MIPS32™ M4K™ Processor Core Integrator's Guide

Document Number: MD00248
Revision 01.03
January 9, 2003

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

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Copyright © 2002 MIPS Technologies, Inc. All rights reserved.

Unpublished rights (if any) reserved under the copyright laws of the United States of America and other countries.

This document contains information that is proprietary to MIPS Technologies, Inc. ("MIPS Technologies"). Any copying, reproducing, modifying or use of this information (in whole or in part) that is not expressly permitted in writing by MIPS Technologies or an authorized third party is strictly prohibited. At a minimum, this information is protected under unfair competition and copyright laws. Violations thereof may result in criminal penalties and fines.

Any document provided in source format (i.e., in a modifiable form such as in FrameMaker or Microsoft Word format) is subject to use and distribution restrictions that are independent of and supplemental to any and all confidentiality restrictions. UNDER NO CIRCUMSTANCES MAY A DOCUMENT PROVIDED IN SOURCE FORMAT BE DISTRIBUTED TO A THIRD PARTY IN SOURCE FORMAT WITHOUT THE EXPRESS WRITTEN PERMISSION OF MIPS TECHNOLOGIES, INC.

MIPS Technologies reserves the right to change the information contained in this document to improve function, design or otherwise. MIPS Technologies does not assume any liability arising out of the application or use of this information, or of any error or omission in such information. Any warranties, whether express, statutory, implied or otherwise, including but not limited to the implied warranties of merchantability or fitness for a particular purpose, are excluded. Except as expressly provided in any written license agreement from MIPS Technologies or an authorized third party, the furnishing of this document does not give recipient any license to any intellectual property rights, including any patent rights, that cover the information in this document.

The information contained in this document shall not be exported or transferred for the purpose of reexporting in violation of any U.S. or non-U.S. regulation, treaty, Executive Order, law, statute, amendment or supplement thereto.

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

MIPS, R3000, R4000, R5000 and R10000 are among the registered trademarks of MIPS Technologies, Inc. in the United States and other countries, and MIPS16, MIPS16e, MIPS32, MIPS64, MIPS-3D, MIPS-based, MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MIPSsim, SmartMIPS, MIPS Technologies logo, 4K, 4Kc, 4Km, 4Kp, 4KE, 4KEc, 4KEm, 4KEp, 4KS, 4KSc, 4KSd, M4K, 5K, 5Kc, 5Kf, 20Kc, 25Kf, ASMACRO, ATLAS, At the Core of the User Experience., BusBridge, CoreFPGA, CoreLV, EC, JALGO, MALTA, MDMX, MGB, PDtrace, Pipeline, Pro, Pro Series, SEAD, SEAD-2, SOC-it and YAMON are among the trademarks of MIPS Technologies, Inc.

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

Template: B1.08, Built with tags: 2B MIPS32 PROC

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table of Contents

Chapter 1 Overview 1
1.1 Environment Variable Setup 1

Chapter 2 Signal Description 2
2.1 Naming Conventions 2
2.2 Detailed Signal Descriptions 3

Chapter 3 SRAM-Style Interface 16
3.1 SRAM Interface Overview 16
3.1.1 Dual or Unified Interfaces 16
3.1.2 Backstalling 17
3.1.3 Redirection 17
3.1.4 Transaction Abort 17
3.1.5 MIPS16e Execution 17
3.1.6 Connecting to Narrower Devices 17
3.1.7 Lock Mechanism 18
3.1.8 Sync Mechanism 18
3.2 SRAM Interface Description 18
3.2.1 Overview of I-side (Dual or Unified Interface) 18
3.2.2 Overview of D-side (Dual Interface) 19
3.2.3 Overview of D-Side (Unified Interface) 21
3.2.4 Locking 21
3.2.5 Sync 22
3.2.6 SimpleBE Mode 22
3.2.7 External Write Buffer 23
3.3 SRAM Interface Timing Constraints 24
3.4 SRAM Interface Transactions 26
3.4.1 Simple Reads and Writes 26
3.4.2 MIPS16e Instruction Fetches 32
3.4.3 Redirection 33
3.4.4 Data Gathering 37
3.4.5 Sync 38
3.4.6 Bus Error 40
3.4.7 Abort 42
3.4.8 EJTAG Hardware Breakpoints 44
3.4.9 Lock 46

Chapter 4 EJTAG Interface 48
4.1 EJTAG versus JTAG 48
4.1.1 EJTAG Similarities to JTAG 48
4.1.2 Sharing EJTAG Resources with JTAG 49
4.2 How to Connect *EJ_** Pins 50
4.2.1 EJTAG Chip-Level Pins 51
4.2.2 EJTAG Device ID Input Pins 52
4.2.3 EJTAG Software Reset Pins 52
4.3 Multi-Core Implementations 53
4.3.1 *TDI/TDO* Daisy-Chain Connection 54
4.3.2 Multi-Core Breakpoint Unit 54
4.4 EJTAG Trace 55

Chapter 5 Coprocessor Interface 56
5.1 Introduction 56

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

5.2 Coprocessor Instructions 56
5.3 Signal Configuration 58
5.4 Interface Protocols 59
5.4.1 Instruction Dispatch 62
5.4.2 To Coprocessor Data Transfer 63
5.4.3 From Coprocessor Data Transfer 64
5.4.4 Condition Code Checking 65
5.4.5 Coprocessor Exceptions 66
5.4.6 Instruction Nullification 68
5.4.7 Instruction Killing 68
5.5 Power Saving Issues 69
5.5.1 No coprocessor Present 69
5.5.2 How to Use *CP2_idle* *69*
5.5.3 Gating the Clock to the Coprocessor 70
5.5.4 Using strobe signals as gating inputs on the sub-interfaces 70
5.6 Template for Coprocessor Modules 71

Chapter 6 VMC Simulation Model 72
6.1 Cycle-Exact Simulation Model 72
6.1.1 Installing the VMC Model 72
6.1.2 Verifying the VMC Installation 73
6.1.3 SWIFT Template Generation 73
6.1.4 Back-Annotating with SDF Timing 73
6.1.5 Register Windows 73
6.1.6 VMC Simulation Configuration 74
6.1.7 Trace Files 76
6.1.8 Simple Testbench 78
6.1.9 Multiple VMC Instances 78
6.1.10 Assertion Checks 79

Chapter 7 Clocking, Reset and Power 80
7.1 Clocking 80
7.1.1 *SI_ClkIn* Clock 80
7.1.2 *EJ_TCK* Clock 80
7.1.3 Handling Clock Insertion Delay 81
7.2 Reset and Hardware Initialization 81
7.2.1 *SI_ColdReset* *82*
7.2.2 *SI_Reset* *82*
7.2.3 *SI_NMI* *82*
7.2.4 *EJ_TRST_N* *82*
7.3 Power Management 82
7.3.1 Reducing *SI_ClkIn* Frequency 82
7.3.2 Software-Induced Sleep Mode 83

Chapter 8 Design For Test Features 84
8.1 Introduction 84
8.2 Scan Test 85
8.3 User-Specific RAM BIST 85

Appendix A References 86

Appendix B Revision History 87

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

List of Figures

Figure 3-1: Dual-Interface SRAM Block Diagram 16
Figure 3-2: Unified-Interface SRAM Block Diagram........ 17
Figure 3-3: External write buffer........ 24
Figure 3-4: Single Cycle Read 27
Figure 3-5: Single Cycle Write........ 28
Figure 3-6: Read with One Waitstate 29
Figure 3-7: Write with One Waitstate 30
Figure 3-8: Read followed by write (single cycle) 31
Figure 3-9: Read followed by write (one waitstate) 32
Figure 3-10: MIPS16e instruction fetches (single cycle, little endian mode) 33
Figure 3-11: Redirected read (single cycle) 34
Figure 3-12: Redirected read (one waitstate) 35
Figure 3-13: Redirected write (single cycle) 36
Figure 3-14: Redirected write (one waitstate) 37
Figure 3-15: Word read, data arriving bytewise........ 38
Figure 3-16: Sync (one waitstate)........ 39
Figure 3-17: Redirected sync (one waitstate) 40
Figure 3-18: Read with error indication (single cycle) 41
Figure 3-19: Read with error indication (one waitstate) 41
Figure 3-20: Aborted read (one waitstate)........ 42
Figure 3-21: Unsuccessful Abort attempt for write (single cycle) 43
Figure 3-22: Aborted write (multi cycle) 44
Figure 3-23: EJTAG data write break (one waitstate)........ 45
Figure 3-24: EJTAG data write break for Unified Interface (one waitstate)........ 46
Figure 3-25: Locking (single cycle) 47
Figure 4-1: Daisy-Chained *TDI-TDO* Between JTAG and EJTAG TAP Controllers 49
Figure 4-2: Multiplexing Between JTAG and EJTAG TAP Controllers 50
Figure 4-3: EJTAG Chip-Level Pin Connection 51
Figure 4-4: Reset Circuitry Implementation........ 53
Figure 4-5: Multi-Core Implementation 54
Figure 5-1: General Transfer Example 60
Figure 5-2: Instruction Dispatch Waveforms 63
Figure 5-3: To Coprocessor Data Waveforms........ 64
Figure 5-4: From Coprocessor Data Waveforms 65
Figure 5-5: Condition Code Check Waveforms 66
Figure 5-6: Exception Waveforms 67
Figure 5-7: Instruction Killing Waveforms 69
Figure 5-8: Use of SI_Sleep for Clock-Gating in the Coprocessor 70
Figure 5-9: Clock-Gating of To Data Registers in Coprocessor 71
Figure 5-10: Clock Gating of Instruction Registers in Coprocessor 71
Figure 8-1: Timing Diagram of Typical Scan Chain and Capture Operation 85

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

List of Tables

Table 2-1: Signal Type Key 2
Table 2-2: Signal Prefix Key 2
Table 2-3: Signal Descriptions 3
Table 3-1: Allowable Byte Enables in SimpleBE Mode 22
Table 3-2: Timing Constraints 24
Table 5-1: Supported Coprocessor 2 instructions 57
Table 5-2: Transfers Required for Each Dispatch 59
Table 5-3: Allowable Interface Latencies from a Coprocessor to the M4K Core 61
Table 5-4: Interface Latencies from the M4K Core to a Coprocessor 61
Table 6-1: Core Signals Visible in VMC model 73
Table 6-2: VMC Configuration Options 74
Table 8-1: Core Input Values for Major Operating Modes 84
Table B-1: Revision History 87

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Overview

This document is targeted for the ASIC designer who is integrating a version of a MIPS32™ M4K™ processor core into the system ASIC. This document is applicable to both those integrators who are using a hard core and those who are integrating a soft core.

In addition to this overview chapter, the document contains the following chapters:

- Chapter 2, “Signal Description,” on page 2 describes the pins of the core.
- Chapter 3, “SRAM-Style Interface,” on page 16 describes the SRAM interface protocol used by the core.
- Chapter 4, “EJTAG Interface,” on page 48 discusses the EJTAG interface used by the core, including the optional EJTAG TAP controller and the PDtrace interface.
- Chapter 5, “Coprocessor Interface,” on page 56 describes the Coprocessor 2 interface and protocol used by the core.
- Chapter 6, “VMC Simulation Model,” on page 72 describes models that can be used in place of the M4K core. One model is described in this chapter, a cycle-exact simulation model compiled with the Synopsys Verilog Model Compiler tool (VMC). The VMC model provides a cycle-exact model of a M4K 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 M4K core.
- Chapter 7, “Clocking, Reset and Power,” on page 80 covers issues related to handling the clock insertion delay of the M4K core. Additionally, the hardware reset requirements of the core, as well as power management techniques, are discussed.
- Chapter 8, “Design For Test Features,” on page 84 discusses general DFT features which may be preset on the M4K core. Details are specific to a particular implementation of the core.

1.1 Environment Variable Setup

Some UNIX paths described in the document refer to the *MIPS_PROJECT* environment variable, which should point to the top level of the M4K core deliverables. To set this variable:

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


Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Chapter 2

Signal Description

This chapter describes the signals on a MIPS32™ M4K™ processor core. Only naming conventions and actual signal names are listed in this chapter. The specific interface protocols to which each signal adheres are described in subsequent chapters.

This chapter contains the following sections:

- Section 2.1, "Naming Conventions"
- Section 2.2, "Detailed Signal Descriptions"

2.1 Naming Conventions

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

Table 2-1 Signal Type Key

Type	Description
In	Input to the core, unless otherwise noted, sampled on the rising edge of the appropriate clock signal.
Out	Output of the core, unless otherwise noted, driven at the rising edge of the appropriate clock signal.
AIn	Asynchronous inputs that are synchronized by the core.
SIn	Static input to the core. These signals control configuration options and are normally tied to either power or ground. They must not change state while *SI_ColdReset* is deasserted.
SOut	Static output from the core. These signals control configuration options in an optional connected Coprocessor 2. These signals are static and do not ever change state.

The names of interface signals present on a M4K core are prefixed with a unique string, according to their primary function. Table 2-2 defines the prefixes used for M4K core interface signals.

Table 2-2 Signal Prefix Key

Prefix	Description
*{I,D}S_*	Signals related to the SRAM-style interface.
*SI_*	General system interface signals, which are not part of the SRAM interface.
*EJ_*	Signals related to the EJTAG interface.
*TC_*	Signals related to the EJTAG Trace interface.
*CP2_*	Signals related to the Coprocessor 2 interface.
gscan/Bist	Signals related to design-for-test features, either scan or memory Built-In-Self-Test (BIST).

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

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

2.2 Detailed Signal Descriptions

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 7, “Clocking, Reset and Power,” on page 80 for more details		
Clock Signals: Refer to Section 7.1, "Clocking" on page 80 for more details		
SI_ClkIn	In	Clock input. All inputs and outputs, except a few of the EJTAG signals, are sampled or asserted relative to the rising edge of this signal.
SI_ClkOut	Out	Reference clock. This clock signal provides a reference for de-skewing any clock insertion delay created by the internal clock buffering in the core.
Reset Signals: Refer to Section 7.2, "Reset and Hardware Initialization" on page 81 for a description of the various types of reset.		
SI_ColdReset	AIn	Hard/Cold reset signal. Causes a Reset Exception in the core.
SI_NMI	AIn	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	AIn	Soft/Warm reset signal. Causes a SoftReset Exception in the core.
Power Management Signals: See Section 7.3, "Power Management" on page 82 for more details		
SI_ERL	Out	This signal reflects 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	Out	This signal reflects 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	Out	This signal reflects 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	Out	This signal is asserted by the core whenever the WAIT instruction is executed. The assertion of this signal indicates that the clock has stopped and that the core is waiting for an interrupt.
Interrupt Signals:		
SI_EICPresent	SIn	Indicates whether an external interrupt controller is present. Value is visible to software in the $Config3_{VEIC}$ register field.
SI_EISS[3:0]	In	General purpose register shadow set number to be used when servicing an interrupt in EIC interrupt mode.
SI_IAck	Out	Interrupt acknowledge indication for use in external interrupt controller mode. This signal is active for a single *SI_ClkIn* cycle when an interrupt is taken. When the processor initiates the interrupt exception, it loads the value of the *SI_Int[5:0]* pins into the $Cause_{RIPL}$ field (overlaid with $Cause_{IP7..IP2}$), and signals the external interrupt controller to notify it that the current interrupt request is being serviced. This allows the controller to advance to another pending higher-priority interrupt, if desired.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 2-3 Signal Descriptions (Continued)

Signal Name	Type	Description
SI_Int[5:0]	In/AIn	Active high Interrupt pins. These signals are driven by external logic and when asserted indicate an interrupt exception to the core. The interpretation of these signals depends on the interrupt mode in which the core is operating; the interrupt mode is selected by software. The *SI_Int* signals go through synchronization logic and can be asserted asynchronously to *SI_ClkIn*. In External Interrupt Controller (EIC) mode, however, the interrupt pins are interpreted as an encoded value, so they must be asserted synchronously to *SI_ClkIn* to guarantee that all bits are received by the core in a particular cycle. The interrupt pins are level sensitive and should remain asserted until the interrupt has been serviced. In Release 1 Interrupt Compatibility mode: • All 6 interrupt pins have the same priority as far as the hardware is concerned. • Interrupts are non-vectored. In Vectored Interrupt (VI) mode: • The *SI_Int* pins are interpreted as individual hardware interrupt requests. • Internally, the core prioritizes the hardware interrupts and chooses an interrupt vector. In External Interrupt Controller (EIC) mode: • An external block prioritizes its various interrupt requests and produces a vector number of the highest priority interrupt to be serviced. • The vector number is driven on the *SI_Int* pins, and is treated as a 6-bit encoded value in the range of 0..63. • When the core starts the interrupt exception, signaled by the assertion of *SI_IAck*, it loads the value of the *SI_Int[5:0]* pins into the $Cause_{RIPL}$ field (overlaid with $Cause_{IP7..IP2}$). The interrupt controller can then signal another interrupt.
SI_IPL[5:0]	Out	Current interrupt priority level from the $Status_{IPL}$ register field, provided for use by an external interrupt controller. This value is updated whenever *SI_IAck* is asserted.
SI_IPTI[2:0]	SIn	Indicates the *SI_Int* hardware interrupt pin that the timer interrupt pin (*SI_TimerInt*) is combined with external to the core. The value of this bus is visible to software in the $IntCtl_{IPTI}$ register field.
SI_SWInt[1:0]	Out	Software interrupt request. These signals represent the value in the *IP[1:0]* field of the *Cause* register. They are provided for use by an external interrupt controller.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 2-3 Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td>SI_TimerInt</td><td>Out</td><td>Timer interrupt indication. This signal is asserted whenever the Count and Compare registers match and is deasserted when the Compare register is written. This hardware pin represents the value of the $Cause_{TI}$ register field.

For Release 1 Interrupt Compatibility mode or Vectored Interrupt mode:

In order to generate a timer interrupt, the SI_TimerInt signal needs to be brought back into the M4K core on one of the six SI_Int interrupt pins in a system-dependent manner. Traditionally, this has been accomplished by muxing SI_TimerInt with SI_Int[5]. Exposing SI_TimerInt as an output allows more flexibility for the system designer. Timer interrupts can be muxed or ORed into one of the interrupts, as desired in a particular system. The SI_Int hardware interrupt pin with which the SI_TimerInt signal is merged is indicated via the SI_IPTI static input pins.

For External Interrupt Controller (EIC) mode:

The SI_TimerInt signal is provided to the external interrupt controller, which then prioritizes the timer interrupt with all other interrupt sources, as desired. The controller then encodes the desired interrupt value on the SI_Int pins. Since SI_Int is usually encoded, the SI_IPTI pins are not meaningful in EIC mode.</td></tr>
<tr><td colspan="3">Configuration Inputs:</td></tr>
<tr><td>SI_CPUNum[9:0]</td><td>SIn</td><td>Unique identifier to specify an individual core in a multi-processor system. The hardware value specified on these pins is available in the $EBase_{CPUNum}$ register field, so it can be used by software to distinguish a particular processor. In a single processor system, this value should be set to zero.</td></tr>
<tr><td>SI_Endian</td><td>SIn</td><td>Indicates the base endianness of the core. Value is visible to software in the $Config0_{BE}$ register field.

<table><tr><th>SI_Endian</th><th>Base Endian Mode</th></tr><tr><td>0</td><td>Little Endian</td></tr><tr><td>1</td><td>Big Endian</td></tr></table></td></tr>
<tr><td>SI_SimpleBE[1:0]</td><td>SIn</td><td>The state of these signals can constrain the core to only generate certain byte enables on SRAM interface writes. This eases connection to some existing bus standards. Value of SI_SimpleBE[0] is visible in the $Config0_{SB}$ register field. See Section 3-25, "Locking (single cycle)" on page 47 for more details.

<table><tr><th>SI_SimpleBE[1:0]</th><th>Byte Enable Mode</th></tr><tr><td>00_2</td><td>All BEs allowed</td></tr><tr><td>01_2</td><td>Naturally aligned bytes, halfwords, and words only</td></tr><tr><td>10_2</td><td>Reserved</td></tr><tr><td>11_2</td><td>Reserved</td></tr></table></td></tr>
<tr><td colspan="3">SRAM-style Interface: Refer to Chapter 3, “SRAM-Style Interface,” on page 16 for more details.</td></tr>
<tr><td colspan="3">The SRAM-style interface allows simple connection to fast, tightly-coupled memory devices.It can be configured with independent interfaces for Instruction and Data, or a Unified interface. Signals related to the I-side interface are prefixed with “IS_”; signals related to the D-side interface are prefixed with “DS_”. When the Unified interface is used, then most D-side signals are obsoleted, since they have an I-side equivalent; only the write data bus, DS_WData, continues to be used from the D-side.</td></tr>
<tr><td>DS_Read</td><td>Out</td><td>Read strobe.</td></tr>
<tr><td>DS_Write</td><td>Out</td><td>Write strobe.</td></tr>
<tr><td>DS_Sync</td><td>Out</td><td>Sync strobe.</td></tr>
</table>

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 2-3 Signal Descriptions (Continued)

Signal Name	Type	Description
DS_WbCtl	Out	Write buffer control. This signal is asserted when the M4K core can guarantee that no D-side read transaction will be started in the current clock cycle. For the purpose of generating this signal, if there is a pending transaction, the M4K core assumes that it will end in this cycle, in order to determine whether a new read transaction might be started or not. Unlike *DS_Read*, there is no asynchronous path from *DS_Stall* or any other input signal to *DS_WbCtl.* Also, it is an earlier signal than *DS_Read*. It is intended to be used by an external agent to control flushing of a write buffer (if a write buffer is present).
DS_Addr[31:2]	Out	Address of transaction. When *DS_Sync* is asserted high, *DS_Addr[10:6]* holds the “sync type” (the “stype” field of the SYNC instruction).
DS_BE[3:0]	Out	Byte enable signals for transaction. *DS_BE[3]* enables byte lane corresponding to bits *31:24*. *DS_BE[2]* enables byte lane corresponding to bits *23:16*. *DS_BE[1]* enables byte lane corresponding to bits *15:8*. *DS_BE[0]* enables byte lane corresponding to bits *7:0*.
DS_WData[31:0]	Out	Write data as defined by *DS_BE[3:0]*/*IS_BE[3:0]*. Used for both D-side and I-side transactions.
DS_Abort	Out	Request for transaction (read, write or sync) to be aborted, if possible. It is optional whether the external logic uses this signal or not, although using it may reduce interrupt latency. Completion of any transaction (aborted or not) is always communicated through *DS_Stall*. Whether the transaction was in fact aborted is signalled using *DS_AbortAck*. *DS_Abort* is asserted through (and including) the cycle where *DS_Stall* is deasserted.
DS_EjtBreakEn	Out	One or more EJTAG data breakpoints are enabled.
DS_EjtBreak	Out	Asserted when an EJTAG data break is detected. May be used by external logic to cancel the current transaction. This signal is asserted one cycle after the transaction start, so when precise breaks are required, the external logic must stall transactions by one cycle if *DS_EjtBreakEn* indicates that a break may occur. *DS_EjtBreak* is asserted through (and including) the cycle where *DS_Stall* is deasserted.
DS_Lock	Out	Asserted when a read transaction is due to an LL (load linked) instruction.
DS_Unlock	Out	Asserted when a write transaction is due to an SC (store conditional) instruction.
DS_Stall	In	Indicates that the transaction is not ready to be completed.
DS_Error	In	Valid in the cycle terminating the transaction (*DS_Stall* deasserted). Asserted high if transaction caused an error. Causes bus error exception to be taken by the core.
DS_AbortAck	In	Valid in the cycle terminating the transaction (*DS_Stall* deasserted). Asserted high if transaction was aborted. If no abort was requested (*DS_Abort* is low), and *DS_AbortAck* is asserted high in the cycle terminating the transaction, a bus error exception is taken.
DS_Redir	In	Valid in the cycle terminating the transaction (*DS_Stall* deasserted). Asserted high if transaction must be redirected to In-side.
DS_UnlockAck	In	Valid in the cycle terminating the transaction (*DS_Stall* deasserted). Result of *DS_Unlock* operation. Should be asserted high if system holds a lock on the address used for the write transaction (SC).

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 2-3 Signal Descriptions (Continued)

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

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 2-3 Signal Descriptions (Continued)

Signal Name	Type	Description
IS_Unlock	Out	Asserted when a write transaction is due to a redirected SC (store conditional) instruction.
IS_UnlockAll	Out	Asserted for one clock cycle when an ERET instruction is executed.
IS_Stall	In	Indicates that the transaction is not ready to be completed.
IS_Error	In	Valid in the cycle terminating the transaction (*IS_Stall* deasserted). Asserted high if transaction caused an error. Causes bus error exception to be taken by the core.
IS_AbortAck	In	Valid in the cycle terminating the transaction (*IS_Stall* deasserted). Asserted high if transaction was aborted. If no abort was requested (*IS_Abort* is low), and *IS_AbortAck* is asserted high in the cycle terminating the transaction, a bus error exception is taken.
IS_UnlockAck	In	Valid in the cycle terminating the transaction (*IS_Stall* deasserted). Result of *IS_Unlock* operation. Should be asserted high if system holds a lock on the address used for the redirected write transaction (SC).
IS_RData[31:0]	In	Read data.
IS_RBE[3:0]	In	Byte enable signals for *IS_RData[31:0]*. *IS_RBE[3]* enables byte lane corresponding to *IS_RData[31:24]*. *IS_RBE[2]* enables byte lane corresponding to *IS_RData[23:16]*. *IS_RBE[1]* enables byte lane corresponding to *IS_RData[15:8]*. *IS_RBE[0]* enables byte lane corresponding to *IS_RData[7:0]*.
EJTAG Interface: Refer to Chapter 4, “EJTAG Interface,” on page 48 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	In	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	In	Test Clock Input (*TCK*) for the EJTAG TAP.
EJ_TMS	In	Test Mode Select Input (*TMS*) for the EJTAG TAP.
EJ_TDI	In	Test Data Input (*TDI*) for the EJTAG TAP.
EJ_TDO	Out	Test Data Output (*TDO*) for the EJTAG TAP.
EJ_TDOzstate	Out	Drive indication for the output of *TDO* for the EJTAG TAP at chip level: 1: The *TDO* output at chip level must be in Z-state 0: The *TDO* output at chip level must be driven to the value of *EJ_TDO*. IEEE Standard 1149.1-1990 defines *TDO* as a 3-stated signal. To avoid having a 3-state core output, the M4K core outputs this signal to drive an external 3-state buffer.
Debug Interrupt:		
EJ_DINTsup	SIn	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	In	Debug exception request when this signal is asserted in a CPU clock period after being deasserted in the previous CPU clock period. The request is cleared when debug mode is entered. Requests when in debug mode are ignored.
Debug Mode Indication		

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 2-3 Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td>EJ_DebugM</td><td>Out</td><td>Asserted when the core is in Debug Mode. This can be used to bring the core out of a low power mode (see Section 7.3, "Power Management" on page 82 for more details). In systems with multiple processor cores, this signal can be used to synchronize the cores when debugging.</td></tr>
<tr><td colspan="3">Device ID Bits: These inputs provide an identifying number visible to the EJTAG probe. If the EJTAG TAP controller is not implemented, then 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.</td></tr>
<tr><td>EJ_ManufID[10:0]</td><td>SIn</td><td>Value of the Device ID$_{ManufID}$ register field. 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/

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.</td></tr>
<tr><td>EJ_PartNumber[15:0]</td><td>SIn</td><td>Value of the Device ID$_{PartNumber}$ register field.</td></tr>
<tr><td>EJ_Version[3:0]</td><td>SIn</td><td>Value of the Device ID$_{Version}$ register field.</td></tr>
<tr><td colspan="3">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.</td></tr>
<tr><td>EJ_SRstE</td><td>Out</td><td>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.</td></tr>
<tr><td>EJ_PerRst</td><td>Out</td><td>Peripheral Reset. EJTAG can assert this signal to request the reset of some or all of the peripheral devices in the system.</td></tr>
<tr><td>EJ_PrRst</td><td>Out</td><td>Processor Reset. EJTAG can assert this signal to request that the core be reset. This can be fed into the SI_Reset signal</td></tr>
<tr><td colspan="3">TCtrace Interface: These signals are the connected to the Trace Capture Block (TCB) inside the core. Except for the TC_ChipTrigIn and the TC_ChipTrigOut, all of the following pins will normally be connected to an on-chip Probe Interface Block (PIB). The PIB is placed close to the physical probe pins, and will handle the final off-chip transmission on the trace port.</td></tr>
<tr><td>TC_PibPresent</td><td>SIn</td><td>Must be asserted when a PIB is attached to the TC Interface. When de-asserted (low) all the other inputs are disregarded.</td></tr>
<tr><td>TC_TrEnable</td><td>Out</td><td>Trace Enable, when asserted the PIB must start the TR_Clk output running and can expect valid data on all other outputs.</td></tr>
<tr><td>TC_ClockRatio[2:0]</td><td>Out</td><td>Clock ratio. This is the software-set clock-ratio from the TCBCONTROLB$_{CR}$ register field. The value will be within the boundaries defined by TC_CRMax and TC_CRMin. The table below shows the encoded values for clock ratio.
<table>
<tr><th>TC_ClockRatio</th><th>Clock Ratio</th></tr>
<tr><td>000</td><td>8:1 (Trace clock is eight times the core clock)</td></tr>
<tr><td>001</td><td>4:1 (Trace clock is four times the core clock)</td></tr>
<tr><td>010</td><td>2:1 (Trace clock is double the core clock)</td></tr>
<tr><td>011</td><td>1:1 (Trace clock is same as the core clock)</td></tr>
<tr><td>100</td><td>1:2 (Trace clock is one half the core clock)</td></tr>
<tr><td>101</td><td>1:4 (Trace clock is one fourth the core clock)</td></tr>
<tr><td>110</td><td>1:6 (Trace clock is one sixth the core clock)</td></tr>
<tr><td>111</td><td>1:8 (Trace clock is one eighth the core clock)</td></tr>
</table></td></tr>
</table>

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 2-3 Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td>TC_CRMax[2:0]</td><td>SIn</td><td>Maximum Clock ratio supported. This static input sets the $TCBCONFIG_{CRMax}$ register field. It defines the capabilities of the PIB module. This field determines the maximum value of TC_ClockRatio.</td></tr>
<tr><td>TC_CRMin[2:0]</td><td>SIn</td><td>Minimum Clock ratio supported. This input sets the $TCBCONFIG_{CRMin}$ register field. It defines the capabilities of the PIB module. This field determines the minimum value of TC_ClockRatio.</td></tr>
<tr><td>TC_ProbeWidth[1:0]</td><td>SIn</td><td>This static input will set the $TCBCONFIG_{PW}$ register field. It specifies the number of actual data trace pins on the probe (4, 8 or 16).

