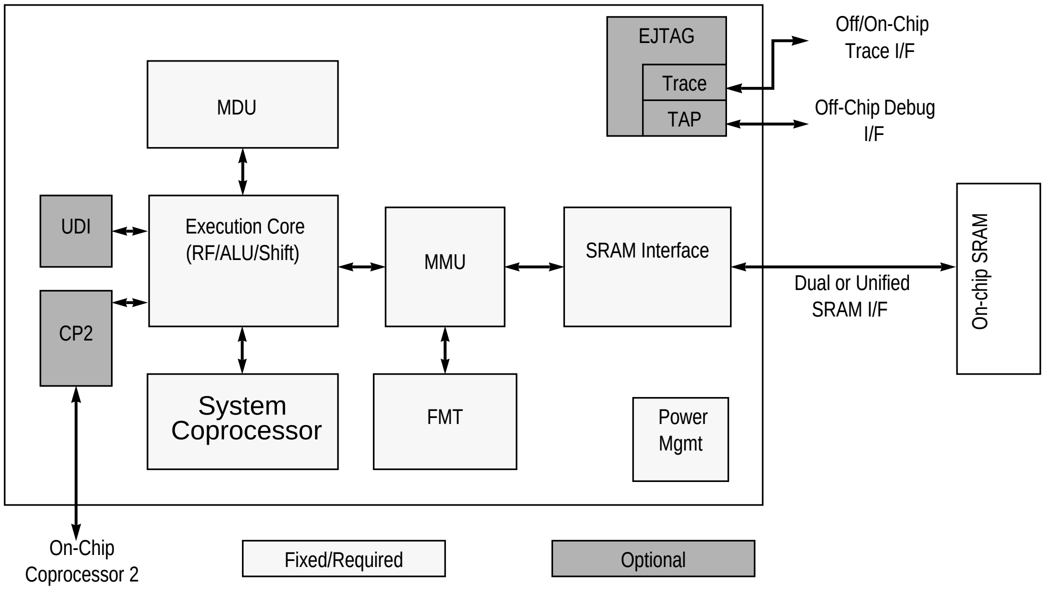
- Power Management

Optional blocks include:

- Enhanced JTAG (EJTAG) Controller
- MIPS16e support
- Coprocessor 2 Interface (CP2)
- CorExtend® User Defined Instructions (UDI)

Figure 1.1 shows a block diagram of a M4K core.

Figure 1.1 M4K™ Processor Core Block Diagram



1.2.1 Required Logic Blocks

The following subsections describe the various required logic blocks of the M4K processor core.

1.2.1.1 Execution Unit

The core execution unit implements a load-store architecture with single-cycle Arithmetic Logic Unit (ALU) operations (logical, shift, add, subtract) and an autonomous multiply-divide unit. The core contains thirty-two 32-bit general-purpose registers(GPRs) used for scalar integer operations and address calculation. Optionally, one or three additional register file shadow sets (each containing thirty-two registers) can be added to minimize context switching overhead during interrupt/exception processing. The register file consists of two read ports and one write port and is fully bypassed to minimize operation latency in the pipeline.

The execution unit includes:

- 32-bit adder used for calculating the data address

- Address unit for calculating the next instruction address
- Logic for branch determination and branch target address calculation
- Load aligner
- Bypass multiplexers used to avoid stalls when executing instruction streams where data-producing instructions are followed closely by consumers of their results
- Zero/One detect unit for implementing the CLZ and CLO instructions
- ALU for performing bitwise logical operations
- Shifter and Store aligner

1.2.1.2 Multiply/Divide Unit (MDU)

The Multiply/Divide unit performs multiply divide operations. Two configuration options exist for the MDU, selectable at build time: an area-efficient iterative MDU and a higher performance 32x16 array. The MDU consists of an iterative or32x16 multiplier, result-accumulation registers (HI and LO), multiply and divide state machines, and all multiplexers and control logic required to perform these functions. The high-performance, pipelined MDU supports execution of a 16x16 or 32x16 multiply operation every clock cycle; 32x32 multiply operations can be issued every other clock cycle. Appropriate interlocks are implemented to stall the issue of back-to-back 32x32 multiply operations. Divide operations are implemented with a simple 1 bit per clock iterative algorithm and require 35 clock cycles in worst case to complete. Early-in to the algorithm detects sign extension of the dividend, if it is actual size is 24, 16 or 8 bit. the divider will skip 7, 15 or 23 of the 32 iterations. An attempt to issue a subsequent MDU instruction while a divide is still active causes a pipeline stall until the divide operation is completed.

The area-efficient, non-pipelined MDU consists of a 32-bit full-adder, result-accumulation registers (HI and LO), a combined multiply/divide state machine, and all multiplexers and control logic required to perform these functions. It performs any multiply using 32 cycles in an iterative 1 bit per clock algorithm. Divide operations are also implemented with a simple 1 bit per clock iterative algorithm (no early-in) and require 35 clock cycles to complete. An attempt to issue a subsequent MDU instruction while a multiply/divide is still active causes a pipeline stall until the operation is completed.

The M4K implements an additional multiply instruction, MUL, which specifies that lower 32-bits of the multiply result be placed in the register file instead of the HI/LO register pair. By avoiding the explicit move from LO (MFLO) instruction, required when using the LO register, and by supporting multiple destination registers, the throughput of multiply-intensive operations is increased.

Two instructions, multiply-add (MADD/MADDU) and multiply-subtract (MSUB/MSUBU), are used to perform the multiply-add and multiply-subtract operations. The MADD instruction multiplies two numbers and then adds the product to the current contents of the HI and LO registers. Similarly, the MSUB instruction multiplies two operands and then subtracts the product from the HI and LO registers. The MADD/MADDU and MSUB/MSUBU operations are commonly used in Digital Signal Processor (DSP) algorithms.

1.2.1.3 System Control Coprocessor (CP0)

In the MIPS architecture, CP0 is responsible for the virtual-to-physical address translation, cache protocols, the exception control system, the processor’s diagnostics capability, operating mode selection (kernel vs. user mode), and the enabling/disabling of interrupts. Configuration information such as presence of build-time options are available by accessing the CP0 registers. Refer to Chapter 5, “CP0 Registers of the M4K™ Core” on page 85 for more information on the CP0 registers. Refer to Chapter 8, “EJTAG Debug Support in the M4K™ Core” on page 127 for more information on EJTAG debug registers.

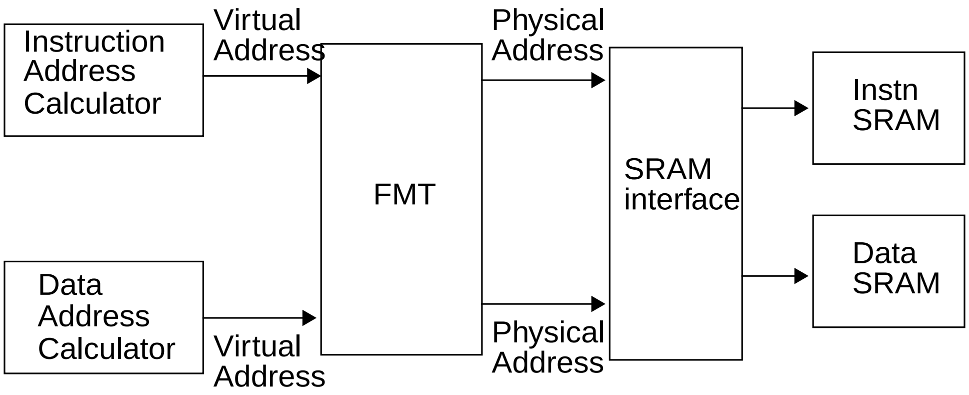
1.2.1.4 Memory Management Unit (MMU)

The M4K core contains an MMU that interfaces between the execution unit and the SRAM controller, shown in Figure 1.2.

The M4K implement a FMT-based MMU.The FMT performs a simple translation to get the physical address from the virtual address. Refer to Chapter 3, “Memory Management of the M4K™ Core” on page 43 for more information on the FMT.

Figure 1.2 shows how the address translation mechanism interacts with SRAM access.

Figure 1.2 Address Translation During a SRAM Access



1.2.1.5 SRAM Interface

Instead of caches, the M4K core contains an interface to SRAM-style memories that can be tightly coupled to the core. This permits deterministic response time with less area than is typically required for caches. The SRAM interface includes separate unidirectional 32-bit buses for address, read data, and write data.

Dual or Unified Interfaces

The SRAM interface includes a build-time option to select either dual or unified instruction and data interfaces.The dual interface enables independent connection to instruction and data devices. It generally yields the highest performance, since the pipeline can generate simultaneous I and D requests which are then serviced in parallel. For simpler or cost-sensitive systems, it is also possible to combine the I and D interfaces into a common interface that services both types of requests. If I and D requests occur simultaneously, priority is given to the D side.

Backstalling

Typically, read or write transactions will complete in a single cycle. If multi-cycle latency is desired, however, the interface can be stalled to allow connection to slower devices.

Redirection

When the dual I/D interface is present, a mechanism exists to divert D-side references to the I-side, if desired. The redirection is employed automatically in the case of PC-relative loads in MIPS16e mode. The mechanism can be explicitly invoked for any other D-side references, as well. When the *DS_Redir* signal is asserted, a D-side request is diverted to the I-side interface in the following cycle, and the D-side will be stalled until the transaction is completed.

Transaction Abort

Because the core does not know whether loads or stores are re-startable, it cannot arbitrarily interrupt a request which has been initiated on the SRAM interface. However, cycles spent waiting for a multi-cycle transaction to complete can directly impact interrupt latency. In order to minimize this effect, the interface supports an abort mechanism. The core requests an abort whenever an interrupt is detected and a transaction is pending. The external system logic can choose to acknowledge the abort, if it wants to reduce interrupt latency.

MIPS16e Execution

When the core is operating in MIPS16e mode, instruction fetches only require 16-bits of data to be returned. For improved efficiency, however, the core will fetch 32-bits of instruction data whenever the address is word-aligned. Thus for sequential MIPS16e code, fetches only occur for every other instruction, resulting in better performance and reduced system power.

Connecting to Narrower Devices

The instruction and data read buses are always 32-bits in width. To facilitate connection to narrower memories, the SRAM interface protocol includes input byte enables that can be used by system logic to signal validity as partial read data becomes available. The input byte enables conditionally register the incoming read data bytes within the core, and thus eliminate the need for external registers to gather the entire 32-bits of data. External muxes are required to redirect the narrower data to the appropriate byte lanes.

Lock Mechanism

The SRAM interface includes a protocol to identify a locked sequence, and is used in conjunction with the LL/SC atomic read-modify-write semaphore instructions.

Sync Mechanism

The interface includes a protocol that externalizes the execution of the SYNC instruction. External logic might choose to use this information to enforce memory ordering between various elements in the system.

External Call Indication

The interface has an indication when a fetch is for the target of a call-type instruction like JAL or BAL. A system with prefetching might choose to save prefetched instructions to be executed when there is a return from the subroutine.

1.2.1.6 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 7, “Power Management of the M4K™ Core” on page 125 for more information on power management.

1.2.2 Optional Logic Blocks

The core consists of the following optional logic blocks as shown in the block diagram in Figure 1.1.

1.2.2.1 MIPS16e™ Application Specific Extension

The M4K core includes optional support for the MIPS16e ASE. This ASE improves code density through the use of 16-bit encodings of MIPS32 instructions plus some MIPS16e-specific instructions. PC relative loads allow quick access to constants. Save/Restore macro instructions provide for single instruction stack frame setup/teardown for efficient subroutine entry/exit. Sign- and zero-extend instructions improve handling of 8bit and 16bit datatypes.

A decompressor converts the MIPS16e 16-bit instructions fetched from the external interface back into 32-bit instructions for execution by the core.

1.2.2.2 EJTAG Controller

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 for the basic software debug of user and kernel code.

Optional EJTAG features include hardware breakpoints. A M4K core may have up to six instruction breakpoints and two data breakpoints and potentially support for complex 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 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 values may apply in the address compare, and byte mask may apply in the value compare.

Complex breakpoints can be configured to match on more intricate scenarios. Complex break features include pass counters to enable the breakpoint after N matching occurrences, requiring matching of both data and instruction breaks on one instruction, priming to enable after another breakpoint condition has been met, and qualifying to enable instruction breaks when certain data conditions have been met.

An optional TAP, enabling communication between an EJTAG probe and the CPU through a dedicated port, may also be applied to the core. This provides the possibility for debugging without debug code in the application, and for download of application code to the system.

Another optional block is EJTAG Trace which enables real-time tracing capability. The trace information can be stored to either an on-chip trace memory or to an off-chip trace probe. The trace of program flow is highly flexible and can include instruction program counter as well as data addresses and data values. The trace features provides a powerful software debugging mechanism.

Refer to Chapter 8, “EJTAG Debug Support in the M4K™ Core” on page 127 for more information on the EJTAG features.

1.2.2.3 Coprocessor 2 Interface (CP2)

The optional coprocessor 2 (CP2) interface provides a full-featured interface for a coprocessor. It provides full support for all the MIPS32 COP2 instructions, with the exception of the 64-bit Load/Store instructions (LDC2/SDC2).

The CP2 interface can provide access to a graphics accelerator coprocessor or a simple register file. There is no support for the floating-point coprocessor COP1, which requires 64-bit data transfers.

Refer to Chapter 10, “M4K™ Processor Core Instructions” on page 207 for more information on the Coprocessor 2 supported instructions.

1.2.2.4 CorExtend® User Defined Instructions (UDI)

This optional module contains support for CorExtend user defined instructions. These instructions must be defined at build-time for the M4K core. Access to UDI requires a separate license from MIPS, and the core is then referred to as the M4K Pro™ core. When licensed, 16 instructions in the opcode map are available for UDI, and each instruction can have single or multi-cycle latency. A UDI 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 local register. Implementation details for UDI can be found in other documents available from MIPS.

Refer to Table 10.3 “Special2 Opcode Encoding of Function Field” for a specification of the opcode map available for user defined instructions.

Chapter 2

Pipeline of the M4K™ Core

The M4K processor core implements a 5-stage pipeline similar to the original R3000 pipeline. The pipeline allows the processor to achieve high frequency while minimizing device complexity, reducing both cost and power consumption. This chapter contains the following sections:

2.1 Pipeline Stages

The pipeline consists of five stages:

- Instruction (I stage)
- Execution (E stage)
- Memory (M stage)
- Align (A stage)
- Writeback (W stage)

A M4K core implements a “Bypass” mechanism that allows the result of an operation to be sent directly to the instruction that needs it without having to write the result to the register and then read it back.

The M4K soft core includes a build-time option that determines the type of multiply/divide unit (MDU) implemented. The MDU can be either a high-performance array or an iterative, area-efficient array. The MDU choice has a significant effect on the MDU pipeline, and the latency of multiply/divide instructions executed on the core. Software can query the type of MDU present on a specific implementation of the core by querying the MDU bit in the Config register (CP0 register 16, select 0); see 5.2.13 "Config Register (CP0 Register 16, Select 0)" for more details.

Figure 2.1 shows the operations performed in each pipeline stage of the M4K processor core, when the high-performance multiplier is present.

Figure 2.1 M4K™ Core Pipeline Stages (with high-performance MDU)

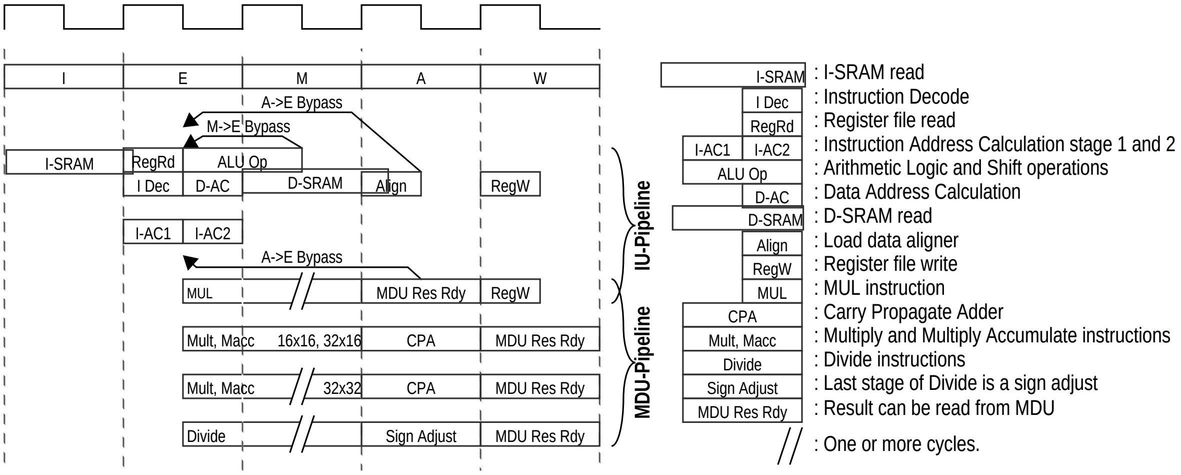
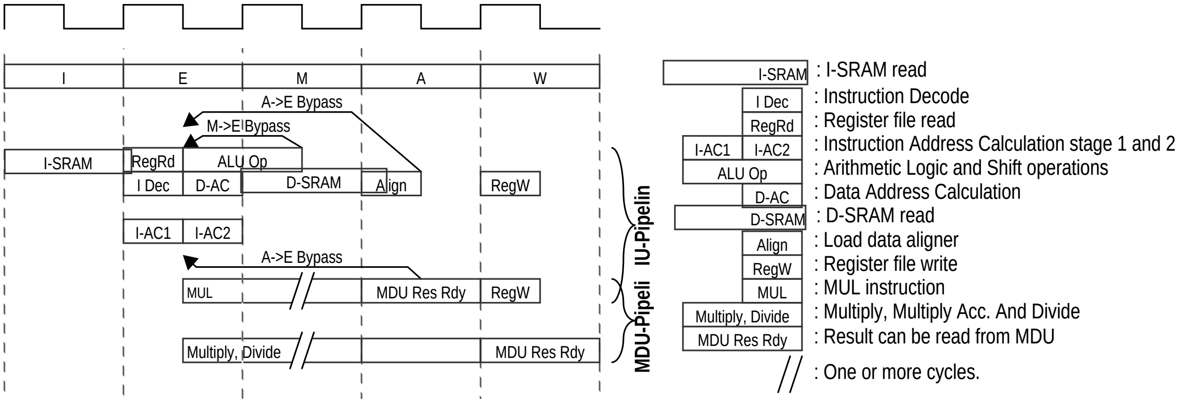

I
E
M
A
W
A->E Bypass
M->E Bypass
I-SRAM
RegRd
ALU Op
I Dec
D-AC
D-SRAM
Align
RegW
I-AC1
I-AC2
A->E Bypass
MUL
MDU Res Rdy
RegW
Mult, Macc
16x16, 32x16
CPA
MDU Res Rdy
Mult, Macc
32x32
CPA
MDU Res Rdy
Divide
Sign Adjust
MDU Res Rdy
IU-Pipeline
MDU-Pipeline
I-SRAM : I-SRAM read
I Dec : Instruction Decode
RegRd : Register file read
I-AC1 I-AC2 : Instruction Address Calculation stage 1 and 2
ALU Op : Arithmetic Logic and Shift operations
D-AC : Data Address Calculation
D-SRAM : D-SRAM read
Align : Load data aligner
RegW : Register file write
MUL : MUL instruction
CPA : Carry Propagate Adder
Mult, Macc : Multiply and Multiply Accumulate instructions
Divide : Divide instructions
Sign Adjust : Last stage of Divide is a sign adjust
MDU Res Rdy : Result can be read from MDU
// : One or more cycles.

Figure 2.2 shows the operations performed in each pipeline stage of the M4K processor core, when the area-efficient multiplier is present.

Figure 2.2 M4K™ Core Pipeline Stages (with area-efficient MDU)


I
E
M
A
W
A->E Bypass
M->E Bypass
I-SRAM
RegRd
ALU Op
I Dec
D-AC
D-SRAM
Align
RegW
I-AC1
I-AC2
A->E Bypass
MUL
MDU Res Rdy
RegW
Multiply, Divide
MDU Res Rdy
IU-Pipelin
MDU-Pipeli
I-SRAM : I-SRAM read
I Dec : Instruction Decode
RegRd : Register file read
I-AC1 I-AC2 : Instruction Address Calculation stage 1 and 2
ALU Op : Arithmetic Logic and Shift operations
D-AC : Data Address Calculation
D-SRAM : D-SRAM read
Align : Load data aligner
RegW : Register file write
MUL : MUL instruction
Multiply, Divide : Multiply, Multiply Acc. And Divide
MDU Res Rdy : Result can be read from MDU
// : One or more cycles.

2.1.1 I Stage: Instruction Fetch

During the Instruction fetch stage:

- An instruction is fetched from the instruction SRAM.
- MIPS16e instructions are converted into MIPS32-like instructions.

2.1.2 E Stage: Execution

During the Execution stage:

- Operands are fetched from the register file.
- Operands from the M and A stage are bypassed to this stage.
- The Arithmetic Logic Unit (ALU) begins the arithmetic or logical operation for register-to-register instructions.
- The ALU calculates the data virtual address for load and store instructions and the MMU performs the fixed virtual-to-physical address translation.
- The ALU determines whether the branch condition is true and calculates the virtual branch target address for branch instructions.
- Instruction logic selects an instruction address and the MMU performs the fixed virtual-to-physical address translation.
- All multiply divide operations begin in this stage.

2.1.3 M Stage: Memory Fetch

During the Memory Fetch stage:

- The arithmetic or logic ALU operation completes.
- The data SRAM access is performed for load and store instructions.
- A 16x16 or 32x16 MUL operation completes in the array and stalls for one clock in the M stage to complete the carry-propagate-add in the M stage (high-performance MDU option).
- A 32x32 MUL operation stalls for two clocks in the M stage to complete the second cycle of the array and the carry-propagate-add in the M stage (high-performance MDU option).
- A multiply operation stalls the MDU pipeline for 31 cycles in the M stage (area-efficient MDU option).
- Multiply and divide calculations proceed in the MDU. If the calculation completes before the IU moves the instruction past the M stage, then the MDU holds the result in a temporary register until the IU moves the instructions to the A stage (and it is consequently known that it won’t be killed).

2.1.4 A Stage: Align

During the Align stage:

- A separate aligner aligns loaded data with its word boundary.
- A MUL operation makes the result available for writeback. The actual register writeback is performed in the W stage.
- From this stage load data or a result from the MDU are available in the E stage for bypassing.

2.1.5 W Stage: Writeback

During the Writeback stage:

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

2.2 Multiply/Divide Operations

The M4K core implement the standard MIPS II™ multiply and divide instructions. Additionally, several new instructions were standardized in the MIPS32 architecture for enhanced performance.

The targeted multiply instruction, MUL, specifies that multiply results be placed in the general purpose register file instead of the HI/LO register pair. By avoiding the explicit MFLO instruction, required when using the LO register, and by supporting multiple destination registers, the throughput of multiply-intensive operations is increased.

Four instructions, multiply-add (MADD), multiply-add-unsigned (MADDU) multiply-subtract (MSUB), and multiply-subtract-unsigned (MSUBU), are used to perform the multiply-accumulate and multiply-subtract operations. The MADD/MADDU instruction multiplies two numbers and then adds the product to the current contents of the HI and LO registers. Similarly, the MSUB/MSUBU instruction multiplies two operands and then subtracts the product from the HI and LO registers. The MADD/MADDU and MSUB/MSUBU operations are commonly used in DSP algorithms.

All multiply operations (except the MUL instruction) write to the HI/LO register pair. All integer operations write to the general purpose registers (GPR). Because MDU operations write to different registers than integer operations, following integer instructions can execute before the MDU operation has completed. The MFLO and MFHI instructions are used to move data from the HI/LO register pair to the GPR file. If a MFLO or MFHI instruction is issued before the MDU operation completes, it will stall to wait for the data.

2.3 MDU Pipeline (High-Performance MDU)

The M4Kprocessor core contains an autonomous multiply/divide unit (MDU) with a separate pipeline for multiply and divide operations. This pipeline operates in parallel with the integer unit (IU) pipeline and does not stall when the IU pipeline stalls. This allows multi-cycle MDU operations, such as a divide, to be partially masked by system stalls and/or other integer unit instructions.

The MDU consists of a 32x16 booth encoded multiplier array, a carry propagate adder, result/accumulation registers (HI and LO), multiply and divide state machines, and all necessary multiplexers and control logic. The first number shown (‘32’ of 32x16) represents the *rs* operand. The second number (‘16’ of 32x16) represents the *rt* operand. The core only checks the latter *(rt)* operand value to determine how many times the operation must pass through the multiplier array. The 16x16 and 32x16 operations pass through the multiplier array once. A 32x32 operation passes through the multiplier array twice.

The MDU supports execution of a 16x16 or 32x16 multiply operation every clock cycle; 32x32 multiply operations can be issued every other clock cycle. Appropriate interlocks are implemented to stall the issue of back-to-back 32x32 multiply operations. Multiply operand size is automatically determined by logic built into the MDU. Divide operations are implemented with a simple 1 bit per clock iterative algorithm with an early in detection of sign extension on the dividend *(rs)*. Any attempt to issue a subsequent MDU instruction while a divide is still active causes an IU pipeline stall until the divide operation is completed.

Table 2.1 lists the latencies (number of cycles until a result is available) for multiply and divide instructions. The latencies are listed in terms of pipeline clocks. In this table ‘latency’ refers to the number of cycles necessary for the first instruction to produce the result needed by the second instruction.

Table 2.1 MDU Instruction Latencies (High-Performance MDU)

Size of Operand 1st Instruction[1]	Instruction Sequence		Latency Clocks
	1st Instruction	2nd Instruction	
16 bit	MULT/MULTU, MADD/MADDU MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU or MFHI/MFLO	1
32 bit	MULT/MULTU, MADD/MADDU, or MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU or MFHI/MFLO	2
16 bit	MUL	Integer operation[2]	2[3]
32 bit	MUL	Integer operation[2]	2[3]
8 bit	DIVU	MFHI/MFLO	9
16 bit	DIVU	MFHI/MFLO	17
24 bit	DIVU	MFHI/MFLO	25
32 bit	DIVU	MFHI/MFLO	33
8 bit	DIV	MFHI/MFLO	10[4]
16 bit	DIV	MFHI/MFLO	18[4]
24 bit	DIV	MFHI/MFLO	26[4]
32 bit	DIV	MFHI/MFLO	34[4]
any	MFHI/MFLO	Integer operation[2]	2
any	MTHI/MTLO	MADD/MADDU or MSUB/MSUBU	1

[1] For multiply operations, this is the *rt* operand. For divide operations, this is the *rs* operand.

[2] Integer Operation refers to any integer instruction that uses the result of a previous MDU operation.

[3] This does not include the 1 or 2 IU pipeline stalls (16 bit or 32 bit) that the MUL operation causes irrespective of the following instruction.These stalls do not add to the latency of 2.

[4] If both operands are positive, then the Sign Adjust stage is bypassed. Latency is then the same as for DIVU.

In Table 2.1 a latency of one means that the first and second instructions can be issued back to back in the code without the MDU causing any stalls in the IU pipeline. A latency of two means that if issued back to back, the IU pipeline will be stalled for one cycle. MUL operations are special because it needs to stall the IU pipeline in order to maintain its register file write slot. Consequently the MUL 16x16 or 32x16 operation will always force a one cycle stall of the IU pipeline, and the MUL 32x32 will force a two cycle stall. If the integer instruction immediately following the MUL operation uses its result, an additional stall is forced on the IU pipeline.

Table 2.2 lists the repeat rates (peak issue rate of cycles until the operation can be reissued) for multiply accumulate/subtract instructions. The repeat rates are listed in terms of pipeline clocks. In this table ‘repeat rate’ refers to the case where the first MDU instruction (in the table below) if back-to-back with the second instruction.

Table 2.2 MDU Instruction Repeat Rates (High-Performance MDU)

Operand Size of 1st Instruction	Instruction Sequence		Repeat Rate
	1st Instruction	2nd Instruction	
16 bit	MULT/MULTU, MADD/MADDU, MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU	1
32 bit	MULT/MULTU, MADD/MADDU, MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU	2

Figure 2.3 below shows the pipeline flow for the following sequence:

1. 32x16 multiply (Mult_1)

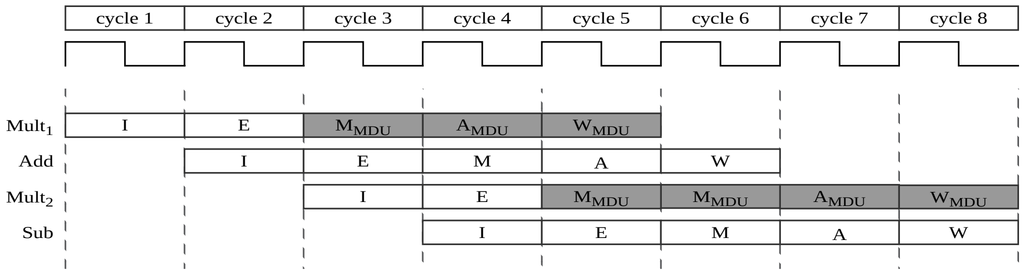
2. Add

3. 32x32 multiply (Mult_2)

4. Subtract (Sub)

The 32x16 multiply operation requires one clock of each pipeline stage to complete. The 32x32 multiply operation requires two clocks in the M_{MDU} pipe-stage. The MDU pipeline is shown as the shaded areas of Figure 2.3 and always starts a computation in the final phase of the E stage. As shown in the figure, the M_{MDU} pipe-stage of the MDU pipeline occurs in parallel with the M stage of the IU pipeline, the A_{MDU} stage occurs in parallel with the A stage, and the W_{MDU} stage occurs in parallel with the W stage. In general this need not be the case. Following the 1st cycle of the M stages, the two pipelines need not be synchronized. This does not present a problem because results in the MDU pipeline are written to the HI and LO registers, while the integer pipeline results are written to the register file.

Figure 2.3 MDU Pipeline Behavior During Multiply Operations



The following is a cycle-by-cycle analysis of Figure 2.3.

1. The first 32x16 multiply operation (Mult_1) is fetched from the instruction cache and enters the I stage.

2. An Add operation enters the I stage. The $Mult_1$ operation enters the E stage. The integer and MDU pipelines share the I and E pipeline stages. At the end of the E stage in cycle 2, the MDU pipeline starts processing the multiply operation ($Mult_1$).

3. In cycle 3 a 32x32 multiply operation ($Mult_2$) enters the I stage and is fetched from the instruction cache. Since the Add operation has not yet reached the M stage by cycle 3, there is no activity in the M stage of the integer pipeline at this time.

4. In cycle 4 the Subtract instruction enters I stage. The second multiply operation ($Mult_2$) enters the E stage. And the Add operation enters M stage of the integer pipe. Since the $Mult_1$ multiply is a 32x16 operation, only one clock is required for the M_{MDU} stage, hence the $Mult_1$ operation passes to the A_{MDU} stage of the MDU pipeline.

5. In cycle 5 the Subtract instruction enters E stage. The $Mult_2$ multiply enters the M_{MDU} stage. The Add operation enters the A stage of the integer pipeline. The $Mult_1$ operation completes and is written back in to the HI/LO register pair in the W_{MDU} stage.

6. Since a 32x32 multiply requires two passes through the multiplier, with each pass requiring one clock, the 32x32 $Mult_2$ remains in the M_{MDU} stage in cycle 6. The Sub instruction enters M stage in the integer pipeline. The Add operation completes and is written to the register file in the W stage of the integer pipeline.

7. The $Mult_2$ multiply operation progresses to the A_{MDU} stage, and the Sub instruction progress to the A stage.

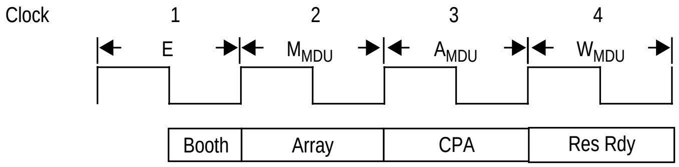
8. The $Mult_2$ operation completes and is written to the HI/LO registers pair the W_{MDU} stage, while the Sub instruction write to the register file in the W stage.

2.3.1 32x16 Multiply (High-Performance MDU)

The 32x16 multiply operation begins in the last phase of the E stage, which is shared between the integer and MDU pipelines. In the latter phase of the E stage, the *rs* and *rt* operands arrive and the booth-recoding function occurs at this time. The multiply calculation requires one clock and occurs in the M_{MDU} stage. In the A_{MDU} stage, the carry-propagate-add (CPA) function occurs and the operation is completed. The result is ready to be read from the HI/LO registers in the W_{MDU} stage.

Figure 2.4 shows a diagram of a 32x16 multiply operation.

Figure 2.4 MDU Pipeline Flow During a 32x16 Multiply Operation

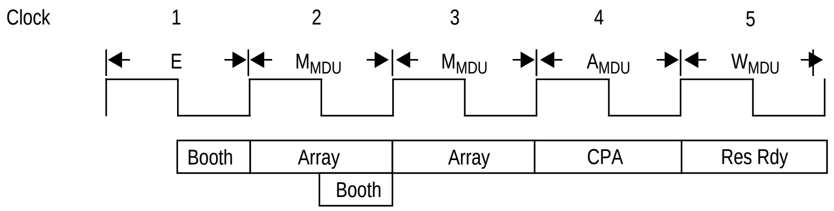


2.3.2 32x32 Multiply (High-Performance MDU)

The 32x32 multiply operation begins in the last phase of the E stage, which is shared between the integer and MDU pipelines. In the latter phase of the E stage, the *rs* and *rt* operands arrive and the booth recoding function occurs at this time. The multiply calculation requires two clocks and occurs in the M_{MDU} stage. In the A_{MDU} stage, the CPA function occurs and the operation is completed.

Figure 2.5 shows a diagram of a 32x32 multiply operation.

Figure 2.5 MDU Pipeline Flow During a 32x32 Multiply Operation



2.3.3 Divide (High-Performance MDU)

Divide operations are implemented using a simple non-restoring division algorithm. This algorithm works only for positive operands, hence the first cycle of the M_{MDU} stage is used to negate the *rs* operand (RS Adjust) if needed. Note that this cycle is spent even if the adjustment is not necessary. During the next maximum 32 cycles (3-34) an iterative add/subtract loop is executed. In cycle 3 an early-in detection is performed in parallel with the add/subtract. The adjusted *rs* operand is detected to be zero extended on the upper most 8, 16 or 24 bits. If this is the case the following 7, 15 or 23 cycles of the add/subtract iterations are skipped.

The remainder adjust (Rem Adjust) cycle is required if the remainder was negative. Note that this cycle is spent even if the remainder was positive. A sign adjust is performed on the quotient and/or remainder if necessary. The sign adjust stage is skipped if both operands are positive. In this case the Rem Adjust is moved to the A_{MDU} stage.

Figure 2.6, Figure 2.7, Figure 2.8 and Figure 2.9 show the latency for 8, 16, 24 and 32 bit divide operations, respectively. The repeat rate is either 11, 19, 27 or 35 cycles (one less if the *sign adjust* stage is skipped) as a second divide can be in the *RS Adjust* stage when the first divide is in the *Reg WR* stage.

Figure 2.6 High-Performance MDU Pipeline Flow During a 8-bit Divide (DIV) Operation

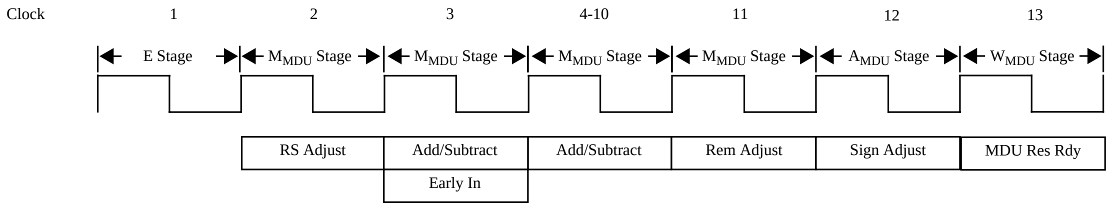


Figure 2.7 High-Performance MDU Pipeline Flow During a 16-bit Divide (DIV) Operation

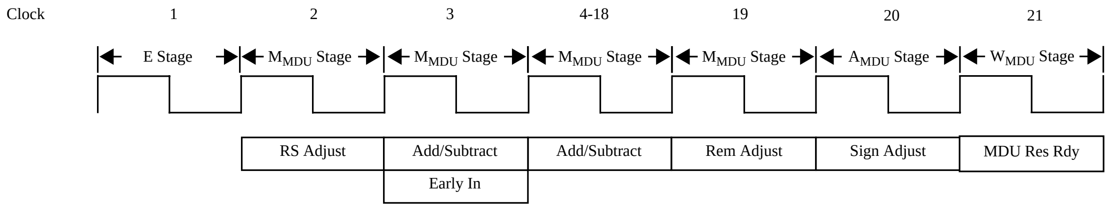


Figure 2.8 High-Performance MDU Pipeline Flow During a 24-bit Divide (DIV) Operation

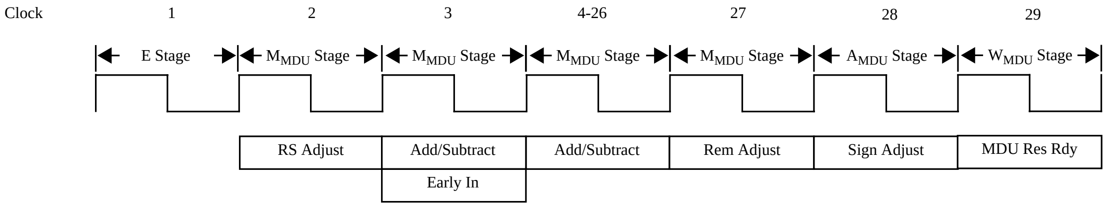
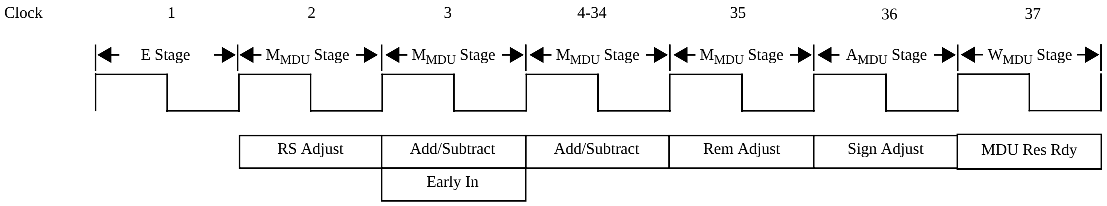


Figure 2.9 High-Performance MDU Pipeline Flow During a 32-bit Divide (DIV) Operation



2.4 MDU Pipeline (Area-Efficient MDU)

The area-efficient multiply/divide unit (MDU) is a separate autonomous block for multiply and divide operations. The MDU is not pipelined, but rather performs the computations iteratively in parallel with the integer unit (IU) pipeline. It does not stall when the IU pipeline stalls. This allows the long-running MDU operations to be partially masked by system stalls and/or other integer unit instructions.

The MDU consists of one 32-bit adder result-accumulate registers (HI and LO), a combined multiply/divide state machine and all multiplexers and control logic. A simple 1-bit per clock recursive algorithm is used for both multiply and divide operations. Using booth's algorithm all multiply operations complete in 32 clocks. Two extra clocks are needed for multiply-accumulate. The non-restoring algorithm used for divide operations will not work with negative numbers. Adjustment before and after are thus required depending on the sign of the operands. All divide operations complete in 33 to 35 clocks.

Table 2.3 lists the latencies (number of cycles until a result is available) for multiply and divide instructions. The latencies are listed in terms of pipeline clocks. In this table 'latency' refers to the number of cycles necessary for the second instruction to use the results of the first.

Table 2.3 M4K™ Core Instruction Latencies (Area-Efficient MDU)

Operand Signs of 1st Instruction (Rs,Rt)	Instruction Sequence		Latency Clocks
	1st Instruction	2nd Instruction	
any, any	MULT/MULTU	MADD/MADDU, MSUB/MSUBU, or MFHI/MFLO	32
any, any	MADD/MADDU, MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU, or MFHI/MFLO	34
any, any	MUL	Integer operation[1]	32
any, any	DIVU	MFHI/MFLO	33

Table 2.3 M4K™ Core Instruction Latencies (Area-Efficient MDU)

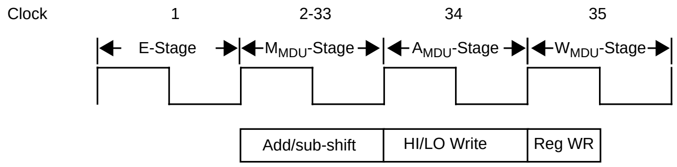
Operand Signs of 1st Instruction (Rs,Rt)	Instruction Sequence		Latency Clocks
	1st Instruction	2nd Instruction	
pos, pos	DIV	MFHI/MFLO	33
any, neg	DIV	MFHI/MFLO	34
neg, pos	DIV	MFHI/MFLO	35
any, any	MFHI/MFLO	Integer operation[1]	2
any, any	MTHI/MTLO	MADD/MADDU, MSUB/MSUBU	1
[1] Integer Operation refers to any integer instruction that uses the result of a previous MDU operation.			

2.4.1 Multiply (Area-Efficient MDU)

Multiply operations are executed using a simple iterative multiply algorithm. Using Booth’s approach, this algorithm works for both positive and negative operands. The operation uses 32 cycles in M_{MDU} stage to complete a multiplication. The register writeback to HI and LO are done in the A stage. For MUL operations, the register file writeback is done in the W_{MDU} stage.

Figure 2.10 shows the latency for a multiply operation. The repeat rate is 33 cycles as a second multiply can be in the first M_{MDU} stage when the first multiply is in A_{MDU} stage.

Figure 2.10 M4K™ Area-Efficient MDU Pipeline Flow During a Multiply Operation

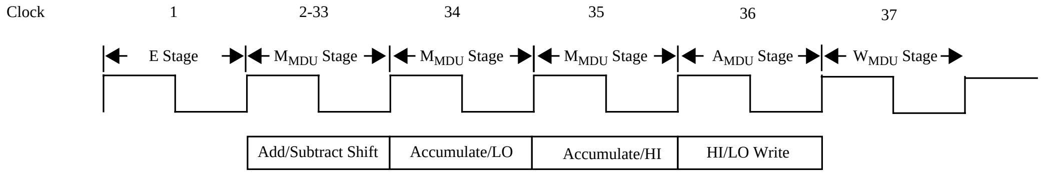


2.4.2 Multiply Accumulate (Area-Efficient MDU)

Multiply-accumulate operations use the same multiply machine as used for multiply only. Two extra stages are needed to perform the addition/subtraction. The operations uses 34 cycles in M_{MDU} stage to complete the multiply-accumulate. The register writeback to HI and LO are done in the A stage.

