# MIPS32® M4K® Processor Core Datasheet

**March 4, 2008**

The MIPS32® M4K® core from MIPS® Technologies is a member of the MIPS32 M4K® 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 M4K core is ideally positioned to support new products for emerging segments of the routing, network access, network storage, residential gateway, and smart mobile device markets. It is especially well-suited for microcontroller and hardware accelerator applications, as well as systems requiring multiple cores, when high performance density is critical.

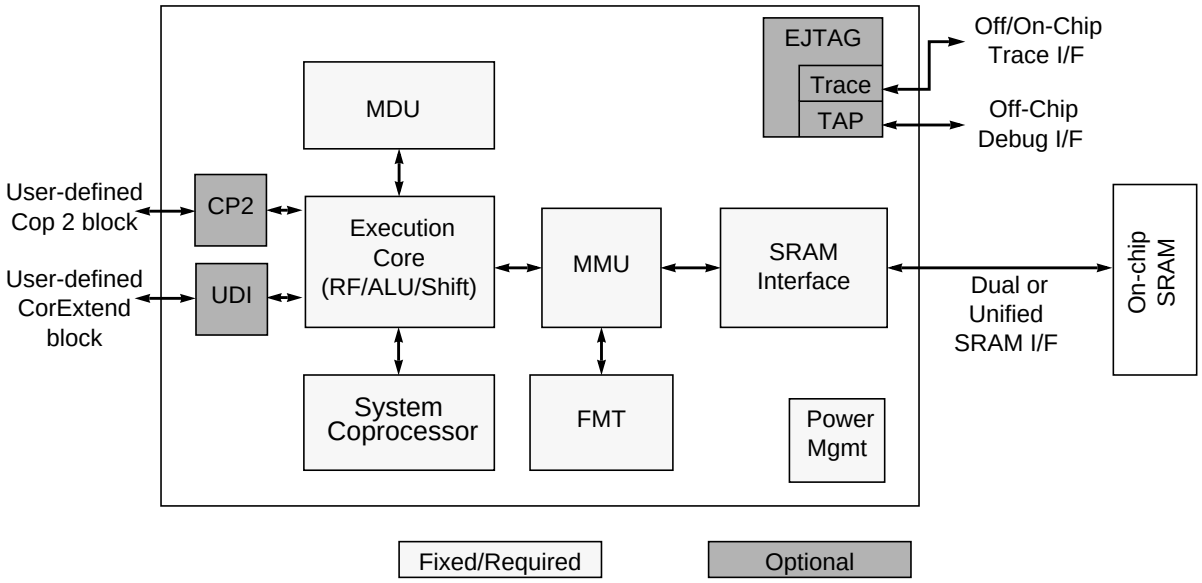
The synthesizable M4K core implements the MIPS32 Release 2 Architecture with the MIPS16e™ ASE.The Memory Management Unit (MMU) consists of a simple, Fixed Mapping Translation (FMT) mechanism for applications that do not require the full capabilities of a Translation Lookaside Buffer (TLB) based MMU. The core includes two different Multiply/Divide Unit (MDU) implementations, selectable at build-time, allowing the implementor to trade off performance and area. The high-performance MDU option that implements single cycle 32x16-bit MAC instructions or two cycle 32x32-bit, which enable DSP algorithms to be performed efficiently. The area-efficient MDU option handles multiplies with a one-bit-per-clock iterative algorithm.

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

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 M4K core. The core is divided into *required* and *optional* blocks as shown.

**Figure 1 MIPS32® M4K® 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)
- 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 (one, three or seven additional shadows can be optionally 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
- Memory Management Unit
    - Simple Fixed Mapping Translation (FMT) mechanism
- Simple SRAM-Style Interface
    - Cacheless operation enables deterministic response and reduces size
    - 32-bit address and data; input byte enables enable simple connection to narrower devices
    - Single or multi-cycle latencies
    - Configuration option for dual or unified instruction/data interfaces
    - Redirection mechanism on dual I/D interfaces permits D-side references to be handled by I-side
    - Transactions can be aborted
- CorExtend® User Defined Instruction Set Extensions (available in Pro Series™ 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 M4K® Pro™ core
- Multi-Core Support
    - External lock indication enables multi-processor semaphores based on LL/SC instructions
    - External sync indication allows memory ordering
    - Reference design provided for cross-core debug triggers
- Multiply/Divide Unit (high-performance configuration)
    - 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)
- Multiply/Divide Unit (area-efficient configuration)
    - 32 clock latency on multiply
    - 34 clock latency on multiply-accumulate
    - 33-35 clock latency on divide (sign-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 and MIPS Trace
    - Support for single stepping
    - Virtual instruction and data address/value breakpoints
    - Complex breakpoint unit allows more detailed specification of break conditions
    - PC and/or data tracing w/ trace compression
    - 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)

# Architecture Overview

The M4K 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 M4K 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)
- SRAM Interface
- Power Management

Optional or configurable blocks include:

- Coprocessor 2 interface
- CorExtend® User Defined Instruction (UDI) interface
- MIPS16e support
- Enhanced JTAG (EJTAG) Controller

The section entitled "MIPS32® M4K® Core Required Logic Blocks" on page 4 discusses the required blocks. The section entitled "MIPS32® M4K® Core Optional or Configurable Logic Blocks" on page 12 discusses the optional blocks.

# Pipeline Flow

The M4K 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.

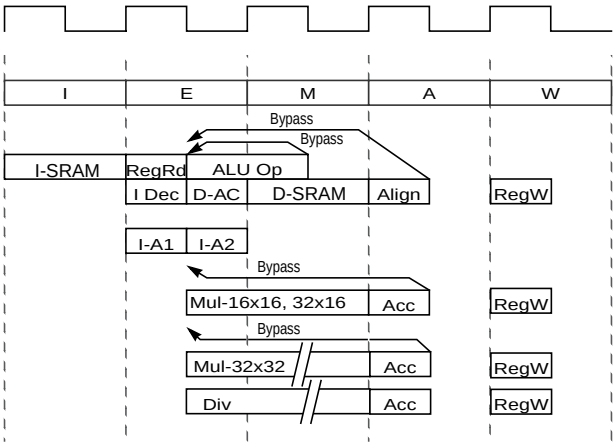
The M4K core pipeline consists of five stages:

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

The M4K 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 M4K core pipeline (shown with the high performance MDU).

**Figure 2 MIPS32® M4K® Core Pipeline**




### *I Stage: Instruction Fetch*

During the Instruction fetch stage:

- An instruction is fetched from instruction SRAM.
- 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, and the MMU performs the fixed virtual-to-physical address translation.
- The ALU determines whether the branch condition is true and calculates the virtual branch target address for branch instructions.
- Instruction logic selects an instruction address.
- 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 SRAM access is performed for load and store instructions.
- A 16x16 or 32x16 multiply calculation completes (high-performance MDU option).

- A 32x32 multiply operation stalls the MDU pipeline for one clock in the M stage (high-performance MDU option).
- A multiply operation stalls the MDU pipeline for 31 clocks in the M stage (area-efficient MDU option).
- A multiply-accumulate operation stalls the MDU pipeline for 33 clocks in the M stage (area-efficient MDU option).
- 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 (only the divider in the fast MDU option supports early-in detection).

#### *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.
- EJTAG complex break conditions are evaluated

#### *W Stage: Writeback*

During the Writeback stage:

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

# MIPS32® M4K® Core Required Logic Blocks

The M4K 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)
- SRAM Interface
- Power Management

## Execution Unit

The M4K core execution unit implements a load/store architecture with single-cycle ALU operations (logical, shift, add, subtract) and an autonomous multiply/divide unit. The M4K core contains thirty-two 32-bit general-purpose registers used for integer operations and address calculation. Optionally, one, three, or seven 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 M4K 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 allows the long-running MDU operations to be partially masked by system stalls and/or other integer unit instructions.

Two configuration options exist for the MDU: an area efficient, iterative block and a higher performance 32x16 array. The selection of the MDU style allows the implementor to determine the appropriate trade-off for his/her application.

### Area-Efficient MDU Option

With the area-efficient option, multiply and divide operations are implemented with a simple 1 bit per clock iterative algorithm. Any attempt to issue a subsequent MDU instruction while a multiply/divide is still active causes an MDU pipeline stall until the operation is completed.

Table 1 lists the latency (number of cycles until a result is available) for the M4K core multiply and divide instructions. The latencies are listed in terms of pipeline clocks.

**Table 1 Area-Efficient Integer Multiply/Divide Unit Operation Latencies**

| Opcode | Operand Sign | Latency |
|---|---|---|
| MUL, MULT, MULTU | any | 32 |
| MADD, MADDU, MSUB, MSUBU | any | 34 |
| DIVU | any | 33 |
| DIV | pos/pos | 33 |
| | any/neg | 34 |
| | neg/pos | 35 |

The MIPS architecture defines that the results 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.

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

### High-Performance MDU

The M4K 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 high-performance 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 M4K 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 2 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 M4K 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 M4K® Processor Core Family Software User’s Manual*.

**Table 2 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, 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 presence of build-time options like MIPS16e or coprocessor 2 interface, is also available by accessing the CP0 registers, listed in Table 3.

**Table 3 Coprocessor 0 Registers in Numerical Order**

| Register Number | Register Name | Function |
|---|---|---|
| 0-6 | Reserved | Reserved in the M4K core. |
| 7 | *HWREna* | Enables access via the RDHWR instruction to selected hardware registers. |
| 8 | *BadVAddr*[1] | Reports the address for the most recent address-related exception. |
| 9 | *Count*[1] | Processor cycle count. |
| 10 | Reserved | Reserved in the M4K core. |
| 11 | *Compare*[1] | Timer interrupt control. |
| 12 | *Status*[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. |
| 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- | Reserved | Reserved in the M4K core. |
| 23 | *Debug*[2] | Debug control and exception status. |
| 23 | *Debug2*[2] | Complex breakpoint 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. |
| 25-29 | Reserved | Reserved in the M4K core. |
| 30 | *ErrorEPC*[1] | Program counter at last error. |
| 31 | *DESAVE*[2] | Debug handler scratchpad register. |

1. Registers used in exception processing.
2. Registers used 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 4 shows the exception types in order of priority.

**Table 4 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 *SI_NMI* signal. |
| Interrupt | Assertion of unmasked hardware or software interrupt signal. |
| DIB | EJTAG debug hardware instruction break matched. |
| AdEL | Fetch address alignment error.<br>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. |
| CEU | Execution of a CorExtend instruction when CorExtend 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). |
| AdEL | Load address alignment error.<br>Load reference to protected address. |
| AdES | Store address alignment error.<br>Store to protected address. |
| DBE | Load or store bus error. |
| DDBL | EJTAG data hardware breakpoint matched in load data compare. |

### Interrupt Handling

The M4K 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 M4K core, so the VInt bit will always read as a 1 for the M4K core.
- External Interrupt Controller (EIC) mode, which redefines the way in which interrupts are handled to provide full support for an external interrupt controller handling prioritization and vectoring of interrupts. This presence of this mode denoted by the VEIC bit in the *Config3* register. Again, this mode is architecturally optional. On the M4K core, the VEIC bit is set externally by the static input, *SI_EICPresent*, to allow system logic to indicate the presence of an external interrupt controller.

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

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 may be a build-time option on some MIPS core. Although Release 2 of the Architecture defines a maximum of 16 shadow sets, the M4K core allows one (the normal GPRs), two, four, or eight shadow sets. The highest number actually implemented is indicated by the $SRSCtl_{HSS}$ field. If this field is zero, only the normal GPRs are implemented.

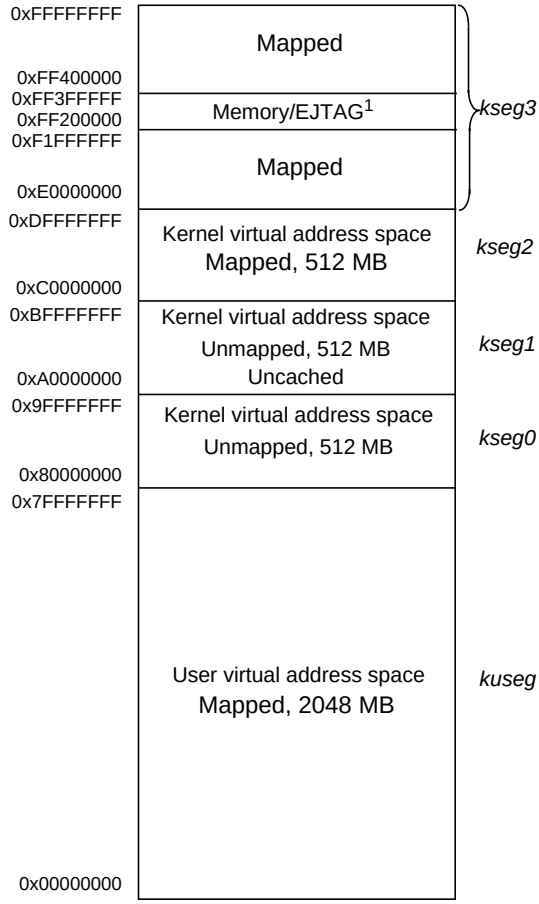
Shadow sets are new copies of the GPRs that can be substituted for the normal GPRs on entry to kernel mode via an interrupt or exception. Once a shadow set is bound to a kernel mode entry condition, reference to GPRs work exactly as one would expect, but they are redirected to registers that are dedicated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

If the processor is operating in VI interrupt mode, binding of a vectored interrupt to a shadow set is done by writing to the *SRSMap* register. If the processor is operating in EIC interrupt mode, the binding of the interrupt to a specific shadow set is provided by the external interrupt controller, and is configured in an implementation-dependent way. Binding of an exception or non-vectored interrupt to a shadow set is done by writing to the *ESS* field of the *SRSCtl* register. When an exception or interrupt occurs, the value of $SRSCtl_{CSS}$ is copied to $SRSCtl_{PSS}$, and $SRSCtl_{CSS}$ is set to the value taken from the appropriate source. On an ERET, the value of $SRSCtl_{PSS}$ is copied back into $SRSCtl_{CSS}$ to restore the shadow set of the mode to which control returns.