If this interface is not driving a PIB module, but some chip-level TCB-like module, then this field should be set to 2’b11 (reserved value for PW).

<table>
<tr><th>TC_ProbeWidth</th><th>Number physical data pin on PIB</th></tr>
<tr><td>00</td><td>4 bits</td></tr>
<tr><td>01</td><td>8 bits</td></tr>
<tr><td>10</td><td>16 bits</td></tr>
<tr><td>11</td><td>Not directly to PIB</td></tr>
</table></td></tr>
<tr><td>TC_DataBits[2:0]</td><td>In</td><td>This input identifies the number of bits picked up by the probe interface module (PIB) in each “cycle”.

If TC_ClockRatio indicates a clock-ratio higher than 1:2, then clock multiplication in the Probe logic is used. The “cycle” is equal to each core clock cycle on SI_ClkIn.

If TC_ClockRatio indicates a clock-ratio lower than or equal to 1:2, then “cycle” is (clock-ratio * 2) of the core clock cycle. For example, with a clock ratio of 1:2, a “cycle” is equal to core clock cycle; with a clock ratio of 1:4, a “cycle” is equal to one half of core clock cycle.

This input controls the down-shifting amount and frequency of the trace word on TC_Data[63:0]. The bit width and the corresponding TC_DataBits value is shown in the table below.

<table>
<tr><th>TC_DataBits[2:0]</th><th>Probe uses following bits from TC_Data each cycle</th></tr>
<tr><td>000</td><td>TC_Data[3:0]</td></tr>
<tr><td>001</td><td>TC_Data[7:0]</td></tr>
<tr><td>010</td><td>TC_Data[15:0]</td></tr>
<tr><td>011</td><td>TC_Data[31:0]</td></tr>
<tr><td>100</td><td>TC_Data[63:0]</td></tr>
<tr><td>Others</td><td>Unused</td></tr>
</table>
This input might change as the value on TC_ClockRatio[2:0] changes.</td></tr>
<tr><td>TC_Valid</td><td>Out</td><td>Asserted when a new trace word is started on the TC_Data[63:0] signals. TC_Valid is only asserted when TC_DataBits is 100.</td></tr>
<tr><td>TC_Stall</td><td>In</td><td>When asserted, an new TC_Valid in the following cycle is stalled. TC_Valid is still asserted, but the TC_Data value and TC_Valid is kept static, until the cycle after TC_Stall is sampled low.

TC_Stall is only sampled in the cycle before a new TC_Valid cycle. And only when TC_DataBits is 100, indicating full word of TC_Data.</td></tr>
</table>

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 2-3 Signal Descriptions (Continued)

Signal Name	Type	Description
TC_Calibrate	Out	This signal is asserted when the Cal bit in *TCBCONTROLB* is set. For a simple PIB which only serves one TCB, this pin can be ignored. For a multi-core capable PIB which also uses *TC_Valid* and *TC_Stall*, the PIB must start producing the calibration pattern when this signal is asserted.
TC_Data[63:0]	Out	Trace word data. The value on this 64-bit interface is shifted down as indicated in *TC_DataBits[2:0]*. First cycle where a new TW is valid on all the bits and *TC_DataBits[2:0]* is 100, *TC_Valid* is also asserted. The Probe Interface Block (PIB) will only be connected to [(N-1):0] bits of this output bus. N is the number of bits picked up by the PIB in each core clock cycle. For clock ratios 1:2 and lower, N is equal to the number of physical trace pins (legal values of N are 4, 8, or 16). For higher clock ratios, N is larger than the number of physical trace pins.
TC_ProbeTrigIn	In	Rising edge trigger input. The source should be the Probe Trigger input. The input is considered asynchronous, i.e., double registered in the core.
TC_ProbeTrigOut	Out	Single cycle (relative to the “cycle” defined the description of *TC_DataBits*) high strobe, trigger output. The target of this trigger is intended to be the external probe’s trigger output.
TC_ChipTrigIn	In	Rising edge trigger input. The source should be on-chip. The input is considered asynchronous, i.e., double registered in the core.
TC_ChipTrigOut	Out	Single cycle (relative to core clock) high strobe, trigger output. The target of this trigger is intended to be an on-chip unit.
Coprocessor 2 Interface: Refer to Chapter 5, “Coprocessor Interface,” on page 56 for more details.		
Instruction Dispatch: These signals are used to transfer an instruction for the M4K core to the COP2 coprocessor.		
CP2_ir_0[31:0]	Out	**Coprocessor Arithmetic and To/From Instruction Word.** Valid in the cycle before *CP2_as_0*, *CP2_ts_0* or *CP2_fs_0* is asserted.
CP2_irenable_0	Out	**Enable Instruction Registering.** When deasserted, no instruction strobes will be asserted in the following cycle. When asserted, there *may* be an instruction strobe asserted in the following cycle. Instruction strobes include *CP2_as_0*, *CP2_ts_0*, *CP2_fs_0*. **Note:** This is the only late signal in the interface. The intended function is to use this signal as a clock gater on the capture latches in the coprocessor for *CP2_ir_0[31:0]*.
CP2_as_0	Out	**Coprocessor 2 Arithmetic Instruction Strobe.** Asserted in the cycle after an arithmetic Coprocessor 2 instruction is available on *CP2_ir_0[31:0]*. If *CP2_abusy_0* was asserted in the previous cycle, this signal may not be asserted. This signal must never be asserted in the same cycle that *CP2_ts_0* or *CP2_fs_0* is asserted.
CP2_abusy_0	In	**Coprocessor 2 Arithmetic Busy.** When asserted, a Coprocessor2 arithmetic instruction may not be dispatched. *CP2_as_0* can not be asserted in the cycle after this signal is asserted.
CP2_ts_0	Out	**Coprocessor 2 To Strobe.** Asserted in the cycle after a To COP2 Op instruction is available on *CP2_ir_0[31:0]*. If *CP2_tbusy* was asserted in the previous cycle, this signal will not be asserted. This signal can never be asserted in the same cycle that *CP2_as_0* or *CP2_fs_0* is asserted.
CP2_tbusy_0	In	**To Coprocessor 2 Busy.** When asserted, a To COP2 Op must not be dispatched. *CP2_ts_0* may not be asserted in the cycle after this signal is asserted.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 2-3 Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td>CP2_fs_0</td><td>Out</td><td>Coprocessor 2 From Strobe. Asserted in the cycle after a From COP2 Op instruction is available on CP2_ir_0[31:0]. If CP2_fbusy_0 was asserted in the previous cycle, this signal must not be asserted. This signal may never be asserted in the same cycle that CP2_as_0 or CP2_ts_0 is asserted.</td></tr>
<tr><td>CP2_fbusy_0</td><td>In</td><td>From Coprocessor 2 Busy. When asserted, a From COP2 Op may not be dispatched. CP2_fs_0 may not be asserted in the cycle after this signal is asserted.</td></tr>
<tr><td>CP2_endian_0</td><td>Out</td><td>Big Endian Byte Ordering. When asserted, the processor is using big endian byte ordering for the dispatched instruction. When deasserted, the processor is using little-endian byte ordering. Valid the cycle before CP2_as_0, CP2_fs_0 or CP2_ts_0 is asserted.</td></tr>
<tr><td>CP2_inst32_0</td><td>SOut</td><td>MIPS32 Compatibility Mode - Instructions. When asserted, the dispatched instruction is restricted to the MIPS32 subset of instructions. Please refer to the MIPS64™ architecture specification for a complete description of MIPS32 compatibility mode. Valid the cycle before CP2_as_0, CP2_fs_0 or CP2_ts_0 is asserted.

Note: The M4K core is a MIPS32 core, and will only issue MIPS32 instructions. Thus CP2_inst32_0 is tied high.</td></tr>
<tr><td>CP2_kd_mode_0</td><td>Out</td><td>Kernel/Debug Mode. When asserted, the processor is in kernel or debug mode. Valid the cycle before CP2_as_0, CP2_fs_0 or CP2_ts_0 is asserted.</td></tr>
<tr><td colspan="3">To Coprocessor Data: These signals are used when data is sent from the M4K core to the COP2 coprocessor, as part of completing a To Coprocessor instruction.</td></tr>
<tr><td>CP2_tds_0</td><td>Out</td><td>Coprocessor To Data Strobe. Asserted when To COP Op data is available on CP2_tdata_0[31:0].</td></tr>
<tr><td>CP2_torder_0[2:0]</td><td>SOut</td><td>Coprocessor To Order. Specifies which outstanding To COP Op the data is for. Valid only when CP2_tds_0 is asserted.

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

Note: The M4K core can never send Data Out-of-Order, thus CP2_torder_0[2:0] is forced to 000_2.</td></tr>
<tr><td>CP2_tordlim_0[2:0]</td><td>SIn</td><td>To Coprocessor Data Out-of-Order Limit. This signal forces the integer processor core to limit how much it can reorder To COP Data. The value on this signal corresponds to the maximum allowed value to be used on CP2_torder_0[2:0].

Note: The M4K core will never send Data Out-of-Order, thus CP2_tordlim_0[2:0] is ignored.</td></tr>
<tr><td>CP2_tdata_0[31:0]</td><td>Out</td><td>To Coprocessor Data. Data to be transferred to the coprocessor. Valid when CP2_tds_0 is asserted.</td></tr>
<tr><td colspan="3">From Coprocessor Data: These signals are used when data is sent to the M4K core from the COP2 coprocessor, as part of completing a From Coprocessor instruction.</td></tr>
</table>

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 2-3 Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td>CP2_fds_0</td><td>In</td><td>Coprocessor From Data Strobe. Asserted when From COP Op data is available on CP2_fdata_0[31:0].</td></tr>
<tr><td>CP2_forder_0[2:0]</td><td>In</td><td>Coprocessor From Order. Specifies which outstanding From COP Op the data is for. Valid only when CP2_fds_0 is asserted.
<table>
<tr><th>CP2_forder_0[2:0]</th><th>Order</th></tr>
<tr><td>000_2</td><td>Oldest outstanding From COP Op data transfer</td></tr>
<tr><td>001_2</td><td>2nd oldest From COP Op data transfer.</td></tr>
<tr><td>010_2</td><td>3rd oldest From COP Op data transfer.</td></tr>
<tr><td>011_2</td><td>4th oldest From COP Op data transfer.</td></tr>
<tr><td>100_2</td><td>5th oldest From COP Op data transfer.</td></tr>
<tr><td>101_2</td><td>6th oldest From COP Op data transfer.</td></tr>
<tr><td>110_2</td><td>7th oldest From COP Op data transfer.</td></tr>
<tr><td>111_2</td><td>8th oldest From COP Op data transfer.</td></tr>
</table>
Note: Only values 000_2 and 001_2 are allowed; see the CP2_fordlim_0[2:0] description below.</td></tr>
<tr><td>CP2_fordlim_0[2:0]</td><td>SOut</td><td>From Coprocessor Data Out-of-Order Limit. This signal sets the limit on how much the coprocessor can reorder From COP Data. The value on this signal corresponds to the maximum allowed value to be used on CP2_forder_0[2:0].

Note: The M4K core can handle one Out-of-Order From Data transfer. CP2_fordlim_0[2:0] is forced to 001_2. The core can also never have more than two outstanding From COP instructions issued, which also automatically limits CP2_forder_0[2:0] to 001_2.</td></tr>
<tr><td>CP2_fdata_0[31:0]</td><td>In</td><td>From Coprocessor Data. Data to be transferred from the coprocessor. Valid when CP2_fds_0 is asserted.</td></tr>
<tr><td colspan="3">Coprocessor Condition Code Check: These signals are used to report the result of a condition code check to the M4K core from the COP2. This is only used for BC2 instructions.</td></tr>
<tr><td>CP2_cccs_0</td><td>In</td><td>Coprocessor Condition Code Check Strobe. Asserted when coprocessor condition code check bits are available on CP2_ccc_0.</td></tr>
<tr><td>CP2_ccc_0</td><td>In</td><td>Coprocessor Conditions Code Check. Valid when CP2_cccs_0 is asserted. When asserted, the branch instruction checking the condition code should take the branch. When deasserted, the branch instruction should not branch.</td></tr>
<tr><td colspan="3">Coprocessor Exceptions: These signals are used by the COP2 to report exception for each instruction.</td></tr>
<tr><td>CP2_excs_0</td><td>In</td><td>Coprocessor Exception Strobe. Asserted when coprocessor exception signalling is available on CP2_exc_0 and CP2_exccode_0.</td></tr>
<tr><td>CP2_exc_0</td><td>In</td><td>Coprocessor Exception. When asserted, a Coprocessor exception is signaled on CP2_exccode_0[4:0]. Valid when CP2_excs_0 is asserted.</td></tr>
</table>

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 2-3 Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td>CP2_exccode_0[4:0]</td><td>In</td><td>Coprocessor Exception Code. Valid when both CP2_excs_0 and CP2_exc_0 are asserted.
<table>
<tr><th>CP2_exccode[4:0]</th><th>Exception</th></tr>
<tr><td>01010_2</td><td>(RI) Reserved Instruction Exception</td></tr>
<tr><td>10000_2</td><td>(IS1) Available for Coprocessor specific Exception</td></tr>
<tr><td>10001_2</td><td>(IS1) Available for Coprocessor specific Exception</td></tr>
<tr><td>10010_2</td><td>C2E Exception</td></tr>
<tr><td>All others</td><td>Reserved</td></tr>
</table></td></tr>
<tr><td colspan="3">Instruction Nullification: These signals are used by the M4K core to signal nullification of each instruction to the COP2 coprocessor.</td></tr>
<tr><td>CP2_nulls_0</td><td>Out</td><td>Coprocessor Null Strobe. Asserted when a nullification signal is available on CP2_null_0.</td></tr>
<tr><td>CP2_null_0</td><td>Out</td><td>Nullify Coprocessor Instruction. When deasserted, the M4K core is signalling that the instruction is not nullified. When asserted, the M4K core is signalling that the instruction is nullified, and no further transactions will take place for this instruction. Valid when CP2_nulls_0 is asserted.</td></tr>
<tr><td colspan="3">Instruction Killing: These signals are used by the M4K core to signal killing of each instruction to the COP2 coprocessor.</td></tr>
<tr><td>CP2_kills_0</td><td>Out</td><td>Coprocessor Kill Strobe. Asserted when kill signalling is available on CP2_kill_0[1:0].</td></tr>
<tr><td>CP2_kill_0[1:0]</td><td>Out</td><td>Kill Coprocessor Instruction. Valid when CP2_kills_0 is asserted.
<table>
<tr><th>CP2_kill_0[1:0]</th><th>Type of Kill</th></tr>
<tr><td>00_2</td><td rowspan="2">Instruction is not killed and results can be committed.</td></tr>
<tr><td>01_2</td></tr>
<tr><td>10_2</td><td>Instruction is killed.
(not due to CP2_exc_0)</td></tr>
<tr><td>11_2</td><td>Instruction is killed.
(due to CP2_exc_0)</td></tr>
</table>
If an instruction is killed, no further transactions will take place on the interface for this instruction.</td></tr>
<tr><td colspan="3">Miscellaneous COP2 signals:</td></tr>
<tr><td>CP2_reset</td><td>Out</td><td>Coprocessor Reset. Asserted when a hard or soft reset is performed by the integer unit.</td></tr>
<tr><td>CP2_present</td><td>SIn</td><td>COP2 Present. Must be asserted when COP2 hardware is connected to the Coprocessor 2 Interface.</td></tr>
<tr><td>CP2_idle</td><td>In</td><td>Coprocessor Idle. Asserted when the coprocessor logic is idle. Enables the processor to go into sleep mode and shut down the clock. Valid only if CP2_present is asserted.</td></tr>
<tr><td colspan="3">Scan Test Interface: These signals provide the interface for testing the core. The use and configuration of these pins are implementation-dependent.</td></tr>
<tr><td>gscanenable</td><td>In</td><td>This signal should be asserted while scanning vectors into or out of the core. The gscanenable signal must be deasserted during normal operation and during capture clocks in test mode.</td></tr>
</table>

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 2-3 Signal Descriptions (Continued)

Signal Name	Type	Description
gscanmode	In	This signal should be asserted during all scan testing both while scanning and during capture clocks. The *gscanmode* signal must be deasserted during normal operation.
gscanin_x	In	This signal is input to a scan chain. (x may be an integer greater than or equal to 0)
gscanout_x	Out	This signal is output from a scan chain. (x may be an integer greater than or equal to 0)
BistIn[n:0]	In	Input to the user-specified BIST controller
BistOut[n:0]	Out	Output from the user-specified BIST controller

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

SRAM-Style Interface

This chapter describes the SRAM-style interface, the primary external interface present on the MIPS32™ M4K™ processor core.

This chapter contains the following major sections:

- Section 3.1, "SRAM Interface Overview"
- Section 3.2, "SRAM Interface Description"
- Section 3.3, "SRAM Interface Timing Constraints"
- Section 3.4, "SRAM Interface Transactions"

3.1 SRAM Interface Overview

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

3.1.1 Dual or Unified Interfaces

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

The dual interface, shown in Figure 3-1, enables independent connection to instruction and data devices. It generally yields the highest performance, since the pipeline can generate simultaneous I and D requests which are then serviced in parallel.

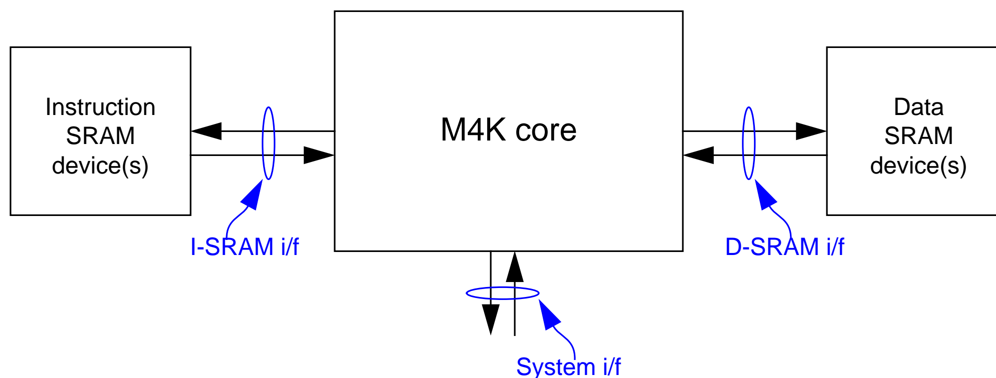

Figure 3-1 Dual-Interface SRAM Block Diagram

For simpler or cost-sensitive systems, it is also possible to combine the I and D interfaces into a shared interface that services both types of requests, as shown in Figure 3-2. If I and D requests occur simultaneously, priority is given to the D side.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

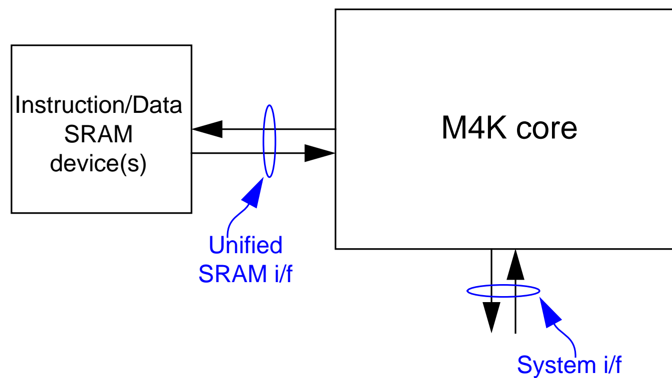

Figure 3-2 Unified-Interface SRAM Block Diagram

3.1.2 Backstalling

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

3.1.3 Redirection

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

3.1.4 Transaction Abort

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

3.1.5 MIPS16e Execution

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

3.1.6 Connecting to Narrower Devices

The instruction and data read buses are always 32-bits in width. To facilitate connection to narrower memories, the SRAM interface protocol includes input byte enables that can be used by system logic to signal validity as partial read data becomes available. The input byte enables conditionally register the incoming read data bytes within the core, and

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

thus eliminate the need for external registers to gather the entire 32-bits of data. External muxes are required to redirect the narrower data to the appropriate byte lanes.

3.1.7 Lock Mechanism

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

3.1.8 Sync Mechanism

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

3.2 SRAM Interface Description

As introduced in Section 3.1.1, "Dual or Unified Interfaces", the M4K core may be build with either separate I- and D-side interfaces or a single Unified Interface.

In case of the Dual Interface, a mechanism is available for the external agent to request a D-side transaction to be redirected to I-side. This redirection can be used for initializing a text segment (store instructions to I-space) or loading constants from a boot-ROM (loading data from the I-side). The external agent will typically decide whether to redirect a transaction based on its address. Note that the M4K core does not set any limitation on how to partition the address range. The I-side and D-side address ranges may even overlap.

In case of the Unified Interface, instruction fetches and load/store transactions share a common interface.

3.2.1 Overview of I-side (Dual or Unified Interface)

The I-side is used for fetching instructions. It is also used for redirected load/store transactions. In case of the Unified Interface, all load/store transactions are immediately redirected to I-side.

For the I-Side, the initial cycle of a transaction is indicated through assertion of one of three mutually-exclusive strobe signals:

- *IS_Read* (read transaction due to instruction fetch or redirected load).
- *IS_Write* (write transaction due to redirected store).
- *IS_Sync* (sync transaction due to redirected sync request, i.e. request to flush external I-side write buffers due to SYNC instruction, see also Section 3.2.5, "Sync").

The following buses/signals are also asserted:

- *IS_Instr* is asserted for read transactions due to instruction fetches.
- *IS_Addr[31:2]* holds the word-aligned address to be accessed in case of a read/write transaction, or the 5 bit “stype” field from the SYNC instruction (on *IS_Addr[10:6]*) in case of a sync transaction.
- *IS_BE[3:0]* selects the byte lanes to be accessed for a read/write transaction:
  - *IS_BE[3]* enables byte lane corresponding to bits 31:24.
  - *IS_BE[2]* enables byte lane corresponding to bits 23:16.
  - *IS_BE[1]* enables byte lane corresponding to bits 15:8.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

  - *IS_BE[0]* enables byte lane corresponding to bits 7:0.
- *DS_WData[31:0]* holds the data to be written in case of a redirected write transaction. This bus is shared with D-side.
- *IS_Lock* is asserted for read transactions caused by a redirected load linked (LL) instruction (see Section 3.2.4, "Locking").
- *IS_Unlock* is asserted for write transaction caused by a redirected store conditional (SC) instruction (see Section 3.2.4, "Locking").

The core may request a transaction to be aborted (e.g. in order to reduce interrupt latency). The core signals an abort request by asserting *IS_Abort*. It is optional whether the external agent aborts the transaction or not. It should only abort the transaction in case it is replayable. See also Section 3.4.7, "Abort".

An I-side EJTAG break is signalled using *IS_EjtBreak*. The external agent may choose to cancel the transaction in case precise breaks are required. This is however optional. *IS_EjtBreak* is asserted the cycle following the initial transaction cycle, so precise breaks will require transactions to be stalled for at least one cycle in case one or more breakpoints are enabled. *IS_EjtBreakEn* indicates whether I-side EJTAG breakpoints are enabled or not. For the Dual Interface, *IS_EjtBreak* is only asserted due to instruction breaks. For the Unified Interface, *IS_EjtBreak* may be asserted due to either instruction or data breaks. See also Section 3.4.8, "EJTAG Hardware Breakpoints".

All the above signals are held until the terminating cycle, i.e. the cycle where the external agent signals completion of the transaction by deasserting *IS_Stall*. *IS_Stall* must be valid starting in the clock cycle following the initial cycle. The initial cycle is the one where *IS_Read*/*IS_Write*/*IS_Sync* is first asserted.

A new transaction may be initiated immediately in the terminating cycle of the previous transaction. Note that this indicates there is a combinational path from *IS_Stall* to the start of a new transaction, for maximum performance. This is not the case for abort and EJTAG break signalling, since *IS_Abort* and *IS_EjtBreak* do not depend on *IS_Stall*. *IS_Abort* and *IS_EjtBreak* are kept asserted up to and including the terminating cycle.

A set of status signals must be driven by the external agent in the terminating cycle. These signals are only used in the terminating cycle; they are otherwise “don’t care” signals.

- *IS_Error* is asserted in case the transaction (read/write/sync) caused any kind of error. This will trigger a precise instruction fetch bus error in case of an instruction fetch or a precise data bus error exception in case of a redirected read/write/sync. See also Section 3.4.6, "Bus Error".
- *IS_AbortAck* is asserted in case the transaction (read/write/sync) was successfully aborted (following assertion of *IS_Abort*). *IS_AbortAck* is ignored if *IS_Error* is asserted. For redirected transactions, if *IS_Abort* was not asserted, *IS_AbortAck* assertion causes a data bus error exception.
- *IS_UnlockAck* is set in response to a write transaction with *IS_Unlock* asserted. Assertion of *IS_UnlockAck* indicates that the core did have a lock on the address, so the redirected store conditional (SC) completes successfully.

For read transactions, the external agent must supply the read data on *IS_RData[31:0]* (assuming the transaction is not redirected), while qualifying the individual byte lanes using *IS_RBE[3:0]*. Partial data gathering is supported as described in Section 3.4.4, "Data Gathering".

The M4K core drives one additional signal, *IS_UnlockAll*. This signal is asserted for one clock pulse whenever an ERET instruction is performed. This may be used by the external agent to unlock all addresses locked by the M4K core (see also Section 3.2.4, "Locking").

3.2.2 Overview of D-side (Dual Interface)

For the D-Side of the Dual Interface, the initial cycle of a transaction is indicated through assertion of one of three mutually-exclusive strobe signals:

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

- *DS_Read* (read transaction due to LW/LH/LB/LWL/LWR/LWC2 instructions).
- *DS_Write* (write transaction due to SW/SH/SB/SWL/SWR/SWC2 instructions).
- *DS_Sync* (sync transaction, i.e. request to flush external D-side write buffers due to SYNC instruction, see also Section 3.2.5, "Sync").