Figure 2.11 shows the latency for a multiply-accumulate operation. The repeat rate is 35 cycles as a second multiply-accumulate can be in the E stage when the first multiply is in the last M_{MDU} stage.

Figure 2.11 M4KC Area-Efficient MDU Pipeline Flow During a Multiply Accumulate Operation



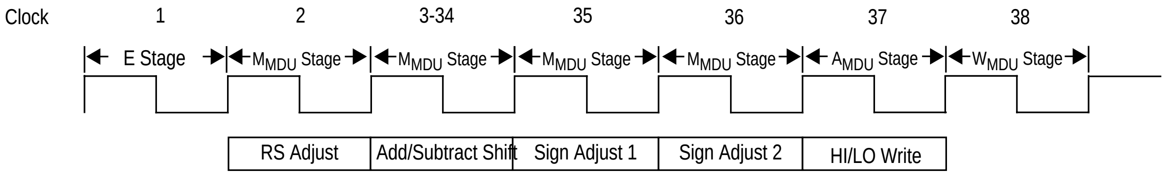
2.4.3 Divide (Area-Efficient MDU)

Divide operations also implement a simple non-restoring algorithm. This algorithm works only for positive operands, hence the first cycle of the M_{MDU} stage is used to negate the rs operand (RS Adjust) if needed. Note that this cycle is executed even if negation is not needed. The next 32 cycle (3-34) executes an interactive add/subtract-shift function.

Two sign adjust (Sign Adjust 1/2) cycles are used to change the sign of one or both the quotient and the remainder. Note that one or both of these cycles are skipped if they are not needed. The rule is, if both operands were positive or if this is an unsigned division; both of the sign adjust cycles are skipped. If the *rs* operand was negative, one of the sign adjust cycles is skipped. If only the *rs* operand was negative, none of the sign adjust cycles are skipped. Register writeback to HI and LO are done in the A stage.

Figure 2.12 shows the pipeline flow for a divide operation. The repeat rate is either 34, 35 or 36 cycles (depending on how many sign adjust cycles are skipped) as a second divide can be in the E stage when the first divide is in the last M_{MDU} stage.

Figure 2.12 M4K™ Area-Efficient MDU Pipeline Flow During a Divide (DIV) Operation



2.5 Branch Delay

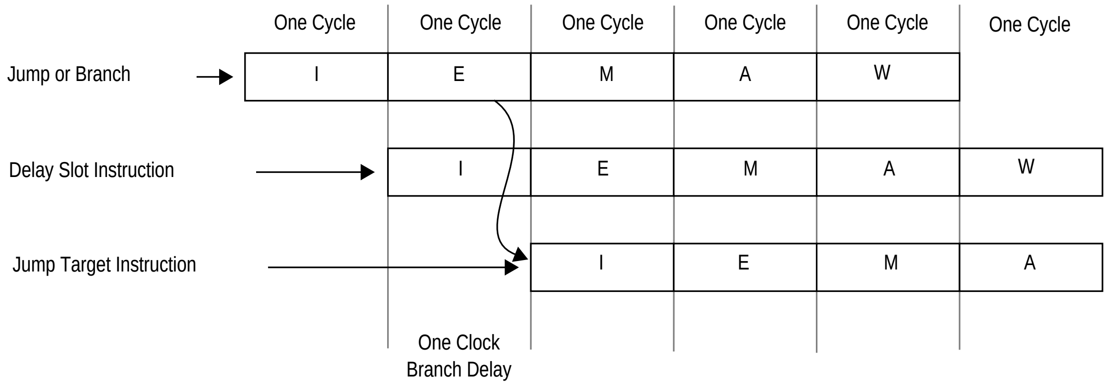
The pipeline has a branch delay of one cycle. The one-cycle branch delay is a result of the branch decision logic operating during the E pipeline stage. This allows the branch target address to be used in the I stage of the instruction following 2 cycles after the branch instruction. By executing the 1st instruction following the branch instruction sequentially before switching to the branch target, the intervening branch delay slot is utilized. This avoids bubbles being injected into the pipeline on branch instructions. Both the address calculation and the branch condition check are performed in the E stage.

The pipeline begins the fetch of either the branch path or the fall-through path in the cycle following the delay slot. After the branch decision is made, the processor continues with the fetch of either the branch path (for a taken branch) or the fall-through path (for the non-taken branch).

The branch delay means that the instruction immediately following a branch is always executed, regardless of the branch direction. If no useful instruction can be placed after the branch, then the compiler or assembler must insert a NOP instruction in the delay slot.

Figure 2.13 illustrates the branch delay.

Figure 2.13 IU Pipeline Branch Delay



2.6 Data Bypassing

Most MIPS32 instructions use one or two register values as source operands. These operands are fetched from the register file in the first part of E stage. The ALU straddles the E to M boundary, and can present the result early in M stage. The result is not written to the register file before the W stage however. If no precautions were made, it would take 3 cycles before the result was available for the following instructions. To avoid this, data bypassing is implemented.

Between the register file and the ALU a data bypass multiplexer is placed on both operands (see Figure 2.14). This enables the M4K core to forward data from a preceding instruction whose target is a source register of a following instruction. An M to E bypass and an A to E bypass feed the bypass multiplexers. A W to E bypass is not needed, as the register file is capable of making an internal bypass of Rd write data directly to the Rs and Rt read ports.

Figure 2.14 IU Pipeline Data bypass

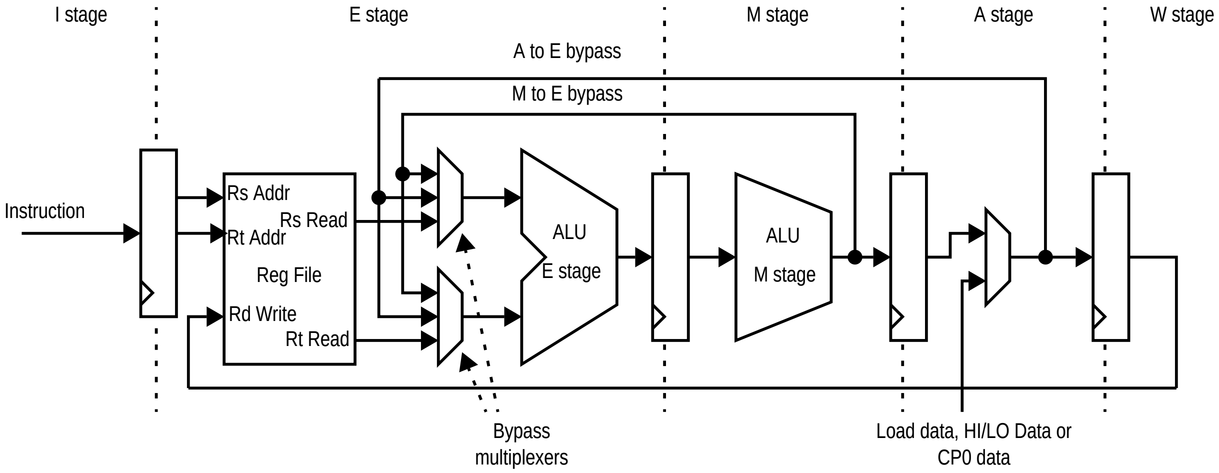
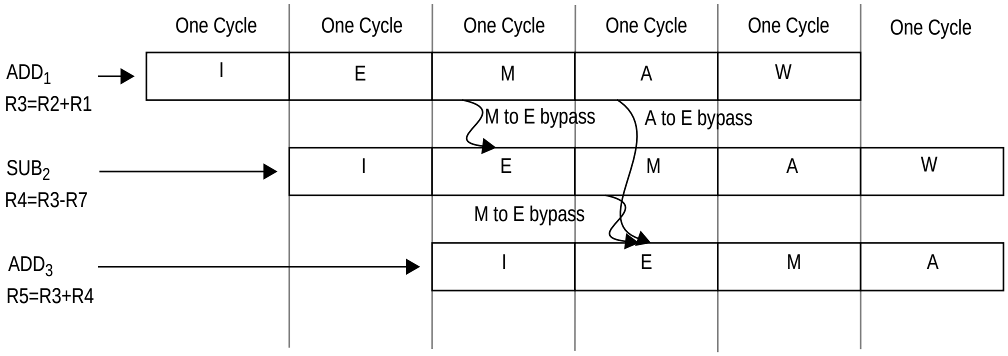


Figure 2.15 shows the data bypass for an Add$_1$ instruction followed by a Sub$_2$ and another Add$_3$ instruction. The Sub$_2$ instruction uses the output from the Add$_1$ instruction as one of the operands, and thus the M to E bypass is used. The following Add$_3$ uses the result from both the first Add$_1$ instruction and the Sub$_2$ instruction. Since the Add$_1$ data is

now in A stage, the A to E bypass is used, and the M to E bypass is used to bypass the Sub_2 data to the Add_2 instruction.

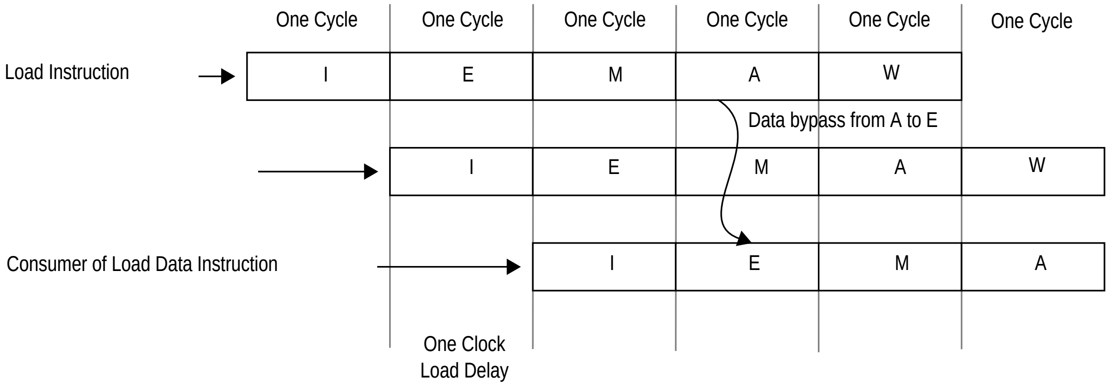
Figure 2.15 IU Pipeline M to E bypass



2.6.1 Load Delay

Load delay refers to the fact, that data fetched by a load instruction is not available in the integer pipeline until after the load aligner in A stage. All instructions need the source operands available in the E stage. An instruction immediately following a load instruction will, if it has the same source register as was the target of the load, cause an instruction interlock pipeline slip in the E stage (see 2.10 “Instruction Interlocks” on page 38). If an instruction following the load by 1 or 2 cycles uses the data from the load, the A to E bypass (see Figure 2.14) serves to reduce or avoid stall cycles. An instruction flow of this is shown in Figure 2.16.

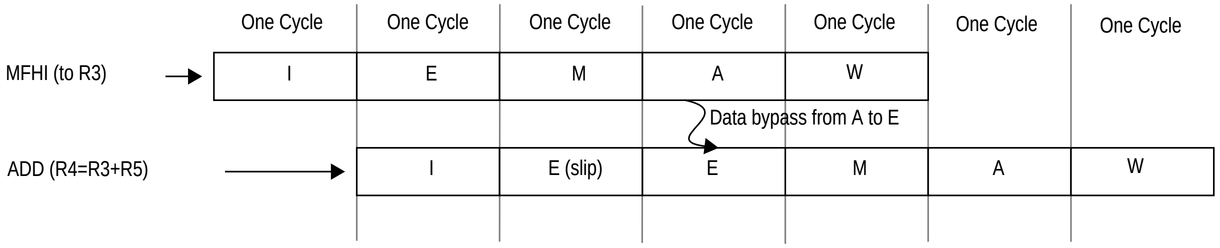
Figure 2.16 IU Pipeline A to E Data bypass



2.6.2 Move from HI/LO and CP0 Delay

As indicated in Figure 2.14, not only load data, but also data moved from the HI or LO registers (MFHI/MFLO) and data moved from CP0 (MFC0) enters the IU-Pipeline in the A stage. That is, data is not available in the integer pipeline until early in the A stage. The A to E bypass is available for this data. But as for Loads, an instruction following immediately after one of these move instructions must be paused for one cycle if the target of the move is among the sources of that following instruction. This then causes an interlock slip in the E stage (see 2.10 “Instruction Interlocks” on page 38). An interlock slip after a MFHI is illustrated in Figure 2.17.

Figure 2.17 IU Pipeline Slip after a MFHI



2.7 Coprocessor 2 Instructions

If a coprocessor 2 is attached to the M4K core, a number of transactions has to take place on the CP2 Interface, for each coprocessor 2 instruction. First of all if the CU[2] bit in the CP0 *Status* register is not set, then no coprocessor 2 related instruction will start a transaction on the CP2 Interface. Rather a Coprocessor Unusable exception will signaled. If the CU[2] bit is set, and a coprocessor 2 instruction is fetched, the following transactions will occur on the CP2 Interface:

1. The Instruction is presented on the instructions bus in E-stage. The coprocessor 2 can do a decode in the same cycle.

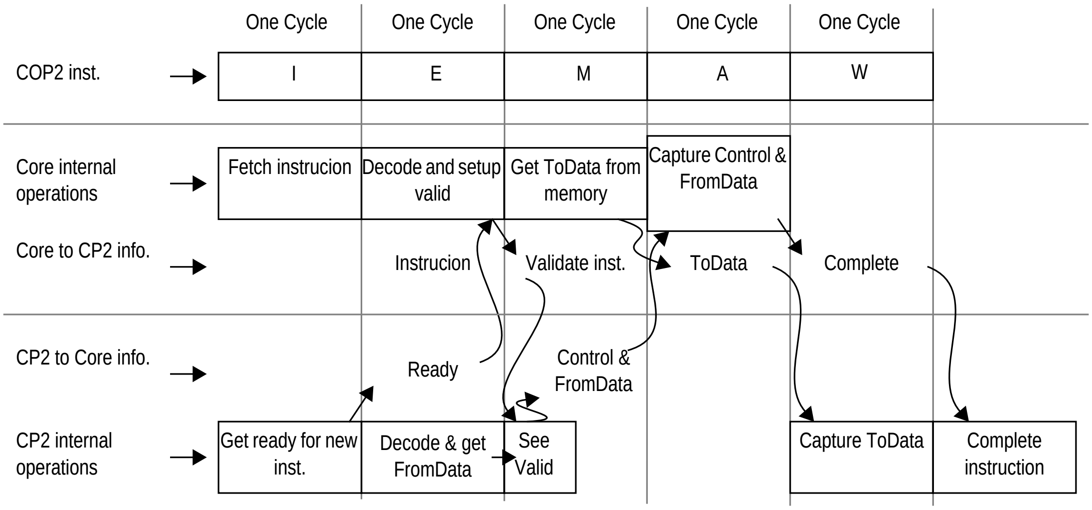
2. The Instruction is validated from the core in M-stage. From this point the core will accept control and data signals back from coprocessor 2. All control and data signals from the coprocessor 2 is captured on input latches to the core.

3. If all the expected control and data signals was presented to the core in the previous M-stage, the core will proceed executing the A-stage. If some return information is missing, the A-stage will not advance and cause a slip on all I, E and M-stage, see 2.9 "Slip Conditions" on page 38.
   If this instruction involved sending data from the core to the coprocessor 2, then this data is send in A-stage.

4. The instruction completion is signaled to the coprocessor 2 in the W-stage. Potential data from the coprocessor is written in the register file.

Figure 2.18 Show the timing relationship between the M4K core and the coprocessor 2 for all coprocessor 2 instruction.

Figure 2.18 Coprocessor 2 Interface Transactions



As can be seen all control and data from the coprocessor must occur in the M-stage. If this is not the case, the A-stage will start slipping in the following cycle, and thus stall the I, E, M and A pipeline stages; but if all expected control and data is available in the M-stage, a Coprocessor 2 instructions can execute with no stalls on the pipeline.

There is only one exception to this, and that is the Branch on Coprocessor conditions (BC2) instruction. All branch instructions, including the regular BEQ, BNE... etc. must be resolved in E-stage. The M4K core does not have branch prediction logic, and thus the target address must be available before the end of E-stage. The BC2 instruction has to follow the same protocol as all other coprocessor 2 instructions on the CP2 Interface. All core interface operations belonging to the E, M and A stages will have to occur in the E-stage for BC2 instructions. This means that a BC2 instructions always slips for a minimum of 2 cycles in E-stage. Any delay in return of branch information from the Coprocessor 2 will add to the number of slip cycles. All other Coprocessor 2 instructions can operate without slips, provided that all control and data information from the Coprocessor 2 is transferred in the M-stage.

2.8 Interlock Handling

Smooth pipeline flow is interrupted when cache misses occur or when data dependencies are detected. Interruptions handled entirely in hardware, such as cache misses, are referred to as *interlocks*. At each cycle, interlock conditions are checked for all active instructions.

Table 2.4 lists the types of pipeline interlocks for the M4K processor core.

Table 2.4 Pipeline Interlocks

Interlock Type	Sources	Slip Stage
I-side SRAM Stall	SRAM Access not complete	E Stage
Instruction	Producer-consumer hazards	E/M Stage
	Hardware Dependencies (MDU)	E Stage
	BC2 waiting for COP2 Condition Check	
D-side SRAM Stall	SRAM Access not complete	A Stage
Coprocessor 2 completion slip	Coprocessor 2 control and/or data delay from coprocessor	A Stage

In general, MIPS processors support two types of hardware interlocks:

- Stalls, which are resolved by halting the pipeline
- Slips, which allow one part of the pipeline to advance while another part of the pipeline is held static

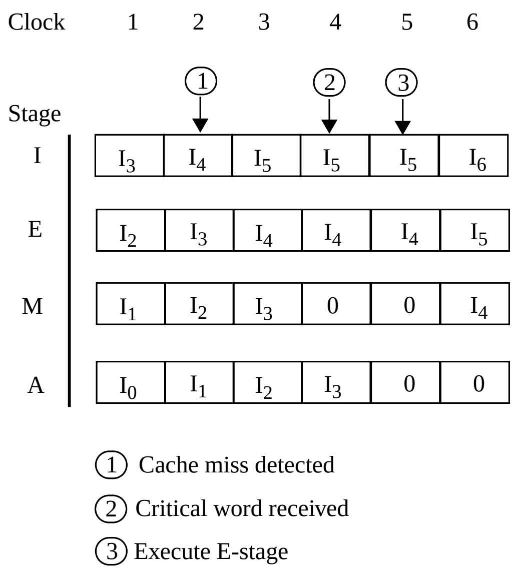
In the M4K processor core, all interlocks are handled as slips.

2.9 Slip Conditions

On every clock internal logic determines whether each pipe stage is allowed to advance. These slip conditions propagate backwards down the pipe. For example, if the M stage does not advance, neither does the E or I stage.

Slipped instructions are retried on subsequent cycles until they issue. The back end of the pipeline advances normally during slips. This resolves the conflict when the slip was caused by a missing result. NOPs are inserted into the bubble in the pipeline. Figure 2.19 shows an instruction cache miss.

Figure 2.19 Instruction Cache Miss Slip



1 Cache miss detected

2 Critical word received

3 Execute E-stage

Figure 2.19 shows a diagram of a two-cycle slip. In the first clock cycle, the pipeline is full and the cache miss is detected. Instruction I0 is in the A stage, instruction I1 is in the M stage, instruction I2 is in the E stage, and instruction I3 is in the I stage. The cache miss occurs in clock 2 when the I4 instruction fetch is attempted. I4 advances to the E-stage and waits for the instruction to be fetched from main memory. In this example it takes two clocks (3 and 4) to fetch the I4 instruction from memory. Once the cache miss is resolved in clock 4 and the instruction is bypassed to the E stage, the pipeline is restarted, causing the I4 instruction to finally execute it’s E-stage operations.

2.10 Instruction Interlocks

Most instructions can be issued at a rate of one per clock cycle. In order to adhere to the sequential programming model, the issue of an instruction must sometimes be delayed. This to ensure that the result of a prior instruction is available. Table 2.5 details the instruction interactions that prevent an instruction from advancing in the processor pipeline.

Table 2.5 Instruction Interlocks

Instruction Interlocks				
First Instruction		**Second Instruction**	**Issue Delay (in Clock Cycles)**	**Slip Stage**
LB/LBU/LH/LHU/LL/LW/LWL/LWR		Consumer of load data	1	E stage
MFC0		Consumer of destination register	1	E stage
MULTx/MADDx/MSUBx (high-performance MDU)	16bx32b	MFLO/MFHI	0	
	32bx32b		1	M stage
MUL (high-performance MDU)	16bx32b	Consumer of target data	2	E stage
	32bx32b		3	E stage
MUL (high-performance MDU)	16bx32b	Non-Consumer of target data	1	E stage
	32bx32b		2	E stage
MFHI/MFLO		Consumer of target data	1	E stage
MULTx/MADDx/MSUBx (high-performance MDU)	16bx32b	MULT/MUL/MADD/MSUB MTHI/MTLO/DIV	0[1]	E stage
	32bx32b		1[1]	E stage
DIV		MUL/MULTx/MADDx/ MSUBx/MTHI/MTLO/ MFHI/MFLO/DIV	Until DIV completes	E stage
MULT/MUL/MADD/MSUB/MTHI/MTLO/MFHI/MFLO/DIV (area-efficient MDU)		MULT/MUL/MADD/MSUB/ MTHI/MTLO/MFHI/MFLO/ DIV	Until 1st MDU op completes	E stage
MUL (area-efficient MDU)		Any Instruction	Until MUL completes	E stage
MFC0/MFC2/CFC2		Consumer of target data	1	E stage

2.11 Hazards

In general, the M4K 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 MIPS32® Architecture, hazards (primarily CP0 hazards) were relegated to implementation-dependent cycle-based solutions, primarily based on the SSNOP instruction. This has been an insufficient and error-prone practice that must be addressed with a firm compact between hardware and software. As such, new instructions have been added to Release 2 of the architecture which act as explicit barriers that eliminate hazards. To the extent that it was possible to do so, the new instructions have been added in such a way that they are backward-compatible with existing MIPS processors.

2.11.1 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.

2.11.1.1 Execution Hazards

Execution hazards are those created by the execution of one instruction, and seen by the execution of another instruction. Table 2.6 lists execution hazards.

Table 2.6 Execution Hazards

Producer	→	Consumer	Hazard On	Spacing (Instructions)
MTC0	→	Coprocessor instruction execution depends on the new value of Status_{CU}	Status_{CU}	1
MTC0	→	ERET	EPC DEPC ErrorEPC	1
MTC0	→	ERET	Status	0
MTC0, EI, DI	→	Interrupted Instruction	Status_{IE}	1
MTC0	→	Interrupted Instruction	Cause_{IP}	3
MTC0	→	RDPGPR WRPGPR	SRSCtl_{PSS}	1
MTC0	→	Instruction not seeing a Timer Interrupt	Compare update that clears Timer Interrupt	4[1]
MTC0	→	Instruction affected by change	Any other CP0 register	2

[1] This is the minimum value. Actual value is system-dependent since it is a function of the sequential logic between the *SI_TimerInt* output and the external logic which feeds *SI_TimerInt* back into one of the *SI_Int* inputs, or a function of the method for handling *SI_TimerInt* in an external interrupt controller.

2.11.1.2 Instruction Hazards

Instruction hazards are those created by the execution of one instruction, and seen by the instruction fetch of another instruction. Table 2.7 lists instruction hazards.

Table 2.7 Instruction Hazards

Producer	→	Consumer	Hazard On	Spacing (Instructions)
MTC0	→	Instruction fetch seeing the new value (including a change to ERL followed by an instruction fetch from the useg segment)	Status	

Table 2.7 Instruction Hazards (Continued)

Producer	→	Consumer	Hazard On	Spacing (Instructions)
Instruction stream write via redirected store	→	Instruction fetch seeing the new instruction stream	Cache entries	3

2.11.2 Instruction Listing

Table 2.8 lists the instructions designed to eliminate hazards. See the document titled *MIPS32® Architecture for Programmers Volume II: The MIPS32® Instruction Set* (MD00086) for a more detailed description of these instructions.

Table 2.8 Hazard Instruction Listing

Mnemonic	Function
EHB	Clear execution hazard
JALR.HB	Clear both execution and instruction hazards
JR.HB	Clear both execution and instruction hazards
SYNCI	Synchronize caches after instruction stream write

2.11.2.1 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 MIPS32 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.

2.11.3 Eliminating Hazards

The Spacing column shown in Table 2.6 and Table 2.7 indicates the number of unrelated instructions (such as NOPs or SSNOPs) that, prior to the capabilities of Release 2, would need to be placed between the producer and consumer of the hazard in order to ensure that the effects of the first instruction are seen by the second instruction. Entries in the table that are listed as 0 are traditional MIPS hazards which are not hazards on the M4K core.

With the hazard elimination instructions available in Release 2, the preferred method to eliminate hazards is to place one of the instructions listed in Table 2.8 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. Since the M4K core does not contain caches, the SYNCI instruction is not strictly necessary, but is still recommended to create portable code that can be run on other MIPS processors that may contain caches.

Chapter 3

Memory Management of the M4K™ Core

The M4K processor core includes a Memory Management Unit (MMU) that interfaces between the execution unit and the cache controller. The core implements a simple Fixed Mapping (FM) style MMU.

This chapter contains the following sections:

- Section 3.1 “Introduction”
- Section 3.2 “Modes of Operation”
- Section 3.3 “Fixed Mapping MMU”
- Section 3.4 “System Control Coprocessor”

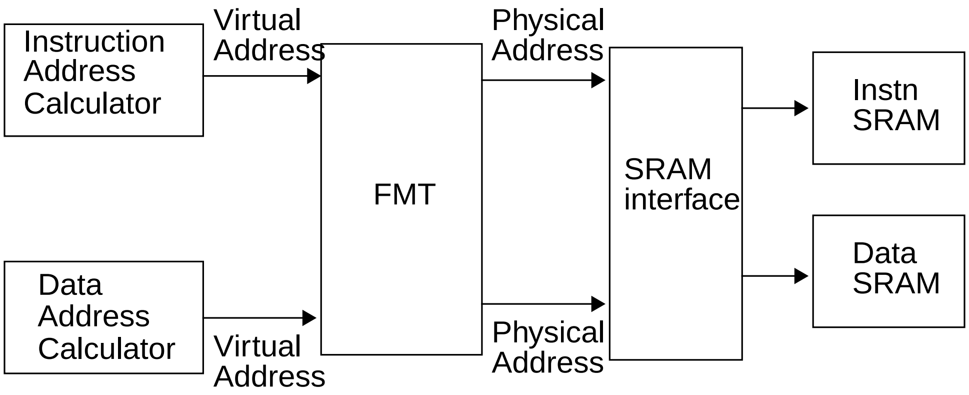
3.1 Introduction

The MMU will translate any virtual address to a physical address before a request is sent to the SRAM interface for an external memory reference.

In the M4K processor core, the MMU is based on a simple algorithm to translate virtual addresses into physical addresses via a Fixed Mapping (FM) mechanism. These translations are different for various regions of the virtual address space (useg/kuseg, kseg0, kseg1, kseg2/3).

Figure 3.1 shows how the memory management unit interacts with the SRAM access in the M4K core.

Figure 3.1 Address Translation During SRAM Access



3.2 Modes of Operation

A M4K processor core supports three modes of operation:

- User mode
- Kernel mode
- Debug mode

User mode is most often used for application programs. 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 most likely occurs within a software development tool.

The address translation performed by the MMU depends on the mode in which the processor is operating.

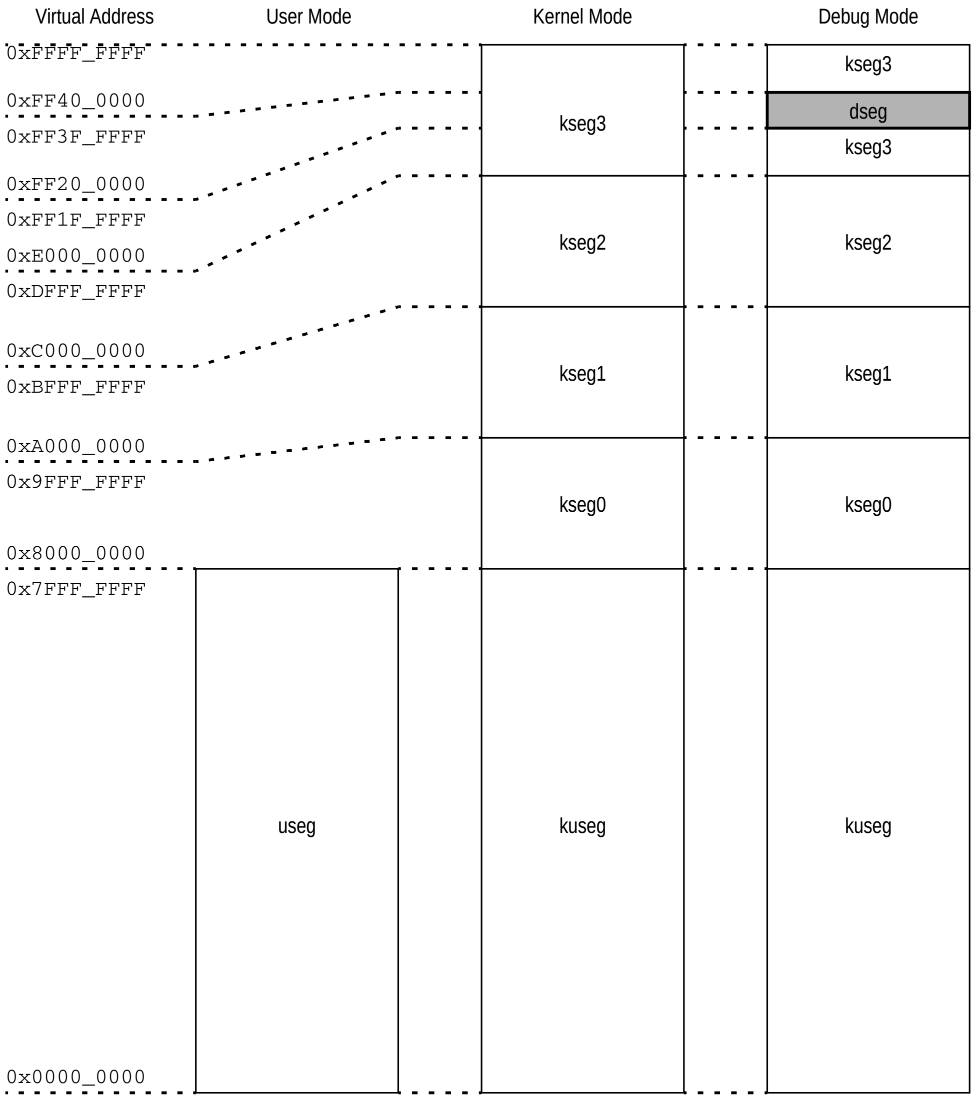
3.2.1 Virtual Memory Segments

The Virtual memory segments are different depending on the mode of operation. Figure 3.2 shows the segmentation for the 4 GByte (2^{32} bytes) virtual memory space addressed by a 32-bit virtual address, for the three modes of operation.

The core enters Kernel mode both at reset and when an exception is recognized. While in Kernel mode, software has access to the entire address space, as well as all CP0 registers. 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.

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 for 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. dseg access in Debug mode can be turned on or off, allowing full access to the entire kseg3 in Debug mode, if so desired.

Figure 3.2 M4K™ processor core Virtual Memory Map



Each of the segments shown in Figure 3.2 are either mapped or unmapped. The following two sub-sections explain the distinction. Then sections 3.2.2 "User Mode", 3.2.3 "Kernel Mode" and 3.2.4 "Debug Mode" specify which segments are actually mapped and unmapped.

3.2.1.1 Unmapped Segments

An unmapped segment does not use the FM to translate from virtual-to-physical addresses.

Unmapped segments have a fixed simple translation from virtual to physical address. This is much like the translations the FM provides for the M4K core, but we will still make the distinction.

All segments are treated as uncached within the M4K core. Cache coherency attributes of cached or uncached can be specified and this information will be sent with the request to allow the system to make a distinction between the two.

3.2.1.2 Mapped Segments

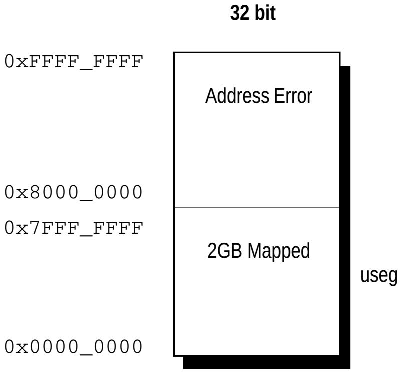
A mapped segment does use the FM to translate from virtual-to-physical addresses.

For the M4K core, the mapped segments have a fixed translation from virtual to physical address.The cacheability of the segment is defined in the CP0 register Config, fields K23 and KU (see 5.2.13 "Config Register (CP0 Register 16, Select 0)"). Write protection of segments is not possible during FM translation.

3.2.2 User Mode

In user mode, a single 2 GByte (2^{31} bytes) uniform virtual address space called the user segment (useg) is available. Figure 3.3 shows the location of user mode virtual address space.

Figure 3.3 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:

- UM = 1
- EXL = 0
- ERL = 0

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

Table 3.1 lists the characteristics of the useg User mode segments.

Table 3.1 User Mode Segments

Address Bit Value	Status Register			Segment Name	Address Range	Segment Size
	Bit Value					
	EXL	ERL	UM			
32-bit A(31) = 0	0	0	1	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. Any attempt to reference an address with the most significant bit set while in user mode causes an address error exception.

The system maps all references to *useg* through the FM.

3.2.3 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:

- UM = 0
- 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.

Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 3.4. Also, Table 3.2 lists the characteristics of the Kernel mode segments.

Figure 3.4 Kernel Mode Virtual Address Space

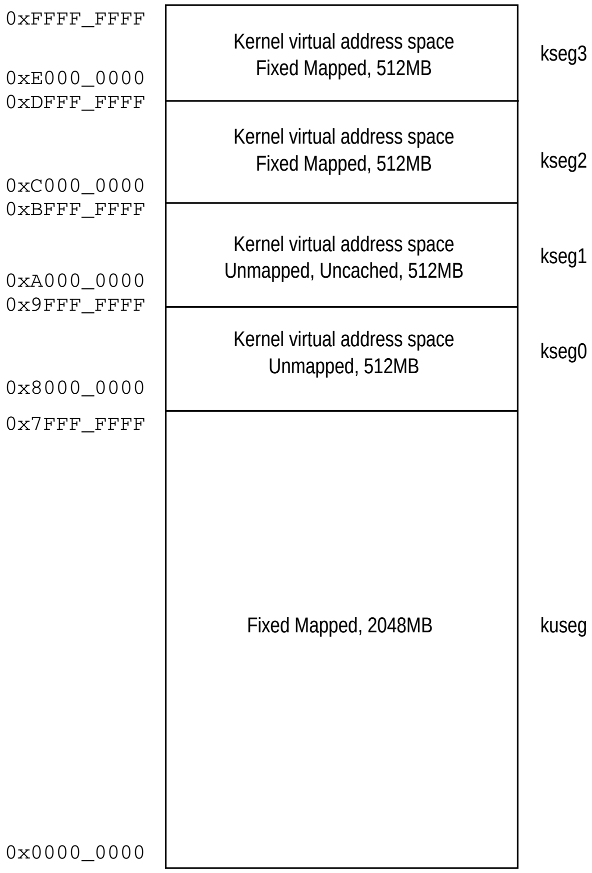


Table 3.2 Kernel Mode Segments

Address Bit Values	Status Register Is One of These Values			Segment Name	Address Range	Segment Size
	UM	EXL	ERL			
A(31) = 0	(UM = 0 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				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)

3.2.3.1 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 2^{31} bytes (2 GBytes) of the current user address space mapped to addresses 0x0000_0000 - 0x7FFF_FFFF.

When ERL = 1 in the *Status* register, the user address region becomes a 2^{31}-byte unmapped and uncached address space. While in this setting, the kuseg virtual address maps directly to the same physical address.

3.2.3.2 Kernel Mode, Kernel Space 0 (kseg0)

In Kernel mode, when the most-significant three bits of the virtual address are 100_2, 32-bit kseg0 virtual address space is selected; it is the 2^{29}-byte (512-MByte) kernel 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.

3.2.3.3 Kernel Mode, Kernel Space 1 (kseg1)

In Kernel mode, when the most-significant three bits of the 32-bit virtual address are 101_2, 32-bit kseg1 virtual address space is selected. kseg1 is the 2^{29}-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.

3.2.3.4 Kernel Mode, Kernel Space 2 (kseg2)

In Kernel mode, when UM = 0, 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. In the M4K core, this 2^{29}-byte (512-MByte) kernel virtual space is located at physical addresses 0xC000_0000 - 0xDFFF_FFFF..

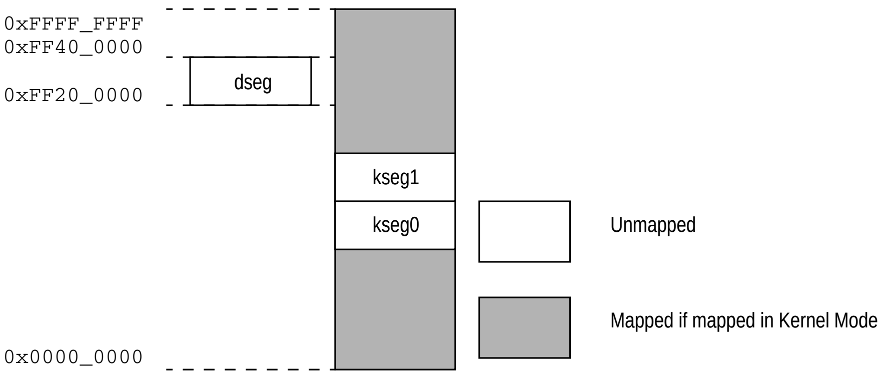
3.2.3.5 Kernel Mode, Kernel Space 3 (kseg3)

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. In the M4K core, this 2^{29}-byte (512-MByte) kernel virtual space is located at physical addresses 0xE000_0000 - 0xFFFF_FFFF.

3.2.4 Debug Mode

Debug mode address space is identical to Kernel mode address space with respect to mapped and unmapped areas, except for *kseg3*. In *kseg3*, a debug segment *dseg* co-exists in the virtual address range 0xFF20_0000 to 0xFF3F_FFFF. The layout is shown in Figure 3.5.

Figure 3.5 Debug Mode Virtual Address Space



The dseg is sub-divided into the dmseg segment at 0xFF20_0000 to 0xFF2F_FFFF which is used when the 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 for the segments are shown in Table 3.3.

Accesses to memory that would normally cause an exception if tried from kernel mode cause the core to re-enter debug mode via a debug mode exception.

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

Table 3.3 Physical Address and Cache Attributes for dseg, dmseg, and drseg Address Spaces

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. drseg maps to the breakpoint registers 0x0_0000 - 0xF_FFFF	Uncached
	drseg	0xFF30_0000 through 0xFF3F_FFFF		

3.2.4.1 Conditions and Behavior for Access to drseg, EJTAG Registers

The behavior of CPU access to the drseg address range at 0xFF30_0000 to 0xFF3F_FFFF is determined as shown in Table 3.4

Table 3.4 CPU Access to drseg Address Range

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 the drseg. Refer to Chapter 8, “EJTAG Debug Support in the M4K™ Core” on page 127 for more information on the DCR.

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.

3.2.4.2 Conditions and Behavior for Access to dmseg, EJTAG Memory

The behavior of CPU access to the dmseg address range at 0xFF20_0000 to 0xFF2F_FFFF is determined by the table shown in Table 3.5 .

Table 3.5 CPU Access to dmseg Address Range

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	
Fetch	0	Don’t care	See comments below
Load / Store	0	0	

The case with access to the dmseg when the ProbEn bit in the DCR register is 0 is not expected to happen. 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 happen, the reference hangs until it is satisfied by the probe. The probe can not assume that there will never be a reference to dmseg if 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.

3.3 Fixed Mapping MMU

The M4K core 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 in the M4K MMU.

The FM also determines the cacheability of each segment. These attributes are controlled via bits in the *Config* register. Table 3.6 shows the encoding for the K23 (bits 30:28), KU (bits 27:25) and K0 (bits 2:0) of the *Config* register. The M4K core does not contain caches and will treat all references as uncached, but these *Config* fields will be sent out to the system with the request and it can choose to use them to control any external caching that may be present.

Table 3.6 Cache Coherency Attributes

Config Register Fields K23, KU, and K0	Cache Coherency Attribute
2	Uncached.
3	Cacheable

In the M4K core, no translation exceptions can be taken, although address errors are still possible.

Table 3.7 Cacheability of Segments with Block Address Translation

Segment	Virtual Address Range	Cacheability
useg/kuseg	0x0000_0000- 0x7FFF_FFFF	Controlled by the KU field (bits 27:25) of the *Config* register. Refer to Table 3.6 for the encoding.
kseg0	0x8000_0000- 0x9FFF_FFFF	Controlled by the K0 field (bits 2:0) of the *Config* register. See Table 3.6 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 *Config* register. Refer to Table 3.6 for the encoding.
kseg3	0xE000_0000- 0xFFFF_FFFF	Controlled by K23 field (bits 30:28) of the *Config* register. Refer to Table 3.6 for the encoding.

The FM performs a simple translation to map from virtual addresses to physical addresses. This mapping is shown in Figure 3.6. When ERL=1, useg and kuseg become unmapped and uncached. The ERL behavior is the same as if there was a TLB. The ERL mapping is shown in Figure 3.7.

The ERL bit is usually never asserted by software. It is asserted by hardware after a Reset, SoftReset or NMI. See 4.8 "Exceptions" on page 72 for further information on exceptions.

