The MIPS32™ 4KEm™ core from MIPS® Technologies is a member of the MIPS32 4KE™ processor core family. It is a high-performance, low-power, 32-bit MIPS RISC core designed 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. It is highly portable across processes, and can be easily integrated into full system-on-silicon designs, allowing developers to focus their attention on end-user products. The 4KEm core is ideally positioned to support new products for emerging segments of the digital consumer, network, systems, and information management markets, enabling new tailored solutions for embedded applications.

The 4KEm core implements the MIPS32 Release 2 Architecture with the MIPS16e™ ASE, and the 32-bit privileged resource architecture. The Memory Management Unit (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.

The synthesizable 4KEm core includes a Multiply/Divide Unit (MDU) that implements single cycle 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 2 cycles.

Instruction and data caches are fully configurable from 0 - 64 Kbytes in size. In addition, each cache can be organized as direct-mapped or 2-way, 3-way, or 4-way set associative. Load and fetch cache misses only block until the critical word becomes available. The pipeline resumes execution while the remaining words are being written to the cache. Both caches are virtually indexed and physically tagged to allow them to be accessed in the same clock that the address is translated.

An optional Enhanced JTAG (EJTAG) block allows for single-stepping of the processor as well as instruction and data virtual address/value breakpoints. Additionally, real-time tracing of instruction program counter, data address, and data values can be supported.

Figure 1 shows a block diagram of the 4KEm core. The core is divided into *required* and *optional* blocks as shown.

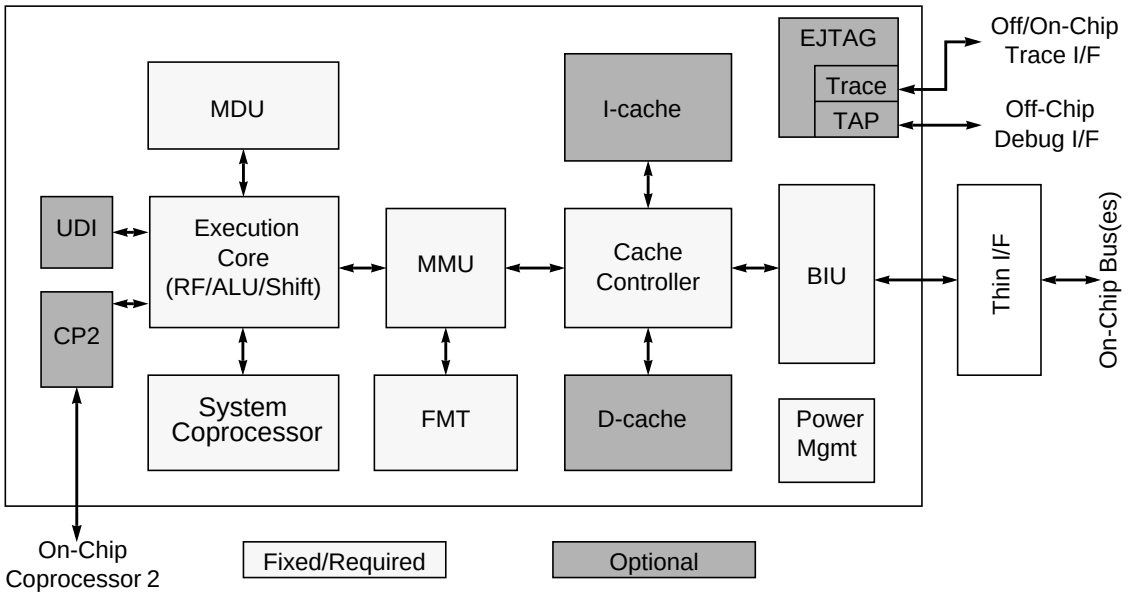


Figure 1 4KEm Core Block Diagram

Features

- 5-stage pipeline
- 32-bit Address and Data Paths
- MIPS32-Compatible Instruction Set
  - Multiply-Accumulate and Multiply-Subtract Instructions (MADD, MADDU, MSUB, MSUBU)
  - Targeted Multiply Instruction (MUL)
  - Zero/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 external interrupt controller
  - Programmable exception vector base
  - Atomic interrupt enable/disable
  - GPR shadow registers (optionally, one or three additional shadows can be added to minimize latency for interrupt handlers)
  - Bit field manipulation instructions
- MIPS16e™ Code Compression
  - 16 bit encodings of 32 bit instructions to improve code density
  - Special PC-relative instructions for efficient loading of addresses and constants
  - SAVE & RESTORE macro instructions for setting up and tearing down stack frames within subroutines
  - Improved support for handling 8 and 16 bit datatypes
- Programmable Cache Sizes
  - Individually configurable instruction and data caches
  - Sizes from 0 - 64KB
  - Direct Mapped, 2-, 3-, or 4-Way Set Associative
  - Loads block only until critical word is available
  - Write-back and write-through support
  - 16-byte cache line size
  - Virtually indexed, physically tagged
  - Cache line locking support
  - Non-blocking prefetches
- Scratchpad RAM Support
  - Can optionally replace 1 way of the I- and/or D-cache with a fast scratchpad RAM
  - Independent external pin interfaces for I- and D-scratchpads
  - 20 index address bits allow access of arrays up to 1MB
  - Interface allows back-stalling the core
- MIPS32 Privileged Resource Architecture
  - Count/Compare registers for real-time timer interrupts
  - I and D watch registers for SW breakpoints
- Memory Management Unit
  - Simple Fixed Mapping Translation (FMT) mechanism
- Simple Bus Interface Unit (BIU)
  - All I/O's fully registered
  - Separate unidirectional 32-bit address and data buses
  - Two 16-byte collapsing write buffers
  - Designed to allow easy conversion to other bus protocols
- CorExtend™ User Defined Instruction Set Extensions (available in 4KEm Pro™ core)
  - Allows user to define and add instructions to the core at build time
  - Maintains full MIPS32 compatibility
  - Supported by industry standard development tools
  - Single or multi-cycle instructions
  - Separately licensed; a core with this feature is known as the 4KEm Pro™ core
- Multiply/Divide Unit
  - Maximum issue rate of one 32x16 multiply per clock
  - Maximum issue rate of one 32x32 multiply every other clock
  - Early-in iterative divide. Minimum 11 and maximum 34 clock latency (dividend (*rs*) sign extension-dependent)
- Coprocessor 2 interface
  - 32 bit interface to an external coprocessor
- Power Control
  - Minimum frequency: 0 MHz
  - Power-down mode (triggered by WAIT instruction)
  - Support for software-controlled clock divider
  - Support for extensive use of local gated clocks
- EJTAG Debug
  - Support for single stepping
  - Virtual instruction and data address/value breakpoints
  - PC and data tracing
  - TAP controller is chainable for multi-CPU debug
  - Cross-CPU breakpoint support
- Testability
  - Full scan design achieves test coverage in excess of 99% (dependent on library and configuration options)
  - Optional memory BIST for internal SRAM arrays

Architecture Overview

The 4KEm core contains both required and optional blocks. Required blocks are the lightly shaded areas of the block diagram in Figure 1 and must be implemented to remain MIPS-compliant. Optional blocks can be added to the 4KEm core based on the needs of the implementation.

The required blocks are as follows:

- Execution Unit
- Multiply/Divide Unit (MDU)
- System Control Coprocessor (CP0)
- Memory Management Unit (MMU)
- Fixed Mapping Translation (FMT)
- Cache Controllers
- Bus Interface Unit (BIU)
- Power Management

Optional blocks include:

- Instruction Cache
- Data Cache
- Scratchpad RAM interface
- Coprocessor 2 interface
- CorExtend™ User Defined Instruction (UDI) support
- MIPS16e support
- Enhanced JTAG (EJTAG) Controller

The section entitled "4KEm Core Required Logic Blocks" on page 4 discusses the required blocks. The section entitled "4KEm Core Optional Logic Blocks" on page 11 discusses the optional blocks.

Pipeline Flow

The 4KEm core implements a 5-stage pipeline with performance similar to the R3000® pipeline. The pipeline allows the processor to achieve high frequency while minimizing device complexity, reducing both cost and power consumption.

The 4KEm core pipeline consists of five stages:

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

The 4KEm core implements a bypass mechanism that allows the result of an operation to be forwarded directly to the instruction that needs it without having to write the result to the register and then read it back.

Figure 2 shows a timing diagram of the 4KEm core pipeline.

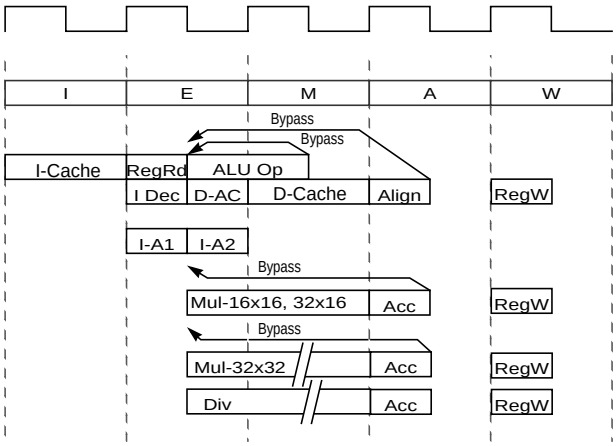


Figure 2 4KEm Core Pipeline

I Stage: Instruction Fetch

During the Instruction fetch stage:

- An instruction is fetched from instruction cache.
- MIPS16e instructions are expanded into MIPS32-like instructions

E Stage: Execution

During the Execution stage:

- Operands are fetched from register file.
- 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.
- 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.
- All multiply and divide operations begin in this stage.

M Stage: Memory Fetch

During the Memory fetch stage:

- The arithmetic ALU operation completes.
- The data cache access and the data virtual-to-physical address translation are performed for load and store instructions.
- Data cache look-up is performed and a hit/miss determination is made.

- A 16x16 or 32x16 multiply calculation completes.
- A 32x32 multiply operation stalls the MDU pipeline for one clock in the M stage.
- A divide operation stalls the MDU pipeline for a maximum of 34 clocks in the M stage. Early-in sign extension detection on the dividend will skip 7, 15, or 23 stall clocks.

A Stage: Align

During the Align stage:

- Load data is aligned to its word boundary.
- A 16x16 or 32x16 multiply operation performs the carry-propagate-add. The actual register writeback is performed in the W stage.
- A MUL operation makes the result available for writeback. The actual register writeback is performed in the W stage.

W Stage: Writeback

During the Writeback stage:

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

4KEm Core Required Logic Blocks

The 4KEm core consists of the following required logic blocks, shown in Figure 1. These logic blocks are defined in the following subsections:

- Execution Unit
- Multiply/Divide Unit (MDU)
- System Control Coprocessor (CP0)
- Memory Management Unit (MMU)
- Fixed Mapping Translation (FMT)
- Cache Controller
- Bus Interface Unit (BIU)
- Power Management

Execution Unit

The 4KEm core execution unit implements a load/store architecture with single-cycle ALU operations (logical, shift, add, subtract) and an autonomous multiply/divide unit. The 4KEm core contains thirty-two 32-bit general-purpose registers used for 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 instructions streams where data producing instructions are followed closely by consumers of their results
- Leading Zero/One detect unit for implementing the CLZ and CLO instructions
- Arithmetic Logic Unit (ALU) for performing bitwise logical operations
- Shifter & Store Aligner

Multiply/Divide Unit (MDU)

The 4KEm core includes a multiply/divide unit (MDU) that contains 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 setup allows long-running 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 recoded multiplier, result/accumulation registers (HI and LO), a divide state machine, and the 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 4KEm core only checks the value of the latter (*rt)* operand to determine how many times the operation must pass through the multiplier. The 16x16 and 32x16 operations pass through the multiplier once. A 32x32 operation passes through the multiplier twice.

The MDU supports execution of one 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 issuance of back-to-back 32x32 multiply operations. The

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. An early-in detection checks the sign extension of the dividend (*rs*) operand. If rs is 8 bits wide, 23 iterations are skipped. For a 16-bit-wide rs, 15 iterations are skipped, and for a 24-bit-wide rs, 7 iterations are skipped. 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 1 lists the repeat rate (peak issue rate of cycles until the operation can be reissued) and latency (number of cycles until a result is available) for the 4KEm core multiply and divide instructions. The approximate latency and repeat rates are listed in terms of pipeline clocks. For a more detailed discussion of latencies and repeat rates, refer to Chapter 2 of the *MIPS32 4KE™ Processor Core Family Software User's Manual*.

Table 1 4KEm Core High-Performance Integer Multiply/ Divide Unit Latencies and Repeat Rates

Opcode	Operand Size (mul *rt*) (div *rs*)	Latency	Repeat Rate
MULT/MULTU, MADD/MADDU, MSUB/MSUBU	16 bits	1	1
	32 bits	2	2
MUL	16 bits	2	1
	32 bits	3	2
DIV/DIVU	8 bits	12	11
	16 bits	19	18
	24 bits	26	25
	32 bits	33	32

The MIPS architecture defines that the result of a multiply or divide operation be placed in the HI and LO registers. Using the Move-From-HI (MFHI) and Move-From-LO (MFLO) instructions, these values can be transferred to the general-purpose register file.