The following buses/signals are also asserted:

- *DS_Addr[31:2]* holds the word-aligned address to be accessed in case of a read/write transaction, or the 5 bit “stype” field from the SYNC instruction (on *DS_Addr[10:6])* in case of a sync transaction.
- *DS_BE[3:0]* selects the byte lanes to be accessed for a read/write transaction:
  - *DS_BE[3]* enables byte lane corresponding to bits 31:24.
  - *DS_BE[2]* enables byte lane corresponding to bits 23:16.
  - *DS_BE[1]* enables byte lane corresponding to bits 15:8.
  - *DS_BE[0]* enables byte lane corresponding to bits 7:0.
- *DS_WData[31:0]* holds the data to be written in case of a write transaction.
- *DS_Lock* is asserted for read transactions caused by a load linked (LL) instruction (see Section 3.2.4, "Locking")..
- *DS_Unlock* is asserted for write transaction caused by a store conditional (SC) instruction (see Section 3.2.4, "Locking").

The core may request a transaction to be aborted in case an interrupt occurs (in order to reduce interrupt latency). The core signals an abort request by asserting *DS_Abort*. It is optional whether the external agent aborts the transaction or not. It should only abort the transaction in case it is replayable. See also Section 3.4.7, "Abort".

An EJTAG data break is signalled using *DS_EjtBreak*. The external agent may choose to cancel the transaction in case precise breaks are required. This is however optional. *DS_EjtBreak* is asserted the cycle following the initial transaction cycle, so precise breaks will require transactions to be stalled for at least one cycle in case one or more breakpoints are enabled. *DS_EjtBreakEn* indicates whether data breakpoints are enabled or not. See also Section 3.4.8, "EJTAG Hardware Breakpoints".

All the above signals are held until the terminating cycle, i.e. the cycle where the external agent signals completion of the transaction by deasserting *DS_Stall*. *DS_Stall* must be valid starting in the clock cycle following the initial cycle. The initial cycle is the one where *DS_Read*/*DS_Write*/*DS_Sync* is first asserted.

A new transaction may be initiated immediately in the terminating cycle of the previous transaction. Note that this indicates there is a combinational path from *DS_Stall* to the start of a new transaction, for maximum performance. This is not the case for abort and EJTAG break signalling, since *DS_Abort* and *DS_EjtBreak* do not depend on *DS_Stall*. *DS_Abort* and *DS_EjtBreak* are kept asserted up to and including the terminating cycle.

A set of status signals must be driven by the external agent in the terminating cycle. These signals are only used in the terminating cycle; they are otherwise “don’t care” signals.

- *DS_Error* is asserted in case the transaction (read/write/sync) caused any kind of error. This will trigger a precise data bus error exception. See also Section 3.4.6, "Bus Error".
- *DS_AbortAck* is asserted in case the transaction (read/write/sync) was successfully aborted (following assertion of *DS_Abort*). *DS_AbortAck* is ignored in case *DS_Error* is asserted. If *DS_Abort* was not asserted, *DS_AbortAck* assertion causes a data bus error exception.
- *DS_Redir* is asserted in case the transaction must be redirected to the I-side. This will cause the transaction (read/write/sync) to be restarted on the I-side. *DS_Redir* is ignored in case *DS_Error* or *DS_AbortAck* are asserted. Note that in case the core attempts to abort a transaction due to an interrupt (by asserting *DS_Abort*), and the external agent signals that the transaction was not aborted (*DS_AbortAck* not asserted), the transaction will not be redirected

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

to I-side even if *DS_Redir* is asserted. Instead of performing the redirection in this case, the transaction will be canceled and the interrupt exception will be taken; typically, the interrupt service routine will return to the instruction that originated the D-side transaction and the transaction will be restarted and redirected this time (if no interrupt is pending). See also Section 3.4.3, "Redirection".

- *DS_UnlockAck* is set in response to a write transaction with *DS_Unlock* asserted. Assertion of *DS_UnlockAck* indicates that the core did have a lock on the address, so the store conditional (SC) completes successfully.

For read transactions, the external agent must supply the read data on *DS_RData[31:0]* (assuming the transaction is not redirected), while qualifying the individual byte lanes using *DS_RBE[3:0]*. Partial data gathering is supported as described in Section 3.4.4, "Data Gathering" on page 37.

3.2.3 Overview of D-Side (Unified Interface)

In case of the Unified Interface, all transactions are redirected immediately to the I-side. The only D-side signal used in this case is the shared write data bus, *DS_WData[31:0]*.

3.2.4 Locking

The following signals are used for the locking mechanism.

- *DS_Lock* (not used for Unified interface).
- *DS_Unlock* (not used for Unified interface).
- *DS_UnlockAck* (not used for Unified interface).
- *IS_Lock*
- *IS_Unlock*
- *IS_UnlockAll*
- *IS_UnlockAck*

DS_Lock / *IS_Lock* are asserted when a load linked (LL) operation is performed on D-side and I-side respectively. External logic may choose to register the corresponding address and, if it is currently unlocked, lock it.

Locking may imply one of the following:

- Prohibiting write to this address by another CPU.
- Monitoring whether such a write occurs and in this case set the status to unlocked.

The first approach will lock the address forever in case the application that caused the lock never unlocks it, or is never “task-switched” (in a multi-task operating system). Note that load linked may be performed by user mode applications.

DS_Unlock / *IS_Unlock* are asserted when a store due to a store conditional (SC) instruction is performed on D-side and I-side respectively. If the CPU currently has a lock on the address, it must be unlocked and *DS_UnlockAck* / *IS_UnlockAck* be asserted in the cycle terminating the transaction. If the CPU does not have a lock, *DS_UnlockAck* / *IS_UnlockAck* must be deasserted in the cycle terminating the access.

DS_UnlockAck / *IS_UnlockAck* will be used as the return value for the SC instruction.

A typical software sequence trying to access a shared resource might look like this:

```
// Set bit 0 of shared resource (register). Address is stored in t1.
1:  LL  t0, 0(t1)     // [IS|DS]_Lock is asserted.
    ORI t0, 1         // Set bit 0.
```


Copyright © 2002 MIPS Technologies Inc. All rights reserved.

```
SC  t0, 0(t1)      // [IS|DS]_Unlock is asserted.
BEQ t0, zero, 1b   // Result depends on internal lock bit and [IS|DS]_UnlockAck.
nop
```

Whenever an ERET instruction is performed, *IS_UnlockAll* is asserted for one cycle. In this case, external logic must unlock all addresses locked by the CPU. An ERET is typically issued for each task-switch performed by the operating system.

3.2.5 Sync

The MIPS32 SYNC instruction is used to guarantee that all read/write operations issued before the sync are finished before issuing any new read/write operations. In order to be able to flush external write buffers, SYNC is externalized using *DS_Sync* and *IS_Sync*.

Whenever a SYNC instruction occurs, *DS_Sync* is asserted (assuming Dual Interface). The external logic asserts *DS_Stall* until all write buffers have been flushed. The external logic may request that the sync operation be redirected to I-side in order to flush I-side write buffers. This is done using normal *DS_Redir* signalling.

For the Unified Interface, a SYNC instruction causes *IS_Sync* to be asserted immediately.

The MIPS32 SYNC instruction includes a 5 bit “stype” field optionally used for indicating one of several “sync types”. This field is externalized using *DS_Addr[10:6]* / *IS_Addr[10:6]*.

A sync operation may be aborted due to interrupts using the same *DS_Abort*/*IS_Abort* signalling as used for reads and writes.

DS_Error/*IS_Error* may be used to signal an error condition the same way as for reads and writes. An error indication will cause a bus error exception to be taken.

3.2.6 SimpleBE Mode

Individual load and store instructions can generate SRAM-interface transactions with byte enable patterns that are not directly supportable on other bus standards. To facilitate connection to these types of buses, the core has a mode where it will only generate bus transactions that are naturally aligned bytes, halfwords, or words. This is referred to as SimpleBE mode, selected when *SI_SimpleBE[1:0]* is set to 01_2. The default mode for the SRAM interface, in which the full range of byte enable combinations may occur, is selected when *SI_SimpleBE[1:0]* is set to 00_2. Note that the *SI_SimpleBE* bus is a static input which must be set to DC values at power-up of the core. The other two possible values of *SI_SimpleBE* are currently reserved and should not be selected.

Allowable byte enables in SimpleBE mode are shown in Table 3-1.

Table 3-1 Allowable Byte Enables in SimpleBE Mode

DS_BE[3:0] or IS_BE[3:0] (binary)
0001
0010
0100
1000
0011

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 3-1 Allowable Byte Enables in SimpleBE Mode

DS_BE[3:0] or IS_BE[3:0] (binary)
1100
1111

The only read operations that attempt to generate a complex byte enable combination result from LWL/LWR load instructions requesting a tri-byte from memory. Since external logic can easily convert a tri-byte read into a full word read if desired, no conversion is performed by the core for this case. Thus, tri-byte read byte enables will be requested on the interface, even in SimpleBE mode.

Tri-byte writes resulting from the SWL/SWR instructions can also attempt to generate complex byte enable combinations. When a tri-byte write transaction is detected internally in SimpleBE mode, the core will split the write into two separate transactions on the bus, each of which uses one of the byte enable values listed in Table 3-1. The first write will always contain a valid halfword, while the second will hold a valid byte.

3.2.7 External Write Buffer

Some system designs may include a write buffer between the M4K core and memory, as shown simplified on Figure 3-3 (shown for D-side of Dual Interface, but something similar might also apply to the I-side). A simple one entry write buffer here can help with several aspects of the system design while maintaining high system performance. If a multi-level memory is present, an address decode would need to be done to determine where a write should go. By always writing into the write buffer, the address decode can happen during the following cycle. Additionally, delaying the write by a cycle can keep EJTAG breaks precise. If a break is detected, the write buffer can be invalidated before memory is updated.

The write buffer contents can be written to memory opportunistically. The core has a signal, *DS_WbCtl,* to identify these opportunities. Assertion of this signal indicates that this clock cycle will NOT be the initial cycle of a read transaction. A read is the only core transaction that will need access to the SRAM. Since writes go to the write buffer, the previous write can be pushed out to the SRAM during a new write cycle. This allows back to back writes with no performance degradation and ensures that there will always be an opportunity to empty the write buffer.

If there is a pending transaction, the M4K core assumes that it will terminate in this cycle, in order to determine whether a new read transaction might be started or not. This eliminates asynchronous paths from core input signals (*DS_Stall* or others) to *DS_WbCtl* (unlike *DS_Read* or *DS_Write). DS_WbCtl* is also a slightly earlier signal than *DS_Read* or *DS_Write*.

The same mechanism is available at the I-side, through *IS_WbCtl*.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

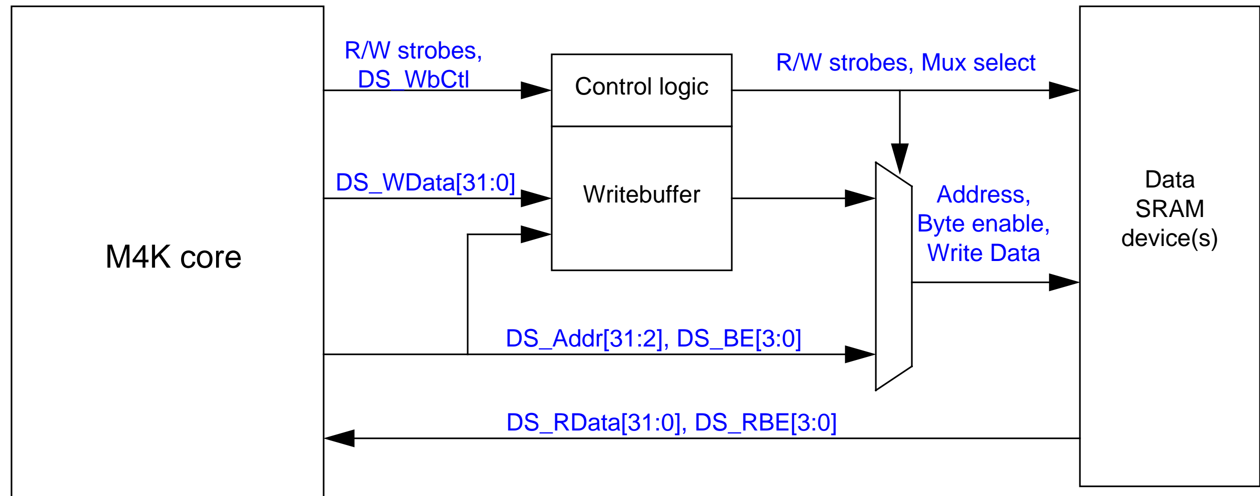

Figure 3-3 External write buffer

3.3 SRAM Interface Timing Constraints

All signals in the SRAM interface are synchronous to the rising edge of the primary input clock, *SI_ClkIn*. Many signals are not fully registered, however. Outputs on the SRAM interface may have a significant amount of logic after the preceding flop(s), and inputs may go through some combinational logic before being registered by the core. Also, there are combinational paths between some input signals and some output signals (for example, *DS_Stall* determines whether a new transaction may be started, so there is a combinational path from *DS_Stall* to *DS_Read*, *DS_Write* etc.). This situation complicates timing analysis associated with the core, but is necessary in order to achieve maximum performance of the interface.

The expression of timing constraints for the SRAM interface depends on many factors, such as maximum target frequency, process technology, standard cell library characteristics, setup/hold and access times for the SRAM devices, etc., so it is difficult to provide a generic set of timing guidelines that will apply in all situations. The "Timing Constraint" column in Table 3-2 shows the timing of SRAM interface signals, expressed as a percentage of the minimum target period, since most users are usually interested in achieving the maximum possible frequency of the core.

Many of the outputs arrive late in a cycle, so the external SRAM block can't perform much additional logic on them in the cycle they are driven, without adversely affecting the overall cycle time of the core. The *IS_Stall*, *DS_Stall*, and *DS_Redir* signals are particularly critical inputs to the core. Care must be taken in the amount of logic performed by the external SRAM block when driving these signals. For lower target frequencies, of course, the timing constraints shown in Table 3-2 can be relaxed.

Table 3-2 Timing Constraints

Signal Name	Type	Timing Constraint, as % of min. cycle
DS_Read	Out	90
DS_Write	Out	90
DS_Sync	Out	90
DS_WbCtl	Out	75

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

DS_Addr[31:2]	Out	90
DS_BE[3:0]	Out	90
DS_WData[31:0]	Out	90
DS_Abort	Out	90
DS_EjtBreakEn	Out	50
DS_EjtBreak	Out	90
DS_Lock	Out	90
DS_Unlock	Out	90
DS_Stall	In	25
DS_Error	In	25
DS_AbortAck	In	25
DS_Redir	In	25
DS_UnlockAck	In	80
DS_RData[31:0]	In	80
DS_RBE[3:0]	In	25
IS_Read	Out	90
IS_Write	Out	90
IS_Sync	Out	90
IS_WbCtl	Out	50
IS_Instr	Out	90
IS_Addr[31:2]	Out	90
IS_BE[3:0]	Out	90
IS_Abort	Out	90
IS_EjtBreakEn	Out	50
IS_EjtBreak	Out	90
IS_Lock	Out	90
IS_Unlock	Out	90
IS_UnlockAll	Out	90
IS_Stall	In	25
IS_Error	In	25
IS_AbortAck	In	25
IS_UnlockAck	In	80
IS_RData[31:0]	In	60
IS_RBE[3:0]	In	25

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

3.4 SRAM Interface Transactions

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

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

The net labeled “clk” shown in all timing waveforms is actually the *SI_ClkIn* primary clock input pin to the core.

3.4.1 Simple Reads and Writes

This section describes several basic read and write transactions.

3.4.1.1 Single Read

Figure 3-4 illustrates the fastest read, a single cycle D-side read operation. The transaction is initiated by the core in cycle 1, as it asserts the read strobe (*DS_Read*), as well as the desired word address (*DS_Addr[31:2]*) and output byte enables (*DS_BE[3:0]*). The byte enables represent the lower two bits of the address, as well as the requested data size, and identify which of the four byte lanes on *DS_RData* in which the core expects the read data to be returned.

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

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

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

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

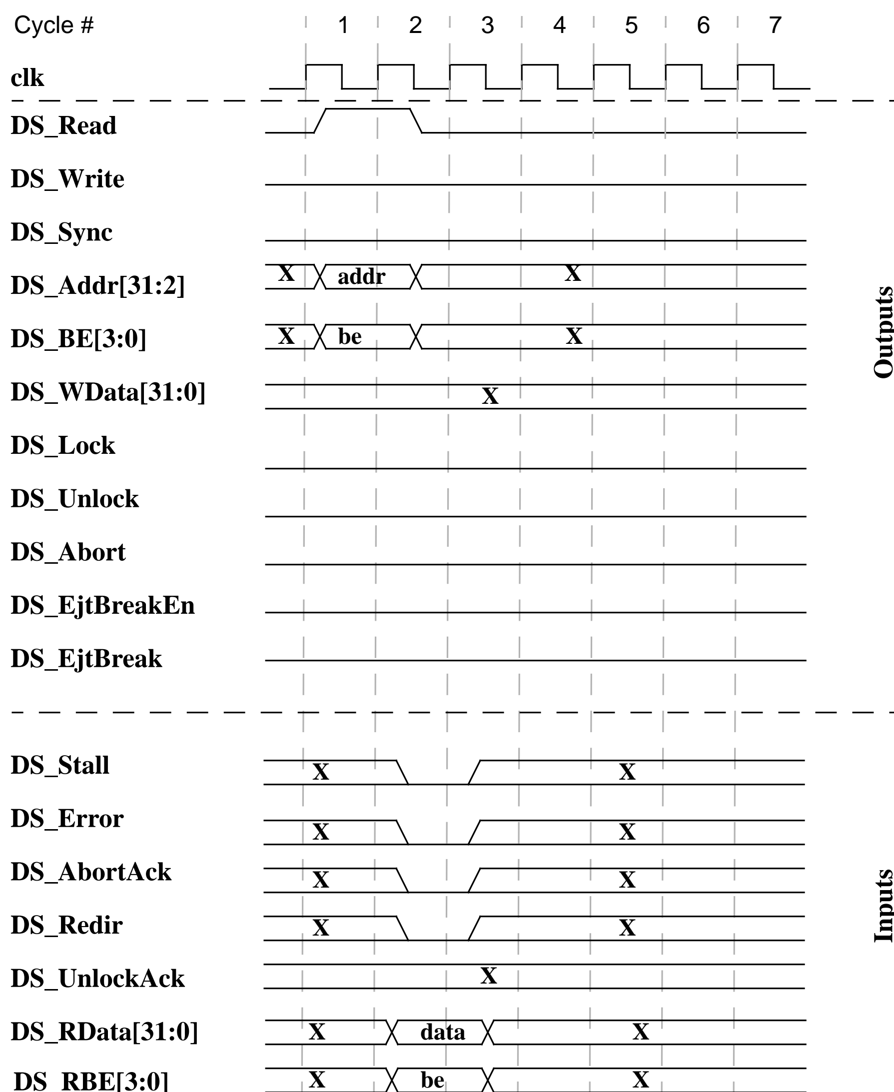

Figure 3-4 Single Cycle Read

3.4.1.2 Single Write

Figure 3-5 illustrates the fastest write, a single cycle D-side write operation. The transaction is initiated by the core in cycle 1, as it asserts the write strobe (*DS_Write*), as well as the desired word address (*DS_Addr[31:2]*), write data (*DS_WData[31:0]*), and output byte enables (*DS_BE[3:0]*). The byte enables identify which of the four byte lanes in *DS_WData* hold valid write data.

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

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

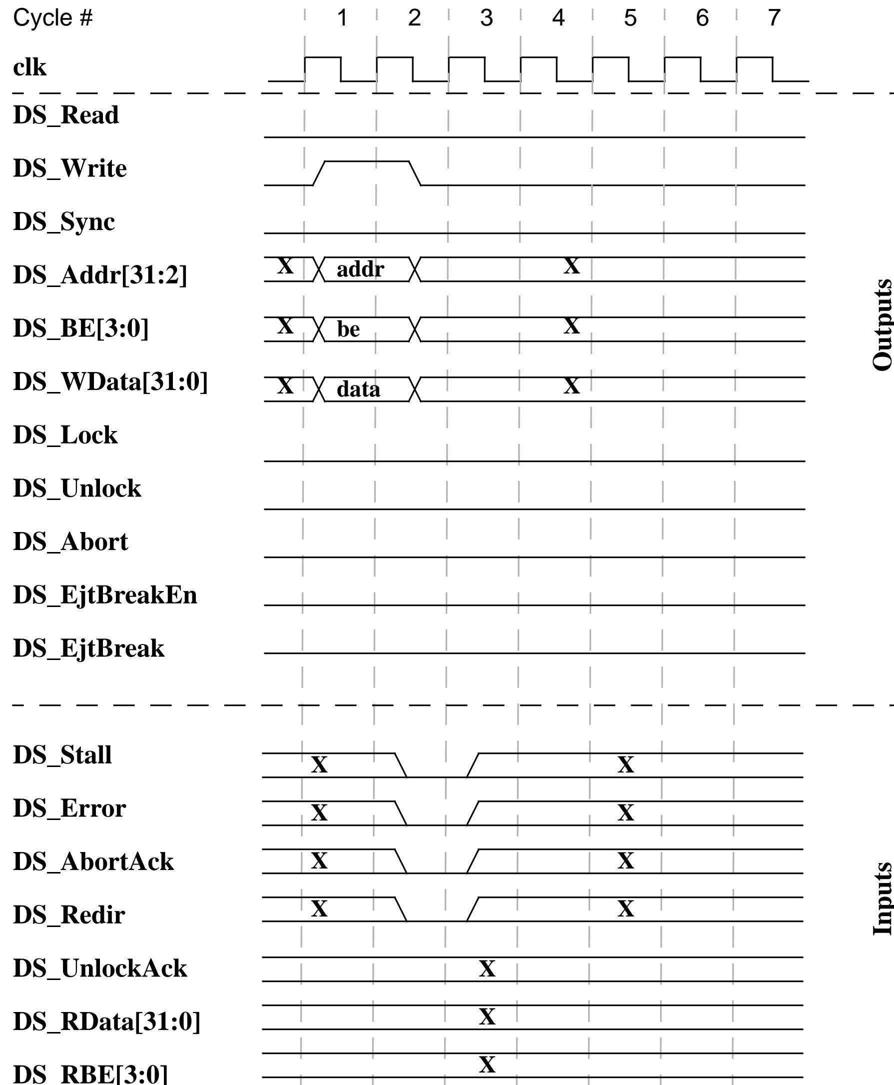

Figure 3-5 Single Cycle Write

3.4.1.3 Read with Waitstate

Figure 3-6 illustrates a D-side read operation with a single waitstate. This transaction is similar to the single-cycle read in Figure 3-4, only now a stall (*DS_Stall*) is asserted for one cycle and the read data is returned a cycle later.

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

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

Also during a stall (cycle 2 in this example), the input byte enables (*DS_RBE[3:0]*) will continue to control which byte lanes on the read data bus (*DS_RData[31:0]*) are registered by the core. To reduce power consumption, an external agent may want to deassert the input byte enables during a long stall, and then the core won't see spurious changes on the read data bus. But it is functionally acceptable to just tie the input read byte enables high all the time, for simplicity.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

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

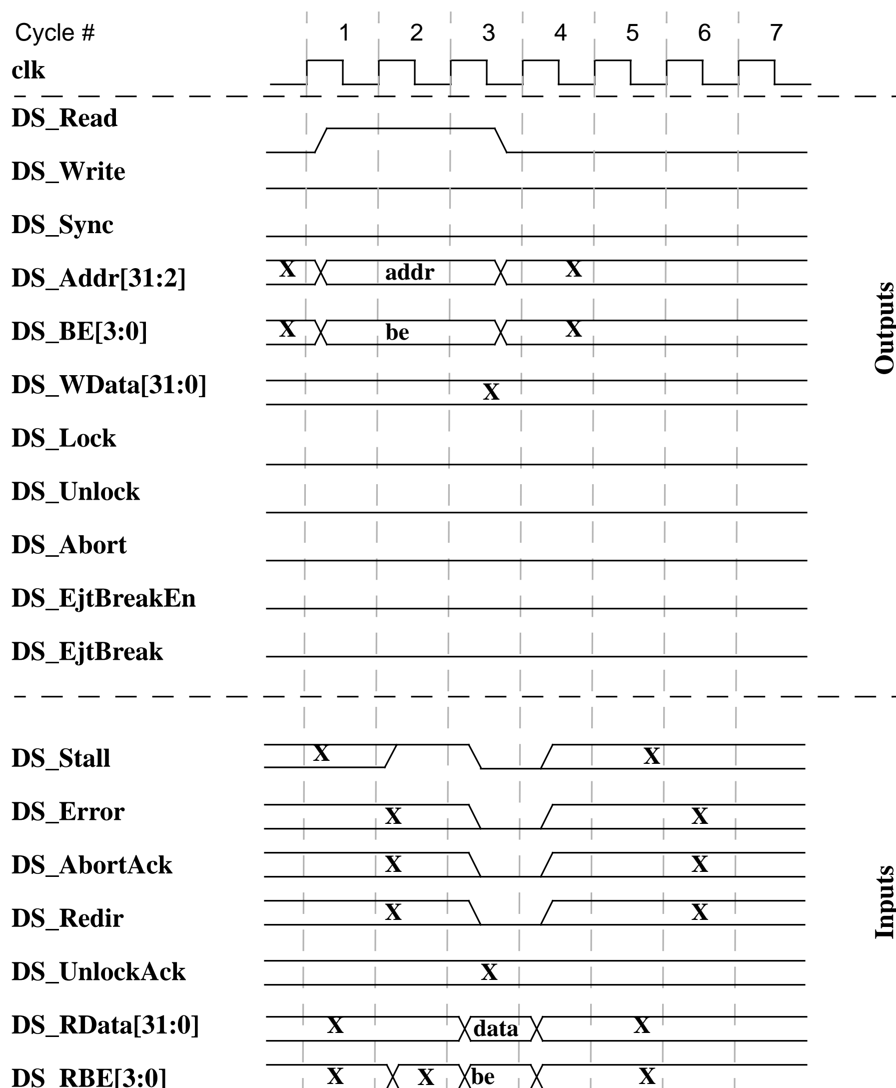

Figure 3-6 Read with One Waitstate

3.4.1.4 Write with Waitstate

Figure 3-7 illustrates a D-side write operation with a single waitstate. This transaction is similar to the single-cycle write in Figure 3-5, only now a stall (*DS_Stall*) is asserted for one cycle and the write is completed a cycle later.

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

The external agent cannot acknowledge the write immediately for some reason, so it asserts *DS_Stall* in cycle 2. The core outputs are held valid through the stall. Finally in cycle 3, the write can be accepted, so *DS_Stall* deasserts, and the error and redirection signals also deassert to indicate a normal completion.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

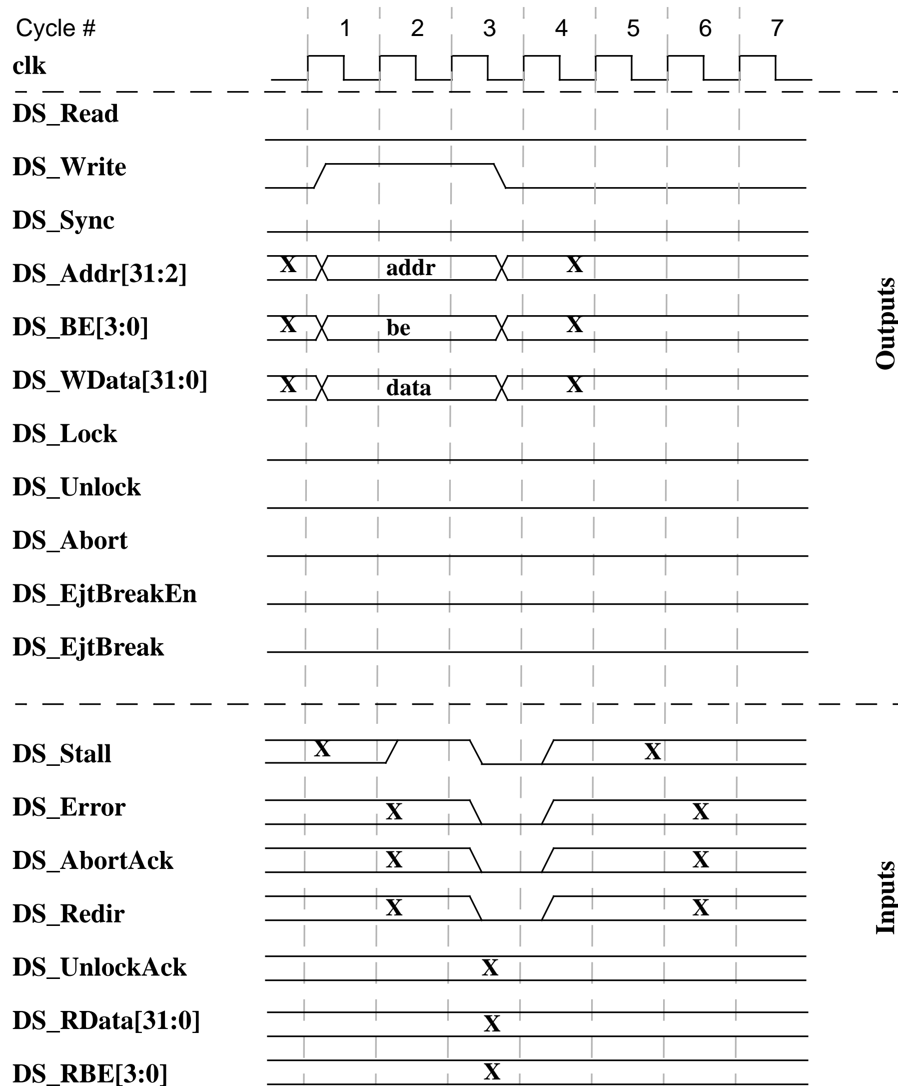

Figure 3-7 Write with One Waitstate

3.4.1.5 Read Followed by Write

Figure 3-8 illustrates a single cycle D-side read operation followed immediately by a single cycle D-side write operation. This example represents the back-to-back concatenation of the single-cycle read shown in Figure 3-4 with the single cycle write from Figure 3-5.