Figure 3.6 FM Memory Map (ERL=0) in the M4K™ Processor Core

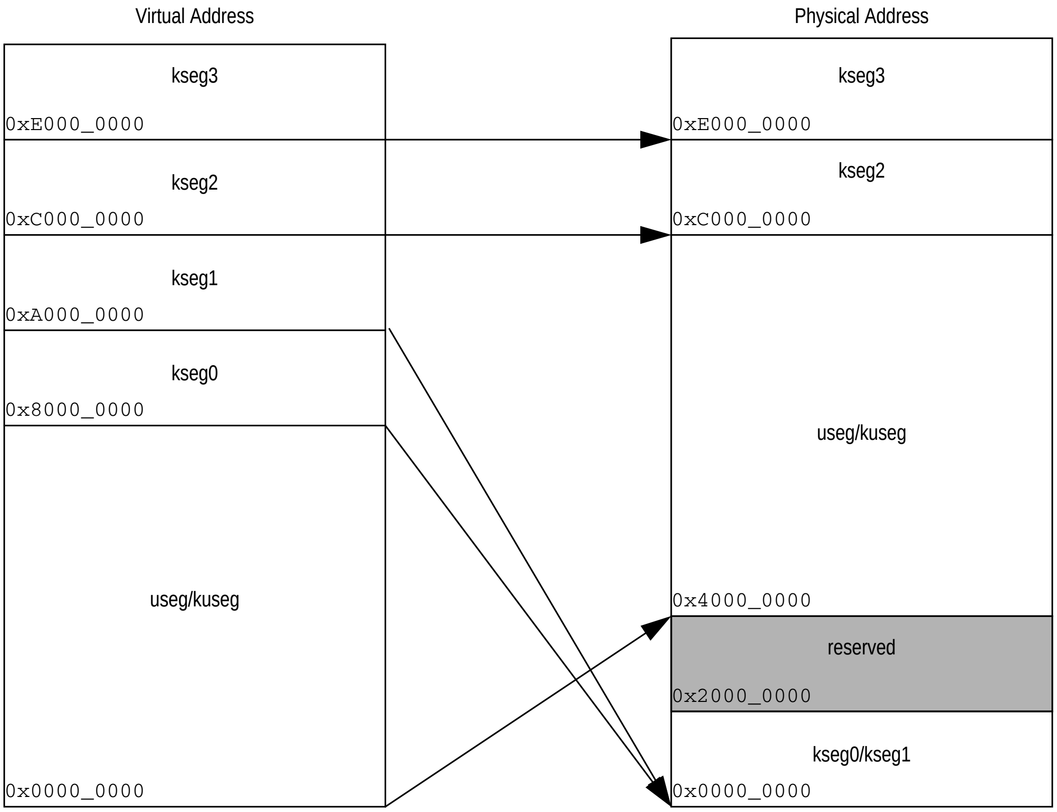
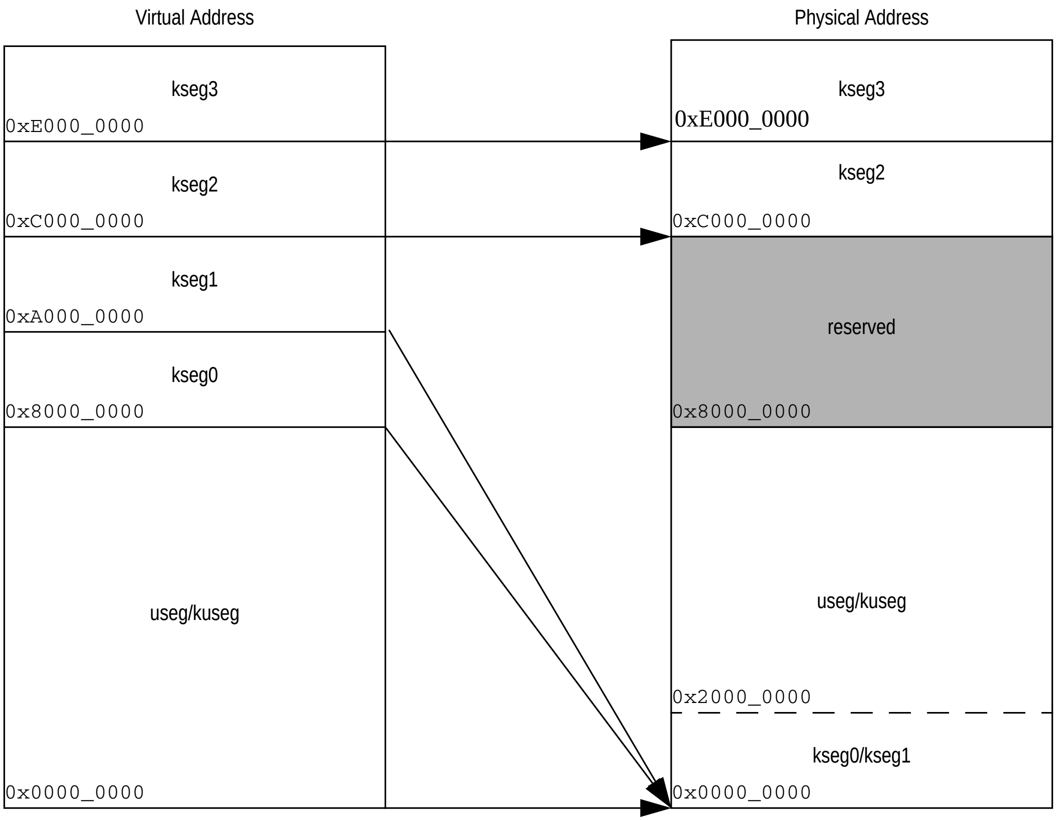


Figure 3.7 FM Memory Map (ERL=1) in the M4K™ Processor Core



3.4 System Control Coprocessor

The System Control Coprocessor (CP0) is implemented as an integral part of M4K processor core and supports memory management, address translation, exception handling, and other privileged operations. Certain CP0 registers are used to support memory management. Refer to Chapter 5, “CP0 Registers of the M4K™ Core” on page 85 for more information on the CP0 register set.

Chapter 4

Exceptions and Interrupts in the M4K™ Core

The M4K processor core receives exceptions from a number of sources, including arithmetic overflows, I/O interrupts, and system calls. When the CPU detects one of these exceptions, the normal sequence of instruction execution is suspended and the processor enters kernel mode.

In kernel mode the core disables interrupts and forces execution of a software exception processor (called a handler) located at a specific address. The 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.

When an exception occurs, the core loads the *Exception Program Counter* (*EPC*) register with a location where execution can restart after the exception has been serviced. Most exceptions are *precise,* which mean that *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, the address of the branch instruction immediately preceding the delay slot. To distinguish between the two, software must read the BD bit in the CP0 *Cause* register. Bus error exceptions and CP2 exceptions may be imprecise. For imprecise exceptions the instruction that caused the exception can not be identified.

This chapter contains the following sections:

4.1 Exception Conditions

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

When an exception condition is detected on an instruction fetch, the core aborts that instruction and all instructions that follow. When this instruction reaches the W 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.

This implementation allows all preceding instructions to complete execution and prevents all subsequent instructions from completing. Thus, the value in the *EPC* (*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.

4.2 Exception Priority

Table 4.1 lists all possible exceptions, and the relative priority of each, highest to lowest. Several of these exceptions can happen simultaneously, in that event the exception with the highest priority is the one taken.

Table 4.1 Priority of Exceptions

Exception	Description
Reset	Assertion of SI_ColdReset signal.
Soft Reset	Assertion of SI_Reset signal.
DSS	EJTAG Debug Single Step.
DINT	EJTAG Debug Interrupt. Caused by the assertion of the external EJ_DINT input, or by setting the EjtagBrk bit in the *ECR* register.
NMI	Asserting edge of SI_NMI signal.
Interrupt	Assertion of unmasked hardware or software interrupt signal.
DIB	EJTAG debug hardware instruction break matched.
AdEL	Fetch address alignment error. User mode fetch reference to kernel address.
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 with CorExtend disabled.
RI	Execution of a Reserved Instruction.
C2E	Execution of coprocessor 2 instruction which caused a general exception in the coprocessor.
IS1	Execution of coprocessor 2 instruction which caused an Implementation Specific exception 1 in the coprocessor.
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) or EJTAG Data Value Break on Store (address and value).
AdEL	Load address alignment error. User mode load reference to kernel address.

Table 4.1 Priority of Exceptions (Continued)

Exception	Description
AdES	Store address alignment error. User mode store to kernel address.
DBE	Load or store bus error.
DDBL	EJTAG data hardware breakpoint matched in load data compare.
CBrk	EJTAG complex breakpoint.

4.3 Interrupts

Older 32-bit cores available from MIPS that implemented Release 1 of the Architecture included support for two software interrupts, six hardware interrupts, and a special-purpose timer interrupt. (Note that the Architecture also defines a performance counter interrupt, but this is not implemented on the M4K core.) The timer interrupt was provided external to the core and typically combined with hardware interrupt 5 in an system-dependent manner. Interrupts were handled either through the general exception vector (offset 16#180) or the special interrupt vector (16#200), based on the value of Cause_{IV}. Software was required to prioritize interrupts as a function of the Cause_{IP} bits in the interrupt handler prologue.

Release 2 of the Architecture, implemented by the M4K core, adds an upward-compatible extension to the Release 1 interrupt architecture that supports vectored interrupts. In addition, Release 2 adds a new interrupt mode that supports the use of an external interrupt controller by changing the interrupt architecture.

4.3.1 Interrupt Modes

The M4K core includes support for three interrupt modes, as defined by Release 2 of the Architecture:

- Interrupt compatibility mode, which acts identically to that in 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. This mode is architecturally optional; but it is always present on the M4K core, so the VInt bit will always read as a 1 for the M4K core.

- External Interrupt Controller (EIC) mode, which redefines the way in which interrupts are handled to provide full support for an external interrupt controller handling prioritization and vectoring of interrupts. This presence of this mode denoted by the VEIC bit in the *Config3* register. Again, this mode is architecturally optional. On the M4K core, the VEIC bit is set externally by the static input, *SI_EICPresent*, to allow system logic to indicate the presence of an external interrupt controller.

The reset state of the processor is to interrupt compatibility mode such that a processor supporting Release 2 of the Architecture, like the M4K core, is fully compatible with implementations of Release 1 of the Architecture.

Table 4.2 shows the current interrupt mode of the processor as a function of the coprocessor 0 register fields that can affect the mode.

Table 4.2 Interrupt Modes

Status_{BEV}	Cause_{IV}	IntCtl_{VS}	Config3_{VINT}	Config3_{VEIC}	Interrupt Mode
1	x	x	x	x	Compatibly
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	Can't happen - IntCtl_{VS} can not be non-zero if neither Vectored Interrupt nor External Interrupt Controller mode is implemented.
"x" denotes don't care					

4.3.1.1 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 though exception vector offset 16#180 (if $\textit{Cause}_{IV}$ = 0) or vector offset 16#200 (if $\textit{Cause}_{IV}$ = 1). This mode is in effect if any of the following conditions are true:

- $\textit{Cause}_{IV}$ = 0
- $\textit{Status}_{BEV}$ = 1
- $\textit{IntCtl}_{VS}$ = 0, which would be the case if vectored interrupts are not implemented, or have been disabled.

A typical software handler for interrupt compatibility mode might look as follows:

```
/*
 * Assumptions:
 *  - CauseIV = 1 (if it were zero, the interrupt exception would have to
 *                be isolated from the general exception vector before getting
 *                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 IntCtl$_{VS}$ */
   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 simply 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 Status$_{IM}$ 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 interupt 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 simple return to the interrupted code.
 */
   eret                          /* Return to interrupted code */

NestedException:
/*
 * Nested exceptions typically require saving the EPC and Status registers,
 * 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 can not 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 */
   sw     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) */
        lw     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 */
```

4.3.1.2 Vectored Interrupt Mode

Vectored Interrupt mode builds on the interrupt compatibility mode by adding a priority encoder to prioritize pending interrupts and to generate a vector with which each interrupt can be directed to a dedicated handler routine. This mode also allows each interrupt to be mapped to a GPR shadow set for use by the interrupt handler. Vectored Interrupt mode is in effect if all of the following conditions are true:

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

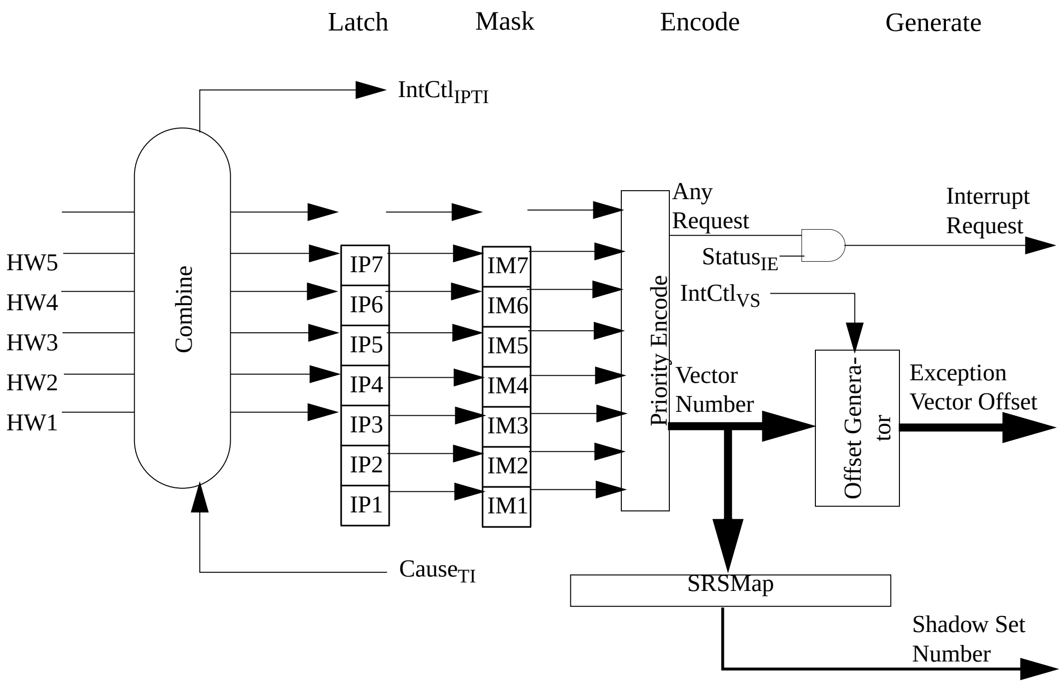
In VI interrupt mode, the six hardware interrupts are interpreted as individual hardware interrupt requests. The timer interrupt is combined in a system-dependent way (external to the core) with the hardware interrupts (the interrupt with which they are combined is indicated by the $IntCtl_{IPTI}$ field) to provide the appropriate relative priority of the timer interrupt 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 4.3.

Table 4.3 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
	Software	SW1	IP1 and IM1	1
Lowest Priority		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 4.1.

Figure 4.1 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 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   k0, C0_EPC          /* Get restart address */
   sw     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) */
   lw     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 */
```

4.3.1.3 External Interrupt Controller Mode

External Internal Interrupt Controller 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, and performance counter interrupts, and directly supplying to the processor the priority level and 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 interrupt mode, the processor sends the state of the software interrupt requests ($Cause_{IP1..IP0}$) and the timer interrupt request ($Cause_{TI}$) to the external interrupt controller, where it prioritizes these interrupts in a system-dependent way with other hardware interrupts. The interrupt controller can be a hard-wired logic block, or it can be configurable based on 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.

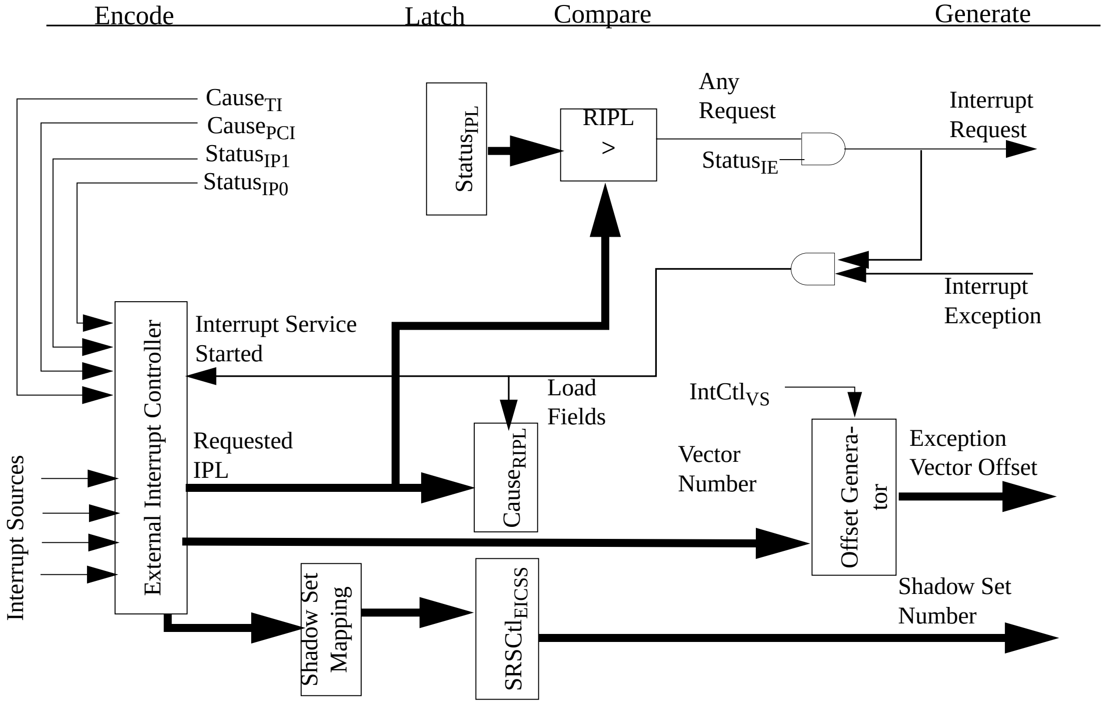
The external interrupt controller prioritizes its interrupt requests and produces the priority level and vector number of the highest priority interrupt to be serviced. The priority level, called the Requested Interrupt Priority Level (RIPL), is a 6-bit encoded value in the range 0..63, inclusive. A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

$\text{Status}_{\text{IPL}}$ (which overlays $Status_{IM7..IM2}$) is interpreted as the Interrupt Priority Level (IPL) at which the processor is currently operating (with a value of zero indicating 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 higher priority than the current IPL. If RIPL is strictly greater than $Status_{IPL}$, and interrupts are enabled ($\text{Status}_{\text{IE}} = 1$, $Status_{EXL} = 0$, 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 $Cause_{IP7..\text{IP2}}$) and signals the external interrupt controller to notify it that the request is being serviced. Because $Cause_{RIPL}$ is only loaded by the processor when an interrupt exception is signaled, it is available to software during interrupt processing. The vector number that the EIC passes to the core is combined with the $IntCtl_{VS}$ to determine where the interrupt service routine is located. The vector number is not stored in any software-visible registers.

In EIC interrupt mode, the external interrupt controller is also responsible for supplying the GPR shadow 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 4.2.

Figure 4.2 Interrupt Generation for External Interrupt Controller Interrupt Mode



A typical software handler for EIC 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, 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 *Cause*$_{RIPL}$ to *Status*$_{IPL}$ to prevent lower priority interrupts from interrupting the handler. 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 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.
 */
```

4.3.2 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 $IntCtl_{VS}$ to create the interrupt offset, which is added to 16#200 to create the exception vector offset. For VI interrupt mode, the vector number is in the range 0..7, inclusive. For EIC interrupt mode, the vector number is in the range 0..63, inclusive. The $IntCtl_{VS}$ field specifies the spacing between vector locations. If this value is zero (the default reset state), the vector spacing is zero and the processor reverts to Interrupt Compatibility Mode. A non-zero value enables vectored interrupts, and Table 4.4 shows the exception vector offset for a representative subset of the vector numbers and values of the $IntCtl_{VS}$ field.

Table 4.4 Exception Vector Offsets for Vectored Interrupts

Vector Number	Value of IntCtl$_{VS}$ Field				
	2#00001	**2#00010**	**2#00100**	**2#01000**	**2#10000**
0	16#0200	16#0200	16#0200	16#0200	16#0200
1	16#0220	16#0240	16#0280	16#0300	16#0400
2	16#0240	16#0280	16#0300	16#0400	16#0600
3	16#0260	16#02C0	16#0380	16#0500	16#0800
4	16#0280	16#0300	16#0400	16#0600	16#0A00
5	16#02A0	16#0340	16#0480	16#0700	16#0C00
6	16#02C0	16#0380	16#0500	16#0800	16#0E00
7	16#02E0	16#03C0	16#0580	16#0900	16#1000
• • •					
61	16#09A0	16#1140	16#2080	16#3F00	16#7C00
62	16#09C0	16#1180	16#2100	16#4000	16#7E00
63	16#09E0	16#11C0	16#2180	16#4100	16#8000

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

```
vectorOffset ← 16#200 + (vectorNumber × (IntCtl_VS ‖ 2#00000))
```

When using large vector spacing and EIC mode, the offset value can overlap with bits that are specified in the EBase register. Software must ensure that any overlapping bits are specified as 0 in EBase. This implementation ORs together the offset and base registers, but it is architecturally undefined and software should not rely on this behavior.

4.4 GPR Shadow Registers

Release 2 of the Architecture optionally removes the need to save and restore GPRs on entry to high priority interrupts or 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 allowing privileged software to associate a shadow set with entry to kernel mode via an interrupt vector or exception. The normal GPRs are logically considered shadow set zero.

The number of GPR shadow sets is a build-time option on the M4K core. Although Release 2 of the Architecture defines a maximum of 16 shadow sets, the core allows one (the normal GPRs), two, four, or eight 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 work 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 (that which was current before the last exception or interrupt occurred).

If the processor is operating in VI interrupt 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-vectored 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 is true. In this case, steps 2 and 3 are skipped.

   - The exception is one that sets $Status_{ERL}$: Reset, Soft 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-vectored 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 is 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 initialize ($Status_{BEV}$ = 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.

4.5 Exception Vector Locations

The Reset, Soft Reset, and NMI exceptions are always vectored to location `16#BFC0.0000`. EJTAG Debug exceptions are vectored to location `16#BFC0.0480`, or to location `16#FF20.0200` if the ProbTrap bit is zero or one, respectively, in the *EJTAG_Control_register*. Addresses for all other exceptions are a combination of a vector offset and a vector base address. In Release 1 of the architecture, the vector base address was fixed. In Release 2 of the architecture, software is allowed to specify the vector base address via the *EBase* register for exceptions that occur when $Status_{BEV}$ equals 0. Table 4.5 gives the vector base address as a function of the exception and whether the *BEV* bit is set in the *Status* register. Table 4.6 gives 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. For implementations of Release 2 of the Architecture, Table 4.4 gives the offset from the base address in the case where $Status_{BEV}$ = 0 and $Cause_{IV}$ = 1. For implementations of Release 1 of the architecture in which $Cause_{IV}$ = 1, the vector offset is as if $IntCtl_{VS}$ were 0. Table 4.7 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, the vector address value assumes that the *EBase* register, as implemented in Release 2 devices, is not changed from its reset state and that $IntCtl_{VS}$ is 0.

Table 4.5 Exception Vector Base Addresses

Exception	$Status_{BEV}$	
	0	1
Reset, Soft Reset, NMI	`16#BFC0.0000`	

Table 4.5 Exception Vector Base Addresses

Exception	Status$_{BEV}$	
	0	1
EJTAG Debug (with ProbEn = 0 in the EJTAG_Control_register)	`16#BFC0.0480`	
EJTAG Debug (with ProbEn = 1 in the EJTAG_Control_register)	`16#FF20.0200`	
Other	*For Release 1 of the architecture:* `16#8000.0000` *For Release 2 of the architecture:* EBase$_{31..12}$ || `16#000` Note that EBase$_{31..30}$ have the fixed value `2#10`	`16#BFC0.0200`

Table 4.6 Exception Vector Offsets

Exception	Vector Offset
General Exception	`16#180`
Interrupt, Cause$_{IV}$ = 1	`16#200` (In Release 2 implementations, this is the base of the vectored interrupt table when Status$_{BEV}$ = 0)
Reset, Soft Reset, NMI	None (Uses Reset Base Address)

Table 4.7 Exception Vectors

Exception	Status$_{BEV}$	Status$_{EXL}$	Cause$_{IV}$	EJTAG ProbEn	Vector For Release 2 Implementations, assumes that EBase retains its reset state and that IntCtl$_{VS}$ = 0
Reset, Soft Reset, NMI	x	x	x	x	`16#BFC0.0000`
EJTAG Debug	x	x	x	0	`16#BFC0.0480`
EJTAG Debug	x	x	x	1	`16#FF20.0200`
Interrupt	0	0	0	x	`16#8000.0180`
Interrupt	0	0	1	x	`16#8000.0200`
Interrupt	1	0	0	x	`16#BFC0.0380`
Interrupt	1	0	1	x	`16#BFC0.0400`
All others	0	x	x	x	`16#8000.0180`
All others	1	x	x	x	`16#BFC0.0380`
'x' denotes don't care					

4.6 General Exception Processing

With the exception of Reset, Soft 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 5.12). The value loaded into the *EPC* register is dependent on whether the processor implements the MIPS16e ASE, and whether the instruction is in the delay slot of a branch or jump which has delay slots. Table 4.8 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 4.8 Value Stored in EPC, ErrorEPC, or DEPC on an Exception

MIPS16e 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 MIPS16e 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 is started at the exception vector.

The value loaded into EPC represents the restart address for the exception and need not be modified by exception handler software in the normal case. Software need not look at the BD bit in the Cause register unless 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 ← 16#180
else
   if InstructionInBranchDelaySlot then
      EPC ← restartPC/* PC of branch/jump */
      Cause_BD ← 1
   else
      EPC ← restartPC               /* PC of instruction */
      Cause_BD ← 0
   endif

   /* Compute vector offsets as a function of the type of exception */
   NewShadowSet ← SRSCtl_ESS         /* Assume exception, Release 2 only */
   if ExceptionType = TLBRefill then
```

```
      vectorOffset ← 16#000
   elseif (ExceptionType = Interrupt) then
      if ($Cause_{IV}$ = 0) then
         vectorOffset ← 16#180
      else
         if ($Status_{BEV}$ = 1) or ($IntCtl_{VS}$ = 0) then
            vectorOffset ← 16#200
         else
            if $Config3_{VEIC}$ = 1 then
               VecNum ← $Cause_{RIPL}$
               NewShadowSet ← $SRSCtl_{EICSS}$
            else
               VecNum ← VIntPriorityEncoder()
               NewShadowSet ← $SRSMap_{IPL \times 4+3..IPL \times 4}$
            endif
            vectorOffset ← 16#200 + (VecNum × ($IntCtl_{VS}$ ‖ 2#00000))
         endif /* if ($Status_{BEV}$ = 1) or ($IntCtl_{VS}$ = 0) then */
      endif /* if ($Cause_{IV}$ = 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 ($SRSCtl_{HSS}$ > 0) and ($Status_{BEV}$ = 0) and
      ($Status_{ERL}$ = 0)) then
      $SRSCtl_{PSS}$ ← $SRSCtl_{CSS}$
      $SRSCtl_{CSS}$ ← NewShadowSet
   endif
endif /* if $Status_{EXL}$ = 1 then */

$Cause_{CE}$ ← FaultingCoprocessorNumber
$Cause_{ExcCode}$ ← ExceptionType
$Status_{EXL}$ ← 1


/* Calculate the vector base address */
if $Status_{BEV}$ = 1 then
   vectorBase ← 16#BFC0.0200
else
   if ArchitectureRevision ≥ 2 then
      /* The fixed value of $EBase_{31..30}$ forces the base to be in kseg0 or kseg1 */
      vectorBase ← $EBase_{31..12}$ ‖ 16#000
   else
      vectorBase ← 16#8000.0000
   endif
endif

/* Exception PC is the sum of vectorBase and vectorOffset */
PC ← $vectorBase_{31..30}$ ‖ ($vectorBase_{29..0}$ + $vectorOffset_{29..0}$)
                              /* No carry between bits 29 and 30 */
```

4.7 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*, *DINT*, *DIBImpr*, *DDBLImpr*, and *DDBSImpr* bits in the *Debug* register are updated appropriately depending on the debug exception type.
- The *Debug2* register is updated with additional information for complex breakpoints.
- *Halt* and *Doze* bits in the *Debug* register are updated appropriately.
- 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*, *DINT*, *DIBImpr*, *DDBLImpr*, and *DDBSImpr* bits in the *Debug* register.

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

Operation:

```
if InstructionInBranchDelaySlot then
   DEPC ← PC-4
   $Debug_{DBD}$ ← 1
else
   DEPC ← PC
   $Debug_{DBD}$ ← 0
endif
$Debug_{D* bits}$ ← DebugExceptionType
$Debug_{Halt}$ ← HaltStatusAtDebugException
$Debug_{Doze}$ ← DozeStatusAtDebugException
$Debug_{DM}$ ← 1
if $EJTAGControlRegister_{ProbTrap}$ = 1 then
   PC ← 0xFF20_0200
else
   PC ← 0xBFC0_0480
endif
```

The same debug exception vector location is used for all debug exceptions. The location is determined by the ProbTrap bit in the EJTAG Control register (ECR), as shown in Table 4.9.

Table 4.9 Debug Exception Vector Addresses

ProbTrap bit in ECR Register	Debug Exception Vector Address
0	0xBFC0_0480
1	0xFF20_0200 in dmseg

4.8 Exceptions

The following subsections describe each of the exceptions listed in the same sequence as shown in Table 4.1.

4.8.1 Reset/SoftReset Exception

A reset exception occurs when the *SI_ColdReset* signal is asserted to the processor. A soft reset occurs when the SI_Reset signals is asserted. These exception is not maskable. When one of these exceptions 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/SoftReset exception, the state of the processor is not defined, with the following exceptions:

- 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 *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:

```
Config ← ConfigurationState
$Status_{RP}$ ← 0
$Status_{BEV}$ ← 1
$Status_{TS}$ ← 0
$Status_{SR}$ ← 0/1 (depending on Reset or SoftReset)
$Status_{NMI}$ ← 0
$Status_{ERL}$ ← 1
if InstructionInBranchDelaySlot then
    ErrorEPC ← PC - 4
else
    ErrorEPC ← PC
endif
PC ← 0xBFC0_0000
```

4.8.2 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 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 the 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 the 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

4.8.3 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

4.8.4 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:

```
$\text{Status}_{\text{BEV}}$ ← 1
$\text{Status}_{\text{TS}}$ ← 0
$\text{Status}_{\text{SR}}$ ← 0
$\text{Status}_{\text{NMI}}$ ← 1
$\text{Status}_{\text{ERL}}$ ← 1
if InstructionInBranchDelaySlot then
    ErrorEPC ← PC - 4
else
    ErrorEPC ← PC
endif
PC ← 0xBFC0_0000
```

4.8.5 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 4.3 “Interrupts” on page 57 for more details about the processing of interrupts.

Register ExcCode Value:

Int

Additional State Saved:

Table 4.10 Register States an Interrupt Exception

Register State	Value
Cause$_{IP}$	indicates the interrupts that are pending.

Entry Vector Used:

See 4.3.2 "Generation of Exception Vector Offsets for Vectored Interrupts" on page 65 for the entry vector used, depending on the interrupt mode the processor is operating in.

4.8.6 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

4.8.7 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 4.11 CP0 Register States on an Address Exception Error

Register State	Value
BadVAddr	failing address

Entry Vector Used:

General exception vector (offset 0x180)

4.8.8 Bus Error Exception — Instruction Fetch or Data Access

A bus error exception occurs when an instruction or data access makes a bus request and that request terminates in an error. The bus error exception can occur on either an instruction fetch or a data access. Bus error exceptions that occur on an instruction fetch have a higher priority than bus error exceptions that occur on a data access.

Bus errors taken on any external access on the M4K core are always precise.

***Cause* Register ExcCode Value:**

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

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

4.8.9 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

4.8.10 Execution Exception — System Call

The system call exception is one of the nine 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)

4.8.11 Execution Exception — Breakpoint

The breakpoint exception is one of the nine 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)

4.8.12 Execution Exception — Reserved Instruction

The reserved instruction exception is one of the nine 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)

4.8.13 Execution Exception — Coprocessor Unusable

The coprocessor unusable exception is one of the nine 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 4.12 Register States on a Coprocessor Unusable Exception

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

Entry Vector Used:

General exception vector (offset 0x180)

4.8.14 Execution Exception — CorExtend Unusable

The CorExtend unusable exception is one of the nine execution exceptions. All of these exceptions have the same priority. A CorExtend Unusable exception occurs when an attempt is made to execute a CorExtend instruction when $\text{Status}_{\text{CEE}}$ is cleared. It is implementation dependent whether this functionality is supported. Generally, the functionality will only be supported if a CorExtend block contains local destination registers

Cause **Register ExcCode Value:**

CEU

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

4.8.15 Execution Exception — Coprocessor 2 Exception

The Coprocessor 2 exception is one of the nine execution exceptions. All of these exceptions have the same priority. A Coprocessor 2 exception occurs when a valid Coprocessor 2 instruction cause a general exception in the Coprocessor 2.

Cause **Register ExcCode Value:**

C2E

Additional State Saved:

Depending on the Coprocessor 2 implementation, additional state information of the exception can be saved in a Coprocessor 2 control register.

Entry Vector Used:

General exception vector (offset 0x180)

4.8.16 Execution Exception — Implementation-Specific 1 Exception

The Implementation-Specific 1 exception is one of the nine execution exceptions. All of these exceptions have the same priority. An implementation-specific 1 exception occurs when a valid coprocessor 2 instruction cause an implementation-specific 1 exception in the Coprocessor 2.

***Cause* Register ExcCode Value:**

IS1

Additional State Saved:

Depending on the coprocessor 2 implementation, additional state information of the exception can be saved in a coprocessor 2 control register.

Entry Vector Used:

General exception vector (offset 0x180)

4.8.17 Execution Exception — Integer Overflow

The integer overflow exception is one of the nine 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)

4.8.18 Execution Exception — Trap

The trap exception is one of the nine 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)

4.8.19 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

4.8.20 Complex Break Exception

A complex data break exception occurs when the complex hardware breakpoint detects an enabled breakpoint. Complex breaks are taken imprecisely—the instruction that actually caused the exception is allowed to complete and the *DEPC* register and DBD bit in the *Debug* register point to a following instruction.

Debug Register Debug Status Bit Set:

DIBImpr, *DDBLImpr*, and/or *DDBSImpr*

Additional State Saved:

Debug2 fields indicate which type(s) of complex breakpoints were detected.

Entry Vector Used:

Debug exception vector

4.9 Exception Handling and Servicing Flowcharts

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

- General exceptions and their exception handler
- Reset, soft rese,t and NMI exceptions, and a guideline to their handler
- Debug exceptions

Figure 4.3 General Exception Handler (HW)

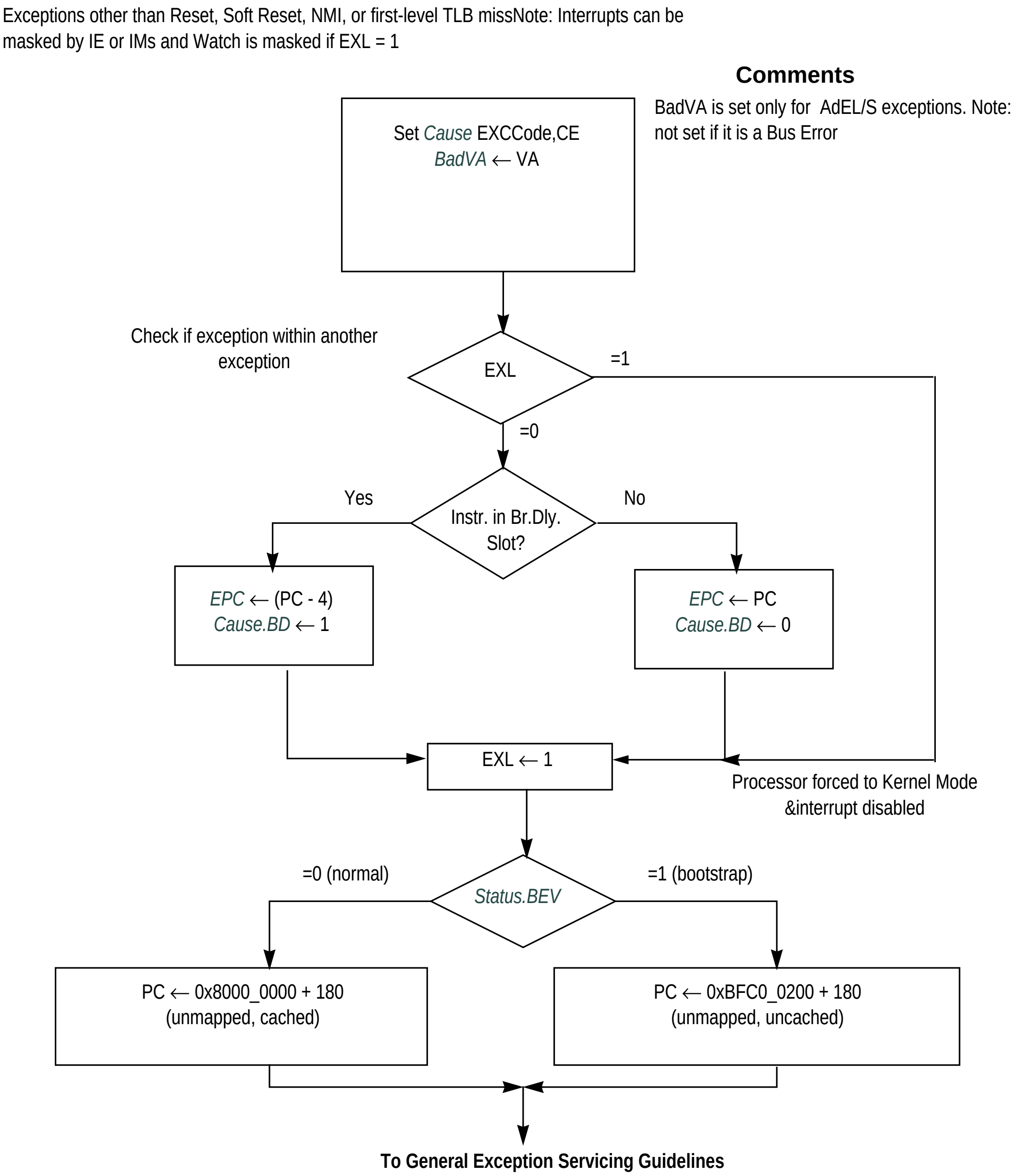

Exceptions other than Reset, Soft Reset, NMI, or first-level TLB missNote: Interrupts can be masked by IE or IMs and Watch is masked if EXL = 1
Comments
Set *Cause* EXCCode,CE
BadVA ← VA
BadVA is set only for AdEL/S exceptions. Note: not set if it is a Bus Error
Check if exception within another exception
EXL
=1
=0
Yes
Instr. in Br.Dly. Slot?
No
EPC ← (PC - 4)
Cause.BD ← 1
EPC ← PC
Cause.BD ← 0
EXL ← 1
Processor forced to Kernel Mode &interrupt disabled
=0 (normal)
Status.BEV
=1 (bootstrap)
PC ← 0x8000_0000 + 180
(unmapped, cached)
PC ← 0xBFC0_0200 + 180
(unmapped, uncached)
To General Exception Servicing Guidelines

Figure 4.4 General Exception Servicing Guidelines (SW)

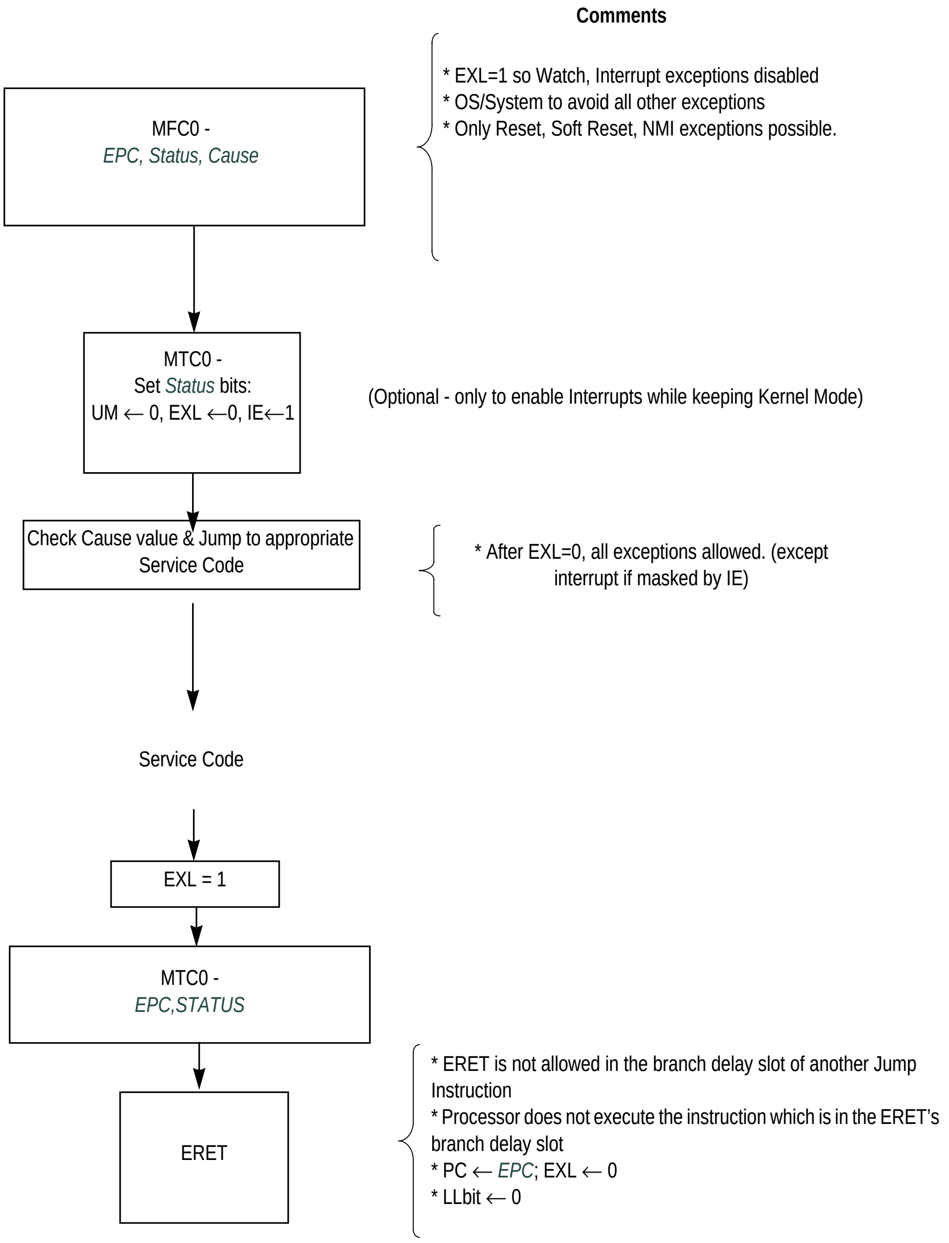
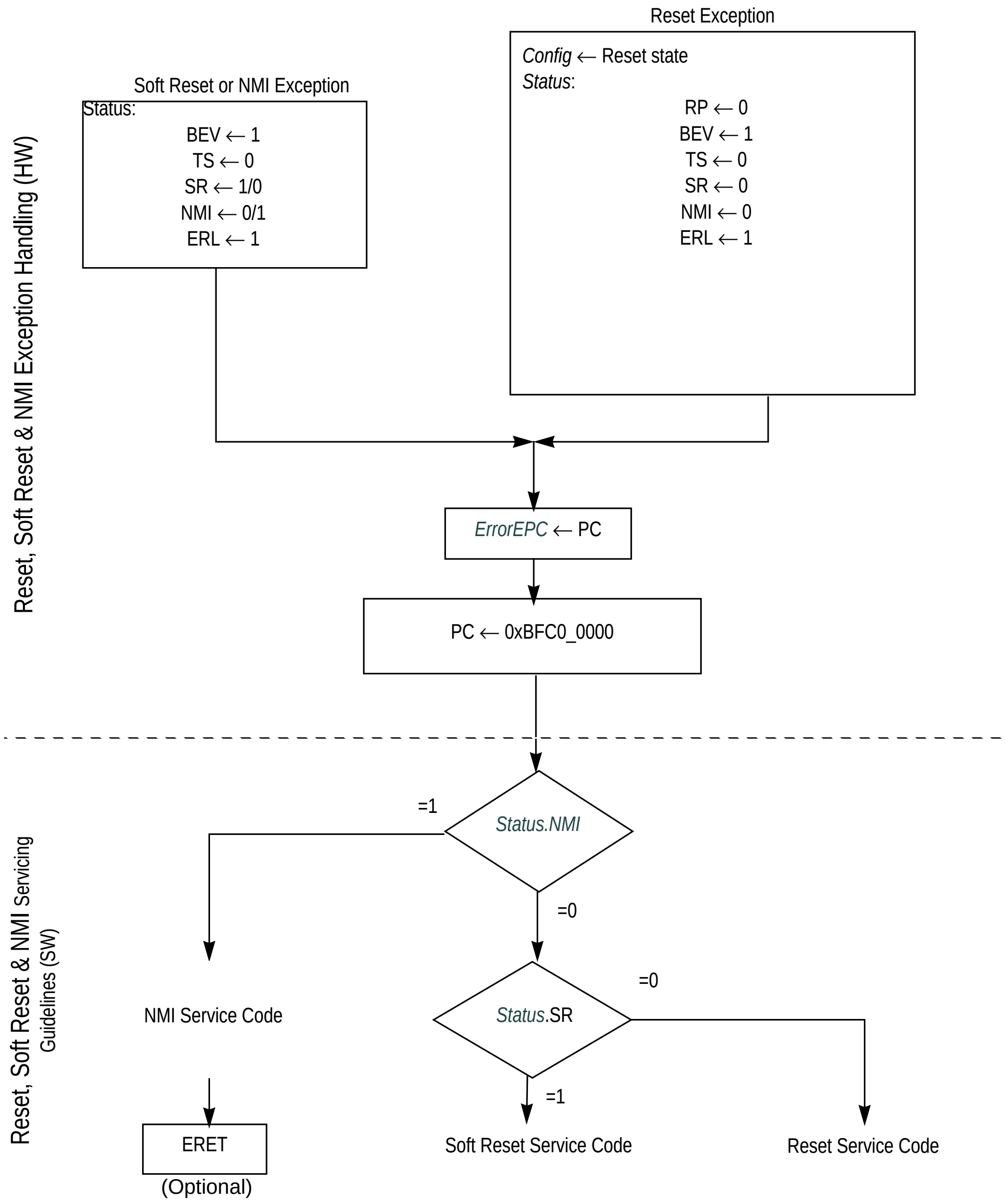

Comments
MFC0 -
EPC, Status, Cause
* EXL=1 so Watch, Interrupt exceptions disabled
* OS/System to avoid all other exceptions
* Only Reset, Soft Reset, NMI exceptions possible.
MTC0 -
Set Status bits:
UM ← 0, EXL ←0, IE←1
(Optional - only to enable Interrupts while keeping Kernel Mode)
Check Cause value & Jump to appropriate Service Code
* After EXL=0, all exceptions allowed. (except interrupt if masked by IE)
Service Code
EXL = 1
MTC0 -
EPC,STATUS
ERET
* ERET is not allowed in the branch delay slot of another Jump Instruction
* Processor does not execute the instruction which is in the ERET's branch delay slot
* PC ← EPC; EXL ← 0
* LLbit ← 0

Figure 4.5 Reset, Soft Reset and NMI Exception Handling and Servicing Guidelines



Chapter 5

CP0 Registers of the M4K™ Core

The System Control Coprocessor (CP0) provides the register interface to the M4K processor core and supports memory management, address translation, exception handling, and other privileged operations. Each CP0 register has a unique number that identifies it; this number is referred to as the *register number*. For instance, the *PageMask* register is register number 5. For more information on the EJTAG registers, refer to Chapter 8, “EJTAG Debug Support in the M4K™ Core” on page 127.

After updating a CP0 register there is a hazard period of zero or more instructions from the update instruction (MTC0) and until the effect of the update has taken place in the core. Refer to Chapter 10, “M4K™ Processor Core Instructions” on page 207 for further details on CP0 hazards.

The current chapter contains the following sections:

- Section 5.1 “CP0 Register Summary”
- Section 5.2 “CP0 Register Descriptions”

5.1 CP0 Register Summary

Table 5.1 lists the CP0 registers in numerical order. The individual registers are described throughout this chapter. Where more than one registers shares the same register number at different values of the “sel” field of the instruction, their names are listed using a slash (/) as separator.

Table 5.1 CP0 Registers

Register Number	Register Name	Function
0-6	Reserved	Reserved in the M4K core.
7	HWREna	Enables access via the RDHWR instruction to selected hardware registers in non-privileged mode.
8	BadVAddr[1]	Reports the address for the most recent address-related exception.
9	Count[1]	Processor cycle count.
10	Reserved	Reserved in the M4K core.
11	Compare[1]	Timer interrupt control.
12	Status/ IntCtl/ SRSCtl/ SRSMap1	Processor status and control; interrupt control; and shadow set control.
13	Cause[1]	Cause of last exception.
14	EPC[1]	Program counter at last exception.

Table 5.1 CP0 Registers (Continued)

Register Number	Register Name	Function
15	PRId/ EBase	Processor identification and revision; exception base address.
16	Config/ Config1/ Config2/ Config3	Configuration registers.
17-22	Reserved	Reserved in the M4K core.
23	Debug/ Debug2/ TraceControl/ TraceControl2/ UserTraceData/ TraceBPC[2]	Debug control/exception status and EJTAG trace control.
24	DEPC[2]	Program counter at last debug exception.
25-29	Reserved	Reserved in the M4K core.
30	ErrorEPC[1]	Program counter at last error.
31	DeSAVE[2]	Debug handler scratchpad register.

[1] Registers used in exception processing.
[2] Registers used in debug.

5.2 CP0 Register Descriptions

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

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

Table 5.2 CP0 Register Field Types

Read/Write 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.	

Table 5.2 CP0 Register Field Types (Continued)

Read/Write Notation	Hardware Interpretation	Software Interpretation
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 can not be read by software. Software reads of this field will return an UNDEFINED value.	
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.

5.2.1 *HWREna* Register (CP0 Register 7, Select 0)

The *HWREna* register contains a bit mask that determines which hardware registers are accessible via the RDHWR instruction.

Figure 5.1 shows the format of the *HWREna* Register; Table 5.3 describes the *HWREna* register fields.

Figure 5.1 HWREna Register Format

31 … 4	3 … 0
0 0000 0000 0000 0000 0000 0000 0000	Mask

Table 5.3 HWREna Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
0	31..4	Must be written with zero; returns zero on read	0	0
Mask	3..0	Each bit in this field enables access by the RDHWR instruction to a particular hardware register (which may not be an actual register). If bit ‘n’ in this field is a 1, access is enabled to hardware register ‘n’. If bit ‘n’ of this field is a 0, access is disabled. See the RDHWR instruction for a list of valid hardware registers.	R/W	0

Privileged software may determine which of the hardware registers are accessible by 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.

5.2.2 *BadVAddr* Register (CP0 Register 8, Select 0)

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

- Address error (AdEL or AdES)

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

Figure 5.2 *BadVAddr* Register Format

31
0
BadVAddr

Table 5.4 *BadVAddr* Register Field Description

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

5.2.3 *Count* Register (CP0 Register 9, Select 0)

The *Count* register acts as a timer, incrementing at a constant rate, whether or not an instruction is executed, retired, or any forward progress is made through the pipeline. The counter increments every other clock, if the *DC* bit in the *Cause* register is 0.

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

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.

Figure 5.3 *Count* Register Format

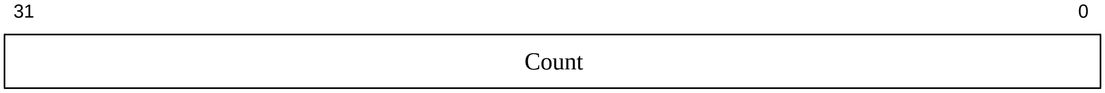


Table 5.5 *Count* Register Field Description

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

5.2.4 *Compare* Register (CP0 Register 11, Select 0)

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

When the value of the *Count* register equals the value of the *Compare* register, the *SI_TimerInt* pin is asserted. This pin will remain asserted until the *Compare* register is written. The *SI_TimerInt* pin 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 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. Writing a value to the *Compare* register, as a side effect, clears the timer interrupt.

Figure 5.4 *Compare* Register Format

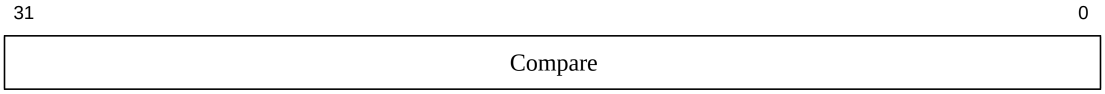


Table 5.6 *Compare* Register Field Description

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

5.2.5 *Status* Register (CP0 Register 12, Select 0)

The *Status* register is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. Fields of this register combine to create operating modes for the processor. Refer to 3.2 “Modes of Operation” on page 43 for a discussion of operating modes, and 4.3 “Interrupts” on page 57 for a discussion of interrupt modes.

Interrupt Enable: Interrupts are enabled when all of the following conditions are true:

- *IE* = 1
- *EXL* = 0
- *ERL* = 0
- *DM* = 0

If these conditions are met, then the settings of the *IM* and *IE* bits enable the interrupts.

Operating Modes: If the *DM* bit in the *Debug* register is 1, then the processor is in debug mode; otherwise the processor is in either kernel or user mode. The following CPU *Status* register bit settings determine user or kernel mode:

- User mode: *UM* = 1, *EXL* = 0, and *ERL* = 0

- Kernel mode: *UM* = 0, or *EXL* = 1, or *ERL* = 1

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 5.5 shows the format of the *Status* register; Table 5.7 describes the *Status* register fields.

Figure 5.5 Status Register Format

31 28	27	26	25	24 23	22	21	20	19	18	17	16	15 10	9 8	7 6 5	4	3	2	1	0
CU3..CU0	RP	FR	RE	R	BEV	TS	SR	NMI	R	CEE	R	IM7..IM2	IM1..IM0	R	UM	R	ERL	EXL	IE
												IPL							

Table 5.7 Status Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>CU3</td><td>31</td><td>Controls access to coprocessor 3. COP3 is not supported. This bit cannot be written and will read as 0.</td><td>R</td><td>0</td></tr>
<tr><td>CU2</td><td>30</td><td>Controls access to coprocessor 2. This bit can only be written if coprocessor is attached to the COP2 interface. (C2 bit in Config1 is set). This bit will read as 0 if no coprocessor is present.</td><td>R/W</td><td>0</td></tr>
<tr><td>CU1</td><td>29</td><td>Controls access to Coprocessor 1. COP1 is not supported. This bit cannot be written and will read as 0.</td><td>R</td><td>0</td></tr>
<tr><td>CU0</td><td>28</td><td>Controls access to coprocessor 0
0: access not allowed
1: access allowed
Coprocessor 0 is always usable when the processor is running in kernel mode, independent of the state of the CU0 bit.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>RP</td><td>27</td><td>Enables reduced power mode. The state of the RP bit is available on the external core interface as the <i>SI_RP</i> signal.</td><td>R/W</td><td>0 for Cold Reset only.</td></tr>
<tr><td>FR</td><td>26</td><td>This bit is related to floating point registers. Since the M4K core does not contain a floating point unit, this bit is ignored on write and read as zero.</td><td>R</td><td>0</td></tr>
<tr><td>RE</td><td>25</td><td>Used to enable reverse-endian memory references while the processor is running in user mode:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>User mode uses configured endianness</td></tr>
<tr><td>1</td><td>User mode uses reversed endianness</td></tr>
</table>
Neither Debug Mode nor Kernel Mode nor Supervisor Mode references are affected by the state of this bit.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>R</td><td>24:23</td><td>Reserved. This field is ignored on write and read as 0.</td><td>R</td><td>0</td></tr>
</table>

Table 5.7 Status Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>BEV</td><td>22</td><td>Controls the location of exception vectors:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Normal</td></tr>
<tr><td>1</td><td>Bootstrap</td></tr>
</table></td><td>R/W</td><td>1</td></tr>
<tr><td>TS</td><td>21</td><td>TLB shutdown.
Since the M4K core does not contain a TLB, this bit is ignored on write and read as 0.</td><td>R</td><td>0</td></tr>
<tr><td>SR</td><td>20</td><td>Indicates that the entry through the reset exception vector was due to a Soft Reset:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Not Soft Reset (NMI or Reset)</td></tr>
<tr><td>1</td><td>Soft Reset</td></tr>
</table>
Software can only write a 0 to this bit to clear it and cannot force a 0-1 transition.</td><td>R/W</td><td>1 for Soft Reset; 0 otherwise</td></tr>
<tr><td>NMI</td><td>19</td><td>Indicates that the entry through the reset exception vector was due to an NMI:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Not NMI (Soft Reset or Reset)</td></tr>
<tr><td>1</td><td>NMI</td></tr>
</table>
Software can only write a 0 to this bit to clear it and cannot force a 0-1 transition.</td><td>R/W</td><td>1 for NMI; 0 otherwise</td></tr>
<tr><td>R</td><td>18</td><td>Reserved. Ignored on write and read as zero.</td><td>R</td><td>0</td></tr>
<tr><td>CEE</td><td>17</td><td>CorExtend Enable: Implementation dependent. If CorExtend block indicates that this bit should be used, any attempt to execute a CorExtend instruction with this bit cleared will result in a CorExtend Unusable exception. This bit is reserved if CorExtend is not present.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>R</td><td>16</td><td>Reserved. Ignored on write and read as zero.</td><td>R</td><td>0</td></tr>
</table>

Table 5.7 Status Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IM7..IM2</td><td>15..10</td><td>Interrupt Mask: Controls the enabling of each of the hardware interrupts. Refer to 4.3 “Interrupts” on page 57 for a complete discussion of enabled interrupts.
An interrupt is taken if interrupts are enabled and the corresponding bits are set in both the Interrupt Mask field of the Status register and the Interrupt Pending field of the Cause register and the IE bit is set in the Status register.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Interrupt request disabled</td></tr>
<tr><td>1</td><td>Interrupt request enabled</td></tr>
</table>
In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled, these bits take on a different meaning and are interpreted as the IPL field, described below.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>IPL</td><td>15..10</td><td>Interrupt Priority Level.
In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled , this field is the encoded (0..63) value of the current IPL. An interrupt will be signaled only if the requested IPL is higher than this value.
If EIC interrupt mode is not enabled, these bits take on a different meaning and are interpreted as the IM7..IM2 bits, described above.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>IM1..IM0</td><td>9..8</td><td>Interrupt Mask: Controls the enabling of each of the software interrupts. Refer to Section 4.3 “Interrupts”for a complete discussion of enabled interrupts.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Interrupt request disabled</td></tr>
<tr><td>1</td><td>Interrupt request enabled</td></tr>
</table>
In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled, these bits are writable, but have no effect on the interrupt system.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>R</td><td>7:5</td><td>Reserved. This field is ignored on write and read as 0.</td><td>R</td><td>0</td></tr>
<tr><td>UM</td><td>4</td><td>This bit denotes the base operating mode of the processor. See 3.2 “Modes of Operation” on page 43 for a full discussion of operating modes. The encoding of this bit is:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Base mode is Kernel Mode</td></tr>
<tr><td>1</td><td>Base mode is User Mode</td></tr>
</table>
Note that the processor can also be in kernel mode if ERL or EXL is set, regardless of the state of the UM bit.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>R</td><td>3</td><td>This bit is reserved. This bit is ignored on write and read as zero.</td><td>R</td><td>0</td></tr>
</table>

Table 5.7 Status Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>ERL</td><td>2</td><td>Error Level; Set by the processor when a Reset, Soft Reset, NMI or Cache Error exception are taken.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Normal level</td></tr>
<tr><td>1</td><td>Error level</td></tr>
</table>
When ERL is set:
• The processor is running in kernel mode
• Interrupts are disabled
• The ERET instruction will use the return address held in ErrorEPC instead of EPC
• The lower 2^{29} bytes of kuseg are treated as an unmapped and uncached region. See Chapter 3, “Modes of Operation” on page 43. This allows main memory to be accessed in the presence of cache errors. The operation of the processor is <b>UNDEFINED</b> if the ERL bit is set while the processor is executing instructions from kuseg.</td><td>R/W</td><td>1</td></tr>
<tr><td>EXL</td><td>1</td><td>Exception Level; Set by the processor when any exception other than Reset, Soft Reset, or NMI exceptions is taken.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Normal level</td></tr>
<tr><td>1</td><td>Exception level</td></tr>
</table>
When EXL is set:
• The processor is running in Kernel Mode
• Interrupts are disabled.
• EPC, Cause$_{BD}$ and SRSCtl (implementations of Release 2 of the Architecture only) will not be updated if another exception is taken</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>IE</td><td>0</td><td>Interrupt Enable: Acts as the master enable for software and hardware interrupts:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Interrupts are disabled</td></tr>
<tr><td>1</td><td>Interrupts are enabled</td></tr>
</table>
In Release 2 of the Architecture, this bit may be modified separately via the DI and EI instructions.</td><td>R/W</td><td>Undefined</td></tr>
</table>

5.2.6 *IntCtl* Register (CP0 Register 12, Select 1)

The *IntCtl* register controls the expanded interrupt capability added in Release 2 of the Architecture, including vectored interrupts and support for an external interrupt controller. This register does not exist in implementations of Release 1 of the Architecture.

Figure 5.6 shows the format of the *IntCtl* register; Table 5.8 describes the *IntCtl* register fields.

Figure 5.6 IntCtl Register Format

31 29	28 26	25 10	9 5	4 0
IPTI	IPPCI	0	VS	0

Table 5.8 IntCtl Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IPTI</td><td>31..29</td><td>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 Cause$_{TI}$ for a potential interrupt.
<table>
<tr><th>Encoding</th><th>IP bit</th><th>Hardware Interrupt Source</th></tr>
<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>
</table>
The value of this bit is set by the static input, *SI_IPTI[2:0]*. This allows external logic to communicate the specific *SI_Int* hardware interrupt pin to which the *SI_TimerInt* 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.</td><td>R</td><td>Externally Set</td></tr>
<tr><td>IPPCI</td><td>28..26</td><td>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 Cause$_{PCI}$ for a potential interrupt.
Since performance counters are not implemented on the M4K core (Config1$_{PC}$ = 0), this field is ignored on write and returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>0</td><td>25..10</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td></tr>
</table>

Table 5.8 IntCtl Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>VS</td><td>9..5</td><td>Vector Spacing. If vectored interrupts are implemented (as denoted by Config3$_{VInt}$ or Config3$_{VEIC}$), this field specifies the spacing between vectored interrupts.
<table>
<tr><th>Encoding</th><th>Spacing Between Vectors (hex)</th><th>Spacing Between Vectors (decimal)</th></tr>
<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>
</table>
All other values are reserved. The operation of the processor is <b>UNDEFINED</b> if a reserved value is written to this field.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>4..0</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td></tr>
</table>

5.2.7 *SRSCtl* Register (CP0 Register 12, Select 2)

The *SRSCtl* register controls the operation of GPR shadow sets in the processor. This register does not exist in implementations of the architecture prior to Release 2.

Figure 5.7 shows the format of the *SRSCtl* register; Table 5.9 describes the *SRSCtl* register fields.

Figure 5.7 SRSCtl Register Format

31 30	29 26	25 22	21 18	17 16	15 12	11 10	9 6	5 4	3 0
0 00	HSS	0 00 00	EICSS	0 00	ESS	0 00	PSS	0 00	CSS

Table 5.9 SRSCtl Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
0	31..30	Must be written as zeros; returns zero on read.	0	0

Table 5.9 SRSCtl Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>HSS</td><td>29..26</td><td>Highest Shadow Set. This field contains the highest shadow set number that is implemented by this processor. A value of zero in this field indicates that only the normal GPRs are implemented.
Possible values of this field for the M4K processor are:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>One shadow set (normal GPR set) is present.</td></tr>
<tr><td>1</td><td>Two shadow sets are present.</td></tr>
<tr><td>3</td><td>Four shadow sets are present.</td></tr>
<tr><td>7</td><td>Eight shadow sets are present</td></tr>
<tr><td>2, 4-6, 9-15</td><td>Reserved</td></tr>
</table>
The value in this field also represents the highest value that can be written to the ESS, EICSS, PSS, and CSS fields of this register, or to any of the fields of the *SRSMap* register. The operation of the processor is **UNDEFINED** if a value larger than the one in this field is written to any of these other fields.</td><td>R</td><td>Preset</td></tr>
<tr><td>0</td><td>25..22</td><td>Must be written as zeros; returns zero on read.</td><td>0</td><td>0</td></tr>
<tr><td>EICSS</td><td>21..18</td><td>EIC interrupt mode shadow set. If Config3$_{VEIC}$ 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 SRSMap register to select the current shadow set for the interrupt.
See 4.3.1.3 “External Interrupt Controller Mode” on page 63 for a discussion of EIC interrupt mode. If Config3$_{VEIC}$ is 0, this field must be written as zero, and returns zero on read.</td><td>R</td><td>Undefined</td></tr>
<tr><td>0</td><td>17..16</td><td>Must be written as zeros; returns zero on read.</td><td>0</td><td>0</td></tr>
<tr><td>ESS</td><td>15..12</td><td>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 <b>UNDEFINED</b> if software writes a value into this field that is greater than the value in the HSS field.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>11..10</td><td>Must be written as zeros; returns zero on read.</td><td>0</td><td>0</td></tr>
</table>

Table 5.9 SRSCtl Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
PSS	9..6	Previous Shadow Set. If GPR shadow registers are implemented, and with the exclusions noted in the next paragraph, this field is copied from the CSS field when an exception or interrupt occurs. An ERET instruction copies this value back into the CSS field if $\text{Status}_{\text{BEV}}$ = 0. This field is not updated on any exception which sets $\text{Status}_{\text{ERL}}$ to 1 (i.e., Reset, Soft Reset, NMI, cache error), an entry into EJTAG Debug mode, or any exception or interrupt that occurs with $\text{Status}_{\text{EXL}}$ = 1, or $\text{Status}_{\text{BEV}}$ = 1. This field is not updated on an exception that occurs while $\text{Status}_{\text{ERL}}$ = 1. The operation of the processor is **UNDEFINED** if software writes a value into this field that is greater than the value in the HSS field.	R/W	0
0	5..4	Must be written as zeros; returns zero on read.	0	0
CSS	3..0	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 PSS field on an ERET. Table 5.10 describes the various sources from which the CSS field is updated on an exception or interrupt. This field is not updated on any exception which sets $\text{Status}_{\text{ERL}}$ to 1 (i.e., Reset, Soft Reset, NMI, cache error), an entry into EJTAG Debug mode, or any exception or interrupt that occurs with $\text{Status}_{\text{EXL}}$ = 1, or $\text{Status}_{\text{BEV}}$ = 1. Neither is it updated on an ERET with $\text{Status}_{\text{ERL}}$ = 1 or $\text{Status}_{\text{BEV}}$ = 1. This field is not updated on an exception that occurs while $\text{Status}_{\text{ERL}}$ = 1. The value of CSS can be changed directly by software only by writing the PSS field and executing an ERET instruction.	R	0

5.2.8 *SRSMap* Register (CP0 Register 12, Select 3)

Table 5.10 Sources for new SRSCtl$_{CSS}$ on an Exception or Interrupt

Exception Type	Condition	SRSCtl$_{CSS}$ Source	Comment
Exception	All	SRSCtl$_{ESS}$	
Non-Vectored Interrupt	Cause$_{IV}$ = 0	SRSCtl$_{ESS}$	Treat as exception
Vectored Interrupt	Cause$_{IV}$ = 1 and Config3$_{VEIC}$ = 0 and Config3$_{VInt}$ = 1	SRSMap$_{VECTNUM}$	Source is internal map register. (for VECTNUM see Table 4.3)
Vectored EIC Interrupt	Cause$_{IV}$ = 1 and Config3$_{VEIC}$ = 1	SRSCtl$_{EICSS}$	Source is external interrupt controller.

The *SRSMap* register contains 8 4-bit fields that provide the mapping from an vector number to the shadow set number to use when servicing such an interrupt. The values from this register are not used for a non-interrupt exception, or a non-vectored interrupt (*Cause$_{IV}$* = 0 or *IntCtl$_{VS}$* = 0). In such cases, the shadow set number comes from *SRSCtl$_{ESS}$*.

If *SRSCtl$_{HSS}$* is zero, the results of a software read or write of this register are **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 *SRSMap* register contains the shadow register set numbers for vector numbers 7..0. The same shadow set number can be established for multiple interrupt vectors, creating a many-to-one mapping from a vector to a single shadow register set number.

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

Figure 5.8 SRSMap Register Format

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
SSV7	SSV6	SSV5	SSV4	SSV3	SSV2	SSV1	SSV0

Table 5.11 SRSMap Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
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

Table 5.11 SRSMap Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
SSV0	3..0	Shadow register set number for Vector Number 0	R/W	0

5.2.9 *Cause* Register (CP0 Register 13, Select 0)

The *Cause* register primarily describes the cause of the most recent exception. In addition, fields also control software interrupt requests and the vector through which interrupts are dispatched. With the exception of the $IP_{1..0}$, *DC*, *IV*, and *WP* fields, all fields in the *Cause* register are read-only. Release 2 of the Architecture added optional support for an External Interrupt Controller (EIC) interrupt mode, in which $IP_{7..2}$ are interpreted as the Requested Interrupt Priority Level (RIPL).

Figure 5.9 shows the format of the *Cause* register; Table 5.12 describes the *Cause* register fields.

Figure 5.9 Cause Register Format

31	30	29 28	27	26	25 24	23	22	21 16	15 10	9 8	7	6 2	1 0
BD	TI	CE	DC	PCI	0	IV	WP	0	IP7..IP2	IP1..IP0	0	Exc Code	0
									RIPL				

Table 5.12 Cause Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
BD	31	Indicates whether the last exception taken occurred in a branch delay slot: Encoding | Meaning 0 | Not in delay slot 1 | In delay slot The processor updates BD only if $Status_{EXL}$ was zero when the exception occurred.	R	Undefined
TI	30	Timer Interrupt. This bit denotes whether a timer interrupt is pending (analogous to the IP bits for other interrupt types): Encoding | Meaning 0 | No timer interrupt is pending 1 | Timer interrupt is pending The state of the TI bit is available on the external core interface as the *SI_TimerInt* signal	R	Undefined
CE	29..28	Coprocessor unit number referenced when a Coprocessor Unusable exception is taken. This field is loaded by hardware on every exception, but is **UNPREDICTABLE** for all exceptions except for Coprocessor Unusable.	R	Undefined

Table 5.12 Cause Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>DC</td><td>27</td><td>Disable Count register. In some power-sensitive applications, the Count register is not used and is the source of meaningful power dissipation. This bit allows the *Count* register to be stopped in such situations.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Enable counting of *Count* register</td></tr>
<tr><td>1</td><td>Disable counting of *Count* register</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td>PCI</td><td>26</td><td>Performance Counter Interrupt. In an implementation of Release 2 of the Architecture, this bit denotes whether a performance counter interrupt is pending (analogous to the IP bits for other interrupt types):
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No timer interrupt is pending</td></tr>
<tr><td>1</td><td>Timer interrupt is pending</td></tr>
</table>
Since performance counters are not implemented ($Config1_{PC}$ = 0), this bit must be written as zero and returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>IV</td><td>23</td><td>Indicates whether an interrupt exception uses the general exception vector or a special interrupt vector:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Use the general exception vector (16#180)</td></tr>
<tr><td>1</td><td>Use the special interrupt vector (16#200)</td></tr>
</table>
In implementations of Release 2 of the architecture, if the $Cause_{IV}$ is 1 and $Status_{BEV}$ is 0, the special interrupt vector represents the base of the vectored interrupt table.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>WP</td><td>22</td><td>Indicates that a watch exception was deferred because $Status_{EXL}$ or $Status_{ERL}$ were a one at the time the watch exception was detected. This bit both indicates that the watch exception was deferred, and causes the exception to be initiated once $Status_{EXL}$ and $Status_{ERL}$ are both zero. As such, software must clear this bit as part of the watch exception handler to prevent a watch exception loop.
Software 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 $Status_{EXL}$ and $Status_{ERL}$ are both zero.
Since watch registers are not implemented on the M4K core, this bit is ignored on write and read as zero.</td><td>R</td><td>0</td></tr>
</table>

Table 5.12 Cause Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IP7..IP2</td><td>15..10</td><td>Indicates an interrupt is pending:
<table>
<tr><th>Bit</th><th>Name</th><th>Meaning</th></tr>
<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>
</table>
If EIC interrupt mode is not enabled, timer interrupts are combined in a system-dependent way with any hardware interrupt. If EIC interrupt mode is enabled, these bits take on a different meaning and are interpreted as the RIPL field, described below.
See 4.3 “Interrupts” on page 57 for a general description of interrupt processing.</td><td>R</td><td>Undefined</td></tr>
<tr><td>RIPL</td><td>15..10</td><td>Requested Interrupt Priority Level.
If EIC interrupt mode is enabled, this field is the encoded (0..63) value of the requested interrupt. A value of zero indicates that no interrupt is requested.
If EIC interrupt mode is not enabled, these bits take on a different meaning and are interpreted as the IP7..IP2 bits, described above.</td><td>R</td><td>Undefined</td></tr>
<tr><td>IP1..IP0</td><td>9..8</td><td>Controls the request for software interrupts:
<table>
<tr><th>Bit</th><th>Name</th><th>Meaning</th></tr>
<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>
</table>
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 *SI_SWInt[1:0]* bus.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>ExcCode</td><td>6..2</td><td>Exception code - see Table 5.13</td><td>R</td><td>Undefined</td></tr>
<tr><td>0</td><td>25..24, 21..16, 7, 1..0</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td></tr>
</table>

Table 5.13 Cause Register ExcCode Field

Exception Code Value			
Decimal	**Hexadecimal**	**Mnemonic**	**Description**
0	16#00	Int	Interrupt
1-3	16#00-16#03	-	Reserved
4	16#04	AdEL	Address error exception (load or instruction fetch)

Table 5.13 Cause Register ExcCode Field (Continued)

Exception Code Value			
Decimal	**Hexadecimal**	**Mnemonic**	**Description**
5	16#05	AdES	Address error exception (store)
6	16#06	IBE	Bus error exception (instruction fetch)
7	16#07	DBE	Bus error exception (data reference: load or store)
8	16#08	Sys	Syscall exception
9	16#09	Bp	Breakpoint exception
10	16#0a	RI	Reserved instruction exception
11	16#0b	CpU	Coprocessor Unusable exception
12	16#0c	Ov	Arithmetic Overflow exception
13	16#0d	Tr	Trap exception
14-15	16#0e-16#0f	-	Reserved
16	16#10	IS1	Implementation-Specific Exception 1 (COP2)
17	16#11	CEU	CorExtend Unusable
18	16#12	C2E	Coprocessor 2 exceptions
19-31	16#13-16#1f	-	Reserved

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

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

For synchronous (precise) exceptions, the *EPC* contains one of the following:

- The virtual address of the instruction that was the direct cause of the exception
- 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.

On new exceptions, the processor does not write to the *EPC* register when the *EXL* bit in the *Status* register is set, however, the register can still be written via the MTC0 instruction.

In processors that implement the MIPS16e ASE, a read of the *EPC* register (via MFC0) returns the following value in the destination GPR:

```
GPR[rt] ← $\text{ExceptionPC}_{31..1}$ || $\text{ISAMode}_0$
```

That is, the upper 31 bits of the exception PC are combined with the lower bit of the *ISAMode* field and written to the GPR.

Similarly, a write to the *EPC* register (via MTC0) takes the value from the GPR and distributes that value to the exception PC and the *ISAMode* field, as follows