In addition to the HI/LO targeted operations, the MIPS32 architecture also defines a multiply instruction, MUL, which places the least significant results in the primary 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.

Two other instructions, multiply-add (MADD) and multiply-subtract (MSUB), are used to perform the multiply-accumulate 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 and MSUB operations are commonly used in DSP algorithms.

System Control Coprocessor (CP0)

In the MIPS architecture, CP0 is responsible for the virtual-to-physical address translation and cache protocols, the exception control system, the processor's diagnostics capability, the operating modes (kernel, user, and debug), and whether interrupts are enabled or disabled. Configuration information, such as cache size and set associativity, is also available by accessing the CP0 registers, listed in Table 2.

Table 2 Coprocessor 0 Registers in Numerical Order

Register Number	Register Name	Function
0-6	Reserved	Reserved in the 4KEm core.
7	HWREna	Enables access via the RDHWR instruction to selected hardware registers.
8	BadVAddr[1]	Reports the address for the most recent address-related exception.
9	Count[1]	Processor cycle count.
10	Reserved	Reserved in the 4KEm core.
11	Compare[1]	Timer interrupt control.
12	Status[1]	Processor status and control.
12	IntCtl[1]	Interrupt system status and control.
12	SRSCtl[1]	Shadow register set status and control.
12	SRSMap[1]	Provides mapping from vectored interrupt to a shadow set.
13	Cause[1]	Cause of last general exception.
14	EPC[1]	Program counter at last exception.

Table 2 Coprocessor 0 Registers in Numerical Order

Register Number	Register Name	Function
15	PRId	Processor identification and revision.
15	EBASE	Exception vector base register.
16	Config	Configuration register.
16	Config1	Configuration register 1.
16	Config2	Configuration register 2.
16	Config3	Configuration register 3.
17	LLAddr	Load linked address.
18	WatchLo[1]	Low-order watchpoint address.
19	WatchHi[1]	High-order watchpoint address.
20-22	Reserved	Reserved in the 4KEm core.
23	Debug[2]	Debug control and exception status.
23	Trace Control[2]	PC/Data trace control register.
23	Trace Control2[2]	Additional PC/Data trace control.
23	User Trace Data[2]	User Trace control register.
23	TraceBPC[2]	Trace breakpoint control.
24	DEPC[2]	Program counter at last debug exception.
25	Reserved	Reserved in the 4KEm core.
26	ErrCtl	Used for software testing of cache arrays.
27	Reserved	Reserved in the 4KEm core.
28	TagLo/ DataLo	Low-order portion of cache tag interface.
29	Reserved	Reserved in the 4KEm 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 during debug.

Coprocessor 0 also contains the logic for identifying and managing exceptions. Exceptions can be caused by a variety of sources, including boundary cases in data, external events, or program errors. Table 3 shows the exception types in order of priority.

Table 3 4KEm Core Exception Types

Exception	Description
Reset	Assertion of *SI_ColdReset* or *SI_Reset* signals.
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	Assertion of *EB_NMI* signal.
Interrupt	Assertion of unmasked hardware or software interrupt signal.
Deferred Watch	Deferred Watch (unmasked by K|DM->!(K|DM) transition).
DIB	EJTAG debug hardware instruction break matched.
WATCH	A reference to an address in one of the watch registers (fetch).
AdEL	Fetch address alignment error. Fetch reference to protected address.
IBE	Instruction fetch bus error.
DBp	EJTAG Breakpoint (execution of SDBBP instruction).
Sys	Execution of SYSCALL instruction.
Bp	Execution of BREAK instruction.
RI	Execution of a Reserved Instruction.
CpU	Execution of a coprocessor instruction for a coprocessor that is not enabled.
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+value).
WATCH	A reference to an address in one of the watch registers (data).
AdEL	Load address alignment error. Load reference to protected address.

Table 3 4KEm Core Exception Types (Continued)

Exception	Description
AdES	Store address alignment error. Store to protected address.
DBE	Load or store bus error.
DDBL	EJTAG data hardware breakpoint matched in load data compare.

Interrupt Handling

The 4KEm core includes support for six hardware interrupt pins, two software interrupts, and a timer interrupt. These interrupts can be used in any of three interrupt modes, as defined by Release 2 of the MIPS32 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 4KEm core, so the VInt bit will always read as a 1 for the 4KEm 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 4KEm 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 4KEm core, is fully compatible with implementations of Release 1 of the Architecture.

VI or EIC interrupt modes can be combined with the optional shadow registers to specify which shadow set should be used upon entry to a particular vector. The shadow registers further improve interrupt latency by avoiding the need to save context when invoking an interrupt handler.

GPR Shadow Registers

Release 2 of the MIPS32 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.

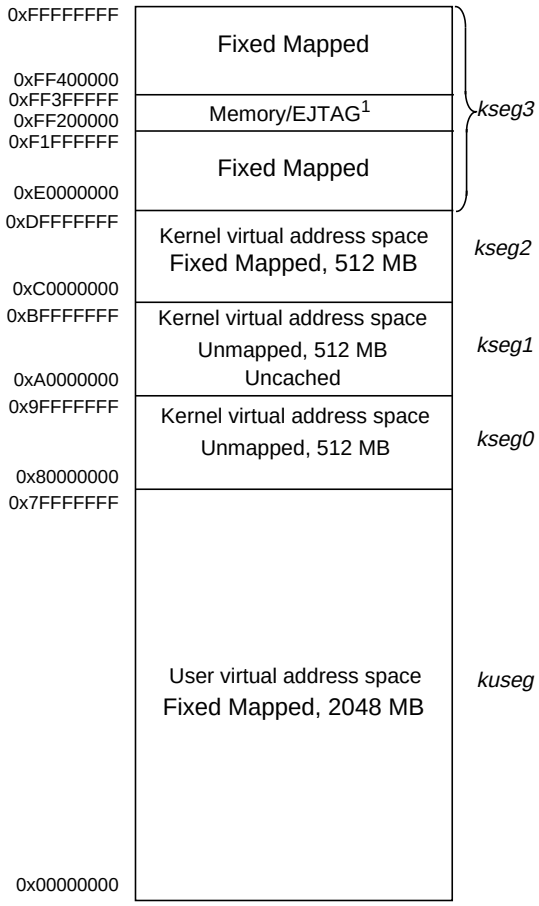
The number of GPR shadow sets is a build-time option on the 4KEm core. Although Release 2 of the Architecture defines a maximum of 16 shadow sets, the core allows one (the normal GPRs), two, or four shadow sets. The highest number actually implemented is indicated by the $SRSCtl_{HSS}$ field. If this field is zero, only the normal GPRs are implemented.

Shadow sets are new copies of the GPRs that can be substituted for the normal GPRs on entry to kernel mode via an interrupt or exception. Once a shadow set is bound to a kernel mode entry condition, reference to GPRs 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.

Modes of Operation

The 4KEm core supports three modes of operation: user mode, kernel mode, and debug mode. User mode is most often used for applications programs. Kernel mode is typically used for handling exceptions and operating system kernel functions, including CP0 management and I/O device accesses. An additional Debug mode is used during system bring-up and software development. Refer to the EJTAG section for more information on debug mode.



1. This space is mapped to memory in user or kernel mode, and by the EJTAG module in debug mode.

Figure 3 4KEm Core Virtual Address Map

Memory Management Unit (MMU)

The 4KEm core contains an MMU that interfaces between the execution unit and the cache controller. The 4KEm core provides a simple Fixed Mapping Translation (FMT) mechanism that is smaller and simpler than a full Translation Lookaside Buffer (TLB) found in other MIPS cores, like the MIPS32 4KEc™ core. Like a TLB, the FMT performs virtual-to-physical address translation and provides attributes for the different segments. Those segments that are unmapped in a TLB implementation (kseg0 and kseg1) are translated identically by the FMT.

Figure 4 shows how the FMT is implemented in the 4KEm core.

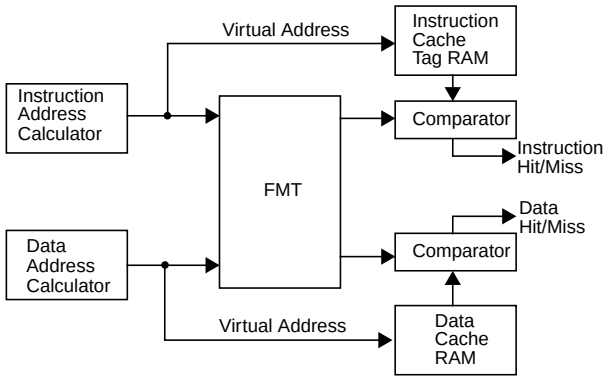


Figure 4 Address Translation During Access

In general, the FMT also determines the cacheability of each segment. These attributes are controlled via bits in the Config register. Table 4 shows the encoding for the K23 (bits 30:28), KU (bits 27:25), and K0 (bits 2:0) fields of the Config register. Table 5 shows how the cacheability of the virtual address segments is controlled by these fields.

Table 4 Cache Coherency Attributes

Config Register Fields K23, KU, and K0	Cache Coherency Attribute
0*	Cacheable, noncoherent, write-through, no write-allocate
1*	Cacheable, noncoherent, write-through, write-allocate
3, 4*, 5*, 6*	Cacheable, noncoherent, write-back, write-allocate
2, 7*	Uncached
*2 and 3 are the required MIPS32 mappings for uncached and cacheable references, other values may have different meanings in other MIPS32 processors	

In the 4KEm core, no translation exceptions can be taken, although address errors are still possible.

Table 5 Cacheability of Segments with Fixed Mapping Translation

Segment	Virtual Address Range	Cacheability
useg/kuseg	0x0000_0000-0x7FFF_FFFF	Controlled by the KU field (bits 27:25) of the Config register. See Table 4 for mapping. This segment is always uncached when ERL = 1.
kseg0	0x8000_0000-0x9FFF_FFFF	Controlled by the K0 field (bits 2:0) of the Config register. See Table 4 for mapping.
kseg1	0xA000_0000-0xBFFF_FFFF	Always uncacheable.
kseg2	0xC000_0000-0xDFFF_FFFF	Controlled by the K23 field (bits 30:28) of the Config register. See Table 4 for mapping.
kseg3	0xE000_0000-0xFFFF_FFFF	Controlled by the K23 field (bits 30:28) of the Config register. See Table 4 for mapping.

The FMT performs a simple translation to map from virtual addresses to physical addresses. This mapping is shown in Figure 5.

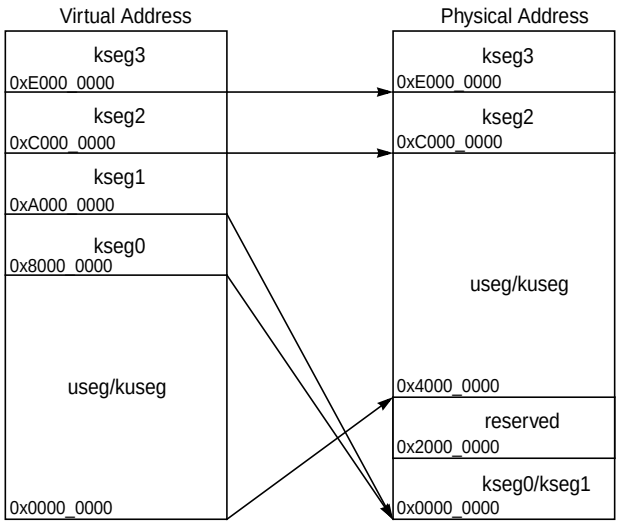


Figure 5 FMT Memory Map (ERL=0) in the 4KEm Core

When ERL=1, useg and kuseg become unmapped (virtual address is identical to the physical address) and uncached. This behavior is the same as if there was a TLB. This mapping is shown in Figure 6.

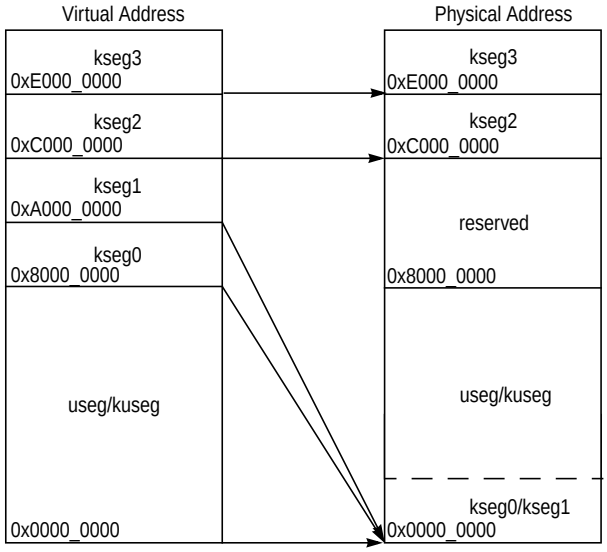


Figure 6 FMT Memory Map (ERL=1) in the 4KEm Core

Cache Controllers

The 4KEm core instruction and data cache controllers support caches of various sizes, organizations, and set-associativity. For example, the data cache can be 2 Kbytes in size and 2-way set associative, while the instruction cache can be 8 Kbytes in size and 4-way set associative. Each cache can each be accessed in a single processor cycle. In addition, each cache has its own 32-bit data path and both caches can be accessed in the same pipeline clock cycle. Refer to the section entitled "4KEm Core Optional Logic Blocks" on page 11 for more information on instruction and data cache organization.

