
# MIPS32 4K™ Processor Core Family Integrator's Guide

**Document Number: MD00036**
**Revision 1.07**
**December 4, 2000**

**MIPS Technologies, Inc.**
**1225 Charleston Road**
**Mountain View, CA 94043-1353**

Copyright © 1999-2000 MIPS Technologies, Inc. All rights reserved.

Unpublished rights reserved under the Copyright Laws of the United States of America.

This document contains information that is proprietary to MIPS Technologies, Inc. (“MIPS Technologies”). Any copying, modifyingor use of this information (in whole or in part) which is not expressly permitted in writing by MIPS Technologies or a contractually-authorized third party is strictly prohibited. At a minimum, this information is protected under unfair competition laws and the expression of the information contained herein is protected under federal copyright laws. Violations thereof may result in criminal penalties and fines.

MIPS Technologies or any contractually-authorized third party reserves the right to change the information contained in this document to improve function, design or otherwise. MIPS Technologies does not assume any liability arising out of the application or use of this information. Any license under patent rights or any other intellectual property rights owned by MIPS Technologies or third parties shall be conveyed by MIPS Technologies or any contractually-authorized third party in a separate license agreement between the parties.

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

MIPS, R3000, R4000, R5000, R8000 and R10000 are among the registered trademarks of MIPS Technologies, Inc., and R4300, R20K, MIPS16, MIPS32, MIPS64, MIPS-3D, MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MDMX, SmartMIPS, 4K, 4Kc, 4Km, 4Kp, 5K, 5Kc, 20K, 20Kc, EC, MGB, SOC-it, SEAD, YAMON, ATLAS, JALGO, CoreLV and MIPS-based are among the trademarks of MIPS Technologies, Inc.

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

# Table of Contents

# List of Figures

# List of Tables

# Overview

This document, the *MIPS32 4K™ Processor Core Family Integrator's Manual*, is targeted for the ASIC designer who is integrating a version of a MIPS32 4K processor core into his/her system ASIC. This document is applicable to both those integrators who are using a hard core and those who are incorporating a soft core.

Chapter 2, "Signal Description," on page 3 describes the pins of the core.

Chapter 3, "EC™ Interface," on page 9 describes the EC™ interface protocol used by the core.

Chapter 4, "EJTAG Interface," on page 25 discusses the EJTAG interface used by the core, including the EJTAG TAP controller.

Chapter 5, "Simulation Models," on page 33 describes models that can be used in place of the 4K core. These include the Bus Functional Model (BFM) and a cycle-accurate simulation model consisting of a model compiled with the Verilog Model Compiler tool (VMC™). The BFM is a fast model that can inject EC interface transactions into a system model to verify its compliance with the EC interface protocol. The VMC model provides a cycle-exact model of a 4K core that is used as a golden reference model in the customer verification environment for soft core licensees. It is also used by hard core integrators, and others who do not receive the RTL, to simulate with the 4K core.

Chapter 6, "Clocking, Reset & Power," on page 43 covers issues related to handling the clock insertion delay of the 4K core. Additionally, the hardware reset requirements of the core, as well as power management techniques, are discussed.

## 1.1 Environment Variable Setup

Some Unix paths described in the document refer to the JADEHOME environment variable, which should point to the top level of your 4K core deliverables. To set this variable:

```
% cd <release directory>
% setenv JADEHOME ‘pwd‘ # Note that these are back-ticks, not single quotes
```

# Signal Description

This chapter describes the signals on a MIPS32 4K™ processor core. Only naming convention and actual pin names are listed here. The specific interface protocol to which each pin adheres is described in subsequent chapters.

## 2.1 Naming Convention

The pin direction key for the signal descriptions is shown in Table 2-1 below.

**Table 2-1 Signal Type Key**

| Type | Description |
|---|---|
| I | Input to the core, unless otherwise noted, sampled on the rising edge of the appropriate clock signal. |
| O | Output of the core, unless otherwise noted, driven at the rising edge of the appropriate clock signal. |
| A | Asynchronous inputs that are synchronized by the core |
| S | Static Input to the core. These signals control configuration options and are normally tied to either power or ground. They should not change state while SI_ColdReset is deasserted. |

The names of interface signals present on a 4K core are prefixed with a unique string, according to their primary function. Table 2-2 defines the prefixes used for 4K core interface signals.

**Table 2-2 Signal Prefix Key**

| Prefix | Description |
|---|---|
| EB_ | Signals directly related to the EC interface. |
| SI_ | General system interface signals, which are not part of the EC interface. |
| EJ_ | Signals related to the EJTAG interface. |
| PM_ | Performance monitoring signals. |
| Scan/Bist | Signals related to design-for-test features, either scan or memory BIST. |

Generally, most signals have high-active assertion levels if not otherwise specified in the tables. Signals ending in the suffix "*_N*" are low active.

## 2.2 Signal Description

All core signals are listed in Table 2-3 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 2-3 Signal Descriptions**

| Signal Name | Type | Description |
|---|---|---|
| System Interface: Refer to Chapter 6, “Clocking, Reset & Power,” on page 43 for more details | | |
| Clock Signals: Refer to Section 6.1 , "Clocking" on page 43 for more details | | |
| 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 is intended to provide a reference for de-skewing any clock insertion delay created by the internal clock buffering in the core. |
| Reset Signals: Refer to Section 6.2 , "Reset and Hardware Initialization" on page 44 for a description of the various types of reset. | | |
| SI_ColdReset | A | Hard/Cold reset signal. Causes a Reset Exception in the core. |
| SI_NMI | A | Non-maskable Interrupt. An edge detect is used on this signal. When this signal is sampled asserted (high) one clock after being sampled deasserted, an NMI is posted to the core. |
| SI_Reset | A | Soft/Warm reset signal. Causes a SoftReset Exception in the core. |
| Power management signals: See Section 6.3 , "Power Management" on page 45 for more details | | |
| SI_ERL | O | This signal represents the state of the ERL bit (2) in the CP0 Status register and indicates the error level. The core asserts SI_ERL whenever a Reset, Soft Reset, or NMI exception is taken. |
| SI_EXL | O | This signal represents the state of the EXL bit (1) in the CP0 Status register and indicates the exception level. The core asserts SI_EXL whenever any exception other than a Reset, Soft Reset, NMI, or Debug exception is taken. |
| SI_RP | O | This signal represents the state of the RP bit (27) in the CP0 Status register. Software can write this bit to indicate that the device can enter a reduced power mode. |
| SI_SLEEP | O | This signal is asserted by the core whenever the WAIT instruction is executed. The assertion of this signal indicates that the clock has stopped and that the core is waiting for an interrupt. |
| Interrupt Signals: | | |
| SI_Int[5:0] | A | Active high Interrupt pins. These signals are driven by external logic and when asserted indicate the corresponding interrupt exception to the core. These signals go through synchronization logic and can be asserted asynchronously to SI_ClkIn |
| SI_TimerInt | O | This signal is asserted whenever the Count and Compare registers match and is deasserted when the Compare register is written. In order to have timer interrupts, this signal needs to be brought back into the 4K core on one of the six SI_Int interrupt pins. Traditionally, this has been accomplished via muxing SI_TimerInt with SI_Int[5]. Exposing SI_TimerInt as an output allows more flexibility for the system designer. Timer interrupts can be muxed or ORed into one of the interrupts, as desired in a particular system. In a complex system, it could even be fed into a priority encoder to allow SI_Int[5:0] to map up to 63 interrupt sources. |
| Configuration Inputs: | | |
| SI_Endian | S | Indicates the base endianness of the core.<br><br>**EB_Endian** / **Base Endian Mode**<br>0 — Little Endian<br>1 — Big Endian |

**Table 2-3 Signal Descriptions (Continued)**

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td>SI_MergeMode[1:0]</td><td>S</td><td>The state of these signals determines the merge mode for the 16-byte collapsing write buffer. See Section 3.5.1 , "Merge Pattern Control" on page 21 for more information about these modes.
<table>
<tr><th>Encoding</th><th>Merge Mode</th></tr>
<tr><td>00</td><td>No Merge</td></tr>
<tr><td>01</td><td>SysAD Valid</td></tr>
<tr><td>10</td><td>Full Merge</td></tr>
<tr><td>11</td><td>Reserved</td></tr>
</table></td></tr>
<tr><td colspan="3">EC™ interface Refer to Chapter 3, “EC™ Interface,” on page 9 for more details.</td></tr>
<tr><td>EB_ARdy</td><td>I</td><td>Indicates whether the target is ready for a new address. The core will not complete the address phase of a new bus transaction until the clock cycle after EB_ARdy is sampled asserted.</td></tr>
<tr><td>EB_AValid</td><td>O</td><td>When asserted, indicates that the values on the address bus and access types lines are valid, signifying the beginning of a new bus transaction. EB_AValid must always be valid.</td></tr>
<tr><td>EB_Instr</td><td>O</td><td>When asserted, indicates that the transaction is an instruction fetch versus a data reference. EB_Instr is only valid when EB_AValid is asserted.</td></tr>
<tr><td>EB_Write</td><td>O</td><td>When asserted, indicates that the current transaction is a write. This signal is only valid when EB_AValid is asserted.</td></tr>
<tr><td>EB_Burst</td><td>O</td><td>When asserted, indicates that the current transaction is part of a cache fill or a write burst. Note that there is redundant information contained in EB_Burst, EB_BFirst, EB_BLast, and EB_BLen. This is done to simplify the system design - the information can be used in whatever form is easiest.</td></tr>
<tr><td>EB_BFirst</td><td>O</td><td>When asserted, indicates beginning of burst. EB_BFirst is always valid.</td></tr>
<tr><td>EB_BLast</td><td>O</td><td>When asserted, indicates end of burst. EB_BLast is always valid.</td></tr>
<tr><td>EB_BLen<1:0></td><td>O</td><td>Indicates length of the burst. This signal is only valid when EB_AValid is asserted.
<table>
<tr><th>EB_BLength<1:0></th><th>Burst Length</th></tr>
<tr><td>0</td><td>reserved</td></tr>
<tr><td>1</td><td>4</td></tr>
<tr><td>2</td><td>reserved</td></tr>
<tr><td>3</td><td>reserved</td></tr>
</table></td></tr>
<tr><td>EB_SBlock</td><td>S</td><td>When sampled asserted, sub block ordering is used. When sampled deasserted, sequential addressing is used.</td></tr>
<tr><td>EB_BE<3:0></td><td>O</td><td>Indicates which bytes of the EB_RData or EB_WData buses are involved in the current transaction. If an EB_BE signal is asserted, the associated byte is being read or written. EB_BE lines are only valid while EB_AValid is asserted
<table>
<tr><th>EB_BE Signal</th><th>Read Data Bits Sampled</th><th>Write Data Bits Driven Valid</th></tr>
<tr><td>EB_BE<0></td><td>EB_RData<7:0></td><td>EB_WData<7:0></td></tr>
<tr><td>EB_BE<1></td><td>EB_RData<15:8></td><td>EB_WData<15:8></td></tr>
<tr><td>EB_BE<2></td><td>EB_RData<23:16></td><td>EB_WData<23:16></td></tr>
<tr><td>EB_BE<3></td><td>EB_RData<31:24></td><td>EB_WData<31:24></td></tr>
</table></td></tr>
</table>

**Table 2-3 Signal Descriptions (Continued)**

| Signal Name | Type | Description |
|---|---|---|
| EB_A<35:2> | O | Address lines for external bus. Only valid when EB_AValid is asserted. EB_A[35:32] are tied to 0 in the 4K cores. |
| EB_WData<31:0> | O | Output data for writes |
| EB_RData<31:0> | I | Input Data for reads |
| EB_RdVal | I | Indicates that the target is driving read data on EB_RData lines. EB_RdVal must always be valid. EB_RdVal may never be sampled asserted until the rising edge after the corresponding EB_ARdy was sampled asserted. |
| EB_WDRdy | I | Indicates that the target of a write is ready. The EB_WData lines can change in the next clock cycle. EB_WDRdy will not be sampled until the rising edge where the corresponding EB_ARdy is sampled asserted. |
| EB_RBErr | I | Bus error indicator for read transactions. EB_RBErr is sampled on every rising clock edge until an active sampling of EB_RdVal. EB_RBErr sampled with asserted EB_RdVal indicates a bus error during read. EB_RBErr must be deasserted in idle phases. |
| EB_WBErr | I | Bus error indicator for write transactions. EB_WBErr is sampled at the rising clock edge following an active sample of EB_WDRdy. EB_WBErr must be deasserted in idle phases. |
| EB_EWBE | I | Indicates that any external write buffers are empty. The external write buffers must deassert EB_EWBE in the cycle after the corresponding EB_WDRdy is asserted and keep EB_EWBE deasserted until the external write buffers are empty. See Section 3.6 , "External Write Buffers" on page 23 for more details |
| EB_WWBE | O | When asserted, indicates that the core is waiting for external write buffers to empty. See Section 3.6 , "External Write Buffers" on page 23 for more details. |
| EJTAG Interface: Refer to Chapter 4, “EJTAG Interface,” on page 25 for more details | | |
| 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. | | |
| EJ_TRST_N | I | Active low Test Reset Input (TRST*) for the EJTAG TAP. EJ_TRST_N must be asserted at power-up to cause the TAP controller to be reset. |
| EJ_TCK | I | Test Clock Input (TCK) for the EJTAG TAP. |
| EJ_TMS | I | Test Mode Select Input (TMS) for the EJTAG TAP. |
| EJ_TDI | I | Test Data Input (TDI) for the EJTAG TAP. |
| EJ_TDO | O | Test Data Output (TDO) for the EJTAG TAP. |
| EJ_TDOzstate | O | Drive indication for the output of TDO for the EJTAG TAP at chip level:<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><br>IEEE Standard 1149.1-1990 defines TDO as a tri-stated signal. To avoid having a tri-state core output, the 4K core outputs this signal to drive an external tri-state buffer. |
| Debug Interrupt: | | |
| EJ_DINTsup | S | Value of DINTsup for the Implementation register. A 1 on this signal indicates that the EJTAG probe can use DINT signal to interrupt the processor. This signal should be asserted if the DINT pin on the EJTAG probe header is connected to the EJ_DINT input of the core. |
| EJ_DINT | I | Debug exception request when this signal is asserted in a CPU clock period after being deasserted in the previous CPU clock period. The request is cleared when debug mode is entered. Requests when in debug mode are ignored. |

**Table 2-3 Signal Descriptions (Continued)**

| Signal Name | Type | Description |
|---|---|---|
| Debug Mode Indication | | |
| EJ_DebugM | O | Asserted when the core is in DebugMode. This can be used to bring the core out of a low power mode (see Section 6.3 , "Power Management" on page 45 for more details). In systems with multiple processor cores, this signal can be used to synchronize the cores when debugging. |
| Device ID bits: These inputs provide an identifying number visible to the EJTAG probe. If the EJTAG TAP controller is not implemented, these inputs are not connected. These inputs are always available for soft core customers. On hard cores, the core “hardener” may 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 Publications106, which can be found at: http://www.jedec.org/<br><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 |
| **Performance Monitoring Interface:** These signals can be used to implement performance counters which can be used to monitor HW/SW performance | | |
| PM_DCacheHit | O | This signal is asserted whenever there is a data cache hit. |
| PM_DCacheMiss | O | This signal is asserted whenever there is a data cache miss. |
| PM_DTLBHit | O | This signal is asserted whenever there is a hit in the data TLB. This signal is valid only on the 4Kc™ core and should be ignored when using the 4Kp™ and 4Km™ cores. |
| PM_DTLBMiss | O | This signal is asserted whenever there is a miss in the data TLB. This signal is valid only on the 4Kc core and should be ignored when using the 4Kp and 4Km cores. |
| PM_ICacheHit | O | This signal is asserted whenever there is an instruction cache hit. |
| PM_ICacheMiss | O | This signal is asserted whenever there is an instruction cache miss. |
| PM_InstComplete | O | This signal is asserted each time an instruction completes in the pipeline. |
| PM_ITLBHit | O | This signal is asserted whenever there is an instruction TLB hit. This signal is valid only on the 4Kc core and should be ignored when using the 4Kp and 4Km cores. |

**Table 2-3 Signal Descriptions (Continued)**

| Signal Name | Type | Description |
|---|---|---|
| PM_ITLBMiss | O | This signal is asserted whenever there is an instruction TLB miss. This signal is valid only on the 4Kc core and should be ignored when using the 4Kp and 4Km cores. |
| PM_JTLBHit | O | This signal is asserted whenever there is a joint TLB hit. This signal is valid only on the 4Kc core and should be ignored when using the 4Kp and 4Km cores. |
| PM_JTLBMiss | O | This signal is asserted whenever there is a joint TLB miss. This signal is valid only on the 4Kc core and should be ignored when using the 4Kp and 4Km cores. |
| PM_WTBMerge | O | This signal is asserted whenever there is a successful merge in the write through buffer. |
| PM_WTBNoMerge | O | This signal is asserted whenever a non-merging store is written to the write through buffer. |
| **Scan Test Interface:** These signals provide the interface for testing the core. The use and configuration of these pins are implementation dependent. | | |
| ScanEnable | I | This signal should be asserted while scanning vectors into or out of the core. The ScanEnable signal must be deasserted during normal operation and during capture clocks in test mode. |
| ScanMode | I | This signal should be asserted during all scan testing both while scanning and during capture clocks. The ScanMode signal must be deasserted during normal operation. |
| ScanIn<n:0> | I | This signal is input to scan chain. |
| ScanOut<n:0> | O | This signal is output from scan chain. |
| BistIn<n:0> | I | Input to the BIST controller |
| BistOut<n:0> | O | Output from the BIST controller |

# EC™ Interface

## 3.1 Introduction

This chapter describes the EC™ interface, which is present on all MIPS32 4K™ processor cores. The EC interface is generally described in the companion document, titled *EC™ Interface Specification*, which is included in this release (file `$JADEHOME/doc/ECiSpec.pdf`). The rest of this chapter discusses the specific 4K implementation of the EC interface.