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

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

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

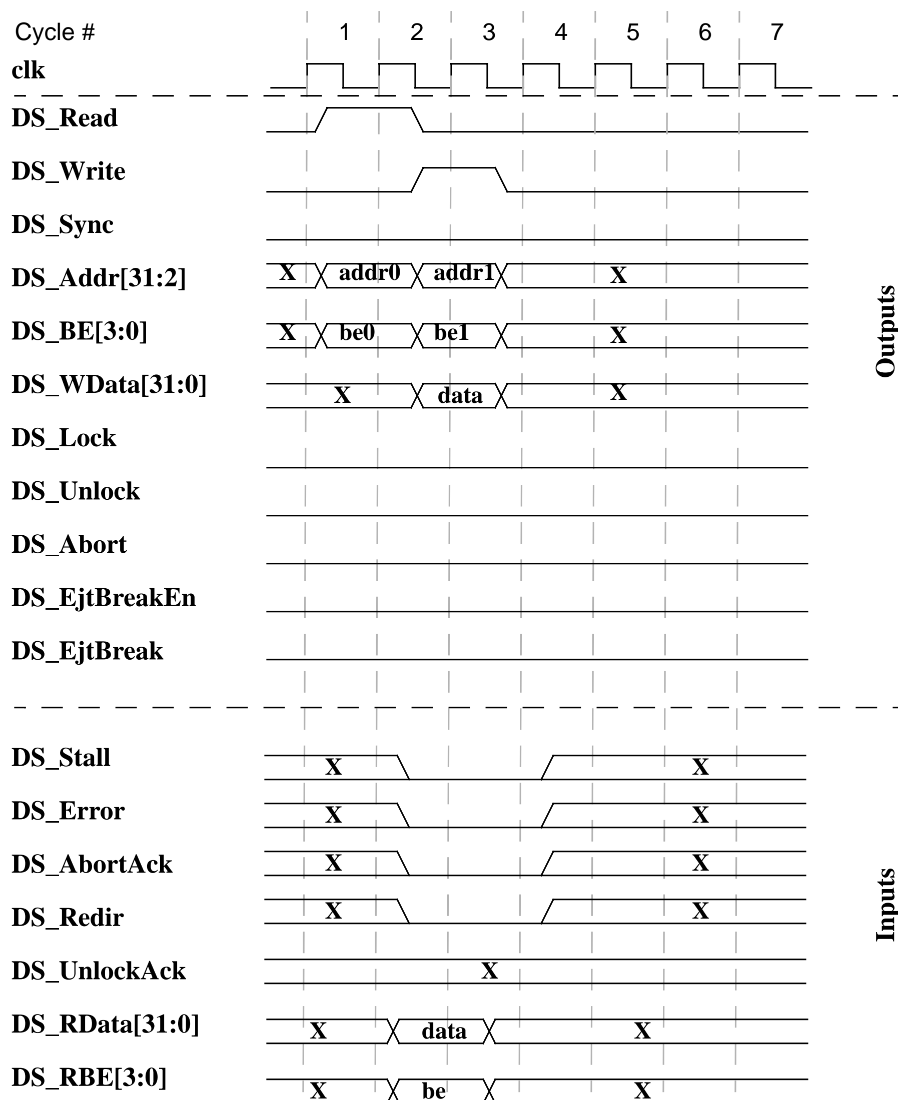

Figure 3-8 Read followed by write (single cycle)

3.4.1.6 Read Followed by Write, with Waitstates

Figure 3-9 illustrates a one waitstate D-side read operation followed immediately by a one waitstate D-side write operation. This example is similar to the back-to-back read/write case in Figure 3-8, only now each of the two transactions includes one waitstate.

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

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

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

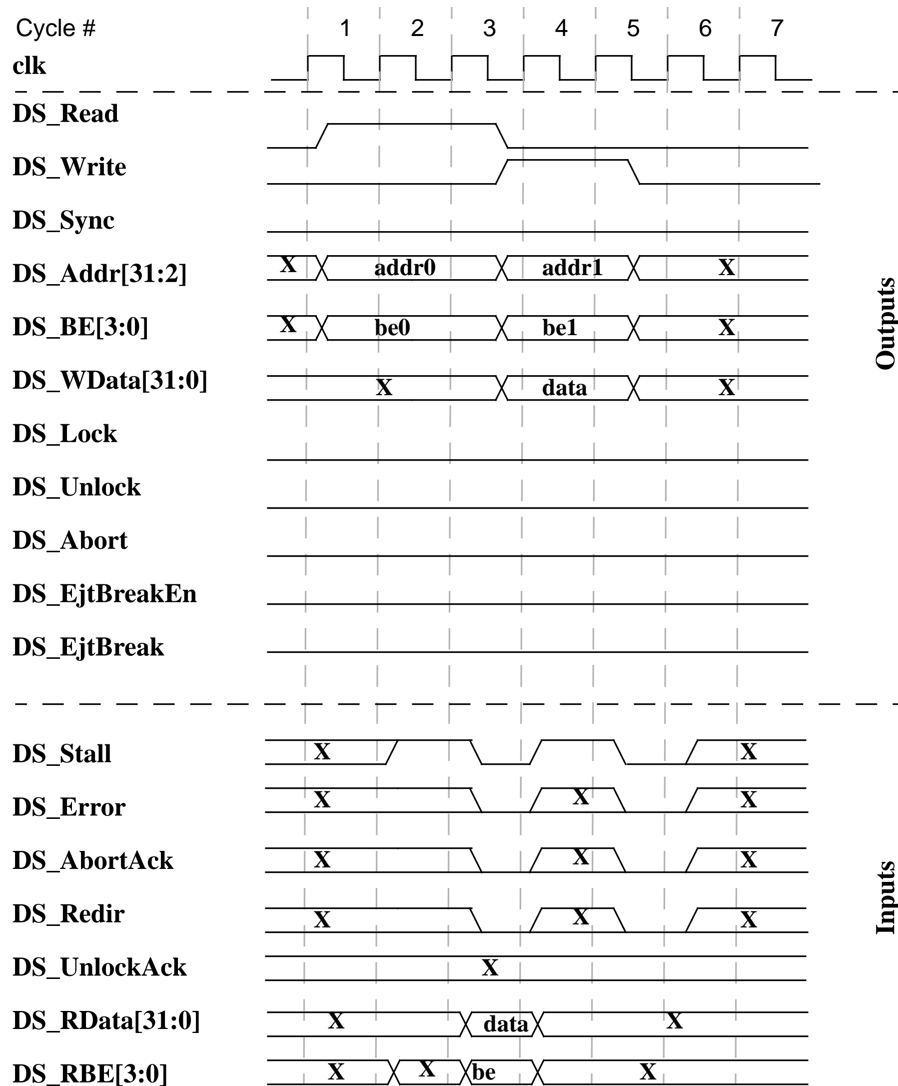

Figure 3-9 Read followed by write (one waitstate)

3.4.2 MIPS16e Instruction Fetches

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

Figure 3-10 illustrates instruction fetches when executing in MIPS16e mode, assuming no waitstates.

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

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

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

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

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

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

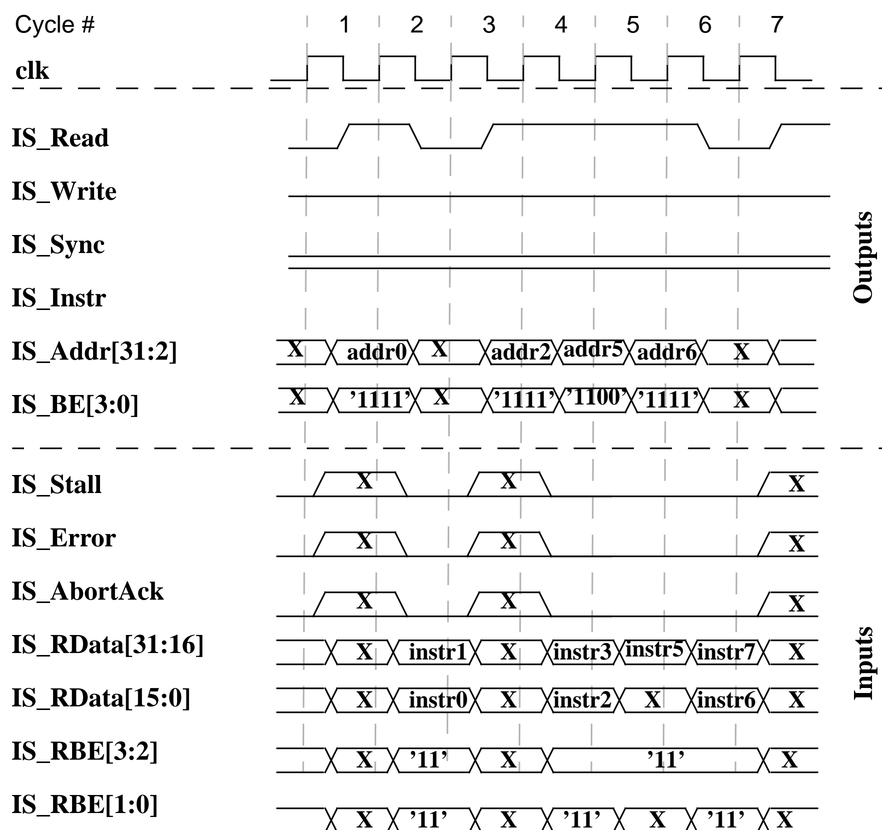

Figure 3-10 MIPS16e instruction fetches (single cycle, little endian mode)

3.4.3 Redirection

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

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

If a redirection request occurs while an interrupt is pending (see Section 3.4.7, "Abort"), the redirection request will actually be ignored by the core to reduce interrupt latency, even if the abort is not explicitly acknowledged by the external agent. The interrupt will be taken on the instruction that initiated the D-side transaction. After execution of the interrupt

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

service routine, this instruction will typically be re-executed, where it will then be redirected (if so requested by *DS_Redir*, and no interrupts are pending this time).

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

3.4.3.1 Redirected Read, Single-Cycle

Figure 3-11 illustrates a single-cycle D-side read operation where *DS_Redir* is used for requesting the operation to be redirected to the I-side. In this example, the I-side read operation is also single cycle.

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

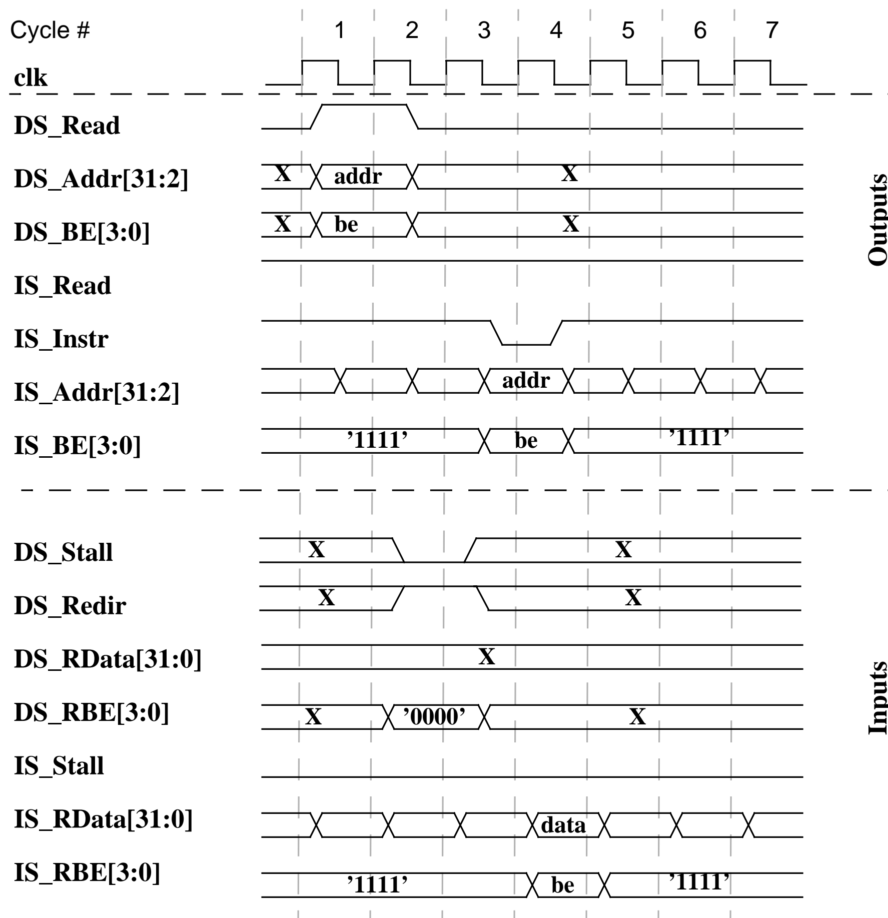

Figure 3-11 Redirected read (single cycle)

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

3.4.3.2 Redirected Read with Waitstate

Figure 3-12 illustrates a one waitstate D-side read operation where *DS_Redir* is used for requesting the operation to be redirected to the I-side. In this example, the I-side read operation also has one waitstate.

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

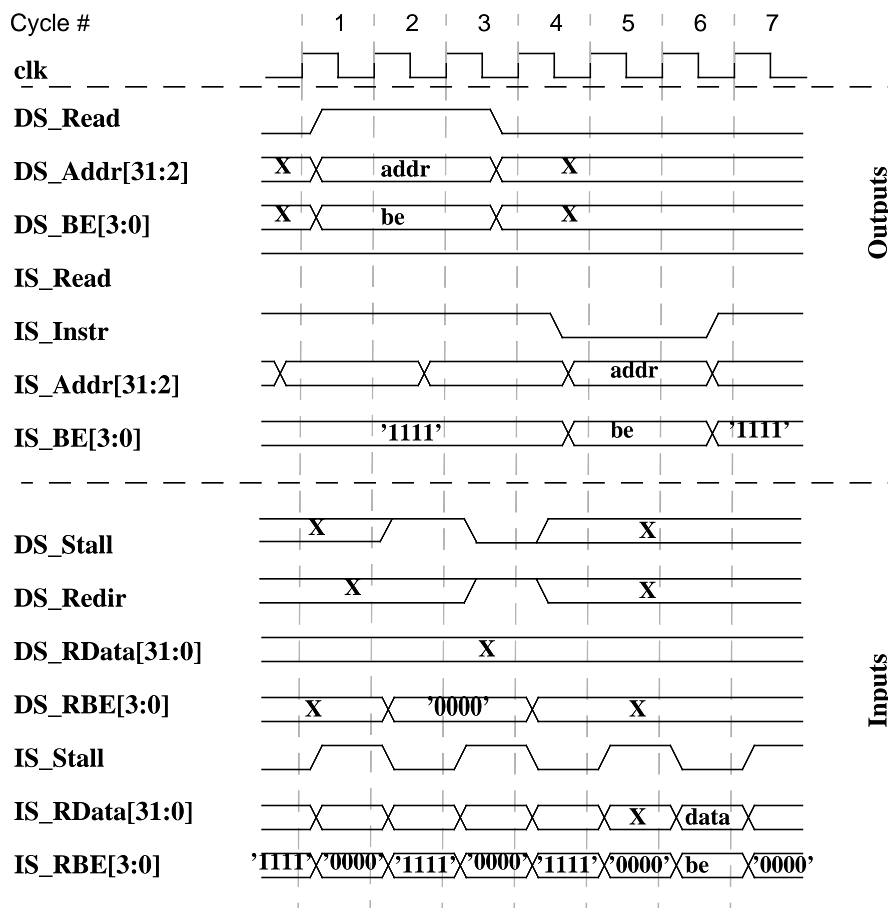

Figure 3-12 Redirected read (one waitstate)

3.4.3.3 Redirected Write, Single-Cycle

Figure 3-13 illustrates a single cycle D-side write operation where *DS_Redir* is used for requesting the operation to be redirected to I-side. In this example, the I-side write operation is also single cycle. Writes redirected to the I-side might be used as a method for initializing the instruction code space, as writes to instruction memory are not otherwise possible from the core.

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

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

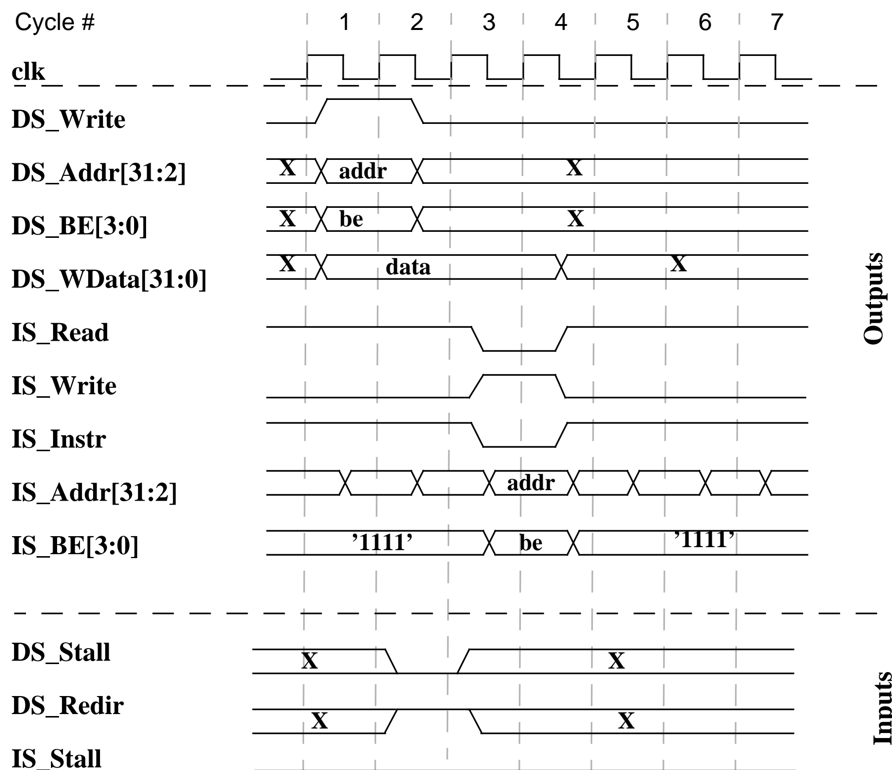

Figure 3-13 Redirected write (single cycle)

3.4.3.4 Redirected Write with Waitstate

Figure 3-14 illustrates a one waitstate D-side write operation where *DS_Redir* is used for requesting the operation to be redirected to I-side. In this example, the I-side write operation also has one waitstate.

The sequence shown in Figure 3-14 is similar to the single cycle write redirection in Figure 3-13, only this time one waitstate is asserted on the D-side before the redirection is signaled, and then another waitstate is signaled on the I-side before the write is accepted.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

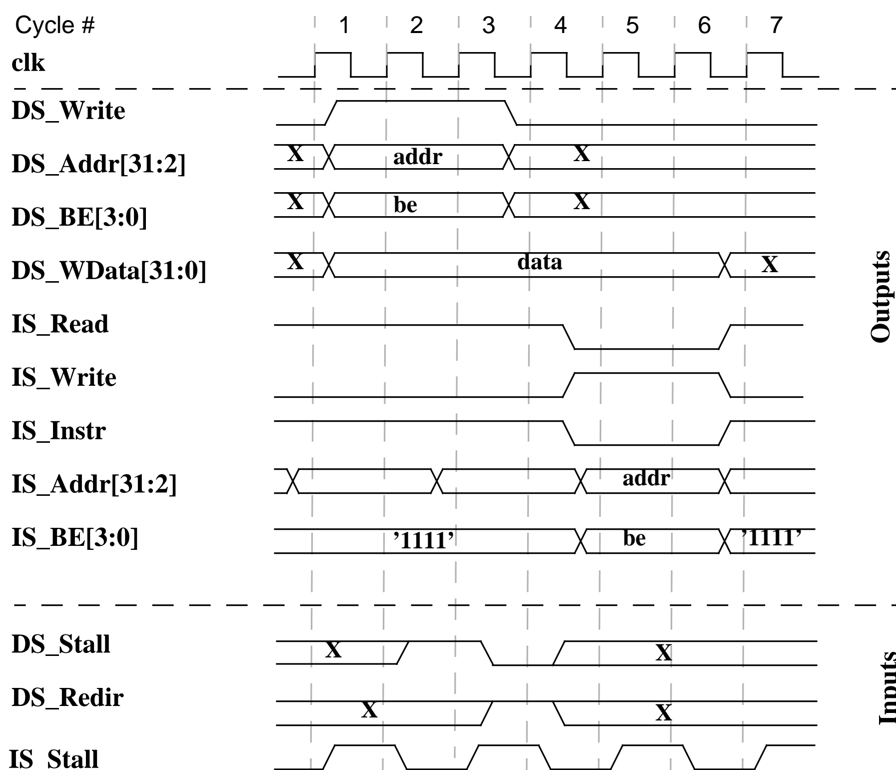

Figure 3-14 Redirected write (one waitstate)

3.4.4 Data Gathering

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

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

The gathering capability is illustrated in Figure 3-15. The data read is initiated by the core in cycle 1, as normal. In this example, the requested read data is 32 bits wide, but it will be returned one byte at a time. The external agent asserts *DS_Stall* for 3 clocks, starting in cycle 2. In cycles 2-4, a single byte of read data is returned each clock, as indicated by the input byte enables (*DS_RBE[3:0]*), while stall remains asserted. Finally in cycle 5, stall is deasserted and the final byte is returned, completing the read transaction.

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

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

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

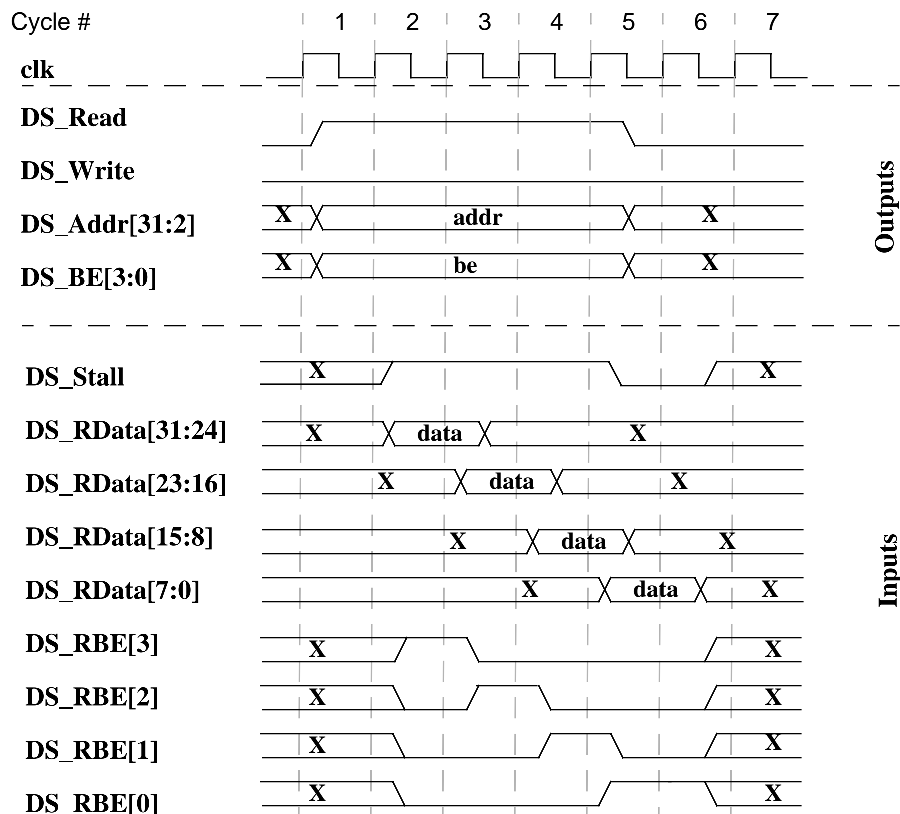

Figure 3-15 Word read, data arriving bytewise

3.4.5 Sync

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

3.4.5.1 Sync with Waitstate

Figure 3-16 illustrates D-side sync signaling for flushing external write buffers. One waitstate is assumed in this example.

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

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

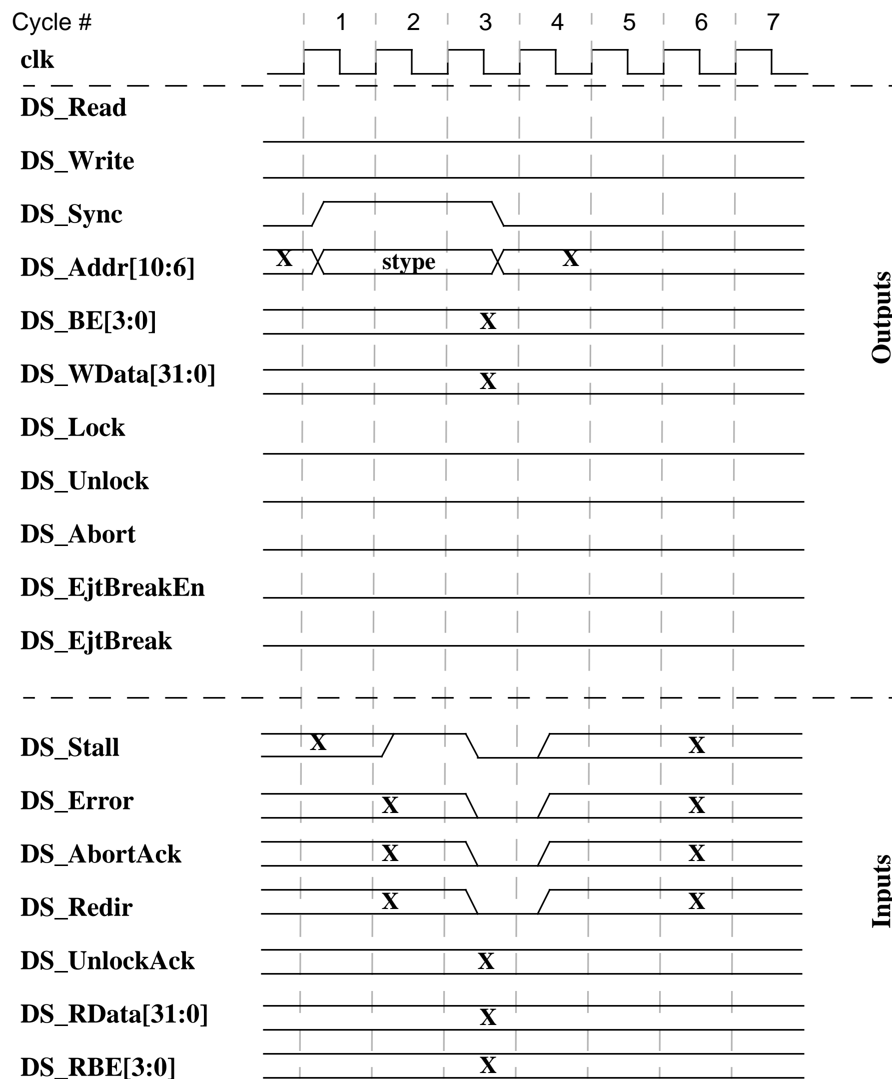

Figure 3-16 Sync (one waitstate)

3.4.5.2 Redirected Sync

Figure 3-17 illustrates sync signaling where the sync operation is requested to be redirected to I-side in order to flush I-side external write buffers. One waitstate for both D- and I-side is assumed in this example.