The cache controllers also have built-in support for replacing one way of the cache with a scratchpad RAM. See the section entitled "Scratchpad RAM" on page 13 for more information on scratchpad RAMs.

Bus Interface (BIU)

The Bus Interface Unit (BIU) controls the external interface signals. Additionally, it contains the implementation of the 32-byte collapsing write buffer. The purpose of this buffer is to store and combine write transactions before issuing them at the external interface. When using the write-through cache policy, the write buffer significantly reduces the number of write transactions on the external interface and reduces the amount of stalling in

the core due to issuance of multiple writes in a short period of time. When using a write-back cache policy, the write buffer gathers the 4 words of dirty line writebacks.

The write buffer is organized as two 16-byte buffers. Each buffer contains data from a single 16-byte aligned block of memory. One buffer contains the data currently being transferred on the external interface, while the other buffer contains accumulating data from the core. Data from the accumulation buffer is transferred to the external interface buffer under one of these conditions:

- When a store is attempted from the core to a different 16-byte block than is currently being accumulated
- SYNC Instruction
- Store to an invalid merge pattern
- Any load or store to uncached memory
- A load to the line being merged
- A complete 16B block has been gathered

Note that if the data in the external interface buffer has not been written out to memory, the core is stalled until the memory write completes. After completion of the memory write, accumulated buffer data can be written to the external interface buffer.

Merge Control

The 4KEm core implements two 16-byte collapsing write buffers that allow byte, halfword, or word writes from the core to be accumulated in the buffer into a 16-byte value before bursting the data onto the bus in word format. Note that writes to uncached areas are never merged.

The 4KEm core provides two options for merge pattern control:

- No merge
- Full merge

In *No Merge* mode, writes to a different word within the same line are accumulated in the buffer. Writes to the same word cause the previous word to be driven onto the bus.

In *Full Merge* mode, all combinations of writes to the same line are collected in the buffer. Any pattern of byte enables is possible.

SimpleBE Mode

To aid in attaching the 4KEm core to structures which cannot easily handle arbitrary byte enable patterns, there is a mode that generates only “simple” byte enables. Only byte enables representing naturally aligned byte, half, and word transactions will be generated. Legal byte enable patterns are shown in Table 6.

Table 6 Valid SimpleBE Byte Enable Patterns

EB_BE[3:0]
0001
0010
0100
1000
0011
1100
1111

The only case where a read can generate “non-simple” byte enables is on an uncached tri-byte load (LWL/LWR). In SimpleBE mode, such reads will be converted into a word read on the external interface.

Writes with non-simple byte enable patterns can arise when a sequence of stores is processed by the merging write buffer, or from uncached tri-byte stores (SWL/SWR). In SimpleBE mode, these stores will be broken into two separate write transactions, one with a valid halfword and a second with a single valid byte. This splitting is independent of the merge pattern control in the write buffer.

Hardware Reset

For historical reasons within the MIPS architecture, the 4KEm core has two types of reset input signals: *SI_Reset* and *SI_ColdReset*.

Functionally, these two signals are ORed together within the core and then used to initialize critical hardware state. Both reset signals can be asserted either synchronously or asynchronously to the core clock, *SI_ClkIn*, and will trigger a Reset exception. The reset signals are active high, and must be asserted for a minimum of 5 *SI_ClkIn* cycles. The falling edge triggers the Reset exception. The primary difference between the two reset signals is that *SI_Reset* sets a bit in the Status register; this bit could be used by software to distinguish between the two reset signals, if desired. The reset behavior is summarized in Table 7.

Table 7 4KEm Reset Types

SI_Reset	SI_ColdReset	Action
0	0	Normal Operation, no reset.
1	0	Reset exception; sets *Status.SR* bit.
X	1	Reset exception.

One (or both) of the reset signals must be asserted at power-on or whenever hardware initialization of the core is desired. A power-on reset typically occurs when the machine is first turned on. A hard reset usually occurs when the machine is already on and the system is rebooted.

In debug mode, EJTAG can request that a soft reset (via the *SI_Reset* pin) be masked. It is system dependent whether this functionality is supported. In normal mode, the *SI_Reset* pin cannot be masked. The *SI_ColdReset* pin is never masked.

Power Management

The 4KEm 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 slowing or halting the clocks, which reduces system power consumption during idle periods.

The 4KEm core provides two mechanisms for system-level low power support:

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

Register-Controlled Power Management

The RP bit in the CP0 Status register provides a software mechanism for placing the system into a low power state. The state of the RP bit is available externally via the *SI_RP* signal. The external agent then decides whether to place the device in a low power mode, such as reducing the system clock frequency.

Three additional bits, $Status_{EXL}$, $Status_{ERL}$, and $Debug_{DM}$ support the power management function by allowing the user to change the power state if an exception or error occurs while the 4KEm core is in a low power state. Depending on what type of exception is taken, one of these three bits will be asserted and reflected on the *SI_EXL*, *SI_ERL*, or *EJ_DebugM* outputs. The external agent can look at these signals and determine whether to leave the low power state to service the exception.

The following 4 power-down signals are part of the system interface and change state as the corresponding bits in the CP0 registers are set or cleared:

- 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 represents the state of the DM bit (30) in the CP0 Debug register.

Instruction-Controlled Power Management

The second mechanism for invoking power-down mode is through execution of the WAIT instruction. When the WAIT instruction is executed, the internal clock is suspended; however, the internal timer and some of the input pins (*SI_Int[5:0]*, *SI_NMI*, *SI_Reset*, and *SI_ColdReset*) continue to run. Once the CPU is in instruction-controlled power management mode, any interrupt, NMI, or reset condition causes the CPU to exit this mode and resume normal operation.

The 4KEm core asserts the *SI_Sleep* signal, which is part of the system interface bus, whenever the WAIT instruction is executed. The assertion of *SI_Sleep* indicates that the clock has stopped and the 4KEm core is waiting for an interrupt.

Local clock gating

The majority of the power consumed by the 4KEm core is in the clock tree and clocking registers. The core has support for extensive use of local gated-clocks. Power conscious implementors can use these gated clocks to significantly reduce power consumption within the core.

4KEm Core Optional Logic Blocks

The 4KEm core contains several optional logic blocks shown in the block diagram in Figure 1.

Instruction Cache

The instruction cache is an optional on-chip memory block of up to 64 Kbytes. Because the instruction cache is

virtually indexed, the virtual-to-physical address translation occurs in parallel with the cache access rather than having to wait for the physical address translation. The tag holds 22 bits of physical address, a valid bit, and a lock bit. The LRU replacement bits (0-6b per set depending on associativity) are stored in a separate array.

The instruction cache block also contains and manages the instruction line fill buffer. Besides accumulating data to be written to the cache, instruction fetches that reference data in the line fill buffer are serviced either by a bypass of that data, or data coming from the external interface. The instruction cache control logic controls the bypass function.

The 4KEm core supports instruction-cache locking. Cache locking allows critical code or data segments to be locked into the cache on a “per-line” basis, enabling the system programmer to maximize the efficiency of the system cache.

The cache-locking function is always available on all instruction-cache entries. Entries can then be marked as locked or unlocked on a per entry basis using the CACHE instruction.

Data Cache

The data cache is an optional on-chip memory block of up to 64 Kbytes. This virtually indexed, physically tagged cache is protected. Because the data cache is virtually indexed, the virtual-to-physical address translation occurs in parallel with the cache access. The tag holds 22 bits of physical address, a valid bit, and a lock bit. There is an additional array holding dirty bits and LRU replacement algorithm bits (0-6b depending on associativity) for each set of the cache.

In addition to instruction-cache locking, the 4KEm core also supports a data-cache locking mechanism identical to the instruction cache. Critical data segments are locked into the cache on a “per-line” basis. The locked contents can be updated on a store hit, but cannot be selected for replacement on a cache miss.

The cache-locking function is always available on all data cache entries. Entries can then be marked as locked or unlocked on a per-entry basis using the CACHE instruction.

Cache Memory Configuration

The 4KEm core incorporates on-chip instruction and data caches that can each be accessed in a single processor cycle. Each cache has its own 32-bit data path and can be accessed in the same pipeline clock cycle. Table 8 lists the 4KEm core instruction and data cache attributes.

Table 8 4KEm Core Instruction and Data Cache Attributes

Parameter	Instruction	Data
Size	0 - 64 Kbytes	0 - 64 Kbytes
Organization	1 - 4 way set associative	1 - 4 way set associative
Line Size	16 bytes	16 bytes
Read Unit	32 bits	32 bits
Write Policies	na	write-through with write allocate, write-through without write allocate, write-back with write allocate
Miss restart after transfer of	miss word	miss word
Cache Locking	per line	per line

Cache Protocols

The 4KEm core supports the following cache protocols:

- **Uncached:** Addresses in a memory area indicated as uncached are not read from the cache. Stores to such addresses are written directly to main memory, without changing cache contents.
- **Write-through, no write allocate:** Loads and instruction fetches first search the cache, reading main memory only if the desired data does not reside in the cache. On data store operations, the cache is first searched to see if the target address is cache resident. If it is resident, the cache contents are updated, and main memory is also written. If the cache look-up misses, only main memory is written.
- **Write-through, write allocate**: Similar to above, but stores missing in the cache will cause a cache refill. The store data is then written to both the cache and main memory

- **Write-back, write allocate:** Stores that miss in the cache will cause a cache refill. Store data, however, is only written to the cache. Caches lines that are written by stores will be marked as dirty. If a dirty line is selected for replacement, the cache line will be written back to main memory.

Scratchpad RAM

The 4KEm core also supports replacing up to one way of each cache with a scratchpad RAM. Scratchpad RAM is accessed via independent external pin interfaces for instruction and data scratchpads. The external block which connects to a scratchpad interface is user-defined and can consist of a variety of devices. The main requirement is that it must be accessible with timing similar to an internal cache RAM. Normally, this means that an index will be driven one cycle, a tag will be driven the following clock, and the scratchpad must return a hit signal and the data in the second clock. The scratchpad can easily contain a large RAM/ROM or memory-mapped registers. Unlike the fixed single-cycle cache timing, however, the scratchpad interface can also accommodate backstalling the core pipeline if data is not available in a single clock. This backstalling capability can be useful for operations which require multi-cycle latency. It can also be used to enable arbitration of external accesses to a shared scratchpad memory.

The core's functional interface to a scratchpad RAM is slightly different than to a regular cache RAM. Additional index bits allow access to a larger array, 1MB of scratchpad RAM versus 4KB for a cache way. The core does not automatically refill the scratchpad way and will not select it for replacement on cache misses. Additionally, stores that hit in the scratchpad will not generate writes to main memory.

MIPS16e Application Specific Extension

The 4KEm core has 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 8-bit and 16-bit datatypes.

Coprocessor 2 Interface

The 4KEm core can be configured to have an interface for an on-chip coprocessor. This coprocessor can be tightly coupled to the processor core, allowing high performance solutions integrating a graphics accelerator or DSP, for example.

The coprocessor interface is extensible and standardized on MIPS cores, allowing for design reuse. The 4KEm core supports a subset of the full coprocessor interface standard: 32b data transfer, no Coprocessor 1 support, single issue, in-order data transfer to coprocessor, one out-of-order data transfer from coprocessor.

The coprocessor interface is designed to ease integration with customer IP. The interface allows high-performance communication between the core and coprocessor. There are no late or critical signals on the interface.

CorExtend User Defined Instruction Extensions

The optional CorExtend User Defined Instruction (UDI) block enables the implementation of a small number of application-specific instructions that are tightly coupled to the core's execution unit. The interface to the UDI block is internal and not defined externally on the 4KEm Pro core.

Such instructions may operate on a general-purpose register, immediate data specified by the instruction word, or local state stored within the UDI block. The destination may be a general-purpose register or local UDI state. The operation may complete in one cycle or multiple cycles, if desired.

EJTAG Debug Support

The 4KEm core provides for an optional Enhanced JTAG (EJTAG) interface for use in the software debug of application and kernel code. In addition to standard user mode and kernel modes of operation, the 4KEm core provides a Debug mode that is entered after a debug exception (derived from a hardware breakpoint, single-step exception, etc.) is taken and continues until a debug exception return (DERET) instruction is executed. During this time, the processor executes the debug exception handler routine.

Refer to the section called "External Interface Signals" on page 21 for a list of EJTAG interface signals.

The EJTAG interface operates through the Test Access Port (TAP), a serial communication port used for transferring test data in and out of the 4KEm core. In addition to the standard JTAG instructions, special instructions defined in the EJTAG specification define what registers are selected and how they are used.

Debug Registers

Three debug registers (DEBUG, DEPC, and DESAVE) have been added to the MIPS Coprocessor 0 (CP0) register set. The DEBUG register shows the cause of the debug exception and is used for setting up single-step operations. The DEPC, or Debug Exception Program Counter, register holds the address on which the debug exception was taken. This is used to resume program execution after the debug operation finishes. Finally, the DESAVE, or Debug Exception Save, register enables the saving of general-purpose registers used during execution of the debug exception handler.