## 3.2 Interface Transactions

The cores implement 32-bit unidirectional data buses: *EB_RData[31:0]* for read operations and *EB_WData[31:0]* for write operations. The following sections describe following bus transactions:



### 3.2.1 Fastest Read Transaction

The core allows data to be returned in the same clock that the address is driven onto the bus. This is the fastest type of read cycle as shown in Figure 3-1.

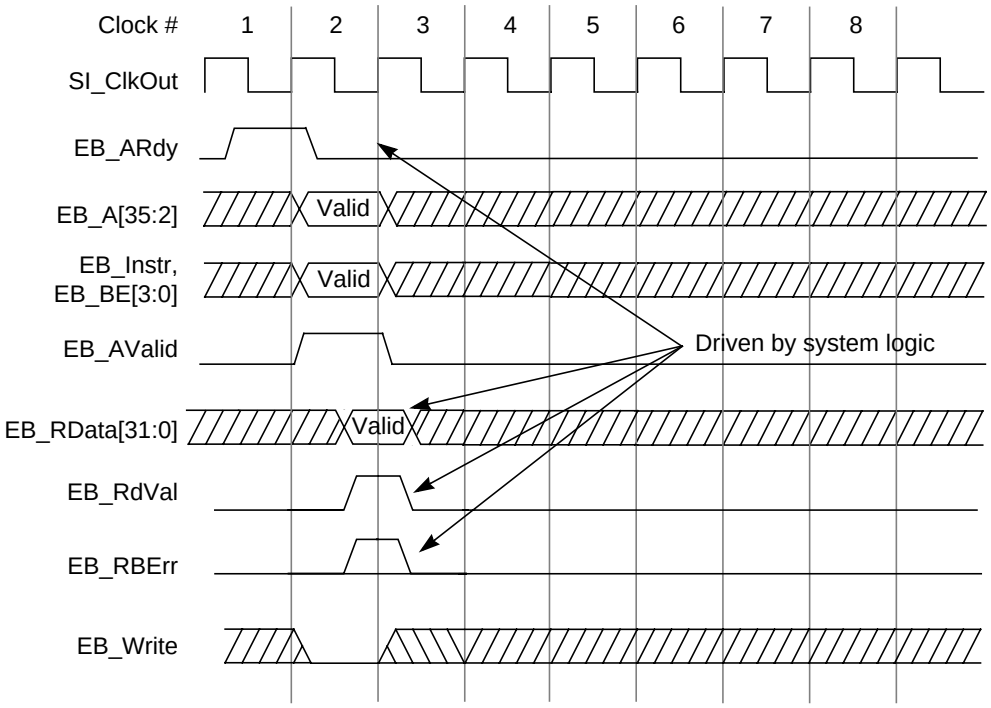


**Figure 3-1 Fastest Read Cycle**

In this transaction, the core drives address and control onto the bus and samples *EB_RdVal* active on the next rising edge of the clock.

### 3.2.2 Single Read with Wait States

Figure 3-2 shows the basic timing relationships of signals during a single read transaction with wait states. The core drives address onto *EB_A[35:2]* and byte enable information onto *EB_BE[3:0]*. To maximize performance, the bus interface does not define a maximum number of outstanding bus cycles. Instead the interface provides the *EB_ARdy* input signal; this signal is driven by external logic and controls the generation of addresses on the bus. Current versions of the 4K cores can only have a maximum of 8 reads and 4 writes outstanding, but future version may have a larger number of outstanding transactions.

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

The *EB_Instr* signal is asserted during a single read cycle if the read is for an instruction fetch. The *EB_AValid* signal is driven in each clock that *EB_A[35:2]* is valid on the bus. The core drives the *EB_Write* signal low to indicate a read transaction.

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

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

Figure 3-2 shows a single read transaction with one address wait state and one data wait state.

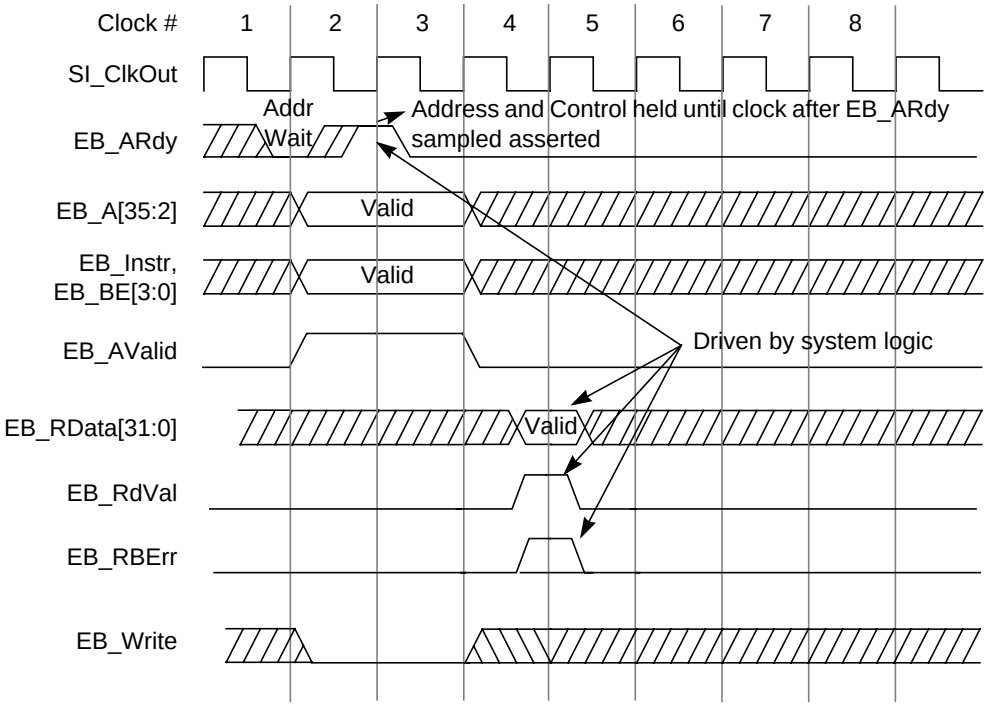


**Figure 3-2 Single Read Transaction with Wait States**

### 3.2.3 Fastest Write Transaction

The core allows the *EB_WDRdy* signal to be driven active in the same clock that address and data are driven onto the bus. This is the fastest type of write cycle as shown in Figure 3-3.

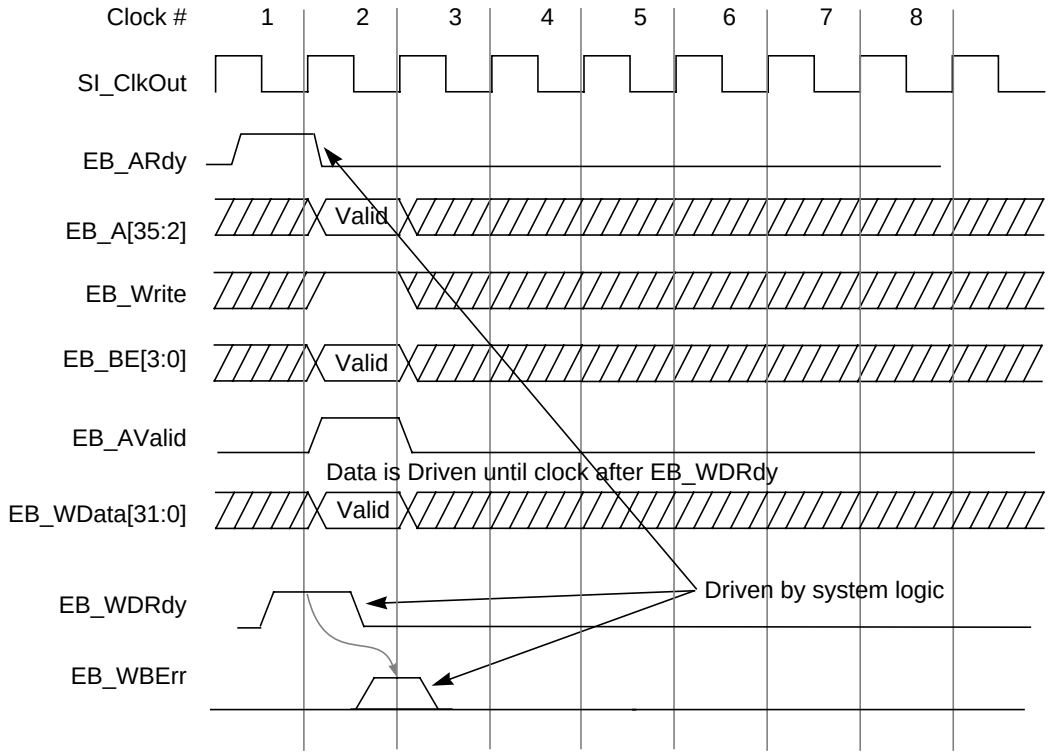


**Figure 3-3 Fastest Write Transaction**

In this transaction, the core drives address and control onto the bus and samples *EB_WDRdy* active on the next rising edge of the clock.

### 3.2.4 Single Write with Wait States

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

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

Figure 3-4 shows a write transaction with one address wait state and two data wait states.

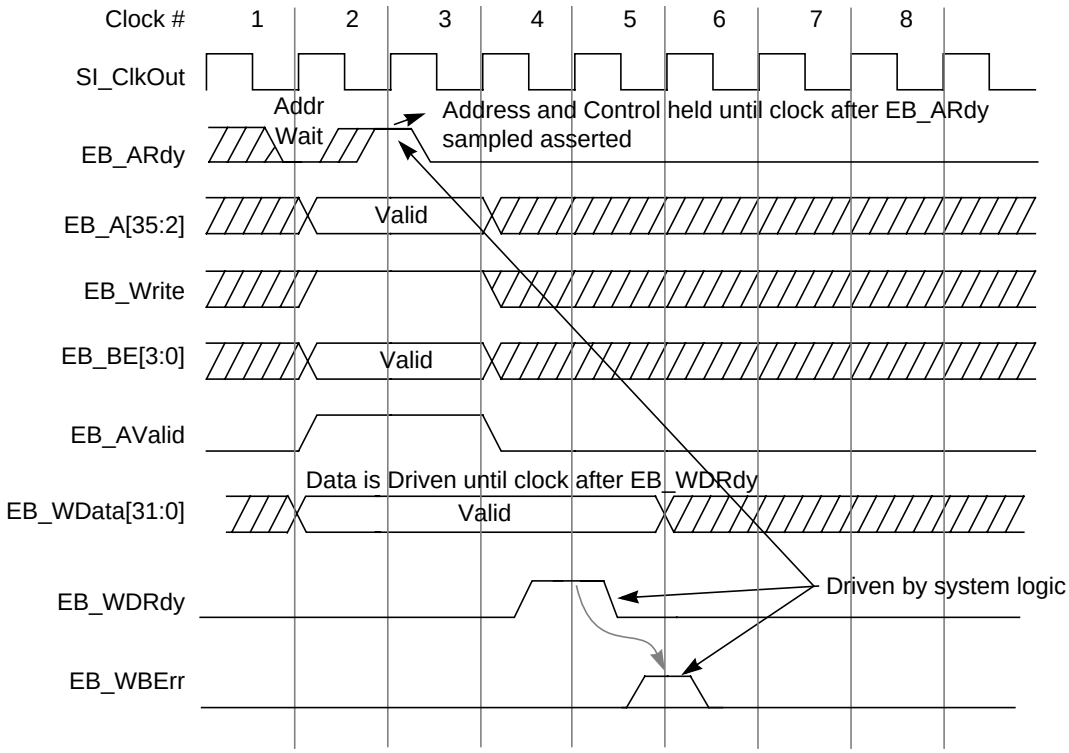


**Figure 3-4 Single Write Transaction with Wait States**

### 3.2.5 Burst Read

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

Figure 3-5 shows an example of a burst read transaction. Burst read transactions initiated by the core always contain four data transfers. In addition, the data requested is always a 16- byte-aligned block. Burst reads are always initiated for cacheable instruction or data reads which have missed in the primary instruction or data cache.

The order of words within this 16-byte block varies depending on which of the words in the block is being requested by the execution unit and the ordering protocol selected. The burst always starts with the word requested by the execution unit and proceeds in either an ascending or descending order wrapping at the end of an aligned block. Table 3-1 and Table 3-2 show the sequence of address bits 2 and 3.

**Table 3-1 Sequential Burst Order**

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

**Table 3-2 SubBlock Burst Order**

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

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

The EB_Instr signal is asserted if the cycle is an instruction fetch. The *EB_Burst* signal is asserted throughout the cycle to indicate that a burst transaction is in progress. The core asserts the *EB_BFirst* signal in the same clock as the first address is driven to indicate the start of a burst cycle. In the clock that the last address is driven, the core asserts *EB_BLast* to indicate the end of the burst transaction.

The core first samples the *EB_RData[31:0]* bus one clock after *EB_ARdy* is sampled asserted. External logic asserts *EB_RdVal* to indicate that valid data is on the bus. The core latches data internally whenever *EB_RdVal* is sampled asserted.

Note that at the rising edge of clock 6 in Figure 3-5 the *EB_RdVal* signal is sampled deasserted, causing a wait state between *Data 2* and *Data 3*. External logic asserts the *EB_RBErr* signal in the same clock as data if a bus error occurs during that data transfer.

