The MIPS32™ 4Km™ core from MIPS® Technologies is a member of the MIPS32 4K™ 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 4Km 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 4Km core implements the MIPS32 Architecture and contains all MIPS II™ instructions; special multiply-accumulate (MAC), conditional move, prefetch, wait, and leading zero/one detect instructions; and the 32-bit privileged resource architecture. The Memory Management Unit consists of a simple, fixed Block Address Translation (BAT) mechanism for applications that do not require the full capabilities of a Translation Lookaside Buffer based MMU.

The synthesizable 4Km core implements single cycle MAC instructions, which enable DSP algorithms to be performed efficiently. The Multiply/Divide Unit (MDU) allows 32-bit x 16-bit MAC instructions to be issued every cycle. A 32-bit x 32-bit MAC instruction can be issued every 2 cycles.

Instruction and data caches are fully configurable from 0 - 16 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 breakpoints.

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

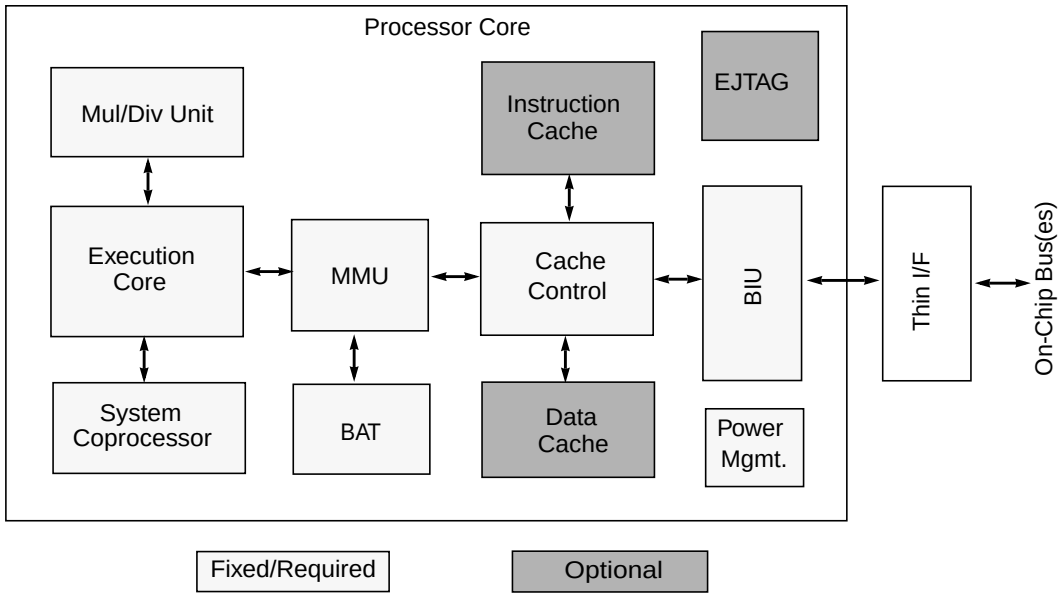


Figure 1 4Km Core Block Diagram

Features

- 32-bit Address and Data Paths
- MIPS32-Compatible Instruction Set
  - All MIPS II Instructions
  - 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)
- Programmable Cache Sizes
  - Individually configurable instruction and data caches
  - Sizes from 0 - 16KB
  - Direct Mapped, 2-, 3-, or 4-Way Set Associative
  - Loads block only until critical word is available
  - Write-through, no write-allocate
  - 16-byte cache line size, word sectored
  - 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
  - 20 index address bits allow access of arrays up to 1MB
  - Memory-mapped registers attached to the scratchpad port can be used as a coprocessor interface
- R4000®-style Privileged Resource Architecture
  - Count/Compare registers for real-time timer interrupts
  - I and D watch registers for SW breakpoints
  - Separate interrupt exception vector
- Memory Management Unit
  - Simple Block Address Translation (BAT) mechanism
- Simple Bus Interface Unit (BIU)
  - All I/Os fully registered
  - Separate unidirectional 32-bit address and data buses
  - Two 16-byte collapsing write buffers
- 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)
- Power Control
  - Minimum frequency: 0 MHz
  - Power-down mode (triggered by WAIT instruction)
  - Support for software-controlled clock divider
- EJTAG Debug Support with single stepping, virtual instruction and data address breakpoints