## Modes of Operation

The M4K 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.

**Figure 3 M4K Core Virtual Address Map**



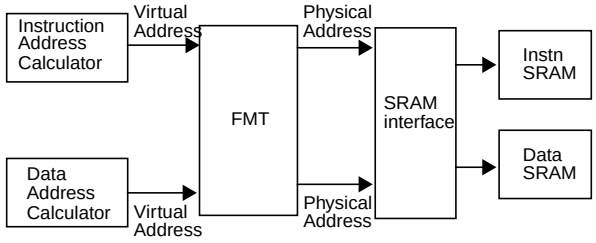

1. This space is mapped to memory in user or kernel mode, and by the EJTAG module in debug mode.

## Memory Management Unit (MMU)

The M4K core contains an MMU that interfaces between the execution unit and the SRAM controller. The M4K 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 M4K core.

**Figure 4 Address Translation During SRAM Access**




In general, the FMT also determines the cacheability of each segment. These attributes are controlled via bits in the *Config* register. Table 5 shows the encoding for the *K23* (bits 30:28), *KU* (bits 27:25), and *K0* (bits 2:0) fields of the *Config* register. Since the M4K core does not contain caches, these fields are not used within the core and all references are treated as uncached. The values are reported on the external bus for use by any external caching mechanisms that may be present. Table 6 shows how the cacheability of the virtual address segments is controlled by these fields.

**Table 5 Cache Coherency Attributes**

| Config Register Fields K23, KU, and K0 | Cache Coherency Attribute |
|---|---|
| 2 | Uncached. |
| 3 | Cached. |

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

**Table 6 Cacheability of Segments with Fixed Mapping Translation**

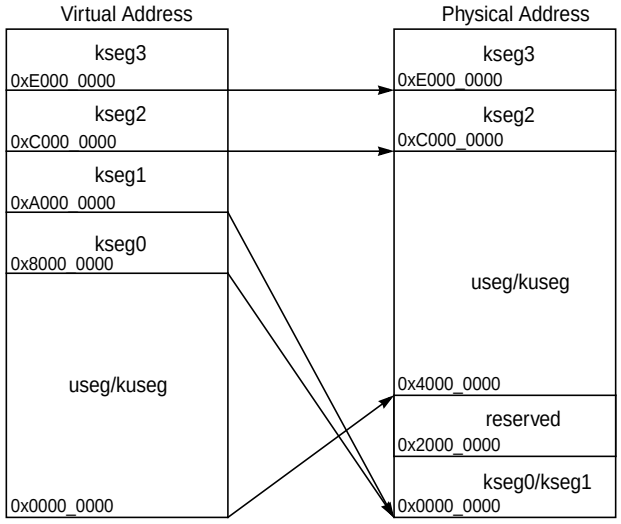
| Segment | Virtual Address Range | Cacheability |
|---|---|---|
| useg/kuseg | 0x0000_0000-0x7FFF_FFFF | Controlled by the *KU* field (bits 27:25) of the *Config* register. See Table 5 for mapping. This segment is always uncached when *ERL* = 1. |
| kseg0 | 0x8000_0000-0x9FFF_FFFF | Controlled by the *K0* field (bits 2:0) of the *Config* register. See Table 5 for mapping. |

**Table 6 Cacheability of Segments with Fixed Mapping Translation (Continued)**

| Segment | Virtual Address Range | Cacheability |
|---|---|---|
| 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 5 for mapping. |
| kseg3 | 0xE000_0000-0xFFFF_FFFF | Controlled by the *K23* field (bits 30:28) of the *Config* register. See Table 5 for mapping. |

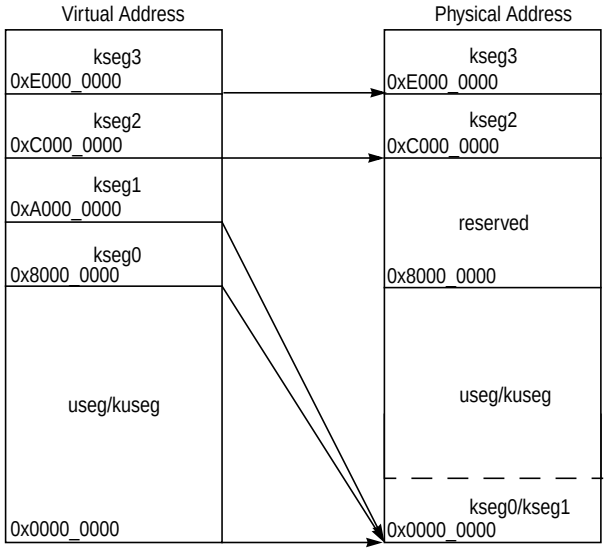
The FMT performs a simple translation to map from virtual addresses to physical addresses. This mapping is shown in Figure 5.

**Figure 5 FMT Memory Map (ERL=0) in the M4K 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 M4K Core**




## SRAM Interface Controller

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

### Dual or Unified Interfaces

The SRAM interface includes a build-time option to select either dual or unified instruction and data interfaces.

The dual interface enables independent connection to instruction and data devices. It generally yields the highest performance, since the The pipeline can generate simultaneous I and D requests which are then serviced in parallel.

For simpler or cost-sensitive systems, it is also possible to combine the I and D interfaces into a common interface that services both types of requests. If I and D requests occur simultaneously, priority is given to the D side.

### Backstalling

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

### Redirection

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

### Transaction Abort

The core may request a transaction (fetch/load/store/sync) to be aborted. This is particularly useful in case of interrupts. Since the core does not know whether transactions are re-startable, it cannot arbitrarily interrupt a request which has been initiated on the SRAM interface. However, cycles spent waiting for a multi-cycle transaction to complete can directly impact interrupt latency. In order to minimize this effect, the interface supports an abort mechanism. The core requests an abort whenever an interrupt is detected and a transaction is pending (abort of an instruction fetch may also be requested in other cases). The external system logic can choose to acknowledge the abort or can choose to ignore the abort request.

### MIPS16e™ Instruction Execution

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

### Connecting to Narrower Devices

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

### Lock Mechanism

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

### Sync Mechanism

The interface includes a protocol that externalizes the execution of the SYNC instruction. External logic might

choose to use this information to enforce memory ordering between various elements in the system.

### External Call Indication

The instruction fetch interface contains signals that indicate that the core is fetching the target of a subroutine call-type instruction such as JAL or BAL. At some point after a call, there will typically be a return to the original code sequence. If a system prefetches instructions, it can make use of this information to save instructions that were prefetched and are likely to be executed after the return.

### SimpleBE Mode

To aid in attaching the M4K 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 7.

**Table 7 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 a tri-byte load (LWL/LWR). Since external logic can easily convert a tri-byte read into a full word read if desired, no conversion is done by the core for this case in SimpleBE mode.

Writes with non-simple byte enable patterns can arise from 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.

## Hardware Reset

For historical reasons within the MIPS architecture, the M4K 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 8.

**Table 8 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 M4K 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 M4K 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 M4K 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 M4K 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 M4K core is waiting for an interrupt.

### Local clock gating

The majority of the power consumed by the M4K 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.

## MIPS32® M4K® Core Optional or Configurable Logic Blocks

The M4K core contains several optional or configurable logic blocks shown in the block diagram in Figure 1.

### MIPS16e™ Application Specific Extension

The M4K 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 M4K 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 M4K 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

An 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 external to the M4K 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 M4K 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 M4K 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 M4K 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, DEBUG2, DEPC, and DESAVE) have been added to the MIPS Coprocessor 0 (CP0) register set. The DEBUG and DEBUG2 registers show 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 M4K core: Instruction breakpoints and Data breakpoints. Additionally, complex hardware breakpoints can be included which allow detection of more intricate sequences of events.

The M4K core can be configured with the following breakpoint options:

- No data, instruction, or complex breakpoints
- One data and two instruction breakpoints without complex breakpoints
- Two data and four instruction breakpoints without complex breakpoints
- Two data and six instruction breakpoints with complex 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.

Complex breakpoints utilize the simple instruction and data breakpoints and break when combinations of events are seen. Complex break features include

- Pass Counters - Each time a matching condition is seen, a counter is decremented. The break or trigger will only be enabled once the counter has counted down to 0
- Tuples - A tuple is the pairing of an instruction and a data breakpoint. The tuple will be taken if both the instruction and data break conditions are met on the same instruction.
- Priming - This allows a breakpoint to be enabled only after other break conditions have been met.
- Qualified - This feature uses a data breakpoint to qualify when an instruction breakpoint can be taken. Once a load matches the data address and the data value, the instruction break will be enabled. If a load matches the address, but has mis-matching data, the instruction break will be disabled.

### MIPS Trace

The M4K core includes optional MIPS Trace 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.

### iFlowtrace™ mechanism

The M4K core also has an option for a simpler trace scheme called the iFlowtrace mechanism. This scheme only traces instruction addresses and not data addresses or values. This simplification allows the trace block to be smaller and the trace compression to be more efficient.

## 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 on-chip trace memory is optional.

Memory BIST can 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.

## Build-Time Configuration Options

The M4K core allows a number of features to be customized based on the intended application. Table 9 summarizes the key configuration options that can be selected when the core is synthesized and implemented.

For a core that has already been built, software can determine the value of many of these options by querying an appropriate register field. Refer to the *MIPS32® M4K® Processor Core Family Software User's Manual* for a more complete description of these fields. The value of some options that do not have a functional effect on the core are not visible to software.

**Table 9 Build-time Configuration Options**

| Option | Choices | Software Visibility |
|---|---|---|
| Integer register file sets | 1, 2, 4, or 8 | $SRSCtl_{HSS}$ |
| Integer register file implementation style | Flops or generator | N/A |
| MIPS16e support | Present or not | $Config1_{CA}$ |
| Multiply/divide implementation style | High performance or min area | $Config_{MDU}$ |
| EJTAG TAP controller | Present or not | N/A |
| Instruction/data hardware breakpoints | 0/0, 2/1, 4/2, or 6/2 | $DCR_{IB}$, $IBS_{BCN}$ $DCR_{DB}$, $DBS_{BCN}$ |
| Complex breakpoints | Present or not | $DCR_{CBT}$ |
| iFlowtrace hardware | Present or not | $Config3_{ITL}$ |
| MIPS Trace support | Present or not | $Config3_{TL}$ |
| MIPS Trace memory location | On-core or off-chip | $TCBCONFIG_{OnT}$, $TCBCONFIG_{OfT}$ |
| MIPS Trace on-chip memory size | 256B - 8MB | $TCBCONFIG_{SZ}$ |
| MIPS Trace triggers | 0 - 8 | $TCBCONFIG_{TRIG}$ |
| CorExtend interface (Pro only) | Present or not | $Config_{UDI}$* |
| * These bits indicate the presence of an external block. Bits will not be set if interface is present, but block is not. | | |

**Table 9 Build-time Configuration Options (Continued)**

| Option | Choices | Software Visibility |
|---|---|---|
| Coprocessor2 interface | Present or not | $Config1_{C2}$* |
| SRAM interface style | Separate instruction/data or unified | $Config_{DS}$ |
| Interrupt synchronizers | Present or not | N/A |
| Clock gating | Top-level, integer register file array, fine-grain, or none | N/A |
| * These bits indicate the presence of an external block. Bits will not be set if interface is present, but block is not. | | |

# Instruction Set

The M4K core instruction set complies with the MIPS32 instruction set architecture. Table 10 provides a summary of instructions implemented by the M4K core.