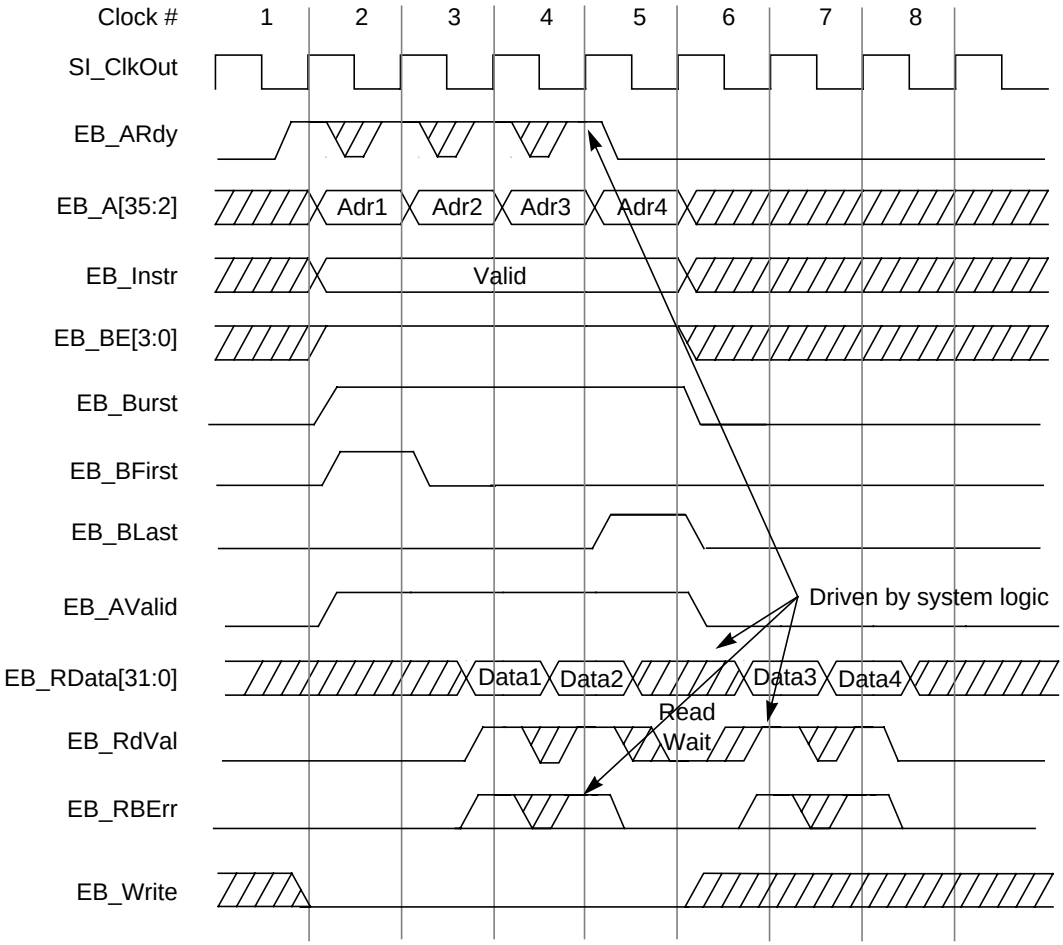


**Figure 3-5 Burst Read Transaction Timing Diagram**

### 3.2.6 Burst Write

Burst write transactions are used to empty one of the write buffers. A burst transaction is only performed if the write buffer contains 16 bytes of data associated with the same aligned memory block; otherwise individual write transactions are performed. Figure 3-6 shows a timing diagram of a burst write transaction. Unlike the read burst, a write burst always begins with EB_A[3:2] equal to 00b.

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

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

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

In Figure 3-6 the first data word (*Data1*) is driven in clocks 2 and 3 due to the *EB_WDRdy* signal being sampled deasserted at the rising edge of clock 2, causing one wait state cycle. When *EB_WDRdy* is sampled asserted at the rising edge of clock 3, the core responds by driving the second word (*Data2*) at the rising edge of clock 4.

External logic drives the EB_WBErr signal one clock after the corresponding assertion of EB_WDRdy if a bus error has occurred as shown by the arrows in Figure 3-6.

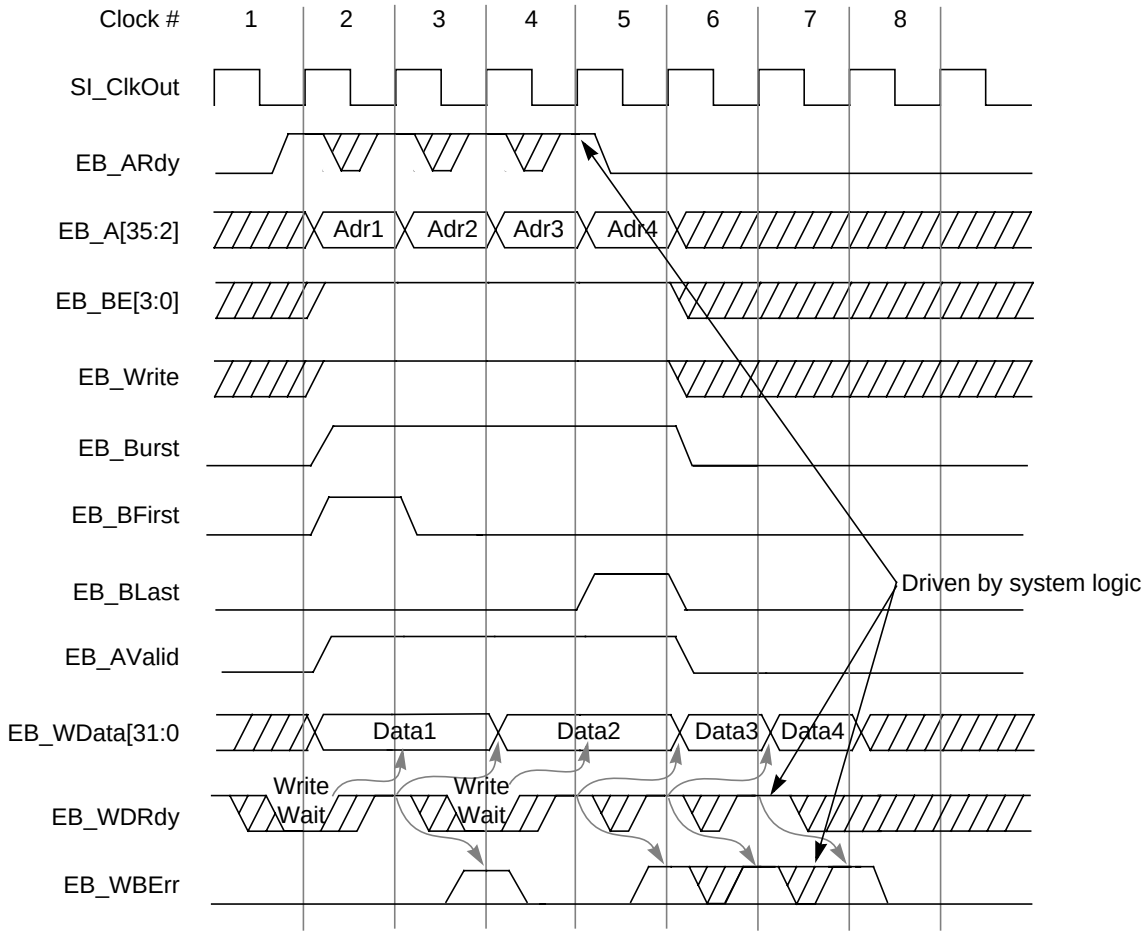


**Figure 3-6 Burst Write Transaction Timing Diagram**

### 3.2.7 Back-to-Back Reads

Figure 3-7 shows the basic timing relationships of signals during a back-to-back read transaction. During a back-to-back read cycle, the core drives addresses for both read cycles onto *EB_A[35:2]* and byte enable information onto *EB_BE[3:0]*. The 4K cores always leave a dead clock between new address transactions (but address transactions within a burst will not have the dead clock). This dead clock is not part of the EC interface specification and future cores may not have this.

To maximize performance, the core does not define a maximum number of outstanding bus cycles. Instead the core provides the *EB_ARdy* input signal. This signal is driven by external logic and controls the generation of addresses on the bus.

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

The *EB_Instr* signal is asserted during a read cycle if the read is for an instruction fetch. The *EB_AValid* signal is driven in each clock that *EB_A[35:2]* is valid on the bus. The core drives the *EB_Write* signal low to indicate a read transaction.

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

For the two back-to-back reads shown in Figure 3-7, the first read has one address wait state. The first read has one data wait state since the *EB_RdVal* for that read is sampled in clock 5, two cycles after the sampled assertion of *EB_ARdy*. The second read data is returned as fast as possible, with no data wait states since its *EB_RdVal* is sampled in clock 6, one clock after the sampling of its *EB_ARdy*.

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

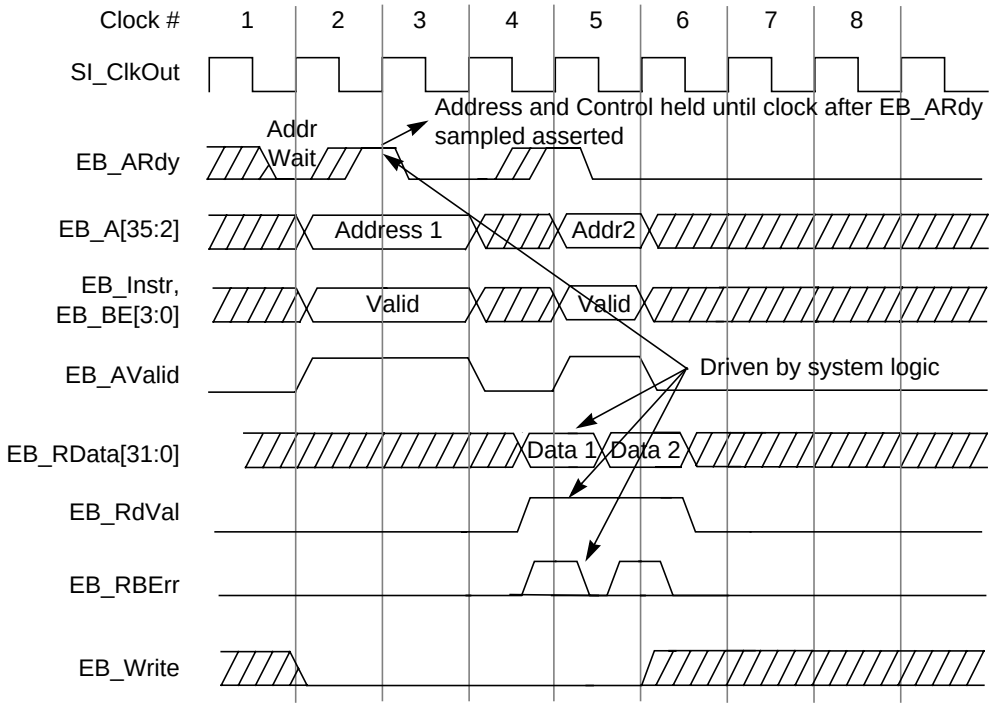


**Figure 3-7 Back-to-Back Read Transaction Timing Diagram**

### 3.2.8 Back-to-Back Writes

Figure 3-8 shows a timing diagram of a back-to-back write operation. In any bus transaction the core drives address, control, and data information onto the bus as it becomes available, regardless of the state of *EB_ARdy*. If the *EB_ARdy* signal is asserted at the time that the address is driven by the processor, indicating that the external agent can accept another address, the processor can drive a new address on the following clock.

In Figure 3-8, address, control, and data (*Write1/Data1*) become available and are driven onto the bus by the core during clock 2. *EB_ARdy* is sampled deasserted at the rising edge of clock 2, indicating that the external agent is not ready to accept a new address. This causes one address wait state for the *Write1* address. The processor continues to drive the bus with the *Write1* address and control until the clock after *EB_ARdy* is sampled asserted. In this case, *EB_ARdy* is sampled asserted at the rising edge of clock 3, allowing the processor to drive new address and control at the rising edge of clock 4. The new address (*Write2*) is not driven until the rising edge of clock 5. Since *EB_ARdy* was immediately sampled asserted, there were no address wait states on *Write2*. The core might drive a new address onto the bus at the rising edge of clock 6 (not shown).

The *EB_WDRdy* signal is driven by the external agent to indicate that it has accepted the data on the bus. In this example the *EB_WDRdy* signal is sampled deasserted by the core at the rising edge of clock 3, causing the core to hold the data (*Data1*) during the following cycle, clock 4. The external agent asserts *EB_WDRdy* during clock 3, which is sampled by

the core on the rising edge of clock 4, indicating that is has accepted the data on the bus. The core continues to drive data until one clock after *EB_WDRdy* is sampled asserted, so the core drives *Data1* until the rising edge of clock 5. Note that *EB_WDRdy* will never be sampled earlier than the rising edge in which the associated *EB_ARdy* is sampled asserted. So if *EB_WDRdy* was asserted on the rising edge of clock 2 (one cycle before the *EB_ARdy*), it would have been ignored.

The core can drive new data (*Data2*) at the rising edge of clock 5. At the rising edge of clock 5 the core samples *EB_WDRdy* deasserted, causing the processor to hold *Data2* in the following cycle. At the rising edge of clock 6 *EB_WDRdy* is sampled asserted, so the core can stop driving *Data2* at the rising edge of clock 7.

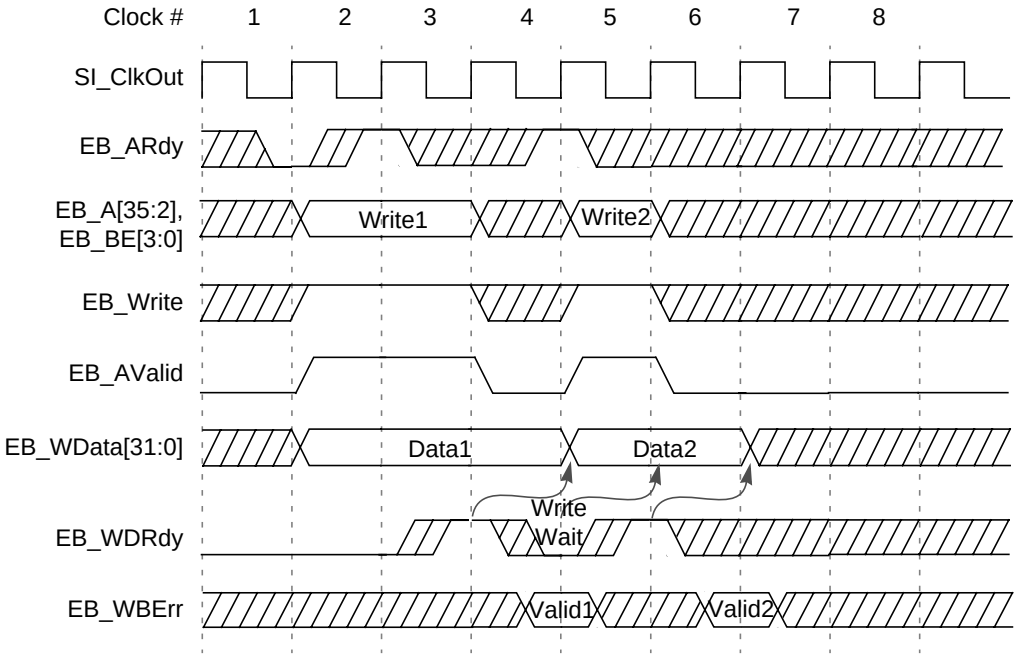


**Figure 3-8 Back-to-Back Write Transactions**

### 3.2.9 Read Followed by Write with Reordering

Figure 3-9 shows a timing diagram of a read followed by write operation with the operations being completed out of order. Since data is transferred for read and write operations on independent unidirectional busses (and their corresponding ready indicators), the bus interface allows read and write operations to complete out of order with respect to how the read and write addresses were initiated.

In any bus transaction the core drives address, control, and data information onto the bus as it becomes available, regardless of the state of *EB_ARdy*. If the *EB_ARdy* signal is asserted at the time that address is driven by the processor, indicating that the external agent can accept the address, the processor can drive a new address on the following clock.