Architecture Overview

The 4Km 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 4Km 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)
- Block Address Translation (BAT)
- Cache Controllers
- Bus Interface Unit (BIU)
- Power Management

Optional blocks include:

- Instruction Cache
- Data Cache
- Scratchpad RAM
- Enhanced JTAG (EJTAG) Controller

The section entitled "4Km Core Required Logic Blocks" on page 3 discusses the required blocks. The section entitled "4Km Core Optional Logic Blocks" on page 10 discusses the optional blocks.

Pipeline Flow

The 4Km 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 4Km core pipeline consists of five stages:

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

- Writeback (W stage)

The 4Km 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 4Km core pipeline.

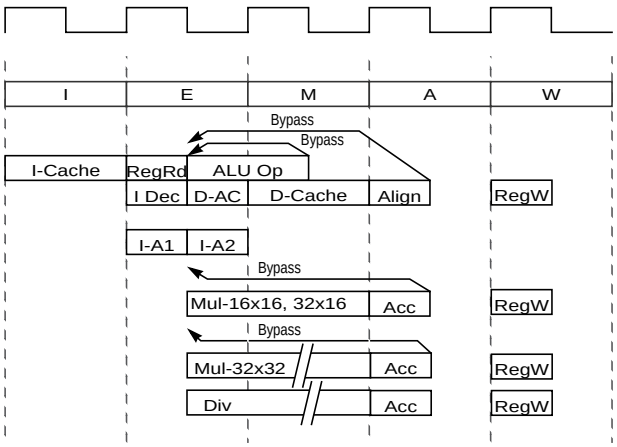


Figure 2 4Km Core Pipeline

I Stage: Instruction Fetch

During the Instruction fetch stage:

- An instruction is fetched from instruction cache.

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 fetch 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 for one clock in the M stage.
- A divide operation stalls 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:

- A separate aligner aligns load data 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

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

4Km Core Required Logic Blocks

The 4Km core consists of the following required logic blocks as 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)
- Block Address Translation (BAT)
- Cache Controller
- Bus Interface Control (BIU)
- Power Management

Execution Unit

The 4Km core execution unit implements a load/store architecture with single-cycle ALU operations (logical, shift, add, subtract) and an autonomous multiply/divide unit. The 4Km core contains thirty-two 32-bit general-purpose registers used for integer operations and address calculation. 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 4Km core contains 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 4Km 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 4Km 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 4K™ Processor Core Family Software User’s Manual*.

Table 1 4Km Core 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.

As an enhancement to the MIPS II ISA, the 4Km core implements an additional multiply instruction, MUL, which specifies that multiply results be placed 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 interrupts enabled or disabled. Configuration information such as cache size and set associativity is available by accessing the CP0 registers, listed in Table 2.

Table 2 Coprocessor 0 Registers in Numerical Order

Register Number	Register Name	Function
0	Index[1]	Reserved in the 4Km core.
1	Random[1]	Reserved in the 4Km4Km core.
2	EntryLo0[1]	Reserved in the 4Km core.
3	EntryLo1[1]	Reserved in the 4Km core.
4	Context[2]	Pointer to page table entry in memory.
5	PageMask[1]	Reserved in the 4Km core.
6	Wired[1]	Reserved in the 4Km core.
7	Reserved	Reserved.
8	BadVAddr[2]	Reports the address for the most recent address-related exception.
9	Count[2]	Processor cycle count.
10	EntryHi[1]	Reserved in the 4Km core.
11	Compare[2]	Timer interrupt control.
12	Status[2]	Processor status and control.
13	Cause[2]	Cause of last general exception.
14	EPC[2]	Program counter at last exception.
15	PRId	Processor identification and revision.
16	Config	Configuration register.
16	Config1	Configuration register 1.
17	LLAddr	Load linked address.
18	WatchLo[2]	Low-order watchpoint address.
19	WatchHi[2]	High-order watchpoint address.
20 - 22	Reserved	Reserved.
23	Debug[3]	Debug control and exception status.
24	DEPC[3]	Program counter at last debug exception.
25 - 27	Reserved	Reserved.
28	TagLo/ DataLo	Low-order portion of cache tag interface.
29	Reserved	Reserved.
30	ErrorEPC[2]	Program counter at last error.
31	DeSave[3]	Debug handler scratchpad register.