**Table 10 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 & (016 \|\| Immed)` |
| B | Unconditional Branch<br>(Assembler idiom for: BEQ r0, r0, offset) | `PC += (int)offset` |
| BAL | Branch and Link<br>(Assembler idiom for: BGEZAL r0, offset) | `GPR[31] = PC + 8`<br>`PC += (int)offset` |
| BC2F | Branch On COP2 Condition False | `if COP2Condition(cc) == 0`<br>`  PC += (int)offset` |
| BC2FL | Branch On COP2 Condition False Likely | `if COP2Condition(cc) == 0`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BC2T | Branch On COP2 Condition True | `if COP2Condition(cc) == 1`<br>`  PC += (int)offset` |
| BC2TL | Branch On COP2 Condition True Likely | `if COP2Condition(cc) == 1`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BEQ | Branch On Equal | `if Rs == Rt`<br>`  PC += (int)offset` |

**Table 10 Core Instruction Set (Continued)**

| Instruction | Description | Function |
|---|---|---|
| BEQL | Branch On Equal Likely | `if Rs == Rt`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BGEZ | Branch on Greater Than or Equal To Zero | `if !Rs[31]`<br>`  PC += (int)offset` |
| BGEZAL | Branch on Greater Than or Equal To Zero And Link | `GPR[31] = PC + 8`<br>`if !Rs[31]`<br>`  PC += (int)offset` |
| BGEZALL | Branch on Greater Than or Equal To Zero And Link Likely | `GPR[31] = PC + 8`<br>`if !Rs[31]`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BGEZL | Branch on Greater Than or Equal To Zero Likely | `if !Rs[31]`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BGTZ | Branch on Greater Than Zero | `if !Rs[31] && Rs != 0`<br>`  PC += (int)offset` |
| BGTZL | Branch on Greater Than Zero Likely | `if !Rs[31] && Rs != 0`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BLEZ | Branch on Less Than or Equal to Zero | `if Rs[31] \|\| Rs == 0`<br>`  PC += (int)offset` |
| BLEZL | Branch on Less Than or Equal to Zero Likely | `if Rs[31] \|\| Rs == 0`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BLTZ | Branch on Less Than Zero | `if Rs[31]`<br>`  PC += (int)offset` |
| BLTZAL | Branch on Less Than Zero And Link | `GPR[31] = PC + 8`<br>`if Rs[31]`<br>`  PC += (int)offset` |
| BLTZALL | Branch on Less Than Zero And Link Likely | `GPR[31] = PC + 8`<br>`if Rs[31]`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BLTZL | Branch on Less Than Zero Likely | `if Rs[31]`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BNE | Branch on Not Equal | `if Rs != Rt`<br>`  PC += (int)offset` |
| BNEL | Branch on Not Equal Likely | `if Rs != Rt`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |

**Table 10 Core Instruction Set (Continued)**

| Instruction | Description | Function |
|---|---|---|
| BREAK | Breakpoint | `Break Exception` |
| 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` |
| DERET | Return from Debug Exception | `PC = DEPC`<br>`Exit Debug Mode` |
| DI | Atomically Disable Interrupts | `Rt = Status; Status`$_{IE}$ `= 0` |
| DIV | Divide | `LO = (int)Rs / (int)Rt`<br>`HI = (int)Rs % (int)Rt` |
| DIVU | Unsigned Divide | `LO = (uns)Rs / (uns)Rt`<br>`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]`<br>`  PC = ErrorEPC`<br>`else`<br>`  PC = EPC`<br>`  SR[1] = 0`<br>`SR[2] = 0`<br>`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`<br>`PC = PC[31:28] \|\| offset<<2` |
| JALR | Jump and Link Register | `Rd = PC + 8`<br>`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`<br>`PC = Rs` |
| JR | Jump Register | `PC = Rs` |
| JR.HB | Jump Register with Hazard Barrier | `Like JR, but also clears execution and instruction hazards` |

**Table 10 Core Instruction Set (Continued)**

| Instruction | Description | Function |
|---|---|---|
| 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]` |
| LL | Load Linked Word | `Rt = Mem[Rs+offset]`<br>`LL = 1`<br>`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 Architecture Reference Manual` |
| LWR | Load Word Right | `See Architecture Reference Manual` |
| MADD | Multiply-Add | `HI \| LO += (int)Rs * (int)Rt` |
| MADDU | Multiply-Add Unsigned | `HI \| LO += (uns)Rs * (uns)Rt` |
| MFC0 | Move From Coprocessor 0 | `Rt = CPR[0, Rd, sel]` |
| MFC2 | Move From 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`<br>`  Rd = Rs` |
| MOVZ | Move Conditional on Zero | `if Rt = 0 then`<br>`  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` |

**Table 10 Core Instruction Set (Continued)**

| Instruction | Description | Function |
|---|---|---|
| MUL | Multiply with register write | HI \| LO =Unpredictable<br>Rd = ((int)Rs * (int)Rt)$_{31..0}$ |
| MULT | Integer Multiply | HI \| LO = (int)Rs * (int)Rd |
| MULTU | Unsigned Multiply | HI \| LO = (uns)Rs * (uns)Rd |
| NOP | No Operation<br>(Assembler idiom for: SLL r0, r0, r0) | |
| NOR | Logical NOR | Rd = ~(Rs \| Rt) |
| OR | Logical OR | Rd = Rs \| Rt |
| ORI | Logical OR Immediate | Rt = Rs \| Immed |
| RDHWR | Read Hardware Register | Allows unprivileged access to registers enabled by HWREna register |
| RDPGPR | Read GPR from Previous Shadow Set | Rt = SGPR[SRSCtl$_{PSS}$, Rd] |
| RESTORE | Restore registers and deallocate stack frame (MIPS16 only) | See Architecture Reference Manual |
| ROTR | Rotate Word Right | Rd = Rt$_{sa-1..0}$ \|\| Rt$_{31..sa}$ |
| ROTRV | Rotate Word Right Variable | Rd = Rt$_{Rs-1..0}$ \|\| Rt$_{31..Rs}$ |
| SAVE | Save registers and allocate stack frame (MIPS16 only) | See Architecture Reference Manual |
| SB | Store Byte | (byte)Mem[Rs+offset] = Rt |
| SC | Store Conditional Word | if LL = 1<br>  mem[Rs+offset] = Rt<br>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<br>  Rd = 1<br>else<br>  Rd = 0 |
| SLTI | Set on Less Than Immediate | if (int)Rs < (int)Immed<br>  Rt = 1<br>else<br>  Rt = 0 |
| SLTIU | Set on Less Than Immediate Unsigned | if (uns)Rs < (uns)Immed<br>  Rt = 1<br>else<br>  Rt = 0 |

**Table 10 Core Instruction Set (Continued)**

| Instruction | Description | Function |
|---|---|---|
| SLTU | Set on Less Than Unsigned | `if (uns)Rs < (uns)Rt`<br>`  Rd = 1`<br>`else`<br>`  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` |
| SWC2 | Store Word From Coprocessor 2 | `Mem[Rs+offset] = CPR[2,n,0]` |
| SWL | Store Word Left | `See Architecture Reference Manual` |
| SWR | Store Word Right | `See Architecture Reference Manual` |
| SYNC | Synchronize | `See Software User's Manual` |
| SYSCALL | System Call | `SystemCallException` |
| TEQ | Trap if Equal | `if Rs == Rt`<br>`  TrapException` |
| TEQI | Trap if Equal Immediate | `if Rs == (int)Immed`<br>`  TrapException` |
| TGE | Trap if Greater Than or Equal | `if (int)Rs >= (int)Rt`<br>`  TrapException` |
| TGEI | Trap if Greater Than or Equal Immediate | `if (int)Rs >= (int)Immed`<br>`  TrapException` |
| TGEIU | Trap if Greater Than or Equal Immediate Unsigned | `if (uns)Rs >= (uns)Immed`<br>`  TrapException` |
| TGEU | Trap if Greater Than or Equal Unsigned | `if (uns)Rs >= (uns)Rt`<br>`  TrapException` |
| TLT | Trap if Less Than | `if (int)Rs < (int)Rt`<br>`  TrapException` |
| TLTI | Trap if Less Than Immediate | `if (int)Rs < (int)Immed`<br>`  TrapException` |
| TLTIU | Trap if Less Than Immediate Unsigned | `if (uns)Rs < (uns)Immed`<br>`  TrapException` |
| TLTU | Trap if Less Than Unsigned | `if (uns)Rs < (uns)Rt`<br>`  TrapException` |
| TNE | Trap if Not Equal | `if Rs != Rt`<br>`  TrapException` |

**Table 10 Core Instruction Set (Continued)**

| Instruction | Description | Function |
|---|---|---|
| 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 | $Rd = Rt_{23..16} \mid\mid Rt_{31..24} \mid\mid Rt_{7..0} \mid\mid Rt_{15..8}$ |
| XOR | Exclusive OR | `Rd = Rs ^ Rt` |
| XORI | Exclusive OR Immediate | `Rt = Rs ^ (uns)Immed` |
| ZEB | Zero extend byte (MIPS16 only) | `Rt = (ubyte) Rs` |
| ZEH | Zero extend half (MIPS16 only) | `Rt = (uhalf) Rs` |

# External Interface Signals

This section describes the signal interface of the M4K microprocessor core.

The pin direction key for the signal descriptions is shown in Table 11 below.

The M4K core signals are listed in Table 12 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 Core Signal Direction Key**

| Dir | Description |
|---|---|
| I | Input to the M4K core sampled on the rising edge of the appropriate CLK signal. |
| O | Output of the M4K 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 M4K core. These signals are normally tied to either power or ground and should not change state while *SI_ColdReset* is deasserted. |

**Table 12 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. |

**Table 12 Signal Descriptions (Continued)**

| Signal Name | Type | Description |
|---|---|---|
| *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 and Processor State 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 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. |
| *SI_Ibs[5:0]* | Out | Reflects state of breakpoint status (*BS*) field in the *Instruction Breakpoint Status* (*IBS*) register. These bits are set when the corresponding break condition has matched, for breaks enabled as either a breakpoints or triggerpoints. If fewer than 6 instruction breakpoints exist, the unimplemented bits are tied to 0. |
| *SI_Dbs[1:0]* | Out | Reflects state of breakpoint status (*BS*) field in the *Data Breakpoint Status* (*DBS*) register. These bits are set when the corresponding break condition has matched, for breaks enabled as either a breakpoints or triggerpoints. If fewer than 2 data breakpoints exist, the unimplemented bits are tied to 0. |
| *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_EICVector[5:0]* | In | Provides the vector number for an interrupt request in External Interrupt Controller (EIC) mode. (Note: This input decouples the interrupt priority from the vector offset. For compatibility with earlier Release 2 cores in EIC mode, connect *SI_Int[5:0]* and *SI_EICVector[5:0]* together.) |
| *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. |

**Table 12 Signal Descriptions (Continued)**

| Signal Name | Type | Description |
|---|---|---|
| *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.<br>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.<br>The interrupt pins are level sensitive and should remain asserted until the interrupt has been serviced.<br>In Release 1 Interrupt Compatibility mode:<br>All 6 interrupt pins have the same priority as far as the hardware is concerned.<br>Interrupts are non-vectored.<br>In Vectored Interrupt (VI) mode:<br>The *SI_Int* pins are interpreted as individual hardware interrupt requests.<br>Internally, the core prioritizes the hardware interrupts and chooses an interrupt vector.<br>In External Interrupt Controller (EIC) mode:<br>An external block prioritizes its various interrupt requests and produces a vector number of the highest priority interrupt to be serviced.<br>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.<br>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. |
| *SI_IPL[5:0]* | O | Current interrupt priority level from the $Cause_{IPL}$ register field, provided for use by an external interrupt controller. This value is updated whenever *SI_IAck* is asserted. |
| *SI_IPTI[2:0]* | S | Timer interrupts can be muxed or ORed into one of the interrupts, as desired in a particular system. This input indicates which *SI_Int* hardware interrupt pin 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. |
| *SI_SWInt[1:0]* | O | 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. |

| SI_IPTI | Combined w/ SI_Int |
|---|---|
| 0-1 | None |
| 2 | SI_Int[0] |
| 3 | SI_Int[1] |
| 4 | SI_Int[2] |
| 5 | SI_Int[3] |
| 6 | SI_Int[4] |
| 7 | SI_int[5] |

**Table 12 Signal Descriptions (Continued)**

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></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.<br>For Release 1 Interrupt Compatibility mode or Vectored Interrupt mode:<br>In order to generate a timer interrupt, the SI_TimerInt signal needs to be brought back into the M4K 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.<br>For External Interrupt Controller (EIC) mode:<br>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_SimpleBE[1:0]</td><td>S</td><td>The state of these signals can constrain the core to only generate certain byte enables on SRAM-style interface writes. 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>SI_SRSDisable[2:0]</td><td>S</td><td>Disable the use of some shadow register sets:<br>000 - Use all register sets<br>100 - Only use 4 register sets<br>110 - Only use 2 register sets<br>111 - Only use 1 register set</td></tr>
<tr><td colspan="3">SRAM-style Interface</td></tr>
<tr><td colspan="3">The SRAM-style interface allows simple connection to fast, tightly-coupled memory devices. It can be configured with independent interfaces for Instruction and Data, or a Unified interface. Signals related to the I-side interface are prefixed with “IS_”; signals related to the D-side interface are prefixed with “DS_”. When the Unified interface is used, then most D-side signals are obsoleted, since they have an I-side equivalent; only the write data, DS_WData, continues to be used from the D-side.</td></tr>
<tr><td>IS_Read</td><td>O</td><td>Read strobe.</td></tr>
</table>