In Figure 3-9, address and control for the read operation become available and are driven onto the bus by the core at the rising edge of clock 2. The external agent has *EB_ARdy* asserted so there are no address wait states for the read. The processor continues to drive the bus until one clock after *EB_ARdy* is sampled asserted. After a dead clock, the write address and control information are driven at the rising edge of clock 4. The external agent asserts *EB_ARdy* for an additional clock, which is sampled by the core at the rising edge of clock 4, so the core could have driven a new address (not shown) at the rising edge of clock 6.

In this example, the external agent asserts the *EB_WDRdy* signal at the rising edge of clock 4, indicating its ability to accept the write data, even though the read operation has not completed. The core drives write data for one clock after *EB_WDRdy* has been sampled asserted. This causes the processor to drive data until the rising edge of clock 5.

In this example read data is driven onto the bus during clock 5, one clock after the write operation has completed. The core samples the *EB_RdVal* signal asserted at the rising edge of clock 6, causing the processor to latch the data and terminates the read cycle. Note that it is the responsibility of the external agent to ensure the correct data is returned when re-ordering data transactions.

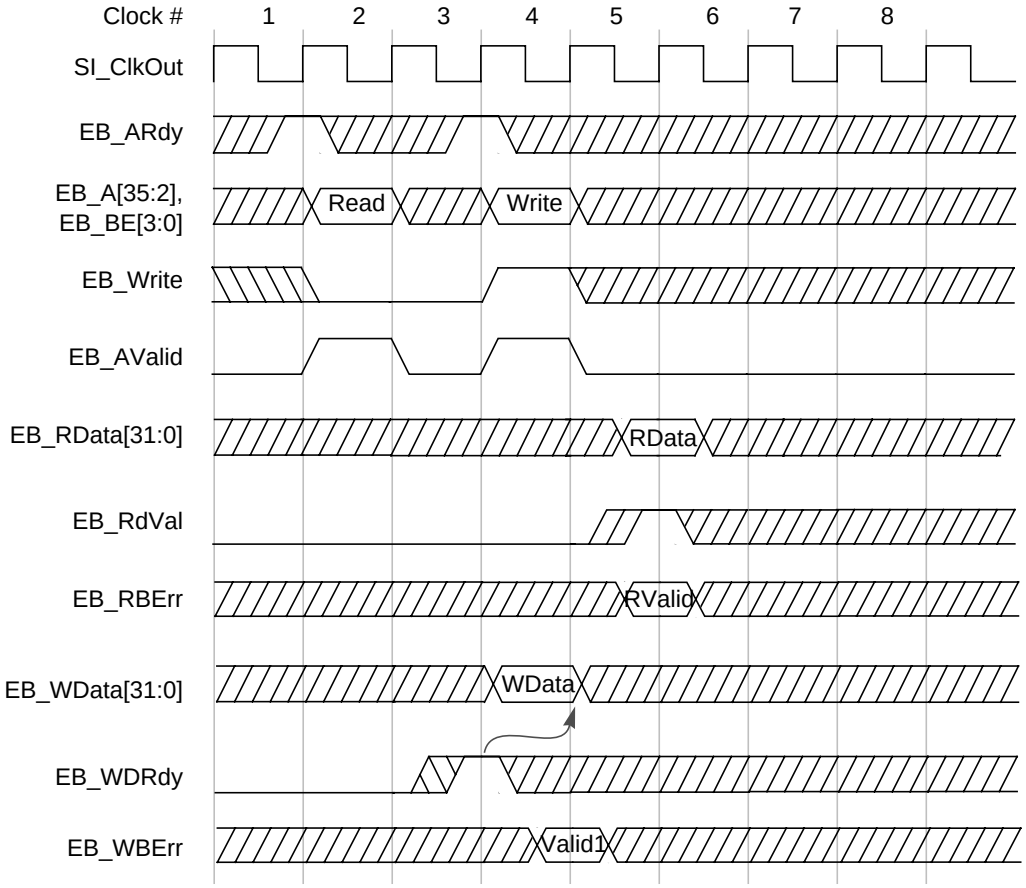


**Figure 3-9 Read Followed by Write Transaction with Reordering**

### 3.2.10 Write Followed by Read with Reordering

Figure 3-10 shows a timing diagram of a write followed by read operation with the operations being completed out of order.

In any bus transaction the core drives address, control, and data information onto the bus as it becomes available, regardless of the state of *EB_ARdy*. If the *EB_ARdy* signal is asserted at the time that address is driven by the processor, indicating that the external agent can accept the address, the processor can drive a new address on the following clock.

In Figure 3-10, address, control, and data for the write operation become available and are driven onto the bus by the core at the rising edge of clock 2. The processor continues to drive the address and control busses until *EB_ARdy* is sampled asserted. Since *EB_ARdy* was not sampled asserted at the rising edge of clock 2, an address wait state results. The assertion of *EB_ARdy* is sampled at the rising edge of clock 3, causing the processor to drive the write address and control until the rising edge of clock 4. The read address and control are then initiated on the bus at the rising edge of clock 5. Note that a new address (not shown) could have been driven on the bus at the rising edge of clock 7.

In this example the external agent drives read data and asserts the *EB_RdVal* signal in clock 5, indicating that valid read data is on the bus, even though the write operation has not completed. The core latches the read data at the rising edge of clock 6, thereby completing the read operation.

By default, the core drives write data at the same time as the write address and continues to drive data for one clock after *EB_WDRdy* is sampled asserted. This causes the processor to drive data in clocks 2 - 7. In this example, the external agent asserts *EB_WDRdy* in clock 6 and is sampled active by the core at the rising edge of clock 7, one clock after the read operation has completed. The core continues to drive data until the rising edge of clock 8 and the write operation is completed. Note that it is the responsibility of the external agent to ensure the correct data is returned when re-ordering data transactions

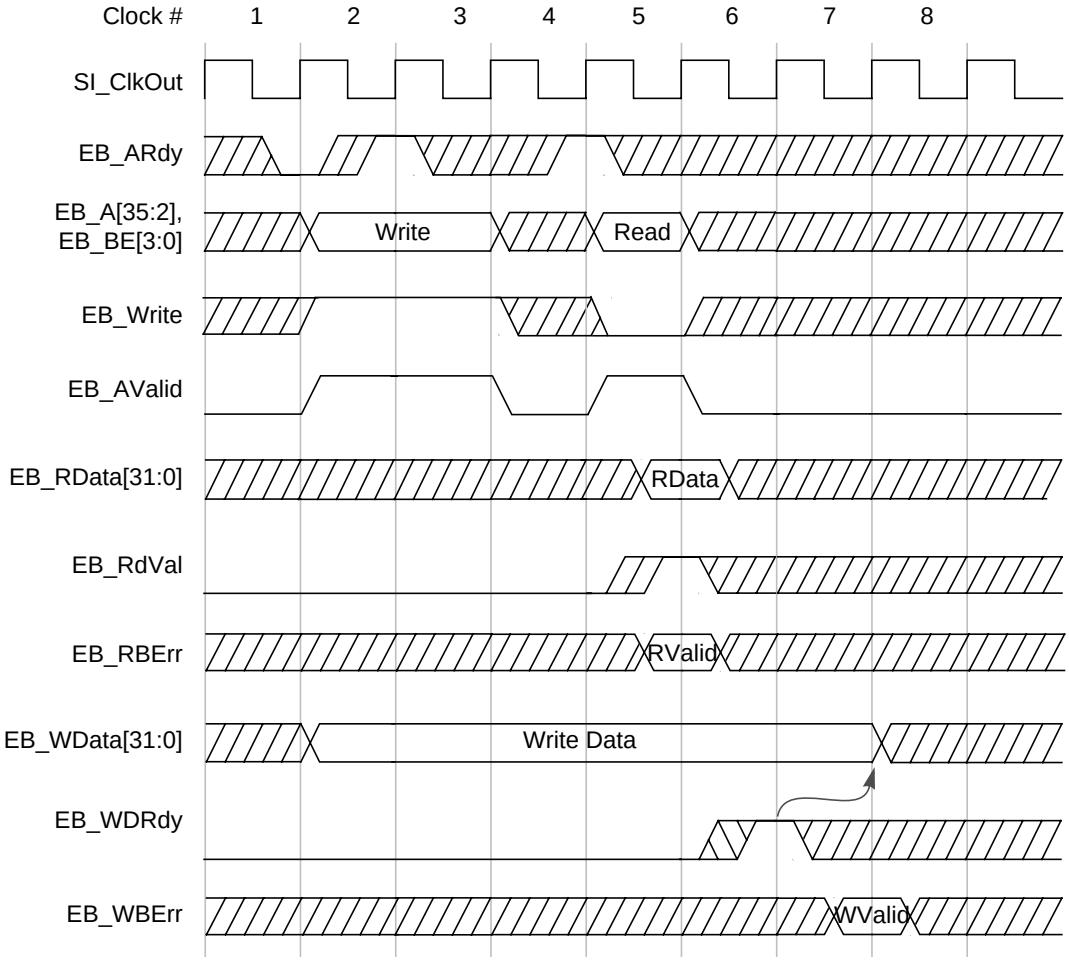


**Figure 3-10 Write Followed by Read Transaction with Reordering**

## 3.3 Outstanding Transactions

The EC interface itself does not limit the number of transactions which may be active at any time. Instead, the number of external transactions can be throttled via control of the *EB_ARdy* input. This input indicates that the external controller can accept a new transaction.

The bus interface implementation of the 4K core contains enough buffering to allow a maximum of 12 transactions to be active simultaneously, as follows:

- Four bursted instruction reads
- Four bursted data reads
- Four bursted writes

When designing a generic EC interface controller, keep in mind that other MIPS processor cores designed to the EC interface may allow a different number of transactions to be active.

## 3.4 Sequential Transactions

The 4K cores always leave a dead clock between address transactions to a new line. There is not a dead clock between the beats of a bursted transaction or between mutliple writes to the same 16B line. This dead clock is not required by the EC interface specification. When designing a generic EC interface controller, keep in mind that other MIPS processor cores may not have this dead clock.

Back to back read accesses of the same type (Instruction or Data) will have an additional timing constraint. Only one request of a given type is allowed to be outstanding. Therefore, the second address will not come out onto the bus until at least 1 cycle after the last read data for the first address was returned. Again, this behavior is not part of the EC interface specification and should not be relied on in a generic EC interface controller.

## 3.5 Write Buffer

The write buffer is organized as two 16 byte buffers. Each buffer contains data from a single 16 byte aligned block of memory. One buffer contains the data currently being transferred on the external interface, while the other buffer contains accumulating data from the core.

Data from the accumulation buffer is transferred to the external interface buffer under one of the conditions:

- When a store is attempted from the core to a different 16-byte block than is currently being accumulated.
- **SYNC** instruction. The **CACHE** instruction performs an implicit **SYNC**.
- Store to a invalid merge pattern.
- Any stores to uncached memory.
- A load to the line being merged.

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

### 3.5.1 Merge Pattern Control

All 4K cores implement two 16 byte collapsing write buffers that allow byte, half-word, or word writes from the core to be accumulated in the buffer into a 16 byte value before bursting the data out onto the bus in word format. Note that writes to uncached areas are never merged.

The core provides three options for merge pattern control:

- No merge
- Full merge
- SysAD valid

The merging option is selected by the *SI_MergeMode[1:0]* input. The encoding is shown in Table 2-3 on page 4.

#### 3.5.1.1 No Merge

In *No Merge* mode writes to a different word within the same line are accumulated in the buffer. A burst write transaction occurs when the buffer becomes full (4 words). Stores to the same word cause the previous word to be driven onto the bus.

#### 3.5.1.2 Full Merge

In *Full Merge* mode all combinations of writes to the buffer are allowed.

#### 3.5.1.3 SysAD Valid Merge

In *SysAD Valid Merge* mode only valid SysAD byte enable patterns to be stored into the write buffer. When the byte enable pattern of a write to the buffer is merged with the pattern for the previous write, the resulting pattern must be a valid SysAD pattern in order for the merge to occur. If the resulting pattern is not a valid SysAD pattern the previous write is driven onto the bus before the current write is written to the buffer. Table 3-3 shows the valid SysAD patterns.

**Table 3-3 Valid SysAD Byte Enable Patterns**

| EB_BE[3:0] |
|---|
| 0001 |
| 0010 |
| 0100 |
| 1000 |
| 0011 |
| 1100 |
| 0111 |
| 1110 |
| 1111 |

Table 3-4 shows a typical code sequence of three store byte instructions and the behavior of each merge mode.

**Table 3-4 MergeMode Example**

| Instruction | EB_BE[3:0] (No Merge) | EB_BE[3:0] (Full Merge) | EB_BE[3:0] (SysAD Merge) |
|---|---|---|---|
| SB 0 | 0001 | merge | 0001 |
| SB 1 | 0010 (0001 written out) | merge (0011) | merge (0011) |
| SB 2 | 1000 (0010 written out) | merge (1011) | 1011 = invalid merge pattern<br>0011 written out, then<br>1000 written to buffer |
| SYNC | flush buffer (1000 written out) | flush buffer (1011 written out) | flush buffer (1000 written out) |

In Table 3-4 above, the following sequence occurs in *No Merge* mode:

- The SB 0 instruction causes a value of 0001 on EB_BE[3:0] to be written to the write buffer.

- The SB 1 instruction causes a value of 0010 on EB_BE[3:0] to be written to the write buffer. Since no merging of data is allowed in this mode, the data from the SB 0 instruction is driven onto the bus along with the 0001 byte enable pattern.
- The SB 2 instruction causes a value of 1000 on EB_BE[3:0] to be written to the write buffer. Since no merging of data is allowed in this mode, the data from the SB 1 instruction is driven onto the bus along with the 0010 byte enable pattern.
- Execution of the SYNC instruction causes the contents of the buffer to be written out. In this example the SB 2 instruction is driven onto the bus along with a byte enable pattern of 1000.

In Table 3-4 above, the following sequence occurs in *Full Merge* mode:

- The SB 0 instruction causes a value of 0001 on EB_BE[3:0] to be written to the write buffer.
- The SB 1 instruction causes a value of 0010 on EB_BE[3:0] to be written to the write buffer. Since all patterns are allowed in Full Merge mode, a value of 0011 (0001 + 0010) is stored in the buffer.
- The SB 2 instruction causes a value of 1000 on EB_BE[3:0] to be written to the write buffer. Since all patterns are allowed in Full Merge mode, a value of 1011 (0001 + 0010 + 1000) is stored in the buffer.
- Execution of the SYNC instruction causes the contents of the buffer to be written out. In this example data from the SB 0, SB 1, and SB 2 instructions are driven onto the bus along with a byte enable pattern of 1011.

In Table 3-4 above, the following sequence occurs in *SysAD Valid Merge* mode:

- The SB 0 instruction causes a byte enable value of 0001 to be written to the write buffer.
- The SB 1 instruction causes a byte enable value of 0010 to be written to the write buffer. Since the merge value of 0011 (0001 + 0010) is a valid SysAD pattern, this value is written to the buffer.
- The SB 2 instruction causes a byte enable value of 1000 to be written to the write buffer. Since the merge value of 1011 (0001 + 0010 + 1000) is not a valid SysAD pattern, the buffer is emptied and the 0011 pattern is driven onto the bus. Once the buffer is empty, the new value (1000) is written to the buffer since it is a valid SysAD pattern.
- Execution of the SYNC instruction causes the contents of the buffer to be written out. In this example data from the SB 2 instruction is driven onto the bus along with a byte enable pattern of 1000.

## 3.6 External Write Buffers

Some systems may have external write buffers to increase bus efficiency and system performance. The core has a two-signal interface which can allow software to have some control over the external write buffers. The **SYNC** instruction is intended to form a barrier between load/store instructions before and after it in the instruction stream. Upon execution of a **SYNC** instruction, the core will complete all pending read requests and flush the internal write buffer. The core will also assert *EB_WWBE* to signal to the system that it is Waiting for the Write Buffer Empty signal. The **SYNC** instruction will not complete until the *EB_EWBE* input is asserted.