1. Registers used in memory management.
2. Registers used in exception processing.
3. 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 4Km Core Exception Types

Exception	Description
Reset	Assertion of *SI_ColdReset* signal.
Soft Reset	Assertion of *SI_Reset* signal.
DSS	EJTAG Debug Single Step.

Table 3 4Km Core Exception Types (Continued)

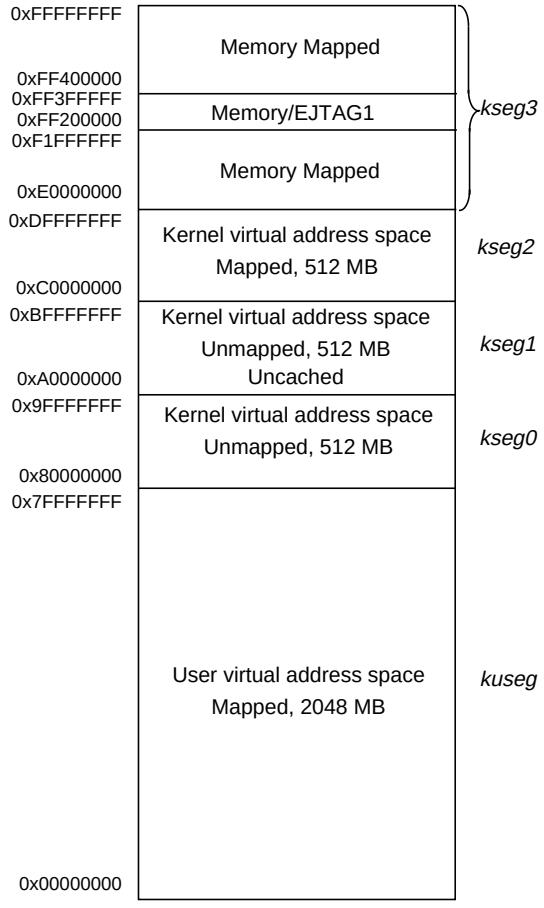
Exception	Description
DINT	EJTAG Debug Interrupt. Caused by the assertion of the external *EJ_DINT* input, or by setting the EjtagBrk bit in the ECR register.
NMI	Assertion of *EB_NMI* signal.
Machine Check	TLB write that conflicts with an existing entry.
Interrupt	Assertion of unmasked hardware or software interrupt signal.
Deferred Watch	Deferred Watch (unmasked by K|DM->!(K|DM) transition).
DIB	EJTAG debug hardware instruction break matched.
WATCH	A reference to an address in one of the watch registers (fetch).
AdEL	Fetch address alignment error. Fetch reference to protected address.
TLBL	Fetch TLB miss.
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.
AdES	Store address alignment error. Store to protected address.

Table 3 4Km Core Exception Types (Continued)

Exception	Description
DBE	Load or store bus error.
DDBL	EJTAG data hardware breakpoint matched in load data compare.

Modes of Operation

The 4Km 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 of kernel mode, and by the EJTAG module in debug mode.

Figure 3 4Km Core Virtual Address Map

Memory Management Unit (MMU)

The 4Km core contains an MMU that interfaces between the execution unit and the cache controller. The 4Km core provides a simple block address translation (BAT) mechanism that is smaller than the TLB in the MIPS32 4Kc™ core and more easily synthesized. Like the TLB, the BAT performs virtual-to-physical address translation and provides attributes for the different segments. Those segments that are unmapped in the 4Kc core's TLB implementation (kseg0 and kseg1) are translated identically by the BAT.

Figure 4 shows how the BAT is implemented in the 4Km core.