To exit debug mode, a Debug Exception Return (DERET) instruction is executed. When this instruction is executed, the system exits debug mode, allowing normal execution of application and system code to resume.

EJTAG Hardware Breakpoints

There are several types of simple hardware breakpoints defined in the EJTAG specification. These stop the normal operation of the CPU and force the system into debug mode. There are two types of simple hardware breakpoints implemented in the 4KEm core: Instruction breakpoints and Data breakpoints.

The 4KEm core can be configured with the following breakpoint options:

- No data or instruction breakpoints
- One data and two instruction breakpoints
- Two data and four instruction breakpoints

Instruction breaks occur on instruction fetch operations, and the break is set on the virtual address. A mask can be applied to the virtual address to set breakpoints on a range of instructions.

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.

EJTAG Trace

The 4KEm core includes optional support for real-time tracing of instruction addresses, data addresses and data values. The trace information is collected in an on-chip or off-chip memory, for post-capture processing by trace regeneration software.

On-chip trace memory may be configured in size from 0 to 8 MB; it is accessed through the existing EJTAG TAP interface and requires no additional chip pins. Off-chip trace memory is accessed through a special trace probe and can be configured to use 4, 8, or 16 data pins plus a clock.

Testability

Testability for production testing of the core is supported through the use of internal scan and memory BIST.

Internal Scan

Full mux-based scan for maximum test coverage is supported, with a configurable number of scan chains. ATPG test coverage can exceed 99%, depending on standard cell libraries and configuration options.

Memory BIST

Memory BIST for the cache arrays and on-chip trace memory is optional, but can be implemented either through the use of integrated BIST features provided with the core, or inserted with an industry-standard memory BIST CAD tool.

Integrated Memory BIST

The core provides an integrated memory BIST solution for testing the internal cache SRAMs, using BIST controllers and logic tightly coupled to the cache subsystem. Several parameters associated with the integrated BIST controllers are configurable, including the algorithm (March C+ or IFA-13).

User-specified Memory BIST

Memory BIST can also be inserted with a CAD tool or other user-specified method. Wrapper modules and signal

buses of configurable width are provided within the core to facilitate this approach.

Instruction Set

The 4KEm core instruction set complies with the MIPS32 instruction set architecture. Table 9 provides a summary of instructions implemented by the 4KEm core.

Table 9 4KEm Core 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`
ADDIUPC	Unsigned Integer Add Immediate to PC (MIPS16 only)	`Rt = PC +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)`
BC2F	Branch On COP2 Condition False	`if COP2Condition(cc) == 0` `  PC += (int)offset`

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`

Table 9 4KEm Core Instruction Set (Continued)

Instruction	Description	Function

BGEZL	Branch on Greater Than or Equal To Zero Likely	`if !Rs[31]`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

BGTZ	Branch on Greater Than Zero	`if !Rs[31] && Rs != 0` `  PC += (int)offset`

BGTZL	Branch on Greater Than Zero Likely	`if !Rs[31] && Rs != 0`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

BLEZ	Branch on Less Than or Equal to Zero	`if Rs[31] || Rs == 0` `  PC += (int)offset`

BLEZL	Branch on Less Than or Equal to Zero Likely	`if Rs[31] || Rs == 0`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

BLTZ	Branch on Less Than Zero	`if Rs[31]` `  PC += (int)offset`

BLTZAL	Branch on Less Than Zero And Link	`GPR[31] = PC + 8`
		`if Rs[31]`
		`  PC += (int)offset`

BLTZALL	Branch on Less Than Zero And Link Likely	`GPR[31] = PC + 8`
		`if Rs[31]`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

BLTZL	Branch on Less Than Zero Likely	`if Rs[31]`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

BNE	Branch on Not Equal	`if Rs != Rt` `  PC += (int)offset`

BNEL	Branch on Not Equal Likely	`if Rs != Rt`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

BREAK	Breakpoint	Break Exception
CACHE	Cache Operation	See Software User's Manual
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 Software User's Manual
COP2	Coprocessor 2 Operation	See Coprocessor 2 Description
CTC2	Move Control Word To Coprocessor 2	`CCR[2, n] = Rt`

Table 9 4KEm Core Instruction Set (Continued)

Instruction	Description	Function
DERET	Return from Debug Exception	PC = DEPC Exit Debug Mode
DI	Atomically Disable Interrupts	Rt = Status; $Status_{IE}$ = 0
DIV	Divide	LO = (int)Rs / (int)Rt HI = (int)Rs % (int)Rt
DIVU	Unsigned Divide	LO = (uns)Rs / (uns)Rt HI = (uns)Rs % (uns)Rt
EHB	Execution Hazard Barrier	Stop instruction execution until execution hazards are cleared
EI	Atomically Enable Interrupts	Rt = Status; $Status_{IE}$ = 1

ERET	Return from Exception	if SR[2]
		  PC = ErrorEPC
		else
		  PC = EPC
		  SR[1] = 0
		SR[2] = 0
		LL = 0

EXT	Extract Bit Field	Rt = ExtractField(Rs, pos, size)
INS	Insert Bit Field	Rt = InsertField(Rs, Rt, pos, size)
J	Unconditional Jump	PC = PC[31:28] || offset<<2
JAL	Jump and Link	GPR[31] = PC + 8 PC = PC[31:28] || offset<<2
JALR	Jump and Link Register	Rd = PC + 8 PC = Rs
JALR.HB	Jump and Link Register with Hazard Barrier	Like JALR, but also clears execution and instruction hazards
JALRC	Jump and Link Register Compact - do not execute instruction in jump delay slot(MIPS16 only)	Rd = PC + 2 PC = Rs
JR	Jump Register	PC = Rs
JR.HB	Jump Register with Hazard Barrier	Like JR, but also clears execution and instruction hazards
JRC	Jump Register Compact - do not execute instruction in jump delay slot (MIPS16 only)	PC = Rs
LB	Load Byte	Rt = (byte)Mem[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]

Table 9 4KEm Core Instruction Set (Continued)

Instruction	Description	Function

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]`
LWPC	Load Word, PC relative	`Rt = Mem[PC+offset]`
LWL	Load Word Left	See Software User's Manual
LWR	Load Word Right	See Software User's Manual
MADD	Multiply-Add	`HI | LO += (int)Rs * (int)Rt`
MADDU	Multiply-Add Unsigned	`HI | LO += (uns)Rs * (uns)Rt`
MFC0	Move From Coprocessor 0	`Rt = CPR[0, Rd, sel]`
MFC2	Move From Coprocessor 2	`Rt = CPR[2, Rd, sel]`
MFHC2	Move From High Half of Coprocessor 2	`Rt = CPR[2, Rd, sel]`$_{63..32}$
MFHI	Move From HI	`Rd = HI`
MFLO	Move From LO	`Rd = LO`
MOVN	Move Conditional on Not Zero	`if Rt ≠ 0 then` `  Rd = Rs`
MOVZ	Move Conditional on Zero	`if Rt = 0 then` `  Rd = 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	`CPR[2, n, sel] = Rt`
MTHC2	Move To High Half of Coprocessor 2	`CPR[2, Rd, sel] = Rt ||` `CPR[2, Rd, sel]`$_{31..0}$
MTHI	Move To HI	`HI = Rs`
MTLO	Move To LO	`LO = Rs`

MUL	Multiply with register write	`HI | LO =Unpredictable`
		`Rd = ((int)Rs *`
		`(int)Rt)`$_{31..0}$

MULT	Integer Multiply	`HI | LO = (int)Rs * (int)Rd`
MULTU	Unsigned Multiply	`HI | LO = (uns)Rs * (uns)Rd`
NOR	Logical NOR	`Rd = ~(Rs | Rt)`
OR	Logical OR	`Rd = Rs | Rt`

Table 9 4KEm Core Instruction Set (Continued)

Instruction	Description	Function
ORI	Logical OR Immediate	`Rt = Rs | Immed`
PREF	Prefetch	Load Specified Line into Cache
RDHWR	Read Hardware Register	Allows unprivileged access to registers enabled by HWREna register
RDPGPR	Read GPR from Previous Shadow Set	Rt = SGPR[$SRSCtl_{PSS}$, Rd]
RESTORE	Restore registers and deallocate stack frame (MIPS16 only)	See Software User's Manual
ROTR	Rotate Word Right	`Rd =` $Rt_{sa-1..0}$ `||` $Rt_{31..sa}$
ROTRV	Rotate Word Right Variable	`Rd =` $Rt_{Rs-1..0}$ `||` $Rt_{31..Rs}$
SAVE	Save registers and allocate stack frame (MIPS16 only)	See Software User's Manual
SB	Store Byte	`(byte)Mem[Rs+offset] = Rt`

SC	Store Conditional Word	`if LL = 1`
		`  mem[Rs+offset] = Rt`
		`Rt = LL`

SDBBP	Software Debug Break Point	Trap to SW Debug Handler
SEB	Sign Extend Byte	Rd = (byte)Rs
SEH	Sign Extend Half	Rd = (half)Rs
SH	Store Half	`(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`
SRAV	Shift Right Arithmetic Variable	`Rd = (int)Rt >> Rs[4:0]`

Table 9 4KEm Core Instruction Set (Continued)

Instruction	Description	Function
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 Software User's Manual
SWR	Store Word Right	See Software User's Manual
SYNC	Synchronize	See Software User's Manual
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	$Rd = Rt_{23..16} \,||\, Rt_{31..24} \,||\, Rt_{7..0} \,||\, Rt_{15..8}$

Table 9 4KEm Core Instruction Set (Continued)

Instruction	Description	Function
XOR	Exclusive OR	`Rd = Rs ^ Rt`
XORI	Exclusive OR Immediate	`Rt = Rs ^ (uns)Immed`
ZEB	Zero extend byte (MIPS16 only)	`Rt = (ubyte) Rs`
ZEH	Zero extend half (MIPS16 only)	`Rt = (uhalf) Rs`

External Interface Signals

This section describes the signal interface of the 4KEm microprocessor core.

The pin direction key for the signal descriptions is shown in Table 10 below.

The 4KEm core signals are listed in Table 11 below. Note that the signals are grouped by logical function, not by expected physical location. All signals, with the exception of *EJ_TRST_N*, are active-high signals. *EJ_DINT* and *SI_NMI* go through edge-detection logic so that only one exception is taken each time they are asserted.

Table 10 4KEm Core Signal Direction Key

Dir	Description
I	Input to the 4KEm core sampled on the rising edge of the appropriate CLK signal.
O	Output of the 4KEm core, unless otherwise noted, driven at the rising edge of the appropriate CLK signal.
A	Asynchronous inputs that are synchronized by the core.
S	Static input to the 4KEm core. These signals are normally tied to either power or ground and should not change state while *SI_ColdReset* is deasserted.

Table 11 4KEm Signal Descriptions

Signal Name	Type	Description
System Interface		
Clock Signals:		
SI_ClkIn	I	Clock Input. All inputs and outputs, except a few of the EJTAG signals, are sampled and/or asserted relative to the rising edge of this signal.
SI_ClkOut	O	Reference Clock for the External Bus Interface. This clock signal provides a reference for deskewing any clock insertion delay created by the internal clock buffering in the core.
Reset Signals:		
SI_ColdReset	A	Hard/Cold Reset Signal. Causes a Reset Exception in the core.
SI_NMI	A	Non-Maskable Interrupt. An edge detect is used on this signal. When this signal is sampled asserted (high) one clock after being sampled deasserted, an NMI is posted to the core.
SI_Reset	A	Soft/Warm Reset Signal. Causes a Reset Exception in the core. Sets Status.SR bit (if *SI_ColdReset* is not asserted), but is otherwise ORed with *SI_ColdReset* before it is used internally.
Power Management Signals:		

Table 11 4KEm Signal Descriptions (Continued)

Signal Name	Type	Description
SI_ERL	O	This signal represents the state of the ERL bit (2) in the CP0 Status register and indicates the error level. The core asserts *SI_ERL* whenever a Reset, Soft Reset, or NMI exception is taken.
SI_EXL	O	This signal represents the state of the EXL bit (1) in the CP0 Status register and indicates the exception level. The core asserts *SI_EXL* whenever any exception other than a Reset, Soft Reset, NMI, or Debug exception is taken.
SI_RP	O	This signal represents the state of the RP bit (27) in the CP0 Status register. Software can write this bit to indicate that a reduced power mode may be entered.
SI_Sleep	O	This signal is asserted by the core whenever the WAIT instruction is executed. The assertion of this signal indicates that the clock has stopped and that the core is waiting for an interrupt.
Interrupt Signals:		
SI_EICPresent	S	Indicates whether an external interrupt controller is present. Value is visible to software in the $Config3_{VEIC}$ register field.
SI_EISS[3:0]	I	General purpose register shadow set number to be used when servicing an interrupt in EIC interrupt mode.
SI_IAck	O	Interrupt acknowledge indication for use in external interrupt controller mode. This signal is active for a single *SI_ClkIn* cycle when an interrupt is taken. When the processor initiates the interrupt exception, it loads the value of the *SI_Int[5:0]* pins into the $Cause_{RIPL}$ field (overlaid with $Cause_{IP7..IP2}$), and signals the external interrupt controller to notify it that the current interrupt request is being serviced. This allows the controller to advance to another pending higher-priority interrupt, if desired.
SI_Int[5:0]	I/A	Active high Interrupt pins. These signals are driven by external logic and when asserted indicate an interrupt exception to the core. The interpretation of these signals depends on the interrupt mode in which the core is operating; the interrupt mode is selected by software. The *SI_Int* signals go through synchronization logic and can be asserted asynchronously to *SI_ClkIn.* In External Interrupt Controller (EIC) mode, however, the interrupt pins are interpreted as an encoded value, so they must be asserted synchronously to *SI_ClkIn* to guarantee that all bits are received by the core in a particular cycle. The interrupt pins are level sensitive and should remain asserted until the interrupt has been serviced. In Release 1 Interrupt Compatibility mode: • All 6 interrupt pins have the same priority as far as the hardware is concerned. • Interrupts are non-vectored. In Vectored Interrupt (VI) mode: • The *SI_Int* pins are interpreted as individual hardware interrupt requests. • Internally, the core prioritizes the hardware interrupts and chooses an interrupt vector. In External Interrupt Controller (EIC) mode: • An external block prioritizes its various interrupt requests and produces a vector number of the highest priority interrupt to be serviced. • The vector number is driven on the *SI_Int* pins, and is treated as a 6-bit encoded value in the range of 0..63. • When the core starts the interrupt exception, signaled by the assertion of *SI_IAck*, it loads the value of the *SI_Int[5:0]* pins into the $Cause_{RIPL}$ field (overlaid with $Cause_{IP7..IP2}$). The interrupt controller can then signal another interrupt.