In most systems you can tie the *EB_EWBE* signal high. Just using the *EB_WWBE* signals does not ensure coherency. If a write is in the external write buffer the core can generate a read request to the given address without asserting *EB_WWBE* (because the core has no knowledge of the external write buffers). Therefore, any write buffers in the system must maintain coherency with reads.

The *EB_WWBE/EB_EWBE* interface can be used to make **SYNC**s “harder” by forcing the flush of the external write buffers. This is a system/SW design issue - you need to decide what if anything you want the system to do when a **SYNC** instruction is executed (and the same will be done for other synchronizing instructions such as **CACHE**).

# EJTAG Interface

This chapter discusses chip-level integration details for the EJTAG-related pins on a 4K core, as well as some system level requirements. A comparison of EJTAG versus JTAG is covered first, to clarify the differences and similarities. Then EJTAG chip and system issues related to one or multiple 4K cores within a single chip are discussed.

The EJTAG TAP controller is an optional feature in a 4K core. If your 4K core does not contain the EJTAG TAP controller, then most of this chapter is irrelevant.

A reference to the general "EJTAG Specification" can be found several times in this chapter. The full title of this document is *EJTAG Specification, Revision 2.5-1* (MIPS Document Number MD00027). The document is included with your 4K core deliverables, in file `$JADEHOME/doc/EJTAGSpec.pdf`. MIPS recommends that you become familiar with the general EJTAG Specification in addition to this chapter, before deciding how to integrate EJTAG into your chip.

## 4.1 EJTAG versus JTAG

The name EJTAG is often confused with IEEE JTAG boundary scan, but EJTAG is not related to boundary scan. EJTAG is a set of hardware-based debugging features on a MIPS processor, which are accessible by debug software. EJTAG is used by software programmers to control and debug code execution, as well as to access hardware resources within a MIPS processor, during code development. The interface for EJTAG access to the core does use a super-set of the JTAG TAP interface, but that is really its only similarity with boundary scan.

Please read the "EJTAG Debug Support" chapter in the *MIPS32 4K™ Processor Core Family Software User's Manual* to learn more about the software debugging capabilities of EJTAG.

### 4.1.1 EJTAG similarities to JTAG

From a functional viewpoint, the following features are inherited from the JTAG TAP interface:

- Protocol for selecting data and control registers using *EJ_TMS*.
- Serial protocol for transmitting data into and out of the selected register using *EJ_TDI* and *EJ_TDO*.
- Asynchronous reset to the EJTAG TAP controller using *EJ_TRST_N* (*TRST**).
- *EJ_TCK* driving the clock input of all the EJTAG TAP controller registers.

Because of these similarities it is possible to share certain physical resources between the TAP controllers in EJTAG and JTAG. MIPS generally recommends NOT sharing any logic or pins between JTAG and EJTAG. However MIPS does recognize that reducing pin count is often necessary in large SOC chip designs. The following section will discuss this issue further.

### 4.1.2 Sharing EJTAG resources with JTAG

It is theoretically possible to share the TAP controller for JTAG and EJTAG functionality, because the EJTAG control commands do not use reserved JTAG commands. This TAP sharing is not supported by the 4K core, however. The 4K core has its own independent TAP controller, reserved exclusively for EJTAG operation.

Because the EJTAG electrical specification is identical to the JTAG specification, it is possible to share the physical chip pins between the two TAP controllers for EJTAG and JTAG. There are two ways this might be accomplished, but both of them have issues which must be considered.

#### 4.1.2.1 Daisy chained TDI-TDO

One method is to hook up the physical pins *TCK*, *TMS* and *TRST** in parallel to both TAP controllers, and then daisy chain the *TDI/TDO* pins in the following manner:

- physical pin *TDI* to JTAG *TDI*
- JTAG *TDO* to EJTAG *EJ_TDI*
- EJTAG *EJ_TDO* to physical pin *TDO*. And EJTAG *EJ_TDOzstate* to output enable of physical *TDO*.

Figure 4-1 show the serial TDI-TDO chain setup with parallel control of the TAP controllers.

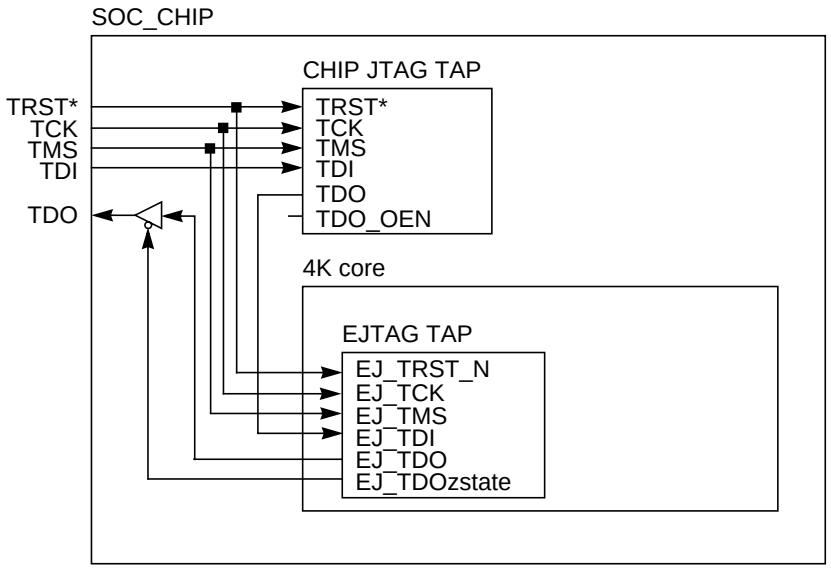


**Figure 4-1 Daisy chained TDI-TDO between JTAG and EJTAG TAP controller**

Some EJTAG debug tool-chains can handle this configuration. You can identify that there is another TAP controller in the path to the EJTAG TAP controller. And then tell the debug software the following items:

- the Instruction word length of the JTAG TAP controller
- the Instruction word command to select the bypass register
- the length of the bypass register

This will enable the debugger to always select the bypass register within the JTAG TAP controller during EJTAG access, and compensate for the bypass register length.

The main problem here is the presence of the serial EJTAG TAP controller in the JTAG TAP path; automatic JTAG test-benches generally do not like the visibility of another TAP controller inside the chip. MIPS strongly recommends NOT using the setup in Figure 4-1 for sharing TAP controller external pins between an EJTAG TAP and a JTAG TAP.

#### 4.1.2.2 Multiplexed pin access

A select signal can choose which TAP controller has access to the physical pins. How you wish to gate off the inputs of the un-selected TAP controller depends on the presence of an asynchronous reset input. In Figure 4-2 a setup which anticipates the existence of *TRST** on the “CHIP JTAG TAP” controller is shown.

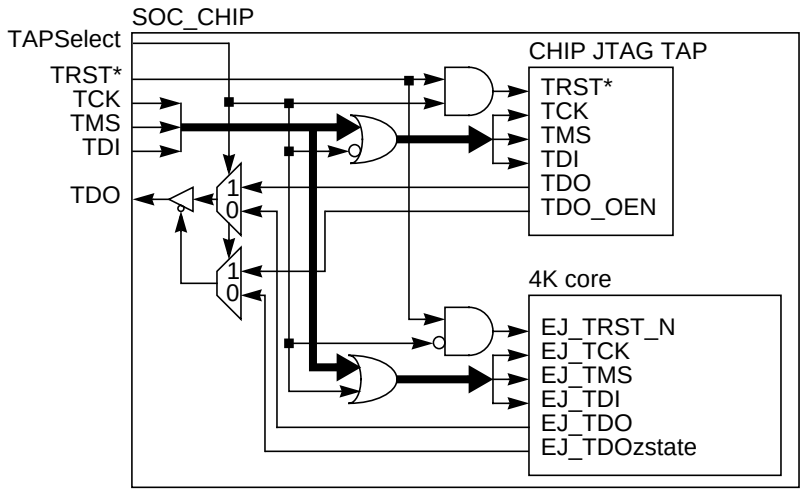


**Figure 4-2 Multiplexing between JTAG and EJTAG TAP controller**

*TAPSelect* in Figure 4-2 is shown as an SOC_CHIP external input, and NOT as internal logic or registered signal. This is so for two important reasons:

1. When doing board level interconnect testing. The JTAG controller should be able to work the boundary scan without any other controlled pins beyond the five JTAG pins.
2. When the board holding the SOC_CHIP is used for software development, EJTAG must be functional on the TAP controller while the 4K core (and thus probably the entire SOC_CHIP) is held in reset. During reset, EJTAG commands can initialize the 4K core to leave the reset state in Debug Mode, and thus the debug interface can control the 4K core before the it attempts to fetch the first instruction.

The two reasons above also imply that *TAPSelect* must be valid and fixed while using either of the two TAP controllers. For system integrity, *TAPSelect* should also be kept valid while there is no probe connected to the TAP Probe Connector. This is a violation of the EJTAG specification. It also leaves the *TAPSelect* input untested by JTAG boundary scan, it should in deed not be part of the boundary scan. This is, however, the only problem in this shared TAP controller configuration. A two-way jumper on the PCB could be created to select the fixed state of *TAPSelect*.

If pin sharing between EJTAG and JTAG TAP controllers is absolutely unavoidable, MIPS recommends the implementation shown in Figure 4-2.

## 4.2 How to connect *EJ_** pins

In the previous section, issues concerning the sharing of EJTAG TAP and JTAG TAP pins were discussed. This section assumes that your chip has a separate set of EJTAG TAP pins. Other non-TAP EJTAG pins on the 4K core will require separate pins on the chip, but most do not. This section will discuss how to connect all the *EJ_** pins in your chip.

### 4.2.1 EJTAG chip-level pins

The EJTAG TAP pins on the 4K core are: *EJ_TCK*, *EJ_TMS*, *EJ_TDI*, *EJ_TRST_N*, *EJ_TDO* and *EJ_TDOzstate*. An extra signal *EJ_DINT* (Debug Interrupt) can also be connected to an external pin. Figure 4-3 shows the intended connection to the chip. Pin names for the chip have been chosen as the usual JTAG TAP pins, with an "*E*" prefix.

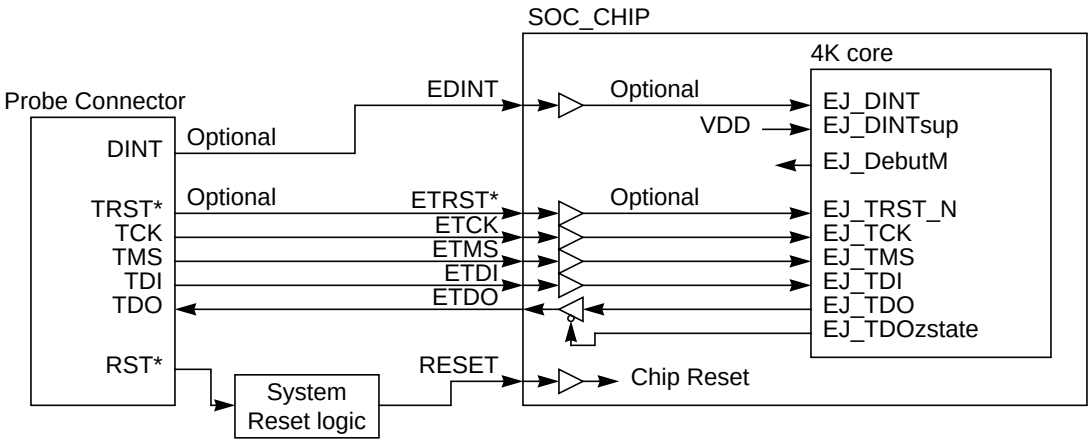


**Figure 4-3 EJTAG chip-level pin connection**

AC timing characteristics for the *ETDO* driver and the input buffers can be found in the “EJTAG Specification”, Section 7.2 “AC Timing Characteristics”. In particular notice here that all the probe pins must have pull-up or pull-down logic attached. As shown in Figure 4-3, all the chip-level pins have corresponding pins on the EJTAG Probe Connector. *RST** is special though, because an assertion (low active) on this pin must result in a system level reset. Please see Figure 4-4 for further details on EJTAG-related reset circuitry.

#### 4.2.1.1 Optional *ETRST** pin

The *ETSRT** in an optional input pin on the chip. However it is strongly recommended that the *ETRST** pin be present. If you choose not to include this pin, you will need on-chip logic which asserts *EJ_TRST_N* at power-up. This assertion can ONLY happen on power-up, or at cold-start. Any soft reset of the chip and 4K core must not affect the *EJ_TRST_N* signal. Special timing also applies to the deassertion of *EJ_TRST_N*. Please refer to “EJTAG Specification”, Section 6.3 “Optional *TRST** Pin” for more details.

#### 4.2.1.2 Optional *EDINT* pin

The *EDINT* input pin is also optional. An assertion of *EJ_DINT* in the 4K core triggers a Debug Interrupt Exception. This will stop the normal program flow within the 4K core, and force it to the Debug Exception Vector. The same effect can be achieved by setting the EjtagBrk Bit in the EJTAG Control Register. You access EJTAG Control Register through the TAP controller pins, but it will take many *ETCK* clock periods to do this.

The difference is that asserting the *EJ_DINT* input has much lower latency, and gives you faster control over forcing the processor into Debug Mode. If you do not need fast entry into Debug Mode, you can remove the *EDINT* pin from the chip.

*EJ_DINT* on the 4K core may also be connected to on chip logic, i.e. in a Multi-Core Breakpoint Unit (see Figure 4-5 on page 31for more details). You should only assert (high-active) the *EJ_DINTsup* (EJTAG Debug Interrupt Pin Supported) input on a 4K core if you have the *EJ_DINT* input connected to the *DINT* pin of the Probe Connector. You will not disable the *EJ_DINT* input if you de-assert the *EJ_DINTsup* input. *EJ_DINTsup* is only used to set the DINTsup bit in the EJTAG Implementation Register.

If you do not plan to connect *EJ_DINT* on the 4K core to an interrupt source, you must deassert both *EJ_DINT* and *EJ_DINTsup*, by connecting them to logic zero.

### 4.2.2 EJTAG Device ID input pins

The Device ID Register in the EJTAG TAP controller gets its values directly from the pins *EJ_ManufID[10:0]*, *EJ_PartNumber[15:0]* and *EJ_Version[3:0]*. If these pins are not already tied off to specific values by a hard core provider, the integrator is free to choose what values to place on *EJ_PartNumeber[15:0]* and *EJ_Version[3:0]*.

#### 4.2.2.1 *EJ_ManufID[10:0]*

*EJ_ManufID[10:0]* must be a compressed form of a JEDEC standard manufacturer's identification code. See "EJTAG Specification", section 5.5.2 "Device Identification Register".

#### 4.2.2.2 *EJ_PartNumber[15:0]*

*EJ_PartNumber[15:0]* is recomended to be a manufacturer specific number identifying this core as a MIPS 4K core. A new physical cache configuration could facilitate a new value on *EJ_PartNumber[15:0]*, but it could also be an increment of the number on the *EJ_Version[3:0]* input.

#### 4.2.2.3 *EJ_Version[3:0]*

*EJ_Version[3:0]* is recomended to be unique for each new physical layout, with the same *EJ_PartNumber[15:0]* input.

### 4.2.3 EJTAG Software Reset pins

Two reset-related EJTAG outputs are controlled by corresponding bits in the EJTAG Control Register. Peripheral Reset (*EJ_PerRst*) is controlled by the PerRst bit. And Processor Reset (*EJ_PrRst*) is controlled by the PrRst bit.

One other software reset-related pin is the Soft Reset Enable (*EJ_SRstE*). This pin is driven from the SRE bit in the Debug Control Register (The DCR is a memory mapped register present within the 4K core, which is accessible in Debug-Mode).