**Table 12 Signal Descriptions (Continued)**

| Signal Name | Type | Description |
|---|---|---|
| *IS_Write* | O | Write strobe. Only asserted due to a redirected data write. |
| *IS_Sync* | O | Sync strobe. |
| *IS_WbCtl* | O | Write buffer control.<br>This signal is asserted when the M4K core can guarantee that no I-side read transaction will be started in the current clock cycle. For the purpose of generating this signal, if there is a pending transaction, the M4K core assumes that it will end in this cycle, in order to determine whether a new read transaction might be started or not.<br>Unlike *IS_Read*, there is no asynchronous path from *IS_Stall* or any other input signal to *IS_WbCtl.* Also, it is an earlier signal than *IS_Read*.<br>It is intended to be used by an external agent to control flushing of a write buffer (if a write buffer is present). |
| *IS_Instr* | O | Indicates instruction fetch when high, or redirected data read/write when low. |
| *IS_Addr[31:2]* | O | Address of transaction. When *IS_Sync* is asserted high, *IS_Addr[10:6]* holds the “sync type” (the “stype” field of SYNC instruction). |
| *IS_BE[3:0]* | O | Byte enable signals for transaction.<br>*IS_BE[3]* enables byte lane corresponding to bits *31:24*.<br>*IS_BE[2]* enables byte lane corresponding to bits *23:16*.<br>*IS_BE[1]* enables byte lane corresponding to bits *15:8*.<br>*IS_BE[0]* enables byte lane corresponding to bits *7:0*. |
| *IS_Abort* | O | Request for transaction to be aborted, if possible. It is optional whether the external logic uses this signal or not, although using it may reduce interrupt latency. Completion of any transaction (aborted or not) is always communicated through *IS_Stall*. Whether the transaction was in fact aborted is signalled using *IS_AbortAck*.<br>*IS_Abort* is asserted through (and including) the cycle where *IS_Stall* is deasserted. |
| *IS_EjtBreakEn* | O | One or more EJTAG instruction breakpoints are enabled. This signal is also asserted for the Unified Interface when one or more data breakpoints are enabled. |
| *IS_EjtBreak* | O | Asserted when an instruction break is detected. Also asserted for the Unified Interface when a data break is detected. May be used by external logic to cancel the current transaction. External logic may determine whether this is an instruction break or a data break based on *IS_Instr*.<br>This signal is asserted one cycle after the transaction start, so when precise breaks are required, the external logic must stall transactions by one cycle if *IS_EjtBreakEn* indicates that a break may occur.<br>*IS_EjtBreak* is asserted through (and including) the cycle where *IS_Stall* is deasserted. |
| *IS_Lock* | O | Asserted when a read transaction is due to a redirected LL (load linked) instruction, |
| *IS_Unlock* | O | Asserted when a write transaction is due to a redirected SC (store conditional) instruction. |
| *IS_UnlockAll* | O | Asserted for one clock cycle when an ERET instruction is executed. |
| *IS_WasCall* | Out | Indicates that a recent fetch was for a control transfer instruction that saves a return address in a GPR (JAL, JALR, JALX, BGEZAL, BGEZALL, BLTZAL, BLTZALL). This indication and a corresponding offset may enable external logic to maintain a buffer of instructions at the return address. |

**Table 12 Signal Descriptions (Continued)**

| Signal Name | Type | Description |
|---|---|---|
| *IS_LinkOffset* | Out | Offset used to determine the link return fetch address relative to the previous fetch address. This signal is only valid when *IS_WasCall* is asserted in the same cycle. The link return address is relative to the fetch immediately prior to the one in which *IS_WasCall* is asserted. When *IS_LinkOffset* is 0, the return address will be within the same word as the prior fetch. When *IS_LinkOffset* is 1, the return address will be within the next sequential word from the prior fetch. |
| *IS_CCA[2:0]* | Out | Provides the cache coherence attribute (CCA) for the current fetch request. The core will limit this value to either 2 (uncacheable) or 3 (cacheable) |
| *IS_Stall* | I | Indicates that the transaction is not ready to be completed. |
| *IS_Error* | I | Valid in the cycle terminating the transaction (*IS_Stall* deasserted). Asserted high if transaction caused an error. Causes bus error exception to be taken by the core. |
| *IS_AbortAck* | I | Valid in the cycle terminating the transaction (*IS_Stall* deasserted). Asserted high if transaction was aborted.<br>If no abort was requested (*IS_Abort* is low), and *IS_AbortAck* is asserted high in the cycle terminating the transaction, a bus error exception is taken. |
| *IS_UnlockAck* | I | Valid in the cycle terminating the transaction (*IS_Stall* deasserted). Result of *IS_Unlock* operation. Should be asserted high if system holds a lock on the address used for the redirected write transaction (SC). |
| *IS_RData[31:0]* | I | Read data. |
| *IS_RBE[3:0]* | I | Byte enable signals for *IS_RData[31:0]*.<br>*IS_RBE[3]* enables byte lane corresponding to *IS_RData[31:24]*.<br>*IS_RBE[2]* enables byte lane corresponding to *IS_RData[23:16]*.<br>*IS_RBE[1]* enables byte lane corresponding to *IS_RData[15:8]*.<br>*IS_RBE[0]* enables byte lane corresponding to *IS_RData[7:0]*. |
| *DS_Read* | O | Read strobe. |
| *DS_Write* | O | Write strobe. |
| *DS_Sync* | O | Sync strobe. |
| *DS_WbCtl* | O | Write buffer control.<br>This signal is asserted when the M4K core can guarantee that no D-side read transaction will be started in the current clock cycle. For the purpose of generating this signal, if there is a pending transaction, the M4K core assumes that it will end in this cycle, in order to determine whether a new read transaction might be started or not.<br>Unlike *DS_Read*, there is no asynchronous path from *DS_Stall* or any other input signal to *DS_WbCtl.* Also, it is an earlier signal than *DS_Read*.<br>It is intended to be used by an external agent to control flushing of a write buffer (if a write buffer is present). |
| *DS_Addr[31:2]* | O | Address of transaction. When *DS_Sync* is asserted high, *DS_Addr[10:6]* holds the “sync type” (the “stype” field of the SYNC instruction). |
| *DS_BE[3:0]* | O | Byte enable signals for transaction.<br>*DS_BE[3]* enables byte lane corresponding to bits *31:24*.<br>*DS_BE[2]* enables byte lane corresponding to bits *23:16*.<br>*DS_BE[1]* enables byte lane corresponding to bits *15:8*.<br>*DS_BE[0]* enables byte lane corresponding to bits *7:0*. |
| *DS_CCA[2:0]* | Out | Provides the cache coherence attribute (CCA) for the current data request. The core will limit this value to either 2 (uncacheable) or 3 (cacheable) |

**Table 12 Signal Descriptions (Continued)**

| Signal Name | Type | Description |
|---|---|---|
| *DS_WData[31:0]* | O | Write data as defined by *DS_BE[3:0]*/*IS_BE[3:0]*. Used for both D-side and I-side transactions. |
| *DS_Abort* | O | Request for transaction (read, write or sync) to be aborted, if possible. It is optional whether the external logic uses this signal or not, although using it may reduce interrupt latency. Completion of any transaction (aborted or not) is always communicated through *DS_Stall*. Whether the transaction was in fact aborted is signalled using *DS_AbortAck*.<br>*DS_Abort* is asserted through (and including) the cycle where *DS_Stall* is deasserted. |
| *DS_EjtBreakEn* | O | One or more EJTAG data breakpoints are enabled. |
| *DS_EjtBreak* | O | Asserted when an EJTAG data break is detected. May be used by external logic to cancel the current transaction. This signal is asserted one cycle after the transaction start, so when precise breaks are required, the external logic must stall transactions by one cycle if *DS_EjtBreakEn* indicates that a break may occur.<br>*DS_EjtBreak* is asserted through (and including) the cycle where *DS_Stall* is deasserted. |
| *DS_Lock* | O | Asserted when a read transaction is due to an LL (load linked) instruction. |
| *DS_Unlock* | O | Asserted when a write transaction is due to an SC (store conditional) instruction. |
| *DS_Stall* | I | Indicates that the transaction is not ready to be completed. |
| *DS_Error* | I | Valid in the cycle terminating the transaction (*DS_Stall* deasserted). Asserted high if transaction caused an error. Causes bus error exception to be taken by the core. |
| *DS_AbortAck* | I | Valid in the cycle terminating the transaction (*DS_Stall* deasserted). Asserted high if transaction was aborted.<br>If no abort was requested (*DS_Abort* is low), and *DS_AbortAck* is asserted high in the cycle terminating the transaction, a bus error exception is taken. |
| *DS_Redir* | I | Valid in the cycle terminating the transaction (*DS_Stall* deasserted). Asserted high if transaction must be redirected to I-side. |
| *DS_UnlockAck* | I | Valid in the cycle terminating the transaction (*DS_Stall* deasserted). Result of *DS_Unlock* operation. Should be asserted high if system holds a lock on the address used for the write transaction (SC). |
| *DS_RData[31:0]* | I | Read data. |
| *DS_RBE[3:0]* | I | Byte enable signals for *DS_RData[31:0]*.<br>*DS_RBE[3]* enables byte lane corresponding to *DS_RData[31:24]*.<br>*DS_RBE[2]* enables byte lane corresponding to *DS_RData[23:16]*.<br>*DS_RBE[1]* enables byte lane corresponding to *DS_RData[15:8]*.<br>*DS_RBE[0]* enables byte lane corresponding to *DS_RData[7:0]*. |
| **CorExtend® User-Defined Instruction Interface** | | |
| On the M4K Pro core, an interface to user-defined instruction block is possible. See *MIPS32® Pro Series® CorExtend® Instruction Integrator's Guide* for a description of this interface. | | |
| **Coprocessor Interface** | | |
| Instruction dispatch: These signals are used to transfer an instruction from the M4K core to the COP2 coprocessor. | | |
| *CP2_ir_0[31:0]* | O | Coprocessor Arithmetic and To/From Instruction Word.<br>Valid in the cycle before *CP2_as_0*, *CP2_ts_0* or *CP2_fs_0* is asserted. |

**Table 12 Signal Descriptions (Continued)**

| Signal Name | Type | Description |
|---|---|---|
| *CP2_irenable_0* | O | 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*.<br>**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]*. |
| *CP2_as_0* | O | 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. |
| *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.<br>**Note**: The M4K 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 M4K 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 12 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.<br><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 M4K 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].<br>Note: The M4K 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 M4K 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.<br><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 12 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].<br>Note: The M4K 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 M4K 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 M4K 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 M4K core is signalling that the instruction is not nullified. When asserted, the M4K 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 M4K 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 12 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.<br>(not due to CP2_exc_0)</td></tr>
<tr><td>$11_2$</td><td>Instruction is killed.<br>(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:<br>1: The TDO output at chip level must be in Z-state<br>0: The TDO output at chip level must be driven to the value of EJ_TDO<br>IEEE Standard 1149.1-1990 defines TDO as a 3-stated signal. To avoid having a 3-state core output, the M4K 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>
<tr><td colspan="3">Debug Mode Indication:</td></tr>
<tr><td>EJ_DebugM</td><td>O</td><td>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.</td></tr>
</table>

**Table 12 Signal Descriptions (Continued)**

| Signal Name | Type | Description |
|---|---|---|
| *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/`<br>*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. |
| *TCtrace Interface* | | |
| This interface behaves differently depending on which trace mechanism is present. The MIPS Trace interface will be described first and the iFlowtrace interface is described in the next section<br>These signals enable an interface to optional off-chip trace memory. The TCtrace interface connects to the Probe Interface Block (PIB) which in turn connects to the physical off-chip trace pins.<br>Note that if MIPS Trace with on-chip trace memory is used, access occurs via the EJTAG TAP interface, and this interface is not required. | | |
| *TC_ClockRatio[2:0]* | O | 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. |

| TC_ClockRatio | Clock Ratio |
|---|---|
| 000 | 8:1 (Trace clock is eight times the core clock) |
| 001 | 4:1 (Trace clock is four times the core clock) |
| 010 | 2:1 (Trace clock is double the core clock) |
| 011 | 1:1 (Trace clock is same as the core clock) |
| 100 | 1:2 (Trace clock is one half the core clock) |
| 101 | 1:4 (Trace clock is one fourth the core clock) |
| 110 | 1:6 (Trace clock is one sixth the core clock) |
| 111 | 1:8 (Trace clock is one eight the core clock) |

**Table 12 Signal Descriptions (Continued)**

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></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.<br>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.<br>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>
<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”.<br>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.<br>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.<br>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.<br>TC_Valid is only asserted when TC_DataBits is 100.</td></tr>
</table>

**Table 12 Signal Descriptions (Continued)**

| Signal Name | Type | Description |
|---|---|---|
| *TC_Stall* | I | 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.<br>*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*. |
| *TC_Data[63:0]* | O | 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.<br>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. |
| *TC_ProbeTrigIn* | A | 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. |
| *TC_ProbeTrigOut* | O | 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. |
| *TC_ChipTrigIn* | A | Rising edge trigger input. The source should be on-chip. The input is considered asynchronous; i.e., it is double registered in the core. |
| *TC_ChipTrigOut* | O | Single cycle (relative to core clock) high strobe, trigger output. The target of this trigger is intended to be an on-chip unit. |
| With the iFlowtrace mechanism, only a subset of the TCtrace interface is used. The following signals are active. Other inputs can be tied off to a fixed value. | | |
| *TC_ClockRatio[2:0]* | O | Clock ratio.This is the clock ratio set by software in *ITCBCtl.OfClk*. The table below shows the encoded values for clock ratio. |
| *TC_Valid* | O | Asserted when a valid new trace word is started on the *TC_Data[63:0]* signals. |
| *TC_Stall* | I | 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_Data[63:0]* | O | Trace word data. Unlike with MIPS Trace, the PIB is responsible for extracting the appropriate data bits from the bus |
| **Scan Test Interface** | | |
| These signals provide an interface for testing the core. The use and configuration of these pins are implementation-dependent. | | |
| *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. |
| *gscanin_X* | I | These signal(s) are the inputs to the scan chain(s). |

| TC_ClockRatio | Clock Ratio |
|---|---|
| 100 | 1:2 (Trace clock is one half the core clock) |
| 101 | 1:4 (Trace clock is one fourth the core clock) |
| Others | Reserved |

**Table 12 Signal Descriptions (Continued)**

| Signal Name | Type | Description |
|---|---|---|
| *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. |

# SRAM-style Interface Transactions

Waveforms illustrating various transactions are shown in the following subsections. The type of transaction is always indicated through assertion of one of the three mutually-exclusive strobe signals:

- *DS_Read*, *DS_Write*, or *DS_Sync* on the D-side
- *IS_Read*, *IS_Write*, or *IS_Sync* on the I-side

Most figures assume that a dual I/D interface is present, and show D-side transactions (in some cases redirected to I-side). However, I-side (and thus Unified Interface) transactions work the same way, except there is no I- to D-side redirection mechanism.