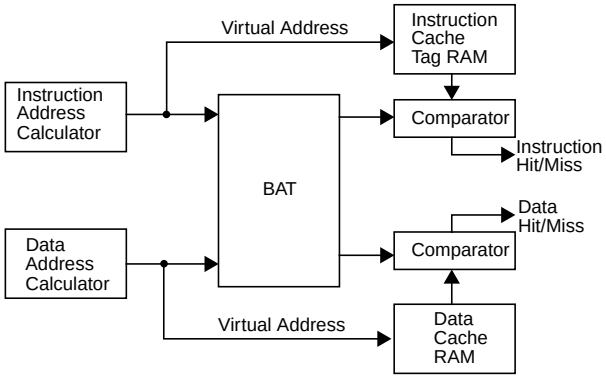


Figure 4 Address Translation During a Cache Access

The BAT 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) bits of the Config register.

Table 4 Cache Coherency Attributes

Config Register Fields K23, KU, and K0	Cache Coherency Attribute
0*, 1*, 3, 4*, 5*, 6*	Cacheable, noncoherent, write-through, no 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 4Km core, no translation exceptions can be taken, although address errors are still possible.

Table 5 Cacheability of Segments with Block Address Translation

Segment	Virtual Address Range	Cacheability
useg/kuseg	0x0000_0000-0x7FFF_FFFF	Controlled by the KU field (bits 27:25) of the Config register. 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 BAT performs a simple translation to map from virtual addresses to physical addresses. This mapping is shown in Figure 5.

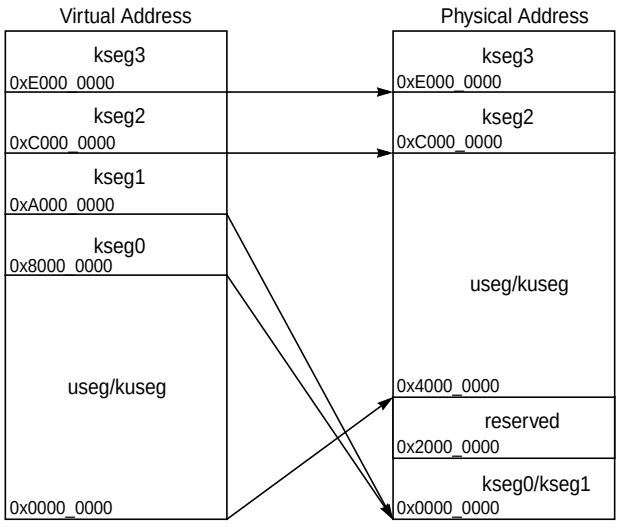


Figure 5 BAT Memory Map (ERL=0) in the 4Km Processor Core

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

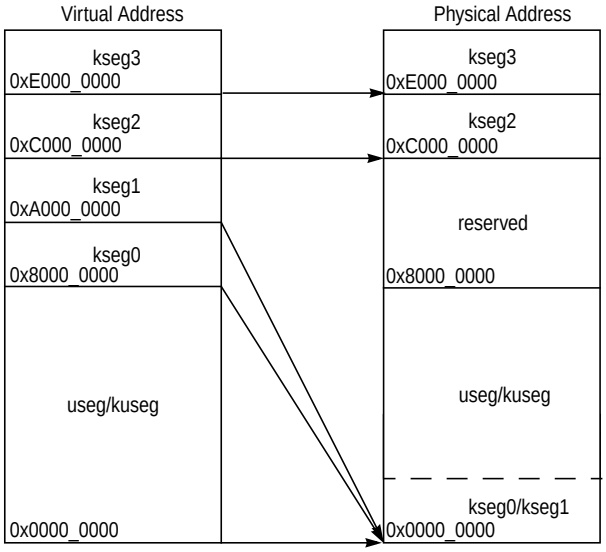


Figure 6 BAT Memory Map (ERL=1) in the 4Km Processor Core

Cache Controllers

The 4Km 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 "4Km Core Optional Logic Blocks" on page 10 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 "4Km Core Optional Logic Blocks" on page 10 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. Since the 4Km core caches follow a write-through cache policy, the write buffer significantly reduces the number of writes 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.

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

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 Pattern Control

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

The 4Km 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 4Km core to existing busses, there is a mode that only generates “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. Writes with illegal byte enable patterns will be broken into two separate write transactions. This splitting is independent of the merge pattern control in the write buffer. The only case where a read can generate illegal byte enables is on an uncached tri-byte load (LWL/LWR). These reads will be converted into a word read on the bus.