#### 4.2.3.1 *EJ_PrRst* pin

Processor Reset should really be interpreted as "System Soft Reset". When the PrRst bit is asserted, by EJTAG debug software, the result must be one of two possible scenarios:

1. The entire system is reset. This could be achieved by connecting *EJ_PrRst* to chip (internal or external) soft reset logic.
2. Nothing happens. Either *EJ_PrRst* is left unconnected, or the assertion is gated of by other logic like the *EJ_SRstE* pin.

A protocol exists, using the Rocc (Reset Occurred) bit, for debug software to identify which of the two scenarios occurs. Figure 4-4 shows one possible implementation for the use of *EJ_PrRst*.

#### 4.2.3.2 *EJ_PerRst* pin

Peripheral Reset can be used as a soft reset of the peripherals surrounding the 4K core. The effect of an asserted *EJ_PerRst* pin is implementation dependent. However it should never result in a reset of the 4K core itself. Figure 4-4 show one possible implementation of the use of *EJ_PerRst*.

#### 4.2.3.3 *EJ_SRstE* pin

As described earlier, this pin can be used to control one or more Soft Reset sources in the system reset logic. See Figure 4-4 for a possible implementation.

#### 4.2.3.4 One possible Reset Logic implementation

Figure 4-4 show a possible implementation of the *EJ_PrRst*, *EJ_PerRst* and *EJ_SRstE* pins in a system. Note that in this example all the Reset control logic is place outside the chip containing the 4K core. This requires 3 extra output pins, but this need not be the case in your system.

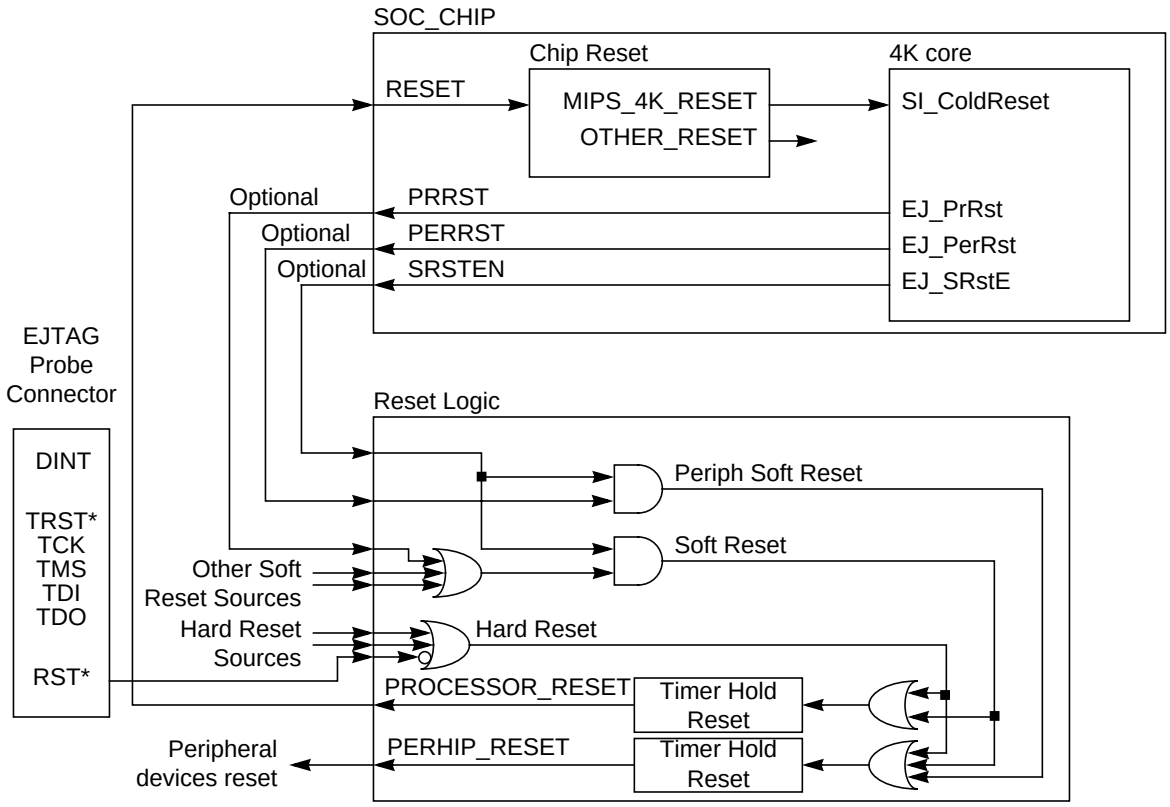


**Figure 4-4 Possible Reset circuitry implementation**

**Note:** The *RST** input to the Reset Logic from the Probe Connector is a required connection, when implementing EJTAG into your system.

## 4.3 Multi-Core implementation

In a chip configuration with multiple 4K cores, all the EJTAG TAP controllers can share one set of EJTAG TAP controller pins. The MIPS-recommended daisy-chain connection for a Multi-Core configuration is shown in Figure 4-5.

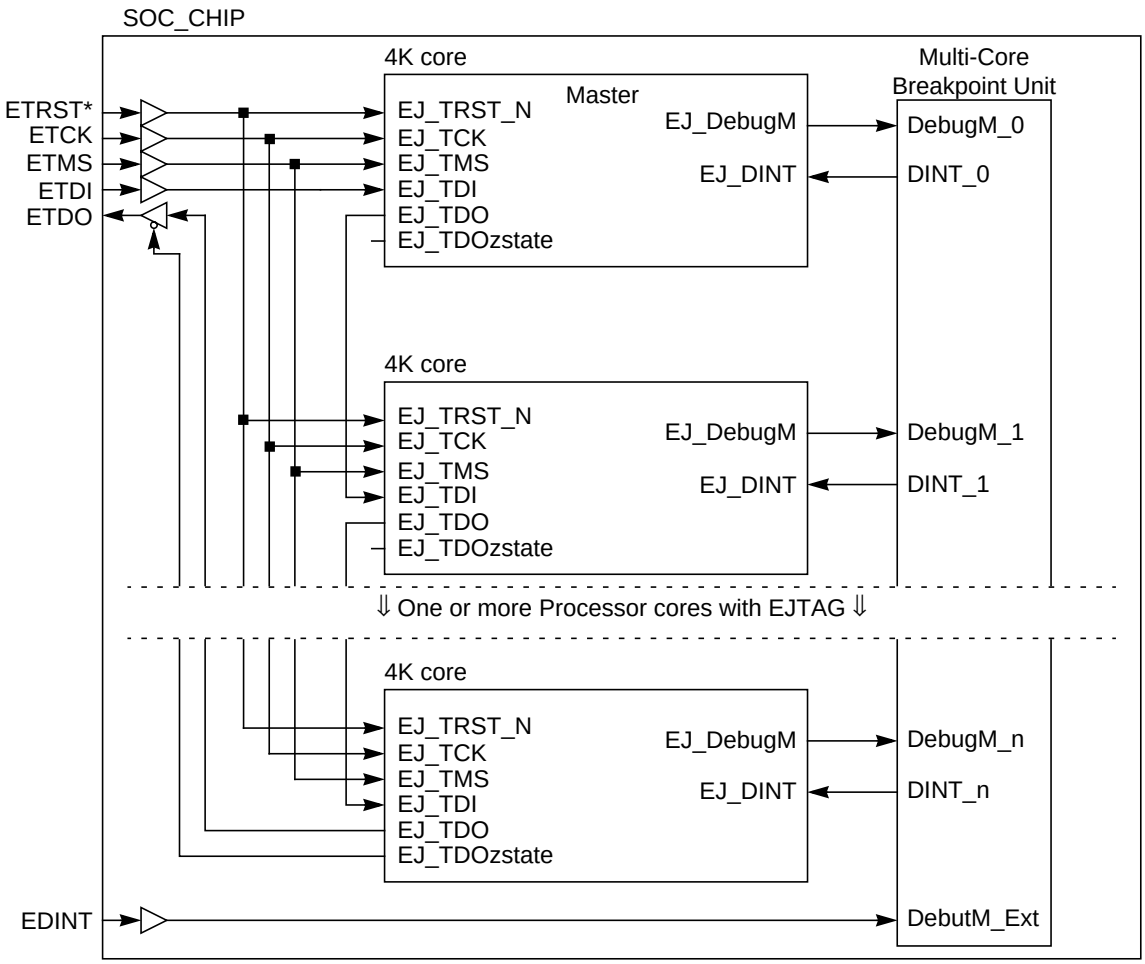


**Figure 4-5 Multi-Core Implementation**

### 4.3.1 *TDI/TDO* daisy-chain connection

In a Multi-Core implementation one of the processor cores will often be the Master. In Figure 4-5 the Master core has been put at first in the *TDI/TDO* daisy-chain. This is done to get a low latency access to control and data registers in the Master core. Only if a large number of EJTAG TAP controllers are connected in the daisy-chain, will the placement of the Master core be of any significance.

Output enable of the chip *ETDO*, is only controlled by *EJ_TDOzstate* of the last core in the chain. This must be so because this is the core which actually drive the *TDO* chip pin.

### 4.3.2 Multi-Core Breakpoint Unit

The Multi-Core Breakpoint Unit (MCBU) shown to the right in Figure 4-5, is an implementation-dependent block. The idea here is that each core could signal whether or not it is in Debug Mode, based on its *EJ_DebugM* output. When doing Multi-Core debug, a low latency entry into Debug Mode may be desired for all or some of the other processor cores on the chip, based on the entry of one of the processors into Debug Mode. For example, a slave core might rely on full operation by the master core; then the master core's entry into Debug Mode can trigger a Debug Interrupt (*EJ_DINT*) to the slave core(s). This would place each slave core in Debug Mode with low latency after the master core entered Debug Mode. (Depending on implementation, the latency would be less than 10 cycles.)

Debugger software can of course detect that the master core has entered Debug Mode, and then trigger this for the slave core(s) also. This might or might not be supported by your Debug software as an automatic feature. Further more the detection and the following slave core(s) debug trigger, would have to go through the serial TAP controller chain, which could take hundreds of cycles before the slave core(s) enter Debug Mode.

The physical implementation and/or programmability of the MCBU is a system decision beyond the scope of this document. However one thing to keep in mind, if you design an MCBU, is that the *EJ_DebugM* signal is a level sensitive signal and *EJ_DINT* is rising edge-triggered. Creating a *DINT_x* signal from a simple OR-function of one or more *DebugM_x* signals, will not have the desired effect. A rising edge detection on a *DebugM_x* output signal is needed to generate the desired rising edge on a *DINT_x* input signal. Once in Debug Mode, the 4K core will ignore any subsequent Debug Interrupts on *EJ_DINT.*

# Simulation Models

This chapter discusses the simulation models included in your MIPS32 4K™ core release. It contains the following sections:



## 5.1 Bus Functional Model

The MIPS32 4K™ core Bus Functional Model (BFM) is intended to provide the user with an abstracted version of the external interface of the 4K core. It's purposes is to provide a simple, highly controllable model of EC™ interface transactions. The BFM provides two mechanisms for managing transactions. A script file can be provided describing the transactions can be provided or the HDL testbench can make task calls to control the sequence of transactions. The following sections describe the function and interfaces for the BFM script sequencer, the HDL interface and the Jade core BFM verilog module.

Figure 5-1 shows the partitioning of the model components. Although the script sequencer and the task I/F are shown together, only one of them should be used with the bus functional model at any given time.

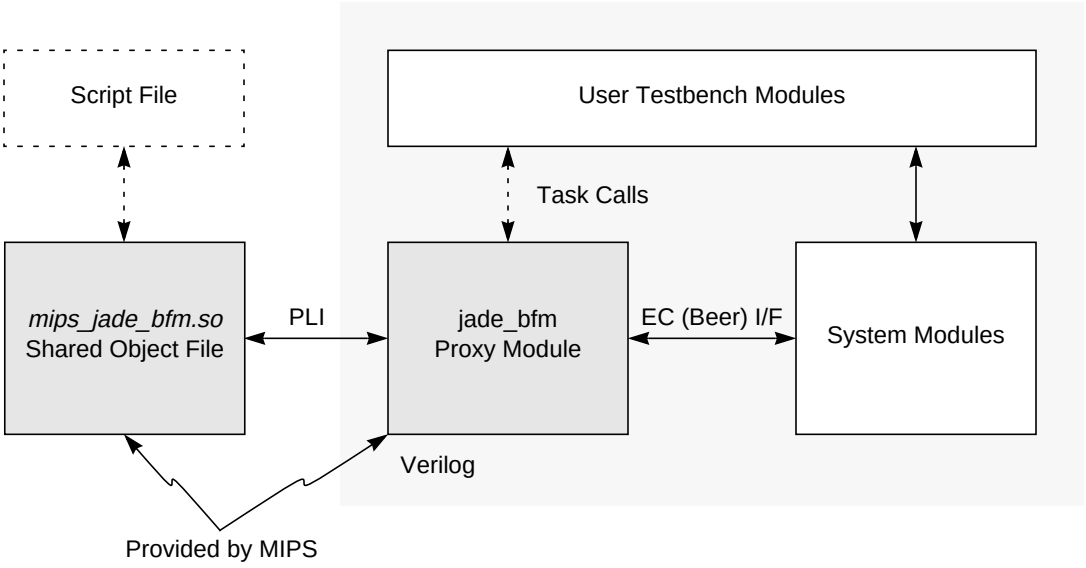


**Figure 5-1 Jade Bus Functional Model**

### 5.1.1 Installing and Using the BFM

The BFM script language and the Task I/F is described in a separate document *BFMUsersManual.pdf*, which is located in the `$JADEHOME/bfm` directory. Please refer to this document for further details on using the Jade BFM.

### 5.1.2 Simple Testbench

To simplify bring-up of the BFM, a simple testbench is included in the *$JADEHOME/bfm/verification* directory. This testbench can be used to verify that the BFM is installed correctly and shows examples of how to use it. The testbench ties off many of the Jade inputs not directly related to the memory access portion of the EC™ interface. It has a verilog memory that is loaded from the `test.hex` file. Random wait states are generated and reads and writes are done from/to the memory.

Two models are included in the `$JADEHOME/bfm/verification` directory. The file `jade_bfm.v` instantiates a single version of the BFM which executes commands from the bfm.script file. The file `dual_bfm.v` is an example of how to include multiple instances of the BFM to model a multi-processor system. Two instances of jade_bfm.v are instantiated. The file `dual_bfm.v` can also drive the BFM through the Task I/F. The verilog ‘define TASK controls whether the `jade_bfm.v` model uses the Task I/F or the script-based I/F.

The Makefile in `$JADEHOME/bfm/verification` provides targets for building either the single or dual BFM with a variety of simulators. The `Constlite.def` file contains a number of configuration options for the testbench. These should not need to be changed, but if you feel like experimenting, there is a brief description of what each of them does in the file.

## 5.2 Cycle-Exact Simulation Model

A VMC™ model is available if cycle-exact simulation is required. VMC is a tool from Synopsys that compiles RTL into a protected binary executable. This resulting executable can then be linked into a SWIFT R41 compatible RTL simulator to simulate a MIPS32 4K™ processor core.