Unless stated otherwise, I-side waveforms assume that 32 bit MIPS32 instruction fetches are being continuously performed.

## Simple Reads and Writes

This section describes several basic read and write transactions.

### Single Read

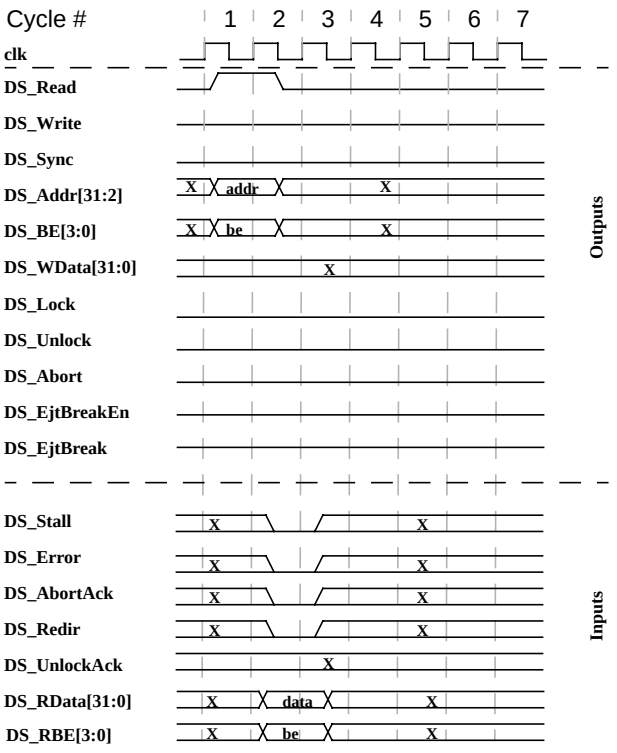
Figure 7 illustrates the fastest read, a single cycle D-side read operation. The transaction is initiated by the core in cycle 1, as it asserts the read strobe (*DS_Read*), as well as the desired word address (*DS_Addr[31:2]*) and output byte enables (*DS_BE[3:0]*). The byte enables represent the lower two bits of the address, as well as the requested data size, and identify which of the four byte lanes on *DS_RData* in which the core expects the read data to be returned.

The external agent is able to process the read immediately, so it deasserts stall while returning the appropriate read data (*DS_RData[31:0]*) and the input byte enables (*DS_RBE[3:0]*) in the following clock, cycle 2, and the transaction completes successfully. The input byte enables control sampling of the corresponding byte lanes for *DS_RData*, and must be asserted appropriately. There is no explicit hardware check that the input byte enables actually corresponded to the requested output byte enables. If some of the necessary input byte enables are not asserted, the core will (probably erroneously) just use the last read data held in the input registers for those byte lanes.

The interface protocol does not include an explicit “read acknowledge” strobe; for simplicity, the transaction is identified to be complete solely by the first cycle following a read strobe in which stall (*DS_Stall*) is deasserted. Other signals (*DS_Error*, *DS_Redir*, *DS_AbortAck*, *DS_UnlockAck*) indicate the status of a transaction, but the completion itself is identified only through the deassertion of *DS_Stall*; the status signals are ignored by the core when *DS_Stall* is asserted.

In a typical system, the read data is returned from an SRAM device that is accessed synchronously on the rising edge of cycle 2, with the address and strobe information provided by the core in cycle 1. The read data can be returned by any device that meets the protocol timing, such as ROM, flash, or memory-mapped registers.

**Figure 7 Single Cycle Read**

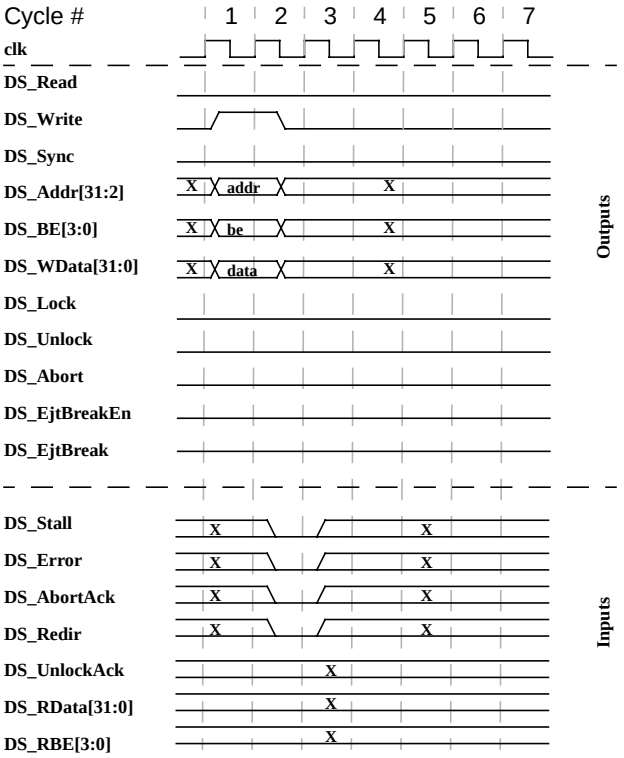


### Single Write

Figure 8 illustrates the fastest write, a single cycle D-side write operation. The transaction is initiated by the core in cycle 1, as it asserts the write strobe (*DS_Write*), as well as the desired word address (*DS_Addr[31:2]*), write data (*DS_WData[31:0]*), and output byte enables (*DS_BE[3:0]*). The byte enables identify which of the four byte lanes in *DS_WData* hold valid write data.

The external agent is able to successfully acknowledge the write immediately, so it deasserts stall (*DS_Stall*) in the following clock, cycle 2, to complete the write. Note that the interface protocol does not include an explicit "write acknowledge" strobe; the transaction is identified to be complete simply by the deassertion of stall.

**Figure 8 Single Cycle Write**




### Read with Waitstate

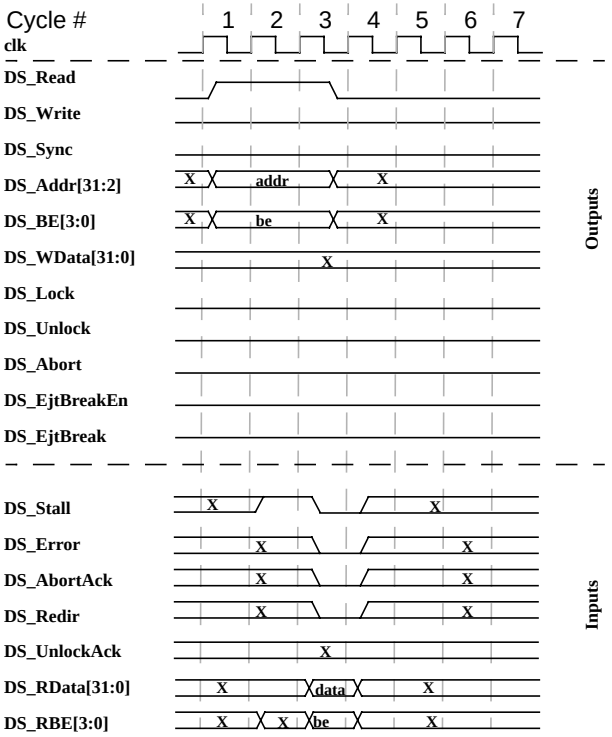
Figure 9 illustrates a D-side read operation with a single waitstate. This transaction is similar to the single-cycle read in Figure 7, only now a stall (*DS_Stall*) is asserted for one cycle and the read data is returned a cycle later.

The transaction is initiated by the core in cycle 1, as it asserts the read strobe (*DS_Read*), as well as the desired word address (*DS_Addr[31:2]*) and output byte enables (*DS_BE[3:0]*).

The external agent is not ready to complete the read immediately, so it asserts *DS_Stall* in cycle 2. Note that during a stall, the core holds the read strobe, address and output byte enables valid, and ignores values driven on the input status signals (*DS_Error*, *DS_Redir*, *DS_AbortAck*).

In cycle 3, the read data becomes available, so the external agent deasserts *DS_Stall* and returns the appropriate read data (*DS_RData[31:0]*) and the input byte enables (*DS_RBE[3:0]*). In this example, no error or redirection is signaled, so the transaction completes successfully in cycle 3.

**Figure 9 Read with One Waitstate**




### Write with Waitstate

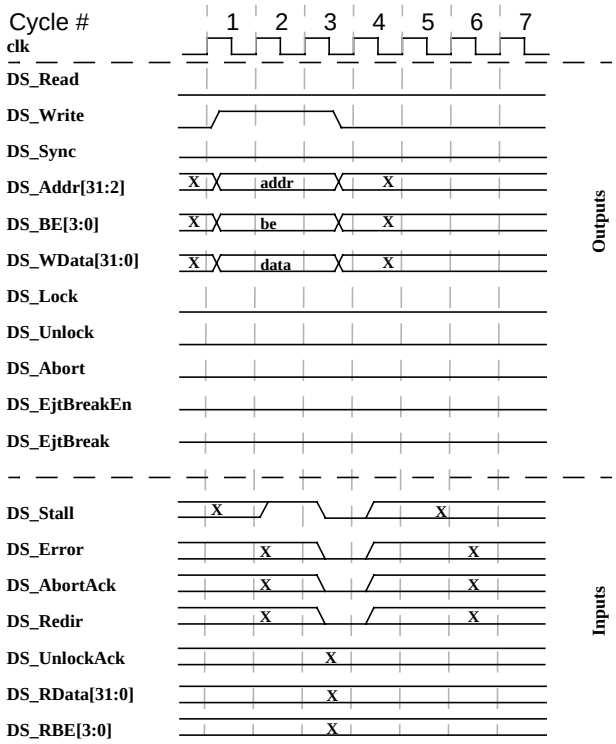
Figure 10 illustrates a D-side write operation with a single waitstate. This transaction is similar to the single-cycle write in Figure 8, only now a stall (*DS_Stall*) is asserted for one cycle and the write is completed a cycle later.

The transaction is initiated by the core in cycle 1, as it asserts the write strobe (*DS_Write*), as well as the desired word address (*DS_Addr[31:2]*), write data (*DS_WData[31:0]*), and output byte enables (*DS_BE[3:0]*).

The external agent cannot acknowledge the write immediately for some reason, so it asserts *DS_Stall* in cycle 2. The core outputs are held valid through the stall. Finally in cycle 3, the write can be accepted, so *DS_Stall* deasserts, and

the error and redirection signals also deassert to indicate a normal completion.

**Figure 10 Write with One Waitstate**



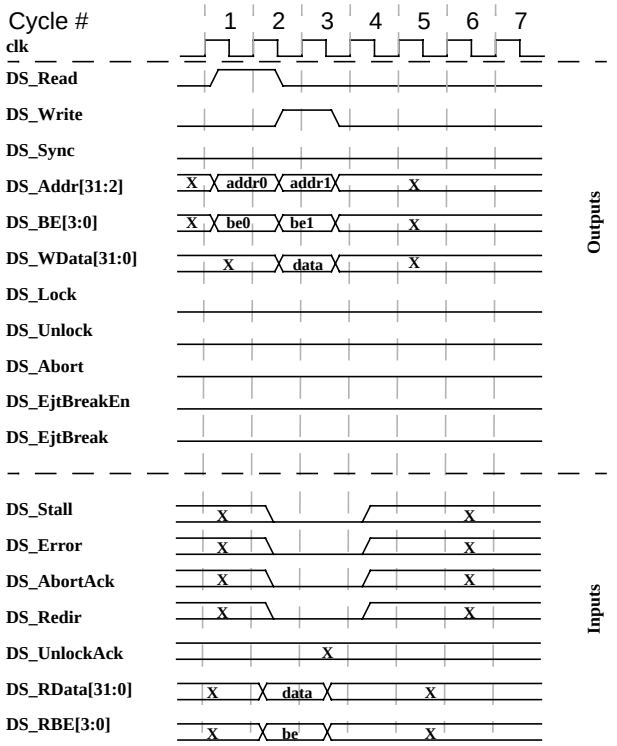

## Read Followed by Write

Figure 11 illustrates a single cycle D-side read operation followed immediately by a single cycle D-side write operation. This example represents the back-to-back concatenation of the single-cycle read shown in Figure 7 with the single cycle write from Figure 8.

The read is initiated in cycle 1, with the core's assertion of the read strobe, read address, and read output byte enables. The external agent is able to fulfill the read request in cycle 2, so it deasserts stall and drives the read data and input byte enables in cycle 2.

Since there is no stall from the read in cycle 2, the core is immediately able to initiate another transaction in the same cycle (if it has one pending), this time a write. Note that the SRAM-style interface logic contains a combinational path from *DS_Stall* to the start of a new transaction, for maximum performance. The external agent can accept the write, so no stall is asserted in cycle 3 and the write finishes.

