MIPS32™ 4KE™ Processor Core Family Integrator's Guide

Document Number: MD00104
Revision 02.00
November 8, 2002

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

Template: B1.07, Build with Conditional Tags: 2B EMERALD MIPS32 PROC

Table of Contents

List of Figures

List of Tables

Overview

This document is targeted for the ASIC designer who is integrating a version of a MIPS32™ 4KE™ 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 3 describes the pins of the core.
- Chapter 3, “EC Interface,” on page 19 describes the EC interface protocol used by the core.
- Chapter 4, “EJTAG Interface,” on page 35 discusses the EJTAG interface used by the core, including the optional EJTAG TAP controller and the PDtrace interface.
- Chapter 5, “Coprocessor Interface,” on page 43 describes the Coprocessor 2 interface and protocol used by the core.
- Chapter 6, “Scratchpad RAM Interface,” on page 59 describes the Scratchpad RAM interface that may optionally be present on the core.
- Chapter 7, “Performance Monitoring Interface,” on page 83 describes the Performance Monitor interface that may be used to count interesting events on the core.
- Chapter 8, “VMC Simulation Model,” on page 87 describes models that can be used in place of the 4KE 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 4KE 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 4KE core.
- Chapter 9, “Clocking, Reset and Power,” on page 97 covers issues related to handling the clock insertion delay of the 4KE core. Additionally, the hardware reset requirements of the core, as well as power management techniques, are discussed.
- Chapter 10, “Design For Test Features,” on page 101 discusses general DFT features which may be preset on the 4KE 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 4KE core deliverables. To set this variable:

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

Signal Description

This chapter describes the signals on a MIPS32™ 4KE™ 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 4KE core are prefixed with a unique string, according to their primary function. Table 2-2 defines the prefixes used for 4KE core interface signals.

Table 2-2 Signal Prefix Key

Prefix	Description
*EB_*	Signals directly related to the EC interface.
*SI_*	General system interface signals, which are not part of the EC interface.
*EJ_*	Signals related to the EJTAG interface.
*TC_*	Signals related to the EJTAG Trace interface.
*CP2_*	Signals related to the Coprocessor 2 interface.
*PM_*	Performance monitoring signals.
*{I,D}SP_*	Instruction/Data ScratchPad RAM interfaces

Table 2-2 Signal Prefix Key

Prefix	Description
gscan/Bist	Signals related to design-for-test features, either scan or memory Built-In-Self-Test (BIST).
gmb	Signals related to integrated memory 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.

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 9, "Clocking, Reset and Power," on page 97 for more details		
Clock Signals: Refer to Section 9.1, "Clocking" on page 97 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 9.2, "Reset and Hardware Initialization" on page 98 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 9.3, "Power Management" on page 99 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.

Table 2-3 Signal Descriptions (Continued)

Signal Name	Type	Description
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.
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.

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 4KE 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_MergeMode[1:0]</td><td>SIn</td><td>The state of these signals determines whether merging is allowed in the 16-byte collapsing write buffer. Value of SI_MergeMode[0] is visible to software in the $Config0_{MM}$ register field.
<table>
<tr><th>SI_MergeMode[1:0]</th><th>Merge Mode</th></tr>
<tr><td>00_2</td><td>No Merge</td></tr>
<tr><td>01_2</td><td>Reserved</td></tr>
<tr><td>10_2</td><td>Full Merge</td></tr>
<tr><td>11_2</td><td>Reserved</td></tr>
</table></td></tr>
<tr><td>SI_SimpleBE[1:0]</td><td>SIn</td><td>The state of these signals can constrain the core to only generate certain byte enables on EC interface transactions. This eases connection to some existing bus standards. Value of SI_SimpleBE[0] is visible in the $Config0_{SB}$ register field. See Section 3.6, "SimpleBE Mode" on page 32 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>
</table>