### 5.2.1 Installing the VMC Model

1. Currently, the Jade VMC model is only supported on the Sun Solaris Unix platform. Contact MIPS Technologies, Inc. via email at “support@mips.com” if you require another platform.
2. The Jade VMC model is a SWIFT R-41 compatible model. This model can be loaded into a site-wide R41 LMC_HOME tree or into its own stand-alone LMC_HOME tree. As appropriate, set the LMC_HOME environment variable to the location you want the installation to reside:
   ```
   % setenv LMC_HOME <your_install_path>
   ```
3. Now invoke the admin install tool, which is supplied in the top level of the release package for the VMC model:
   ```
   % $JADEHOME/vmc/jade_vmc_release/sl_admin.csh
   ```
   1. A dialog box labeled “Install From...” should pop up.
   2. Make sure the text input box points to the package, “jade_vmc_release”.
   3. Press “Open” to continue.
   4. Now you should get another dialog box used to select the models that will be installed. You should only see one choice available in this release, a model called “jade_vmc_model” followed by a version number. Click on that model to bring it into the “Models to Install” window.
   5. Click “Continue” to close this dialog box.
   6. Next you’ll get another dialog box to select the platforms for this model installation. Since this release only supports the Sun Solaris platform, the platform default should be correct. You’ll also need to specify the appropriate simulator package you’ll be using under the “EDAV Packages” heading. If you’ll be using VCS as a simulator, then the default push-button selection of “Other” is appropriate. If your simulator is Verilog-XL, NC-Verilog, or ModelSim, then select the “Cadence Design Systems” push-button, as the support package needed for all of these simulators is identical. Or if you’re using one of the other simulators listed, choose that push-button. Then press “Install” to continue.

7. You should get an “Install complete” message in the main message window and you can exit from the sl_admin tool.

4. During the installation, a documentation directory will be created at `$LMC_HOME/doc`. There are pdf files in this directory structure which contain additional details about the install process, administering and using SmartModels, and licensing.

5. The 4K VMC model requires a GLOBEtrotter FLEXlm license in order to run. You can get this license from MIPS through your IP vendor. For details on how to install the license, see the “Network Licensing” chapter of `$LMC_HOME/doc/smartmodel/manuals/install.pdf`.

### 5.2.2 Verifying the VMC Installation

A utility called `swiftcheck` is available in the VMC installation to ensure that your model, environment variables and FLEXlm license key are set up properly. You should run this command before attempting to simulate with the 4K VMC model. Invocation is as follows:

```
% $LMC_HOME/bin/swiftcheck jade_vmc_model
```

The file `swiftcheck.out` will be produced by the command. You should check it to verify that there are no errors as reported at the end of the file.

### 5.2.3 SWIFT Template Generation

In order to instantiate the Jade VMC model in your RTL simulation environment, you need to create a SWIFT template of the 4K VMC model, which is then instantiated in your RTL design. This template file provides a conversion from the VMC model to your simulator’s SWIFT interface. The SWIFT template is simulator-specific, so your simulator documentation should provide additional details on creating a SWIFT template and including the template in your design.

To create a SWIFT template under Synopsys VCS, the following command can be used:

```
% vcs -lmc-swift-template jade_vmc_model
```

To generate a SWIFT template for Verilog-XL, NC-Verilog, and ModelSim, a script called `vsg` which is included in the `$LMC_HOME/bin` area of your installed VMC area is used. The invocation is:

```
% vsg -z jade_vmc_model
```

For reference, two SWIFT templates for the 4K VMC model are included in each release, under the directory `vmc/jade_vmc_release/readme/swift_template`. Templates are included for the VCS and Verilog-XL Verilog simulators in separate directories.

If you are using the `vsg` script to create your SWIFT template, the module it creates leaves the bits of a bus as individual ports in the input/output header and does not “busify” them. The instantiation of the SWIFT template is usually more convenient if the bits of a bus are concatenated together in the module’s port header. An example of the raw output from `vsg` is provided in the file `vmc/jade_vmc_release/readme/swift_template/jade_vmc_model.v`. An example of the `vsg` output which has been modified to concatenate bus bits in the port header is provided in the file `vmc/jade_vmc_release/readme/swift_template/jade_vmc_model.v.mod`. If you run `vsg` directly, however, you will need to perform the bus concatenation manually if you desire it for your SWIFT template.

The SWIFT template created by VCS (version 5.1 and later) automatically busifies the port header.

### 5.2.4 Back-annotating with SDF Timing

This is not currently supported.

### 5.2.5 Register Windows

To increase the visibility into the VMC model, a number of core signals are made available via register windows. This added information can make it easier to determine what the core is doing and help debug any integration/software problems. Table 5-1 shows the signals available via register windows.

**Table 5-1 Core signals visible in VMC model**

| Name | Size | Description |
|---|---|---|
| RFn | [31:0] | Contents of register n. Entries 1-31 of the register file are available. (Entry 0 is always 0). |
| CPZ_xxx | [31:0] | Contents of Coprocessor 0 register xxx. All possible 4K Cop0 registers are included, but TLB-related ones are not valid when using the Fixed Block Address Translation instead of the TLB. |
| InstnVA | [31:0] | Virtual Address for the Instruction Fetch. |
| InstnPA | [31:12] | Physical Address for the Instruction Fetch (bits [11:0] are untranslated and thus the same as the VA). |
| InstnCacheable | [0:0] | Indicates whether the Instruction Fetch is a cacheable reference. |
| ICacheHit | [0:0] | Indicates that Instruction reference hit in the I$. |
| InstnData | [31:0] | Instruction Data returned for Instruction Fetch. |
| DataVA | [31:0] | Virtual Address for the Load/Store reference. |
| DataPA | [31:12] | Physical Address for the Load/Store reference (bits [11:0] are untranslated and thus the same as the VA). |
| DataCacheable | [0:0] | Indicates whether the Load/Store reference is cacheable. |
| DCacheHit | [0:0] | Indicates that Load/Store reference hit in the D$. |
| LoadData | [31:0] | Load Data returned on a Load. |
| BusType | [2:0] | Indicates what type of Load/Store operation is occurring. Use to qualify DataVA etc.<br>0-No operation, 1-load, 2-store, 3-prefetch, 4-sync, 5-ICacheOp, 6-DCacheOp |
| BIU_LWptr | [3:0] | Bus Interface Unit read transaction tracking. LWptr is bumped every time a read address is accepted by the system. LRptr is bumped every time read data is returned. When LWptr != LRptr, the 4K core is waiting for read data to be returned. Useful for debugging system problems. Core “hangs” are often the result of a system not returning all requested data. |
| BIU_LRptr | [3:0] | See above. |

#### 5.2.5.1 Enabling VMC Window Signals in Synopsys VCS

Enabling the register window signals so they are visible is dependent on the simulator you are using. For Synopsys VCS, the register windows are globally enabled with the following code, which must be included somewhere in your testbench:

```
initial $swift_window_monitor_on(“<instance_path_to_jade_vmc_model>”);
```

#### 5.2.5.2 Enabling VMC Window Signals in Other Verilog Simulators

For Verilog-XL, NC-Verilog, and ModelSim, you need to individually specify every window signal you want to view. The code required is most easily placed in the SWIFT template produced by the `vsg` command, as described in Section 5.2.3 , "SWIFT Template Generation". The format of the enabling code is:

```
$lm_monitor_vec_map(<verilog_register>, “<instance_path_to_jade_vmc_model>”,
“<window_signal_name>”);
```

In the SWIFT template created by `vsg`, the <verilog_register> statements exist in the template but are dangling. You can use these dangling registers in the command required to enable each window signal. Here is an example of the code required to view some specific window signals:

```
initial
   begin
   $lm_monitor_vec_map(RF1, “<instance_path_to_jade_vmc_model>”, “RF1”);
   $lm_monitor_vec_map(RF2, “<instance_path_to_jade_vmc_model>”, “RF2”);
   ...
   end
```

The example `vsg`-generated template provided in `vmc/jade_vmc_release/readme/swift_template/verilog-xl/jade_vmc_model.v.mod` includes the full code needed to enable all the window signals, so you can look there as a reference.

### 5.2.6 VMC Simulation configuration

The VMC model is configurable so that all 4K™ cores can be run. The available options are shown in Table 5-2 on page 37 and include processor model (4Kc™, 4Km™, or 4Kp™ core), cache config, and configuration of optional EJTAG features. The configuration is done by setting up a memory file which is read in and used to select between the different modules. The memory file is called `memory.jade_config` and needs to be in a swift readmem format which is:

```
#Comment
<Address>/<Data>;
```

The available configuration options are shown in the following table.

**Table 5-2 VMC Configuration Options**

| Name: | Addr (hex) | Description | Legal Values | Default |
|---|---|---|---|---|
| ICacheAssoc | 1 | Associativity of the instruction cache. | 1,2,3,4 | 2 |
| ICacheWaySize | 2 | Size of each way of instruction cache (in KB). | 0(no I$), 1, 2, 4 | 4 |
| DCacheAssoc | 3 | Associativity of the data cache. | 1,2,3,4 | 2 |
| DCacheWaySize | 4 | Size of each way of data cache (in KB). | 0(no D$), 1, 2, 4 | 4 |
| InitCaches | 5 | Magically flush caches at time 0 to avoid simulation cycles for software cache initialization. | 0 - No Magic Init<br>1- Magic Init | 1 |
| BATMMU | 6 | Use Fixed Block Address Translation instead of TLB. | 0 - Use TLB (4Kc core)<br>1- Use Fixed MMU (4Km core / 4Kp core) | 0 |
| LITEMDU | 7 | Use smaller, iterative multiplier. | 0 - Fast MDU (4Kc core/4Km core)<br>1 - Iterative MDU (4Kp core) | 0 |
| EJSModule | 8 | Which ejtag simple break module should be used. | 0 - No SB<br>1 - 2I/1D SB<br>2 - 4I/2D SB | 2 |

**Table 5-2 VMC Configuration Options**

| Name: | Addr (hex) | Description | Legal Values | Default |
|---|---|---|---|---|
| EJTModule | 9 | Use ejtag TAP module. | 0 - No TAP<br>1 - Use TAP | 1 |
| Inst | A | Unique instance identifier. Tags output messages and trace files to more easily support multiple instances. | 0-63 | 0 |
| dispEn | B | Display Enable. Controls printing of warning or error messages coming from the VMC model. | 0 - No messages<br>1 - Messages | 1 |
| haltit | C | Controls stopping of VMC model. Determines which conditions will cause a $finish within the model. | 0 - Never stop<br>1 - Stop on FATAL errors<br>2 - Stop on any warning or error | 1 |
| bus_trace | D | Enables logging of all transactions on the cores EC™ interface (external bus). | 0 - No log<br>1 - Log bus transactions | 1 |
| dumpTrace | E | Enables instruction trace. | 0 - No tracing<br>1 - Trace file will be created | 1 |

An example `memory.jade_config` file is shown below:

```
# Memory Image File containing simulation configuration information
# Variable Number/Variable Value

#DCacheWaySize
4/2;
#ICacheWaySize
2/4;
#LITEMDU
7/0;
#BATMMU
6/0;
#EJSModule
8/2;
#EJTModule
9/1;
#DCacheAssoc
3/4;
#ICacheAssoc
1/4;
#InitCaches
5/0;
#Inst
A/0;
#dispEn
B/1;
#haltIt
C/1;
#bus_trace
D/1;
#dumpTrace
E/1;
```

### 5.2.7 Trace Files

The VMC model is capable of producing two types of trace files: a log of all transactions on the EC™ interface and a trace of all instructions executed.

#### 5.2.7.1 Bus Trace

The bus trace file (`vmc.bus(.Inst).trace`) contains information about all transactions on the EC interface. The fields are:

- Idle: Indicates how many idle cycles immediately preceded this transaction on the bus. For bursted transactions, the value for the first beat of the burst is used for all beats of the burst.
- Pipe: Indicates the pipeline depth - how many transactions were outstanding when this transaction started. Again, all beats of a burst reflect the value for the first beat of the burst.
- Type: Transaction type: RI- Instruction read, RD- Data read, W- Data write.
- Beat: Indicates which beat of the burst this is and the total length of the burst. “1 of 1x” indicates a non-bursted transaction. “3 of 4” indicates the 3rd beat of a 4 beat burst.
- EB_A<35:0>: Address value.
- EB_R/WData<31:0>: Read or Write data. The value in parentheses is the valid mask. A zero in any bit position indicates that there was an x in the corresponding bit of the data.
- BE<3:0>: Byte Enables - indicates which byte lanes are active for this transaction.
- Error: Indicates whether a bus error was signalled on this transaction.

- A wait states: Indicates the number of address wait states seen by this transaction.
- D wait states: Indicates the number of data wait states seen by this transaction.
- Cycle: Indicates a cycle number when this transaction completed. (Cycles are counted from the falling edge of the first Cold Reset). For bursts, all beats of the burst report the cycle that the burst completed.

#### 5.2.7.2 Instruction Trace

The instruction trace file (`vmc(.Inst).trace`) tracks the instruction flow in the processor. The architectural-visible effects of each instruction (register updates, memory writes, etc.) are also logged. The trace comes out in a raw format and is most easily read after a post-processing step. The `bin/rtlSort` script does this post-processing. It sorts the trace file to group all lines associated with a given instruction, adds instruction disassembly (using `bin/MIPSdis`) and slightly reformats the trace.

```
[Ins:4 Cyc:6 ]bfc00000 1fc00000 2:  00000000    NOP
|<-----a---->|<--------b-------->|<-------c------->|
a) Each line is tagged with an instruction number and a cycle number. Gaps in the
instruction number sequence can occur near exceptions. The cycle number reflects
the cycle at which the information was dumped. Most of the information is dumped
from a canonical point in the pipeline, so most of the lines for a given
instruction will have the same cycle number. The exception is the update of the
HI/LO registers in the MDU. Because the MDU pipeline can run independently from the
main pipeline, these register updates can be reported in a different cycle.
b) For instructions that do not take a fetch exception, the first line of the
instruction will be a fetch line. This field shows the hex values of the Virtual
Address, Physical Address, and Cache Coherency Attribute (CCA) for the instruction
fetch. On the 4K cores, only two of the eight CCA values are truly supported. When
simulating a 4Kc core, the entire CCA is not maintained in the ITLB, so the CCA for
mapped instruction addresses will always be reported as 2 (uncacheable) or 3
(cacheable)
c) This field is the instruction opcode and disassembly.
[Ins:954 Cyc:8166 ]Write GPR[26] = 80024230(ffffffff)
|<-------a------->|<------d------>|<-------e------->|
d) This indicates that the instruction caused a register update. Possible registers
are GPR[1-31] for the general purpose registers, HI and LO for the MDU registers,
and C0* for Coprocessor Zero registers.
e) This is the data value in hex. The value in parentheses is the valid mask. A 0
indicates that the corresponding bit in the data was an x. A dash in the data value
is used for sub-word loads and stores to indicate invalid bytes on the memory
read/write line.
[Ins:972 Cyc:8359 ]Mem Read [80024168 00024168 3] = 00000000(ffffffff)
|<--------a------>|<---f--->|<--------g-------->|<---------e-------->|
f) This is for memory accesses. Mem Read indicates a load that missed in the cache.
Cache Read indicates a load that hit in the cache. Mem Write indicates a store
```

```
(since the 4K cores have write-through caches, memory is always written so there is
no distinction for cache hit/miss).
g) This is the virtual address, physical address, and cache coherency algorithm for
the data access.
[Ins:187 Cyc:1978 ]Write TLB Entry[15]: PageMask(mask) = 00000000(ffffffff)
[Ins:187 Cyc:1978 ]      TLB Entry[15]: EnHi(mask)     = 80000000(ffffffff)
[Ins:187 Cyc:1978 ]      TLB Entry[15]: EnLo1(mask)    = 00000000(ffffffff)
[Ins:187 Cyc:1978 ]      TLB Entry[15]: EnLo0(mask)    = 00000000(ffffffff)
|<-------a------->|<------------------------h------------------------->|
h) A TLB write is shown on multiple lines indicating the TLB entry number and the
source registers for data being written into the entry.
```

### 5.2.8 Simple Testbench

To simplify bring-up of the VMC model, a simple testbench is included in the `$JADEHOME/vmc/verification` directory. This testbench can be used to verify that the VMC model is installed correctly and shows examples of how to use it. The testbench ties off many of the Jade inputs not directly related to the memory access portion of the EC™ interface. It has a verilog memory that is loaded from the `test.hex` file. The included `test.hex` has a simple boot sequence that executes a few instructions, then does a store to a trick box in the system model. When that store is seen, the system model does a $finish to stop the simulation.