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

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

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

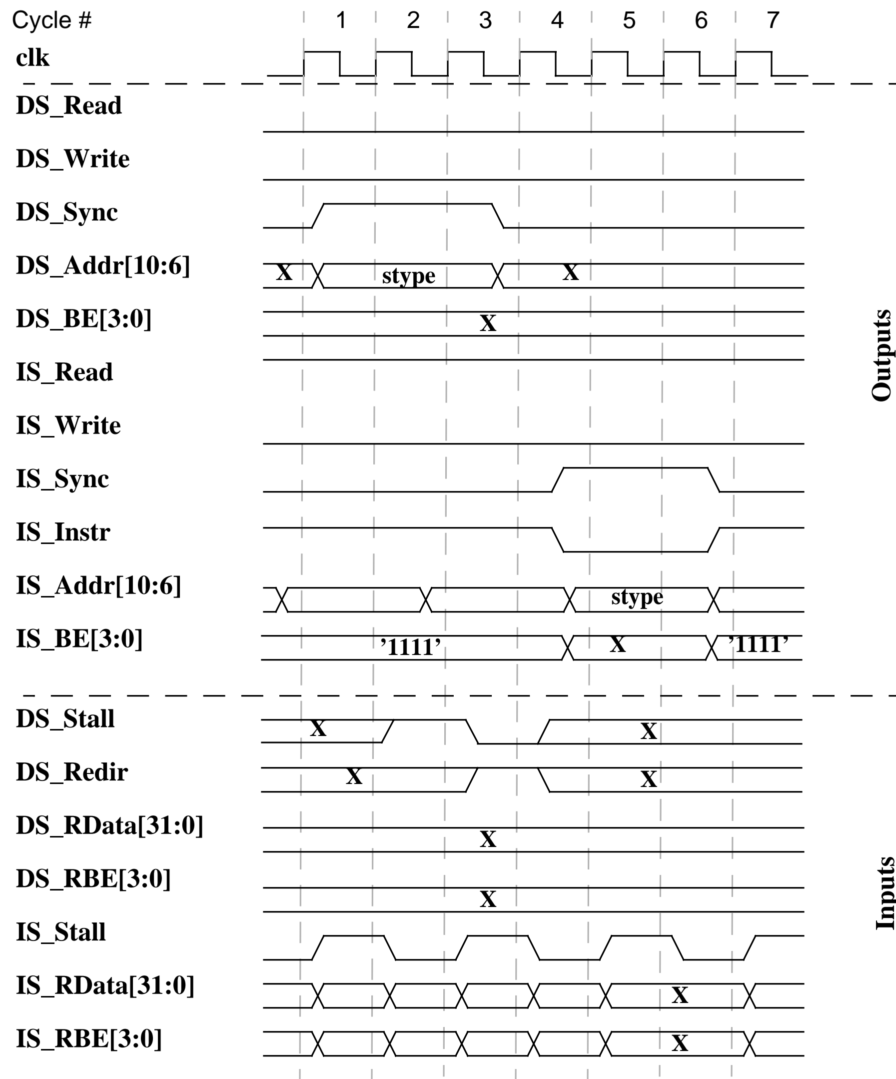

Figure 3-17 Redirected sync (one waitstate)

3.4.6 Bus Error

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

3.4.6.1 Bus Error on Single Cycle Read

Figure 3-18 illustrates a single-cycle D-side read operation causing a bus error, signalled via *DS_Error*.

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

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

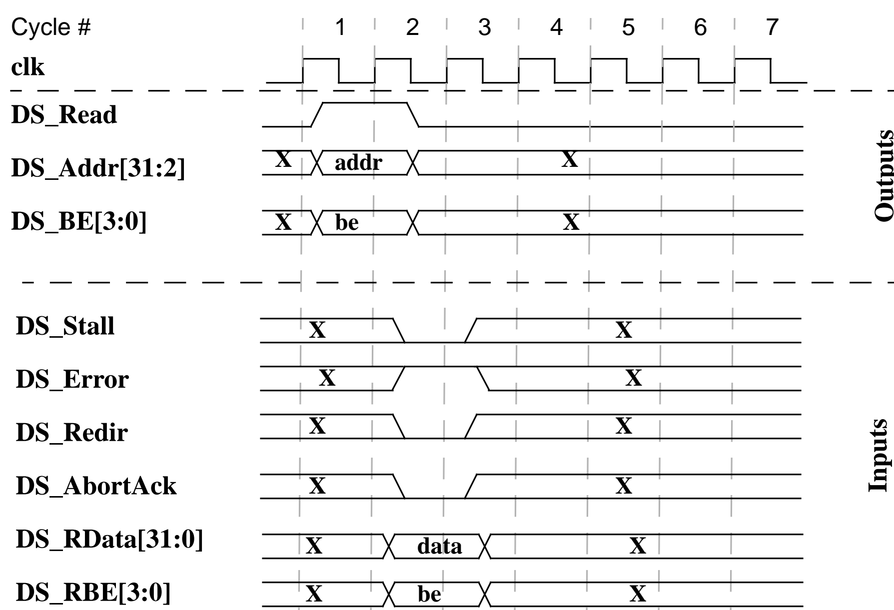

Figure 3-18 Read with error indication (single cycle)

3.4.6.2 Bus Error on Read with Waitstate

Figure 3-19 illustrates a one waitstate D-side read operation causing a bus error.

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

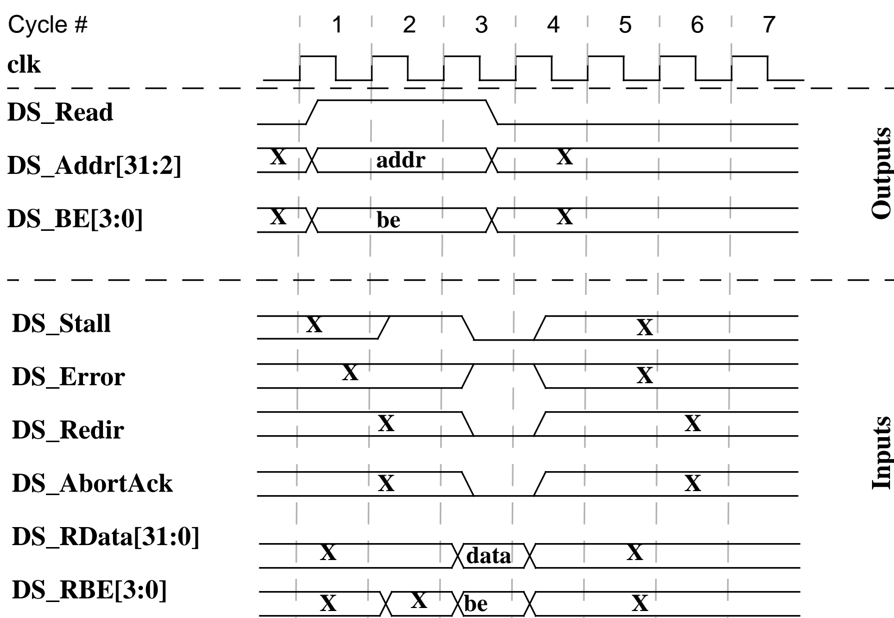

Figure 3-19 Read with error indication (one waitstate)

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

3.4.7 Abort

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

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

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

Examples of aborted transactions are discussed in the following subsections.

3.4.7.1 Aborted Read

Figure 3-20 illustrates a one waitstate D-side read operation with an abort request. In this example, external logic was able to abort the operation, and signals the acknowledgment through assertion of *DS_AbortAck*.

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

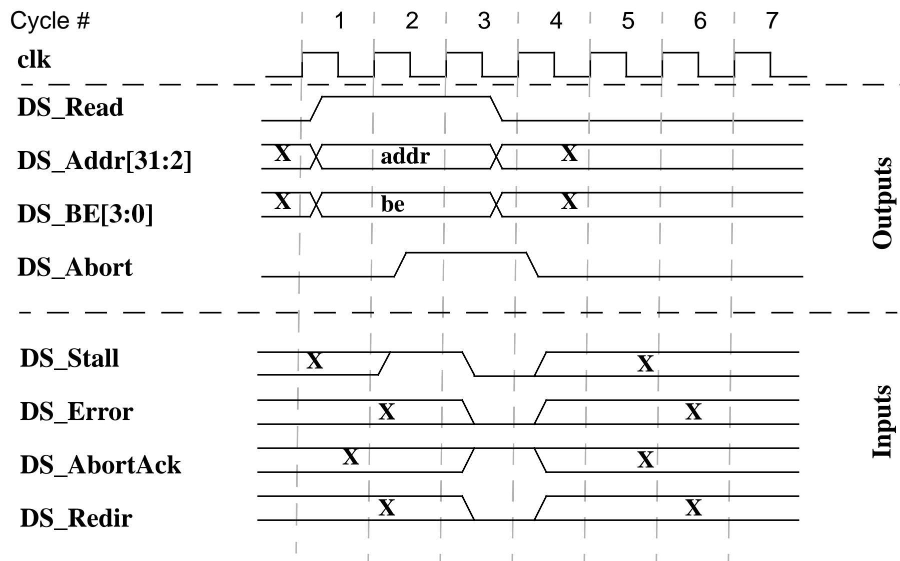

Figure 3-20 Aborted read (one waitstate)

3.4.7.2 Unsuccessful Abort for Single-Cycle Write

Figure 3-21 illustrates a single-cycle D-side write operation with an abort request. In this example, the external logic ignores the request and does not abort the operation.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

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

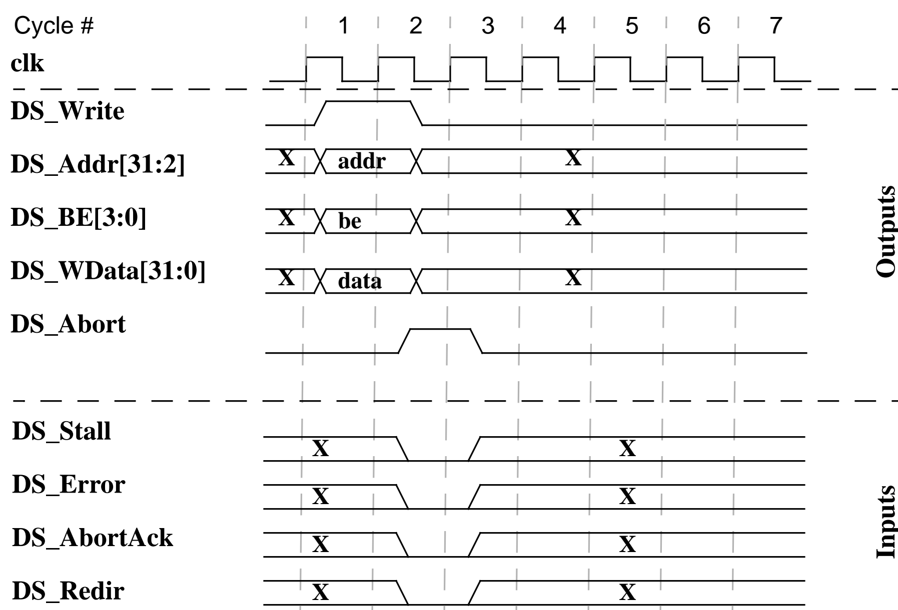

Figure 3-21 Unsuccessful Abort attempt for write (single cycle)

3.4.7.3 Aborted Multi-Cycle Write

Figure 3-22 illustrates another case of a successfully aborted operation. This example demonstrates that the abort request can be signaled several cycles after the transaction has started.

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

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

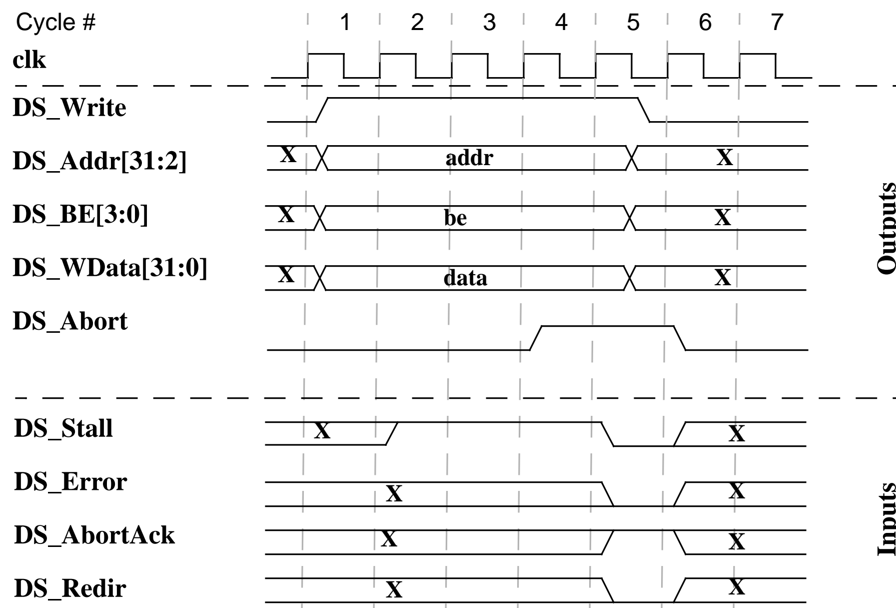

Figure 3-22 Aborted write (multi cycle)

3.4.8 EJTAG Hardware Breakpoints

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

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

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

3.4.8.1 EJTAG Break on Data Write

Figure 3-23 illustrates a one-waitstate D-side write operation causing an EJTAG data break. The EJTAG data break is signalled using *DS_EjtBreak*.

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

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

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

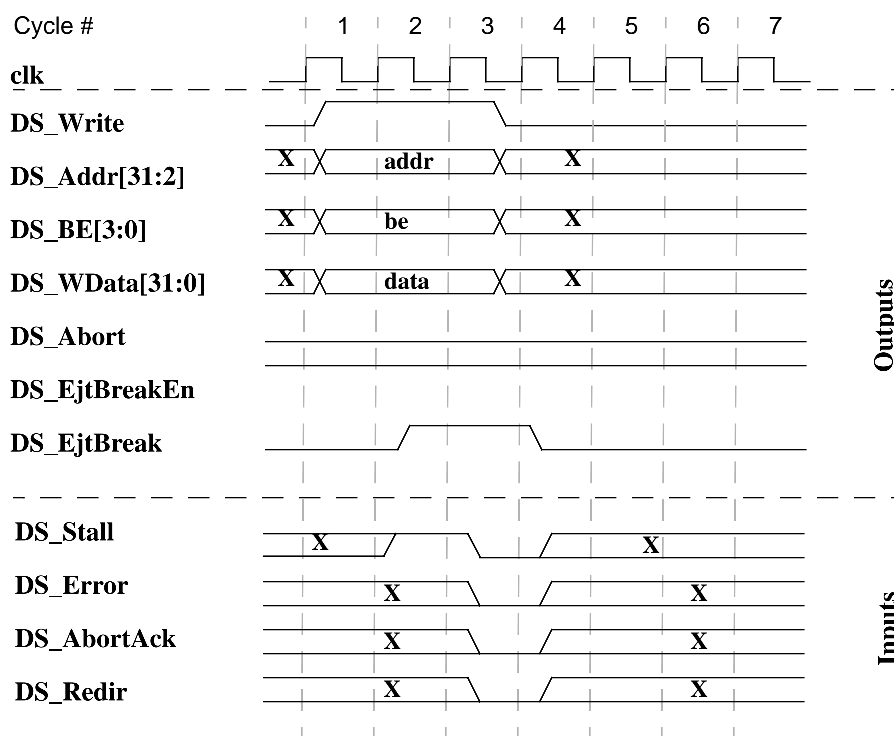

Figure 3-23 EJTAG data write break (one waitstate)

3.4.8.2 EJTAG Break for Data Write, Unified Interface

Figure 3-24 illustrates a data write operation on the Unified Interface. The data write causes an EJTAG data break, which is signalled using *IS_EjtBreak*.

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

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

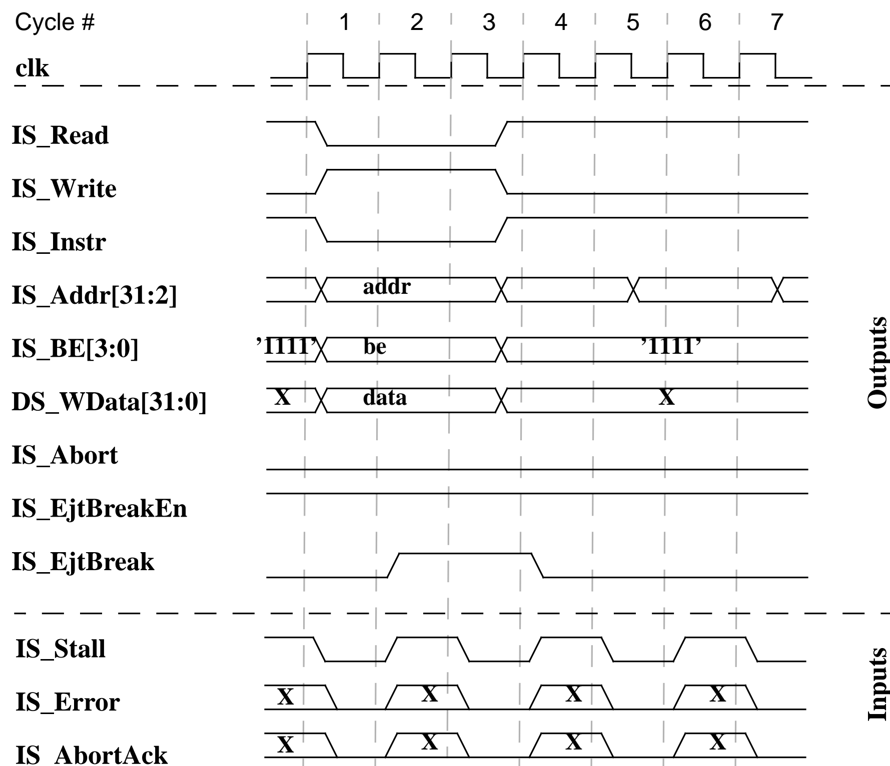

Figure 3-24 EJTAG data write break for Unified Interface (one waitstate)

3.4.9 Lock

Figure 3-25 illustrates the locking mechanism available to handle semaphores on the interface. This mechanism is used during the execution of D-side "load linked" / "store conditional" (LL/SC) operations.

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

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

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

An additional signal, *IS_UnlockAll*, is related to the locking mechanism but not shown in Figure 3-25. *IS_UnlockAll* is asserted for one cycle whenever an ERET instruction is performed. This signal is only present on the I-side (and therefore the Unified Interface), and has no equivalent on the D-side. Whenever an ERET instruction is executed, *IS_UnlockAll* is asserted for one cycle. When this occurs, external logic can unlock all addresses locked by that CPU. An ERET is typically issued for each task-switch performed by the operating system.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

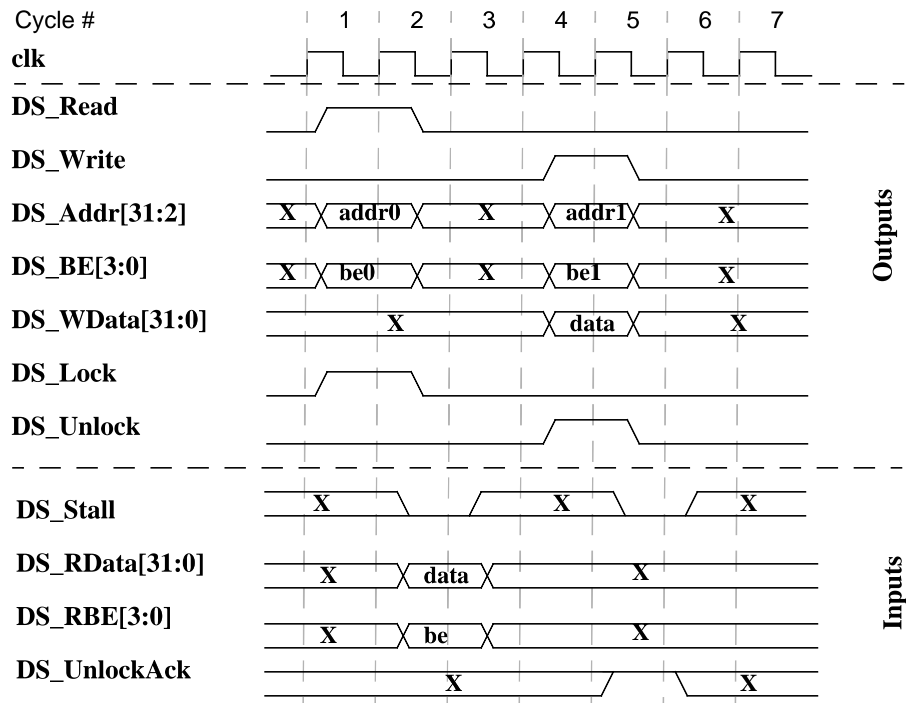

Figure 3-25 Locking (single cycle)

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Chapter 4

EJTAG Interface

This chapter discusses chip-level integration details for the EJTAG-related signals on a MIPS32™ M4K™ 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 M4K cores within a single chip are discussed.

This chapter contains the following sections:

- Section 4.1, "EJTAG versus JTAG"
- Section 4.2, "How to Connect EJ_* Pins"
- Section 4.3, "Multi-Core Implementations"
- Section 4.4, "EJTAG Trace"

An EJTAG TAP controller is an optional feature in a M4K core. If the M4K core under use does not contain the EJTAG TAP controller, then much of this chapter is irrelevant.

Reference to the general *EJTAG Specification* [2] can be found several times in this chapter. 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, 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 uses a superset of the JTAG TAP interface, but that is really its only similarity with boundary scan.

Read the "EJTAG Debug Support" chapter in the *MIPS32™ M4K™ Processor Core Software User's Manual* [1]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 in 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 recommends NOT sharing any logic or pins between JTAG and EJTAG. MIPS recognizes that reducing pin count is often necessary in large System-on-a-Chip (SOC) chip designs.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

4.1.2 Sharing EJTAG Resources with JTAG

It is theoretically possible to share the TAP controller for JTAG and EJTAG purposes because the EJTAG control commands do not use reserved JTAG commands. This TAP sharing is not supported by the M4K core, however. The M4K core has its own independent TAP controller that is 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 between 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*.
- EJTAG *EJ_TDOzstate* to output enable of physical *TDO*.

Figure 4-1 on page 49 shows the serial *TDI-TDO* chain setup with parallel control of the TAP controllers.

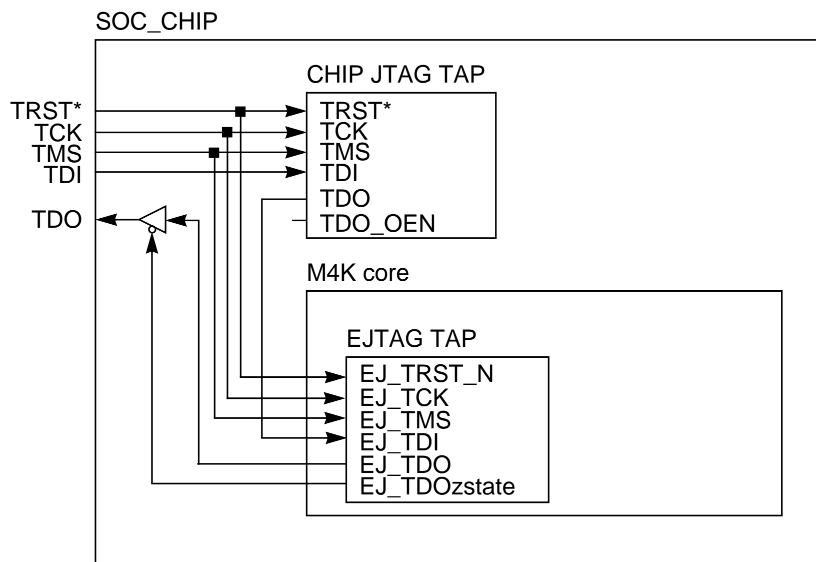

Figure 4-1 Daisy-Chained *TDI-TDO* Between JTAG and EJTAG TAP Controllers

Some EJTAG debug tool chains can handle this configuration. If another TAP controller in the path to the EJTAG TAP controller can be identified, then the debug software must be told the following items:

- the Instruction word length of the JTAG TAP controller
- the Instruction word command to select the bypass register (usually all 1's)
- the length of the bypass register (usually one bit)

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.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

The main problem is the presence of the serial EJTAG TAP controller in the JTAG TAP path; automatic JTAG testbenches do not like the visibility of another TAP controller inside the chip. MIPS strongly recommends NOT using the setup in Figure 4-1 on page 49 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 the user wishes to gate off the inputs of the unselected TAP controller depends on the presence of an asynchronous reset input. In Figure 4-2 on page 50, a setup which anticipates the existence of *TRST** on the “CHIP JTAG TAP” controller is shown.

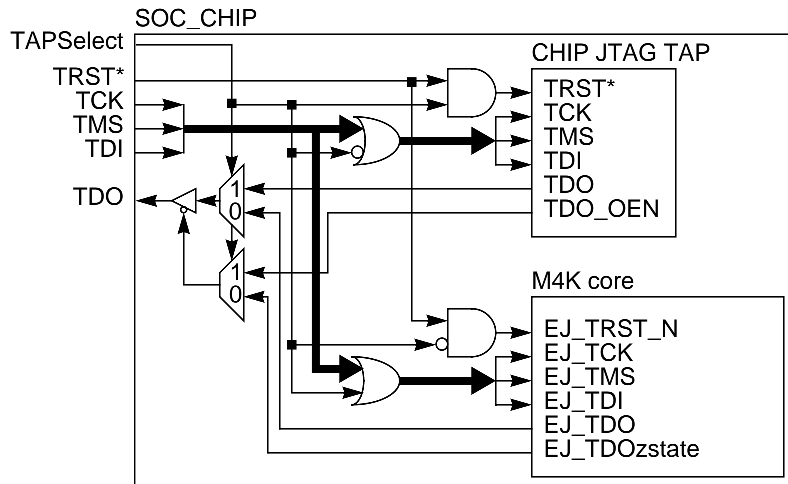

Figure 4-2 Multiplexing Between JTAG and EJTAG TAP Controllers

TAPSelect in Figure 4-2 on page 50 is shown as an SOC_CHIP external input, and NOT as internal logic or registered signal. This is 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 M4K core (and thus probably the entire SOC_CHIP) is held in reset. During reset, EJTAG commands can initialize the M4K core to leave the reset state in Debug Mode, and thus the debug interface can control the M4K core before 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. One small implication to this is, that the *TAPSelect* input can not be tested by JTAG boundary scan. It might be wise to NOT have boundary scan include the *TAPSelect* input logic. 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 on page 50.

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 the chip has a separate set of EJTAG TAP pins. Other non-TAP EJTAG pins on the M4K core will require separate pins on the chip. This section will discuss how to connect all the *EJ_** pins in the chip.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

4.2.1 EJTAG Chip-Level Pins

The EJTAG TAP signals on the M4K 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 on page 51 shows the intended connection to the chip. Pin names for the chip have been chosen as the usual JTAG TAP signals, with an "*E*" prefix.

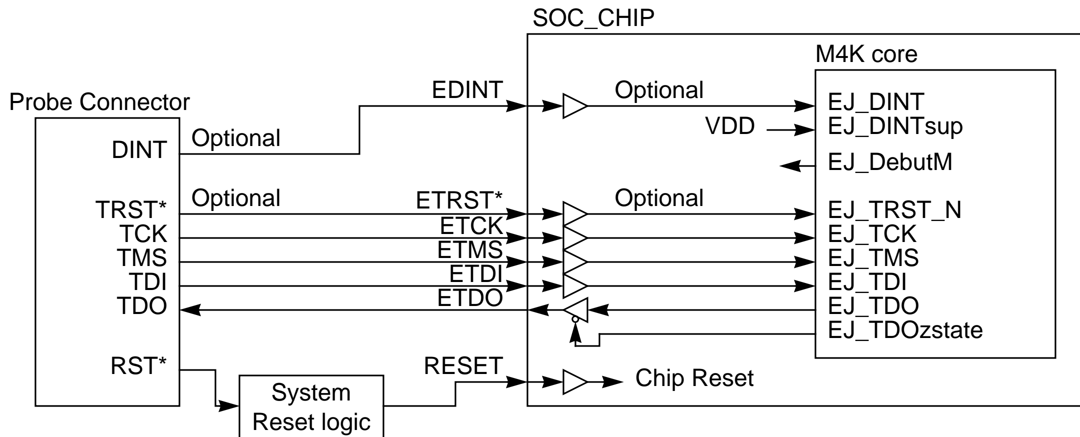

Figure 4-3 EJTAG Chip-Level Pin Connection

AC timing characteristics for the *ETDO* driver and the input buffers can be found in Section 7.2, "AC Timing Characteristics", of the EJTAG Specification. In particular notice that all the probe pins must have pull-up or pull-down logic attached. As shown in Figure 4-3 on page 51, all the chip-level pins have corresponding pins on the EJTAG Probe Connector. *RST** is special, because an assertion (active low) on this pin must result in a system level reset. Refer to Figure 4-4 on page 53 for further details on EJTAG-related reset circuitry.