Table 11 4KEm Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td>SI_IPL[5:0]</td><td>O</td><td>Current interrupt priority level from the $Status_{IPL}$ register field, provided for use by an external interrupt controller. This value is updated whenever SI_IAck is asserted.</td></tr>
<tr><td>SI_IPTI[2:0]</td><td>S</td><td>Indicates the SI_Int hardware interrupt pin that the timer interrupt pin (SI_TimerInt) is combined with external to the core. The value of this bus is visible to software in the $IntCtl_{IPTI}$ register field.</td></tr>
<tr><td>SI_SWInt[1:0]</td><td>O</td><td>Software interrupt request. These signals represent the value in the IP[1:0] field of the Cause register. They are provided for use by an external interrupt controller.</td></tr>
<tr><td>SI_TimerInt</td><td>O</td><td>Timer interrupt indication. This signal is asserted whenever the Count and Compare registers match and is deasserted when the Compare register is written. This hardware pin represents the value of the $Cause_{TI}$ register field.

For Release 1 Interrupt Compatibility mode or Vectored Interrupt mode:

In order to generate a timer interrupt, the SI_TimerInt signal needs to be brought back into the 4KEm core on one of the six SI_Int interrupt pins in a system-dependent manner. Traditionally, this has been accomplished by muxing SI_TimerInt with SI_Int[5]. Exposing SI_TimerInt as an output allows more flexibility for the system designer. Timer interrupts can be muxed or ORed into one of the interrupts, as desired in a particular system. The SI_Int hardware interrupt pin with which the SI_TimerInt signal is merged is indicated via the SI_IPTI static input pins.

For External Interrupt Controller (EIC) mode:

The SI_TimerInt signal is provided to the external interrupt controller, which then prioritizes the timer interrupt with all other interrupt sources, as desired. The controller then encodes the desired interrupt value on the SI_Int pins. Since SI_Int is usually encoded, the SI_IPTI pins are not meaningful in EIC mode.</td></tr>
<tr><td colspan="3">Configuration Inputs:</td></tr>
<tr><td>SI_CPUNum[9:0]</td><td>S</td><td>Unique identifier to specify an individual core in a multi-processor system. The hardware value specified on these pins is available in the CPUNum field of the EBase register, so it can be used by software to distinguish a particular processor. In a single processor system, this value should be set to zero.</td></tr>
<tr><td>SI_Endian</td><td>S</td><td>Indicates the base endianness of the core.

<table>
<tr><th>EB_Endian</th><th>Base Endian Mode</th></tr>
<tr><td>0</td><td>Little Endian</td></tr>
<tr><td>1</td><td>Big Endian</td></tr>
</table>
</td></tr>
<tr><td>SI_MergeMode[1:0]</td><td>S</td><td>The state of these signals determines the merge mode for the 16-byte collapsing write buffer.

<table>
<tr><th>Encoding</th><th>Merge Mode</th></tr>
<tr><td>00_2</td><td>No Merge</td></tr>
<tr><td>01_2</td><td>Reserved</td></tr>
<tr><td>10_2</td><td>Full Merge</td></tr>
<tr><td>11_2</td><td>Reserved</td></tr>
</table>
</td></tr>
</table>

Table 11 4KEm Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td>SI_SimpleBE[1:0]</td><td>S</td><td>The state of these signals can constrain the core to only generate certain byte enables on EC™ interface transactions. This eases connection to some existing bus standards.

<table><tr><th>SI_SimpleBE[1:0]</th><th>Byte Enable Mode</th></tr><tr><td>00_2</td><td>All BEs allowed</td></tr><tr><td>01_2</td><td>Naturally aligned bytes, half-words, and words only</td></tr><tr><td>10_2</td><td>Reserved</td></tr><tr><td>11_2</td><td>Reserved</td></tr></table></td></tr>
<tr><td colspan="3">External Bus Interface</td></tr>
<tr><td>EB_ARdy</td><td>I</td><td>Indicates whether the target is ready for a new address. The core will not complete the address phase of a new bus transaction until the clock cycle after EB_ARdy is sampled asserted.</td></tr>
<tr><td>EB_AValid</td><td>O</td><td>When asserted, indicates that the values on the address bus and access types lines are valid, signifying the beginning of a new bus transaction. EB_AValid must always be valid.</td></tr>
<tr><td>EB_Instr</td><td>O</td><td>When asserted, indicates that the transaction is an instruction fetch versus a data reference. EB_Instr is only valid when EB_AValid is asserted.</td></tr>
<tr><td>EB_Write</td><td>O</td><td>When asserted, indicates that the current transaction is a write. This signal is only valid when EB_AValid is asserted.</td></tr>
<tr><td>EB_Burst</td><td>O</td><td>When asserted, indicates that the current transaction is part of a cache fill or a write burst. Note that there is redundant information contained in EB_Burst, EB_BFirst, EB_BLast, and EB_BLen. This is done to simplify the system design—the information can be used in whatever form is easiest.</td></tr>
<tr><td>EB_BFirst</td><td>O</td><td>When asserted, indicates the beginning of the burst. EB_BFirst is always valid.</td></tr>
<tr><td>EB_BLast</td><td>O</td><td>When asserted, indicates the end of the burst. EB_BLast is always valid.</td></tr>
<tr><td>EB_BLen[1:0]</td><td>O</td><td>Indicates the length of the burst. This signal is only valid when EB_AValid is asserted.

<table><tr><th>EB_BLength[1:0]</th><th>Burst Length</th></tr><tr><td>0</td><td>reserved</td></tr><tr><td>1</td><td>4</td></tr><tr><td>2</td><td>reserved</td></tr><tr><td>3</td><td>reserved</td></tr></table></td></tr>
<tr><td>EB_SBlock</td><td>S</td><td>Static input which determines burst order. When asserted, sub-block ordering is used. When deasserted, sequential addressing is used.</td></tr>
</table>

Table 11 4KEm Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td>EB_BE[3:0]</td><td>O</td><td>Indicates which bytes of the EB_RData or EB_WData buses are involved in the current transaction. If an EB_BE signal is asserted, the associated byte is being read or written. EB_BE lines are only valid while EB_AValid is asserted.
<table>
<tr><th>EB_BE Signal</th><th>Read Data Bits Sampled</th><th>Write Data Bits Driven Valid</th></tr>
<tr><td>EB_BE[0]</td><td>EB_RData[7:0]</td><td>EB_WData[7:0]</td></tr>
<tr><td>EB_BE[1]</td><td>EB_RData[15:8]</td><td>EB_WData[15:8]</td></tr>
<tr><td>EB_BE[2]</td><td>EB_RData[23:16]</td><td>EB_WData[23:16]</td></tr>
<tr><td>EB_BE[3]</td><td>EB_RData[31:24]</td><td>EB_WData[31:24]</td></tr>
</table></td></tr>
<tr><td>EB_A[35:2]</td><td>O</td><td>Address lines for external bus. Only valid when EB_AValid is asserted. EB_A[35:32] are tied to 0 in this core.</td></tr>
<tr><td>EB_WData[31:0]</td><td>O</td><td>Output data for writes.</td></tr>
<tr><td>EB_RData[31:0]</td><td>I</td><td>Input Data for reads.</td></tr>
<tr><td>EB_RdVal</td><td>I</td><td>Indicates that the target is driving read data on EB_RData lines. EB_RdVal must always be valid. EB_RdVal may never be sampled asserted until the rising edge after the corresponding EB_ARdy was sampled asserted.</td></tr>
<tr><td>EB_WDRdy</td><td>I</td><td>Indicates that the target of a write is ready. The EB_WData lines can change in the next clock cycle. EB_WDRdy will not be sampled until the rising edge where the corresponding EB_ARdy is sampled asserted.</td></tr>
<tr><td>EB_RBErr</td><td>I</td><td>Bus error indicator for read transactions. EB_RBErr is sampled on every rising clock edge until an active sampling of EB_RdVal. EB_RBErr sampled with asserted EB_RdVal indicates a bus error during read. EB_RBErr must be deasserted in idle phases.</td></tr>
<tr><td>EB_WBErr</td><td>I</td><td>Bus error indicator for write transactions. EB_WBErr is sampled on the rising clock edge following an active sample of EB_WDRdy. EB_WBErr must be deasserted in idle phases.</td></tr>
<tr><td>EB_EWBE</td><td>I</td><td>Indicates that any external write buffers are empty. The external write buffers must deassert EB_EWBE in the cycle after the corresponding EB_WDRdy is asserted and keep EB_EWBE deasserted until the external write buffers are empty.</td></tr>
<tr><td>EB_WWBE</td><td>O</td><td>When asserted, indicates that the core is waiting for external write buffers to empty.</td></tr>
<tr><td colspan="3">Coprocessor Interface</td></tr>
<tr><td colspan="3">Instruction dispatch: These signals are used to transfer an instruction from the 4KEm core to the COP2 coprocessor.</td></tr>
<tr><td>CP2_ir_0[31:0]</td><td>O</td><td>Coprocessor Arithmetic and To/From Instruction Word.
Valid in the cycle before CP2_as_0, CP2_ts_0 or CP2_fs_0 is asserted.</td></tr>
<tr><td>CP2_irenable_0</td><td>O</td><td>Enable Instruction Registering. When deasserted, no instruction strobes will be asserted in the following cycle. When asserted, there may be an instruction strobe asserted in the following cycle. Instruction strobes include CP2_as_0, CP2_ts_0, CP2_fs_0.
Note: This is the only late signal in the interface. The intended function is to use this signal as a clock gate condition on the capture latches in the coprocessor for CP2_ir_0[31:0].</td></tr>
<tr><td>CP2_as_0</td><td>O</td><td>Coprocessor2 Arithmetic Instruction Strobe. Asserted in the cycle after an arithmetic coprocessor2 instruction is available on CP2_ir_0[31:0]. If CP2_abusy_0 was asserted in the previous cycle, this signal will not be asserted. This signal will never be asserted in the same cycle that CP2_ts_0 or CP2_fs_0 is asserted.</td></tr>
</table>

Table 11 4KEm Signal Descriptions (Continued)

Signal Name	Type	Description
CP2_abusy_0	I	Coprocessor2 Arithmetic Busy. When asserted, a coprocessor2 arithmetic instruction will not be dispatched. *CP2_as_0* will not be asserted in the cycle after this signal is asserted.
CP2_ts_0	O	Coprocessor2 To Strobe. Asserted in the cycle after a To COP2 Op instruction is available on *CP2_ir_0[31:0]*. If *CP2_tbusy* was asserted in the previous cycle, this signal will not be asserted. This signal will never be asserted in the same cycle that *CP2_as_0* or *CP2_fs_0* is asserted.
CP2_tbusy_0	I	To Coprocessor2 Busy. When asserted, a To COP2 Op will not be dispatched. *CP2_ts_0* will not be asserted in the cycle after this signal is asserted.
CP2_fs_0	O	Coprocessor2 From Strobe. Asserted in the cycle after a From COP2 Op instruction is available on *CP2_ir_0[31:0]*. If *CP2_fbusy_0* was asserted in the previous cycle, this signal will not be asserted. This signal will never be asserted in the same cycle that *CP2_as_0* or *CP2_ts_0* is asserted.
CP2_fbusy_0	I	From Coprocessor2 Busy. When asserted, a From COP2 Op will not be dispatched. *CP2_fs_0* will not be asserted in the cycle after this signal is asserted.
CP2_endian_0	O	Big Endian Byte Ordering. When asserted, the processor is using big endian byte ordering for the dispatched instruction. When deasserted, the processor is using little-endian byte ordering. Valid the cycle before *CP2_as_0*, *CP2_fs_0* or *CP2_ts_0* is asserted.
CP2_inst32_0	O	MIPS32 Compatibility Mode - Instructions. When asserted, the dispatched instruction is restricted to the MIPS32 subset of instructions. Please refer to the MIPS64 architecture specification for a complete description of MIPS32 compatibility mode. Valid the cycle before *CP2_as_0*, *CP2_fs_0* or *CP2_ts_0* is asserted. **Note**: The 4KEm core is a MIPS32 core, and will only issue MIPS32 instructions. Thus *CP2_inst32_0* is tied high.
CP2_kd_mode_0	O	Kernel/Debug Mode. When asserted, the processor is running in kernel or debug mode. Can be used to enable “privileged” coprocessor instructions. Valid the cycle before *CP2_as_0, CP2_fs_0* or *CP2_ts_0* is asserted.
To Coprocessor Data: These signals are used when data is sent from the 4KEm core to the COP2 coprocessor, as part of completing a To Coprocessor instruction.		
CP2_tds_0	O	Coprocessor To Data Strobe. Asserted when To COP Op data is available on *CP2_tdata_0[31:0]*.