In order to use the VMC model, you will need a verilog template. This template is specific to your simulator (including the particular version in some cases). There are directions for creating the template file in the file `$JADEHOME/vmc/jade_vmc_release/readme/README.txt`. There are two sample templates in the `verification` directory: `jade_vmc_model.vcs.v` is a template for vcs, and `jade_vmc_model.vxl.v` is a template for VerilogXL, ModelSim, and NC Verilog. These templates are slightly modified from how they were generated. Buses are broken up into bits by VMC. These templates have the bits of the buses reassembled so that the interface looks identical to the original RTL.

The Makefile in `$JADEHOME/bfm/verification` provides targets for building the VMC model in this testbench. Support for several simulators is included.

### 5.2.9 Multiple VMC Instances

It is possible to instantiate multiple 4K VMC models to simulate a multi-CPU system. The swift template file is parameterized to control which configuration file is read in. By reading a unique configuration file, each instance can be configured differently. By specifying unique instance tags in the memory file, the log output and trace files from the different models can be distinguished. The following example shows how this multiple instantiation can be accomplished. The following Verilog code will instantiate two VMC models, with instance names “vmc1” and “vmc2”, which will read the `memory1.jade_config` and `memory2.jade_config` configuration files respectively. Note that you must manually create the unique configuration files with the desired options for each instance, as described in Section 5.2.6 , "VMC Simulation configuration" on page 37.

```
jade_vmc_model vmc1 (....);
defparam vmc1.InstanceName = “vmc1”;
defparam vmc1.MemoryFile = “memory1”

jade_vmc_model vmc2 (...);
defparam vmc2.InstanceName = “vmc2”;
defparam vmc2.MemoryFile = “memory2”;
```

### 5.2.10 Assertion Checks

A variety of assertion checks are embedded within the 4K VMC model. These checkers look for error conditions and unknown state on critical signals. These checks are divided into a few basic categories:

- Fatal HW Errors - These errors should never occur and indicate a problem with the CPU. MIPS support (support@mips.com) should be contacted with the details of the problem.
- Fatal SW Errors - These errors indicate that the chip cannot proceed due to unknown state on internal signals. These errors can be caused by faulty software or incorrect chip hook up.
- XWarning - This indicates unknown state inside the chip from which it is theoretically possible to recover. Typically, these warnings will give a more descriptive message and better point to start debugging from than the eventual Fatal SW Error.
- I/O Warning - This indicates that the chip is possibly not hooked up correctly. For example, this will be flagged if the reset inputs are asserted for more than 2000 cycles. This is symptomatic of someone assuming that the reset inputs are active low rather than active high, but it might be the desired behavior in the system testbench or simulation environment. Thus these events are classified as warnings and not fatal errors.
- Fatal I/O Errors - These errors indicate illegal conditions on the primary I/O. Examples of this include undriven inputs or insufficient reset pulse width.

Recall that configuration options are available to enable or disable the display of these assertion messages, and to control whether or not a fatal error will stop simulation; see Section 5.2.6 , "VMC Simulation configuration" on page 37 for more details.

# Clocking, Reset & Power

This chapter describes the clocking and initialization interface on a MIPS32 4K™ processor core, when the core is integrated into a system environment. The power-reduction features available on a 4K™ core are also discussed.

## 6.1 Clocking

There are potentially two input clocks which must be generated and driven to a 4K core. The main clock input is named *SI_ClkIn*, and exists on every 4K core. An optional clock input is called *EJ_TCK*, and is only present if an EJTAG TAP controller is implemented within the core. Both clocks are used internally at 1x their respective input frequencies; no frequency multiplication or division is performed internally. No phase-locked loop is present within the 4K core. Typically no minimum frequency is required, so the frequency of the input clocks can be quickly changed or stopped if desired, as long as edge rate integrity is maintained.

The following discussion describes general clocking characteristics of a typical 4K core implemented with a standard ASIC physical design methodology. It is possible that a specific hard core implementation may differ from the general clock guidelines discussed here; e.g., dynamic circuit implementation techniques may mandate that a minimum clock frequency be met for a particular hard core. So the general clocking assumptions described here must be validated for the specific 4K core which is being integrated before proceeding with system clock design.

### 6.1.1 *SI_ClkIn* Clock

*SI_ClkIn* is the primary 1x input clock to the 4K core. It is used to enable the vast majority of sequential logic, as well as time the synchronous SRAMs normally used to implement the caches, within the 4K core.

Generally, only the positive edge of the *SI_ClkIn* clock is used internally to the core, so there is no specific duty cycle requirement. Transparent-low latches usually do exist within the core, so the duty cycle should still be within 40-60% of the period. Since no dynamic logic or PLL is present, the minimum frequency is 0 MHz; i.e., *SI_ClkIn* can be stopped if desired. The maximum *SI_ClkIn* frequency depends on the specific 4K core implementation.

### 6.1.2 *EJ_TCK* Clock

*EJ_TCK* is an optional 1x clock input to the 4K core, which only exists if the core implements an EJTAG TAP controller. *EJ_TCK* is the test input clock used to synchronize the serial shifting of data into and out of the TAP controller. The *EJ_TCK* clock is completely asynchronous to the *SI_ClkIn* clock, in terms of both frequency and phase.

The minimum frequency of *EJ_TCK* is 0 MHz, so it can be stopped when the TAP controller is not used. The maximum frequency is specified as 40 MHz (25 ns period), due to limitations of the probes which usually interface to the EJTAG TAP port. Both the rising and falling edges of *EJ_TCK* are used to control flops. The minimum clock high and low times are specified as 10 ns, yielding a duty cycle requirement of 40 to 60% at 40MHz.

### 6.1.3 Handling Clock Insertion Delay

When a 4K core is implemented, clock trees are usually created to buffer and distribute the *SI_ClkIn* and *EJ_TCK* clocks throughout the core. These clock trees impart a finite delay from the primary clock inputs to the eventual usage of the

buffered clocks at the sequential elements within the core. The exact amount of clock insertion delay is a characteristic of each specific 4K core implementation.

The clock insertion delay presents an issue which must be managed when the 4K core is instantiated in the rest of the system. Any clock insertion delay from the clock input to the actual clock usage at the sequential elements for the primary inputs and outputs of the core reduces the primary input setup times but increases the input hold times as well as the clock-> out delays on the primary outputs. Since all 4K core inputs are received directly by flops, and the core outputs come directly from flops, the setup and hold times for the primary inputs and outputs can be balanced at the system level.

Several different techniques can be used to manage the 4K core's internal clock insertion delay.

- Tolerate the core clock insertion delay at the system level, if possible, within the system logic which interfaces to the 4K core. This may entail adding delay elements when driving inputs, so that hold times are not violated, and receiving "late" outputs, which reduces the number of logic stages that can exist in the same cycle the outputs are driven since the clock insertion delay is visible. This may not be acceptable for all system designs, but is usually the simplest approach.
- When creating the system clock tree for the sequential logic which interfaces to the 4K core, match this system clock to the core's internal insertion delay. Clock tree generation tools have the ability to match relative clock delays, so knowing the core's internal clock insertion delay will allow the internal clocks to be specified as matching points (within reasonable skew limits). With this approach, input hold times and output delays can be minimized which allows more time in the cycle for useful work.
- Use the *SI_ClkOut* reference clock. *SI_ClkOut* is an output of the 4K core which is tapped from the internal clock tree so that it is identical (within reasonable skew limits) to the clock seen by the sequential elements within the 4K core. The difference between *SI_ClkIn* and *SI_ClkOut* represents the clock insertion delay of the primary clock used within the 4K core. (Note that there is no corresponding reference clock output for the *EJ_TCK* clock, so this technique cannot be applied to that clock domain.) Due to loading limitations, the *SI_ClkOut* clock probably can't be used directly to control system logic that interfaces to the core, but it can be used, for example, as the reference clock to a de-skewing phase-locked loop in the system to "hide" the core's clock insertion delay.

## 6.2 Reset and Hardware Initialization

Hardware initialization is accomplished through the *SI_ColdReset*, *SI_Reset* and *SI_NMI* input pins, and via the *EJ_TRST_N* pin if the optional EJTAG tap controller is present within the 4K core. This section describes how these pins are typically used in systems. These reset input pins must always be driven, either to a logic "1" or "0", to the 4K core, and not left floating or indeterminate. Each of the reset-related *SI_** inputs trigger a different type of exception within the 4K core; the *MIPS32 4K™ Processor Core Family Software User's Manual* describes more details about these exceptions.

The initialization process for a 4K core requires a combination of hardware and software. This section describes the basic hardware initialization interface. In accordance with the MIPS-32™ Architecture, only a minimal amount of state is reset by hardware; so much internal state, like the Translation Look-Aside Buffer (TLB) and the cache tag arrays, must be initialized via software before it can be used. The *MIPS32 4K™ Processor Core Family Software User's Manual* describes the software initialization requirements of a 4K core.

### *6.2.1 SI_ColdReset*

The *SI_ColdReset* input is a hard reset signal which initializes the internal hardware state of the 4K core without saving any state information. It is active high, and must be asserted for a minimum of 5 *SI_ClkIn* cycles. The falling edge triggers a reset exception which is taken by the core as the highest priority. Typically, *SI_ColdReset* is driven by a power-on-reset circuit in the system. For reliable operation, the power supply must be stable and the *SI_ClkIn* clock must be running before *SI_ColdReset* is deasserted.

### *6.2.2 SI_Reset*

The *SI_Reset* input is a warm reset input to the 4K core. It is active high, and must be asserted for a minimum of 5 *SI_ClkIn* cycles. The falling edge triggers a soft reset exception which is taken by the core. Typically, *SI_Reset* is driven by the OR of *SI_ColdReset* and the reset "button" in the system. Historically, MIPS processors have required Reset to be asserted during a ColdReset. The 4K cores do not require this, so an assertion of *SI_ColdReset* does not need to force the assertion of *SI_Reset.* For reliable operation, the power supply must be stable and the *SI_ClkIn* clock must be running before *SI_Reset* is deasserted.

### *6.2.3 SI_NMI*

The *SI_NMI* input signals a non-maskable interrupt (NMI). This signal is active high and rising edge sensitive, but must be asserted for a minimum of one clock cycle in order to be recognized. The sampling of the rising edge triggers an NMI exception to be taken by the core. Typically, *SI_NMI* is used to indicate time-critical information, like impending loss of power in the system.

### *6.2.4 EJ_TRST_N*

An additional reset signal is required when the EJTAG TAP controller is present. *EJ_TRST_N* is an active low reset signal that resets the TAP controller. This is an asynchronous reset and *EJ_TCK* does not need to be toggling for it to take effect. *EJ_TRST_N* must be asserted during power-on reset in order for the TAP controller and processor to be properly initialized. In general, the low-asserted pulse width should be the equivalent of at least one *EJ_TCK* cycle wide.

## 6.3 Power Management

Two primary mechanisms exist for managing system power with a 4K core: the hardware method of slowing down (or stopping) the primary *SI_ClkIn* clock and the software method of initiating "sleep" mode via the execution of the **WAIT** instruction.

### 6.3.1 Reducing *SI_ClkIn* frequency

The most global method of power control is to hold the primary *SI_ClkIn* input static, or at a lower frequency, when the 4K core is not in use, if desired by your system logic. The 4K core is internally fully static so the clock can be held either high or low, and the input frequency can be changed from maximum to a lower frequency, including zero, (and vice-versa) in a single cycle since there is no internal PLL.

The core outputs some pins which can be used, if desired, by the system logic to control entry or exit to this low-power state. The *SI_RP* output is directly driven from the internal CP0 Status register, as an external indication that it is desirable to place the 4K core in a low-power state by reducing the clock frequency. When the RP bit in the Status register is set by software, system logic can detect the assertion of the *SI_RP* output and choose to place the 4K core in a lower power state by reducing the clock frequency. Additionally, the *SI_ERL* and *SI_EXL* outputs, derived from the ERL and EXL bits in the Status register, indicate that an error or exception has been taken, and can be sensed to speed the clock frequency up again if desired. *EJ_DebugM* indicates that a debug exception has been taken. This can also be used speed the clock back up. These output pins need not be used to control the core's clock frequency, if other system logic is available to indicate that the 4K core is not being used.

### 6.3.2 Software-induced sleep mode

Upon execution of the software **WAIT** instruction, the 4K core will enter a low-power state once all outstanding bus activity has completed. Most of the clocks in the 4K core will be stopped, but a handful of flops will remain active to

sense an external hardware event which will awaken the core again. The external events which can wake the core back up are any enabled interrupt, NMI, debug interrupt (via *EJ_DINT*), or reset. Power is reduced since the global gated clock which goes to the vast majority of flops within the 4K core is held idle during this sleep mode. The *SI_Sleep* pin will be asserted when the core enters this low power mode. This can be used by the system logic to achieve further power savings. There will be no bus activity while the core is in sleep mode, so the system bus logic which interfaces to the 4K core could be placed into a low power state as well.

# Revision History

| Revision | Date | Who | Description |
|---|---|---|---|
| 0.9.4 | Dec 20, 1999 | | • Updated BFM references to be more in-line with the current BFM structure. |
| 0.9.5 | Jan 19, 2000 | | • Updated simulation model chapters with details on simple testbench.<br>• Fixed trademark usage.<br>• Added more text describing the *EB_WWBE/EB_EWBE* interface. |
| 01.00 | Jan 31, 2000 | | • Added *PM_DTLB{Hit,Miss}* signals.<br>• Updated text for *EB_WWBE/EB_EWBE*.<br>• Added simple testbench for BFM and VMC models.<br>• Updated references to *MIPS32 4K™ Processor Core Family Software User's Manual* to reflect name change. |
| 01.01 | Mar 21, 2000 | | • Changed *SI_TimerOut* to *SI_TimerInt* to correctly reflect the RTL name of this output.<br>• Describe new VMC features - better tracing and controllability.<br>• Pulled VMC installation information into this document from *README.txt*. |
| 01.02 | April 14, 2000 | | • Renamed *BFMScript.pdf* document to *BFMUsersManual.pdf* in the *bfm* subdirectory.<br>• Added discussion about handling multiple instances of the VMC model.<br>• Added general description of assertion messages which can emanate from the VMC model.<br>• Switched to a fixed-width font in the explanation of the VMC instruction trace file format. |

| Revision | Date | Who | Description |
|---|---|---|---|
| 01.03 | May 30, 2000 | | • Added description of the maximum outstanding transactions that will exist in a 4K core.<br>• Clarified instructions for “EDAV Packages” selection when installing the VMC model.<br>• Added description of prefixes used in names of system interface signals.<br>• Updated copyright notice to conform to latest standard. |
| 01.04 | July 18, 2000 | | • Reformatted cover sheet, added MD number.<br>• Added details about assertion of *EJ_TRST_N* in Clocking, Reset & Power chapter. |
| 01.05 | October 18, 2000 | | • Added Chapter 4, “EJTAG Interface.”<br>• Split System Interface chapter into two separate chapters for Chapter 2, “Signal Description,” and Chapter 3, “EC™ Interface.”.<br>• Modified timing diagrams in EC™ Interface chapter to reflect Jade-specific behavior on back to back transactions. |
| 01.06 | October 24, 2000 | Steve/ franzt | • Converted document to new template. |
| 01.07 | December 4, 2000 | | • Removed the EJ_PartNumber inputs from the Multicore figure.<br>• Updated Section 4.2.2 , "EJTAG Device ID input pins" on page 29.<br>• Added discussion about busification of module ports in SWIFT template for VMC model.<br>• Refer to the VMC model as “cycle-exact” instead of “cycle-accurate”.<br>• Added description about running `swiftcheck` to verify the VMC installation. |