4.2.1.1 Optional *ETRST** Pin

Although the *ETRST** is an optional input pin on the chip, it is strongly recommended that the *ETRST** pin be present. If this pin is not used, on-chip logic is needed that 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 M4K core must not affect the *EJ_TRST_N* signal. Special timing also applies to the deassertion of *EJ_TRST_N*. Refer to Section 6.3 of the EJTAG Specification, "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 M4K core triggers a Debug Interrupt Exception. This will stop the normal program flow within the M4K 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. The EJTAG Control Register is accessed through the TAP controller pins, which takes multiple *ETCK* clock periods.

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

EJ_DINT on the M4K core may also be connected to on-chip logic, such as a Multi-Core Breakpoint Unit (see Figure 4-5 on page 54 for more details). The *EJ_DINTsup* (EJTAG Debug Interrupt Pin Supported) input on a M4K core is asserted only if the *EJ_DINT* input connected to the *DINT* pin of the Probe Connector. The *EJ_DINT* input may not be disabled if the the *EJ_DINTsup* input is deasserted. *EJ_DINTsup* is only used to set the DINTsup bit in the EJTAG Implementation Register.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

If *EJ_DINT* on the M4K core to an interrupt source is not connected, then both *EJ_DINT* and *EJ_DINTsup* must be deasserted 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 *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 "Section 4.2.2, "EJTAG Device ID Input Pins" on page 52".

4.2.2.2 EJ_PartNumber[15:0]

EJ_PartNumber[15:0] is recommended to be a manufacturer-specific number identifying this core as a MIPS M4K core. A new physical cache configuration could facilitate a new value on *EJ_PartNumber[15:0]*, but 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 recommended 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.

Another software reset-related pin is 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 M4K core, accessible in Debug Mode).

4.2.3.1 *EJ_PrRst* Signal

Processor Reset can 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 off 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 on page 53 shows one possible implementation for the use of *EJ_PrRst*.

4.2.3.2 *EJ_PerRst* Signal

Peripheral Reset can be used as a soft reset of the peripherals surrounding the M4K core. The effect of an asserted *EJ_PerRst* is implementation-dependent; however, it should never result in a reset of the M4K core itself. Figure 4-4 on page 53 shows one possible implementation of the use of *EJ_PerRst*.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

4.2.3.3 *EJ_SRstE* pin

As described earlier, this signal can be used to control one or more Soft Reset sources in the system reset logic. See Figure 4-4 on page 53 for a possible implementation.

4.2.3.4 A Reset Logic Implementation

Figure 4-4 on page 53 shows 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 M4K core. This requires 3 extra output signals, but this need not be the case.

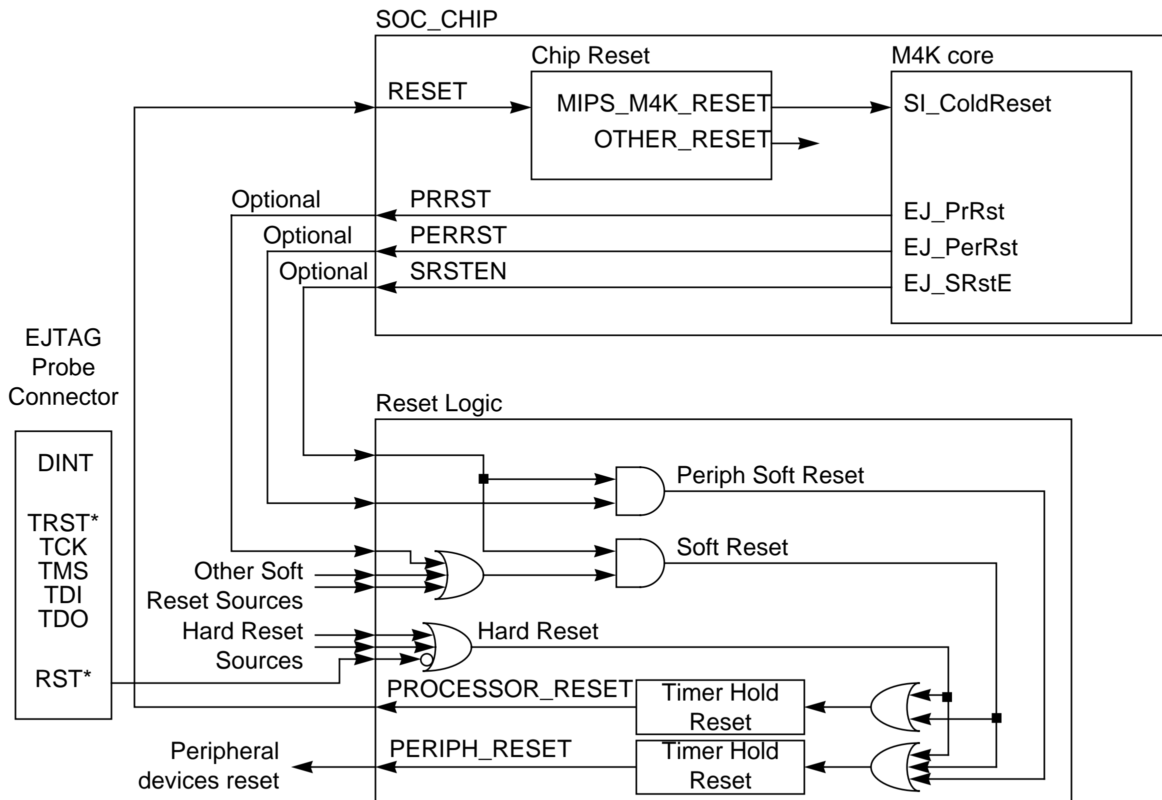

Figure 4-4 Reset Circuitry Implementation

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

4.3 Multi-Core Implementations

In a chip configuration with multiple M4K cores, all 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 on page 54.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

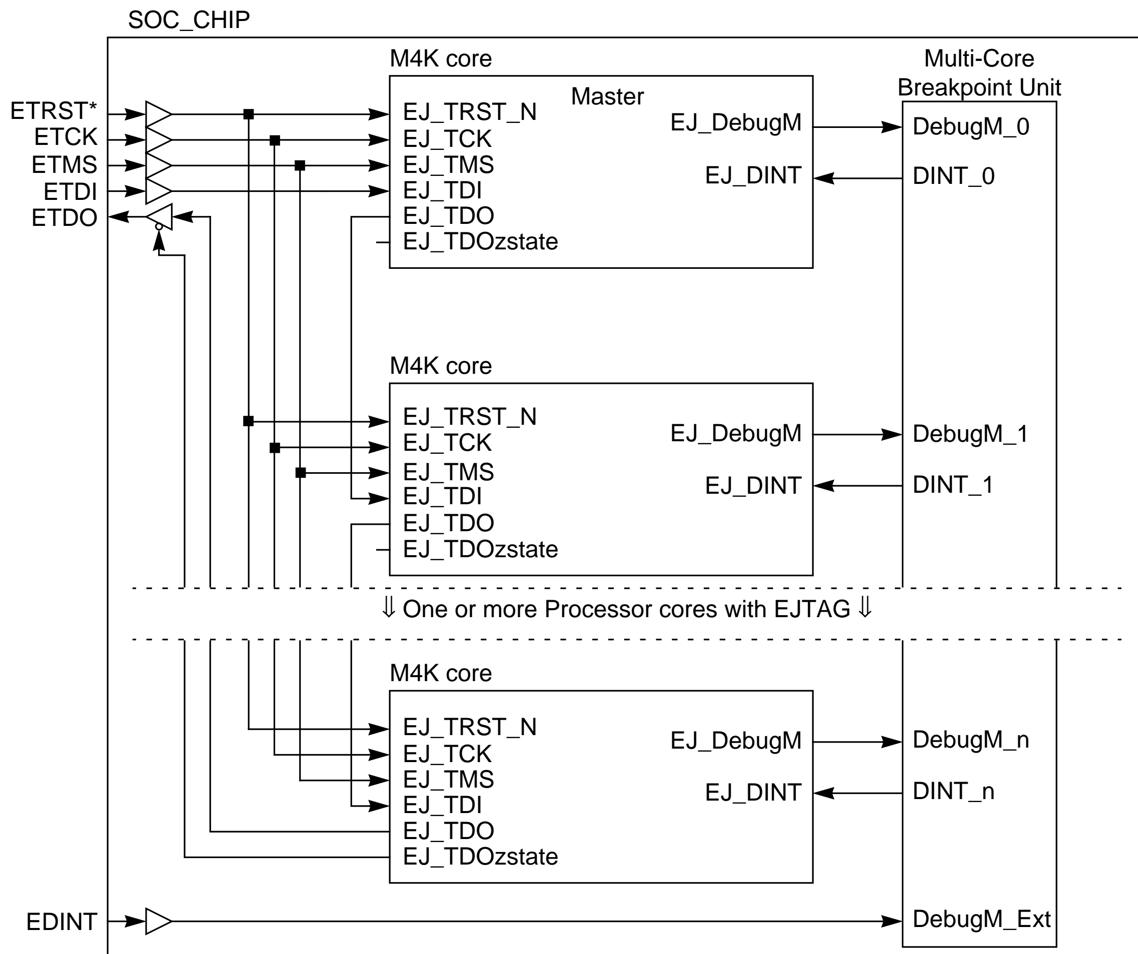

Figure 4-5 Multi-Core Implementation

4.3.1 *TDI/TDO* Daisy-Chain Connection

In a Multi-Core implementation, one of the processor cores is often be the Master. In Figure 4-5 on page 54, the Master core is first in the *TDI/TDO* daisy-chain to get a low latency access to control and data registers in the Master core. When a large number of EJTAG TAP controllers are connected in the daisy-chain, the placement of the Master core be of any significance.

The chip's ETDO output enable is controlled by EJ_TDOzstate in the last core in the chain because this core drives 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 on page 54 is an implementation-dependent block. Each core can 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).

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Debugger software can detect that the Master core has entered Debug Mode, and trigger this for the Slave core(s). This might be supported by your Debug software as an automatic feature. 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, if an MCBU is designed, 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 does 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 M4K core ignores any subsequent Debug Interrupts on *EJ_DINT*.

4.4 EJTAG Trace

A M4K core can support EJTAG Trace features, which enables real-time tracing of the Program Counter and load/store address and data values. The trace logic is included as a build time option. Four basic options are possible:

1. No trace logic included.
2. Trace logic to on-chip trace memory (embedded within the core).
3. Trace logic to support an off-chip trace probe (with off-chip trace memory).
4. Combination of options 2 and 3.

If options 1 or 2 are present, then the *TC_* output pins on the core will be statically driven to zero, and all the *TC_* inputs are ignored. With option 2, access to the trace features and on-chip trace memory occurs through the standard EJTAG probe.

If options 3 or 4 are present, then the TCtrace Interface on the M4K core is active and the *TC_* inputs and outputs must be connected to a core external Probe Interface Block (PIB), or tied off. If a PIB is not implemented then all the *TC_* inputs should be tied low.

The specific implementation details for the PIB and how to connect it to the core can be found in the *EJTAG Trace Control Block Specification* [3].

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Chapter 5

Coprocessor Interface

This chapter describes the MIPS Core Coprocessor Interface supported by the MIPS32™ M4K™ processor core. The MIPS Core Coprocessor Interface is described in the companion document, titled *Core Coprocessor Interface Specification* [4]. The Core Coprocessor Interface is an optional feature in a M4K core. If the M4K core does not contain the Core Coprocessor Interface logic, then this chapter is irrelevant. This chapter discusses the specific M4K implementation of the Core Coprocessor Interface, in the following sections:

- Section 5.1, "Introduction"
- Section 5.2, "Coprocessor Instructions"
- Section 5.3, "Signal Configuration"
- Section 5.4, "Interface Protocols"
- Section 5.5, "Power Saving Issues"
- Section 5.6, "Template for Coprocessor Modules"

5.1 Introduction

The M4K core Coprocessor Interface allows a single Coprocessor 2 (COP2) to be connected to the integer unit. The function of Coprocessor 2 is user-definable and is intended to allow special-purpose engines, such as a graphics accelerator that is integrated into the architecture. The M4K core does *not* support an interface to a floating-point unit, which is dedicated to Coprocessor 1 in the MIPS32™ architecture. The special handling for floating-point instructions needed in the integer unit, as well as the extra signaling needed between the integer unit and a floating-point unit, is not present in a M4K core.

The Coprocessor Interface has the following features:

- No late or critical signals are part of the interface. This allows for easier design and synthesis for coprocessor designers.
- By keeping the interface as simple as possible, designers can concentrate on the coprocessor functionality rather than its interface.
- Minimal required interface logic, thereby minimizing area and power overhead.
- Performance is not compromised. This interface is compatible with all high-performance features of the M4K processor core.
- Fully compliant to the MIPS Core Coprocessor Interface standard.

5.2 Coprocessor Instructions

A M4K core supports all MIPS32-compliant COP2 instructions, except the load double (LDC2) and store double (SDC2) instructions. Table 5-1 lists all the supported instructions and how they are decoded.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 5-1 Supported Coprocessor 2 instructions

Instruction	Decode	Description
LWC2	$IR[31:26] = 110010_2$	Load Word from memory to a Coprocessor 2 register. COP2 register number = IR[20:16], sub-select = 0[a].
SWC2	$IR[31:26] = 111010_2$	Store Word to memory from a Coprocessor 2 register. COP2 register number = IR[20:16], sub-select = 0[a].
MFC2	$IR[31:26] = 010010_2$ & $IR[25:21] = 00000_2$	Move word from Coprocessor 2 register to processor general-purpose register. COP2 register number = IR[15:11], sub-select = IR[2:0][b].
CFC2	$IR[31:26] = 010010_2$ & $IR[25:21] = 00010_2$	Move word from Coprocessor 2 control register to processor general-purpose register. COP2 control register number = IR[15:11][c].
MTC2	$IR[31:26] = 010010_2$ & $IR[25:21] = 00100_2$	Move word to Coprocessor 2 register from processor general-purpose register. COP2 register number = IR[15:11], sub-select = IR[2:0][b].
CTC2	$IR[31:26] = 010010_2$ & $IR[25:21] = 00110_2$	Move word to Coprocessor 2 control register from processor general-purpose register. COP2 control register number = IR[15:11][c].
BC2F BC2FL	$IR[31:26] = 010010_2$ & $IR[25:23] = 010_2$ & $IR[16] = 0_2$	Branch on Coprocessor 2 condition false (likely)[d]. The condition code check from the coprocessor should be set if the condition is False. Condition is specified by IR[22:18].
BC2T BC2TL	$IR[31:26] = 010010_2$ & $IR[25:23] = 010_2$ & $IR[16] = 1_2$	Branch on Coprocessor 2 condition true (likely)[d]. The condition code check from the coprocessor should be set if the condition is True. Condition is specified by IR[22:18].
COP2	$IR[31:26] = 010010_2$ & $IR[25] = 1_2$	Perform Coprocessor 2 operation. Operation is specified by IR[24:0].

Note: [a] The LWC2 and SWC2 instructions has no room to specify a sub-select COP2 register value. sub-select 0 must be assumed.

Note: [b] The MFC2 and MTC2 instructions target a COP2 register (0-31) with a sub-select (0-7), effectively making the COP2 register file of size: 32x8 = 256 registers.

Note: [c] The CFC2 and CTC2 instructions target COP2 control registers (0-31). There is no sub-select field, making the COP2 control register file of size: 32 registers.

Note: [d] The BC2 instructions use IR[17] to select between branch and branch likely type instructions. The coprocessor would typically not care to look at IR[17] for BC2 instruction decodes.

Only instructions with the decode specified in Table 5-1 may be sent to the coprocessor. If an instruction is not supported by the coprocessor, then a reserved instruction (RI) exception must be sent back to the M4K core (see Section 5.4.5, "Coprocessor Exceptions").

The M4K core only dispatches instructions to the coprocessor if the CU2 bit in the CP0 *Status* register is set. Refer to the *MIPS32 M4K Processor Core Software User's Manual* for details on Coprocessor 2 instructions and CP0 registers.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

5.3 Signal Configuration

The M4K core Coprocessor 2 interface supports a subset of the possible features specified in the *Core Coprocessor Interface Specification*. Following is a list of the supported features of the M4K core Coprocessor Interface:

- A single COP2 coprocessor is supported. No support for the floating-point COP1 coprocessor.
- Data transfers are 32 bits. No support for 64-bit buses and 64-bit instructions (LDC2/SDC2).
- One issue group is supported (group 0). No support for dual (or more) issue.
- Data from the coprocessor can only be one instruction out-of-order.
- Data to the coprocessor is always sent in order.
- An instruction is never nullified.

From a static pin configuration point of view, the supported features listed above have the following consequences (refer to Table 2-3 on page 3 for a listing of all the M4K core signals).

The *CP2_inst32_0* output is tied high (logic 1). The M4K core is a MIPS32 compliant core only, and does not support any 64-bit features. All instructions assume the coprocessor behaves as a 32 bit device, mandated by always asserting *CP2_inst32_0*. A possible *CP2_tx32_0* output from a coprocessor[1] to the core is not defined on the interface of the core, and can be left unconnected on the coprocessor.

The *CP2_tdata_0[31:0]* and the *CP2_fdata_0[31:0]* data buses are only 32 bits wide. 64-bit transfers are not supported.

The *CP2_tordlim_0[2:0]* input is ignored and the *CP2_torder_0[2:0]* output is tied to 000_2, since the M4K core never sends data out of order. The coprocessor attached to a M4K core does not need to limit the use of out-of-order-ness. This might not be true for other MIPS cores using the same interface. If a coprocessor is built which does not allow data it receives to be sent out-of-order, then it can drive the *CP2_tordlim_0[2:0]* signal to 000_2.

The *CP2_fordlim_0[2:0]* output is tied to 001_2 and the *CP2_forder_0[2:1]* input is ignored. No more than one out-of-order data return is supported. Only *CP2_forder_0[0]* is needed to define the out-of-order-ness of the data received from the coprocessor. If data is sent to the M4K core more than one out-of-order, then it would be a protocol violation and the result from this is undefined.

The *CP2_null_0* output is tied low (logic 0). With the M4K core, the only instruction that may be nullified is an instruction in a branch likely delay slot (when the branch isn't taken). The branch condition is evaluated so early that dispatch of the delay slot instruction can be suppressed. The *CP2_nulls_0* signal will still strobe once for each instruction dispatched as required by the protocol. But no instruction is ever nullified.

Note: If the *CP2_null_0* always being low when implementing the coprocessor is relied upon, then might not be compatible with future versions of the M4K or other MIPS cores.

The *CP2_reset* output is driven directly from a register. This register is driven by the internal reset, and clocked by the core clock (*SI_ClkIn* after clock tree). This means that the assertion/deassertion is one cycle later than what the core sees. This is not a problem as the first instruction after reset can never be a Coprocessor 2 instruction.

The *CP2_present* input determines the presence of a coprocessor. If this input is deasserted (logic 0), then the Coprocessor Interface is disabled. All inputs should be driven static to their inactive values, and all outputs must be ignored. It is not possible to set the CU2 bit in the CP0 *Status* register if *CP2_present* is deasserted (0).

[1] Static signal from a coprocessor, used to indicate it can only handle 32-bit transactions.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

5.4 Interface Protocols

Refer to Table 2-3 on page 3 for a complete listing of all the pins of the M4K core.

The Coprocessor Interface is composed of several simple transfers:

- **Instruction Dispatch** - Starts coprocessor instructions.
- **To COP Data** - Transfers data to the coprocessor.
- **From COP Data** - Transfers data from the coprocessor.
- **Coprocessor Condition Code Check** - Transfers coprocessor condition check result to the M4K core.
- **Coprocessor Exceptions** - Notifies the M4K core whether any coprocessor exceptions happened for an instruction or not.
- **Instruction Nullification** - Notifies the coprocessor whether instructions are nullified or not.
- **Instruction Killing** - Notifies the coprocessor whether instructions can commit state or not.

All transfers use the following protocol:

- All transfers are synchronously strobed, that is, a transfer is only valid for one cycle (when the strobe signal is asserted). The strobe signal is a synchronous signal and should not be used to clock registers.
- No handshake confirmation of transfer.
- Except for instruction dispatch, no flow control.
- Except for To/From COP data transfers, out of order transfers are not allowed. All transfers of a given type, except To/From COP data transfers, must be in dispatch order.
- Ordering of different types of transfers for the same instruction is not restricted.

After an instruction is dispatched, additional information about that instruction must be later transferred between the coprocessor and the M4K processor core. The additional information and the transfers required are summarized in Table 5-2.

Note: For each dispatch type given in the table, all listed transfers are *required* to be completed. No transfers are optional. however, after an instruction is killed or nullified, any additional transfers that have not already happened will not occur. Once an instruction is killed or nullified, no further transfers for that instruction can happen. Additionally, if an instruction is killed, then all transfers for all previously dispatched instructions will not happen either, including instructions dispatched in the same cycle that the kill of an older instruction is sent.

Table 5-2 Transfers Required for Each Dispatch

Dispatch Type	Required Transfers
To COP Op (LWC2/ MTC2/ CTC2)	• Instruction nullification or not[a] • To Coprocessor data transfer • Coprocessor exceptions or not • Instruction killing or not
From COP Op (SWC2/ MFC2/ CFC2)	• Instruction nullification or not[a] • From Coprocessor data transfer • Coprocessor exceptions or not • Instruction killing or not

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 5-2 Transfers Required for Each Dispatch (Continued)

Dispatch Type	Required Transfers
Arithmetic Op (COP2[b])	• Instruction nullification or not[a] • Coprocessor exceptions or not • Instruction killing or not
Arithmetic Op, Branch (BC2b)	• Instruction nullification[a] • Condition code check results • Coprocessor exceptions or not • Instruction killing or not
Note: [a] The M4K core will always signal not-nullified on all instructions. Note: [b] For a description of this instruction, refer to the MIPS ISA definition.	

Each transfer can occur as early as the cycle after dispatch, and there is no maximum limit on how late the transfer can occur. Only the dispatch interfaces have flow control, so that once dispatched, all transfers can occur immediately.

All transfers are strobed. The data is not buffered and is transferred in the cycle that the strobe signal is asserted—if the strobe signal is asserted for 2 cycles, then two transfers occur. For instruction dispatches (Arithmetic, To COP, and From COP instructions) the strobe signal (*CP2_as_0*, *CP2_ts_0* or *CP2_fs_0*) is asserted in the cycle after the instruction is dispatched. This is done in order to insulate the strobe signals from poor timing. The dispatch cycle is the cycle where the instruction bus *CP2_ir_0[31:0]* is valid.

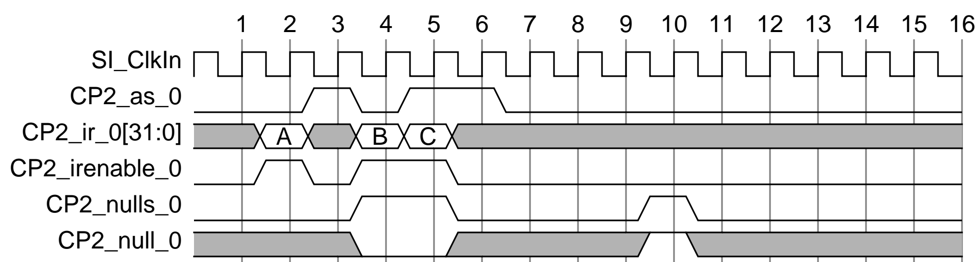

Figure 5-1 General Transfer Example

Figure 5-1 on page 60 above shows examples of the transfer of nullification information. All non-dispatch transfers follow the same protocol.

On edge 4, `CP2_nulls_0` is asserted, signifying the null transfer for instruction A. Since `CP2_null_0` is deasserted on edge 4, instruction A is not nullified. Instruction B is dispatched on edge 4 and it receives the null transfer in the next cycle at edge 5. Since it is the cycle after dispatch, this is the earliest possible time any transfer for instruction B could happen.Instruction C is dispatched at edge 5. The nullification transfer is delayed for some reason until edge 10. In this general example the instruction C is nullified. This will never happen on the M4K core, also the nullify strobe is always send in the cycle after dispatch on the M4K core.

For all transfers except To COP Data and From COP Data, the ordering of the transfers is simple: all transfers of a specific type (for example, nullification transfers) in a specific issue group must be in the same order as the order in which the instructions were dispatched. Other kinds of transfers can be interspersed—for example, if four arithmetic instructions were dispatched, there could be two nullification transfers, followed by four exception transfers, followed by two nullification transfers.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Note: If an instruction is killed or nullified, no remaining transfers for that instruction occur. In the cycle that the instruction is being killed or nullified, transfers may occur, but will be ignored. Additionally, if an instruction is killed, all instructions dispatched after the killed instruction are also killed.

The Coprocessor Interface is designed to operate with coprocessors of any pipeline structure and latency; if the M4K core requires a specific transfer by a certain cycle, then it will stall until the transfer has completed.

For transfers from the coprocessor to the integer unit, the allowable latencies are shown in Table 5-3. The “Stage Needed” column shows the integer unit pipeline stage where the data is used; if data is not available by the end of this stage, then the integer pipeline will stall. The “Min” column shows the minimum time after dispatch that the integer unit can accept the data (always one cycle). The “Max” column shows the maximum time after dispatch that the integer unit could receive the data (always an infinite number of cycles). The “Max Without Stalling” column shows the longest time after dispatch that the integer unit could receive the data without stalling.

Table 5-3 Allowable Interface Latencies from a Coprocessor to the M4K Core

From	To	Stage Needed	Min (cycles)	Max (cycles)	M4K Max Without Stalling (cycles)
Instruction Dispatch	Coprocessor Exceptions	M	1	∞	1
From COP Instruction Dispatch	From Coprocessor Data Transfer	M	1	∞	1
Branch Instruction Dispatch	Coprocessor Condition Code Check	E[a]	1	∞	-1[b]

Note: [a] The M4K cores does not have any branch prediction logic. Because of this, the new address (Branch taken or not) must be available in the E stage in order to have the address ready for the instruction following the branch delay slot.

Note: [b] The minus one (-1) indicates that the Coprocessor 2 Branch instruction will always cause a minimum of two stall cycles, while waiting for the Condition Code Check to be returned.

Because of its pipeline structure, the M4K core does not generate all allowable latencies for transfers from the integer unit to the coprocessor. Table 5-4 summarizes these latencies. The “Stage Sent” column shows the integer unit pipeline stage in which the transfer is performed. The “Min” column shows the shortest amount of time after dispatch that the integer unit will send the data. The “Max” column shows the longest time after dispatch that the data could be sent.

Table 5-4 Interface Latencies from the M4K Core to a Coprocessor

From	To	Stage Sent	Min (cycles)	Max
Instruction Dispatch	Instruction Nullification	E+1	1[a]	N/A
To COP instruction Dispatch	To Coprocessor Data Transfer	A	2	1 dispatch later (2 outstanding transfers)
Instruction Dispatch	Instruction Killing	A+1~	3	2 dispatches later (3 outstanding transfers)

Note: [a] The null strobe (CP2_nulls_0) is an OR function of the dispatch strobes (CP2_as_0, CP2_ts_0 and CP2_fs_0).

The “Max” latency is given in dispatches and thus defines the number of pending transfers to be made. It is the number of pending transfers that defines the interface logic required in the coprocessor.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

5.4.1 Instruction Dispatch

This transfer is used to signal the coprocessor to start coprocessor instructions. Data transfer instructions include those that move data to the coprocessor from the integer processor core (To COP Ops), and those that move data from the coprocessor to the integer processor core (From COP Ops).