Table 6 Valid SimpleBE Byte Enable Patterns

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

4Km Core Reset

The 4Km core has two types of reset input signals: *Reset* and *ColdReset*.

The *ColdReset* signal must be asserted on either a power-on reset or a cold reset. In a typical application, a power-on reset occurs when the machine is first turned on. A cold reset (also called a hard reset) typically occurs when the machine is already on and the system is rebooted. A cold reset completely initializes the internal state machines of the 4Km core without saving any state information. The *Reset* and *ColdReset* signals work in conjunction with one another to determine the type of reset operation (see Table 7).

Table 7 4Km Reset Types

Reset	ColdReset	Action
0	0	Normal Operation, no reset.
1	0	Warm or Soft reset.

Table 7 4Km Reset Types

Reset	ColdReset	Action
X	1	Cold or Hard reset.

The *Reset* signal is asserted for a warm reset. A warm reset restarts the 4Km core and preserves more of the processors internal state than a cold reset. The *Reset* signal can be asserted synchronously or asynchronously during a cold reset, or synchronously to initiate a warm reset. The assertion of *Reset* causes a soft reset exception within the 4Km core. In debug mode, EJTAG can request that the soft reset function be masked. It is system dependent whether this functionality is supported. In normal mode, the soft reset cannot be masked.

Power Management

The 4Km core offers a number of power management features, including low-power design, active power management, and power-down modes of operation. The 4Km core is a static design that supports slowing or halting the clocks, which reduces system power consumption during idle periods.

The 4Km 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 low power mode, such as by 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 4Km 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 4Km 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 4Km core is waiting for an interrupt.

4Km Core Optional Logic Blocks

The 4Km core consists of the following optional logic blocks as shown in the block diagram in Figure 1.

Instruction Cache

The instruction cache is an optional on-chip memory block of up to 16 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, 4 valid bits, a lock bit, and the fill replacement bit.

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 4Km4Km 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 16 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, 4 valid bits, a lock bit, and the fill replacement bit.

In addition to instruction-cache locking, the 4Km 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 4Km 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 4Km core instruction and data cache attributes.

Table 8 4Km Core Instruction and Data Cache Attributes

Parameter	Instruction	Data
Size	0 - 16 Kbytes	0 - 16 Kbytes
Organization	1 - 4 way set associative	1 - 4 way set associative

Table 8 4Km Core Instruction and Data Cache Attributes

Parameter	Instruction	Data
Line Size	16 bytes	16 bytes
Read Unit	32 bits	32 bits
Write Policy	na	write-through *without write-allocate*
Miss restart after transfer of	miss word	miss word
Cache Locking	per line	per line

Cache Protocols

The 4Km 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:** 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.

Scratchpad RAM

The 4Km core also supports replacing up to one way of each cache with a scratchpad RAM. The scratchpad RAM is user-defined and can consist of a variety of devices. The main requirement is that it must be accessible with timing similar to a regular cache RAM. 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 thus easily contain a large RAM/ROM or memory-mapped registers.

The core's 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 write-throughs to main memory.

EJTAG Debug Support

The 4Km 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 4Km 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 "4Km Core Signal Descriptions" on page 16 for a list of signals 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 4Km 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 the setting up of 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 4Km core: Instruction breakpoints and Data breakpoints.

The 4Km 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 on the bus between the CPU and the instruction cache. Instruction breaks can also be made on the ASID value used by the MMU. Finally, 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 and ASID values, similar to the Instruction breakpoint. Data breakpoints can be set on a load, a store, or both. Data breakpoints can also be set based on the value of the load/store operation. Finally, masks can be applied to both the virtual address and the load/store value.