**Figure 11 Read Followed by Write (Single Cycle)**



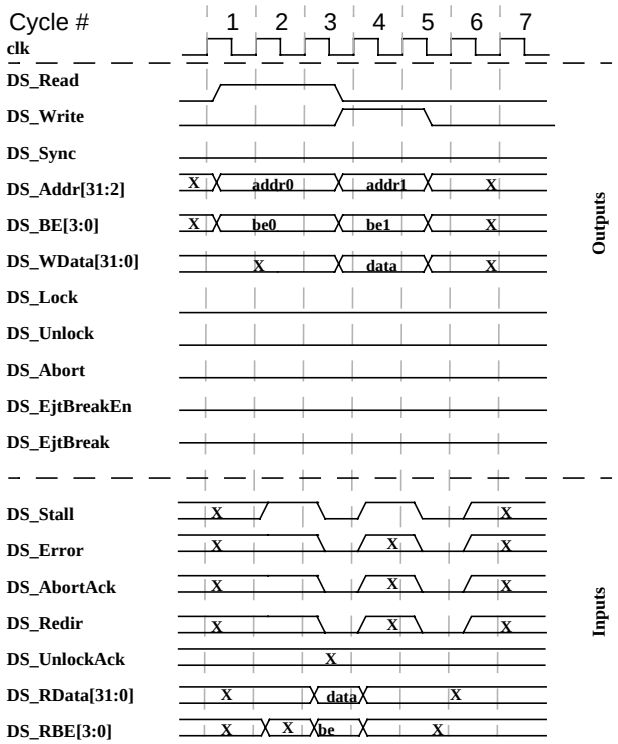

## Read Followed by Write, with Waitstates

Figure 12 illustrates a one waitstate D-side read operation followed immediately by a one waitstate D-side write operation. This example is similar to the back-to-back read/write case in Figure 11, only now each of the two transactions includes one waitstate.

The read is initiated in cycle 1, with the core's assertion of the read strobe, read address, and read output byte enables. The external agent cannot complete the read immediately, so it asserts stall in cycle 2. This forces the core to hold its read-related outputs for another cycle, and precludes the core from starting a new transaction. In cycle 3, stall deasserts and the read data and input byte enables are driven valid, completing the read.

The stall deassertion in cycle 3 allows the core to start its next pending transaction, this time a write. The external agent is not ready to accept the write, so it asserts stall again in cycle 4. Finally in cycle 5, the write can complete, so stall deasserts and the write finishes.

**Figure 12 Read Followed by Write (One Waitstate)**




## MIPS16e™ Instruction Fetches

Most instruction fetches are performed as a full word read (32 bits) on the I-side interface, so all bits of *IS_BE[3:0]* are usually asserted. Even in MIPS16e mode, where 16-bit instructions are executed, most fetches are still performed as full word fetches in order to optimize the I-side bandwidth. The core holds the full word in an internal buffer, and therefore usually only needs to perform a fetch when executing every other MIPS16e instruction. When a jump or branch occurs to the middle of a word in MIPS16e mode, however, the core will perform a halfword (16-bit) fetch.

Figure 13 illustrates instruction fetches when executing in MIPS16e mode, assuming no waitstates.

A word-aligned fetch at addr0 is requested in cycle 1. This causes a 32 bit word (for example, containing two non-extended MIPS16e instructions, “instr0” and “instr1”) to be fetched (the current as well as the following instruction).

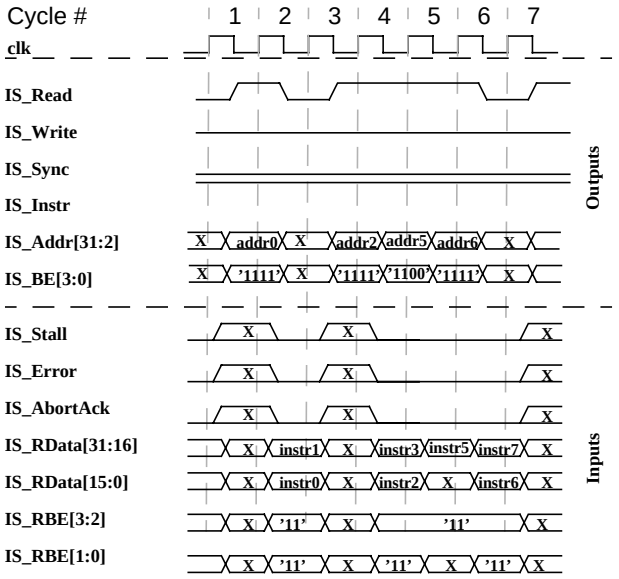
This example assumes that the code is executed sequentially up to this point, so no read is necessary for the next instruction (i.e. no read request in cycle 2). The example assumes that “instr1” is a jump to a non word aligned address (addr5).

In cycle 3, a word-aligned fetch from addr2 is requested. Again, a full instruction word is fetched, but in this case it is assumed that only one 16 bit instruction is used (“instr2”, which is the jump delay slot of “instr1”).

In cycle 4, a fetch occurs for the instruction at the jump target address (addr5). The figure illustrates the case where addr5 is not word aligned, so only 16 bits (“instr5”) are read. Endianness is assumed to be little, so *IS_BE[3:0]* = “1100”. In the big endian case, *IS_BE[3:0]* would have been “0011”.

In cycle 5, a full word fetch occurs for the following 2 instructions after the jump target, stored at addr6.

**Figure 13 MIPS16e™ Instruction Fetches (Single Cycle, Little Endian Mode)**




## Redirection

When dual I and D interfaces are present, it is possible to redirect a D-side operation to the I-side for completion. This mechanism might be useful if the system wants to read data that is stored in an I-side device, or to initialize an I-side SRAM with data store instructions that would normally be presented to the D-side. There is no mechanism to redirect I-side references to the D-side. Also, the PC-relative load instructions present in the MIPS16e ASE use an internal method within the core to present loads to the I-side, and therefore do not use the explicit external redirection mechanism.

When a D-side transaction has been redirected to the I-side, the core will never initiate a new D-side transaction until the redirected one has completed on the I-side.

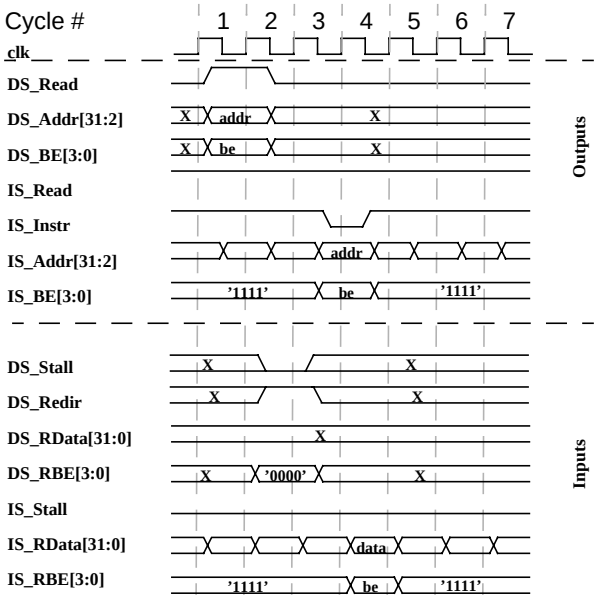
Several examples of D-side operations redirected to I-side are illustrated. The examples assume that the redirected D-side transaction immediately gets access to the I-side external interface. This is the typical case since redirected D-side accesses have priority over I-side instruction fetches.

### Redirected Read, Single-Cycle

Figure 14 illustrates a single-cycle D-side read operation where *DS_Redir* is used for requesting the operation to be redirected to the I-side. In this example, the I-side read operation is also single cycle.

The data read begins in cycle 1, like the simple read introduced in Figure 7. The external agent decides that the read must be handled by the I-side array, so it deasserts *DS_Stall* while asserting *DS_Redir* in cycle 2. The D-side transaction is thus terminated, but with the status that it must be redirected to the I-side for completion. The I-side is able to start the request immediately, so the read strobe (*IS_Read*), address (*IS_Addr[31:2]*) and byte enables (*IS_BE[3:0]*) from the original data read request are driven in cycle 3. Note that *IS_Instr* is deasserted in cycle 3. The external agent returns the requested read data (*IS_RData*[31:0]) and input byte enables (*IS_RBE[3:0]*) in cycle 4, and the redirected transaction completes since there is no stall.

**Figure 14 Redirected Read (Single Cycle)**

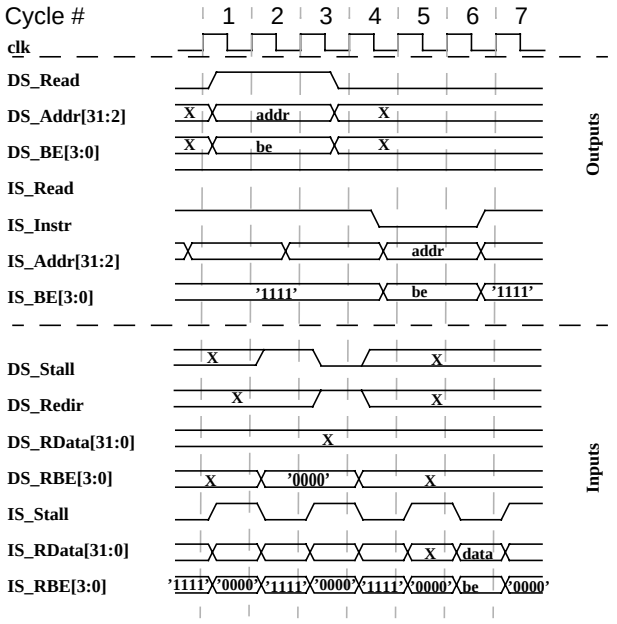



### Redirected Read with Waitstate

Figure 15 illustrates a one waitstate D-side read operation where *DS_Redir* is used for requesting the operation to be redirected to the I-side. In this example, the I-side read operation also has one waitstate.

The data read again begins in cycle 1. The external agent decides to stall the core for one cycle starting in cycle 2, by asserting *DS_Stall*. Then in cycle 3, the agent decides to redirect the data read request to the I-side. In cycle 4, the core drives the original data read signals on the I-side interface. The I-side is not available for some reason, so the external agent asserts *IS_Stall* in cycle 5, causing the core to hold its strobe, address, and byte enables valid for another cycle. Finally in cycle 6, the agent deasserts stall, returns the requested read data, and the transaction completes.

**Figure 15 Redirected Read (One Waitstate)**

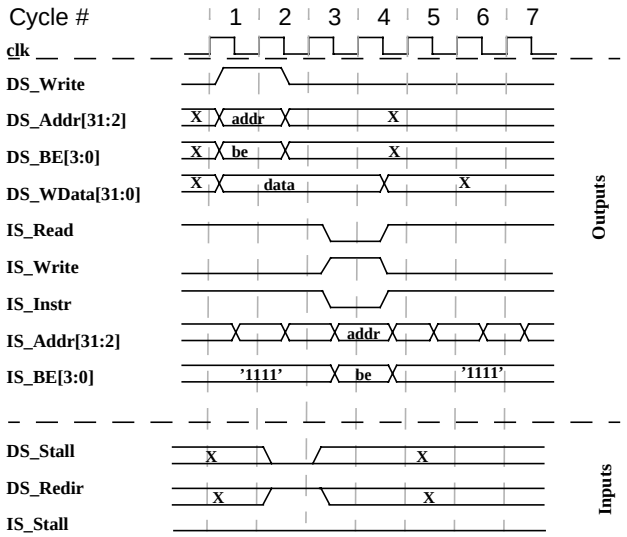



### Redirected Write, Single-Cycle

Figure 16 illustrates a single cycle D-side write operation where *DS_Redir* is used for requesting the operation to be redirected to I-side. In this example, the I-side write operation is also single cycle. Writes redirected to the I-side might be used as a method for initializing the instruction code space, as writes to instruction memory are not otherwise possible from the core.

The D-side write initiated in cycle 1 is requested for redirection in cycle 2. In cycle 3, the core drives the I-side write strobe, address, byte enables, and data. A redirected write is the only way that the *IS_Write* strobe is asserted. There is no write data bus on the I-side, so the write data continues to be held on the *DS_WData[31:0]* bus. The external agent can accept the data immediately, so the transaction completes in cycle 4 since there is no stall.

**Figure 16 Redirected Write (Single Cycle)**

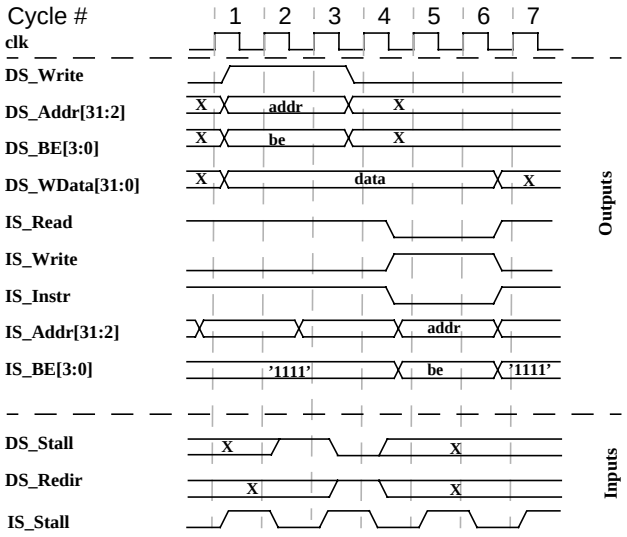



### Redirected Write with Waitstate

Figure 17 illustrates a one waitstate D-side write operation where *DS_Redir* is used for requesting the operation to be redirected to I-side. In this example, the I-side write operation also has one waitstate.