Because data transfers for the To COP and From COP instructions occur later than the dispatch of the instructions, the coprocessor itself must keep track of data hazards and stall its pipeline accordingly. The integer processor core does not track coprocessor data hazards.

In a M4K core, instructions are dispatched to the coprocessor in the last cycle of the E-stage of the integer pipeline. Although the interface allows the coprocessor and integer pipelines to operate independently, it is important that the dispatch occurs to both in the same cycle to ensure that all subsequent transfers are properly synchronized. The M4K core may not dispatch a coprocessor instruction when the integer pipeline is stalled. This is necessary to allow proper CP0 exception handling.

CP2_as_0, *CP2_ts_0* and *CP2_fs_0* are asserted in the cycle after the instruction is driven. These signals are delayed strobe signals, and although this delay complicates the functional interface, it enables the processor to achieve very good timing on these signals. Without this delay, these signals would have been timing-critical.

Because the above instruction strobes are delayed, the coprocessor would normally be required to register *CP2_ir_0[31:0]* in every cycle and conditionally use it in the following cycle depending on the instruction strobes. This protocol has the side effect of registering non-coprocessor instructions and partially processing them, thus potentially increasing power consumption. The *CP2_irenable_0* signal compensates for this effect by enabling the coprocessor to avoid registering instructions that will never be dispatched to it. *CP2_irenable_0* low guarantee that this cycle is not a dispatch cycle. *CP2_irenable_0* high (1) indicates that this cycle might be a dispatch cycle. *CP2_irenable_0* is a late signal, making its timing critical. It should only be used to drive the enable input of the instructions latches.

Because of the tight relation between dispatch and required return from the coprocessor on the M4K core, it is recommended to do some amount of instruction decode in the dispatch cycle, and latch this decode based on *CP2_irenable_0*. This makes it more likely that data/exception returns from the coprocessor can be sent in the cycle after dispatch, and provide stall free operation in the M4K core.

Only one instruction strobe can be asserted at one time: *CP2_as_0*, *CP2_ts_0*, and *CP2_fs_0*.

CP2_inst32_0 and *CP2_endian_0* are both part of an instruction dispatch. They instruct the coprocessor to:

- work in MIPS32-compatibility mode (*CP2_inst32_0* high)
- Handle internal byte/halfword coprocessor instructions as big-endian operations (*CP2_endian_0* high)

Because the M4K core is a MIPS32-compatible core and does not support any MIPS64 specific features, the signal *CP2_inst32_0* is tied high (1).

The *CP2_endian_0* signals are asserted during dispatch to notify the coprocessor of the proper byte-ordering mode to use.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

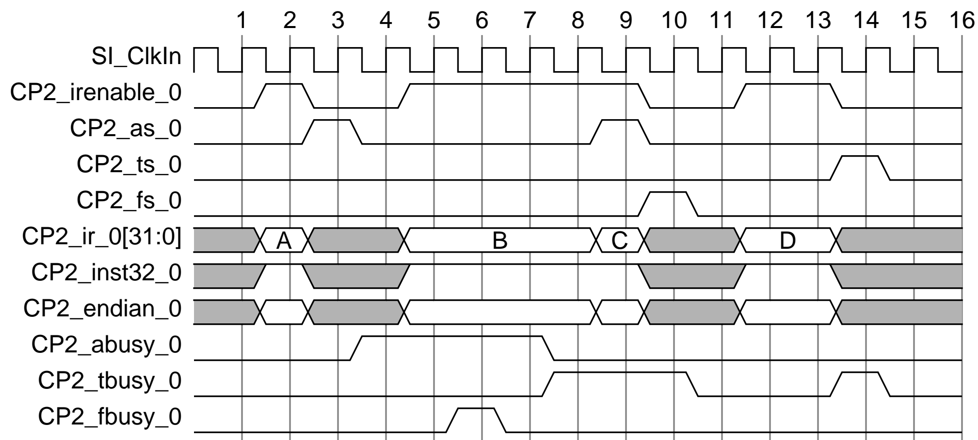

Figure 5-2 Instruction Dispatch Waveforms

Figure 5-2 shows example waveforms of four instruction dispatches.

- On edge 2, instruction A is dispatched. *CP2_ir_0[31:0]*, *CP2_inst32_0* and *CP2_endian_0* are all valid and *CP2_irenable_0* is driven high to indicate that this might be a dispatch cycle. On edge 3, instruction A is strobed as an arithmetic instruction by *CP2_as_0*.
- On edge 5, instruction B is valid on *CP2_ir_0[31:0]*. Instruction B is also an arithmetic instruction. because the *CP2_abusy_0* signal is detected high on edge 5, preventing arithmetic instruction strobes, the instruction is not strobed on edge 6. On edge 8, *CP2_abusy_0* is detected low, and the instruction is then strobed on edge 9 using *CP2_as_0*.
- On edge 6 CP2_fbusy_0 was asserted. Because no From COP Op instruction was attempted dispatched in this cycle this assertion is ignored.
- On edge 9, instruction C is dispatched. This is a From COP Op, requesting data from the coprocessor to be sent to the M4K core. *CP2_fbusy_0* is not driven high on edge 9, and thus instruction C is strobed on edge 10.
- On edge 12, instruction D is valid, and *CP2_irenable_0* is driven high. Instruction D is a To COP Op instruction. *CP2_tbusy_0* is not asserted on edge 12, but for some internal reason in the M4K core. Instruction D is not strobed until edge 14. On edge 14 *CP2_tbusy_0* is driven high from the coprocessor, but this is too late to prevent the instruction strobe on *CP2_ts_0*.

The CP2_abusy_0, CP2_tbusy_0 and CP2_fbusy_0 signals are the only means for the coprocessor to prevent the M4K core to dispatch instructions. When dispatched, all subsequent transactions for each instruction can happen immediately and the coprocessor must have buffers available to receive any information that might be transmitted from the core to the coprocessor. The reason to have 3 different instruction strobes is to enable a coprocessor to prevent one type of instruction

5.4.2 To Coprocessor Data Transfer

The Coprocessor Interface transfers data to the coprocessor after a To COP Op has been dispatched. Only To COP Ops utilize this transfer. The coprocessor must have a buffer available for this data after the To COP Op has been dispatched. If no buffers are available, then the coprocessor must prevent dispatch by asserting *CP2_tbusy_0*.

The Coprocessor Interface allows out-of-order data transfers. Data can be sent to the coprocessor in a different order from the order in which the instructions were dispatched. When data is sent to the coprocessor, the *CP2_torder_0[2:0]* signal is also sent. This signal tells the coprocessor if the data word is for the oldest outstanding To COP data transfer or the second oldest. The coprocessor can prevent the M4K from reordering To COP Data by driving *CP2_tordlim_0[2:0]* to 000_2.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Note: The M4K never sends data out of order. Thus *CP2_torder_0[2:0]* is tied to 000_2 and *CP2_tordlim_0[2:0]* is ignored.

Only word transfers are supported and the data is sent on *CP2_tdata_0[31:0]*.

The integer unit can transfer data to the coprocessor in the cycle after it is received from the memory subsystem. In the event of a cache miss, this can potentially happen many cycles after dispatch.

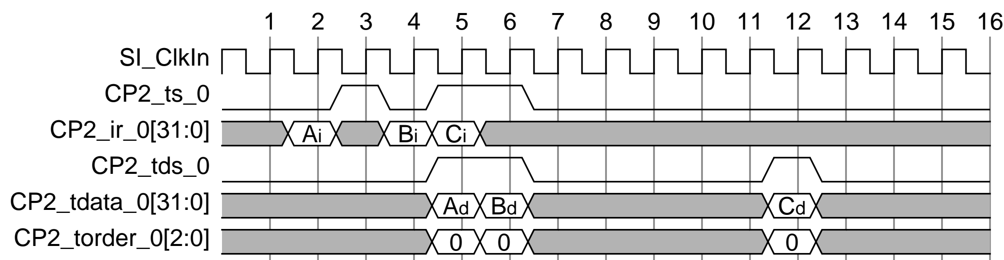

Figure 5-3 To Coprocessor Data Waveforms

Figure 5-3 shows waveforms for 3 To COP Op instructions and the data transfer associated with this instruction. On edges 2, 4 and 5 the To COP Op instructions A, B and C respectively are dispatched to the coprocessor. Because they are To COP Ops, the *CP2_ts_0* strobe is used to strobe the instruction dispatch.

On edge 5, the data associated with instruction A is valid. This is indicated by the *CP2_tds_0* driven high (1). Because *CP2_torder_0[2:0]* is 000_2 ties the data to the oldest outstanding To COP Op, which is instruction A.

On edge 6, data for instruction B is valid. This is the earliest after dispatch, that data will be sent from the M4K core. The interface must however support data to be sent as early as the cycle after dispatch (edge 5 for instruction B) to be compliant with other MIPS cores using the Core Coprocessor Interface.

Data for instruction C is not sent until edge 12. This could be due to a data-cache miss, but could have many other M4K core internal reasons. The Coprocessor must support any cycle delay from instruction dispatch to data transmit on To COP Ops.

5.4.3 From Coprocessor Data Transfer

The Coprocessor Interface transfers data from the coprocessor to the integer processor core after a From COP Op has been dispatched. Only From COP Ops utilize this transfer. Note that the M4K core has buffers for this data that enables the transfer to occur as early as the cycle after dispatch.

The Coprocessor Interface allows out-of-order transfer of data. That is, data can be sent from the coprocessor in a different order from the order in which the instructions were dispatched. When data is sent from the coprocessor, the *CP2_forder_0[2:0]* signal is also sent. This signal tells the integer processor core if the data is for the oldest outstanding From COP data transfer or the second oldest. The M4K core supports a maximum of 1 out-of-order transfer and drives *CP2_fordlim_0[2:0]* = $1\ 001_2$.

Note: It is illegal for a coprocessor to drive *CP2_forder_0[2:0]* > $1\ 001_2$.

Only word transfers are supported, and the data must be sent on *CP2_fdata_0[31:0]*.

For both memory stores (SWC2) and move instructions (MFC2/CFC2), the integer pipeline can stall if data is not available by the M stage. This is because the data to be stored/moved to a register is needed early in the following A-stage. By receiving the data in the M-stage, the Coprocessor Interface can have non-critical timing.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

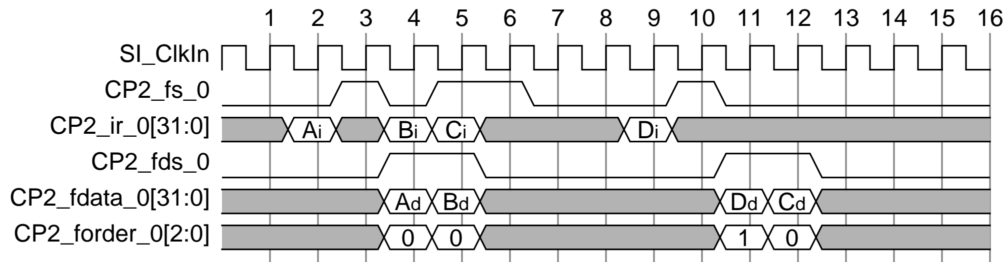

Figure 5-4 From Coprocessor Data Waveforms

Figure 5-1 shows example waveforms for 4 From COP Op instructions, and the data transfer associated with these instructions. On edge 2, 4, 5 and 9 the From COP Ops A, B, C and D respectively are dispatched from the integer core. They are all From COP Ops, thus *CP2_fs_0* is used to strobe the instruction.

On edges 5 and 6, data for instruction A and B are returned from the coprocessor. The data is returned in order of instruction dispatch, and *CP2_forder_0[2:0]* is consequently driven to 000_2. Data for instruction B is sent in the cycle after dispatch. This is needed to ensure stall free operation in the M4K core. The data for instruction A is one cycle delayed, causing one stall cycle in the M4K core.

On edge 11, data for instruction D is returned to the integer core. This is the second oldest outstanding data transfer, *CP2_forder_0[2:0]* is driven to 1 001_2 to indicate one out of order in the data transfer.

On edge 12, the data for instruction C is finally returned. *CP2_forder_0[2:0]* is driven to 0 000_2 because this is the oldest outstanding data transfer.

5.4.4 Condition Code Checking

The Coprocessor Interface provides signals for transferring the result of a condition code check from the coprocessor to the integer processor core. Only BC2 instructions utilize this transfer. These instructions are dispatched to both the integer processor core and the coprocessor.

For each instruction dispatched, a result is sent back to the integer processor core that says whether or not to take the branch.

For this reason, the coprocessor must interpret the type of instruction to decide whether or not to execute it. Customer-defined BC2 instructions are thus possible. Four main flavors of BC2 instructions exists (BC2T, BC2TL, BC2F and BC2FL). The integer core does not care if it is a True or False branch. It will only distinguish between a branch and a branch likely type instruction. The coprocessor is the unit that determines if the branch should be taken or not. A taken branch is indicated by asserting the condition code check *CP2_ccc_0* = 1. The not taken branch is indicated by *CP2_ccc_0* = 0.

With the M4K core, the address of the second instruction following a branch is calculated in the branch instruction’s E-stage, which is the dispatch stage. The condition contributes to the address calculation. The BC2 instruction is dispatched to the coprocessor, but stalled in the IU’s E-stage until the coprocessor returns the condition result.

The condition code check from the coprocessor is registered on the input to the M4K core. The values are not available until the cycle after return from the coprocessor.

Note: The M4K core always stalls for a minimum of 2 cycles in E-stage for any BC2 instruction sent to the coprocessor.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

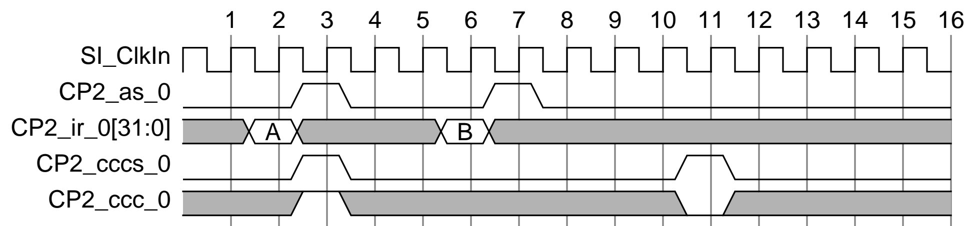

Figure 5-5 Condition Code Check Waveforms

Figure 5-5 shows an example waveform for two BC2 instructions. BC2 instructions belong to the arithmetic COP Op group of instructions and the dispatch is thus strobed using the CP2_as_0 strobe.

On edges 2 and 6, BC2 instructions are dispatched from the integer unit. The condition code check for instruction A is returned as fast as possible, which is on edge 3. This means that the stall penalty was kept at the minimum of 2 cycles. *CP2_ccc_0* is set (1_2) indicating to the integer core to go ahead and take the branch.

On edge 11, condition code for instruction B is returned. The four cycle extra delay means that the M4K core will stall for a minimum of 6 cycles for this BC2 instruction. *CP2_ccc_0* is driven low indicating to the integer core that the branch is not to be taken.

5.4.5 Coprocessor Exceptions

All instructions dispatched utilize this transfer. It is used to signal if an instruction caused an exception in the coprocessor. This transfer must happen even if the instruction did not cause an exception in the coprocessor.

When a coprocessor instruction causes an exception, the coprocessor must signal this to the integer processor core so it can start execution from the exception vector. The coprocessor can signal a Reserved Instruction exception (RI) for any instruction dispatched.

Signalling for Reserved Instruction exceptions is divided between the integer processor core and the coprocessor as follows:

- The integer processor core signals Reserved Instruction exceptions for non-arithmetic coprocessor instructions that are not valid To COP Ops or From COP Ops:
  - (IR[31:26] = 010010_2) & (IR[25:24] = 00_2) & (IR[22:21] = 11_2): Reserved To/From COP Ops.
  - (IR[31:26] = 010010_2) & (IR[25:24] = 00_2) & (IR[22:21] = 01_2): unimplemented DMFC2/DMTC2 COP Ops.
  - (IR[31:30] = 11_2) & (IR[28:26] = 110_2): unimplemented LDC2/SDC2.
- The coprocessor hardware must signal Reserved Instruction exceptions for all unimplemented arithmetic coprocessor instructions:
  - (IR[31:26] = 010010_2) & (IR[25] = 1_2) & (IR[24:0] = unimplemented COP2 instruction)
  - (IR[31:26] = 010010_2) & (IR[25:24] = 01_2) & (IR[23:21] = unimplemented Branch instruction).

Note: The M4K core does not dispatch the instructions that it is responsible for RI exception signaling. This might not be the case for other integer cores featuring this interface. In this case, the instruction can always later be nullified or killed. A fully compliant coprocessor must be able to handle this and is allowed to signal no-exception on these instructions.

The coprocessor should only signal Coprocessor 2 exceptions (C2E) for any implemented COP2 instruction which has an execution problem. All unimplemented legal COP2 instructions should signal an RI exception.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Note: For imprecise exceptions, the exception sent is not related to the current instruction, the C2E exception can only be sent on dispatched COP Ops that are NOT part of the instructions that the integer core are guaranteed to signal RI as defined above.

The coprocessor may also signal one of two implementation-specific exception codes (IS1 and IS2). These exception codes can be used to trigger special software exception handling routines. A special handler can be started quicker as the exception handler does not need to read a specific coprocessor *Cause* register, as might be needed on the general C2E exception. The rules for C2E exception also apply to IS1 and IS2 exceptions.

Note: A coprocessor can signal an exception for all To/From COP Ops. An exception on a To/From COP Op cannot depend on the associated data, except for the data sent from the integer core on a CTC2 instruction[1].

The integer processor core detects Coprocessor Unusable exceptions for all coprocessor instructions.

The M4K core needs the exception transfer for all instructions in the M-stage to avoid stalling. It must signal exceptions in the first cycle of the A-stage, and will stall in the M-stage if it has to wait for the transfer.

If imprecise coprocessor exceptions are allowed, then the coprocessor can use the "No exception" signal immediately after dispatch. This will prevent stalling in the integer pipeline while waiting for precise results; if an exception does occur for that instruction, then a subsequent coprocessor instruction can be flagged as exceptional (although imprecise), or else an interrupt could be signalled through the normal integer processor core interrupt inputs (*SI_Int[5:0]*).

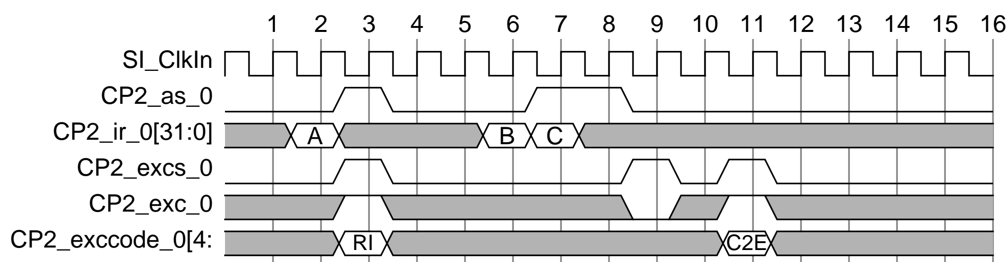

Figure 5-6 Exception Waveforms

Figure 5-6 shows example waveforms for an exception return from three coprocessor instructions. In this example, the exception returns are all arithmetic COP Ops, and *CP2_as_0* is used to strobe the dispatch.

On edges 2, 6 and 7, instructions A, B and C respectively are dispatched. A is an unimplemented arithmetic instruction, causing a Reserved Instruction exception (RI). B is an implemented arithmetic instruction, as is C, but some errors occurred while executing the instruction, causing a C2E exception.

On edge 3, an RI exception for instruction A is returned to the integer core. *CP2_excs_0* set (1_2) signals that the *CP2_exc_0* is valid. *CP2_exc_0* driven high (1_2) signals that a valid exception is on CP2_exccode[4:0]. Refer to Table 2-3 on page 3 for descriptions of the valid exception bit values.

On edge 9, no exception is returned for instruction B. On edge 11, the C2E exception for instruction C is returned to the integer core.

[1] Exception based on the data sent on a CTC2 is possible if the control value written indicates that the instruction should always cause exception.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

5.4.6 Instruction Nullification

All instructions dispatched utilize this transfer. Used to signal if an instruction was nullified in the integer processor core, this transfer happens even if an instruction was not nullified so that the coprocessor knows when it can begin operation of subsequent operations that depend on the result of the current instruction.

Normally, an instruction is killed only when the pipeline is being flushed because an exception occurred. In this case, all subsequent instructions in the pipeline (both coprocessor and integer core pipelines) are also killed. An instruction may also be killed because it is in the delay slot of a branch-likely instruction that did not branch. This type of killing is called instruction nullification. In this case, subsequent instructions in the pipeline are unaffected by the nullification.

Nullification must be performed in an early stage of the pipeline to ensure that subsequent instructions can begin with the correct operands.

In the cycle that an instruction is nullified, other transfers for that instruction may still occur, but no further transfers for that instruction can occur in subsequent cycles. Exceptions caused by a nullified instruction are masked by the integer processor core.

Note: The M4K core never nullifies an instruction. No nullify is always transferred in the cycle after dispatch.

Nullification transfers follow the generic example given in Figure 5-1 on page 60.

5.4.7 Instruction Killing

All instructions dispatched utilize this transfer. This is used to signal if an instruction can commit state or not. This transfer happens even if an instruction is not being killed so that the coprocessor knows when it can writeback results for the instruction.

Due to various exceptional conditions, any instruction may need to be killed. The integer processor core contains logic which tells the coprocessor when to kill coprocessor instructions.

When a coprocessor instruction is being killed because of a coprocessor-signalled exception, the coprocessor may need to perform special operations. For example, if an arithmetic COP2 instruction signalled a C2E exception, then later is killed due to this exception. Some internal status bits might need to be updated before clearing the pipe. On the other hand, if that same instruction was killed because of a higher priority exception, those status bits must not be updated. For this reason, as part of the kill transfer, the integer processor core tells the coprocessor if the instruction is killed due to a coprocessor-signalled exception or not.

When a coprocessor instruction is killed, all subsequent coprocessor instructions that have been dispatched are also killed. This is necessary because the killed instruction(s) may affect the operation of subsequent instructions (for example, because of bypassing). In the cycle in which an instruction is killed, other transfers may occur, but after that cycle, no further transfers occur for any of the killed instructions. A side-effect of this is that the other instructions that are killed do not have a kill transfer of their own. In effect, they are immediately killed and thus their remaining transfers cannot be sent, including their own kill transfer. Previously nullified instructions do not have a kill transfer either, because once nullified, no further transfers can occur.

Note: If the integer processor core dispatches a coprocessor instruction in the same cycle that a kill is being signalled to the coprocessor, then that instruction is also considered killed.

The integer unit knows in an instruction's A stage whether the instruction is to be killed or not. In order to avoid critical timing signals being passed directly to the coprocessor, the integer unit will register its A stage kill signal before sending it to the coprocessor.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

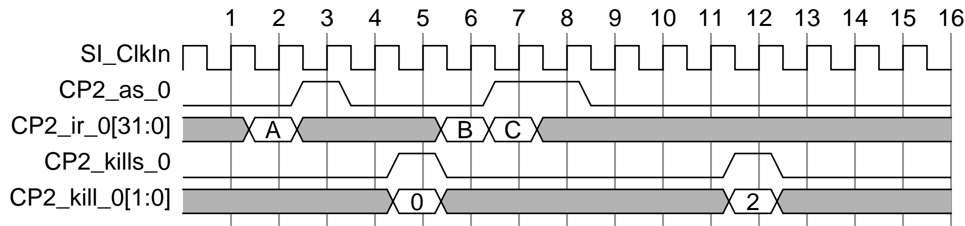

Figure 5-7 Instruction Killing Waveforms

Figure 5-7 shows example waveforms for instruction killing.

On edges 2, 6 and 7, instructions A, B and C are dispatched.

On edge 5, instruction A is notified of a no-kill. This instruction can now commit internal state and register writes in the coprocessor.

On edge 12, instruction B is killed. The value of (10_2) on *CP2_kill_0[1:0]*, indicates that the instruction was not killed due to an exception sent by itself. Instruction B therefore does not commit any state or register bits in the coprocessor. If *CP2_kill_0[1:0]* was (11_2), then the B instruction could commit state bits, indicating the cause of the exception it sent (not shown).

Instruction C never gets a *CP2_kills_0* strobe, because the killing of instruction B also killed instruction C. An indirectly killed instruction like instruction C can never commit any state or register bits in the coprocessor.

5.5 Power Saving Issues

The power saving issues have already been touched on in the previous sections. This section specifies what to do and what not to do in order to minimize power dissipation in the M4K core and the coprocessor.

5.5.1 No coprocessor Present

If a hard-core version of a M4K core is being used that includes the Coprocessor Interface, but there is no plan to connect a coprocessor to the core, then the following must be observed:

- Tie *CP2_present* low (0). Tying this input low, will prevent any use of the Coprocessor Interface.
- Tie all strobe inputs (*CP2_fds_0*, *CP2_cccs_0* and *CP2_excs_0*) low (0). If the M4K core is implemented using gated clocks on local registers, then the strobe inputs on each bus are used as the enable signal in the clock gating logic for the input capture registers.
- Tie all other inputs to a static value. All other inputs are ignored, when *CP2_present* is low (0).

The above rules are very simple to implement. Tie all *CP2_xx* and *CP2_xx* inputs to the M4K core low (0) if there is no coprocessor attached to the integer core.

5.5.2 How to Use *CP2_idle*

CP2_idle is an input to the M4K core. When a coprocessor is attached to the core, it is important to use this input properly in order for the **WAIT** instruction to work effectively.

The **WAIT** instruction enables power saving features within the M4K core. When **WAIT** is executed, the M4K core will stall the front of the pipe, and wait for all older instruction and pending bus activity to complete. Once this is detected,

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

all but about one hundred flops have their clock gated off via one top-level clock gating circuit. The only way to reawaken the core is to signal an interrupt on *SI_Int[5:0]*, *SI_NMI* or *EJ_DINT*, or by resetting the core using *SI_Reset* or *SI_ColdReset*.

While the WAIT instruction ensures that no new instructions go down the pipe in the integer core, nothing is implicitly done to tell the coprocessor to prepare for a possible stopping of its clock. This is where the *CP2_idle* signal is used. The coprocessor must assert this signal high whenever no instruction execution occurs within the coprocessor. *CP2_idle* is part of the logic that determines when the top level clock gating element can turn off the clock. If this signal is deasserted then the clock will never be gated off in the M4K core, and the whole purpose of the **WAIT** instruction is lost. The *CP2_idle* input is ignored when *CP2_present* is low.

It is important to note that the *CP2_idle* input *cannot* be used to reawaken the M4K core. After the **WAIT** instruction has actively stopped the main clock to most of the M4K core flops, a deassertion of *CP2_idle* will restarts this clock but leaves the processor issuing NOPs down the pipe. The coprocessor cannot awaken the core by deasserting *CP2_idle.* If some external source requires service from either the integer core or the coprocessor (via the integer core), then this external source must assert an interrupt directly to the M4K core.

5.5.3 Gating the Clock to the Coprocessor

For power reasons, the designer of the coprocessor is encouraged to use a top-level clock gater on the clock tree distributed within the coprocessor. The M4K core has an output, *SI_Sleep,* which indicates when the internal clock in the integer core is stopped. Figure 5-8 shows an example of how to implement and control a top-level clock gater in the coprocessor.

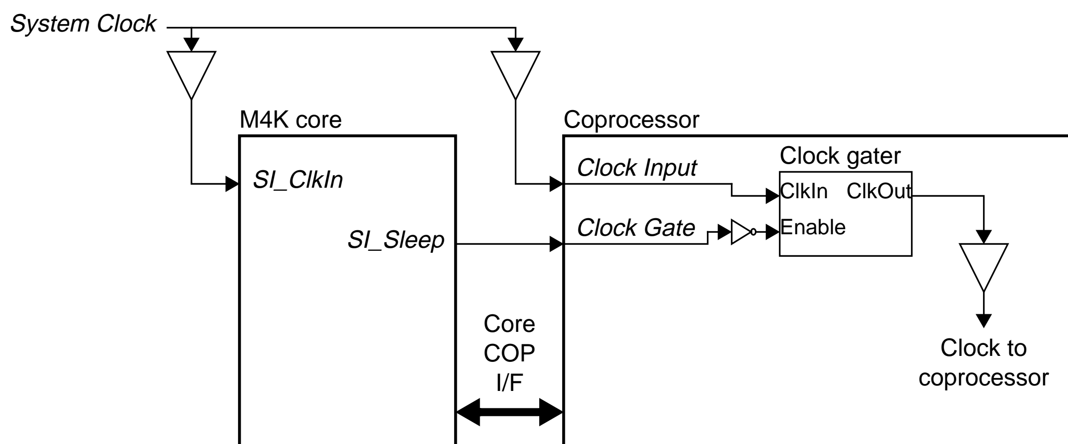

Figure 5-8 Use of SI_Sleep for Clock-Gating in the Coprocessor

5.5.4 Using strobe signals as gating inputs on the sub-interfaces

Each of the sub-interfaces of the Coprocessor Interface has a strobe signal associated with it.