```
ExceptionPC ← $\text{GPR[rt]}_{31..1}$ || 0
ISAMode ← 2#0 || $\text{GPR[rt]}_0$
```

That is, the upper 31 bits of the GPR are written to the upper 31 bits of the exception PC, and the lower bit of the exception PC is cleared. The upper bit of the ISAMode field is cleared and the lower bit is loaded from the lower bit of the GPR.

Figure 5.10 *EPC* Register Format

31 ... 0
EPC

Table 5.14 *EPC* Register Field Description

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
EPC	31:0	Exception Program Counter.	R/W	Undefined

5.2.11 Processor Identification (CP0 Register 15, Select 0)

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 5.11 *PRId* Register Format

31 ... 24	23 ... 16	15 ... 8	7 5 4 2 1 0
R	Company ID	Processor ID	Revision

Table 5.15 *PRId* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
R	31:24	Reserved. Must be ignored on write and read as zero	R	0
Company ID	23:16	Identifies the company that designed or manufactured the processor. In the M4K this field contains a value of 1 to indicate MIPS Technologies, Inc.	R	1
Processor ID	15:8	Identifies the type of processor. This field allows software to distinguish between the various types of MIPS Technologies processors.	R	0x87
Revision	7:0	Specifies the revision number of the processor. This field allows software to distinguish between one revision and another of the same processor type. This field is broken up into the following three subfields	R	Preset
Major Revision	7:5	This number is increased on major revisions of the processor core	R	Preset
Minor Revision	4:2	This number is increased on each incremental revision of the processor and reset on each new major revision	R	Preset
Patch Level	1:0	If a patch is made to modify an older revision of the processor, this field will be incremented	R	Preset

5.2.12 *EBase* Register (CP0 Register 15, Select 1)

The *EBase* register is a read/write register containing the base address of the exception vectors used when Status$_{\text{BEV}}$ 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 Status$_{\text{BEV}}$ is 0. The exception vector base address comes from the fixed defaults (see 4.5 "Exception Vector Locations" on page 67) when Status$_{\text{BEV}}$ 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.

If the value of the exception base register is to be changed, this must be done with Status$_{\text{BEV}}$ equal 1. The operation of the processor is **UNDEFINED** if the Exception Base field is written with a different value when Status$_{\text{BEV}}$ is 0.

Combining bits 31..20 with the *Exception Base* field allows the base address of the exception vectors to be placed at any 4KByte page boundary. If vectored interrupts are used, a vector offset greater than 4KBytes can be generated. In this case, bit 12 of the *Exception Base* field must be zero. The operation of the processor is **UNDEFINED** if software writes bit 12 of the *Exception Base* field with a 1 and enables the use of a vectored interrupt whose offset is greater than 4KBytes from the exception base address.

Figure 5.12 shows the format of the *EBase* Register; Table 5.16 describes the *EBase* register fields.

Figure 5.12 EBase Register Format

31	30	29 ... 12	11 10	9 ... 0
1	0	Exception Base	0 0	CPUNum

Table 5.16 EBase Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
1	31	This bit is ignored on write and returns one on read.	R	1
0	30	This bit is ignored on write and returns zero on read.	R	0
Exception Base	29..12	In conjunction with bits 31..30, this field specifies the base address of the exception vectors when Status$_{\text{BEV}}$ is zero.	R/W	0
0	11..10	Must be written as zero; returns zero on read.	0	0
CPUNum	9..0	This field specifies the number of the CPU in a multi-processor system and can be used by software to distinguish a particular processor from the others. The value in this field is set by the SI_CPUNum[9:0] static input pins to the core. In a single processor system, this value should be set to zero.	R	Externally Set

5.2.13 *Config* Register (CP0 Register 16, Select 0)

The *Config* register specifies various configuration and capabilities information. Most of the fields in the *Config* register are initialized by hardware during the Reset exception process, or are constant.

Figure 5.13 *Config* Register Format — Select 0

31	30 28	27 25	24 23	22	21	20	19 17	16	15	14 13	12 10	9 7	6 3	2 0
M	K23	KU	0	UDI	SB	MDU	0	DS	BE	AT	AR	MT	0	K0

Figure 5.14 *Config* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
M	31	This bit is hardwired to ‘1’ to indicate the presence of the Config1 register.	R	1
K23	30:28	This field controls the cacheability of the kseg2 and kseg3 address segments in FM implementations. Refer to Table 5.17 for the field encoding.	FM: R/W	FM: 010
KU	27:25	This field controls the cacheability of the kuseg and useg address segments in FM implementations. Refer to Table 5.17 for the field encoding.	FM: R/W	FM: 010
0	24:23	Must be written as 0. Returns zero on reads.	0	0
UDI	22	This bit indicates that CorExtend User Defined Instructions have been implemented. 0 = No User Defined Instructions are implemented 1 = User Defined Instructions are implemented	R	Preset
SB	21	Indicates whether SimpleBE bus mode is enabled. Set via *SI_SimpleBE[0]* input pin. 0 = No reserved byte enables on SRAM interface 1 = Only simple byte enables allowed on SRAM interface	R	Externally Set
MDU	20	This bit indicates the type of Multiply/Divide Unit present. 0 = Fast, high-performance MDU 1 = Iterative, area-efficient MDU	R	Preset
0	19:17	Must be written as 0. Returns zero on reads.	0	0
DS	16	Dual SRAM interface. 0: Unified instruction/data SRAM interface 1: Dual instruction/data SRAM interfaces	R	Preset
BE	15	Indicates the endian mode in which the processor is running. Set via *SI_Endian* input pin. 0: Little endian 1: Big endian	R	Externally Set
AT	14:13	Architecture type implemented by the processor. This field is always 00 to indicate the MIPS32 architecture.	R	00
AR	12:10	Architecture revision level. This field is always 001 to indicate MIPS32 Release 2. 0: Release 1 1: Release 2 2-7: Reserved	R	001

Figure 5.14 *Config* Register Field Descriptions (Continued)

Fields				
Name	**Bit(s)**	**Description**	**Read/Write**	**Reset State**
MT	9:7	MMU Type: 3: Fixed Mapping 0-2, 4-7: Reserved	R	3
0	6:3	Must be written as zeros; returns zeros on reads.	0	0
K0	2:0	Kseg0 coherency algorithm. Refer to Table 5.17 for the field encoding.	R/W	010

Table 5.17 Cache Coherency Attributes

C(2:0) Value	Cache Coherency Attribute
2	Uncached.
3	Cached (Core treats as uncached, but passes attribute to the system for use with any external caching mechanisms)

5.2.14 *Config1* Register (CP0 Register 16, Select 1)

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

Figure 5.15 *Config1* Register Format — Select 1

31	30 25	24 22	21 19	18 16	15 13	12 10	9 7	6	5	4	3	2	1	0
M	MMU Size	IS	IL	IA	DS	DL	DA	C2	MD	PC	WR	CA	EP	FP

Table 5.18 *Config1* Register Field Descriptions — Select 1

Fields				
Name	**Bit(s)**	**Description**	**Read/Write**	**Reset State**
M	31	This bit is hardwired to ‘1’ to indicate the presence of the Config2 register.	R	1
MMU Size	30:25	This field contains the number of entries in the TLB minus one. The field is read as 0 decimal in the M4K cores, since no TLB is present.	R	0
IS	24:22	This field contains the number of instruction cache sets per way. Since the M4K core does not include caches, this field is always read as 0.	R	0
IL	21:19	This field contains the instruction cache line size. Since the M4K core does not include caches, this field is always read as 0.	R	0
IA	18:16	This field contains the level of instruction cache associativity. Since the M4K core does not include caches, this field is always read as 0.	R	0

Table 5.18 *Config1* Register Field Descriptions — Select 1 (Continued)

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
DS	15:13	This field contains the number of data cache sets per way. Since the M4K core does not include caches, this field is always read as 0.	R	0
DL	12:10	This field contains the data cache line size. Since the M4K core does not include caches, this field is always read as 0.	R	0
DA	9:7	This field contains the type of set associativity for the data cache. Since the M4K core does not include caches, this field is always read as 0.	R	0
C2	6	Coprocessor 2 present. 0: No coprocessor is attached to the COP2 interface 1: A coprocessor is attached to the COP2 interface If the Cop2 interface logic is not implemented, this bit will read 0.	R	Preset
MD	5	MDMX implemented. This bit always reads as 0 because MDMX is not supported.	R	0
PC	4	Performance Counter registers implemented. Always a 0 since the M4K core does not contain Performance Counters.	R	0
WR	3	Watch registers implemented. 0: No Watch registers are present 1: One or more Watch registers are present This bit is always read as 0 since the M4K core does not contain Watch registers.	R	0
CA	2	Code compression (MIPS16e) implemented. 0: No MIPS16e present 1: MIPS16e is implemented	R	Preset
EP	1	EJTAG present: This bit is always set to indicate that the core implements EJTAG.	R	1
FP	0	FPU implemented. This bit is always zero since the core does not contain a floating point unit.	R	0

5.2.15 *Config2* Register (CP0 Register 16, Select 2)

The *Config2* register is an adjunct to the *Config* register and is reserved to encode additional capabilities information. *Config2* is allocated for showing the configuration of level 2/3 caches. These fields are reset to 0 because L2/L3 caches are not supported by the M4K core. All fields in the *Config2* register are read-only.

Figure 5.16 *Config2* Register Format — Select 2

31	30 0
M	0

Table 5.19 *Config1* Register Field Descriptions — Select 1

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
M	31	This bit is hardwired to ‘1’ to indicate the presence of the Config3 register.	R	1
0	30:0	These bits are reserved.	R	0

5.2.16 *Config3* Register (CP0 Register 16, Select 3)

The *Config3* register encodes additional capabilities. All fields in the *Config3* register are read-only.

Figure 5.17 shows the format of the *Config3* register; Table 5.20 describes the *Config3* register fields.

Figure 5.17 Config3 Register Format

31	30 9	8	7	6	5	4	3 2	1	0
M	0 000 0000 0000 0000 0000 0000 0	ITL	0	VEIC	VInt	SP	0	SM	TL

Table 5.20 Config3 Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
M	31	This bit is reserved to indicate that a Config4 register is present. With the current architectural definition, this bit should always read as a 0.	R	0
0	30:9,7,3:2	Must be written as zeros; returns zeros on read	0	0
ITL	8	Indicates that IFlowTrace hardware is present	R	Preset
VEIC	6	Support for an external interrupt controller is implemented. **Encoding** / **Meaning** 0 / Support for EIC interrupt mode is not implemented 1 / Support for EIC interrupt mode is implemented The value of this bit is set by the static input, SI_EICPresent. This allows external logic to communicate whether an external interrupt controller is attached to the processor or not.	R	Externally Set
VInt	5	Vectored interrupts implemented. This bit indicates whether vectored interrupts are implemented. **Encoding** / **Meaning** 0 / Vector interrupts are not implemented 1 / Vectored interrupts are implemented On the M4K core, this bit is always a 1 since vectored interrupts are implemented.	R	1

Table 5.20 Config3 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>SP</td><td>4</td><td>Small (1KByte) page support is implemented, and the <i>PageGrain</i> register exists. This bit will always read as 0 on the M4K core, since no TLB is present.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Small page support is not implemented</td></tr><tr><td>1</td><td>Small page support is implemented</td></tr></table></td><td>R</td><td>0</td></tr>
<tr><td>SM</td><td>1</td><td>SmartMIPS™ ASE implemented. This bit indicates whether the SmartMIPS ASE is implemented. Since SmartMIPS is not present on the M4K core, this bit will always be 0.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>SmartMIPS ASE is not implemented</td></tr><tr><td>1</td><td>SmartMIPS ASE is implemented</td></tr></table></td><td>R</td><td>0</td></tr>
<tr><td>TL</td><td>0</td><td>Trace Logic implemented. This bit indicates whether PC or data trace is implemented..<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Trace logic is not implemented</td></tr><tr><td>1</td><td>Trace logic is implemented</td></tr></table></td><td>R</td><td>Preset</td></tr>
</table>

5.2.17 *Debug* Register (CP0 Register 23, Select 0)

The *Debug* register is used to control the debug exception and provide information about the cause of the debug exception and when re-entering at the debug exception vector due to a normal exception in debug mode. 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.

Only the *DM* bit and the *EJTAGver* field are valid when read from non-debug mode; the values of all other bits and fields are UNPREDICTABLE. Operation of the processor is UNDEFINED if the *Debug* register is written from non-debug mode.

Some of the bits and fields are only updated on debug exceptions and/or exceptions in debug mode, as shown below:

- *DSS*, *DBp*, *DDBL*, *DDBS*, *DIB*, *DINT*, *DIBImpr*, *DDBLImpr*, *DDBSImpr* are updated on both debug exceptions and on exceptions in debug modes
- *DExcCode* is updated on exceptions in debug mode, and is undefined after a debug exception
- *Halt* and *Doze* are updated on a debug exception, and are undefined after an exception in debug mode
- *DBD* is updated on both debug and on exceptions in debug modes

All bits and fields are undefined when read from normal mode, except those explicitly described to be defined, e.g. *EJTAGver* and *DM*.

Figure 5.18 *Debug* Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19
DBD	DM	NoDCR	LSNM	Doze	Halt	CountDM	IBusEP	MCheckP	CacheEP	DBusEP	IEXI	DDB-SImpr

18	17 15	14 10	9	8	7	6	5	4	3	2	1	0
DDBLImpr	Ver	DExcCode	NoSSt	SSt	R	DIBImpr	DINT	DIB	DDBS	DDBL	DBp	DSS

Table 5.21 *Debug* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
DBD	31	Indicates whether the last debug exception or exception in debug mode, occurred in a branch delay slot: 0: Not in delay slot 1: In delay slot	R	Undefined
DM	30	Indicates that the processor is operating in debug mode: 0: Processor is operating in non-debug mode 1: Processor is operating in debug mode	R	0
NoDCR	29	Indicates whether the dseg memory segment is present and the Debug Control Register is accessible: 0: dseg is present 1: No dseg present	R	0
LSNM	28	Controls access of load/store between dseg and main memory: 0: Load/stores in dseg address range goes to dseg. 1: Load/stores in dseg address range goes to main memory.	R/W	0
Doze	27	Indicates that the processor was in any kind of low power mode when a debug exception occurred: 0: Processor not in low power mode when debug exception occurred 1: Processor in low power mode when debug exception occurred	R	Undefined
Halt	26	Indicates that the internal system bus clock was stopped when the debug exception occurred: 0: Internal system bus clock stopped 1: Internal system bus clock running	R	Undefined
CountDM	25	Indicates the Count register behavior in debug mode. 0: Count register stopped in debug mode 1: Count register is running in debug mode	R/W	1
IBusEP	24	Instruction fetch Bus Error exception Pending. Set when an instruction fetch bus error event occurs or if a 1 is written to the bit by software. Cleared when a Bus Error exception on instruction fetch is taken by the processor, and by reset. If IBusEP is set when IEXI is cleared, a Bus Error exception on instruction fetch is taken by the processor, and IBusEP is cleared.	R/W1	0

Table 5.21 *Debug* Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
MCheckP	23	Indicates that an imprecise Machine Check exception is pending. All Machine Check exceptions are precise on the M4K processor so this bit will always read as 0.	R	0
CacheEP	22	Indicates that an imprecise Cache Error is pending. Cache Errors cannot be taken by the M4K core so this bit will always read as 0	R	0
DBusEP	21	Data access Bus Error exception Pending. Covers imprecise bus errors on data access, similar to behavior of IBusEP for imprecise bus errors on an instruction fetch.	R/W1	0
IEXI	20	Imprecise Error eXception Inhibit controls exceptions taken due to imprecise error indications. Set when the processor takes a debug exception or exception in debug mode. Cleared by execution of the DERET instruction; otherwise modifiable by debug mode software. When IEXI is set, the imprecise error exception from a bus error on an instruction fetch or data access, cache error, or machine check is inhibited and deferred until the bit is cleared.	R/W	0
DDBSImpr	19	Indicates that an imprecise Debug Data Break Store exception was taken. Imprecise data breaks only occur on complex breakpoints.	R	Undefined
DDBLImpr	18	Indicates that an imprecise Debug Data Break Load exception was taken. Imprecise data breaks only occur on complex breakpoints.	R	Undefined
Ver	17:15	EJTAG version.	R	010
DExcCode	14:10	Indicates the cause of the latest exception in debug mode. The field is encoded as the ExcCode field in the Cause register for those normal exceptions that may occur in debug mode. Value is undefined after a debug exception.	R	Undefined
NoSST	9	Indicates whether the single-step feature controllable by the SSt bit is available in this implementation: 0: Single-step feature available 1: No single-step feature available	R	0
SSt	8	Controls if debug single step exception is enabled: 0: No debug single-step exception enabled 1: Debug single step exception enabled	R/W	0
R	7	Reserved. Must be written as zeros; returns zeros on reads.	R	0
DIBImpr	6	Indicates that an Imprecise debug instruction break exception occurred (due to a complex breakpoint). Cleared on exception in debug mode.	R	Undefined
DINT	5	Indicates that a debug interrupt exception occurred. Cleared on exception in debug mode. 0: No debug interrupt exception 1: Debug interrupt exception	R	Undefined

Table 5.21 *Debug* Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
DIB	4	Indicates that a debug instruction break exception occurred. Cleared on exception in debug mode. 0: No debug instruction exception 1: Debug instruction exception	R	Undefined
DDBS	3	Indicates that a debug data break exception occurred on a store. Cleared on exception in debug mode. 0: No debug data exception on a store 1: Debug instruction exception on a store	R	Undefined
DDBL	2	Indicates that a debug data break exception occurred on a load. Cleared on exception in debug mode. 0: No debug data exception on a load 1: Debug instruction exception on a load	R	Undefined
DBp	1	Indicates that a debug software breakpoint exception occurred. Cleared on exception in debug mode. 0: No debug software breakpoint exception 1: Debug software breakpoint exception	R	Undefined
DSS	0	Indicates that a debug single-step exception occurred. Cleared on exception in debug mode. 0: No debug single-step exception 1: Debug single-step exception	R	Undefined

5.2.18 *Trace Control* Register (CP0 Register 23, Select 1)

The *TraceControl* register configuration is shown below. Note the special behavior of the ASID_M, ASID, and G fields for the M4K processor.

This register is only implemented if the EJTAG Trace capability is present.

Figure 5.19 *TraceControl* Register Format

31	30	29 28	27	26	25	24	23	22	21	20 13	12 5	4	3 1	0
TS	UT	0	TB	IO	D	E	K	S	U	ASID_M	ASID	G	Mode	On

Table 5.22 *TraceControl* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
TS	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 this software control register.	R/W	0

Table 5.22 *TraceControl* Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bits			
UT	30	This bit is used to indicate 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. The actual triggering of a user trace record happens on a write to the *UserTraceData* register.	R/W	Undefined
0	29:28	Reserved for future use; Must be written as zero; returns zero on read.	0	0
TB	27	Trace All Branch. When set to one, this tells the processor to trace the PC value for all taken branches, not just the ones whose branch target address is statically unpredictable.	R/W	Undefined
IO	26	Inhibit Overflow. This signal is used to indicate to the core trace logic that slow but complete tracing is desired. When set to one, the core tracing logic does not allow a FIFO overflow or 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
D	25	When set to one, this enables tracing in Debug Mode (see 8.9.1 “Processor Modes” on page 175). For trace to be enabled in Debug mode, the On bit must be one. When set to zero, trace is disabled in Debug Mode, irrespective of other bits.	R/W	Undefined
E	24	When set to one, this enables tracing in Exception Mode (see 8.9.1 “Processor Modes” on page 175). For trace to be enabled in Exception mode, the On bit must be one. When set to zero, trace is disabled in Exception Mode, irrespective of other bits.	R/W	Undefined
K	23	When set to one, this enables tracing in Kernel Mode (see 8.9.1 “Processor Modes” on page 175). For trace to be enabled in Kernel mode, the On bit must be one. When set to zero, trace is disabled in Kernel Mode, irrespective of other bits.	R/W	Undefined
0	22	This bit is reserved. Must be written as zero; returns zero on read.	0	0
U	21	When set to one, this enables tracing in User Mode (see 8.9.1 “Processor Modes” on page 175). For trace to be enabled in User mode, the On bit must be one. When set to zero, trace is disabled in User Mode, irrespective of other bits.	R/W	Undefined
ASID_M	20:13	In the M4K core where ASID is not supported, this field is ignored on write and returns zero on read.	R	0

Table 5.22 *TraceControl* Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>ASID</td><td>12:5</td><td>In the M4K core where ASID is not supported, this field is ignored on write and returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>G</td><td>4</td><td>In the M4K core where ASID is not supported, this field is ignored on write and returns 1 on read. This causes all match equations to work correctly in the absence of an ASID.</td><td>R</td><td>1</td></tr>
<tr><td>Mode</td><td>3:1</td><td>These three bits control the trace mode function.
<table>
<tr><th>Mode</th><th>Trace Mode</th></tr>
<tr><td>000</td><td>Trace PC</td></tr>
<tr><td>001</td><td>Trace PC and load address</td></tr>
<tr><td>010</td><td>Trace PC and store address</td></tr>
<tr><td>011</td><td>Trace PC and both load/store addresses</td></tr>
<tr><td>100</td><td>Trace PC and load data</td></tr>
<tr><td>101</td><td>Trace PC and load address and data</td></tr>
<tr><td>110</td><td>Trace PC and store address and data</td></tr>
<tr><td>111</td><td>Trace PC and both load/store address and data</td></tr>
</table>
The TraceControl2$_{\text{ValidModes}}$ field determines which of these encodings are supported by the processor. The operation of the processor is <b>UNPREDICTABLE</b> if this field is set to a value which is not supported by the processor.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>On</td><td>0</td><td>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.</td><td>R/W</td><td>0</td></tr>
</table>

5.2.19 *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 8.11 “Trace Control Block (TCB) Registers (Hardware Control)” on page 180). 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 EJTAG Trace capability is present.

Figure 5.20 *TraceControl2* Register Format

31 7	6 5	4	3	2 0
0	ValidModes	TBI	TBU	SyP

Table 5.23 *TraceControl2* Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>0</td><td>31:5</td><td>Reserved for future use; Must be written as zero; returns zero on read.</td><td>0</td><td>0</td></tr>
<tr><td>ValidModes</td><td>6:5</td><td>This field specifies the type of tracing that is supported by the processor, as follows:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<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>
</table></td><td>R</td><td>10</td></tr>
<tr><td>TBI</td><td>4</td><td>This bit indicates how many trace buffers are implemented by the TCB, as follows:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<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 trace is currently written.</td></tr>
</table></td><td>R</td><td>Per implementation</td></tr>
<tr><td>TBU</td><td>3</td><td>This bit denotes to which trace buffer the trace is currently being written and is used to select the appropriate interpretation of the *TraceControl2*$_{SyP}$ field.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<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>
</table></td><td>R</td><td>Undefined</td></tr>
</table>

Table 5.23 *TraceControl2* Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>SyP</td><td>2:0</td><td>Used to indicate the synchronization period.
The period (in cycles) between which the periodic synchronization information is to be sent is defined as shown below, for both when the trace buffer is on-chip and off-chip.
<table>
<tr><th>SyP</th><th>On-chip</th><th>Off-chip</th></tr>
<tr><td>000</td><td>2^{2}</td><td>2^{7}</td></tr>
<tr><td>001</td><td>2^{3}</td><td>2^{8}</td></tr>
<tr><td>010</td><td>2^{4}</td><td>2^{9}</td></tr>
<tr><td>011</td><td>2^{5}</td><td>2^{10}</td></tr>
<tr><td>100</td><td>2^{6}</td><td>2^{11}</td></tr>
<tr><td>101</td><td>2^{7}</td><td>2^{12}</td></tr>
<tr><td>110</td><td>2^{8}</td><td>2^{13}</td></tr>
<tr><td>111</td><td>2^{9}</td><td>2^{14}</td></tr>
</table>
The “On-chip” column value is used when the trace data is being written to an on-chip trace buffer (e.g, *TraceControl2*$_{\text{TBU}}$ = 0). Conversely, the “Off-chip” column is used when the trace data is being written to an off-chip trace buffer (e.g, *TraceControl2*$_{\text{TBU}}$ = 1).</td><td>R</td><td>Undefined</td></tr>
</table>

5.2.20 *User Trace Data* Register (CP0 Register 23, Select 3)

A software write to any bits in the *UserTraceData* register will trigger a trace record to be written indicating a type 1 or type 2 user format. The type is based on the *UT* bit in the *TraceControl* register. This register cannot be written in consecutive cycles. The trace output data is UNPREDICTABLE if this register is written in consecutive cycles.

This register is only implemented if the EJTAG Trace capability is present.

Figure 5.21 *User Trace Data* Register Format \

31	0
Data	

Table 5.24 *UserTraceData* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
Data	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

5.2.21 *TraceBPC* Register (CP0 Register 23, Select 4)

This register is used to control start and stop of tracing using an EJTAG Hardware breakpoint. The 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 EJTAG Trace capability are present.

Figure 5.22 *Trace BPC* Register Format

31	30 … 18	17 16	15	14 … 6	5 … 0
DE	0	DBPOn	IE	0	IBPOn

Table 5.25 *TraceBPC* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bits**			
DE	31	Used to specify whether the trigger signal from EJTAG data breakpoint should trigger tracing functions or not: 0: disables trigger signals from data breakpoints 1: enables trigger signals from data breakpoints	R/W	0
0	30:18	Reserved	0	0
DBPOn	17:16	Each of the 2 bits corresponds to the 2 possible EJTAG hardware data breakpoints that may be implemented. For example, bit 16 corresponds to the first data breakpoint. If 2 data breakpoints are present in the EJTAG implementation, then they correspond to bits 16 and 17. The rest are always ignored by the tracing logic since they will never be triggered. A value of one for each bit implies that a trigger from the corresponding data breakpoint should start tracing. And a value of zero implies that tracing should be turned off with the trigger signal.	R/W	0
IE	15	Used to specify whether the trigger signal from EJTAG instruction breakpoint should trigger tracing functions or not: 0: disables trigger signals from instruction breakpoints 1: enables trigger signals from instruction breakpoints	R/W	0
0	14:6	Reserved	0	0
IBPOn	5:0	Each of the 6 bits corresponds to the 6 possible EJTAG hardware instruction breakpoints that may be implemented. Bit 0 corresponds to the first instruction breakpoint, and so on. If only 2 instruction breakpoints are present in the EJTAG implementation, then only bits 0 and 1 are used. The rest are always ignored by the tracing logic since they will never be triggered. A value of one for each bit implies that a trigger from the corresponding instruction breakpoint should start tracing. And a value of zero implies that tracing should be turned off with the trigger signal.	R/W	0

5.2.22 *Debug2* Register (CP0 Register 23, Select 6)

This register holds additional information about Complex Breakpoint exceptions.

This register is only implemented if complex hardware breakpoints are present.

Figure 5.23 *Debug2* Register Format

31 … 4	3	2	1	0
0	Prm	DQ	Tup	PaCo

Table 5.26 *Debug2* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bits**			
0	31:4	Reserved	0	0
Prm	3	Primed - indicates whether a complex breakpoint with an active priming condition was seen on the last debug exception.	R	Undefined
DQ	2	Data Qualified - indicates whether a complex breakpoint with an active data qualfier was seen on the last debug exception.	R	Undefined
Tup	1	Tuple - indicates whether a tuple breakpoint was seen on the last debug exception.	R	Undefined
PaCo	0	Pass Counter - indicates whether a complex breakpoint with an active pass counter was seen on the last debug exception	R	Undefined

5.2.23 Debug Exception Program Counter Register (CP0 Register 24, Select 0)

The Debug Exception Program Counter (*DEPC*) register is a read/write register that contains the address at which processing resumes after a debug exception or debug mode exception has been serviced.

For synchronous (precise) debug and debug mode exceptions, the *DEPC* contains either:

- The virtual address of the instruction that was the direct cause of the debug exception, or
- The virtual address of the immediately preceding branch or jump instruction, when the debug exception causing instruction is in a branch delay slot, and the *Debug Branch Delay* (*DBD*) bit in the *Debug* register is set.

For asynchronous debug exceptions (debug interrupt, complex break), the *DEPC* contains the virtual address of the instruction where execution should resume after the debug handler code is executed.

In processors that implement the MIPS16e ASE, a read of the *DEPC* register (via MFC0) returns the following value in the destination GPR:

$$\mathtt{GPR[rt]} \leftarrow \mathtt{DebugExceptionPC}_{31..1} \;||\; \mathtt{ISAMode}_{0}$$

That is, the upper 31 bits of the debug exception PC are combined with the lower bit of the *ISAMode* field and written to the GPR.

Similarly, a write to the *DEPC* register (via MTC0) takes the value from the GPR and distributes that value to the debug exception PC and the *ISAMode* field, as follows

$$\begin{aligned}&\texttt{DebugExceptionPC} \leftarrow \texttt{GPR[rt]}_{31..1} \mid\mid \texttt{0}\\&\texttt{ISAMode} \leftarrow \texttt{2\#0} \mid\mid \texttt{GPR[rt]}_{0}\end{aligned}$$

That is, the upper 31 bits of the GPR are written to the upper 31 bits of the debug exception PC, and the lower bit of the debug exception PC is cleared. The upper bit of the *ISAMode* field is cleared and the lower bit is loaded from the lower bit of the GPR.

Figure 5.24 *DEPC* Register Format

31	0
DEPC	

Table 5.27 *DEPC* Register Formats

Fields				
Name	Bit(s)	Description	Read/Write	Reset
DEPC	31:0	The DEPC 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 DEPC.	R/W	Undefined

5.2.24 *ErrorEPC* (CP0 Register 30, Select 0)

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

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

- The virtual address of the instruction that caused the exception
- The virtual address of the immediately preceding branch or jump instruction when the error causing instruction is in a branch delay slot

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

In processors that implement the MIPS16e ASE, a read of the *ErrorEPC* register (via MFC0) returns the following value in the destination GPR:

$$\texttt{GPR[rt]} \leftarrow \texttt{ErrorExceptionPC}_{31..1} \mid\mid \texttt{ISAMode}_{0}$$

That is, the upper 31 bits of the error exception PC are combined with the lower bit of the ISAMode field and written to the GPR.

Similarly, a write to the *ErrorEPC* register (via MTC0) takes the value from the GPR and distributes that value to the error exception PC and the *ISAMode* field, as follows

```
ErrprExceptionPC ← GPR[rt]$_{31..1}$ || 0
ISAMode ← 2#0 || GPR[rt]$_{0}$
```

That is, the upper 31 bits of the GPR are written to the upper 31 bits of the error exception PC, and the lower bit of the error exception PC is cleared. The upper bit of the *ISAMode* field is cleared and the lower bit is loaded from the lower bit of the GPR.

Figure 5.25 *ErrorEPC* Register Format

31		0
	ErrorEPC	

Table 5.28 *ErrorEPC* Register Field Description

Fields				
Name	Bit(s)	Description	Read/Write	Reset State
ErrorEPC	31:0	Error Exception Program Counter.	R/W	Undefined

5.2.25 *DeSave* Register (CP0 Register 31, Select 0)

The *Debug Exception Save* (*DeSave*) register is a read/write register that functions as a simple memory location. This register is used by the debug exception handler to save one of the GPRs that is then used to save the rest of the context to a pre-determined memory area (such as in the EJTAG Probe). This register allows the safe debugging of exception handlers and other types of code where the existence of a valid stack for context saving cannot be assumed.

Figure 5.26 *DeSave* Register Format

31		0
	DESAVE	

Table 5.29 *DeSave* Register Field Description

Fields				
Name	Bit(s)	Description	Read/Write	Reset State
DESAVE	31:0	Debug exception save contents.	R/W	Undefined

Chapter 6

Hardware and Software Initialization of the M4K™ Core

A M4K 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 6.1 “Hardware-Initialized Processor State”
- Section 6.2 “Software Initialized Processor State”

6.1 Hardware-Initialized Processor State

A M4K 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. *SI_ColdReset* is asserted after power-up to bring the device into a known state. Soft reset can be forced by asserting the *SI_Reset* pin. This distinction is made for compatibility with other MIPS processors. In practice, both resets are handled identically with the exception of the setting of $Status_{SR}$.

6.1.1 Coprocessor 0 State

Much of the hardware initialization occurs in Coprocessor 0.

- $Status_{BEV}$ - cleared to 1 on Reset/SoftReset
- $Status_{TS}$ - cleared to 0 on Reset/SoftReset
- $Status_{SR}$ - cleared to 0 on Reset, set to 1 on SoftReset
- $Status_{NMI}$ - cleared to 0 on Reset/SoftReset
- $Status_{ERL}$ - set to 1 on Reset/SoftReset
- $Status_{RP}$ - cleared to 0 on Reset/SoftReset
- *Config* fields related to static inputs - set to input value by Reset/SoftReset
- $Config_{K0}$ - set to 010 (uncached) on Reset/SoftReset
- $Config_{KU}$ - set to 010 (uncached) on Reset/SoftReset
- $Config_{K23}$ - set to 010 (uncached) on Reset/SoftReset

- *DebugDM* - cleared to 0 on Reset/SoftReset (unless EJTAGBOOT option is used to boot into DebugMode, see Chapter 8, “EJTAG Debug Support in the M4K™ Core” on page 127 for details)
- $Debug_{LSNM}$ - cleared to 0 on Reset/SoftReset
- $Debug_{IBusEP}$ - cleared to 0 on Reset/SoftReset
- $Debug_{DBusEP}$ - cleared to 0 on Reset/SoftReset
- $Debug_{IEXI}$ - cleared to 0 on Reset/SoftReset
- $Debug_{SSt}$ - cleared to 0 on Reset/SoftReset

6.1.2 Bus State Machines

All pending bus transactions are aborted and the state machines in the SRAM interface unit are reset when a Reset or SoftReset exception is taken.

6.1.3 Static Configuration Inputs

All static configuration inputs should only be changed during Reset.

6.1.4 Fetch Address

Upon Reset/SoftReset, 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.

6.2 Software Initialized Processor State

Software is required to initialize the following parts of the device.

6.2.1 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 in hardware. However, when simulating the operation of the core, unknown values can cause problems. Thus, initializing the register file in the boot code may avoid simulation problems.

6.2.2 Coprocessor 0 State

Miscellaneous COP0 states need to be initialized prior to leaving 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), SW0/1 (Software Interrupts) should be cleared.
- *Config*: Typically, the K0, KU and K23 fields should be set to the desired Cache Coherency Algorithm (CCA) value prior to accessing the corresponding memory regions. But in the M4K core, all CCA values are treated identically, so the hardware reset value of these fields need not be modified.

- *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. The write to compare will also clear any pending Timer Interrupts (Thus, *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 writeable, 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.

Chapter 7

Power Management of the M4K™ Core

A M4K processor coreoffers 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 discussed in the following sections.

- Section 7.1 “Register-Controlled Power Management”
- Section 7.2 “Instruction-Controlled Power Management”

7.1 Register-Controlled Power Management

The RP 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 via the *SI_RP* output signal. Three additional pins, *SI_EX*L, *SI_ERL*, and *EJ_DebugM* support the power management function 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 whether to reduce the clock frequency and place the core into power down mode.

If an interrupt is taken 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. The setting of 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. At this time 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. At this time the external agent can 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. 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.
- 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.

7.2 Instruction-Controlled Power Management

The second mechanism for invoking power down mode is through execution of the WAIT instruction. If the bus is idle at the time the WAIT instruction reaches the M stage of the pipeline 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*, *SI_ColdReset*, and *EJ_DINT*) continue to run. If the bus is not idle at the time the WAIT instruction reaches the M stage, the pipeline stalls until the bus becomes idle, at which time the clocks are stopped. Once 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 part is in this low-power mode, the *SI_SLEEP* signal is asserted to indicate to external agents what the state of the chip is.

Chapter 8

EJTAG Debug Support in the M4K™ Core

The EJTAG debug logic in the M4K processor core provides three optional modules:

1. Hardware breakpoints
2. Test Access Port (TAP) for a dedicated connection to a debug host
3. Tracing of program counter/data address/data value trace to On-chip memory or to a Trace probe

These features are covered in the following sections:

- Section 8.1 “Debug Control Register”
- Section 8.2 “Hardware Breakpoints”
- Section 8.3 “Complex Breakpoint Usage”
- Section 8.4 “Test Access Port (TAP)”
- Section 8.5 “EJTAG TAP Registers”
- Section 8.6 “TAP Processor Accesses”
- Section 8.7 “Trace Mechanisms”
- Section 8.8 “iFlowtrace™ Mechanism”
- Section 8.9 “EJTAG Trace”
- Section 8.10 “PDtrace™ Registers (Software Control)”
- Section 8.11 “Trace Control Block (TCB) Registers (Hardware Control)”
- Section 8.12 “EJTAG Trace Enabling”
- Section 8.13 “TCB Trigger logic”
- Section 8.14 “EJTAG Trace Cycle-by-Cycle Behavior”
- Section 8.15 “TCB On-Chip Trace Memory”

8.1 Debug Control Register

The Debug Control Register (*DCR*) register 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 none, some or all sources for soft reset. The soft reset masking may 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 both hard and soft resets.

Debug Control Register

31 30	29	28 18	17	16	15	14	13 11	10	9	8 6	5	4	3	2	1	0
Res	ENM	Res	DB	IB	IVM	DVM	Res	CBrk	PCS	PCR	PCSe	INTE	NMIE	NMIP	SRE	PE

Table 8.1 *Debug Control Register* Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
Res	31:30	Reserved	R	0
ENM	29	Endianess in Kernel and Debug mode. 0: Little Endian 1: Big Endian	R	Preset
Res	28:18	Reserved	R	0
DB	17	Data Break Implemented. 0: No Data Break feature implemented 1: Data Break feature is implemented	R	Preset
IB	16	Instruction Break Implemented. 0: No Instruction Break feature implemented 1: Instruction Break feature is implemented	R	Preset
IVM	15	Inverted Value Match. Indicates that the data hardware breakpoints (if implemented) support an inverted value match.	R	1
DVM	14	Data Value Match Register. Indicates that a DRSEG mapped register is present that will capture the load data value on precise data value breakpoints.	R	1
Res	13:11	Reserved	R	0
CBrk	10	Indicates that Complex Breakpoint logic is implemented	R	Preset

Table 8.1 *Debug Control Register* Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
PCS	9	Program Counter Sampling implemented. Not supported on M4K core so this bit will read as 0	R	0
PCR	8:6	PC Sampling Rate. Controls how often the program counter is sampled if PC Sampling is implemented	R	0
PCE	5	PC Sampling Enable. Enables sampling of PC if implemented	R	0
INTE	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: 0: Interrupts disabled. 1: Interrupts enabled (depending on other enabling mechanisms).	R/W	1
NMIE	3	Non-Maskable Interrupt Enable for non-debug mode 0: NMI disabled. 1: NMI enabled.	R/W	1
NMIP	2	NMI Pending Indication. 0: No NMI pending. 1: NMI pending.	R	0
SRE	1	Soft Reset Enable This bit allows the system to mask soft resets. The core does not internally mask soft resets. Rather the state of this bit appears on the *EJ_SRstE* external output signal, allowing the system to mask soft resets if desired.	R/W	1
PE	0	Probe Enable This bit reflects the ProbEn bit in the EJTAG Control register. 0: No accesses to dmseg allowed 1: EJTAG probe services accesses to dmseg	R	Same value as ProbEn in ECR (see Table 9-4)

8.2 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 for many aspects, and are thus described in parallel in the following. The term hardware is not generally added to breakpoint, unless required to distinguish it from a software breakpoint.

There are two types of simple hardware breakpoints implemented in the M4K core; Instruction breakpoints and Data breakpoints. The M4K core may also contain a complex breakpoint unit.

A core may be configured with the following breakpoint options:

- No data or instruction breakpoints, without complex break support
- Two instruction and one data breakpoint, without complex break support

- Four instruction and two data breakpoints, without complex break support
- Six instruction and two data breakpoints, with support for complex breaks

8.2.1 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. 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) with the registers for each instruction breakpoint including masking of address. 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.

8.2.2 Features of Data Breakpoint

Data breakpoints occur on load/store transactions. Breakpoints are set on virtual address 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), 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.

8.2.3 Features of Complex Breakpoints

The complex breakpoint unit utilizes the instruction and data breakpoint hardware and looks for more specific matching conditions. There are several different types of enabling that allow more exact breakpoint specification. Tuples add an additional condition to data breakpoints of requiring an instruction breakpoint on the same instructions. Pass counters are counters that decrement each time a matching breakpoint condition is taken. Once the counter reaches 0, the break or trigger effect of the breakpoint is enabled. Priming allows a breakpoint to only be enabled once another trigger condition has been detected. Data qualification allows instruction breakpoints to only be enabled once a corresponding load data triggerpoint has matched both address and data. Data qualified breakpoints are also disabled if a load is executed that matches on the address portion of the triggerpoint, but has a mismatching data value. The complex breakpoint features can be combined to create very complex sequences to match on.

In addition to the breakpoint logic, the complex break unit also includes a Stopwatch Timer block. This counter can be used to measure time spent in various sections. It can either be free-running, or it can be set up to start and stop counting based on a trigger from instruction breakpoints.

8.2.4 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.

8.2.4.1 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. 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 =
          ( <all 1's> == ( $IBMn_{IBM}$ | ~ ( PC ^ $IBAn_{IBA}$ ) )
```

The match indication for instruction breakpoints is always precise, i.e. indicated on the instruction causing the IB_match to be true.

8.2.4.2 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 ) && ! $DBCn_{NoLB}$ ) ||
            ( ( TYPE == store ) && ! $DBCn_{NoSB}$ ) ) &&
          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) 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 =
          ( <all 1's> == ( $DBMn_{DBM}$ | ~ ( ADDR ^ $DBAn_{DBA}$ ) ) ) &&
          ( <all 0's> != ( ~ BAI & BYTELANE ) )
```

The size of $DBCn_{BAI}$ and BYTELANE is 4 bits.

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> == ( $DBCn_{BLM}$ | $DBCn_{BAI}$ | ~ BYTELANE ) )
```

The size of $DBCn_{BLM}$, $DBCn_{BAI}$ and BYTELANE is 4 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 DBC_{IVM} bit inverts the sense of the match - if set, the value match term will be high if the data value is not the same as the data in the *DBVn* register. The endianess 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 endianess.

```
DB_value_match =
    $DBCn_{IVM}$ ^
    (((DATA[7:0] == $DBVn_{DBV[7:0]}$) || ! BYTELANE[0] || $DBCn_{BLM[0]}$ || $DBCn_{BAI[0]}$ ) &&
     ((DATA[15:8] == $DBVn_{DBV[15:8]}$) || ! BYTELANE[1] || $DBCn_{BLM[1]}$ || $DBCn_{BAI[1]}$ ) &&
     ((DATA[23:16] == $DBVn_{DBV[23:16]}$) || ! BYTELANE[2] || $DBCn_{BLM[2]}$ || $DBCn_{BAI[2]}$) &&
     ((DATA[31:24] == $DBVn_{DBV[31:24]}$) || ! BYTELANE[3] || $DBCn_{BLM[3]}$ || $DBCn_{BAI[3]}$ ))
```

The match for a data breakpoint 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.

8.2.5 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.

8.2.5.1 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.

8.2.5.2 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 debug data break exception occurs when a data breakpoint 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.
- A load transaction for a breakpoint with data value compare must occur from the memory system, since the value is required in order to evaluate the breakpoint.

The result of this is that the load or store instruction causing the debug data break exception appears as not executed, with the exception that a load from the memory system does occur for a breakpoint with data value compare, but the register file is not updated by the load.

If both data breakpoints without and with data value compare would match the same transaction and generate a debug exception, then the following rules apply with respect to updating the *BS*[*n*] bits.

- On both a load and store the *BS*[*n*] bits are required to be set for all matching breakpoints without a data value compare.
- On a store the *BS*[*n*] bits are allowed but not required to be set for all matching breakpoints with a data value compare, but either all or none of the *BS*[*n*] bits must be set for these breakpoints.
- On a load then none of the *BS*[*n*] bits for breakpoints with data value compare are allowed to be set, since the load is not allowed to occur due to the debug exception from a breakpoint without a data value compare, and a valid data value is therefore not returned.

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. This re-execution may result in a repeated load from system memory, since the load may have occurred previously in order to evaluate the breakpoint as described above. I/O devices with side effects on loads may not be reaccessible without changing the system behavior. The Load Data Value register was introduced to capture the value that was read and allow debug software to synthesize the load instruction without reaccessing memory. Debug software is responsible for disabling breakpoints when returning to the instruction, otherwise the debug data break exception will reoccur.

8.2.6 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.

The triggerpoint feature can be used to start and stop tracing. See 8.12 “EJTAG Trace Enabling” for details.

8.2.7 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 8.2.

Table 8.2 Addresses for Instruction Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
0x1000	*IBS*	Instruction Breakpoint Status
0x1100 + n * 0x100	*IBAn*	Instruction Breakpoint Address n
0x1108 + n * 0x100	*IBMn*	Instruction Breakpoint Address Mask n
0x1110 + n * 0x100	*IBASIDn*	Instruction Breakpoint ASID n
0x1118 + n * 0x100	*IBCn*	Instruction Breakpoint Control n
0x1120 + n * 0x100	IBCCn	Instruction Breakpoint Complex Control n
0x1128 + n * 0x100	IBPCn	Instruction Breakpoint Pass Counter n
n is breakpoint number in range 0 to 5 (or 3 or 1, depending on the implemented hardware)		

An example of some of the registers; *IBA0* is at offset 0x1100 and *IBC2* is at offset 0x1318.

8.2.7.1 Instruction Breakpoint Status (*IBS*) Register (0x1000)

Compliance Level: Implemented only if instruction breakpoints are implemented.

The Instruction Breakpoint Status (*IBS*) register holds implementation and status information about the instruction breakpoints.

IBS Register Format

31	30	29 28	27 24	23 6	5 0
Res	ASIDsup	Res	BCN	Res	BS

Table 8.3 *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
ASIDsup	30	Indicates that ASID compare is supported in instruction breakpoints. 0: No ASID compare. 1: ASID compare (IBASIDn register implemented).	R	0
Res	29:28	Must be written as zero; returns zero on read.	R	0
BCN	27:24	Number of instruction breakpoints implemented.	R	2, 4, or 6[a]
Res	23:6	Must be written as zero; returns zero on read.	R	0

Table 8.3 *IBS* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
BS	5:0	Break status for breakpoint n is at BS[n], with n from 0 to 5[b]. The bit is set to 1 when the condition for the corresponding breakpoint has matched and *IBCn*$_{TE}$ or *IBCn*$_{BE}$ are set	R/W	Undefined

[a] Based on actual hardware implemented.
[b] In case of fewer than 6 Instruction breakpoints the upper bits become reserved.

8.2.7.2 Instruction Breakpoint Address n (*IBAn*) Register (0x1100 + n * 0x100)

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

IBAn Register Format

31	0
IBA	

Table 8.4 *IBAn* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
IBA	31:0	Instruction breakpoint address for condition.	R/W	Undefined

8.2.7.3 Instruction Breakpoint Address Mask n (*IBMn*) Register (0x1108 + n*0x100)

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. A 1 indicates that the corresponding address bit will not be considered in the match. A mask value of all 0’s would require an exact address match, while a mask value of all 1’s would match on any address.

IBMn Register Format

31	0
IBM	

Table 8.5 *IBMn* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
IBM	31:0	Instruction breakpoint address mask for condition: 0: Corresponding address bit not masked. 1: Corresponding address bit masked.	R/W	Undefined

8.2.7.4 Instruction Breakpoint ASID n (*IBASIDn*) Register (0x1110 + n*0x100)

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. On the M4K processor, this register is reserved and reads as 0.

***IBASIDn* Register Format**

31 8	7 0
Res	ASID

Table 8.6 *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
ASID	7:0	Instruction breakpoint ASID value for a compare.	R	0

8.2.7.5 Instruction Breakpoint Control n (*IBCn*) Register (0x1118 + n*0x100)

Compliance Level: Implemented only for implemented instruction breakpoints.

The Instruction Breakpoint Control n (*IBCn*) register controls the setup of instruction breakpoint n.

***IBCn* Register Format**

31 24	23	22 3	2	1	0
Res	ASIDuse	Res	TE	Res	BE

Table 8.7 *IBCn* 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
ASIDuse	23	Use ASID value in compare for instruction breakpoint n: 0: Don't use ASID value in compare 1: Use ASID value in compare	R	0
Res	22:3	Must be written as zero; returns zero on read.	R	0
TE	2	Use instruction breakpoint n as triggerpoint: 0: Don't use it as triggerpoint 1: Use it as triggerpoint	R/W	0
Res	1	Must be written as zero; returns zero on read.	R	0
BE	0	Use instruction breakpoint n as breakpoint: 0: Don't use it as breakpoint 1: Use it as breakpoint	R/W	0

8.2.7.6 Instruction Breakpoint Complex Control n (*IBCCn*) Register (0x1120 + n*0x100)

Compliance Level: Implemented only if complex breakpoints are implemented and only for implemented instruction breakpoints.

The Instruction Breakpoint Complex Control n (*IBCCn*) register controls the complex break conditions for instruction breakpoint n.

IBCCn **Register Format**

31 14	13 10	9	8 5	4	3 2 1 0
Res	PrCnd	CBE	DBrkNum	Q	Res

Table 8.8 *IBCCn* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
Res	31:14, 9, 3:0	Must be written as zero; returns zero on read.	R	0
PrCnd	13:12	Upper bits of priming condition for I breakpoint n. M4K only supports 4 priming conditions so the upper 2 bits are read only as 0	R	0
PrCnd	11:10	Priming condition for I Breakpoint n. 00 - Bypass, no priming needed Other - vary depending on the break number, refer to Table 8.10 for mapping	R/W	0
CBE	9	Complex Break Enable - enables this breakpoint for use in a complex sequence - as a priming condition for another breakpoint, to start or stop the stopwatch timer, or as part of a tuple breakpoint.	R/W	0
DBrkNum	8:5	Indicates which data breakpoint channel is used to qualify this instruction breakpoint	R	IBCC0..2 - 0 IBCC3..6 - 1
Q	4	Qualify this breakpoint based on the data breakpoint indicated in DBrkNum. 0 - Not dependent on qualification 1 - Breakpoint must be qualified to be taken	R/W	0

8.2.7.7 Instruction Breakpoint Pass Counter n (*IBPCn*) Register (0x1128 + n*0x100)

Compliance Level: Implemented only if complex breakpoints are implemented and only for implemented instruction breakpoints.

The Instruction Breakpoint Pass Counter n (*IBPCn*) register controls the pass counter associated with instruction breakpoint n.

If complex breakpoints are implemented, there will be an 8b pass counter for each of the instruction breakpoints on the M4K core.