The sequence shown in Figure 17 is similar to the single cycle write redirection in Figure 16, only this time one waitstate is asserted on the D-side before the redirection is signaled, and then another waitstate is signaled on the I-side before the write is accepted.

**Figure 17 Redirected Write (One Waitstate)**




## Data Gathering

The SRAM interface includes a “data gathering” capability that uses input byte enable signals, *DS_RBE[3:0],* to control input data registers and allow the read data to be registered within the core as it becomes available. The same mechanism is available for the I-side, using *IS_RBE[3:0]*.

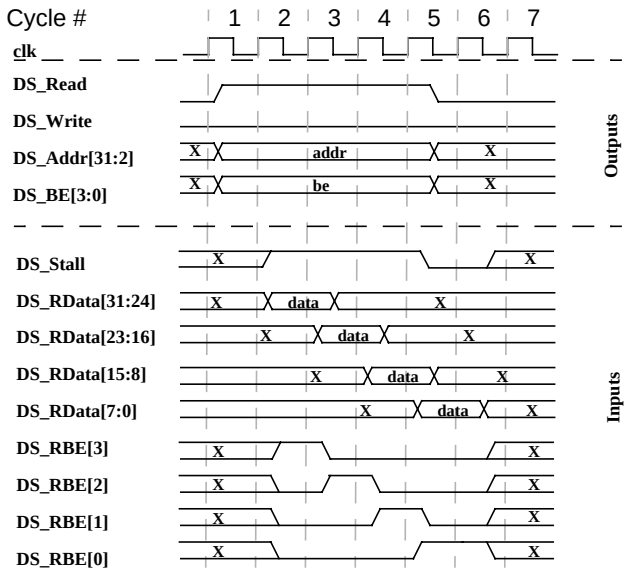
As the core contains 32-bit interfaces for read data, the gathering capability enables the connection to narrower memories with minimal logic external to the core. Read data must be aligned to the appropriate byte lane by external logic, but the input byte enables remove the need for external flops to hold partial read data while it is collected.

The gathering capability is illustrated in Figure 18. The data read is initiated by the core in cycle 1, as normal. In this example, the requested read data is 32 bits wide, but it will be returned one byte at a time. The external agent asserts *DS_Stall* for 3 clocks, starting in cycle 2. In cycles 2-4, a single byte of read data is returned each clock, as indicated by the input byte enables (*DS_RBE[3:0]*), while stall remains asserted. Finally in cycle 5, stall is deasserted and the final byte is returned, completing the read transaction.

The input byte enables, *DS_RBE[3:0]*, simply act as enables on the conditional flops that capture the read data bus, *DS_RData[31:0]*. The core does not perform any explicit checking to ensure that the requested bytes, as indicated by *DS_BE[3:0]*, were actually returned, as indicated by *DS_RBE[3:0]*. It is up to the external agent to ensure that the appropriate read data is actually returned. If the necessary input byte enables were not asserted before the transaction completes, the core will use the last data held by the byte-wide input flops, which will probably not be the desired behavior.

While stall is asserted, minimal system power will usually be achieved when the valid data byte is strobed only once via the appropriate *DS_RBE* signal. However, the core input flops will be overwritten each cycle that a *DS_RBE* bit is asserted, while the transaction is still active.

**Figure 18 Word Read, Data Arriving Bytewise**




## Sync

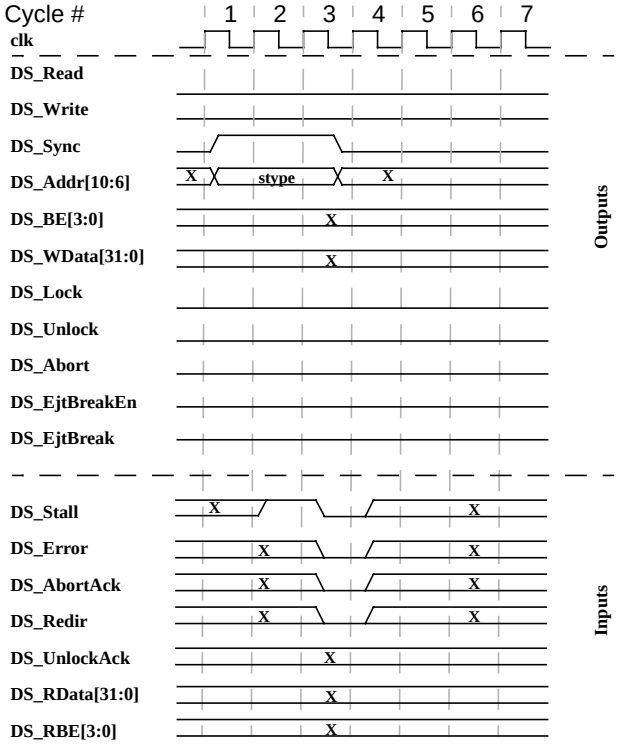
This section illustrates several examples of the protocol associated with the execution of a SYNC instruction. An external indication of SYNC execution is provided to allow external agents to order memory operations, if desired.

### Sync with Waitstate

Figure 19 illustrates D-side sync signaling for flushing external write buffers. One waitstate is assumed in this example.

The sync signaling is initiated in cycle 1, as indicated by the sync strobe, *DS_Sync*. The 5-bit “stype” field encoded within the SYNC instruction is provided on the address bus, *DS_Addr[10:6]*. The location of the stype field on the address bus matches its field position within the SYNC instruction word. A sync transaction is terminated just like a normal read, in the first non-stall cycle after the sync strobe. If an external agent wants to flush external write buffers, or allow other pending memory traffic to propagate through the system, it can stall acknowledgment of the sync by asserting the normal stall signal, *DS_Stall*. In this example, one such stall cycle is shown, starting in cycle 2. Then in cycle 3, stall deasserts and the sync transaction is terminated. In a sync transaction, no read data is returned, so the values on the *DS_RData* and *DS_RBE* signals are ignored by the core.

**Figure 19 Sync (One Waitstate)**



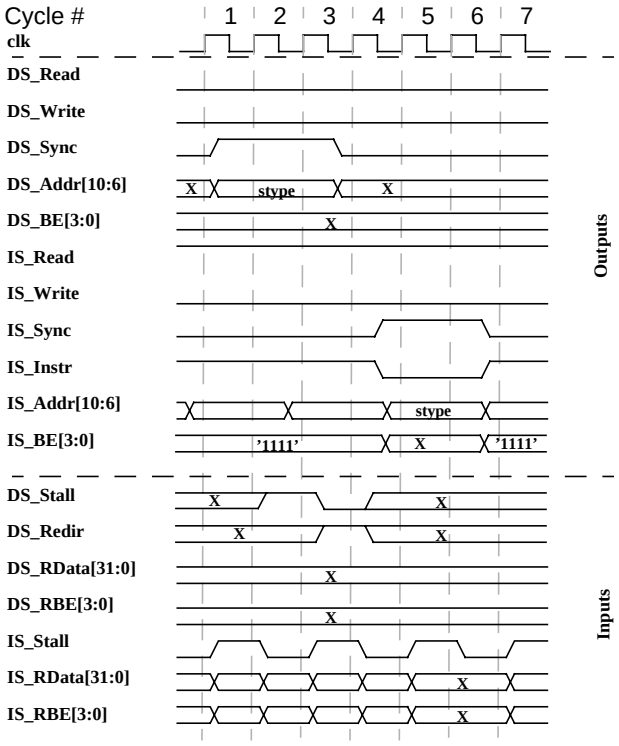

### Redirected Sync

Figure 20 illustrates sync signaling where the sync operation is requested to be redirected to I-side in order to flush I-side external write buffers. One waitstate for both D- and I-side is assumed in this example.

Usually, memory ordering around D-side transactions is desired, so the sync would only take effect on the D-side. But the sync transaction, much like a read, can also be redirected to the I-side, if desired.

In this example, the sync is initiated on the D-side in cycle 1. The external agent responds with a stall in cycle 2, then a redirection request to the I-side in cycle 3. In cycle 4, the core drives the I-side strobe (*IS_Sync*) and stype information on the address bus (*IS_Addr[10:6]*). Note that *IS_Instr* also deasserts in cycle 4, to indicate that the I-side transaction is not due to an instruction fetch. The external agent cannot acknowledge the sync immediately, so it asserts stall in cycle 5. Finally in cycle 6, the stall deasserts and the redirected sync transaction is completed.

**Figure 20 Redirected Sync (One Waitstate)**




## Bus Error

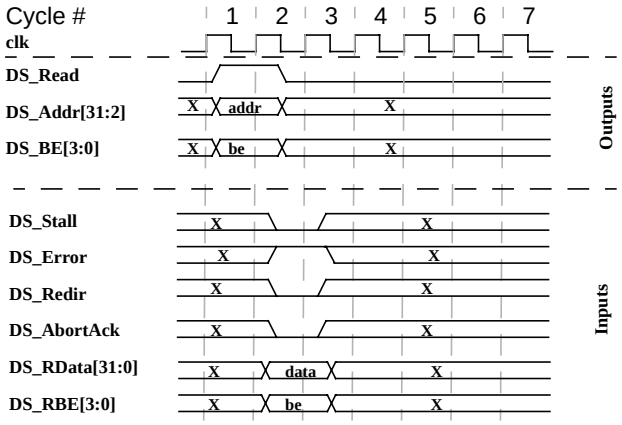
Examples of the error protocol are shown in this section. An error is indicated through the *DS_Error* or *IS_Error* pins, and ultimately results in a precise data or instruction bus error exception within the core. The assertion of *DS_Error* will always result in a data bus error exception. The assertion of *IS_Error* will result in an instruction bus error exception if the transaction is a fetch, or a data bus error exception if the transaction is a data request (redirected or unified interface).

### Bus Error on Single Cycle Read

Figure 21 illustrates a single-cycle D-side read operation causing a bus error, signalled via *DS_Error*.

The read is initiated in cycle 1, as normal. This time, the external agent has identified an error condition for some reason, so it responds by deasserting *DS_Stall* while asserting *DS_Error* in cycle 2. This terminates the read transaction on the bus with an error status. Any values returned on the *DS_RData* and *DS_RBE* buses will be captured by the input data registers, but are otherwise ignored by the core. The termination of a read transaction with *DS_Error* will result in a data bus error exception within the core.

**Figure 21 Read with Error Indication (Single Cycle)**

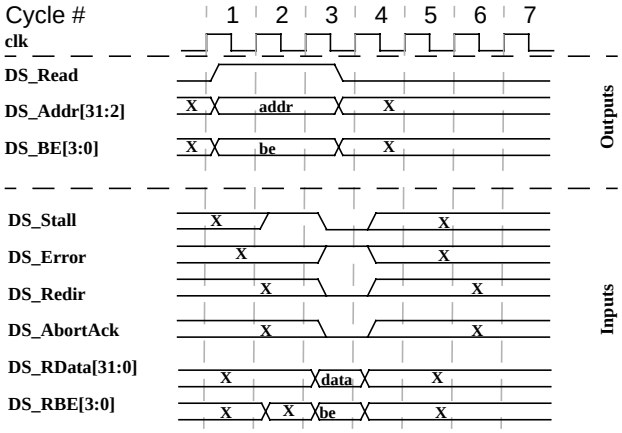



### Bus Error on Read with Waitstate

Figure 22 illustrates a one waitstate D-side read operation causing a bus error.

Again, the read transaction begins normally in cycle 1. A stall is asserted in cycle 2. Finally in cycle 3, the external agent has identified an error condition so it deasserts stall and terminates the read transaction with error status, via the assertion of *DS_Error*. The value of *DS_Error*, as well as any other core input for that matter, is ignored by the core whenever *DS_Stall* is asserted.

**Figure 22 Read with Error Indication (One Waitstate)**




## Abort

Due to the nature of the core pipeline, it may sometimes be desirable to abort a transaction on the SRAM-style interface before it completes.

Normally, interrupts are taken on the E-M boundary of the pipeline. Since a D-side interface transaction occurs during the M-stage, a pending interrupt must wait for the outstanding transaction to complete. If this transaction has multiple waitstates, interrupt latency will be degraded. To improve interrupt latency, a mechanism exists on the SRAM interface that allows an outstanding transaction to be aborted. Generally, a transaction must have at least one waitstate or it doesn't make sense to abort it.

Use of the abort mechanism is optional. If a load/store/sync transaction is successfully aborted following an interrupt, then the interrupt will be taken on the load/store/sync instruction that initiated the transaction. In this case, care must be taken to ensure that the aborted transaction can be replayed with no ill effects in the system. If the transaction is not aborted, then the interrupt is simply taken on the instruction following the load/store/sync.

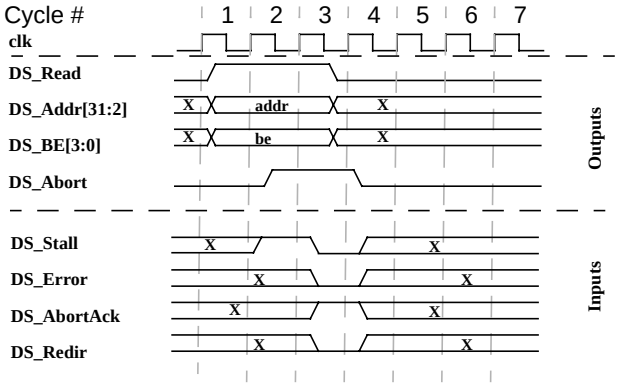
Examples of aborted transactions are discussed in the following subsections.