Figure 5-9 on page 71 shows how this strobe signal can be used as the enable input to a clock gater driving the clock to the corresponding data portion of the interface. The “To Data” interface is shown as an example. Instruction nullification and instruction killing can use the same scheme, but the low number of bits in the data portion of these two sub-interfaces might not make it worth the effort.

The instruction dispatch interface is different as its strobe signals arrive one cycle after the instruction word.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

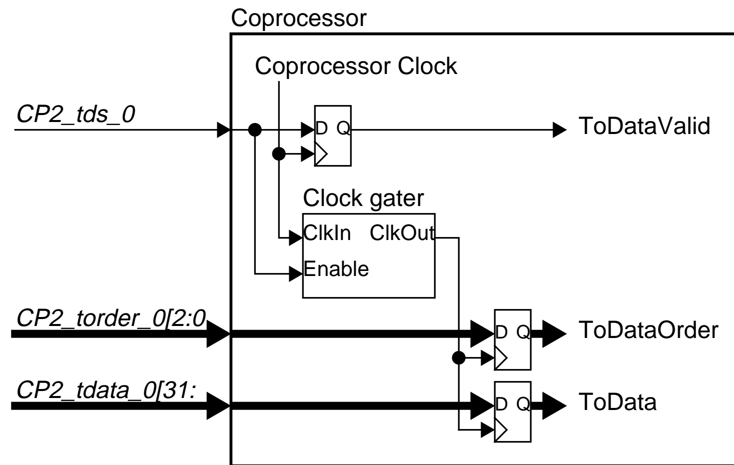

Figure 5-9 Clock-Gating of To Data Registers in Coprocessor

Figure 5-10 shows the intended use of *CP2_irenable_0*. *CP2_irenable_0 is used* only as a gated-clock enabling signal when the clock-gating on the capture of the instruction word is introduced. For all other purposes, the *CP2_as_0*, *CP2_ts_0* and *CP2_fs_0* are the true qualifiers for a valid instruction.

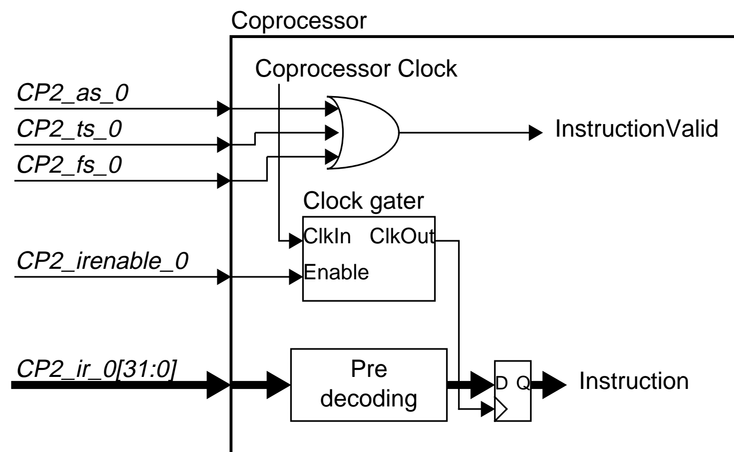

Figure 5-10 Clock Gating of Instruction Registers in Coprocessor

The Pre-decoding block in Figure 5-10 represents combination logic before the receiving flops for the instruction register. This block is most likely needed before the Instruction register if stall-free operation on coprocessor instructions in the M4K core is to be maintained. Refer to Table 5-4 on page 61, for information on allowable latencies to maintain stall-free operation.

5.6 Template for Coprocessor Modules

A template for coprocessor 2 modules is included in the processor release. This template provides a simple implementation of Cop2 interface logic. It can be used as is for many coprocessor designs or can be used as a reference for designing coprocessor interface logic. There is an application note, *Core Coprocessor 2 Module Template Application Note* (MD00130) in the `$MIPS_PROJECT/doc` directory as well as RTL in the `$MIPS_PROJECT/ref_design/cop2tpl` directory.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

VMC Simulation Model

This chapter discusses the simulation models included in a MIPS32™ M4K™ core release. It contains the following section:

- Section 6.1, "Cycle-Exact Simulation Model"

6.1 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 M4K processor core.

6.1.1 Installing the VMC Model

1. The M4K VMC model is supported under the Sun Solaris UNIX and x86 RedHat Linux platforms.
2. The M4K VMC model is a SWIFT R41 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 where the installation is to reside:
```
% setenv LMC_HOME <your_install_path>
```
   In a normal MIPS32 M4K soft core installation, for example, a local LMC_HOME location might be set like this:
```
% cd $MIPS_PROJECT
% mkdir vmc_install
% setenv LMC_HOME $MIPS_PROJECT/vmc_install
```
3. Invoke the admin install tool supplied in the top level of the release package for the VMC model:
```
% $MIPS_PROJECT/vmc[_sun,_linux]/mm4k_vmc_release/sl_admin.csh
```
   1. A dialog box labeled “Install From...” should pop up.
   2. Make sure the text input box points to the package, “mm4k_vmc_release”.
   3. Press “Open” to continue.
   4. Another dialog box is used to select the models that will be installed. Only one choice is available in this release, a model called “mm4k_vmc_model” followed by a version number. Click on that model to bring it into the “Models to Install” window.
   5. Click “Continue” to close this dialog box.
   6. Another dialog box to select the platforms for this model installation will appear. Each release package will only contain the model for one platform and that check box should be selected. The appropriate simulator packages used under the “EDAV Packages” heading also need to be specified. Both Verilog-XL and NC-Verilog are covered by the “Cadence Design Systems” push button. Modelsim and VCS have their own buttons. Multiple EDAV packages can be selected and the packages for all simulators that will be used should be selected. Push the “Install” button to continue.
   7. An “Install complete” message in the main message window is received and then exit from the sl_admin tool.
4. During the installation, a documentation directory will be created at `$LMC_HOME/doc`. There are pdf files in this directory structure that contain additional details about the install process, administering and using SmartModels, and licensing.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

5. The M4K VMC model requires a GLOBEtrotter FLEXlm license in order to run. This license can be received push button 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.`
6. For Linux installations only: A directory needs to be added to the LD_LIBRARY_PATH to make the VMC model work.
   - $LMC_HOME/lib/x86_linux.lib/

```
% setenv LD_LIBRARY_PATH $LMC_HOME/lib/x86_linux.lib:$LD_LIBRARY_PATH
```

6.1.2 Verifying the VMC Installation

A utility called `swiftcheck` is available in the VMC installation to ensure that the model and environment variables are set up properly. This command must be run before attempting to simulate with the M4K VMC model. Invocation is as follows:

```
% $LMC_HOME/bin/swiftcheck mm4k_vmc_model
```

The file `swiftcheck.out` is produced by the command. Check it to verify that there are no errors as reported at the end of the file.

6.1.3 SWIFT Template Generation

In order to instantiate the M4K VMC model in the RTL simulation environment, a SWIFT template of the M4K VMC model needs to be created, which is then instantiated in the RTL design. This template file provides a conversion from the VMC model to the simulator’s SWIFT interface. The SWIFT template is simulator-specific, so simulator documentation provides additional details on creating a SWIFT template, including the template in the design.

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

```
% vcs -lmc-swift-template mm4k_vmc_model
```

To generate a SWIFT template for Verilog-XL, NC-Verilog, and ModelSim, a script called `vsg` that is included in the `$LMC_HOME/bin` area of the installed VMC area is used (This script is included as part of the Cadence EDAV package as described in step 3.6 above). The invocation is:

```
% $LMC_HOME/bin/vsg -z mm4k_vmc_model
```

Two example templates are included in the `$MIPS_PROJECT/vmc_sun/verification` directory.

6.1.4 Back-Annotating with SDF Timing

This is not supported.

6.1.5 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 6-1 shows the signals available via register windows.

Table 6-1 Core Signals Visible in VMC model

Name	Bits	Description
CPZ_x	[31:0]	Contents of Coprocessor 0 register xxx. The following registers are available: Context, Count, Compare, BadVA, Status, Cause, EPC, DEPC, ErrorEPC, DeSave, Debug, Config0, and Config1. .

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 6-1 Core Signals Visible in VMC model (Continued)

Name	Bits	Description
RFx_xx	[31:0]	Contents of the General Purpose Register File. Shadow register sets are denoted by the first number, the register number by the second.

6.1.5.1 Enabling VMC Window Signals in Synopsys VCS

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

```
initial $swift_window_monitor_on("<instance_path_to_mm4k_vmc_model>");
```

6.1.5.2 Enabling VMC Window Signals in Other Verilog Simulators

For Verilog-XL, NC-Verilog, and ModelSim, every window signal to be viewed needs to be individually specified. The code required is most easily placed in the SWIFT template produced by the `vsg` command, as described in Section 6.1.3, "SWIFT Template Generation". The format of the enabling code is:

```
$lm_monitor_vec_map(<verilog_register>, "<instance_path_to_mm4k_vmc_model>",
"<window_signal_name>");
```

In the SWIFT template created by `vsg`, the <verilog_register> statements exist in the template but are dangling. Dangling registers can be used 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_mm4k_vmc_model>", "RF1");
   $lm_monitor_vec_map(RF2, "<instance_path_to_mm4k_vmc_model>", "RF2");
   ...
   end
```

6.1.6 VMC Simulation Configuration

The VMC model is configurable so that all functionally visible features of the M4K core are visible. The available options are shown in Table 6-2 and include processor type (M4K core), selection of various functional features within the core, and debug switches that determine whether optional trace files are produced. The configuration is performed by setting up a memory file which is read in and used to select between the different modules. The memory file is called `memory.m4k_config` and needs to be in a SWIFT readmem format which is:

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

The available configuration options are shown in Table 6-2.

Table 6-2 VMC Configuration Options

Name	Addr (hex)	Description	Legal Values	Default
LITEMDU	7	Choose multiply/divide unit (MDU) type.	0 - Fast, high-performance MDU 1 - Small, iterative MDU	0
EJSModule	8	Which EJTAG simple break module should be used.	0 - No SB 1 - 2I/1D SB 2 - 4I/2D SB	2

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 6-2 VMC Configuration Options (Continued)

Name	Addr (hex)	Description	Legal Values	Default
EJTModule	9	Use EJTAG TAP module.	0 - No TAP 1 - Use TAP	1
Inst	A	Unique instance identifier. Tags output messages and trace files to more easily support multiple instances. Must be specified as a hex value.	0 - 3f (hex; corresponds to 0-63 decimal)	0
dispEn	B	Display Enable. Controls printing of warning or error messages coming from the VMC model.	0 - No messages 1 - Messages	1
bus_trace	D	Enables logging of all transactions on the cores EC interface (external bus) to file `vmc.bus[.Inst].trace`.	0 - No log 1 - Log bus transactions	1
dumpTrace	E	Enables instruction trace to file `vmc[.Inst].trace`.	0 - No tracing 1 - Trace file will be created	0
MIPS16e	F	Indicates that MIPS16e decoder is present	0 - No MIPS16e support 1 - MIPS16e support is present	0
CP2Module	11	Include Coprocessor 2 interface module.	0 - No CP2 Interface 1 - CP2 Interface included	1
PDTModule	12	Include EJTAG PDtrace and Trace Control Block modules	0 - No trace blocks 1 - PDtrace and TCB blocks included	0
UDI	13	Indicates that user-defined instruction (UDI) features are present. This field only affects the setting of a bit in the CP0 register (*Config.UDI*). The VMC model does not emulate the actual function of UDIs.	0 - No UDI present 1 - UDI is present	0
Gated clocks for ucreg	14	Indicates whether gated clocks are used internally for certain unconditional registers whose state is a logical don't care in certain situations. This field does not affect the instruction-level or cycle-by-cycle functionality of the core, but can affect the state as seen at the pins.	0 - No gated clocks for ucregs 1 - Gated clocks present for ucregs	1
PDtrace dump enable	17	This bit enables the creation of files tracing activity on the internal PDtrace and TCB interfaces, to files `vmc[.Inst].pdtrace` and `vmc[.Inst].tcbtrace`.	0 - No tracing 1 - tracing enabled	0
TCB On-chip	18	Select whether the TCB (Trace Capture Buffer) has an on-chip memory interface or not	0 - No on-chip memory 1 - On-chip memory present	1
TCB On-chip Size	19	Size of the on-chip TCB memory in 64-bit trace words. Must be specified as a hex value.	5-14 (hex; 5-20 decimal): On-chip memory is 2^^N trace words	14
TCB Off-chip	1A	Select whether the TCB has an off-chip memory interface or not	0 - No off-chip memory I/F 1 - Off-chip memory I/F present	1
TCB Triggers	1B	Number of TCB trigger registers implemented	0-8: N trigger registers	8

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Table 6-2 VMC Configuration Options (Continued)

Name	Addr (hex)	Description	Legal Values	Default
PIB Data Width	1C	Number of bits for the TRDATA port to the Probe Interface Block (PIB). Must be specified as a hex value. Only valid for Lead Vehicle VMC models	4,8,10 (hex; corresponds to 4,8,16 decimal)	8
Global Clock-gate	1F	Selects whether the global clock gating for the WAIT instruction is enabled. This switch will change the exact cycle behavior just before, during and after a WAIT instruction.	0 - No Global clock-gating 1 - Global clock-gaiting enabled	1
SRAM Interface	21	Selects between dual and unified SRAM interfaces.	0 - Unified I/F 1 - Dual I/F	0
GPR Shadow Sets	23	Selects the total number of General Purpose Register shadow sets	1 - One GPR is present 2 - Two GPR sets are present 4 - Four GPR sets are present	1

An example `memory.m4k_config` file is shown below:

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

#LITEMDU
7/0;
#EJSModule
8/2;
#EJTModule
9/1;
#Inst
A/0;
#dispEn
B/1;
#haltIt
C/1;
#bus_trace
D/1;
#dumpTrace
E/1;
#Cop2 Interface Module
11/1;
```

6.1.7 Trace Files

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

6.1.7.1 SRAM Interface Trace

The SRAM trace file (`vmc.bus[.Inst].trace`) contains information about all transactions on the SRAM interface. The fields in this file are:

- Type: Transaction type: RI- Instruction read, RD- Data read, W- Data write.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

- BE<3:0>: Byte Enables - indicates which byte lanes are active for this transaction.
- Addr<31:0>: Address value.
- RData/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.
- RBE<3:0>: Read Byte Enables - indicates which byte lanes of the incoming data were valid.
- Side: Indicates whether request was on the I-side or D-side SRAM interface.
- Stall: Indicates when partial data is returned (RBE asserted with Stall)
- Abort: Indicates when an abort was requested and if it was acknowledged: xA - abort was requested but not acknowledged, A - abort was requested and acknowledged, 0 - no abort requested
- L/U - Indicates that a lock or unlock request was made and whether it was successful, L- lock requested, U - unlock requested and successful, xU - unlock requested but not successful, 0 - no lock/unlock requested
- Ejt - Indicates that there was an EJTAG break that cancelled this request.
- Error - Indicates whether there was an error signalled on this request.
- Cycle: Indicates a cycle number when this transaction completed. (Cycles are counted from the falling edge of the first Cold Reset).

6.1.7.2 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 0 Cyc:6 ]bfc00000 1fc00000 2:  00000000    NOP
|<-----a------>|<--------b-------->|<-------c------->|
```

a) Each line is tagged with an instruction number, sequence number, and a cycle number. Gaps in the instruction number sequence can occur near exceptions. The sequence number indicates a sub-instruction in a macro sequence (SAVE/RESTORE instructions). This will be 0 for instructions that are not part of a macro sequence. 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 M4K core, the CCA values are not used and are not traced. The CCA will always be reported as 2 (uncacheable).

c) This field is the instruction opcode and disassembly.

```
[Ins:954 0 Cyc:8166 ]Write GPR[26][1]= 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. The second bracketed term indicates the shadow set for GPR writes. It is omitted if the write is to shadow set 0.

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.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

```
[Ins:972 0 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 went to memory.
- Probe Read indicates a load that went to DRSEG in EJTAG space
- Mem Write indicates a store that went to memory
- Probe Write indicates a store that went to DRSEG in EJTAG space

g) This is the virtual address, physical address, and cache coherency attribute for the data access.

```
[Ins:127 0 Cyc:1838 ]# Branch Taken
[Ins:2 0 Cyc:0 ]# PDT Mode Change 0: AllowOverflow TraceNormalBranch
|<-----a------>|<-------------------------h------------------------>|
```

h) Lines beginning with a # are comments. These do not track architectural state. These comments provide additional information about program flow and processor state that is used in our internal verification environment. For example, the two lines above show a comment tracking a branch condition and one indicating the PDtrace mode.

6.1.8 Simple Testbench

To simplify bring-up of the VMC model, a simple testbench is included in the directory `$MIPS_PROJECT/vmc_sun/verification`. 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 M4K 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, a Verilog template is needed. This template is specific to the simulator (including the particular version in some cases). See Section 6.1.3, "SWIFT Template Generation" for details on how to create a template. There are two sample templates in the `verification` directory: `mm4k_vmc_model.vcs.v` is a template for vcs, and `mm4k_vmc_model.vxl.v` is a template for Verilog-XL, ModelSim, and NC-Verilog.

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

6.1.9 Multiple VMC Instances

It is possible to instantiate multiple M4K VMC models to simulate a multi-CPU system. The SWIFT template file is parameterized to control which configuration file is read. 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.m4k_config` and `memory2.m4k_config` configuration files. Note that the unique configuration files with the desired options for each instance must be manually created, as described in Section 6.1.6, "VMC Simulation Configuration" on page 74.

```
mm4k_vmc_model vmc1 (....);
defparam vmc1.InstanceName = “vmc1”;
defparam vmc1.MemoryFile = “memory1”

mm4k_vmc_model vmc2 (...);
```


Copyright © 2002 MIPS Technologies Inc. All rights reserved.

```
defparam vmc2.InstanceName = “vmc2”;
defparam vmc2.MemoryFile = “memory2”;
```

6.1.10 Assertion Checks

A variety of assertion checks are embedded within the M4K VMC model. These checkers look for error conditions and unknown states 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 states on internal signals. These errors can be caused by faulty software or incorrect chip hook up.
- XWarning - This indicates an 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 might be the desired behavior in the system testbench or simulation environment. 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 an insufficient reset pulse width.
- Fatal Config Errors - These errors indicate that the processor configuration is not valid.

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 6.1.6, "VMC Simulation Configuration" on page 74 for more details.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Chapter 7

Clocking, Reset and Power

This chapter describes the clocking and initialization interface on a MIPS32™ M4K™ processor core, when the core is integrated into a system environment. The power-reduction features available on a M4K core are also discussed.

This chapter contains the following sections:

- Section 7.1, "Clocking"
- Section 7.2, "Reset and Hardware Initialization"
- Section 7.3, "Power Management"

7.1 Clocking

There are potentially two input clocks that must be generated and driven to a M4K core. The main clock input is named *SI_ClkIn,* and exists on every M4K 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 M4K 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 M4K 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 M4K core that is being integrated before proceeding with system clock design.

7.1.1 *SI_ClkIn* Clock

SI_ClkIn is the primary 1x input clock to the M4K core and 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 M4K core.

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 M4K core implementation.

7.1.2 *EJ_TCK* Clock

EJ_TCK is an optional 1x clock input to the M4K core, only existing 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, and 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 that 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 40 MHz.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

7.1.3 Handling Clock Insertion Delay

When a M4K 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 M4K core implementation.

The clock insertion delay presents an issue that must be managed when the M4K 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 most M4K core inputs are received directly by flops, and most 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 M4K core's internal clock insertion delay:

- Tolerate the core clock insertion delay at the system level, if possible, within the system logic that interfaces to the M4K core. This may entail adding delay elements when driving inputs, so hold times are not violated, and receiving "late" outputs, reducing 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 that interfaces to the M4K 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 M4K core that 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 M4K core. The difference between *SI_ClkIn* and *SI_ClkOut* represents the clock insertion delay of the primary clock used within the M4K 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.

7.2 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 M4K 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 M4K core, and not left floating or indeterminate. Each of the reset-related *SI_** inputs triggers a different type of exception within the M4K core; the *MIPS32 M4K™ Processor Core Software User's Manual* [1] describes more details about these exceptions.

The initialization process for a M4K core requires a combination of hardware and software. This section describes the basic hardware initialization interface. In accordance with the MIPS32 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. See Reference [1] for a description of the software initialization requirements of a M4K core.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

7.2.1 *SI_ColdReset*

The high-active *SI_ColdReset* input is a hard reset signal that initializes the internal hardware state of the M4K core without saving any state information. This input is active-high and must be asserted for a minimum of 5 *SI_ClkIn* cycles. The falling edge triggers a reset exception that 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.

7.2.2 *SI_Reset*

The high-active *SI_Reset* input is a warm reset input to the M4K core. The input 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 M4K core does 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.

Within the core, *SI_ColdReset* and *SI_Reset* are handled almost identically. The only difference is that *SI_Reset* sets the $Status_{SR}$ field to identify a soft reset exception.

7.2.3 *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.

7.2.4 *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 neither *EJ_TCK* or *SI_ClkIn* 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.

7.3 Power Management

Two primary mechanisms exist for managing system power with a M4K 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.

7.3.1 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 M4K core is not in use, if desired by your system logic. The M4K 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 M4K core in a low-power state by reducing the clock frequency. When the RP bit in the Status

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

register is set by software, system logic can detect the assertion of the *SI_RP* output and choose to place the M4K 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 to 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 M4K core is not being used.

7.3.2 Software-Induced Sleep Mode

Upon execution of the software WAIT instruction, the M4K core will enter a low-power state once all outstanding bus activity has completed. Most of the clocks in the M4K core will be stopped, but a handful of flops will remain active to sense an external hardware event that will awaken the core again. The external events that 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 goes to the vast majority of flops within the M4K 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 M4K core could be placed into a low power state as well.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Design For Test Features

This chapter describes the Design For Test (DFT) features of the MIPS32™ M4K™ processor core. The MIPS-supplied DFT features are optional, so their existence on a particular core is dependent on choices made during implementation.

This chapter contains the following major sections:

- Section 8.1, "Introduction"
- Section 8.2, "Scan Test"
- Section 8.3, "User-Specific RAM BIST"

8.1 Introduction

An implementation of a M4K core may contain DFT features useful for supporting manufacturing test of the core within an SOC environment. Typically, the DFT features will include one or more of the following:

- Scan test
- Memory BIST using a user-specified method
- Other implementation-dependent features

Table 8-1 summarizes the key pin usage related to test modes present on the core. This table should be considered a typical usage only, and other documentation related to the implementation details of a specific core must be consulted. The column labeled “Integrated BIST” can be ignored, since it is not relevant for the M4K core.

Table 8-1 Core Input Values for Major Operating Modes

Input Pin	Mode			
	Normal (non-test)	**Scan**	**Integrated BIST**	**User-specified BIST**
SI_ClkIn	toggles	toggles	toggles	toggles
EJ_TCK	toggles when TAP active	toggles	-	-
SI_ColdReset	asserted for initialization	-	1	impl-dependent
gscanmode	0	1	0	0
gscanenable	0	1: chain operation 0: capture cycles	0	0
BistIn[n:0]	0	0	0	impl-dependent

The remaining sections in this chapter discuss the major test modes in more detail.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

8.2 Scan Test

The scan methodology normally used on a M4K core is muxed scan. The exact scan functionality is dependent on the choices made when the core was created. Specific details about scan operation are therefore implementation-dependent and beyond the scope of this document, but a few general comments are worth noting.

Three specific scan control pins besides the actual scan chain inputs and outputs are normally present. The scan control pins are: *gscanmode* and *gscanenable*. If the scan insertion scripts for Mentor DFTAdvisor, provided with a soft M4K core, have been used for the scan insertion, then the scan-chains inputs and outputs are normally called *gscanin_x* and *gscanout_x*, where x is an integer greater than or equal to 0 identifying the input and output of each separate scan chain.

With muxed scan, the two primary inputs clocks, *SI_ClkIn* and *EJ_TCK*, must be running when the scan chains are loaded and unloaded. During a capture cycle(s), one or both of the primary clocks may be active.

The typical use of the scan control pins is illustrated in Figure 8-1. Note that this figure denotes typical scan operation only, and may not be relevant for a specific core. *gscanmode* must be asserted during any scan operations. *gscanenable* is asserted when the scan chains are loaded and unloaded, but not during the capture cycles.

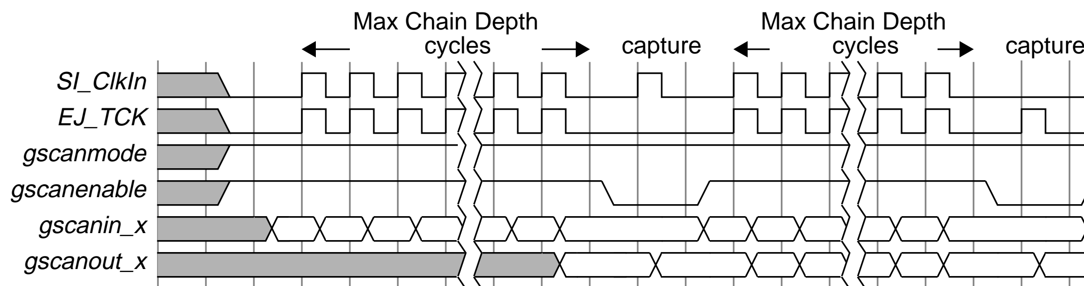

Figure 8-1 Timing Diagram of Typical Scan Chain and Capture Operation

8.3 User-Specific RAM BIST

User-specific RAM BIST utilizes the top-level *BistIn* and *BistOut* buses to test the on-chip trace SRAM array. The usage and meaning of these pins are implementation-dependent.

Depending on a specific implementation, some of the scan related pins and *SI_ColdReset* might have to be asserted to specific values during User-specified RAM BIST mode. It is normally required that the *BistIn* bus be tied to all zero's to enable normal functional mode and disable any User-specific RAM BIST.

If User-specific RAM BIST is not implemented, then simply tie the *BistIn* bus to all zero's and ignore the *BistOut* output bus.

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

References

This appendix lists other documents available from MIPS Technologies, Inc. that are referenced elsewhere in this document. These documents may be included in the `$MIPS_PROJECT/doc` area of a typical M4K soft or hard core release, or be available on the MIPS web site, under **http://www.mips.com/publications/index.html**.

1. MIPS32™ M4K™ Processor Core Software User's Manual
   MIPS document: MD00249
2. EJTAG™ Specification
   MIPS document: MD00047
3. EJTAG Trace Control Block Specification
   MIPS document: MD00148
4. Core Coprocessor Interface Specification
   MIPS document: MD00068
5. MIPS32™ Architecture For Programmers Volume III: The MIPS32™ Privileged Resource Architecture
   MIPS document: MD00090
6. MIPS64™ 5Kc™ Processor Core Software User's Manual
   MIPS document: MD00012
7. Core Coprocessor 2 Module Template Application Note
   MIPS document: MD00130

Copyright © 2002 MIPS Technologies Inc. All rights reserved.

Revision History

Table B-1 Revision History

Revision	Date	Description
00.90	June 27, 2002	• Preliminary release.
01.00	August 28, 2002	• Commercial release. • Added entry for selection of number of GPR shadow sets in VMC model, Table 6-2. • Modified abort description on SRAM interface, as abort requests are not only caused by interrupts. • Updated description of write buffer control signals on SRAM interface.
01.01	August 29, 2002	• Updated Simulation Models chapter with more recent VMC information. • Added more details to the write buffer description
01.02	August 30, 2002	• Corrected title of document. It was inadvertently called “Data Sheet”, not “Integrator’s Guide”, in the last two releases.
01.03	January 9, 2003	• Renamed SI_IAck pin • Added note about availability of cop2 template • Corrected use of “maximum” and “minimum” in description of TC_CRMax and TC_CRMin signals in Table 2-3 on page 4. • Updated Simulation Models chapter with more recent VMC information

Copyright © 2002 MIPS Technologies Inc. All rights reserved.