Table 11 4KEm Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td>CP2_torder_0[2:0]</td><td>O</td><td>Coprocessor To Order. Specifies which outstanding To COP Op the data is for. Valid only when CP2_tds_0 is asserted.

<table>
<tr><th>CP2_torder_0[2:0]</th><th>Order</th></tr>
<tr><td>000_2</td><td>Oldest outstanding To COP Op data transfer</td></tr>
<tr><td>001_2</td><td>2nd oldest To COP Op data transfer.</td></tr>
<tr><td>010_2</td><td>3rd oldest To COP Op data transfer.</td></tr>
<tr><td>011_2</td><td>4th oldest To COP Op data transfer.</td></tr>
<tr><td>100_2</td><td>5th oldest To COP Op data transfer.</td></tr>
<tr><td>101_2</td><td>6th oldest To COP Op data transfer.</td></tr>
<tr><td>110_2</td><td>7th oldest To COP Op data transfer.</td></tr>
<tr><td>111_2</td><td>8th oldest To COP Op data transfer.</td></tr>
</table>

Note: The 4KEm core will never send Data Out-of-Order, thus CP2_torder_0[2:0] is tied to 000_2.</td></tr>
<tr><td>CP2_tordlim_0[2:0]</td><td>S</td><td>To Coprocessor Data Out-of-Order Limit. This signal forces the integer processor core to limit how much it can reorder To COP Data. The value on this signal corresponds to the maximum allowed value to be used on CP2_torder_0[2:0].
Note: The 4KEm core will never send Data Out-of-Order, thus CP2_tordlim_0[2:0] is ignored.</td></tr>
<tr><td>CP2_tdata_0[31:0]</td><td>O</td><td>To Coprocessor Data. Data to be transferred to the coprocessor. Valid when CP2_tds_0 is asserted.</td></tr>
<tr><td colspan="3">From Coprocessor Data: These signals are used when data is sent to the 4KEm core from the COP2 coprocessor, as part of completing a From Coprocessor instruction.</td></tr>
<tr><td>CP2_fds_0</td><td>I</td><td>Coprocessor From Data Strobe. Asserted when From COP Op data is available on CP2_fdata_0[31:0].</td></tr>
<tr><td>CP2_forder_0[2:0]</td><td>I</td><td>Coprocessor From Order. Specifies which outstanding From COP Op the data is for. Valid only when CP2_fds_0 is asserted.

<table>
<tr><th>CP2_forder_0[2:0]</th><th>Order</th></tr>
<tr><td>000_2</td><td>Oldest outstanding From COP Op data transfer</td></tr>
<tr><td>001_2</td><td>2nd oldest From COP Op data transfer.</td></tr>
<tr><td>010_2</td><td>3rd oldest From COP Op data transfer.</td></tr>
<tr><td>011_2</td><td>4th oldest From COP Op data transfer.</td></tr>
<tr><td>100_2</td><td>5th oldest From COP Op data transfer.</td></tr>
<tr><td>101_2</td><td>6th oldest From COP Op data transfer.</td></tr>
<tr><td>110_2</td><td>7th oldest From COP Op data transfer.</td></tr>
<tr><td>111_2</td><td>8th oldest From COP Op data transfer.</td></tr>
</table>

Note: Only values 000_2 and 001_2 are allowed see CP2_fordlim_0[2:0] below</td></tr>
</table>

Table 11 4KEm Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td>CP2_fordlim_0[2:0]</td><td>O</td><td>From Coprocessor Data Out-of-Order Limit. This signal sets the limit on how much the coprocessor can reorder From COP Data. The value on this signal corresponds to the maximum allowed value to be used on CP2_forder_0[2:0].
Note: The 4KEm core can handle one Out-of-Order From Data transfer. CP2_fordlim_0[2:0] is therefore tied to 001_2. The core will also never have more than two outstanding From COP instructions issued, which also automatically limits CP2_forder_0[2:0] to 001_2.</td></tr>
<tr><td>CP2_fdata_0[31:0]</td><td>I</td><td>From Coprocessor Data. Data to be transferred from coprocessor. Valid when CP2_fds_0 is asserted.</td></tr>
<tr><td colspan="3">Coprocessor Condition Code Check: These signals are used to report the result of a condition code check to the 4KEm core from the COP2 coprocessor. This is only used for BC2 instructions.</td></tr>
<tr><td>CP2_cccs_0</td><td>I</td><td>Coprocessor Condition Code Check Strobe. Asserted when coprocessor condition code check bits are available on CP2_ccc_0.</td></tr>
<tr><td>CP2_ccc_0</td><td>I</td><td>Coprocessor Conditions Code Check. Valid when CP2_cccs_0 is asserted. When asserted, the branch instruction checking the condition code should take the branch. When deasserted, the branch instruction should not branch.</td></tr>
<tr><td colspan="3">Coprocessor Exceptions: These signals are used by the COP2 coprocessor to report exception for each instruction.</td></tr>
<tr><td>CP2_excs_0</td><td>I</td><td>Coprocessor Exception Strobe. Asserted when coprocessor exception signalling is available on CP2_exc_0 and CP2_exccode_0.</td></tr>
<tr><td>CP2_exc_0</td><td>I</td><td>Coprocessor Exception. When asserted, a Coprocessor exception is signaled on CP2_exccode_0[4:0]. Valid when CP2_excs_0 is asserted.</td></tr>
<tr><td>CP2_exccode_0[4:0]</td><td>I</td><td>Coprocessor Exception Code. Valid when both CP2_excs_0 and CP2_exc_0 are asserted.
<table>
<tr><th>CP2_exccode[4:0]</th><th>Exception</th></tr>
<tr><td>01010_2</td><td>(RI) Reserved Instruction Exception</td></tr>
<tr><td>10000_2</td><td>(IS1) Available for Coprocessor specific Exception</td></tr>
<tr><td>10001_2</td><td>(IS1) Available for Coprocessor specific Exception</td></tr>
<tr><td>10010_2</td><td>C2E Exception</td></tr>
<tr><td>All others</td><td>Reserved</td></tr>
</table></td></tr>
<tr><td colspan="3">Instruction Nullification: These signals are used by the 4KEm core to signal nullification of each instruction to the COP2 coprocessor.</td></tr>
<tr><td>CP2_nulls_0</td><td>O</td><td>Coprocessor Null Strobe. Asserted when a nullification signal is available on CP2_null_0.</td></tr>
<tr><td>CP2_null_0</td><td>O</td><td>Nullify Coprocessor Instruction. When deasserted, the 4KEm core is signalling that the instruction is not nullified. When asserted, the 4KEm core is signalling that the instruction is nullified, and no further transactions will take place for this instruction. Valid when CP2_nulls_0 is asserted.</td></tr>
<tr><td colspan="3">Instruction Killing: These signals are used by the 4KEm core to signal killing of each instruction to the COP2 coprocessor.</td></tr>
<tr><td>CP2_kills_0</td><td>O</td><td>Coprocessor Kill Strobe. Asserted when kill signalling is available on CP2_kill_0[1:0].</td></tr>
</table>

Table 11 4KEm Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td>CP2_kill_0[1:0]</td><td>O</td><td>Kill Coprocessor Instruction. Valid when CP2_kills_0 is asserted.

<table><tr><th>CP2_kill_0[1:0]</th><th>Type of Kill</th></tr><tr><td>00_2</td><td rowspan="2">Instruction is not killed and results can be committed.</td></tr><tr><td>01_2</td></tr><tr><td>10_2</td><td>Instruction is killed.
(not due to CP2_exc_0)</td></tr><tr><td>11_2</td><td>Instruction is killed.
(due to CP2_exc_0)</td></tr></table>
If an instruction is killed, no further transactions will take place on the interface for this instruction.</td></tr>
<tr><td colspan="3">Miscellaneous COP2 signals:</td></tr>
<tr><td>CP2_reset</td><td>O</td><td>Coprocessor Reset. Asserted when a hard or soft reset is performed by the integer unit.</td></tr>
<tr><td>CP2_present</td><td>S</td><td>COP2 Present. Must be asserted when COP2 hardware is connected to the Coprocessor 2 Interface.</td></tr>
<tr><td>CP2_idle</td><td>I</td><td>Coprocessor Idle. Asserted when the coprocessor logic is idle. Enables the processor to go into sleep mode and shut down the clock. Valid only if CP2_present is asserted.</td></tr>
<tr><td colspan="3">EJTAG Interface</td></tr>
<tr><td colspan="3">TAP interface. These signals comprise the EJTAG Test Access Port. These signals will not be connected if the core does not implement the TAP controller.</td></tr>
<tr><td>EJ_TRST_N</td><td>I</td><td>Active-low Test Reset Input (TRST*) for the EJTAG TAP. At power-up, the assertion of EJ_TRST_N causes the TAP controller to be reset.</td></tr>
<tr><td>EJ_TCK</td><td>I</td><td>Test Clock Input (TCK) for the EJTAG TAP.</td></tr>
<tr><td>EJ_TMS</td><td>I</td><td>Test Mode Select Input (TMS) for the EJTAG TAP.</td></tr>
<tr><td>EJ_TDI</td><td>I</td><td>Test Data Input (TDI) for the EJTAG TAP.</td></tr>
<tr><td>EJ_TDO</td><td>O</td><td>Test Data Output (TDO) for the EJTAG TAP.</td></tr>
<tr><td>EJ_TDOzstate</td><td>O</td><td>Drive indication for the output of TDO for the EJTAG TAP at chip level:
1: The TDO output at chip level must be in Z-state
0: The TDO output at chip level must be driven to the value of EJ_TDO
IEEE Standard 1149.1-1990 defines TDO as a 3-stated signal. To avoid having a 3-state core output, the 4KEm core outputs this signal to drive an external 3-state buffer.</td></tr>
<tr><td colspan="3">Debug Interrupt:</td></tr>
<tr><td>EJ_DINTsup</td><td>S</td><td>Value of DINTsup for the Implementation register. When high, this signal indicates that the EJTAG probe can use the DINT signal to interrupt the processor.</td></tr>
<tr><td>EJ_DINT</td><td>I</td><td>Debug exception request when this signal is asserted in a CPU clock period after being deasserted in the previous CPU clock period. The request is cleared when debug mode is entered. Requests when in debug mode are ignored.</td></tr>
</table>

Table 11 4KEm Signal Descriptions (Continued)

Signal Name	Type	Description
Debug Mode Indication:		
EJ_DebugM	O	Asserted when the core is in Debug Mode. This can be used to bring the core out of a low power mode. In systems with multiple processor cores, this signal can be used to synchronize the cores when debugging.
Device ID bits:		
These inputs provide an identifying number visible to the EJTAG probe. If the EJTAG TAP controller is not implemented, these inputs are not connected. These inputs are always available for soft core customers. On hard cores, the core “hardener” can set these inputs to their own values.		
EJ_ManufID[10:0]	S	Value of the ManufID[10:0] field in the Device ID register. As per IEEE 1149.1-1990 section 11.2, the manufacturer identity code shall be a compressed form of JEDEC standard manufacturer’s identification code in the JEDEC Publications 106, which can be found at: `http://www.jedec.org/` ManufID[6:0] bits are derived from the last byte of the JEDEC code by discarding the parity bit. ManufID[10:7] bits provide a binary count of the number of bytes in the JEDEC code that contain the continuation character (0x7F). Where the number of continuations characters exceeds 15, these 4 bits contain the modulo-16 count of the number of continuation characters.
EJ_PartNumber[15:0]	S	Value of the PartNumber[15:0] field in the Device ID register.
EJ_Version[3:0]	S	Value of the Version[3:0] field in the Device ID register.
System Implementation Dependent Outputs:		
These signals come from EJTAG control registers. They have no effect on the core, but can be used to give EJTAG debugging software additional control over the system.		
EJ_SRstE	O	Soft Reset Enable. EJTAG can deassert this signal if it wants to mask soft resets. If this signal is deasserted, none, some, or all soft reset sources are masked.
EJ_PerRst	O	Peripheral Reset. EJTAG can assert this signal to request the reset of some or all of the peripheral devices in the system.
EJ_PrRst	O	Processor Reset. EJTAG can assert this signal to request that the core be reset. This can be fed into the *SI_Reset* signal.
EJTAG Trace Interface		
These signals enable an interface to optional off-chip trace memory. The EJTAG Trace interface connects to the Probe Interface Block (PIB) which in turn connects to the physical off-chip trace pins. Note that if on-chip trace memory is used, access occurs via the EJTAG TAP interface, and this interface is not required.		