IBPCn **Register Format**

31 8	7 0
0	PassCnt

Table 8.9 *IBPCn* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
0	31:8	Ignored on write, returns zero on read.	R	0
PassCnt	7:0	Prevents a break/trigger action until the matching conditions on breakpoint n have been seen this number of times. Each time the matching condition is seen, this value will be decremented by 1. Once the value reaches 0, subsequent matches will cause a break or trigger as requested and the counter will stay at 0. The break or trigger action is imprecise if the PassCnt register was last written to a non-zero value. It will remain imprecise until this register is written to 0 by software. The instruction pass counter should not be set on instruction breakpoints that are being used as part of a tuple breakpoint.	R/W	0

8.2.8 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 8.10.

Table 8.10 Addresses for Data Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
0x2000	*DBS*	Data Breakpoint Status
0x2100 + 0x100 * n	*DBAn*	Data Breakpoint Address n
0x2108 + 0x100 * n	*DBMn*	Data Breakpoint Address Mask n
0x2110 + 0x100 * n	*DBASIDn*	Data Breakpoint ASID n
0x2118 + 0x100 * n	*DBCn*	Data Breakpoint Control n
0x2120 + 0x100 * n	*DBVn*	Data Breakpoint Value n
0x2128 + 0x100 * n	DBCCn	Data Breakpoint Complex Control n
0x2130 + 0x100 * n	DBPCn	Data Breakpoint Pass Counter n
0x2ff0	DVM	Data Value Match Register
n is breakpoint number as 0 or 1 (or just 0, depending on the implemented hardware)		

An example of some of the registers; *DBM0* is at offset 0x2108 and *DBV1* is at offset 0x2220.

8.2.8.1 Data Breakpoint Status (*DBS*) Register (0x2000)

Compliance Level: Implemented if data breakpoints are implemented.

The Data Breakpoint Status (*DBS*) register holds implementation and status information about the data breakpoints.

DBS **Register Format**

31	30	29 28	27 24	23 2	1 0
Res	ASIDsup	Res	BCN	Res	BS

Table 8.11 *DBS* Register Field Descriptions

Fields				
Name	Bit(s)	Description	Read/Write	Reset State
Res	31	Must be written as zero; returns zero on read.	R	0
ASID	30	Indicates that ASID compares are supported in data breakpoints. 0: Not supported 1: Supported	R	0
Res	29:28	Must be written as zero; returns zero on read.	R	0
BCN	27:24	Number of data breakpoints implemented.	R	2 or 1[a]
Res	23:2	Must be written as zero; returns zero on read.	R	0
BS	1:0	Break status for breakpoint n is at BS[n], with n from 0 to 1[b]. The bit is set to 1 when the condition for the corresponding breakpoint has matched.	R/W0	Undefined

[a] Based on actual hardware implemented.
[b] In case of only 1 data breakpoint bit 1 become reserved.

8.2.8.2 Data Breakpoint Address n (*DBAn*) Register (0x2100 + 0x100 * n)

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.

DBAn **Register Format**

31 0
DBA

Table 8.12 *DBAn* Register Field Descriptions

Fields				
Name	Bit(s)	Description	Read/Write	Reset State
DBA	31:0	Data breakpoint address for condition.	R/W	Undefined

8.2.8.3 Data Breakpoint Address Mask n (*DBMn*) Register (0x2108 + 0x100 * n)

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. A 1 indicates that the corresponding address bit will not be considered in the match. A mask value of all 0's would require an exact address match, while a mask value of all 1's would match on any address.

DBMn **Register Format**

31 0
DBM

Table 8.13 *DBMn* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
DBM	31:0	Data breakpoint address mask for condition: 0: Corresponding address bit not masked 1: Corresponding address bit masked	R/W	Undefined

8.2.8.4 Data Breakpoint ASID n (*DBASIDn*) Register (0x2110 + 0x100 * n)

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. On the M4K processor, this register is reserved and reads as 0.

DBASIDn **Register Format**

31 8	7 0
Res	ASID

Table 8.14 *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	0

8.2.8.5 Data Breakpoint Control n (*DBCn*) Register (0x2118 + 0x100 * n)

Compliance Level: Implemented only for implemented data breakpoints.

The Data Breakpoint Control n (*DBCn*) register controls the setup of data breakpoint n.

DBCn **Register Format**

31 24	23	22 18	17 14	13	12	11 8	7 4	3	2	1	0
Re	ASIDuse	Res	BAI	NoSB	NoLB	Res	BLM	Res	TE	IVM	BE

Table 8.15 *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: 0: Don't use ASID value in compare 1: Use ASID value in compare	R	0
Res	22:18	Must be written as zero; returns zero on reads.	R	0
BAI	17:14	Byte access ignore controls ignore of access to a specific byte. BAI[0] ignores access to byte at bits [7:0] of the data bus, BAI[1] ignores access to byte at bits [15:8], etc. 0: Condition depends on access to corresponding byte 1: Access for corresponding byte is ignored	R/W	Undefined
NoSB	13	Controls if condition for data breakpoint is not fulfilled on a store transaction: 0: Condition may be fulfilled on store transaction 1: Condition is never fulfilled on store transaction	R/W	Undefined
NoLB	12	Controls if condition for data breakpoint is not fulfilled on a load transaction: 0: Condition may be fulfilled on load transaction 1: Condition is never fulfilled on load transaction	R/W	Undefined
Res	11:8	Must be written as zero; returns zero on reads.	R	0
BLM	7:4	Byte lane mask for value compare on data breakpoint. BLM[0] masks byte at bits [7:0] of the data bus, BLM[1] masks byte at bits [15:8], etc.: 0: Compare corresponding byte lane 1: Mask corresponding byte lane	R/W	Undefined
Res	3	Must be written as zero; returns zero on reads.	R	0
TE	2	Use data breakpoint n as triggerpoint: 0: Don't use it as triggerpoint 1: Use it as triggerpoint	R/W	0
IVM	1	Invert Value Match: When set, the data value compare will be inverted - a break or trigger will be taken if the value does not match the specified value	R/W	0
BE	0	Use data breakpoint n as breakpoint: 0: Don't use it as breakpoint 1: Use it as breakpoint	R/W	0

8.2.8.6 Data Breakpoint Value n (*DBVn*) Register (0x2120 + 0x100 * n)

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.

***DBVn* Register Format**

31 0
DBV

Table 8.16 *DBVn* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
DBV	31:0	Data breakpoint value for condition.	R/W	Undefined

8.2.8.7 Data Breakpoint Complex Control n (*DBCCn*) Register (0x2128 + n*0x100)

Compliance Level: Implemented only if complex breakpoints are implemented and only for implemented data breakpoints.

The Data Breakpoint Complex Control n (*DBCCn*) register controls the complex break conditions for data breakpoint n.

***DBCCn* Register Format**

31 20	19 16	15	14	13 10	9	8 5	4	3 2 1 0
Res	TIBrkNum	TUP	R	PrCnd	CBE	DBrkNum	Q	Res

Table 8.17 *DBCCn* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
Res	31:14, 9, 3:0	Must be written as zero; returns zero on read.	R	0
TIBrkNum	19:16	Tuple Instruction Break Number - Indicates which instruction breakpoint will be paired with this data breakpoint to form a tuple breakpoint	R	DBCC0 - 0 DBCC1 - 3
TUP	15	Tuple Enable - qualify this data breakpoint with a match on the TIBrkNum instruction breakpoint on the same instruction.	R/W	0
PrCnd	13:12	Upper bits of priming condition for D breakpoint n. M4K only supports 4 priming conditions so the upper 2 bits are read only as 0	R	0
PrCnd	11:10	Priming condition for D Breakpoint n. 00 - Bypass, no priming needed Other - vary depending on the break number, refer to Table 8.20 for mapping	R/W	0
CBE	9	Complex Break Enable - enables this breakpoint for use as a priming or qualifying condition for another breakpoint.	R/W	0

Table 8.17 *DBCCn* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
DQBrkNum	8:5	Indicates which data breakpoint channel is used to qualify this data breakpoint Data qualification of data breakpoints is not supported on a M4K core and this field will read as 0 and cannot be written.	R	0
DQ	4	Qualify this breakpoint based on the data breakpoint indicated in DBrkNum. Data qualification of data breakpoints is not supported on a M4K core and this field will read as 0 and cannot be written.	R	0

8.2.8.8 Data Breakpoint Pass Counter n (*DBPCn*) Register (0x2130 + n*0x100)

Compliance Level: Implemented only if complex breakpoints are implemented and only for implemented data breakpoints.

The Data Breakpoint Pass Counter n (*DBPCn*) register controls the pass counter associated with data breakpoint n.

If complex breakpoints are implemented, there will be an 16b pass counter for each of the data breakpoints on the M4K core.

DBPCn Register Format

31 16	15 0
0	PassCnt

Table 8.18 *DBPCn* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
0	31:16	Ignored on write, returns zero on read.	R	0
PassCnt	15:0	Prevents a break/trigger action until the matching conditions on data breakpoint n have been seen this number of times. Each time the matching condition is seen, this value will be decremented by 1. Once the value reaches 0, subsequent matches will cause a break or trigger as requested and the counter will stay at 0. The break or trigger action is imprecise if the PassCnt register was last written to a non-zero value. It will remain imprecise until this register is written to 0 by software.	R/W	0

8.2.8.9 Data Value Match (*DVM*) Register (0x2ffo)

Compliance Level: Implemented only if data breakpoints are implemented.

The Data Value Match (DVM) register captures the data value of a load that takes a precise data value breakpoint. This allows debug software to synthesize the load instruction without reexecuting it in case it is to a system register that has destructive reads.

DVM **Register Format**

31 ... 0
LDV

Table 8.19 *DVM* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
LDV	31:0	Load data value for the last precise load data value breakpoint taken	R	Undefined

8.2.9 Complex Breakpoint Registers

The registers for complex 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 8.20.

Table 8.20 Addresses for Complex Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
0x1120 + 0x100 * n	IBCCn	Instruction Breakpoint Complex Control n - described above with instruction breakpoint registers
0x1128 + 0x100 * n	*IBPCn*	Instruction Breakpoint Pass Counter n - described above with instruction breakpoint registers
0x2128 + 0x100 * n	DBCCn	Data Breakpoint Complex Control n - described above with data breakpoint registers
0x2130 + 0x100 * n	*DBPCn*	Data Breakpoint Pass Counter n - described above with data breakpoint registers
0x8000	*CBTControl*	Complex Break and Triggerpoint Control - indicates which of the complex breakpoint features are implemented
0x8300 + 0x20 * n	*PrCndAIn*	Prime Condition Register A for Instruction breakpoint n
0x84e0 + 0x20 * n	PrCndADn	Prime Condition Register A for Data breakpoint n
0x8900	STCtl	Stopwatch Timer Control
0x8908	STCnt	Stopwatch Timer Count
n is breakpoint number from 0 to 5 (range dependent on implemented hardware)		

8.2.9.1 Complex Break and Trigger Control (CBTC) Register (0x8000)

Compliance Level: Implemented only if complex breakpoints are implemented.

The CBTC register contains configuration bits that indicate which features of complex break are implemented as well as a control bit for the stopwatch timer. On a M4K core, if complex break is implemented, all of the separate features will be present.

***CBTC* Register Format**

31 9	8	7 5	4	3	2	1	0
Res	STMode	Res	STP	PP	DQP	TP	PCP

Table 8.21 *CBTC* Register Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read/Write**	**Reset State**
Res	31:9, 7:5	Reserved	R	0
STMode	8	Stopwatch Timer Mode: controls whether the stopwatch timer is free-running or controlled by triggerpoints 0 - free-running 1 - started and stopped by instruction triggers	R/W	1
STP	4	Stopwatch Timer Present - indicates whether stopwatch timer is implemented.	R	1
PP	3	Priming Present - indicates whether primed breakpoints are supported	R	1
DQP	2	Data Qualifiy Present - indicates whether data qualified breakpoints are supported.	R	1
TP	1	Tuple Present - indicates whether any tuple breakpoints are implemented	R	1
PCP	0	Pass Counters Present - indicates whether any breakpoints have pass counters associated with them	R	1

8.2.9.2 Priming Condition A (*PrCndAI/Dn*) Registers

Compliance Level: Implemented if complex breakpoints are implemented.

The Prime Condition registers hold implementation specific information about which triggerpoints are used for the priming conditions for each breakpoint register. On a M4K core, these connections are predetermined and these registers are read-only.

The architecture allows for up to 16 priming conditions to be specified and there can be up to 4 priming condition registers per breakpoint (A/B/C/D). A M4K core only allows for 4 priming conditions and thus only implements the PrCndA registers. The general description is shown in Table 8.22. The actual priming conditions for each of the breakpoints are shown in Table 8.23.

***PrCndA* Register Format**

31 24	23 16	15 8	7 0
Cond3	Cond2	Cond1	Cond0

Table 8.22 *PrCndA* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
CondN	31:24 23:16 15:8 7:0	Specifies which triggerpoint is connected to priming condition 3, 2, 1, or 0[a] for the current breakpoint.	R	Preset
	31:30 23:22 15:14 7:6	Reserved	R	0
	29:28 21:20 13:12 5:4	Trigger type 00 - Special/Bypass 01 - Instruction 10 - Data 11 - Reserved	R	Preset
	27:24 19:16 11:8 3:0	Break Number, 0-14	R	Preset

[a] Condition 0 is always Bypass and will read as 8’b0

Table 8.23 Priming Conditions and Register Values

Break	Cond0	Cond1	Cond2	Cond3	PrCndA Value	drseg offset
Inst0	Bypass	Data0	Inst1	Inst2	0x1211_2000	0x8300
Inst1	Bypass	Data0	Inst0	Inst2	0x1210_2000	0x8320
Inst2	Bypass	Data0	Inst0	Inst1	0x1110_2000	0x8340
Inst3	Bypass	Data1	Inst4	Inst5	0x1514_2100	0x8360
Inst4	Bypass	Data1	Inst3	Inst5	0x1513_2100	0x8380
Inst5	Bypass	Data1	Inst3	Inst4	0x1413_2100	0x83a0
Data0	Bypass	Inst0	Inst1	Inst2	0x1211_1000	0x84e0
Data1	Bypass	Inst3	Inst4	Inst5	0x1514_1300	0x8500

8.2.9.3 Stopwatch Timer Control (*STCtl*) Register (0x8900)

Compliance Level: Implemented if stopwatch timer is implemented.

The Stopwatch Timer Control (STCtl) register gives configuration information about how the stopwatch timer register is controlled. On a M4K core, the break channels that control the stopwatch timer are fixed and this register is read-only.

***STCtl* Register Format**

31 18	17 14	13 10	9	8 5	4 1	0
Res	StopChan1	StartChan1	En1	StopChan0	StartChan0	En0

Table 8.24 *STCtl* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
Res	31:18	Must be written as zero; returns zero on read.	R	0
StopChan1	17:14	Indicates the instruction breakpoint channel that will stop the counter if the timer is under pair1 breakpoint control	R	0
StartChan1	13:10	Indicates the instruction breakpoint channel that will start the counter if the timer is under pair1 breakpoint control	R	0
En1	9	Enables the second pair (pair1) of breakpoint registers to control the timer when under breakpoint control. If the stopwatch timer is configured to be under breakpoint control (by setting *CBTControl*$_{STM}$)and this bit is set, the breakpoints indicated in the StartChan1 and StopChan1 fields will control the timer. The M4K core only supports 1 pair of stopwatch control breakpoints so this field is not writeable and will read as 0	R	0
StopChan0	8:5	Indicates the instruction breakpoint channel that will stop the counter if the timer is under pair0 breakpoint control	R	0x4
StartChan0	4:1	Indicates the instruction breakpoint channel that will start the counter if the timer is under pair0 breakpoint control	R	0x1
En0	0	Enables the first pair (pair0) of breakpoint registers to control the timer when under breakpoint control. If the stopwatch timer is configured to be under breakpoint control (by setting *CBTControl*$_{STM}$)and this bit is set, the breakpoints indicated in the StartChan0 and StopChan0 fields will control the timer. The M4K core only supports 1 pair of stopwatch control breakpoints so this field is not writeable and will read as 1	R	1

8.2.9.4 Stopwatch Timer Count (*STCnt*) Register (0x8908)

Compliance Level: Implemented if stopwatch timer is implemented.

The Stopwatch Timer Count (STCnt) register is the count value for the stopwatch timer.

STCnt Register Format

31	0
Count	

Table 8.25 _STCtl_ Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
Count	31:0	Current counter value	R/W	0

8.3 Complex Breakpoint Usage

8.3.1 Checking for Presence of Complex Break Support

Software should verify that the complex breakpoint hardware is implemented prior to attempting to use it. The full sequence of steps is shown below for general use. Spots where the a M4K core has restricted behavior are noted.

1. Read the *Config1*$_{EP}$ bit to check for the presence of EJTAG logic. EJTAG logic is always present on a M4K core.

2. Read the *Debug*$_{NoDCR}$ bit to check for the presence of the Debug Control Register(DCR). The DCR will always be implemented on a M4K core.

3. Read the *DCR*$_{CBT}$ bit to check for the presence of any complex break and trigger features

4. Read the *CBTControl* register to check for the presence of each individual feature. If a M4K core implements any complex break and trigger features, it will implement all of them

5. If Pass Counters are implemented, they may not be implemented for all break channels and may have different counter sizes. To determine the size and presence of each pass counter, software can write -1 to each of the *IBPCn* and *DBPCn* registers and read it back. If a M4K core implements pass counters, it will implement an 8b counter for each instruction breakpoint and a 16b counter for each data breakpoint.

6. If tuples are implemented, they may only be supported on a subset of the data breakpoint channels. This can be checked by seeing if the *DBBCn*$_{TUP}$ bit can be set to 1. Additionally, some cores may support dynamically changing which instruction breakpoint is associated with a given data breakpoint. This can be checked by attempting to write the *DBCCn*$_{TIBrkNum}$ field. If a M4K core implements tuple support, it will support it for all data breakpoint channels and the instruction breakpoint association will be fixed.

7. If Priming Conditions are supported, a core may only support a subset of the possible priming condition values. This can be checked by 4’hf to the *xBCCn*$_{PrCnd}$ field. If only 1 or 2 bits can be written, the available priming conditions will be described in the *PrCndA* registers. If 3 bits are writeable, *PrCndA* and *PrCndB* will describe the conditions, and if all 4 bits are writeable, the *PrCndA*,*PrCndB*,*PrCndC*, and *PrCndD* registers will all exist. Some cores may also support changing the priming conditions and this can be checked by attempting to write to the *PrCnd* registers. If a M4K core supports priming conditions, it will support 4 statically mapped priming conditions per breakpoint which will be described in the *PrCndA* registers.

8. If support for qualified breakpoints is indicated, it may only be supported for some of the breakpoints. Additionally, the data breakpoint used for the qualification may be configurable. Software can check this by writing to the *$xBCCn_{DQ}$* and *$xBCCn_{DQBrkNum}$* fields. If a M4K core support qualified breakpoints, it will only support it on instruction breakpoints and the data break used for qualification will be fixed for each instruction breakpoint.

9. If the stopwatch timer is implemented, either one or two pairs of instruction breakpoints may be available for controlling it and it may be possible to dynamically select which instruction breakpoints are used. This can be tested by writing to the *STCtl* register.

8.3.2 General Complex Break Behavior

There is some general complex break behavior that is common to all of the features. This behavior is described below:

- Resets to a disabled state - when the core is reset, the complex break functionality will be disabled and debug software that is not aware of complex break should continue to function normally.
- Complex break state is not updated on exceptional instructions
- Complex breakpoints are evaluated at the end of the pipeline and complex breakpoint exceptions are taken imprecisely on the following instruction.
- There is no hazard between enabling and enabled events. When an instruction causes an enabling event, the following instruction sees the enabled state and reacts accordingly.

8.3.3 Usage of Pass Counters

Pass counters specify that the breakpoint conditions must match N times before the breakpoint action will be enabled.

- Controlled by writing to the per-breakpoint pass counter register
- Resets to 0
- Writing to a non-zero value enables the pass counter. When enabled, each time the breakpoint conditions match, the counter will be decremented by 1. After the counter value reaches 0, the breakpoint action (breakpoint exception, trigger, or complex break enable) will occur on any subsequent matches and the counter will not decrement further. The action does not occur on the match that causes the 1->0 counter decrement.
- If the breakpoint also has priming conditions and/or data qualified specified, the pass counter will only decrement when the priming and/or qualified conditions have been met
- If a data breakpoint is configured to be a tuple breakpoint, the data pass counter will only decrement on instructions where both the instruction and data break conditions match. The pass counter for the instruction break involved in a tuple should not be enabled if the tuple is enabled.
- Once a pass counter has been enabled, it will be treated as enabled until the pass counter is explicitly written to 0. Namely, breakpoint exceptions will continue to be taken imprecisely until the pass counter is disabled by writing to 0.
- The counter register will be updated as matches are detected. The current count value can be read from the register while operating in debug mode. Note that this behavior is architecturally recommended, but not required.

8.3.4 Usage of Tuple Breakpoints

A tuple breakpoint is the logical AND of a data breakpoint and an instruction breakpoint. Tuple breakpoints are specified as a condition on a data breakpoint. If the *$DBCCn_{TUP}$* bit is set, the data breakpoint will not match unless there the corresponding instruction breakpoint conditions are also met.

- Uses the data breakpoint resources to specify the break action, break status, pass counters, and priming conditions.
- The instruction breakpoint involved in the tuple should be configured as follows:
  - $IBCCn_{CBE}$ = 1
  - $IBCCn_{PrCnd}$ = $IBCCn_{DQ}$ = $IBCn_{TE}$ = $IBCn_{BE}$ = IBPCn = 0

8.3.5 Usage of Priming Conditions

Priming conditions provide a way to have one breakpoint enabled by another one. Prior to the priming condition being satisfied, any breakpoint matches are ignored.

- Priming condition resets to bypass which specifies that no priming is required
- 3 other priming conditions are available for each breakpoint. These condition vary from breakpoint to breakpoint (since it makes no sense for a breakpoint to prime itself). The conditions for each of the breakpoints are listed in Table 8.23.
- The priming breakpoint must have *$xBCn_{TE}$* or *$xBCCn_{CBE}$* set.
- Once the priming condition has been seen, the primed breakpoint will remain primed until its *xBCCn* register is written
- The primed state is stored with the breakpoint being primed and not with the breakpoint that is doing the priming.
- Each Prime condition is the comparator output after it has been qualified by its own Prime condition, data qualification, and pass counter. Using this, several stages of priming are possible (e.g. data cycle D followed by instruction A followed by instruction B N times followed by instruction C).

8.3.6 Usage of Data Qualified Breakpoints

Each of the instruction breakpoints can be set to be data qualified. In qualified mode, a breakpoint will recognize its conditions only after the specified data breakpoint matches both address and data. If the data breakpoint matches address, but has a mismatch on the data value, the instruction breakpoint will be unqualified and will not match until a subsequent qualifying match.

This feature can be used similarly to the ASID qualification that is available on cores with TLBs. If an RTOS loads a process ID for the current process, that load can be used as the qualifying breakpoint. When a matching process ID is loaded (entering the desired RTOS process), qualified instruction breakpoints will be enabled. When a different process ID is loaded (leaving the desired RTOS process), the qualified instruction breakpoints are disabled. Alternatively, with the InvertValueMatch feature of the data breakpoint, the instruction breakpoints could be enabled on any process ID other than the specified one.

- The qualifying data break must have $DBCn_{TE}$ or $DBCCn_{CBE}$ set.
- The qualifying data break should have data comparison enabled (via settings of $DBCn_{BLM}$ and $DBCn_{BAI}$)
- The qualifying data break should not have pass counters, priming conditions, or tuples enabled.
- The qualifying data access can be either a load or store, depending on the settings of $DBCn_{NoSB}$ and $DBCn_{NoLB}$
- The Qualified/Unqualified state is stored with the instruction breakpoint that is being qualified. Writing it's *IBCCn* register will unqualify that breakpoint.
- Qualified instruction breakpoint can also have priming conditions and/or pass counters enabled. The pass counter will only decrement when the priming and qualifying conditions have been met. The instruction breakpoint action (break, trigger, or complex enable) will only occur when all priming, qualifying, and pass counter conditions have been met.
- Qualified instruction breakpoint can be used to prime another breakpoint

8.3.7 Usage of Stopwatch Timers

The stopwatch timer is a drseg memory mapped count register. It can be configured to be free running or controlled by instruction breakpoints. This could be used to measure the amount of time that is spent in a particular function by starting the counter upon function entry and stopping it upon exit.

- Count value is reset to 0
- Reset state has counter stopped and under breakpoint control so that the counter is not running when the core is not being debugged.
- Bit in CBTControl register controls whether the counter is free-running or breakpoint controlled.
- Counter does not count in debug mode
- When breakpoint controlled, the involved instruction breakpoints must have $IBCn_{TE}$ or $IBCCn_{CBE}$ set in order to start or stop the timer.

8.4 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.

8.4.1 EJTAG Internal and External Interfaces

The external interface of the EJTAG module consists of the 5 signals defined by the IEEE standard.

Table 8.26 EJTAG Interface Pins

Pin	Type	Description
TCK	I	Test Clock Input Input clock used to shift data into or out of the Instruction or data registers. The *TCK* clock is independent of the processor clock, so the EJTAG probe can drive *TCK* independently of the processor clock frequency. The core signal for this is called *EJ_TCK*
TMS	I	Test Mode Select Input The *TMS* input signal is decoded by the TAP controller to control test operation. *TMS* is sampled on the rising edge of *TCK*. The core signal for this is called *EJ_TMS*
TDI	I	Test Data Input Serial input data (*TDI*) is shifted into the Instruction register or data registers on the rising edge of the *TCK* clock, depending on the TAP controller state. The core signal for this is called *EJ_TDI*
TDO	O	Test Data Output Serial output data is shifted from the Instruction or data register to the *TDO* pin on the falling edge of the *TCK* clock. When no data is shifted out, the *TDO* is 3-stated. The core signal for this is called *EJ_TDO* with output enable controlled by *EJ_TDOzstate*.
TRST_N	I	Test Reset Input (Optional pin) The *TRST_N* 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 *TRST_N*. The core signal for this is called *EJ_TRST_N* 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.

8.4.2 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 the 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 8.1. 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* by 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.

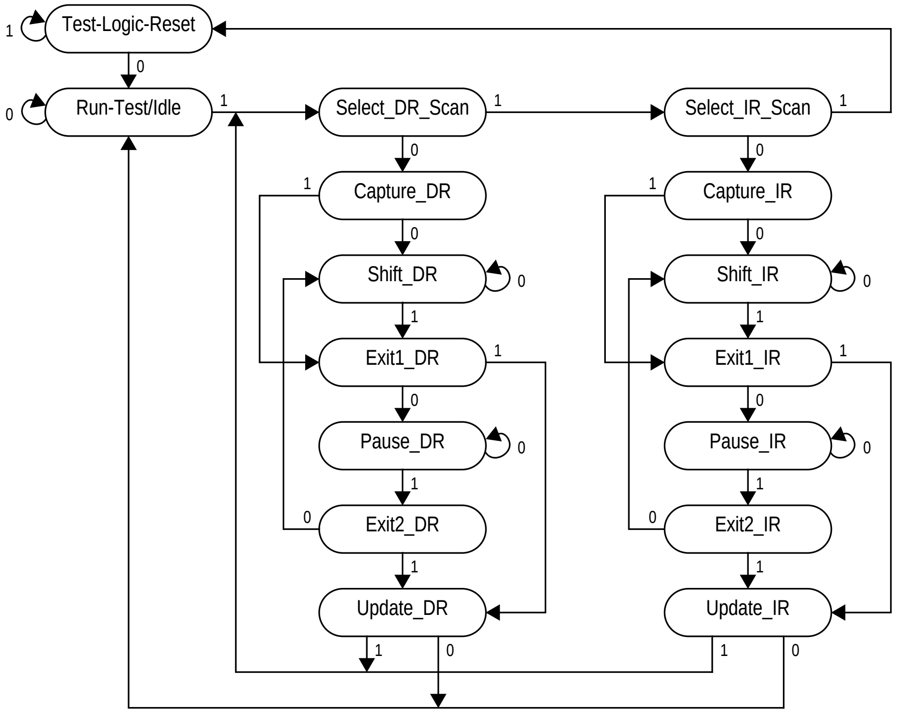
When test access is required, a protocol is applied via the *TMS* and *TCK* inputs, causing the TAP to exit the *Test-Logic-Reset* state and move through the appropriate states. From the *Run-Test/Idle* state, an Instruction register scan or a data register scan can be issued to transition the TAP through the appropriate states shown in Figure 8.1.

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 8.1 TAP Controller State Diagram



8.4.2.1 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.

8.4.2.2 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.

8.4.2.3 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.

8.4.2.4 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.

8.4.2.5 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.

8.4.2.6 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.

8.4.2.7 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.

8.4.2.8 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.

8.4.2.9 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.

8.4.2.10 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.

8.4.2.11 Capture_IR State

In this state the shift register contained in the Instruction register loads a fixed pattern (00001_2) 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.

8.4.2.12 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.

8.4.2.13 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.

8.4.2.14 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.

8.4.2.15 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.

8.4.2.16 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.

8.4.3 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 8.27 Implemented EJTAG Instructions

Value	Instruction	Function
0x01	IDCODE	Select Chip Identification data register
0x03	IMPCODE	Select Implementation register
0x08	ADDRESS	Select Address register
0x09	DATA	Select Data register
0x0A	CONTROL	Select EJTAG Control register
0x0B	ALL	Select the Address, Data and EJTAG Control registers
0x0C	EJTAGBOOT	Set EjtagBrk, ProbEn and ProbTrap to 1 as reset value
0x0D	NORMALBOOT	Set EjtagBrk, ProbEn and ProbTrap to 0 as reset value
0x0E	FASTDATA	Selects the Data and Fastdata registers
0x10	TCBCONTROLA	Selects the *TCBTCONTROLA* register in the Trace Control Block
0x11	TCBCONTROLB	Selects the *TCBTCONTROLB* register in the Trace Control Block
0x12	TCBDATA	Selects the *TCBDATA* register in the Trace Control Block
0x1F	BYPASS	Bypass mode

8.4.3.1 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.

8.4.3.2 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.

8.4.3.3 IMPCODE Instruction

This instruction selects the Implementation register for output, which is always 32 bits.

8.4.3.4 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.

8.4.3.5 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.

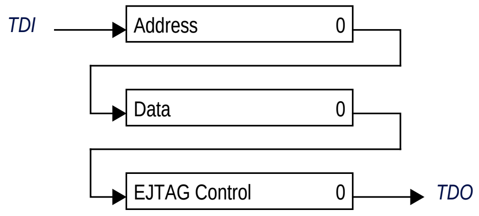
8.4.3.6 CONTROL Instruction

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*.

8.4.3.7 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 8.2 Concatenation of the EJTAG Address, Data and Control Registers



8.4.3.8 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 hard or soft 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 hard or soft reset, without fetching or executing any instructions from the normal memory area. This can be used for download of code to a system which have no code in ROM.

The Bypass register is selected when the EJTAGBOOT instruction is given.

8.4.3.9 NORMALBOOT Instruction

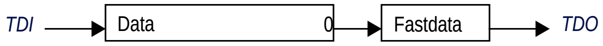
When the NORMALBOOT instruction is given and the Update-IR state is left, then the reset value of the ProbTrap, ProbEn and EjtagBrk bits in the EJTAG Control register are set to 0 after hard or soft reset.

The Bypass register is selected when the NORMALBOOT instruction is given.

8.4.3.10 FASTDATA Instruction

This selects the Data and the Fastdata registers at once, as shown in Figure 8.3.

Figure 8.3 TDI to TDO Path When in Shift-DR State and FASTDATA Instruction is Selected



8.4.3.11 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 Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

8.4.3.12 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 Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

8.4.3.13 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 Control Block is present. If no TCB 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.

8.5 EJTAG TAP Registers

The EJTAG TAP Module has one Instruction register and a number of data registers, all accessible through the TAP:

8.5.1 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 00001_2, 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 8.27.

8.5.2 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.

This description applies in general to the following data registers:

- Bypass Register
- Device Identification Register
- Implementation Register
- EJTAG Control Register (ECR)
- Processor Access Address Register
- Processor Access Data Register
- FastData Register

8.5.2.1 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.

8.5.2.2 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 8.28 shows the bit assignments defined for the read-only Device Identification Register, and 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.

***Device Identification* Register Format**

31 28	27 12	11 1	0
Version	PartNumber	ManufID	R

Table 8.28 Device Identification Register

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
Version	31:28	**Version** (4 bits) This field identifies the version number of the processor derivative.	R	*EJ_Version[3:0]*
PartNumber	27:12	**Part Number** (16 bits) This field identifies the part number of the processor derivative.	R	*EJ_PartNumber[15:0]*
ManufID	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	*EJ_ManufID[10:0]*

Table 8.28 Device Identification Register

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
R	0	reserved	R	1

8.5.2.3 *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.

Implementation **Register Format**

31 29	28 25	24	23 21	20 17	16	15	14	13 0
EJTAGver	reserved	DINT-sup	ASIDsize	reserved	MIPS16	0	NoDMA	reserved

Table 8.29 *Implementation* Register Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
EJTAGver	31:29	EJTAG Version. 2: Version 2.6	R	2
reserved	28:25	reserved	R	0
DINTsup	24	DINT Signal Supported from Probe This bit indicates if the DINT signal from the probe is supported: 0: DINT signal from the probe is not supported 1: Probe can use DINT signal to make debug interrupt.	R	*EJ_DINTsup*
ASIDsize	23:21	Size of ASID field in implementation: 0: No ASID in implementation 1: 6-bit ASID 2: 8-bit ASID 3: Reserved	R	0
reserved	20:17	reserved	R	0
MIPS16	16	Indicates whether MIPS16 is implemented 0: No MIPS16 support 1: MIPS16 implemented	R	
reserved	15	reserved	R	0
NoDMA	14	No EJTAG DMA Support	R	1
reserved	13:0	reserved	R	0

8.5.2.4 EJTAG Control Register

This 32-bit register controls the various operations of the TAP modules. This register is selected by shifting in 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 31, is either 0 or written to 0. This is in order to ensure prober handling of processor accesses.

The value used for reset indicated in the table below takes effect on both hard and soft CPU resets, but not on TAP controller resets by e.g. *TRST_N*. *TCK* clock is not required when the hard or soft CPU reset occurs, but the bits are still updated to the reset value when the *TCK* applies. The first 5 *TCK* clocks after hard or soft CPU resets may result in reset of the bits, due to synchronization between clock domains.

EJTAG Control **Register Format**

31	30 29	28 23	22	21	20	19	18	17	16	15	14	13	12	11 4	3	2 0
Rocc	Psz	Res	Doze	Halt	PerRst	PRnW	PrAcc	Res	PrRst	ProbEn	ProbTrap	Res	EjtagBrk	Res	DM	Res

Table 8.30 *EJTAG Control* Register Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
Rocc	31	Reset Occurred The bit indicates if a hard or soft reset has occurred: 0: No reset occurred since bit last cleared. 1: Reset occurred since bit last cleared. The Rocc bit will keep the 1 value as long as a hard or soft reset is applied. This bit must be cleared by the probe, to acknowledge that the incident was detected. The EJTAG Control register is not updated in the *Update-DR* state unless Rocc is 0, or written to 0. This is in order to ensure proper handling of processor access.	R/W	1

Table 8.30 *EJTAG Control* Register Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>Psz[1:0]</td><td>30:29</td><td>Processor Access Transfer Size
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.
<table>
<tr><th>PAA[1:0]</th><th>Psz[1:0]</th><th>Transfer Size</th></tr>
<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>
</table>
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 endianess.</td><td>R</td><td>Undefined</td></tr>
<tr><td>Res</td><td>28:23</td><td>reserved</td><td>R</td><td>0</td></tr>
<tr><td>Doze</td><td>22</td><td>Doze state
The Doze bit indicates any kind of low power mode. The value is sampled in the Capture-DR state of the TAP controller:
0: CPU not in low power mode.
1: CPU is in low power mode
Doze includes the Reduced Power (RP) and WAIT power-reduction modes.</td><td>R</td><td>0</td></tr>
<tr><td>Halt</td><td>21</td><td>Halt state
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:
0: Internal system clock is running
1: Internal system clock is stopped</td><td>R</td><td>0</td></tr>
</table>

Table 8.30 *EJTAG Control* Register Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
PerRst	20	Peripheral Reset 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 *TCK* clock domain gets effect in the CPU clock domain, and in peripherals. 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. This bit controls the *EJ_PerRst* signal on the core.	R/W	0
PRnW	19	Processor Access Read and Write 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: 0: Read transaction 1: Write transaction	R	Undefined
PrAcc	18	Processor Access (PA) Read value of this bit indicates if a Processor Access (PA) to the EJTAG memory is pending: 0: No pending processor access 1: Pending processor access The probe's software must clear this bit to 0 to indicate the end of the PA. Write of 1 is ignored. A pending Processor Access is cleared when Rocc is set, but another PA may occur just after the reset if a debug exception occurs. Finishing a Processor Access is not accepted while the Rocc 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. The FASTDATA access can clear this bit.	R/W0	0
Res	17	reserved	R	0
PrRst	16	Processor Reset (Implementation dependent behavior) 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 *TCK* clock domain gets effect in the CPU clock domain, and in peripherals. 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. This bit controls the *EJ_PrRst* 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 EJTAG Control register is reset by hard or soft reset.	R/W	0

Table 8.30 *EJTAG Control* Register Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
ProbEn	15	Probe Enable This bit indicates to the CPU if the EJTAG memory is handled by the probe so processor accesses are answered: 0: The probe does not handle EJTAG memory transactions 1: The probe does handle EJTAG memory transactions It is an error by the software controlling the probe if it sets the ProbTrap bit to 1, but resets the ProbEn to 0. The operation of the processor is UNDEFINED in this case. The ProbEn bit is reflected as a read-only bit in the ProbEn bit, bit 0, in the Debug Control Register (DCR). The read value indicates the effective value in the DCR, due to synchronization issues between *TCK* and CPU clock domains; however, it is ensured that change of the ProbEn prior to setting the EjtagBrk bit will have effect for the debug handler executed due to the debug exception. The reset value of the bit depends on whether the EJTAGBOOT indication is given or not: No EJTAGBOOT indication given: 0 EJTAGBOOT indication given: 1	R/W	0 or 1 from EJTAGBOOT
ProbTrap	14	Probe Trap This bit controls the location of the debug exception vector: 0: In normal memory 0xBFC0.0480 1: In EJTAG memory at 0xFF20.0200 in dmseg Valid setting of the ProbTrap bit depends on the setting of the ProbEn bit, see comment under ProbEn bit. The ProbTrap should not be set to 1, for debug exception vector in EJTAG memory, unless the ProbEn 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 *TCK* and CPU clock domains; however, it is ensured that change of the ProbTrap bit prior to setting the EjtagBrk bit will have effect for the EjtagBrk. The reset value of the bit depends on whether the EJTAGBOOT indication is given or not: No EJTAGBOOT indication given: 0 EJTAGBOOT indication given: 1	R/W	0 or 1 from EJTAGBOOT
Res	13	reserved	R	0

Table 8.30 *EJTAG Control* Register Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
EjtagBrk	12	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: No EJTAGBOOT indication given: 0 EJTAGBOOT indication given: 1	R/W1	0 or 1 from EJTAGBOOT
Res	11:4	reserved	R	0
DM	3	Debug Mode This bit indicates the debug or non-debug mode: 0: Processor is in non-debug mode 1: Processor is in debug mode The bit is sampled in the *Capture-DR* state of the TAP controller.	R	0
Res	2:0	reserved	R	0

8.5.3 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.

8.5.3.1 Processor Access Data Register

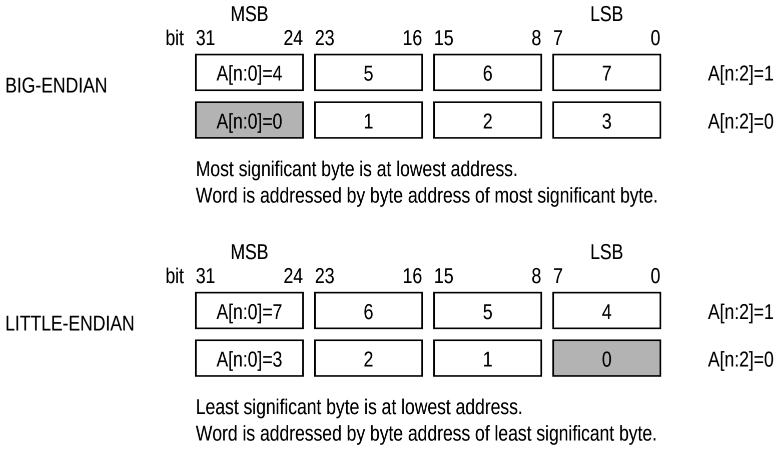
The Processor Access Data (*PAD*) register is used to provide data value to and from a processor access. The length of the Data register is 32 bits, and this register is selected by shifting in the DATA instruction.

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 fora 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 endianess of the core, as shown in Figure 8.4. The endian mode for debug/kernel mode is determined by the state of the *SI_Endian* input at power-up.

Figure 8.4 Endian Formats for the *PAD* Register


MSB
LSB
bit 31 24 23 16 15 8 7 0
BIG-ENDIAN
A[n:0]=4
5
6
7
A[n:2]=1
A[n:0]=0
1
2
3
A[n:2]=0
Most significant byte is at lowest address.
Word is addressed by byte address of most significant byte.
MSB
LSB
bit 31 24 23 16 15 8 7 0
LITTLE-ENDIAN
A[n:0]=7
6
5
4
A[n:2]=1
A[n:0]=3
2
1
0
A[n:2]=0
Least significant byte is at lowest address.
Word is addressed by byte address of least significant byte.

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*.

8.5.4 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).

***Fastdata* Register Format**

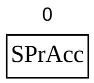


Table 8.31 Fastdata Register Field Description

Fields		Description	Read/ Write	Power-up State
Name	Bits			
SPrAcc	0	Shifting in a zero value requests completion of the Fastdata access. The PrAcc bit in the EJTAG Control register is overwritten with zero when the access succeeds. (The access succeeds if PrAcc is one and the operation address is in the legal dmseg Fastdata area.) When successful, a one is shifted out. Shifting out a zero indicates a Fastdata access failure. Shifting in a one does not complete the Fastdata access and the PrAcc 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 + 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. These are:

- 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 8.32 shows the values of the PrAcc and SPrAcc bits and the results of a Fastdata access. .