4Km Core Instructions

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

Table 9 4Km 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`
ADDU	Unsigned Integer Add	`Rd = Rs +U Rt`
AND	Logical AND	`Rd = Rs & Rt`
ANDI	Logical AND Immediate	`Rt = Rs & (0`$_{16}$` || Immed)`
BEQ	Branch On Equal	`if Rs == Rt` `  PC += (int)offset`

BEQL	Branch On Equal Likely	`if Rs == Rt`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

BGEZ	Branch on Greater Than or Equal To Zero	`if !Rs[31]` `  PC += (int)offset`

BGEZAL	Branch on Greater Than or Equal To Zero And Link	`GPR[31] = PC + 8`
		`if !Rs[31]`
		`  PC += (int)offset`

BGEZALL	Branch on Greater Than or Equal To Zero And Link Likely	`GPR[31] = PC + 8`
		`if !Rs[31]`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

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

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

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

Table 9 4Km Core Instruction Set (Continued)

Instruction	Description	Function
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
COP0	Coprocessor 0 Operation	See Software User's Manual
CLO	Count Leading Ones	`Rd = NumLeadingOnes(Rs)`
CLZ	Count Leading Zeroes	`Rd = NumLeadingZeroes(Rs)`
DERET	Return from Debug Exception	`PC = DEPC` `Exit Debug Mode`
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`

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

J	Unconditional Jump	`PC = PC[31:28] || offset<<2`

Table 9 4Km Core Instruction Set (Continued)

Instruction	Description	Function
JAL	Jump and Link	`GPR[31] = PC + 8` `PC = PC[31:28] || offset<<2`
JALR	Jump and Link Register	`Rd = PC + 8` `PC = Rs`
JR	Jump Register	`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]`

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]`
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, n, sel] = Rt`
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`
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)`

Table 9 4Km Core Instruction Set (Continued)

Instruction	Description	Function
OR	Logical OR	`Rd = Rs | Rt`
ORI	Logical OR Immediate	`Rt = Rs | Immed`
PREF	Prefetch	Load Specified Line into Cache
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
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]`
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`
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`

Table 9 4Km Core Instruction Set (Continued)

Instruction	Description	Function
TEQ	Trap if Equal	`if Rs == Rt TrapException`
TEQI	Trap if Equal Immediate	`if Rs == (int)Immed TrapException`
TGE	Trap if Greater Than or Equal	`if (int)Rs >= (int)Rt TrapException`
TGEI	Trap if Greater Than or Equal Immediate	`if (int)Rs >= (int)Immed TrapException`
TGEIU	Trap if Greater Than or Equal Immediate Unsigned	`if (uns)Rs >= (uns)Immed TrapException`
TGEU	Trap if Greater Than or Equal Unsigned	`if (uns)Rs >= (uns)Rt TrapException`
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
XOR	Exclusive OR	`Rd = Rs ^ Rt`
XORI	Exclusive OR Immediate	`Rt = Rs ^ (uns)Immed`

4Km Core Signal Descriptions

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

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

Table 10 4Km Core Signal Direction Key

Dir	Description
I	Input to the 4Km core sampled on the rising edge of the appropriate CLK signal.
O	Output of the 4Km 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 4Km core. These signals are normally tied to either power or ground and should not change state while *SI_ColdReset* is deasserted.

The 4Km 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 11 4Km 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 SoftReset Exception in the core.
Power Management Signals:		
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 the device can enter a reduced power mode.
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_Int[5:0]	A	Active-high Interrupt Pins. These signals are driven by external logic and, when asserted, indicate the corresponding interrupt exception to the core. These signals go through synchronization logic and can be asserted asynchronously to *SI_ClkIn*.
SI_TimerInt	O	This signal is asserted whenever the Count and Compare registers match and is deasserted when the Compare register is written. In order to have timer interrupts, this signal needs to be brought back into the 4K core on one of the 6 *SI_Int* interrupt pins. Traditionally, this has been accomplished via 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. In a complex system, it could even be fed into a priority encoder to allow *SI_Int[5:0]* to map up to 63 interrupt sources.