Table 11 4KEm Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td>TC_ClockRatio[2:0]</td><td>O</td><td>Clock ratio. This is the clock ratio set by software in TCBCONTROLB.CR. The value will be within the boundaries defined by TC_CRMax and TC_CRMin. The table below shows the encoded values for clock ratio.
<table>
<tr><th>TC_ClockRatio</th><th>Clock Ratio</th></tr>
<tr><td>000</td><td>8:1 (Trace clock is eight times the core clock)</td></tr>
<tr><td>001</td><td>4:1 (Trace clock is four times the core clock)</td></tr>
<tr><td>010</td><td>2:1 (Trace clock is double the core clock)</td></tr>
<tr><td>011</td><td>1:1 (Trace clock is same as the core clock)</td></tr>
<tr><td>100</td><td>1:2 (Trace clock is one half the core clock)</td></tr>
<tr><td>101</td><td>1:4 (Trace clock is one fourth the core clock)</td></tr>
<tr><td>110</td><td>1:6 (Trace clock is one sixth the core clock)</td></tr>
<tr><td>111</td><td>1:8 (Trace clock is one eight the core clock)</td></tr>
</table></td></tr>
<tr><td>TC_CRMax[2:0]</td><td>S</td><td>Maximum clock ratio supported. This static input sets the CRMax field of the TCBCONFIG register. It defines the capabilities of the Probe Interface Block (PIB) module.This field determines the minimum value of TC_ClockRatio.</td></tr>
<tr><td>TC_CRMin[2:0]</td><td>S</td><td>Minimum clock ratio supported. This input sets the CRMin field of the TCBCONFIG register. It defines the capabilities of the PIB module. This field determines the maximum value of TC_ClockRatio.</td></tr>
<tr><td>TC_ProbeWidth[1:0]</td><td>S</td><td>This static input will set the PW field of the TCBCONFIG register.
If this interface is not driving a PIB module, but some chip-level TCB-like module, then this field should be set to 2’b11 (reserved value for PW).
<table>
<tr><th>TC_ProbeWidth</th><th>Number physical data pin on PIB</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>Not directly to PIB</td></tr>
</table></td></tr>
<tr><td>TC_PibPresent</td><td>S</td><td>Must be asserted when a PIB is attached to the TC Interface. When de-asserted (low) all the other inputs are disregarded.</td></tr>
<tr><td>TC_TrEnable</td><td>O</td><td>Trace Enable, when asserted the PIB must start running its output clock and can expect valid data on all other outputs.</td></tr>
<tr><td>TC_Calibrate</td><td>O</td><td>This signal is asserted when the Cal bit in the TCBCONTROLB register is set.
For a simple PIB which only serves one TCB, this pin can be ignored. For a multi-core capable PIB which also uses TC_Valid and TC_Stall, the PIB must start producing the calibration pattern when this signal is asserted.</td></tr>
</table>

Table 11 4KEm Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td>TC_DataBits[2:0]</td><td>I</td><td>This input identifies the number of bits picked up by the probe interface module in each “cycle”.
If TC_ClockRatio indicates a clock-ratio higher than 1:2, then clock multiplication in the Probe logic is used. The “cycle” is equal to each core clock cycle.
If TC_ClockRatio indicates a clock-ratio lower than or equal to 1:2, then “cycle” is (clock-ratio * 2) of the core clock cycle. For example, with a clock ratio of 1:2, a “cycle” is equal to core clock cycle; with a clock ratio of 1:4, a “cycle” is equal to one half of core clock cycle.
This input controls the down-shifting amount and frequency of the trace word on TC_Data[63:0]. The bit width and the corresponding TC_DataBits value is shown in the table below.
<table>
<tr><th>TC_DataBits[2:0]</th><th>Probe uses following bits from TC_Data each cycle</th></tr>
<tr><td>000</td><td>TC_Data[3:0]</td></tr>
<tr><td>001</td><td>TC_Data[7:0]</td></tr>
<tr><td>010</td><td>TC_Data[15:0]</td></tr>
<tr><td>011</td><td>TC_Data[31:0]</td></tr>
<tr><td>100</td><td>TC_Data[63:0]</td></tr>
<tr><td>Others</td><td>Unused</td></tr>
</table>
This input might change as the value on TC_ClockRatio[2:0] changes.</td></tr>
<tr><td>TC_Valid</td><td>O</td><td>Asserted when a valid new trace word is started on the TC_Data[63:0] signals.
TC_Valid is only asserted when TC_DataBits is 100.</td></tr>
<tr><td>TC_Stall</td><td>I</td><td>When asserted, a new TC_Valid in the following cycle is stalled. TC_Valid is still asserted, but the TC_Data value and TC_Valid are held static, until the cycle after TC_Stall is sampled low.
TC_Stall is only sampled in the cycle before a new TC_Valid cycle, and only when TC_DataBits is 100, indicating a full word of TC_Data.</td></tr>
<tr><td>TC_Data[63:0]</td><td>O</td><td>Trace word data. The value on this 64-bit interface is shifted down as indicated in TC_DataBits[2:0]. In the first cycle where a new trace word is valid on all the bits and TC_DataBits[2:0] is 100, TC_Valid is also asserted.
The Probe Interface Block (PIB) will only be connected to [(N-1):0] bits of this output bus. N is the number of bits picked up by the PIB in each core clock cycle. For clock ratios 1:2 and lower, N is equal to the number of physical trace pins (legal values of N are 4, 8, or 16). For higher clock ratios, N is larger than the number of physical trace pins.</td></tr>
<tr><td>TC_ProbeTrigIn</td><td>A</td><td>Rising edge trigger input. The source should be the Probe Trigger input. The input is considered asynchronous; i.e., it is double registered in the core.</td></tr>
<tr><td>TC_ProbeTrigOut</td><td>O</td><td>Single cycle (relative to the “cycle” defined the description of TC_DataBits) high strobe, trigger output. The target of this trigger is intended to be the external probe’s trigger output.</td></tr>
<tr><td>TC_ChipTrigIn</td><td>A</td><td>Rising edge trigger input. The source should be on-chip. The input is considered asynchronous; i.e., it is double registered in the core.</td></tr>
<tr><td>TC_ChipTrigOut</td><td>O</td><td>Single cycle (relative to core clock) high strobe, trigger output. The target of this trigger is intended to be an on-chip unit.</td></tr>
<tr><td colspan="3">Performance Monitoring Interface</td></tr>
</table>

Table 11 4KEm Signal Descriptions (Continued)

Signal Name	Type	Description
These signals can be used to implement performance counters, which can be used to monitor hardware/software performance.		
PM_DCacheHit	O	This signal is asserted whenever there is a data cache hit.
PM_DCacheMiss	O	This signal is asserted whenever there is a data-cache miss.
PM_DTLBHit	O	This signal is not used in the 4KEm processor core and is tied to ground.
PM_DTLBMiss	O	This signal is not used in the 4KEm processor core and is tied to ground.
PM_ICacheHit	O	This signal is asserted whenever there is an instruction-cache hit.
PM_ICacheMiss	O	This signal is asserted whenever there is an instruction-cache miss.
PM_InstComplete	O	This signal is asserted each time an instruction completes in the pipeline.
PM_ITLBHit	O	This signal is not used in the 4KEm processor core and is tied to ground.
PM_ITLBMiss	O	This signal is not used in the 4KEm processor core and is tied to ground.
PM_JTLBHit	O	This signal is not used in the 4KEm processor core and is tied to ground.
PM_JTLBMiss	O	This signal is not used in the 4KEm processor core and is tied to ground.
PM_WTBMerge	O	This signal is asserted whenever there is a successful merge in the write-through buffer.
PM_WTBNoMerge	O	This signal is asserted whenever a non-merging store is written to the write-through buffer.
ScratchPad RAM interface		
This interface allows a ScratchPad RAM (SPRAM) array to be connected in parallel with the cache arrays, enabling fast access to data. There are independent interfaces for Instruction and Data ScratchPads. Signals related to the Instruction Scratchpad interface are prefixed with “ISP_”. Signals related to the Data Scratchpad interface are prefixed with “DSP_”. Note: In order to achieve single cycle access, the ScratchPad interface is not registered, unlike the other core interfaces. This requires more careful timing considerations.		
DSP_TagAddr[19:4]	O	Virtual index into the SPRAM used for tag reads and writes.
DSP_TagRdStr	O	Tag Read Strobe - Hit, Stall, TagRdValue use this strobe.
DSP_TagWrStr	O	Tag Write Strobe - If SPRAM tag is software configurable, this signal will indicate when to update the tag value.
DSP_TagCmpValue[23:0]	O	Tag Compare Value - This bus is used for both tag comparison and tag write value. For tag comparison, the bus usage is {PA[31:10], 2’b0} and contains the address to determine hit/miss. For tag writes, the bus contains {PA[31:10], Lock, Valid} from the *TagLo* register.
DSP_DataAddr[19:2]	O	Virtual index into the SPRAM used for data reads and writes.
DSP_DataWrValue[31:0]	O	Data Write Value - Data value to be written to the data array.
DSP_DataRdStr	O	Data Read Strobe - Indicates that the data array should be read.
DSP_DataWrStr	O	Data Write Strobe - Indicates that the data array should be written.
DSP_DataWrMask[3:0]	O	Data Write Mask - Byte enables for a data write.
DSP_DataRdValue[31:0]	I	Data Read Value - Data value read from the data array.
DSP_TagRdValue[23:0]	I	Tag Read Value - Tag value read from the tag array. Written to *TagLo* register on a CACHE instruction. Read value maps into these *TagLo* fields: {PA[31:10], Lock, Valid}

Table 11 4KEm Signal Descriptions (Continued)

Signal Name	Type	Description
DSP_Hit	I	Hit - Indicates that this read was to an address covered by the SPRAM.
DSP_Stall	I	Stall - Indicates that the read has not yet completed.
DSP_Present	S	Present - Indicates that a SPRAM array is connected to this port.
ISP_Addr[19:2]	O	Virtual index into the SPRAM used for both reads and writes of tag and data.
ISP_RdStr	O	Read Strobe - indicates a read of the tag and data arrays. Hit and Stall signals are also based off of this strobe.
ISP_TagWrStr	O	Tag Write Strobe - If SPRAM tag is software configurable, this signal will indicate when to update the tag value.
ISP_DataTagValue[31:0]	O	Write/Compare Data For data writes, this is the value to be written to the data array. For tag writes the bus contains the {8'b0, PA[31:10], Lock, Valid} from the TagLo register. For tag comparison, the bus has the address to be used for hit/miss determination in the format {8'b0, PA[31:10], Uncacheable, 1'b0}. When high, the Uncacheable bit indicates that the physical address bits (PA[31:10]) are to an uncacheable address; when the Uncacheable bit is low, the physical address is to a cacheable address.
ISP_DataWrStr	O	Data Write Strobe - Indicates that the data array should be written.
ISP_DataRdValue[31:0]	I	Data Read Value - Data value read from the data array.
ISP_TagRdValue[23:0]	I	Tag Read Value - Tag value read from the tag array. Written to *TagLo* register on a CACHE instruction. Read value maps into these *TagLo* fields: {PA[31:10], Lock, Valid}
ISP_Hit	I	Hit - Indicates that this read was to an address covered by the SPRAM.
ISP_Stall	I	Stall - Indicates that the read has not yet completed.
ISP_Present	S	Present - Indicates that a SPRAM array is connected to this port.
Integrated Memory BIST Interface		
These signals provide the interface to optional integrated memory BIST capability for testing the SRAM arrays within the core.		
gmbinvoke	I	Enable signal for integrated BIST controllers.
gmbdone	O	Common completion indicator for all integrated BIST sequences.
gmbddfail	O	When high, indicates that the integrated BIST test failed on the data cache data array.
gmbtdfail	O	When high, indicates that the integrated BIST test failed on the data cache tag array.
gmbwdfail	O	When high, indicates that the integrated BIST test failed on the data cache way select array.
gmbdifail	O	When high, indicates that the integrated BIST test failed on the instruction cache data array.
gmbtifail	O	When high, indicates that the integrated BIST test failed on the instruction cache tag array.
gmbwifail	O	When high, indicates that the integrated BIST test failed on the instruction cache way select array.
Scan Test Interface		
These signals provide an interface for testing the core. The use and configuration of these pins are implementation-dependent.		

Table 11 4KEm Signal Descriptions (Continued)

Signal Name	Type	Description
gscanenable	I	This signal should be asserted while scanning vectors into or out of the core. The *gscanenable* signal must be deasserted during normal operation and during capture clocks in test mode.
gscanmode	I	This signal should be asserted during all scan testing both while scanning and during capture clocks. The *gscanmode* signal must be deasserted during normal operation.
gscanramwr	I	This signal controls the read and write strobes to the cache SRAM when *gscanmode* is asserted.
gscanin_X	I	These signal(s) are the inputs to the scan chain(s).
gscanout_X	O	These signal(s) are the outputs from the scan chain(s).
BistIn[n:0]	I	Input to user-specified BIST controller.
BistOut[n:0]	O	Output from user-specified BIST controller.