Table 8.32 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 FASTDATA	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 is no restriction 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 Control register is not used for the FASTDATA operation.

8.6 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 soft or hard reset.

8.6.1 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 found 1, it means that 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.

Note: probe accesses and external bus accesses are serialized by the core. A probe access will not begin until all external bus requests have completed. Similarly, a new probe or external bus access will not begin until a pending probe access has completed.

8.7 Trace Mechanisms

There are two optional trace mechanisms that are available to extract additional information about program execution. EJTAG Trace is a powerful mechanism that allows for the tracing of the program flow as well as load and store addresses and data values. EJTAG Trace can be configured to only trace in specific modes and can produce cycle accurate trace information. Tracing can be controlled by either a hardware (probe) or software interface. In contrast, the iFlowtrace™ mechanism is much lighter weight. It only can only be controlled by debug software executing on the core and it only provides the ability to trace the program flow. The reduced capabilities also reduce the silicon area required to implement it and reduces the costs associated with tracing, while still providing valuable information for software debugging.

These two trace mechanisms are described in further detail in the rest of the chapter.

8.8 iFlowtrace™ Mechanism

The iFlowtrace mechanism provides a means to reconstruct a simple instruction trace from an execution stream. This light-weight instruction-only tracing scheme is sufficient to reconstruct the execution flow in an M4K core under conditions that are classified as appropriate.

The presence of the iFlowtrace mechanism is indicated by the *CP0 Config3*$_{ITL}$ register bit.

8.8.1 A Simple Instruction-Only Tracing Scheme

A trace methodology can often be mostly defined by its inputs and outputs. Hence this basic scheme is described by the inputs to the core tracing logic and by the trace output format from the core. We assume here that the execution flow of the program is traced at the end of the execution path in the core similar to PDtrace.

8.8.1.1 Trace Inputs

1. In_TraceOn: when on, legal trace words are coming from the core and at the point when it is turned on, that is for the first traced instruction, a full PC value is output. When off, it cannot be assumed that legal trace words are available at the core interface.

2. In_Stall: This says, stall the processor to avoid buffer overflow that can lose trace information. When off, a buffer overflow will simply throw away trace data and start over again. When on, the processor is signalled from the tracing logic to stall until the buffer is sufficiently drained and then the pipeline is restarted.

8.8.1.2 Trace Outputs

1. Stall cycles in the pipe are ignored by the tracing logic and are not traced. This is indicated by a valid signal Out_Valid that is turned off when no valid instruction is being traced. When the valid signal is on, instructions are traced out as described in the rest of this section. The traced instruction PC is a virtual address.

2. In the output format, every sequentially executed instruction is traced as bit 0.

3. Every instruction that is not sequential to the previous one is traced as either a 10 or an 11. This implies that the target instruction of a branch or jump is traced this way, not the actual branch or jump instruction (this is similar to PDtrace):

4. A 10 instruction implies a taken branch for a conditional branch instruction whose condition is unpredictable statically, but whose branch target can be computed statically and hence the new PC does not need to be traced out. Note that if this branch was not taken, it would have been indicated by a 0 bit, that is sequential flow.

5. A 11 instruction implies a taken branch for an indirect jump-like instruction whose branch target could not be computed statically and hence the taken branch address is now given in the trace. This includes, for example, instructions like jr, jalr, and interrupts:

   - 11 00 - followed by 8 bits of 1-bit shifted offset from the last PC. The bit assignments of this format on the bus between the core tracing logic and the ITCB is:
     [3:0] = 4’b0011
     [11:4] = PCdelta[8:1]
     [35:12] = 24’b0

   - 11 01 - followed by 16 bits of 1-bit shifted offset from the last PC. The bit assignments of this format on the bus between the core tracing logic and the ITCB is:
     [3:0] = 4’b1011
     [19:4] = PCdelta[16:1]
     [35:20] - 16’b0

   - 11 10 - followed by 31 of the most significant bits of the PC value, followed by a bit (NCC) that indicates no code compression. Note that for a MIPS32 or MIPS64 instruction, NCC=1, and for MIPS16e instruction NCC=0. This trace record will appear at all transition points between MIPS32/MIPS64 and MIPS16e instruction execution.
     This form is also a special case of the 11 format and it is used when the instruction is not a branch or jump,

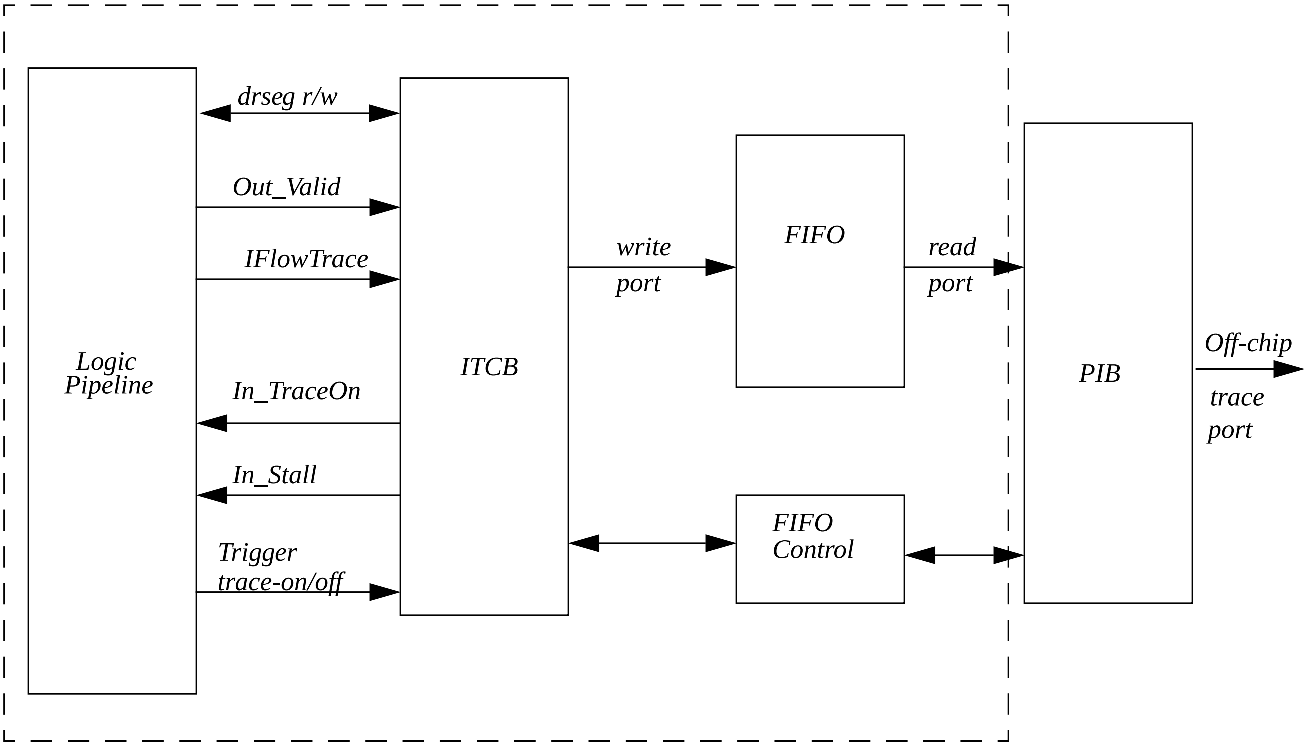
but nevertheless the full PC value needs to be reconstructed. This is used for synchronization purposes, similar to the Sync in PDtrace. A preset sync period of 256 instructions is counted down and when an internal counter runs through all the values, this format is used. The bit assignments of this format on the bus between the core tracing logic and the ITCB is:
[3:0] = 4’b0111
[34:4] = PC[31:1]
[35] = NCC

- 11 11 - Used to indicate trace resumption after a discontinuity occurred. The next format is a 1110 that sends a full PC value. A discontinuity might happen due to various reasons, for example, an internal buffer overflow, and at trace-on/trace-off trigger action.

8.8.2 ITCB Overview

The IFlowTrace Control Block (ITCB) is responsible for accepting trace signals from the CPU core, formatting them, and storing them into an on-chip FIFO. The figure also shows the Probe Interface Block (PIB) which reads the FIFO and outputs the memory contents through a narrow off-chip trace port.

Figure 8.5 Trace Logic Overview



8.8.3 ITCB IFlowTrace Interface

The IFlowTrace interface consists of 36 data signals plus a valid signal. The 36 data signals encode information about what the CPU is doing in each clock cycle. Valid indicates that the CPU is executing an instruction in this cycle and therefore the 36 data signals carry valid execution information. The IFlowTrace data bus is encoded as shown in Table 8.33. Note that all the non-defined upper bits of the bus are zeroes.

Table 8.33 Data Bus Encoding

Valid	Data (LSBs)	Description
0	X	No instructions executed in this cycle
1	0	Sequential instruction executed
1	01	Branch executed, destination predictable from code
1	<8>0011	Discontinuous instruction executed, PC offset is 8 bit signed offset
1	<16>1011	Discontinuous instruction executed, PC offset is 16 bit signed offset
1	<NCC><31>0111	Discontinuous instruction or synchronization record, No Code Compression (NCC) bit included as well as 31 MSBs of the PC value
1	1111	Internal overflow

The ITCB controls trace using the In_TraceOn signal. When 0, all data appearing on the IFlowTrace outputs is considered invalid. To turn on trace, the ITCB switches In_TraceOn from 0 to 1. A 1011 record represents the first instruction executed thereafter with a full PC indicating the current execution point.

8.8.4 ITCB IFlowTrace Storage Representation

Records from IFlowTrace are inserted into a memory stream exactly as they appear on the IFlowTrace data output. Records are concatenated into a continuous stream starting at the LSB. When a trace word is filled, it is written to memory along with some tag bits. Each record consists of a 64-bit word, which comprises 58 message bits and 6 tag bits or header bits that clarify information about the message in that word.

The ITCB includes a 58-bit shift register to accumulate trace messages. Once 58 or more bits are accumulated, the 58 bits and 6 tag bits are sent to the memory write interface. Messages may span a trace word boundary. In this case, the 6 tag bits indicate the bit number of the first full trace message in the 58-bit data field.

The tag bits are not strictly binary because they serve a secondary purpose of indicating to off-chip trace hardware when a valid trace word transmission begins. At least one of the 4 LSB's of the tag is always a 1. The longest trace message is 36 bits, so the starting position indicated by the tag bits is always between 0 and 35.

When trace stops (ON set to zero), any partially filled trace words are written to memory. Any unused space above the final message is filled with 1's. The decoder distinguishes 1111 patterns used for fill in this position from an 1111 overflow message by recognizing that it is the last trace word.

These trace formats are written to a trace memory that is off-chip. No particular size of SRAM is specified; the size is user selectable based on the application needs and area trade-offs. Each trace word can typically store about 20 to 30 instructions, so a 1 KWord trace memory could store the history of 20K to 30K executed instructions.

8.8.5 ITCB IFlowTrace Interface

The ITCB includes a drseg memory interface to allow the MIPS CPU to set up tracing and read current status. There are two drseg register locations in the ITCB as shown in Table 8.34.

Table 8.34 Registers in the ITCB

drseg Location Offset	Register	Defined Bits	Code	Description
0x3FC0	Control/Status	0	ON	Software control of trace collection. 0 disables all collection and flushes out any partially filled trace words.
		1	EN	Trace enable. This bit may be set by software or by Trace-on/Trace-off action bits caused by EJTAG hardware breaks. Software writes EN with the desired initial state of tracing when the ITCB is first turned on and EN is controlled by hardware thereafter. EN turning on and off does not flush partly filled trace words.
		2	IO	Inhibit overflow. If set, the CPU is stalled whenever the trace memory is full. Ignored unless OfC is also set.
		3	OfC	Offchip. 1 enables the PIB (if present) to unload the trace memory. 0 disables the PIB and would be used when on-chip storage is desired or if a PIB is not present. The M4K core only supports off-chip storage so this bit will be a read-only 1.
		4	OfClk	Controls the Off-chip clock ratio. When the bit is set, this implies 1:2, that is the trace clock is running at 1/2 the core clock, and when the bit is clear, implies 1:4 ratio, that is the trace clock is at 1/4 the core clock
0x3FC8	Trace write address pointer	N:0	WAddr	This register is used only if the SRAM is supported in on-chip mode. The current write pointer for trace memory. Each completed trace word is written to memory, then WAddr increments. When trace concludes, WAddr contains the first address in trace memory not yet written.
		31	Wrap	Trace wrapped. This bit indicates that the entire trace depth has been written at least once. After trace concludes, this bit along with WAddr is used by software to determine the oldest and youngest words in the buffer.

8.8.6 ITCB IFlowTrace Off-Chip Interface

The off-chip interface consists of a 4-bit data port (TR_DATA) and a trace clock (TR_CLK). TR_CLK can be a DDR clock, that is, both edges are significant. TR_DATA and TR_CLK follow the same timing and have the same output structure as the PDtrace TCB described in MIPS specifications. The trace clock is the same as the system clock or related to the system clock as either divided or multiplied. The OfClk bit in the Control/Status register is of the form X:Y, where X is the trace clock and Y is the core clock. The Trace clock is always 1/2 of the trace port data rate, hence the “full speed” ITCB outputs data at the CPU core clock rate but the trace clock is half that, hence the 1:2 OfClk value is the full speed, and the 1:4 OfClk ratio is half-speed.

When a 64-bit trace word is ready to transmit, the PIB reads it from the FIFO and begins sending it out on TR_DATA. It is sent in 4-bit increments starting at the LSB’s. In a valid trace word, the 4 LSB’s are never all zero, so a probe listening on the TR_DATA port can easily determine when the transmission begins and then count 15 additional cycles to collect the whole 64-bit word. Between valid transmissions, TR_DATA Is held at zero and TR_CLK continues to run.

TR_CLK runs continuously whenever a probe is connected. An optional signal TR_PROBE_N may be pulled high when a probe is not connected and could be used to disable the off-chip trace port. If not present, this signal must be tied low at the PIB input.

The following encoding is used for the 6 tag bits to tell the PIB receiver that a valid transmission is starting:

```
//   if (srcount == 0), EncodedSrCount = 111000 = 56
//   else if (srcount == 16) EncodedSrCount = 111001 = 57
//   else if (srcount == 32) EncodedSrCount = 111010 = 58
//   else EncodedSrCount = srcount
```

8.8.7 Breakpoint-Based Enabling of Tracing

Each hardware breakpoint in the EJTAG block has a control bit associated with it that enables a trigger signal to be generated on a break match condition. This trigger signal can be used to turn trace on or off, thus allowing a user to control the trace on/off functionality using breakpoints. For the simple hardware breakpoints, there are already defined registers TraceIBPC, TraceDBPC, etc in PDtrace that are used to control tracing functionality. Similar registers need to be defined to control the start and stop of IFlowTrace. And in addition, the new complex Tuple breakpoints need to be added to the list of breakpoints that can trigger trace. The details on the actual register names and drseg addresses are shown in Table 8.35.

Table 8.35 Registers that Enable/Disable Trace from Complex Triggers and their drseg Addresses

Register Name	drseg Address	Reset value	Description
ITrigiFlowTrcEn	0x3FD0	0	Instruction break Trigger iFlowtrace Enable register
DTrigiFlowTrcEn	0x3FD8	0	Data break Trigger iFlowtrace Enable register

The bits in each register are defined as follows:

- Bit 28 (IE/DE) : Used to specify whether the trigger signal from EJTAG simple or complex instruction or data break should trigger iFlowtrace tracing functions or not. Value of 0 disables trigger signals from EJTAG instruction breaks, and 1 enables triggers for the same.

- Bits 14..0 (IBrk/DBrk): Used to explicitly specify which instruction or data breaks enable or disable iFlowtrace. A value of 0 implies that trace is turned off (unconditional trace stop) and a value of 1 specifies that the trigger enables trace (unconditional trace start). If both trace on and trace off events happen on the same instruction, tracing will be enabled.

8.9 EJTAG Trace

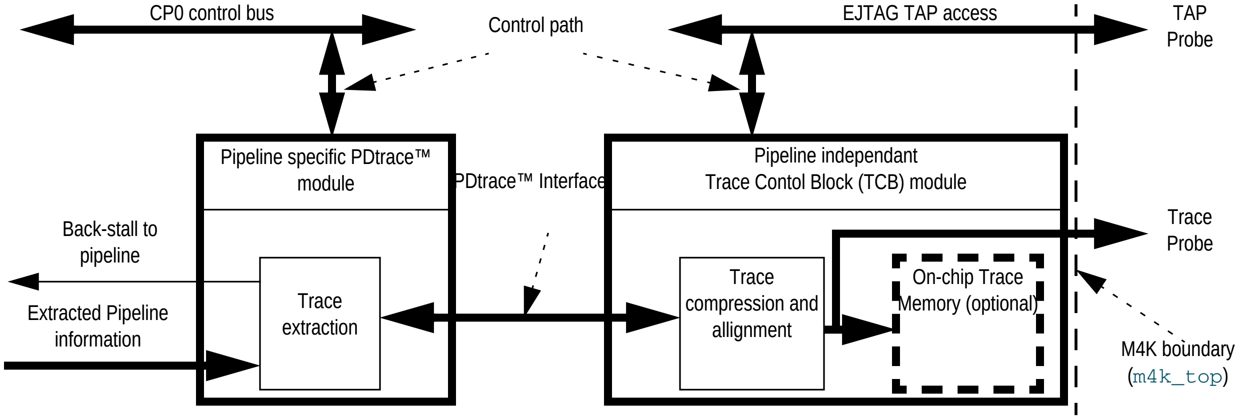
EJTAG Trace enables the ability to trace program flow, load/store addresses and load/store data. Several run-time options exist for the level of information which is traced, including tracing only when in specific processor modes (i.e. UserMode or KernelMode). EJTAG Trace is an optional block in the M4K core. If EJTAG Trace is not implemented, the rest of this chapter is irrelevant. If EJTAG Trace is implemented, the *CP0 Config3*$_{TL}$ bit is set.

The pipeline specific part of EJTAG Trace is architecturally specified in the *PDtrace™ Interface Specification*. The PDtrace module extracts the trace information from the processor pipeline, and presents it to a pipeline-independent module called the Trace Control Block (TCB). The TCB is specified in the *EJTAG Trace Control Block Specification*. The collective implementation of the two is called *EJTAG Trace*.

When EJTAG Trace is implemented, the M4K core includes both the PDtrace and the Trace Control Block (TCB) modules. The two modules “talk” to each other on the generic pin-interface called the PDtrace™ Interface. This interface is embedded inside the M4K core, and will not be discussed in detail here (read the *PDtrace™ Interface Specifi-*

cation for a detailed description). While working closely together, the two parts of EJTAG Trace are controlled separately by software. Figure 8.6 shows an overview of the EJTAG Trace modules within the core.

Figure 8.6 EJTAG Trace Modules in the M4K™ Core


CP0 control bus
Control path
EJTAG TAP access
TAP Probe
Pipeline specific PDtrace™ module
PDtrace™ Interface
Pipeline independant Trace Contol Block (TCB) module
Back-stall to pipeline
Trace Probe
Trace extraction
Trace compression and allignment
On-chip Trace Memory (optional)
Extracted Pipeline information
M4K boundary (m4k_top)

To some extent, the two modules both provide similar trace control features, but the access to these features is quite different. The PDtrace controls can only be reached through access to CP0 registers. The TCB controls can only be reached through EJTAG TAP access. The TCB can then control what is traced through the PDtrace™ Interface.

Before describing the EJTAG Trace implemented in the M4K core, some common terminology and basic features are explained. The remaining sections of this chapter will then provide a more thorough explanation.

8.9.1 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.

$\texttt{DebugMode} \leftarrow (\texttt{Debug}_{DM} = 1)$
$\texttt{ExceptionMode} \leftarrow (\texttt{not DebugMode) and ((Status}_{EXL} = 1) \texttt{ or (Status}_{ERL} = 1))$
$\texttt{KernelMode} \leftarrow (\texttt{not (DebugMode or ExceptionMode)) and (Status}_{UM} = 0)$
$\texttt{UserMode} \leftarrow (\texttt{not (DebugMode or ExceptionMode)) and (Status}_{UM} = 1)$

8.9.2 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 for how trace is controlled. 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 TCB, via the PDtrace interface, is used to select the trace modes. The *TraceControl.TS* bit determines whether software or hardware control is active.

8.9.3 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 EJTAG Trace this is done by providing the minimal cycle-by-cycle information necessary on the PDtrace™ interface 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-flag[1].
- 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 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.

All the instruction flags above are combined into one 3-bit value, to minimize the bit information to trace. The possible processor modes are explained in 8.9.1 "Processor Modes" on page 175.

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 is required to 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.

8.9.4 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.
- 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.

[1] A SC (Store Conditional) instruction is not flagged as a store instruction if the load-locked bit prevented the actual store.

- 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.

8.9.5 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 8.9.1 "Processor Modes" on page 175. .

Additionally, an EJTAG Simple Break trigger point can override the processor mode and turn them all on. Another trigger point can disable this override again.

8.9.6 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 load address.

- PC and store address.

- PC and load and store address.

- PC and load address and load data.

- PC and store address and store data.

- PC and load and store address and load and store data.

- PC and load data only.

The last option is helpful when used together with instruction accurate simulators. 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.

8.9.6.1 User Data Trace

In addition to the above, a special CP0 register, *UserTraceData*, can generate a data trace. When this register is written, and the global Trace On is set, then the 32-bit data written is put in the trace as special User Data information.

Remark: The User Data is sent even if the processor is operating in an un-traced processor mode.

8.9.7 Enable Trace to Probe/On-Chip Memory

When trace is On, based on the options listed in 8.9.5 "Programmable Processor Trace Mode Options", the trace information is continuously sent on the PDtrace™ interface to the TCB. The TCB must, however, be enabled to transmit the trace information to the Trace probe or to on-chip trace memory, by having the *TCBCONTROLB*$_{EN}$ bit set. It is possible to enable and disable the TCB in two ways:

- Set/clear the *TCBCONTROLB*$_{EN}$ bit via an EJTAG TAP operation.
- Initialize a TCB trigger to set/clear the *TCBCONTROLB*$_{EN}$ bit.

8.9.8 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.

Trace triggers may prove useful for various types of system debug. If the system has a reasonable capability to program the external triggers, a wide variety of system information can be included in the trace:

- Insert system events into a trace.
  - Using a timer event as a trigger that inserted a trace record would allow for performance analysis (at a coarser granularity than cycle accurate mode, but with better compression)
  - The trace could be annotated with interesting system events like each time a packet is received or transmitted
- Trigger traces
  - Stop tracing when a bus error is detected so that the trace buffer contains the code sequence leading up to the error

Note that trace triggers are independent from EJTAG triggerpoints and the presence or absence of trace triggers does not impact the ability to start or stop trace with triggerpoints.

8.9.9 Cycle by Cycle Information

All of the trace information listed in 8.9.3 "Trace Information" and 8.9.4 "Load/Store Address and Data Trace Information", will be collected from the PDtrace™ interface by the TCB. The trace will then be 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.

8.9.10 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). One important feature is that all Trace Formats have at least one non-zero bit.

8.9.11 Trace Word Format

After the PDtrace™ data has been turned into 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. This complicates how to store this information into an on-chip memory of fixed width without too much wasted space. It also complicates how to transmit data 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 is built of all zeros. A TW which contains one or more valid TF's is guaranteed to have a non-zero value on one of the four least significant bits [3:0]. During operation of the TCB, each TW is built from the TF's 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.

8.10 PDtrace™ Registers (Software Control)

The CP0 registers associated with PDtrace are listed in Table 8.36 and described in Chapter 5, "CP0 Registers of the M4K™ Core" on page 85

Table 8.36 A List of Coprocessor 0 Trace Registers

Register Number	Se l	Register Name	Reference
23	1	TraceControl	5.2.18 "Trace Control Register (CP0 Register 23, Select 1)" on page 112
23	2	TraceControl2	5.2.19 "Trace Control2 Register (CP0 Register 23, Select 2)" on page 114
23	3	UserTraceData	5.2.20 "User Trace Data Register (CP0 Register 23, Select 3)"
23	4	TraceBPC	5.2.21 "TraceBPC Register (CP0 Register 23, Select 4)"

8.11 Trace Control Block (TCB) Registers (Hardware Control)

The TCB registers used to control its operation are listed in Table 8.37 and Table 8.38. These registers are accessed via the EJTAG TAP interface.

Table 8.37 TCB EJTAG registers

EJTAG Register	Name	Reference	Implemented
0x10	TCBCONTROLA	8.11.1 “TCBCONTROLA Register” on page 180	Yes
0x11	TCBCONTROLB	8.11.2 “TCBCONTROLB Register” on page 183	Yes
0x12	TCBDATA	8.11.3 “TCBDATA Register” on page 187	Yes

Table 8.38 Registers selected by *TCBCONTROLB*

TCBCONTROLB$_{REG}$ field	Name	Reference	Implemented
0	TCBCONFIG	8.11.4 “TCBCONFIG Register (Reg 0)” on page 188	Yes
4	TCBTW	8.11.5 “TCBTW Register (Reg 4)” on page 189	Yes if on-chip memory exists. Otherwise No
5	TCBRDP	8.11.6 “TCBRDP Register (Reg 5)” on page 190	
6	TCBWRP	8.11.7 “TCBWRP Register (Reg 6)” on page 190	
7	TCBSTP	8.11.8 “TCBSTP Register (Reg 7)” on page 190	
16-23	TCBTRIGx	8.11.9 “TCBTRIGx Register (Reg 16-23)” on page 191	Only the number indicated by *TCBCONFIG*$_{TRIG}$ are implemented.

8.11.1 *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 8.39.

***TCBCONTROLA* Register Format**

31 26	25 24	23	22 20	19	18	17	16	15	14	13	12 5	4	3 1	0
0	VModes	ADW	SyP	TB	IO	D	E	0	K	U	ASID	G	Mode	On

Table 8.39 *TCBCONTROLA* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
0	31:26	Reserved. Must be written as zero; returns zero on read.	R	0

Table 8.39 *TCBCONTROLA* Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>VModes</td><td>25:24</td><td>This field specifies the type of tracing that is supported by the processor, as follows:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<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>
</table>
This field is preset to the value of <i>PDO_ValidModes</i>.</td><td>R</td><td>10</td></tr>
<tr><td>ADW</td><td>23</td><td><i>PDO_AD</i> bus width.
0: The <i>PDO_AD</i> bus is 16 bits wide.
1: The <i>PDO_AD</i> bus is 32 bits wide.</td><td>R</td><td>0</td></tr>
<tr><td>SyP</td><td>22:20</td><td>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, when the trace buffer is either on-chip or off-chip (as determined by the <i>TCBCONTROLB</i>$_{OfC}$ bit).
<table>
<tr><th>SyP</th><th>On-chip</th><th>Off-chip</th></tr>
<tr><td>000</td><td>2^{2}</td><td>2^{7}</td></tr>
<tr><td>001</td><td>2^{3}</td><td>2^{8}</td></tr>
<tr><td>010</td><td>2^{4}</td><td>2^{9}</td></tr>
<tr><td>011</td><td>2^{5}</td><td>2^{10}</td></tr>
<tr><td>100</td><td>2^{6}</td><td>2^{11}</td></tr>
<tr><td>101</td><td>2^{7}</td><td>2^{12}</td></tr>
<tr><td>110</td><td>2^{8}</td><td>2^{13}</td></tr>
<tr><td>111</td><td>2^{9}</td><td>2^{14}</td></tr>
</table>
This field defines the value on the <i>PDI_SyncPeriod</i> signal.</td><td>R/W</td><td>100</td></tr>
<tr><td>TB</td><td>19</td><td>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.
This field defines the value on the <i>PDI_TraceAllBranch</i> signal.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>IO</td><td>18</td><td>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.
This field defines the value on the <i>PDI_InhibitOverflow</i> signal.</td><td>R/W</td><td>Undefined</td></tr>
</table>

Table 8.39 *TCBCONTROLA* Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
D	17	When set to one, this enables tracing in Debug mode, i.e., when the DM bit is one in the *Debug* register. For trace to be enabled in Debug mode, the On bit must be one. When set to zero, trace is disabled in Debug mode, irrespective of other bits. This field defines the value on the *PDI_DM* signal.	R/W	Undefined
E	16	This controls when tracing is enabled. When set, tracing is enabled when either of the EXL or ERL bits in the *Status* register is one, provided that the On bit (bit 0) is also se. This field defines the value on the *PDI_E* signal.	R/W	Undefined
0	15	Reserved. Must be written as zero; returns zero on read.	R	0
K	14	When set, this enables tracing when the On bit is set and the core is in Kernel mode. Unlike the usual definition of Kernel Mode, this bit enables tracing only when the ERL and EXL bits in the *Status* register are zero. This is provided the On bit (bit 0) is also set. This field defines the value on the *PDI_K* signal.	R/W	Undefined
U	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 On bit (bit 0) is also set. This field defines the value on the *PDI_U* signal.	R/W	Undefined
ASID	12:5	The ASID field to match when the G bit is zero. When the G bit is one, this field is ignored. This field is ignored on the M4K core because there is no ASID. This field defines the value on the *PDI_ASID* signal.	R/W	Undefined
G	4	When set, this implies that tracing is to be enabled for all processes, provided that other enabling functions (like U, S, etc.,) are also true. This field is ignored on the M4K core because there is no ASID. This field defines the value on the *PDI_G* signal.	R/W	Undefined

Table 8.39 *TCBCONTROLA* Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Mode</td><td>3:1</td><td>When tracing is turned on, this signal specifies what information is to be traced by the core.
<table>
<tr><th>Mode</th><th>Trace Mode</th></tr>
<tr><td>000</td><td>Trace PC</td></tr>
<tr><td>001</td><td>Trace PC and load address</td></tr>
<tr><td>010</td><td>Trace PC and store address</td></tr>
<tr><td>011</td><td>Trace PC and both load/store addresses</td></tr>
<tr><td>100</td><td>Trace PC and load data</td></tr>
<tr><td>101</td><td>Trace PC and load address and data</td></tr>
<tr><td>110</td><td>Trace PC and store address and data</td></tr>
<tr><td>111</td><td>Trace PC and both load/store address and data</td></tr>
</table>
The VModes field determines which of these encodings are supported by the processor. The operation of the processor is <b>UNPREDICTABLE</b> if Mode is set to a value which is not supported by the processor
This field defines the value on the <i>PDI_TraceMode</i> signal.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>On</td><td>0</td><td>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 of the <i>PDI_*</i> input pins.
When set to one, tracing is enabled whenever the other enabling functions are also true.
This field defines the value on the <i>PDI_TraceOn</i> signal.</td><td>R/W</td><td>0</td></tr>
</table>

8.11.2 *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 8.40.

TCBCONTROLB Register Format

31	30 26	25 21	20	19 17	16	15	14	13 12	11	10 8	7	6 3	2	1	0
WE	0	REG	WR	0	RM	TR	BF	TM	0	CR	Cal	0	CA	OfC	EN

Table 8.40 *TCBCONTROLB* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
WE	31	Write Enable. Only when set to 1 will the other bits be written in *TCBCONTROLB*. This bit will always read 0.	R	0
0	30:26	Reserved. Must be written as zero; returns zero on read.	R	0

Table 8.40 *TCBCONTROLB* Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
REG	25:21	Register select: This field select the registers accessible through the *TCBDATA* register. Legal values are shown in Table 8.38.	R/W	0
WR	20	Write Registers: When set, the register selected by REG field is read and written when *TCBDATA* is accessed. Otherwise the selected register is only read.	R/W	0
0	19:17	Reserved. Must be written as zero; returns zero on read.	R	0
RM	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 *TCBTW* register (through the *TCBDATA* register), will automatically increment the read pointer (*TCBRDP* register) after each read. [Note: The read pointer does not auto-increment if the WR field is one.] When the write pointer is reached, this bit is automatically reset to 0, and the *TCBTW* register will read all zeros. Once set to 1, writing 1 again will have no effect. The bit is reset by setting the TR bit or by reading the last Trace word in *TCBTW*. This bit is reserved if on-chip memory is not implemented.	R/W1	0
TR	15	Trace memory reset. When written to one, the address pointers for the on-chip trace memory are reset to zero. Also the RM 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
BF	14	Buffer Full indicator that the TCB uses to communicate to external software in the situation that the on-chip trace memory is being deployed in the **trace-from** and **trace-to** mode. (See 8.15 “TCB On-Chip Trace Memory”) This bit is cleared when writing 1 to the TR bit This bit is reserved if on-chip memory is not implemented.	R	0

Table 8.40 *TCBCONTROLB* Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>TM</td><td>13:12</td><td>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>
<tr><th>TM</th><th>Trace Mode</th></tr>
<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>
</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 *PDO_IamTracing* is asserted, and until the on-chip trace memory is full.
In both cases, de-asserting the EN bit in this register will also stop fill to the trace memory.
If a *TCBTRIGx* 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.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>11</td><td>Reserved. Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>CR</td><td>10:8</td><td>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 8.41.
<b>Remark:</b> 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.</td><td>R/W</td><td>100</td></tr>
</table>

Table 8.40 *TCBCONTROLB* Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Cal</td><td>7</td><td>Calibrate off-chip trace interface.
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.
<table>
<tr><th></th><th colspan="4">Calibrations pattern</th></tr>
<tr><td></td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td rowspan="13">This pattern is replicated for every 4 bits of <em>TR_DATA</em> pins.</td><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>
</table>
<b>Note:</b> The clock source of the TCB and PIB must be running.
This bit is reserved if off-chip trace option is not implemented.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>6:3</td><td>Reserved. Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>CA</td><td>2</td><td>Cycle accurate trace.
When set to 1, the trace will include stall information.
When set to 0, the trace will exclude stall information, and remove bit zero from all transmitted TF’s.
The stall information included/excluded is:
• TF6 formats with TCBcode 0001 and 0101.
• All TF1 formats.</td><td>R/W</td><td>0</td></tr>
<tr><td>OfC</td><td>1</td><td>If set to 1, trace is sent to off-chip memory using <em>TR_DATA</em> pins.
If set to 0, trace info is sent to on-chip memory.
This bit is read only if a single memory option exists (either off-chip or on-chip only).</td><td>R/W</td><td>Preset</td></tr>
</table>

Table 8.40 *TCBCONTROLB* Register Field Descriptions (Continued)

Fields				
Name	**Bits**	**Description**	**Read/Write**	**Reset State**
EN	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 *PDO_** pins. 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 *PDO_** pins is ignored. A potential TF6-stop (from a stop trigger) is generated as the last information, the TCB pipe-line is flushed, and trace output is stopped.	R/W	0

Table 8.41 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)

8.11.3 *TCBDATA* Register

The *TCBDATA* register (0x12) is used to access the registers defined by the *TCBCONTROLB*$_{REG}$ field; see Table 8.38. 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.

The format of the *TCBDATA* register is shown below, and the field is described in Table 8.42. The width of *TCBDATA* is 64 bits when on-chip trace words (TWs) are accessed (*TCBTW* access).

TCBDATA Register Format

31(63) 0
Data

Table 8.42 *TCBDATA* Register Field Descriptions

Fields				
Names	Bits	Description	Read/Write	Reset State
Data	31:0 63:0	Register fields or data as defined by the *TCBCONTROLB*$_{REG}$ field	Only writable if *TCBCONTROLB*$_{WR}$ is set	0

8.11.4 *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 8.43.

***TCBCONFIG* Register Format**

31	30 25	24 21	20 17	16 14	13 11	10 9	8 6	5	4	3 0
CF1	0	TRIG	SZ	CRMax	CRMin	PW	PiN	OnT	OfT	REV

Table 8.43 *TCBCONFIG* Register Field Descriptions

Fields				
Name	Bits	Description	Read/Write	Reset State
CF1	31	This bit is set if a *TCBCONFIG1* register exists. In this revision, *TCBCONFIG1* 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
TRIG	24:21	Number of triggers implemented. This also indicates the number of *TCBTRIGx* registers that exist.	R	Preset Legal values are 0 - 8
SZ	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
CRMax	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 8.41. This bit is reserved if off-chip trace option is not implemented.	R	Preset
CRMin	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 8.41. This bit is reserved if off-chip trace option is not implemented.	R	Preset

Table 8.43 *TCBCONFIG* Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>PW</td><td>10:9</td><td>Probe Width: Number of bits available on the off-chip trace interface <i>TR_DATA</i> pins. The number of TR_DATA pins is encoded, as shown in the table.
<table>
<tr><th>PW</th><th>Number of bits used on <i>TR_DATA</i></th></tr>
<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>
</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.</td><td>R</td><td>Preset</td></tr>
<tr><td>PiN</td><td>8:6</td><td>Pipe number.
Indicates the number of execution pipelines.</td><td>R</td><td>0</td></tr>
<tr><td>OnT</td><td>5</td><td>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.</td><td>R</td><td>Preset</td></tr>
<tr><td>OfT</td><td>4</td><td>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).</td><td>R</td><td>Preset</td></tr>
<tr><td>REV</td><td>3:0</td><td>Revision of TCB. An implementation that conforms to the described architecture in this document must have revision 0.</td><td>R</td><td>0</td></tr>
</table>

8.11.5 *TCBTW* Register (Reg 4)

The *TCBTW* register is used to read Trace Words from the on-chip trace memory. The 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.

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 8.44.

***TCBTW* Register Format**

63 ... 0
Data

Table 8.44 *TCBTW* Register Field Descriptions

Fields				
Names	Bits	Description	Read/Write	Reset State
Data	63:0	Trace Word	R/W	0

8.11.6 *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 8.45. 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.

***TCBRDP* Register Format**

31 n+1	n 0
	Address

Table 8.45 *TCBRDP* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Names	Bits			
Data	31:(n+1)	Reserved. Must be written zero, reads back zero.	0	0
Address	n:0	Byte address of on-chip trace memory word.	R/W	0

8.11.7 *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 8.46. 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.

***TCBWRP* Register Format**

31 n+1	n 0
	Address

Table 8.46 *TCBWRP* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Names	Bits			
Data	31:(n+1)	Reserved. Must be written zero, reads back zero.	0	0
Address	n:0	Byte address of on-chip trace memory word.	R/W	0

8.11.8 *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, *TSBSTP* 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 8.47. 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.

TCBSTP **Register Format**

31 n+1	n 0
	Address

Table 8.47 *TCBSTP* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Names	Bits			
Data	31:(n+1)	Reserved. Must be written zero, reads back zero.	0	0
Address	n:0	Byte address of on-chip trace memory word.	R/W	0

8.11.9 *TCBTRIGx* Register (Reg 16-23)

Up to eight Trigger Control registers are possible. Each register is named *TCBTRIGx*, 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*$_{\text{TRIG}}$ 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 Chapter 8, "TCB Trigger logic" on page 197, for detailed description of trigger logic issues.

The format of the *TCBTRIGx* register is shown below, and the fields are described in Table 8.48.

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 8.48 *TCBTRIGx* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Names	Bits			
TCBinfo	31:24	TCBinfo to be used in a possible TF6 trace format when this trigger fires.	R/W	0
Trace	23	When set, generate TF6 trace information when this trigger fires. Use TCBinfo field for the TCBinfo of TF6 and use Type field for the two MSB of the TCBtype of TF6. The two LSB of TCBtype 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 *TCBTRIGx* register is written.	R/W	0
0	22:16	Reserved. Must be written as zero; returns zero on read.	R	0

Table 8.48 *TCBTRIGx* Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Names	Bits			
CHTro	15	When set, generate a single cycle strobe on *TC_ChipTrigOut* when this trigger fires.	R/W	0
PDTro	14	When set, generate a single cycle strobe on *TC_ProbeTrigOut* when this trigger fires.	R/W	0
0	13:7	Reserved. Must be written as zero; returns zero on read.	R	0
DM	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 *TCBTRIGx* register is written.	R/W	0
CHTri	5	When set, this Trigger will fire when a rising edge on *TC_ChipTrigIn* 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 *TCBTRIGx* register is written.	R/W	0
PDTri	4	When set, this Trigger will fire when a rising edge on *TC_ProbeTrigIn* 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 *TCBTRIGx* register is written.	R/W	0

Table 8.48 *TCBTRIGx* Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Names</th><th>Bits</th></tr>
<tr><td>Type</td><td>3:2</td><td>Trigger Type: The Type indicates the action to take when this trigger fires. The table below show the Type values and the Trigger action.
<table>
<tr><th>Type</th><th>Trigger action</th></tr>
<tr><td>00</td><td><b>Trigger Start:</b> Trigger start-point of trace.</td></tr>
<tr><td>01</td><td><b>Trigger End:</b> Trigger end-point of trace.</td></tr>
<tr><td>10</td><td><b>Trigger About:</b> Trigger center-point of trace.</td></tr>
<tr><td>11</td><td><b>Trigger Info:</b> No action trigger, only for trace info.</td></tr>
</table>
The actual action is to set or clear the *TCBCONTROLB*$_{\mathrm{EN}}$ bit. A Start trigger will set *TCBCONTROLB*$_{\mathrm{EN}}$, a End trigger will clear *TCBCONTROLB*$_{\mathrm{EN}}$. The About trigger will clear *TCBCONTROLB*$_{\mathrm{EN}}$ half way through the trace memory, from the trigger. The size determined by the *TCBCONFIG*$_{\mathrm{SZ}}$ field for on-chip memory. Or from the *TCBCONTROLA*$_{\mathrm{SyP}}$ field for off-chip trace.
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.
If the *TCBCONTROLB*$_{\mathrm{TM}}$ field is implemented it must be set to Trace-To mode (00), for the Type field to control on-chip trace fill. 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 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 *TCBTRIGx* register is written.</td><td>R/W</td><td>0</td></tr>
<tr><td>FO</td><td>1</td><td>Fire Once. When set, this trigger will not re-fire until the TR bit is de-asserted. When de-asserted this trigger will fire each time one of the trigger sources indicates trigger.</td><td>R/W</td><td>0</td></tr>
<tr><td>TR</td><td>0</td><td>Trigger happened. When set, this trigger fired since the TR bit was last written 0.
This bit is used to inspect whether the trigger fired since this bit was last written zero.
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.
Also when set the Type field and the Trace field will have read values which indicate if the trigger action was ever suppressed by a higher priority trigger.</td><td>R/W0</td><td>0</td></tr>
</table>

8.11.10 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.