### Aborted Read

Figure 23 illustrates a one waitstate D-side read operation with an abort request. In this example, external logic was able to abort the operation, and signals the acknowledgment through assertion of *DS_AbortAck*.

The read begins normally in cycle 1, due to a load instruction. An interrupt is pending, so the core signals an abort request, by asserting *DS_Abort* in cycle 2. Whether the external agent responds to the abort request is completely optional. Also in cycle 2, the external agent is not ready to complete the read, so it asserts stall. In cycle 3, the external agent decides to abort the pending read transaction, so it deasserts stall while asserting *DS_AbortAck* and the transaction is aborted. The interrupt will be taken on the load instruction. Depending on the interrupt handler, instruction flow will likely return to this load after processing the interrupt, and the aborted read transaction will be replayed.

**Figure 23 Aborted Read (One Waitstate)**

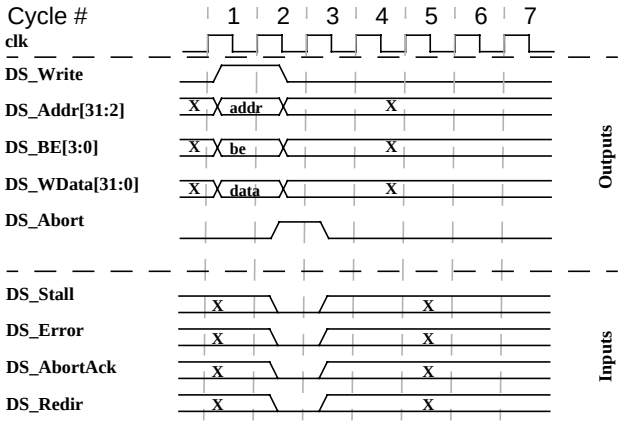



### Unsuccessful Abort for Single-Cycle Write

Figure 24 illustrates a single-cycle D-side write operation with an abort request. In this example, the external logic ignores the request and does not abort the operation.

The write is initiated in cycle 1. Due to a pending interrupt, the core signals an abort request in cycle 2. The external agent chooses not to abort the write, so it does not assert *DS_AbortAck*. The transaction completes normally in cycle 2, since no stall was asserted and the error, redirection and abort acknowledge status signals were deasserted.

**Figure 24 Unsuccessful Abort Attempt for Write (Single Cycle)**

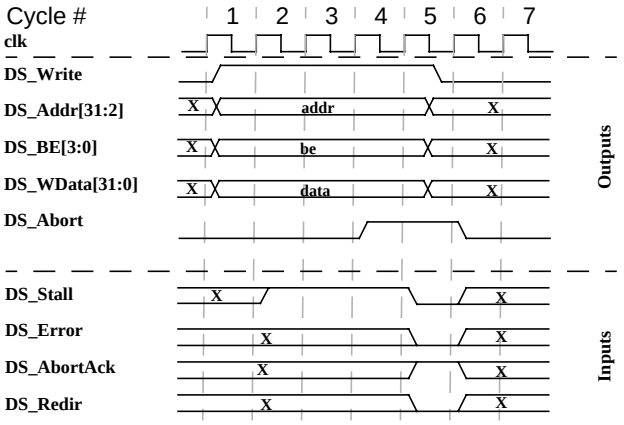



### Aborted Multi-Cycle Write

Figure 25 illustrates another case of a successfully aborted operation. This example demonstrates that the abort request can be signaled several cycles after the transaction has started.

This time, a write request is initiated in cycle 1. The external agent is not ready to complete the write, so it asserts stall in cycles 2 and 3. In cycle 4, an interrupt causes the core to signal an abort request. This causes the external agent to terminate the access in cycle 5 (deasserting *DS_Stall*), while asserting *DS_AbortAck* to indicate that the write was aborted.

**Figure 25 Aborted Write (Multi Cycle)**




## EJTAG Hardware Breakpoints

EJTAG hardware breakpoints present another twist on the SRAM-style interface. Hardware breakpoints are one method to achieve entry into EJTAG debug mode. When a breakpoint occurs, a debug exception must be taken on the instruction fetch, data load, or data store instruction itself, but the exception is not known until the transaction has already started on the interface. Hence, the breakpointed transaction may have accessed memory, but will be replayed after returning from the debug exception. If this transaction is not replay-able, it should not be allowed to access or modify memory until it is certain that no breakpoint will occur. At least one waitstate is necessary to identify a transaction that may potentially take an EJTAG breakpoint exception.

Note that no acknowledge is signalled as response to EJTAG break indications (*DS_EjtBreak* or *IS_EjtBreak*). The exception is always taken on the instruction fetch, data load, or data store instruction causing the break.

Also note that for a data read operation, a data break may depend on the data value read and so may be triggered after the read has finished. In case the read is followed by a new transaction, the new transaction may already have been initiated when the break is detected. In this case, the EJTAG break is signalled in the cycle following the cycle in which the read was terminated and the new access was initiated.

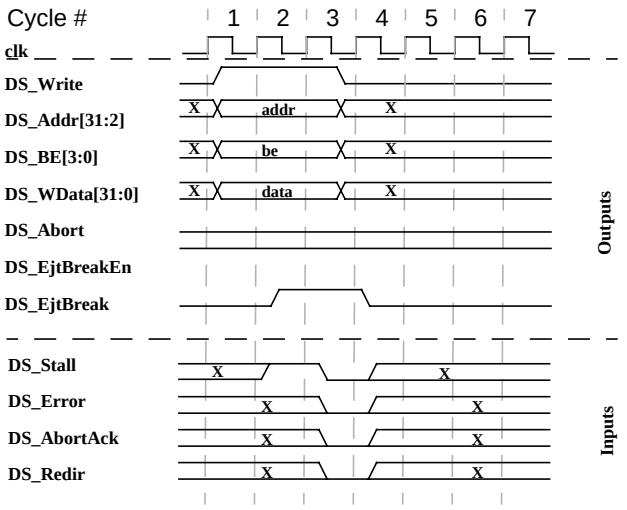
### EJTAG Break on Data Write

Figure 26 illustrates a one-waitstate D-side write operation causing an EJTAG data break. The EJTAG data break is signalled using *DS_EjtBreak*.

The write begins in cycle 1, as usual. *DS_EjtBreakEn* has been asserted for a while, indicating that EJTAG data breakpoints are enabled. The external agent can elect to use this signal to conditionally add waitstates, if replays cannot be tolerated when a breakpoint event ultimately occurs. In cycle 2, the core asserts *DS_EjtBreak* to indicate that a hardware breakpoint has been detected. Also in cycle 2, the external agent asserts a stall. Finally in cycle 3, the agent terminates the write transaction by deasserting *DS_Stall*. The core pipeline will take a debug exception on the store instruction that caused the write transaction, go into debug mode, and eventually upon exit from the debug handler will restart the store that caused the EJTAG break.

If the system cannot tolerate replay of the breakpointed transaction, then it should not allow the transaction to access memory. However, it must indicate a completion of the breakpointed transaction by deasserting stall; otherwise, the core will be stalled indefinitely.

**Figure 26 EJTAG Data Write Break (One Waitstate)**

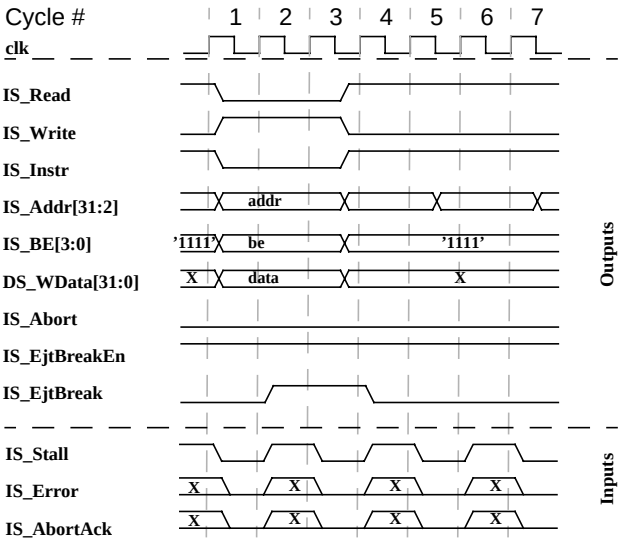



### EJTAG Break for Data Write, Unified Interface

Figure 27 illustrates a data write operation on the Unified Interface. The data write causes an EJTAG data break, which is signalled using *IS_EjtBreak*.

The data write begins in cycle 1. Note that the *IS_Write* strobe is asserted, while *IS_Read* and *IS_Instr* are deasserted, to indicate that a data write is occurring on the Unified Interface. *IS_EjtBreakEn* signal is asserted, since data breakpoints and/or instruction breakpoints, have been enabled. In cycle 2, the core detects a data breakpoint, and indicates it by asserting *IS_EjtBreak*. The external agent also stalls the write by asserting *IS_Stall* in cycle 2. Finally in cycle 3, the external agent terminates the transaction by deasserting *IS_Stall*. The external agent must signal the completion of the transaction in the normal manner (by deasserting stall). Again, the system is free to decide whether it actually allows the breakpointed write to update unified memory, according to its tolerance for replay.

**Figure 27 EJTAG Data Write Break for Unified Interface (One Waitstate)**




## Lock

Figure 28 illustrates the locking mechanism available to handle semaphores on the interface. This mechanism is used during the execution of D-side “load linked” / “store conditional” (LL/SC) operations.

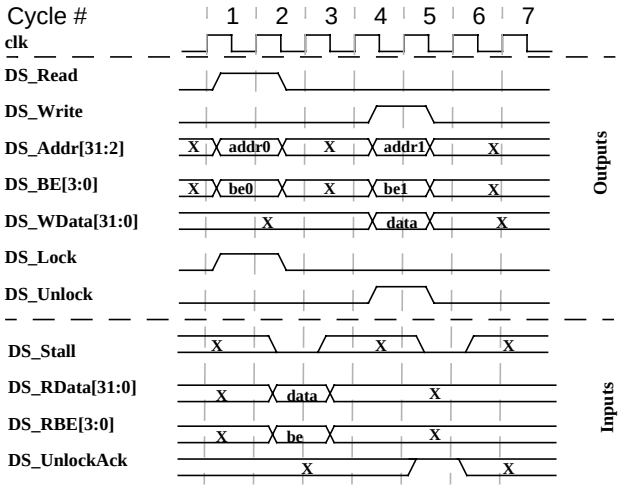
The data read resulting from an LL instruction is initiated in cycle 1. The LL is indicated by the core’s high-active assertion of the *DS_Lock* signal in cycle 1. External logic can use this information to attempt to set a lock on the requested address, and prevent other devices from accessing the address if the lock is obtained. The read completes in a single clock, in cycle 2. Then in cycle 4, the core starts a write resulting from an SC instruction, as indicated by its assertion of the *DS_Unlock* signal. The external agent can signal whether it was able to maintain the desired lock, by returning the status on *DS_UnlockAck*. The value returned on *DS_UnlockAck* is written by the core into the destination register specified by the SC instruction.

In this example, the read address from the LL (addr0) and the write address from the SC (addr1) are different. It is completely up to the external logic as to whether locks it maintains are address-specific or not.

While this example has assumed a data operation occuring on a the D-side of a Dual Interface, I-side signaling is used for redirected (or Unified Interface) LL/SC operations. I-side lock signaling works the same way as the D-side.

An additional signal, *IS_UnlockAll*, is related to the locking mechanism but not shown in Figure 28. *IS_UnlockAll* is asserted for one cycle whenever an ERET instruction is performed. This signal is only present on the I-side (and therefore the Unified Interface), and has no equivalent on the D-side. Whenever an ERET instruction is executed, *IS_UnlockAll* is asserted for one cycle. When this occurs, external logic can unlock all addresses locked by that CPU. An ERET is typically issued for each task-switch performed by the operating system.

**Figure 28 Locking (Single Cycle)**




# Revision History

Change bars (vertical lines) in the margins of this document indicate significant changes in the document since its last release. Change bars are removed for changes that are more than one revision old.

This document may refer to Architecture specifications (for example, instruction set descriptions and EJTAG register definitions), and change bars in these sections indicate changes since the previous version of the relevant Architecture document.

| Revision | Date | Description |
|---|---|---|
| 00.20 | May 8, 2002 | • Preliminary release. |

| Revision | Date | Description |
|---|---|---|
| 00.90 | June 27, 2002 | • Added more details about interrupt modes.<br>• Added external signals related to an optional external interrupt controller.<br>• Improved description of GPR shadow sets. |
| 01.00 | August 28, 2002 | • Commercial release.<br>• Added this revision history table.<br>• Changed K0, KU, and K23 fields in Config register to be read-only, with static value of 2.<br>• Modified abort description on SRAM interface, as abort requests are not only caused by interrupts.<br>• Updated description of write buffer control signals on SRAM interface. |
| 01.01 | January 8, 2003 | • Changed case of signal name SI_IAck. Added assembler idioms such as b, bal. |
| 01.02 | September 1, 2004 | • Externalized CorExtend interface.<br>• Added CEU (CorExtend Unusable) exception type.<br>• Exception table referred to EB_NMI instead of SI_NMI.<br>• Added table summarizing key build time configuration options. |
| 02.00 | June 5, 2006 | • Added complex breakpoints<br>• Added iFlowtrace interface<br>• Updated configuration options supported<br>• Added external call indication<br>• Enabled setting of cacheability |
| 02.01 | March 4, 2008 | • Update template<br>• Fixed SLTU description |

Template: nDb1.03, Built with tags: 2B