EC Interface Transactions

The 4KEm core implements the EC™ interface for its bus transactions. This interface uses a pipelined, in-order protocol with independent address, read data, and write data buses. The following subsections describe the four basic bus transactions: single read, single write, burst read, and burst write.

Single Read

Figure 7 shows the basic timing relationships of signals during a simple read transaction. During a single read cycle, the 4KEm core drives the address onto *EB_A[35:2]* and byte enable information onto *EB_BE[3:0]*. To maximize performance, the EC interface does not define a maximum number of outstanding bus cycles. Instead it provides the *EB_ARdy* input signal. This signal is driven by external logic and controls the generation of addresses on the bus.

In the 4KEm core, the address is driven whenever it becomes available, regardless of the state of *EB_ARdy*. However, the 4KEm core always continues to drive the address until the clock after *EB_ARdy* is sampled asserted. For example, at the rising edge of the clock 2 in Figure 7, the *EB_ARdy* signal is sampled low, indicating that external logic is not ready to accept the new address. However, the 4KEm core still drives *EB_A[35:2]* in this clock as shown. On the rising edge of clock 3, the 4KEm core samples *EB_ARdy* asserted and continues to drive the address until the rising edge of clock 4.

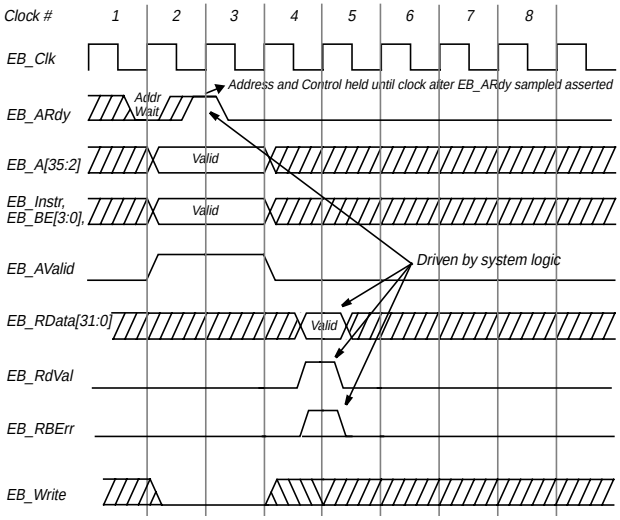


Figure 7 Single Read Transaction Timing Diagram

The *EB_Instr* signal is only asserted during a single read cycle if there is an instruction fetch from non-cacheable memory space. The *EB_AValid* signal is driven in each clock that *EB_A[35:2]* is valid on the bus. The 4KEm core drives *EB_Write* low to indicate a read transaction.

The *EB_RData[31:0]* and *EB_RdVal* signals are first sampled on the rising edge of clock 4, one clock after *EB_ARdy* is sampled asserted. Data is sampled on every clock thereafter until *EB_RdVal* is sampled asserted.

If a bus error occurs during the data transaction, external logic asserts *EB_RBErr* in the same clock as *EB_RdVal*.

Single Write

Figure 8 shows a typical write transaction. The 4KEm core drives address and control information onto the *EB_A[35:2]* and *EB_BE[3:0]* signals on the rising edge of clock 2. As in the single read cycle, these signals remain active until the clock edge after the *EB_ARdy* signal is sampled asserted. The 4KEm core asserts the *EB_Write* signal to indicate that a valid write cycle is on the bus and *EB_AValid* to indicate that valid address is on the bus.

The 4KEm core drives write data onto *EB_WData[31:0]* in the same clock as the address and continues to drive data until the clock edge after the *EB_WDRdy* signal is sampled asserted. If a bus error occurs during a write operation, external logic asserts the *EB_WBErr* signal one clock after asserting *EB_WDRdy.*

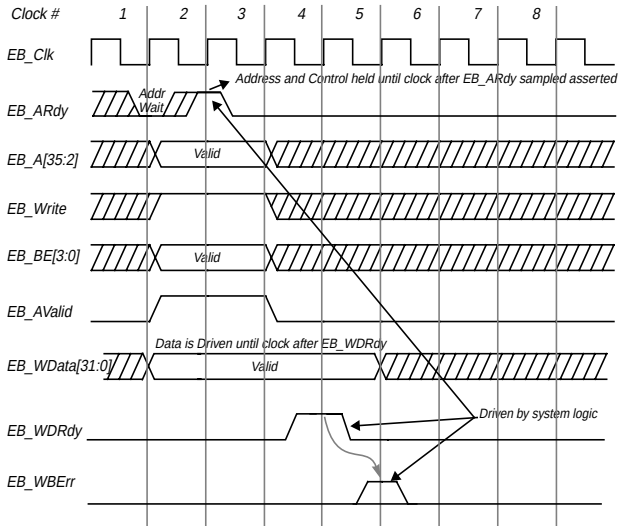


Figure 8 Single Write Transaction Timing Diagram

Burst Read

The 4KEm core is capable of generating burst transactions on the bus. A burst transaction is used to transfer multiple data items in one transaction.

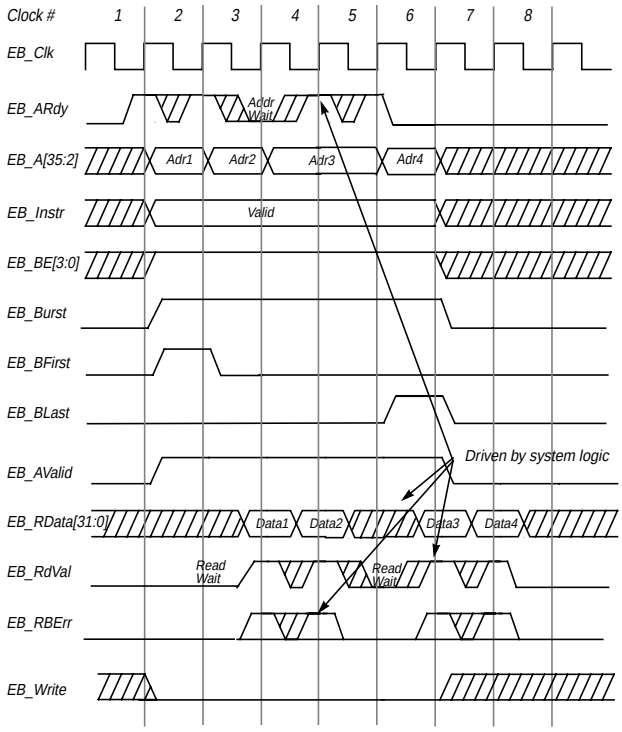


Figure 9 Burst Read Transaction Timing Diagram

Figure 9 shows an example of a burst read transaction. Burst read transactions initiated by the 4KEm core always contain four data transfers in a sequence determined by the critical word (the address that caused the miss) and *EB_SBlock*. In addition, the data requested is always a 16-byte aligned block.

The order of words within this 16-byte block varies depending on which of the words in the block is being requested by the execution unit and the ordering protocol selected. The burst always starts with the word requested by the execution unit and proceeds in either an ascending or descending address order, wrapping when the block boundary is reached. Table 12 and Table 13 show the sequence of address bits 2 and 3.

Table 12 Sequential Ordering Protocols

Starting Address EB_A[3:2]	Address Progression of EB_A[3:2]
00	00, 01, 10, 11
01	01, 10, 11, 00
10	10, 11, 00, 01
11	11, 00, 01, 10

Table 13 Sub-Block Ordering Protocols

Starting Address EB_A[3:2]	Address Progression of EB_A[3:2]
00	00, 01, 10, 11
01	01, 00, 11, 10
10	10, 11, 00, 01
11	11, 10, 01, 00

The 4KEm core drives address and control information onto the *EB_A[35:2]* and *EB_BE[3:0]* signals on the rising edge of clock 2. As in the single read cycle, these signals remain active until the clock edge after the *EB_ARdy* signal is sampled asserted. The 4KEm core continues to drive *EB_AValid* as long as a valid address is on the bus.

The *EB_Instr* signal is asserted if the burst read is for an instruction fetch. The *EB_Burst* signal is asserted while the address is on the bus to indicate that the current address is part of a burst transaction. The 4KEm core asserts the *EB_BFirst* signal in the same clock as the first address is driven and the *EB_BLast* signal in the same clock as the last address to indicate the start and end of a burst cycle.

The 4KEm core first samples the *EB_RData[31:0]* signals two clocks after *EB_ARDy* is sampled asserted. External logic asserts *EB_RdVal* to indicate that valid data is on the bus. The 4KEm core latches data internally whenever *EB_RdVal* is sampled asserted.

Note that on the rising edge of clocks 3 and 6 in Figure 9, the *EB_RdVal* signal is sampled deasserted, causing wait states in the data return. There is also an address wait state caused by *EB_ARdy* being sampled deasserted on the rising edge of clock 4. Note that the core holds address 3 on the *EB_A* bus for an extra clock because of this wait state. External logic asserts the *EB_RBErr* signal in the same clock as data if a bus error occurs during that data transfer.

Burst Write

Burst write transactions are used to empty one of the write buffers. A burst write transaction is only performed if the write buffer contains 16 bytes of data associated with the same aligned memory block, otherwise individual write transactions are performed. Figure 10 shows a timing diagram of a burst write transaction. Unlike the read burst, a write burst always begins with *EB_A[3:2]* equal to 00b.

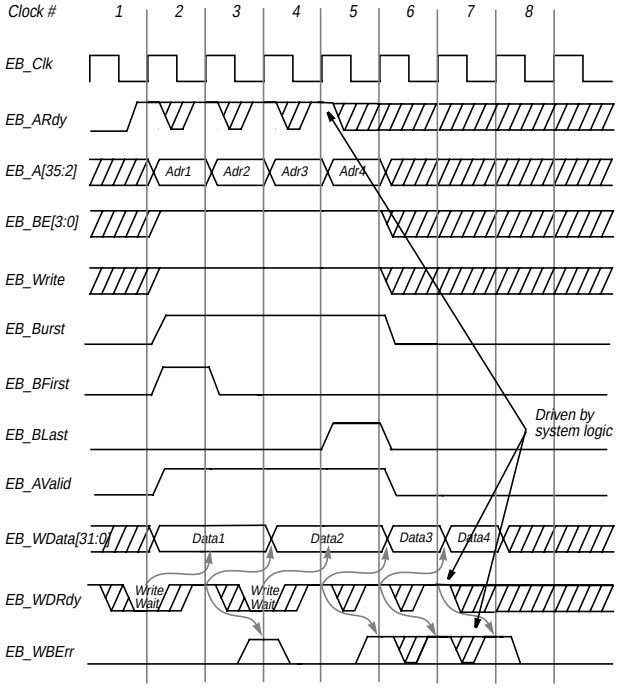


Figure 10 Burst Write Transaction Timing Diagram

The 4KEm core drives address and control information onto the *EB_A[35:2]* and *EB_BE[3:0]* signals on the rising edge of clock 2. As in the single read cycle, these signals remain active until the clock edge after the *EB_ARdy* signal is sampled asserted. The 4KEm core continues to drive *EB_AValid* as long as a valid address is on the bus.

The 4KEm core asserts the *EB_Write*, *EB_Burst*, and *EB_AValid* signals during the time the address is driven. *EB_Write* indicates that a write operation is in progress. The assertion of *EB_Burst* indicates that the current operation is a burst. *EB_AValid* indicates that valid address is on the bus.

The 4KEm core asserts the *EB_BFirst* signal in the same clock as address 1 is driven to indicate the start of a burst cycle. In the clock that the last address is driven, the 4KEm core asserts *EB_BLast* to indicate the end of the burst transaction.

In Figure 10, the first data word (Data1) is driven in clocks 2 and 3 due to the *EB_WDRdy* signal being sampled deasserted at the rising edge of clock 2, causing a wait state. When *EB_WDRdy* is sampled asserted on the rising edge of clock 3, the 4KEm core responds by driving the second word (Data2).

External logic drives the *EB_WBErr* signal one clock after the corresponding assertion of *EB_WDRdy* if a bus error has occurred as shown by the arrows in Figure 10.

Revision History

In the left hand page margins of this document you may find vertical change bars to note the location of significant changes to this document since its last release. Significant changes are defined as those which you should take note of as you use the MIPS IP. Changes to correct grammar, spelling errors or similar may or may not be noted with change bars. Change bars will be removed for changes which are more than one revision old.

Please note: Limitations on the authoring tools make it difficult to place change bars on changes to figures. Change bars on figure titles are used to denote a potential change in the figure itself. Certain parts of this document (Instruction set descriptions, EJTAG register definitions) are references to Architecture specifications, and the change bars within these sections indicate alterations since the previous version of the relevant Architecture document.

Revision	Date	Description
02.00	November 8, 2002	• Added this revision history table. • Various updates to describe new MIPS32 Release 2 capabilities, included in version 3.0 or higher core releases.

Template: D1.06, Build with Conditional Tags: 2B