8.12 EJTAG Trace Enabling

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.

8.12.1 Trace Trigger from EJTAG Hardware Instruction/Data Breakpoints

If hardware instruction/data simple breakpoints are implemented in the M4K 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 8.2.7.5 "Instruction Breakpoint Control n (IBCn) Register (0x1118 + n*0x100)" on page 136 and 8.2.8.5 "Data Breakpoint Control n (DBCn) Register (0x2118 + 0x100 * n)" on page 140, 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 5.2.21 "TraceBPC Register (CP0 Register 23, Select 4)" on page 117 for detail in how to define a start/stop trigger.

8.12.2 Turning On PDtrace™ Trace

Trace enabling and disabling from software is similar to the hardware method, with the exception 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:

```
(
    (
        (TraceControl$_{TS}$ and TraceControl$_{On}$) or
        ((not TraceControl$_{TS}$) and TCBCONTROLA$_{On}$)
    )
    and
    (MatchEnable or TriggerEnable)
)
```

where,

```
MatchEnable ←
(
    TraceControl$_{TS}$
    and
    (
        (TraceControl$_{U}$ and UserMode)      or
        (TraceControl$_{K}$ and KernelMode)    or
        (TraceControl$_{E}$ and ExceptionMode) or
        (TraceControl$_{D}$ and DebugMode)
    )
)
```

```
or
(
    (not $TraceControl_{TS}$)
    and
    (
        ($TCBCONTROLA_{U}$ and UserMode)       or
        ($TCBCONTROLA_{K}$ and KernelMode)     or
        ($TCBCONTROLA_{E}$ and ExceptionMode)  or
        ($TCBCONTROLA_{DM}$ and DebugMode)
    )
)
```

and where,

```
TriggerEnable ←
(
    $DBCi_{TE}$          and
    $DBS_{BS[i]}$        and
    $TraceBPC_{DE}$      and
    ($TraceBPC_{DBPOn[i]}$ = 1)
)
or
(
    $IBCi_{TE}$          and
    $IBS_{BS[i]}$        and
    $TraceBPC_{IE}$      and
    ($TraceBPC_{IBPOn[i]}$ = 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 , 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*$_{On}$ and all the other signals in the second part of expression *MatchEnable*. When using the EJTAG hardware triggers to turn trace on and off, it is best if *TCBCONTROLA*$_{On}$ is asserted and 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. Tracing can be controlled via software with the *TraceControl* register in a similar manner.

8.12.3 Turning Off PDtrace™ Trace

Trace is turned off when the following expression evaluates true:

```
(
        (TraceControl$_{TS}$ and (not TraceControl$_{On)}$)) or
        ((not TraceControl$_{TS}$) and (not TCBCONTROLA$_{On}$))
    )
    or
    (
        (not MatchEnable)       and
        (not TriggerEnable)     and
        TriggerDisable
    )
```

where,

```
    TriggerDisable ←
    (
        DBCi$_{TE}$           and
        DBS$_{BS[i]}$         and
        TraceBPC$_{DE}$       and
        (TraceBPC$_{DBPOn[i]}$ = 0)
    )
    or
    (
        IBCi$_{TE}$           and
        IBS$_{BS[i]}$         and
        TraceBPC$_{IE}$       and
        (TraceBPC$_{IBPOn[i]}$ = 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. EJTAG hardware breakpoints 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, then tracing will still be turned on. This condition is reflected in presence of the “(not TriggerEnable)” term in the expression above.

8.12.4 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, 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 8.13 “TCB Trigger logic” for details.

8.12.5 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.

8.13 TCB Trigger logic

The TCB is optionally implemented with trigger unit. If this is the case, then the *TCBCONFIG*$_{TRIG}$ field is non-zero. This section will explain some of the issues around triggers in the TCB.

8.13.1 Trigger Units Overview

TCB trigger logic features three main parts:

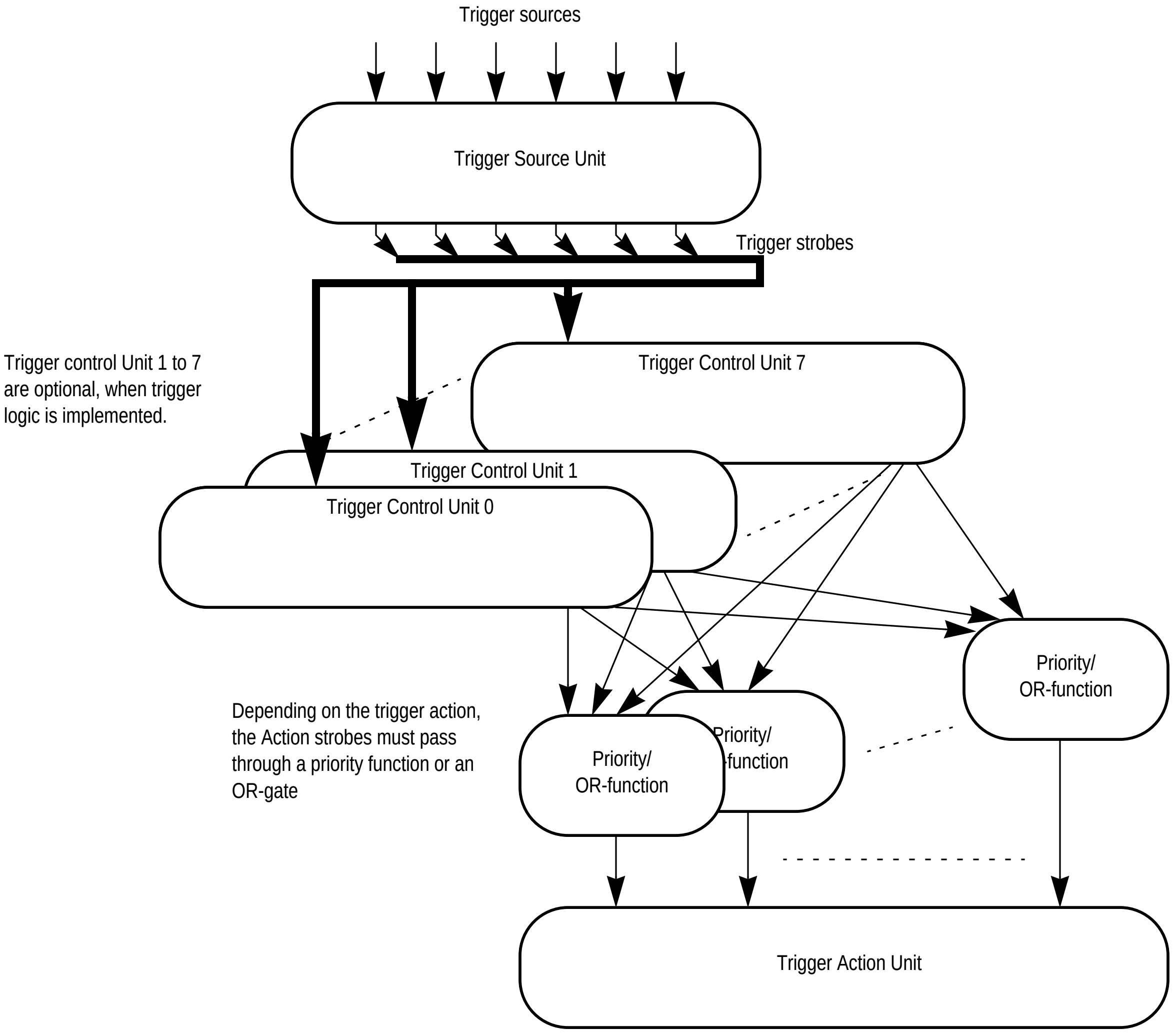
1. A common Trigger Source detection unit.

2. 1 to 8 separate Trigger Control units.

3. A common Trigger Action unit.

Figure 8.7 show the functional overview of the trigger flow in the TCB.

Figure 8.7 TCB Trigger Processing Overview



8.13.2 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.

8.13.3 Trigger Control Units

Up to eight Trigger Control Units are possible. Each of them has its own Trigger Control Register (*TCBTRIGx, 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 event and they can fire one or more of the Trigger Actions. This is all defined in the Trigger Control register *TCBTRIGx* (see 8.11.9 "TCBTRIGx Register (Reg 16-23)" on page 191).

8.13.4 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 8.11.9 "TCBTRIGx Register (Reg 16-23)" on page 191. Please also read the next 8.13.5 "Simultaneous Triggers".

8.13.5 Simultaneous Triggers

Two or more triggers can fire simultaneously. The resulting behavior depends on trigger action set for each of them, and whether they should produce a TF6 trace information output or not. There are two groups of trigger actions: Prioritized and OR'ed.

8.13.5.1 Prioritized Trigger Actions

For prioritized simultaneous trigger actions, the trigger control unit which has the lowest number takes precedence over the higher numbered units. The *x* in *TCBTRIGx* registers defines the number. The oldest trigger takes precedence over everything.

The following trigger actions are prioritized when two or more units fire simultaneously:

- Trigger Start, End and About type triggers (*TCBTRIGx*$_{Type}$ 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 *TCBTRIGx*$_{TR}$ bit is set when the trigger fires. This is so even if a trigger action is suppressed by a higher priority trigger action. If the trigger is set to only fire once (the *TCBTRIGx*$_{FO}$ bit is set), then the suppressed trigger action will not happen until after *TCBTRIGx*$_{TR}$ is written 0.

If a Trigger action is suppressed by a higher priority trigger, then the read value, when the *TCBTRIGx*$_{TR}$ bit is set, for the *TCBTRIGx*$_{Trace}$ field will be 0 for suppressed TF6 trace information actions. The read value in the *TCBTRIGx*$_{Type}$ 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 *TCBTRIGx*$_{FO}$ 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 *TCBTRIGx*$_{Type}$ field. But, if the *TCBTRIGx*$_{Trace}$ bit is set, a TF6 trace information will still go in the trace.

8.13.5.2 OR'ed Trigger Actions

The simple trigger actions CHTro and PDTro from each trigger unit, are effectively OR'ed together to produce the final trigger. One or more expected trigger strobes on i.e. *TC_ChipTrigOut* can thus disappear. External logic should not rely on counting of strobes, to predict a specific event, unless simultaneous triggers are known not to occur.

8.14 EJTAG Trace Cycle-by-Cycle Behavior

A key reason for using trace, and not single stepping to debug a software problem, is often to get a picture of the real-time behavior. However the trace logic itself can, when enabled, affect the exact cycle-by-cycle behavior,

8.14.1 Fifo Logic in PDtrace and TCB Modules

Both the PDtrace module and the TCB module contain a fifo. This might seem like extra overhead, but there are good reasons for this. The vast majority of the information compression happens in the PDtrace module. Any data information, like PC and load/store address values (delta or full), load/store data and processor mode changes, are all sent on the same 16 data bus to the TCB on the PDtrace™ interface. When an instruction requires more than 16 bits of information to be traced properly, the PDtrace fifo will buffer the information, and send it on subsequent clock cycles.

In the TCB, the on-chip trace memory is defined as a 64-bit wide synchronous memory running at core-clock speed. In this case the fifo is not needed. For off-chip trace through the Trace Probe, the fifo comes into play, because only a limited number of pins (4, 8 or 16) exist. Also the speed of the Trace Probe interface can be different (either faster or slower) from that of the M4K core. So for off-chip tracing, a specific TCB TW fifo is needed.

8.14.2 Handling of Fifo Overflow in the PDtrace Module

Depending on the amount of trace information selected for trace, and the frequency with which the 16-bit data interface is needed, it is possible for the PDtrace fifo 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*$_{\mathrm{IO}}$ or the *TCBCONTROLA*$_{\mathrm{IO}}$ bit, depending on the setting of *TraceControl*$_{\mathrm{TS}}$ 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 was 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.

If PC trace information is the only thing enabled (in *TraceControl*$_{\mathrm{MODE}}$ or *TCBCONTROLA*$_{\mathrm{MODE}}$, depending on the setting of *TraceControl*$_{\mathrm{TS}}$), and Trace of all branches is turned off (via *TraceControl*$_{\mathrm{TB}}$ or *TCBCONTROLA*$_{\mathrm{TB}}$, depending on the setting of *TraceControl*$_{\mathrm{TS}}$), then the fifo is unlikely to overflow very often, if at all. This is of course very dependent on the code executed, and the frequency of exception handler jumps, but with this setting there is very little information overhead.

8.14.3 Handling of Fifo Overflow in the TCB

The TCB also holds a fifo, used to buffer the TW's 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 to the core-clock). See 8.14.3.1 "Probe Width and Clock-Ratio Settings" for a description of probe width and clock-ratio options. The combination between the probe width (4, 8 or 16) and the data speed, allows for data rates through the trace probe from 256 bits per core-clock cycle down 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 fifo 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*$_{\mathrm{MODE}}$ or *TCBCONTROLA*$_{\mathrm{MODE}}$. 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 is only one way to handle this:

1. Prevent the overflow by asserting a stall-signal back to the core (*PDI_StallSending*). 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 8.14.2 "Handling of Fifo Overflow in the PDtrace Module" and below in 8.14.4 "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.

8.14.3.1 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. This means that the data pins (*TR_DATA*) change their value on both edges of the trace clock. When the trace clock is running at clock ratio of 1:2 (one half) of core clock, the data output registers are running a core-clock frequency. The clock ratio is set in the *TCBCONTROLB*$_{CR}$ field. The legal range for the clock ratio is defined in *TCBCONFIG*$_{CRMax}$ and *TCBCONFIG*$_{CRMin}$ (both values inclusive). If *TCBCONTROLB*$_{CR}$ is set to an unsupported value, the result is UNPREDICABLE. The maximum possible value for *TCBCONFIG*$_{CRMax}$ is 8:1 (*TR_CLK* is running 8 times faster than core-clock). The minimum possible value for *TCBCONFIG*$_{CRMin}$ is 1:8 (*TR_CLK* is running at one eighth of the core-clock). See Table 8.41 for a description of the encoding of the clock ratio fields.

8.14.4 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 module and not the PDtrace module. The extra bit therefore only affects the likelihood of the TCB fifo overflowing.

8.15 TCB On-Chip Trace Memory

When on-chip trace memory is available (*TCBCONFIG*$_{OnT}$ is set) the memory is typically of smaller size than if it were external in a trace probe. 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 (*TCBTRIGx*) 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.

8.15.1 On-Chip Trace Memory Size

The supported On-chip trace memory size can range from 256 byte to 8Mbytes, in powers of 2. The actual size is shown in the *TCBCONFIG*$_{SZ}$ field.

8.15.2 Trace-From Mode

In the Trace-From mode, tracing begins when the processor enters into a processor mode 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*$_{\mathrm{BF}}$. When external software polling this register finds the *TCBCONTROLB*$_{\mathrm{BF}}$ 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*$_{\mathrm{BF}}$ bit is reset and if the core is sending valid trace data (i.e., *PDO_IamTracing* not equal 0).

8.15.3 Trace-To Mode

In the Trace-To mode, the TCB keeps writing into the internal trace memory, wrapping over and overwriting the oldest information, until the processor is reaches an end of trace condition. End of trace is reached by leaving the processor mode which is traced, or when an EJTAG hardware breakpoint trace trigger turns tracing off. At this point, the on-chip trace buffer is then dumped out in a manner similar to that described above in 8.15.2 “Trace-From Mode”.

Chapter 9

Instruction Set Overview

This chapter provides a general overview on the three CPU instruction set formats of the MIPS architecture: Immediate, Jump, and Register. Refer to Chapter 10, “M4K™ Processor Core Instructions” on page 207 for a complete listing and description of instructions.

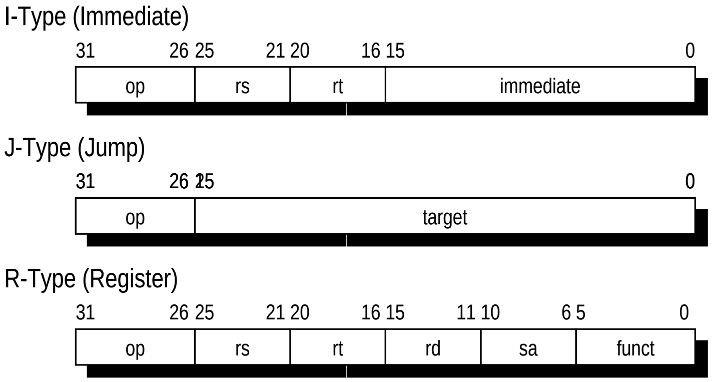
This chapter discusses the following topics

- Section 9.1 “CPU Instruction Formats”
- Section 9.2 “Load and Store Instructions”
- Section 9.3 “Computational Instructions”
- Section 9.4 “Jump and Branch Instructions”
- Section 9.5 “Control Instructions”
- Section 9.6 “Coprocessor Instructions”

9.1 CPU Instruction Formats

Each CPU instruction consists of a single 32-bit word, aligned on a word boundary. There are three instruction formats immediate (I-type), jump (J-type), and register (R-type)—as shown in Figure 9.1. 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.

Figure 9.1 Instruction Formats



op	6-bit operation code
rs	5-bit source register specifier
rt	5-bit target (source/destination) register or branch condition
immediate	16-bit immediate value, branch displacement or address displacement
target	26-bit jump target address
rd	5-bit destination register specifier
sa	5-bit shift amount
funct	6-bit function field

9.2 Load and Store Instructions

9.2.1 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 M4K 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.

9.2.2 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, define the bytes accessed within the addressed word as shown in Table 9.1. Only the combinations shown in Table 9.1 are permissible; other combinations cause address error exceptions.

Table 9.1 Byte Access Within a Word

				Bytes Accessed							
	Low Order Address Bits			Big Endian (31---------------------0)				Little Endian (31---------------------0)			
Access Type	**2**	**1**	**0**	**Byte**				**Byte**			
Word	0	0	0	0	1	2	3	3	2	1	0
Triplebyte	0	0	0	0	1	2	▨	▨	2	1	0
	0	0	1	▨	1	2	3	3	2	1	▨
Halfword	0	0	0	0	1	▨	▨	▨	▨	1	0
	0	1	0	▨	▨	2	3	3	2	▨	▨
Byte	0	0	0	0	▨	▨	▨	▨	▨	▨	0
	0	0	1	▨	1	▨	▨	▨	▨	1	▨
	0	1	0	▨	▨	2	▨	▨	2	▨	▨
	0	1	1	▨	▨	▨	3	3	▨	▨	▨

9.3 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.

Computational instructions perform the following operations on register values:

- Arithmetic
- Logical
- Shift
- Multiply
- Divide

These operations fit in the following four categories of computational instructions:

- ALU Immediate instructions
- Three-operand Register-type Instructions
- Shift Instructions
- Multiply And Divide Instructions

9.3.1 Cycle Timing for Multiply and Divide Instructions

Any multiply instruction in the integer pipeline is transferred to the multiplier as 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 M4K™ Core” on page 23 for more information on instruction latency and repeat rates.

9.4 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 (this is known as the instruction in the *delay slot*) always executes while the target instruction is being fetched from storage.

9.4.1 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 take 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 10.3 “MIPS32® Instruction Set for the M4K™ core” on page 210.

9.4.2 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.

9.5 Control Instructions

Control instructions allow the software to initiate traps; they are always R-type.

9.6 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 10, “M4K™ Processor Core Instructions” on page 207 for a listing of CP0 instructions.

Chapter 10

M4K™ Processor Core Instructions

This chapter supplements the MIPS32 Architecture Reference Manual by describing instruction behavior that is specific to a MIPS32 M4K processor core. The chapter is divided into the following sections:

- Section 10.1 “Understanding the Instruction Descriptions”
- Section 10.2 “M4K™ Opcode Map”
- Section 10.3 “MIPS32® Instruction Set for the M4K™ core”

The M4K processor core also supports the MIPS16 ASE to the MIPS32 architecture. The MIPS16 ASE instruction set is described in Chapter 11, “MIPS16e™ Application-Specific Extension to the MIPS32® Instruction Set” on page 229.

10.1 Understanding the Instruction Descriptions

Refer to Volume II of the MIPS32 Architecture Reference Manual for more information about the instruction descriptions. There is a description of the instruction fields, definition of terms, and a description function notation available in that document.

10.2 M4K™ Opcode Map

Key

- CAPITALIZED text indicates an opcode mnemonic
- *Italicized* text indicates to look at the specified opcode submap for further instruction bit decode
- Entries containing the α symbol indicate that a reserved instruction fault occurs if the core executes this instruction.
- Entries containing the β symbol indicate that a coprocessor unusable exception occurs if the core executes this instruction

Table 10.1 Encoding of the *Opcode* Field

opcode		bits 28..26							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	Special	RegImm	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	COP0	β	COP2	β	BEQL	BNEL	BLEZL	BGTZL
3	011	α	α	α	α	Special2	ϑΑΛΞ	α	Σπεχιαλ3
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	α
5	101	SB	SH	SWL	SW	α	α	SWR	CACHE
6	110	LL	β	LWC2	PREF	α	β	α	α
7	111	SC	β	SWC2	α	α	β	α	α

Table 10.2 *Special* Opcode encoding of Function Field

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	SLL	β	SRL/ ROTR	SRA	SLLV	α	SRLV/ ROTRV	SRAV
1	001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	α	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	α	α	α	α
3	011	MULT	MULTU	DIV	DIVU	α	α	α	α
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	α	α	SLT	SLTU	α	α	α	α
6	110	TGE	TGEU	TLT	TLTU	TEQ	α	TNE	α
7	111	α	α	α	α	α	α	α	α

Table 10.3 *Special2* Opcode Encoding of Function Field

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	MADD	MADDU	MUL	α	MSUB	MSUBU	α	α
1	001		α	α	α	α	α	α	α
2	010	*UDI*[1] or α							
3	011								
4	100	CLZ	CLO	α	α	α	α	α	α
5	101	α	α	α	α	α	α	α	α
6	110	α	α	α	α	α	α	α	α
7	111	α	α	α	α	α	α	α	SDBBP

[1] CorExtend instructions are a build-time option of the M4K Pro core, if not implemented this instructions space will cause a reserved instruction exception. If assembler support exists, the mnemonics for CorExtend instructions are most likely UDI0, UDI1, .., UDI15.

Table 10.4 *Special3* Opcode Encoding of Function Field

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	EXT	α	α	α	INS	α	α	α
1	001	α	α	α	α	α	α	α	α
2	010	α	α	α	α	α	α	α	α
3	011	α	α	α	α	α	α	α	α
4	100	BSHFL	α	α	α	α	α	α	α
5	101	α	α	α	α	α	α	α	α
6	110	α	α	α	α	α	α	α	α
7	111	α	α	α	ΡΔΗΩΡ	α	α	α	α

Table 10.5 *RegImm* Encoding of rt Field

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

Table 10.6 *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	α	CFC2	ΜΦΗΧ2	MTC2	α	CTC2	MTHX2
1	01	BC2	*BC2*[1]						
2	10	CO							
3	11								

[1] The core will treat the entire row as a *BC2* instruction. However compiler and assembler support only exists for the first one. Some compiler and assembler products may allow the user to add new instructions.

Table 10.7 *COP2* Encoding of rt Field When rs=*BC2*

rt	bits 16	
bits 17	0	1
0	BC2F	BC2T
1	BC2FL	BC2TL

Table 10.8 *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	α	α	α	MTC0	α	α	α
1	01	α	α	ΡΔΠΓΠΡ	ΜΦΜΧ0	α	α	ΩΡΠΓΠΡ	α
2	10	CO							
3	11								

Table 10.9 *COP0* Encoding of Function Field When rs=*CO*

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	α	α	α	α	α	α	α	α
1	001	α	α	α	α	α	α	α	α
2	010	α	α	α	α	α	α	α	α
3	011	ERET	IAXK	α	α	α	α	α	DERET
4	100	WAIT	α	α	α	α	α	α	α
5	101	α	α	α	α	α	α	α	α
6	110	α	α	α	α	α	α	α	α
7	111	α	α	α	α	α	α	α	α

10.3 MIPS32® Instruction Set for the M4K™ core

This section describes the MIPS32 instructions for the M4K cores. Table 10.10 lists the instructions in alphabetical order. Instructions that have implementation dependent behavior are described afterwards. The descriptions for other instructions exist in the architecture reference manual and are not duplicated here.

Table 10.10 Instruction Set

Instruction	Description	Function
ADD	Integer Add	Rd = Rs + Rt
ADDI	Integer Add Immediate	Rt = Rs + Immed
ADDIU	Unsigned Integer Add Immediate	Rt = Rs $+_U$ Immed
ADDU	Unsigned Integer Add	Rd = Rs $+_U$ Rt
AND	Logical AND	Rd = Rs & Rt
ANDI	Logical AND Immediate	Rt = Rs & (0_{16} || Immed)
B	Unconditional Branch (Assembler idiom for: BEQ r0, r0, offset)	PC += (int)offset

BAL	Branch and Link
	(Assembler idiom for: BGEZAL r0, offset)
		GPR[31] = PC + 8
		PC += (int)offset

BC2F	Branch On COP2 Condition False	if COP2Condition(cc) == 0
		PC += (int)offset

Table 10.10 Instruction Set (Continued)

Instruction	Description	Function

BC2FL	Branch On COP2 Condition False Likely	if COP2Condition(cc) == 0
		PC += (int)offset
		else
		Ignore Next Instruction

BC2T	Branch On COP2 Condition True	if COP2Condition(cc) == 1
		PC += (int)offset

BC2TL	Branch On COP2 Condition True Likely	if COP2Condition(cc) == 1
		PC += (int)offset
		else
		Ignore Next Instruction

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

Table 10.10 Instruction Set (Continued)

Instruction	Description	Function

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

BNEL	Branch on Not Equal Likely	if Rs != Rt
		  PC += (int)offset
		else
		  Ignore Next Instruction

BREAK	Breakpoint	Break Exception
CACHE	Cache Operation	See Cache Description
CFC2	Move Control Word From Coprocessor 2	Rt = CCR[2, n]
CLO	Count Leading Ones	Rd = NumLeadingOnes(Rs)
CLZ	Count Leading Zeroes	Rd = NumLeadingZeroes(Rs)
COP0	Coprocessor 0 Operation	See Coprocessor Description
COP2	Coprocessor 2 Operation	See Coprocessor 2 Description
CTC2	Move Control Word To Coprocessor 2	CCR[2, n] = Rt
DERET	Return from Debug Exception	PC = DEPC Exit Debug Mode
DI	Disable Interrupts	Rt=Status $\text{Status}_{\text{IE}}$=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	Stall until execution hazards are cleared
EI	Enable Interrupts	Rt=Status $\text{Status}_{\text{IE}}$=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,msbd,lsb)
INS	Insert Bit Field	Rt=InsertField(Rt,Rs,msb,lsb)

Table 10.10 Instruction Set (Continued)

Instruction	Description	Function
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

JALR.HB	Jump and Link Register with Hazard Barrier	Rd = PC + 8
		PC = Rs
		Stall until all execution and instruction hazards are cleared

JR	Jump Register	PC = Rs
JR.HB	Jump Register with Hazard Barrier	PC = Rs Stall until all execution and instruction hazards are cleared
LB	Load Byte	Rt = (byte)Mem[Rs+offset]
LBU	Unsigned Load Byte	Rt = (ubyte))Mem[Rs+offset]
LH	Load Halfword	Rt = (half)Mem[Rs+offset]
LHU	Unsigned Load Halfword	Rt = (uhalf)Mem[Rs+offset]

LL	Load Linked Word	Rt = Mem[Rs+offset]
		LL = 1
		LLAdr = Rs + offset

LUI	Load Upper Immediate	Rt = immediate << 16
LW	Load Word	Rt = Mem[Rs+offset]
LWC2	Load Word To Coprocessor 2	CPR[2, n, 0] = Mem[Rs+offset]
LWL	Load Word Left	See LWL instruction.
LWR	Load Word Right	See LWR instruction.
MADD	Multiply-Add	HI, LO += (int)Rs * (int)Rt
MFC0	Move From Coprocessor 0	Rt = CPR[0, n, sel]
MFC2	Move From Coprocessor 2	Rt = CPR[2, n, $\text{sel}_{31..0}$]
MFHC2	Move From High Word Coprocessor2	Rt= $\text{CPR[2,n,sel]}_{63..32}$
MFHI	Move From HI	Rd = HI
MFLO	Move From LO	Rd = LO

MOVN	Move Conditional on Not Zero	if GPR[rt] ≠ 0 then
		GPR[rd] = GPR[rs]

MOVZ	Move Conditional on Zero	if GPR[rt] = 0 then
		GPR[rd] = GPR[rs]

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 Coprocessor 2	$\text{CPR[2, n, sel]}_{31..0}$ = Rt
MTHC2	Move To High Word Coprocessor 2	$\text{CPR[2, n, sel]}_{63..32}$ = Rt

Table 10.10 Instruction Set (Continued)

Instruction	Description	Function
MTHI	Move To HI	HI = Rs
MTLO	Move To LO	LO = Rs
MUL	Multiply with register write	HI | LO =Unpredictable Rd = LO
MULT	Integer Multiply	HI | LO = (int)Rs * (int)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	Nop
RDHWR	Read HardWare Register	Rt=HWR[Rd]
RDPGPR	Read GPR from Previous Shadow Set	Rd=SGPR[$SRSCtl_{PSS}$, Rt]
ROTR	Rotate Word Right	Rd = $Rt_{sa-1..0}$ || $Rt_{31..sa}$
ROTRV	Rotate Word Right Variable	Rd = $Rt_{Rs-1..0}$ || $Rt_{31..Rs}$
SB	Store Byte	(byte)Mem[Rs+offset] = Rt

SC	Store Conditional Word	if LL =1
		  mem[Rxoffs] = Rt
		Rt = LL

SDBBP	Software Debug Breakpoint	Trap to SW Debug Handler
SEB	Sign Extend Byte	Rd=SignExtend($Rt_{7..0}$)
SEH	Sign Extend Half	Rd=SignExtend($Rt_{15..0}$)
SH	Store Halfword	(half)Mem[Rs+offset] = Rt
SLL	Shift Left Logical	Rd = Rt << sa
SLLV	Shift Left Logical Variable	Rd = Rt << Rs[4:0]

SLT	Set on Less Than	if (int)Rs < (int)Rt
		  Rd = 1
		else
		  Rd = 0

SLTI	Set on Less Than Immediate	if (int)Rs < (int)Immed
		  Rt = 1
		else
		  Rt = 0

SLTIU	Set on Less Than Immediate Unsigned	if (uns)Rs < (uns)Immed
		  Rt = 1
		else
		  Rt = 0

SLTU	Set on Less Than Unsigned	if (uns)Rs < (uns)Immed
		  Rd = 1
		else
		  Rd = 0

SRA	Shift Right Arithmetic	Rd = (int)Rt >> sa

Table 10.10 Instruction Set (Continued)

Instruction	Description	Function
SRAV	Shift Right Arithmetic Variable	Rd = (int)Rt >> Rs[4:0]
SRL	Shift Right Logical	Rd = (uns)Rt >> sa
SRLV	Shift Right Logical Variable	Rd = (uns)Rt >> Rs[4:0]
SSNOP	Superscalar Inhibit No Operation	Nop
SUB	Integer Subtract	Rt = (int)Rs - (int)Rd
SUBU	Unsigned Subtract	Rt = (uns)Rs - (uns)Rd
SW	Store Word	Mem[Rs+offset] = Rt
SWC2	Store Word From Coprocessor 2	Mem[Rs+offset] = CPR[2, n, 0]
SWL	Store Word Left	See SWL instruction description.
SWR	Store Word Right	See SWR instruction description.
SYNC	Synchronize	See SYNC instruction below.
SYNCI	Synchronize Caches to Make Instruction Writes Effective	Nop
SYSCALL	System Call	SystemCallException
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
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
WRPGPR	Write to GPR in Previous Shadow Set	SGPR[SRSCtl$_{PSS}$,Rd]=Rt
WSBH	Word Swap Bytes within Halfwords	Rd=SwapBytesWithinHalfs(Rt)

Table 10.10 Instruction Set (Continued)

Instruction	Description	Function
XOR	Exclusive OR	Rd = Rs ^ Rt
XORI	Exclusive OR Immediate	Rt = Rs ^ (uns)Immed

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

Format: `CACHE op, offset(base)` **MIPS32**

Purpose: Perform Cache Operation

To perform the cache operation specified by op.

Description:

CACHE is always treated as a NOP on the M4K core (as long as access to Coprocessor 0 is enabled), since it does not contain caches.

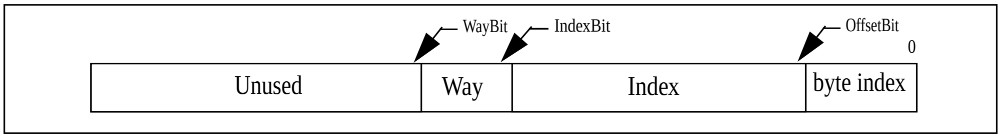
The 16-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used in one of the following ways based on the operation to be performed and the type of cache as described in the following table.

Table 10.1 Usage of Effective Address

Operation Requires an	Type of Cache	Usage of Effective Address

Index	N/A	Assuming that the total cache size in bytes is CS, the associativity is A, and the number of bytes per tag is BPT, the following calculations give the fields of the address which specify the way and the index:
		`OffsetBit ← Log2(BPT)`
		`IndexBit ← Log2(CS / A)`
		`WayBit ← IndexBit + Ceiling(Log2(A))`
		Way ← $\text{Addr}_{\text{WayBit-1..IndexBit}}$
		Index ← $\text{Addr}_{\text{IndexBit-1..OffsetBit}}$

Figure 10.1 Usage of Address Fields to Select Index and Way



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

The effective address may be an arbitrarily-aligned by address. The CACHE instruction never causes an Address Error Exception due to an non-aligned address.

A Cache Error exception may occur as a by-product of some operations performed by this instruction. For example, if a Writeback operation detects a cache or bus error during the processing of the operation, that error is reported via a Cache Error exception. Similarly, a Bus Error Exception may occur if a bus operation invoked by this instruction is terminated in an error. However, cache error exceptions must not be triggered by an Index Load Tag or Index Store tag operation, as these operations are used for initialization and diagnostic purposes.

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

Table 10.2 Encoding of Bits[17:16] of CACHE Instruction

Code	Name	Cache
0b00	I	Primary Instruction
0b01	D	Primary Data
0b10	T	
0b11	S	

Bits [20:18] of the instruction specify the operation to perform.

Table 10.3 Encoding of Bits [20:18] of the CACHE Instruction

Code	Caches	Name	Effective Address Operand Type	Operation	?
0b000	I	Index Invalidate	Index	Set the state of the cache block 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.	
	D		Index	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.	
	S, T		Index		
0b001		Index Load Tag	Index		
0b010		Index Store Tag	Index	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 *TagLo* and *TagHi* registers associated with the cache be initialized first.	
0b011	All		Unspecified		
0b100	I, D	Hit Invalidate	Address	If the cache block contains the specified address, set the state of the cache block 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.	
	S, T		Address		

Table 10.3 Encoding of Bits [20:18] of the CACHE Instruction (Continued)

Code	Caches	Name	Effective Address Operand Type	Operation	?
0b101	I	Fill	Address	Fill the cache from the specified address.	
	D		Address	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.	
	S, T		Address		
0b110	D		Address		
	S, T		Address		

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 */
```

31 26	25 21	20 16	15 0
LL 110000	base	rt	offset
6	5	5	16

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] ← 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 in **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 addressed location may be uncached for the M4K core.

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 $vAddr_{1..0} \neq 0^2$ 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.

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

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 byproduct 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 10.1 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	
4	load_streamed	Use: Prefetched data is expected to be read (not modified) but not reused extensively; it “streams” through cache.
5	store_streamed	Use: Prefetched data is expected to be stored or modified but not reused extensively; it “streams” through cache.
6	load_retained	Use: Prefetched data is expected to be read (not modified) and reused extensively; it should be “retained” in the cache.

Table 10.1 Values of *hint* Field for PREF Instruction

7	store_retained	Use: Prefetched data is expected to be stored or modified and reused extensively; it should be "retained" in the cache.
8-24	Reserved	
25	writeback_invalidate (also known as "nudge")	
26-29		
30		
31		

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 byproduct 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.

31 26	25 21	20 16	15 0
SC 111000	base	rt	offset
6	5	5	16

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 memory[GPR[base] + offset] ← GPR[rt], GPR[rt] ← 1 else GPR[rt] ← 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*. On the M4K core, the SRAM interface supports a lock protocol and the success or failure can be indicated by external hardware.

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:

```
vAddr ← sign_extend(offset) + GPR[base]
if $vAddr_{1..0} \ne 0^2$ 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] ← $0^{31}$ || 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.

31 26	25 21 20 16 15 11	10 6	5 0
SPECIAL 000000	0 00 0000 0000 0000 0	stype	SYNC 001111
6	15	5	6

Format: `SYNC (stype = 0 implied)` **MIPS32**

Purpose: Synchronize Shared Memory

To order loads and stores.

Description:

Simple Description:

- 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:

- SYNC does not guarantee the order in which instruction fetches are performed. The *stype* values 1-31 are reserved for future extensions to the architecture. A value of zero will always be defined such that it performs all defined synchronization operations. Non-zero values may be defined to remove some synchronization operations. As such, software should never use a non-zero value of the *stype* field, as this may inadvertently cause future failures if non-zero values remove synchronization operations.
- The SYNC instruction is externalized on the SRAM interface of the M4K core. External logic can use this information in a system-dependent manner to enforce memory ordering between various memory elements in the system.

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 a non-masked interrupt is taken (*SI_NMI*, *SI_Int*, or *EJ_DINT*). Note that the M4K core does not use the code field in this instruction.

If the pipeline restarts as the result of an enabled interrupt, that interrupt is taken between the WAIT instruction and the following instruction (EPC for the interrupt points at the instruction following the WAIT instruction).

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

Chapter 11

MIPS16e™ Application-Specific Extension to the MIPS32® Instruction Set

This chapter describes the MIPS16e ™ ASE as implemented in the M4K core. Refer to Volume IV-a of the MIPS32 Architecture Reference Manual for a general description of the MIPS16e ASE as well as instruction descriptions.

This chapter covers the following topics:

- Section 11.1 “Instruction Bit Encoding”
- Section 11.2 “Instruction Listing”

11.1 Instruction Bit Encoding

Table 11.2 through Table 11.9 describe the encoding used for the MIPS16e ASE. Table 11.1 describes the meaning of the symbols used in the tables.

Table 11.1 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
*	Operation or field codes marked with this symbol are reserved for future use. Executing such an instruction cause a Reserved Instruction Exception.
δ	(Also *italic* field name.) Operation or field codes marked with this symbol denotes a field class. The instruction word must be further decoded by examining additional tables that show values for another instruction field.
β	Operation or field codes marked with this symbol represent a valid encoding for a higher-order MIPS ISA level. Executing such an instruction cause 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 (*SPECIAL2* encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).
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 11.2 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 11.3 MIPS16e JAL(X) Encoding of the x Field

x	bit 26	
	0	1
	JAL	JALX

Table 11.4 MIPS16e SHIFT Encoding of the f Field

f	bits 1..0			
	0	1	2	3
	00	01	10	11
	SLL	β	SRL	SRA

Table 11.5 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 11.6 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 11.7 MIPS16e RRR Encoding of the f Field

f	bits 1..0			
	0	1	2	3
	00	01	10	11
	β	ADDU	β	SUBU

Table 11.8 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	*J(AL)R(C)* δ	SDBBP	SLT	SLTU	SLLV	BREAK	SRLV	SRAV
1	01	β	*	CMP	NEG	AND	OR	XOR	NOT
2	10	MFHI	*CNVT* δ	MFLO	β	β	*	β	β
3	11	MULT	MULTU	DIV	DIVU	β	β	β	β

Table 11.9 MIPS16e I8 Encoding of the s Field when funct=SVRS

s	bit 7	
	0	1
	RESTORE	SAVE

Table 11.10 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 11.11 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	*

11.2 Instruction Listing

Table 11.12 through 11.19 list the MIPS16e instruction set.

Table 11.12 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 11.13 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 11.14 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 11.15 MIPS16e Arithmetic Two or Three Operand Register Instructions

Mnemonic	Instruction	Extensible Instruction
ADDU	Add Unsigned	No
AND	AND	No
CMP	Compare	No

Table 11.15 MIPS16e Arithmetic Two or Three Operand Register Instructions

Mnemonic	Instruction	Extensible Instruction
MOVE	Move	No
NEG	Negate	No
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 11.16 MIPS16e Special Instructions

Mnemonic	Instruction	Extensible Instruction
BREAK	Breakpoint	No
SDBBP	Software Debug Breakpoint	No
EXTEND	Extend	No

Table 11.17 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 11.18 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
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 11.19 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

Appendix A

Revision History

Change bars (vertical lines) in the margins of this document indicate significant changes in the document since its last release. Change bars are removed for changes that 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

Revision	Date	Description
00.90	June 27, 2002	Preliminary release
01.00	August 28, 2002	• Initial commercial release. • Removed TLB-related instruction descriptions from Chapter 10, "M4K™ Processor Core Instructions" on page 207 The associated opcodes are shown as reserved in Table 10.9. • Updated HSS field in *SRSCtl* register to show possible values. • Added description of MT field in *Config* register that was previously missing. • Changed K0, KU, and K23 fields in *Config* register to be read-only, with a static value of 2.
01.01	August 29, 2002	• Removed EIC field from *IntCtl* register, per change in MIPS32 Release 2 Architecture. External interrupt controller mode is specified by $Config3_{VEIC}$.
01.02	December 15, 2003	• CP0 *Config1* register: Added CA field description, corrected typo in IS, IL, IA, DS, DL, DA field description. • Trademark updates • Replaced reference to obsolete MD00232 with MD00086 • Updated crossrefs in Status register description
01.03	October 29, 2004	• Added CorExtend Unusable exception • Added note that EJTAG accesses and external memory accesses are serialized by the core
02.00	June 22, 2006	• Corrected minor errors related to EJTAG trace. • Clarified read-only nature of several CP0 register fields and removed several references to ASID since the M4K core does not contain a TLB. • Clarified description of mapped and unmapped segments with FM-based memory management unit. • Added description on possible uses for trace triggers.
02.01	September 28, 2006	• Minor changes for addition of M4K Lite core to the M4K family.
02.02	March 21, 2008	• Fixed select number for *Debug2* register
02.03	August 29, 2008	• Fixed address for Data Value Match Register