Table 11 4Km Signal Descriptions

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td colspan="3">Configuration Inputs:</td></tr>
<tr><td>SI_Endian</td><td>S</td><td>Indicates the base endianess 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>
<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>
</table>

Table 11 4Km Signal Descriptions

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></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>SI</td><td>When sampled asserted, sub-block ordering is used. When sampled deasserted, sequential addressing is used.</td></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">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>
</table>

Table 11 4Km Signal Descriptions

Signal Name	Type	Description
EJ_TRST_N	I	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.
EJ_TCK	I	Test Clock Input (TCK) for the EJTAG TAP.
EJ_TMS	I	Test Mode Select Input (TMS) for the EJTAG TAP.
EJ_TDI	I	Test Data Input (TDI) for the EJTAG TAP.
EJ_TDO	O	Test Data Output (TDO) for the EJTAG TAP.
EJ_TDOzstate	O	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 4K core outputs this signal to drive an external 3-state buffer.
Debug Interrupt:		
EJ_DINTsup	S	Value of DINTsup for the Implementation register. A 1 on this signal indicates that the EJTAG probe can use the DINT signal to interrupt the processor.
EJ_DINT	I	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.
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.

Table 11 4Km Signal Descriptions

Signal Name	Type	Description
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.
Performance Monitoring Interface		
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 4Km processor core and is tied to ground.
PM_DTLBMiss	O	This signal is not used in the 4Km 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 4Km processor core and is tied to ground.
PM_ITLBMiss	O	This signal is not used in the 4Km processor core and is tied to ground.
PM_JTLBHit	O	This signal is not used in the 4Km processor core and is tied to ground.
PM_JTLBMiss	O	This signal is not used in the 4Km 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.
Scan Test Interface		
These signals provide the interface for testing the core. The use and configuration of these pins are implementation-dependent.		
ScanEnable	I	This signal should be asserted while scanning vectors into or out of the core. The *ScanEnable* signal must be deasserted during normal operation and during capture clocks in test mode.
ScanMode	I	This signal should be asserted during all scan testing both while scanning and during capture clocks. The *ScanMode* signal must be deasserted during normal operation.
ScanIn<n:0>	I	This signal is input to the scan chain.
ScanOut<n:0>	O	This signal is output from the scan chain.
BistIn<n:0>	I	Input to the BIST controller.
BistOut<n:0>	O	Output from the BIST controller.

4Km Core Bus Transactions

The 4Km 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 4Km 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 4Km4Km core, the address is driven whenever it becomes available, regardless of the state of *EB_ARdy.* However, the 4Km4Km 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 4Km core still drives *EB_A[35:2]* in this clock as shown. On the rising edge of clock 3, the 4Km core samples *EB_ARdy* asserted and continues to drive the address until the rising edge of clock 4.

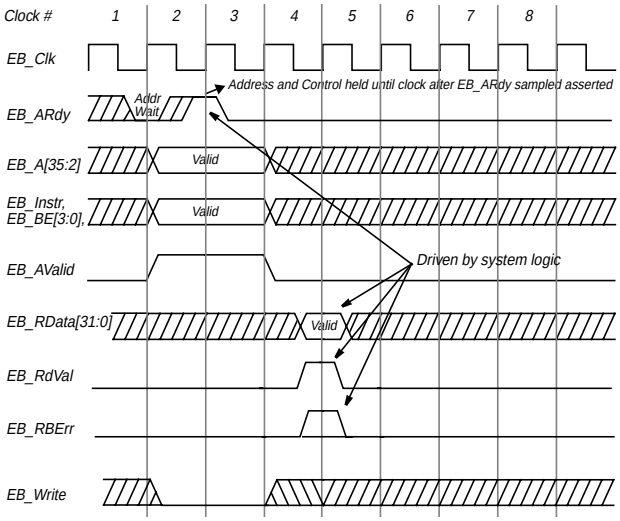


Figure 7 Single Read Transaction Timing Diagram

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 4Km 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 4Km 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 4Km 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 4Km 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.*

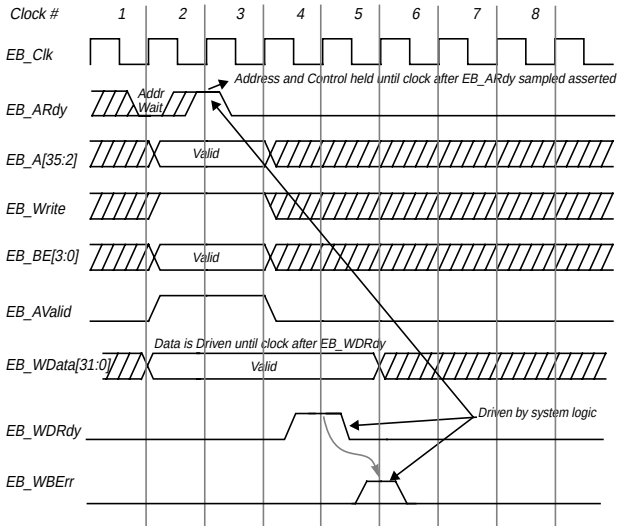


Figure 8 Single Write Transaction Timing Diagram

Burst Read

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

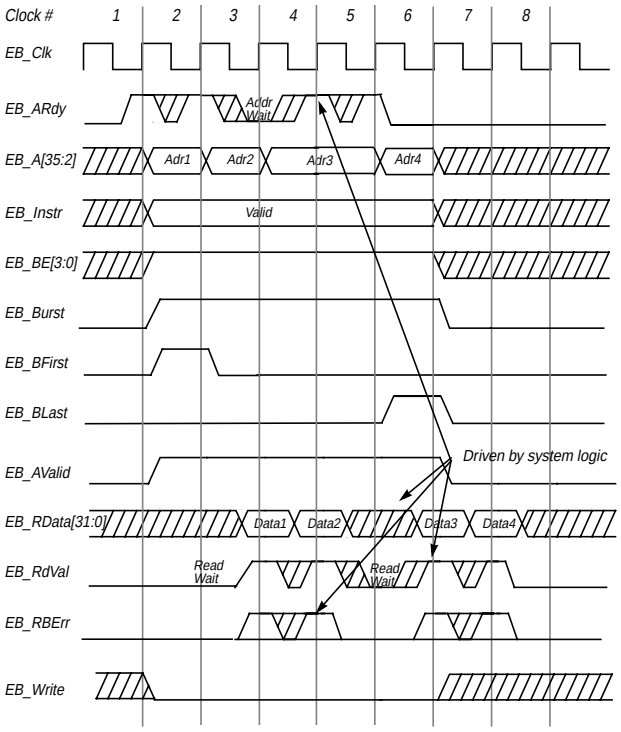


Figure 9 Burst Read Transaction Timing Diagram

Figure 9 shows an example of a burst read transaction. Burst read transactions initiated by the 4Km 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 4Km4Km 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 4Km 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 4Km 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 4Km 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 4Km core latches data internally whenever *EB_RVal* 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.

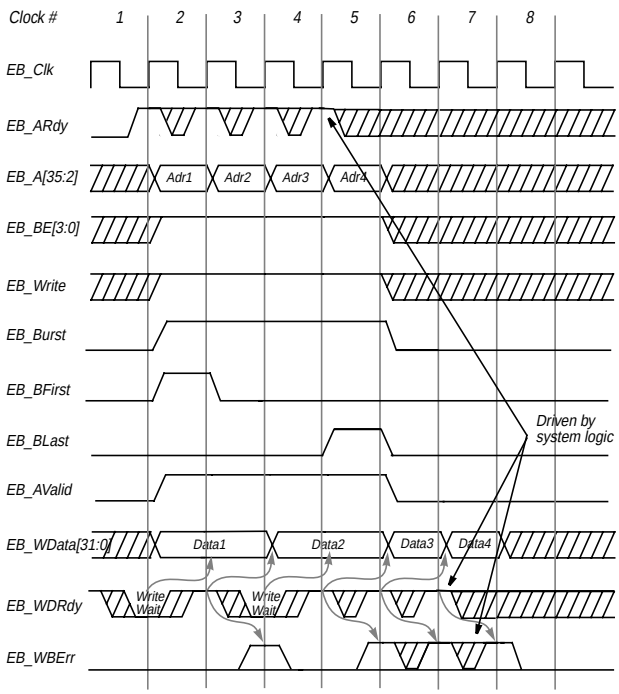


Figure 10 Burst Write Transaction Timing Diagram

The 4Km 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 4Km core continues to drive *EB_AValid* as long as a valid address is on the bus.

The 4Km 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 4Km 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 4Km 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 4Km 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.

Document Number: MD00040
01.03-2B