Table 2-3 Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td colspan="3">EC Interface Refer to Chapter 3, “EC Interface,” on page 19 for more details.</td></tr>
<tr><td>EB_ARdy</td><td>In</td><td>Indicates whether the target is ready for a new address. The core will not complete the address phase of a new bus transaction until the clock cycle after EB_ARdy is sampled asserted.</td></tr>
<tr><td>EB_AValid</td><td>Out</td><td>When asserted, indicates that the values on the address bus and access types lines are valid, signifying the beginning of a new bus transaction. EB_AValid is always be valid (unqualified).</td></tr>
<tr><td>EB_Instr</td><td>Out</td><td>When asserted, indicates that the transaction is an instruction fetch not a data reference. EB_Instr is only valid when EB_AValid is asserted.</td></tr>
<tr><td>EB_Write</td><td>Out</td><td>When asserted, indicates that the current transaction is a write. This signal is only valid when EB_AValid is asserted.</td></tr>
<tr><td>EB_Burst</td><td>Out</td><td>When asserted, indicates that the current transaction is part of a cache fill or a write burst. Note that there is redundant information contained in EB_Burst, EB_BFirst, EB_BLast, and EB_BLen[1:0]. This is done to simplify the system design - the information can be used in whatever form is easiest.</td></tr>
<tr><td>EB_BFirst</td><td>Out</td><td>When asserted, indicates beginning of the burst. EB_BFirst is always valid (unqualified).</td></tr>
<tr><td>EB_BLast</td><td>Out</td><td>When asserted, indicates end of burst. EB_BLast is always valid.</td></tr>
<tr><td>EB_BLen[1:0]</td><td>Out</td><td>Indicates length of the burst. This signal is only valid when EB_AValid is asserted.
<table>
<tr><th>EB_BLen[1:0]</th><th>Burst Length</th></tr>
<tr><td>00_2</td><td>reserved</td></tr>
<tr><td>01_2</td><td>4</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>EB_SBlock</td><td>SIn</td><td>When sampled asserted, sub block ordering is used. When sampled deasserted, sequential addressing is used. Value is visible to software in the BM field of the Config0 register.</td></tr>
<tr><td>EB_BE[3:0]</td><td>Out</td><td>Indicates which bytes of the EB_RData[31:0] or EB_WData[31:0] buses are involved in the current transaction. If an EB_BE[3:0] signal is asserted, the associated byte is being read or written. EB_BE[3:0] lines are only valid while EB_AValid is asserted.
<table>
<tr><th>EB_BE[3:0] Signal</th><th>Read Data Bits Sampled</th><th>Write Data Bits Driven Valid</th></tr>
<tr><td>EB_BE[0]</td><td>EB_RData[7:0]</td><td>EB_WData[7:0]</td></tr>
<tr><td>EB_BE[1]</td><td>EB_RData[15:8]</td><td>EB_WData[15:8]</td></tr>
<tr><td>EB_BE[2]</td><td>EB_RData[23:16]</td><td>EB_WData[23:16]</td></tr>
<tr><td>EB_BE[3]</td><td>EB_RData[31:24]</td><td>EB_WData[31:24]</td></tr>
</table></td></tr>
<tr><td>EB_A[35:2]</td><td>Out</td><td>Address lines for external bus. Only valid when EB_AValid is asserted. EB_A[35:32] are tied to 0000_2 in the 4KE cores.</td></tr>
<tr><td>EB_WData[31:0]</td><td>Out</td><td>Output data for writes.</td></tr>
<tr><td>EB_RData[31:0]</td><td>In</td><td>Input data for reads.</td></tr>
<tr><td>EB_RdVal</td><td>In</td><td>Indicates that the target is driving read data on EB_RData lines. EB_RdVal must always be valid. EB_RdVal may never be sampled asserted until the rising edge after the corresponding EB_ARdy was sampled asserted.</td></tr>
</table>

Table 2-3 Signal Descriptions (Continued)

Signal Name	Type	Description
EB_WDRdy	In	Indicates that the target of a write is ready. The *EB_WData* lines can change in the next clock cycle. *EB_WDRdy* may not be sampled until the rising edge where the corresponding *EB_ARdy* is sampled asserted.
EB_RBErr	In	Bus error indicator for read transactions. *EB_RBErr* is sampled on every rising clock edge until an active sampling of *EB_RdVal*. *EB_RBErr* sampled with asserted *EB_RdVal* indicates a bus error during read. *EB_RBErr* must be deasserted in idle phases.
EB_WBErr	In	Bus error indicator for write transactions. *EB_WBErr* is sampled at the rising clock edge following an active sample of *EB_WDRdy*. *EB_WBErr* must be deasserted in idle phases.
EB_EWBE	In	Indicates that any external write buffers are empty. The external write buffers must deassert *EB_EWBE* in the cycle after the corresponding *EB_WDRdy* is asserted and keep *EB_EWBE* deasserted until the external write buffers are empty. See Section 3.7, "External Write Buffers" on page 32 for more details.
EB_WWBE	Out	When asserted, indicates that the core is waiting for external write buffers to empty. See Section 3.7, "External Write Buffers" on page 32 for more details.
EJTAG Interface: Refer to Chapter 4, “EJTAG Interface,” on page 35 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 4KE 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		
EJ_DebugM	Out	Asserted when the core is in Debug Mode. This can be used to bring the core out of a low power mode (see Section 9.3, "Power Management" on page 99 for more details). In systems with multiple processor cores, this signal can be used to synchronize the cores when debugging.
Device ID Bits: These inputs provide an identifying number visible to the EJTAG probe. If the EJTAG TAP controller is not implemented, 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.		

Table 2-3 Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></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>
<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>
</table>

Table 2-3 Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></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>
<tr><td>TC_Calibrate</td><td>Out</td><td>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.</td></tr>
</table>

Table 2-3 Signal Descriptions (Continued)

Signal Name	Type	Description
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 43 for more details.		
Instruction Dispatch: These signals are used to transfer an instruction for the 4KE 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.
CP2_fs_0	Out	**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.
CP2_fbusy_0	In	**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.

Table 2-3 Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></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 4KE 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 4KE 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 4KE 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 4KE 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 4KE core from the COP2 coprocessor, as part of completing a From Coprocessor instruction.</td></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>
</table>

Table 2-3 Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></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 4KE 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 4KE 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>
<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>
</table>

Table 2-3 Signal Descriptions (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td colspan="3">Instruction Nullification: These signals are used by the 4KE 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 4KE core is signalling that the instruction is not nullified. When asserted, the 4KE 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 4KE 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">Performance Monitoring Interface: These signals can be used to implement performance counters which can be used to monitor hardware/software performance</td></tr>
<tr><td>PM_DCacheHit</td><td>Out</td><td>This signal is asserted whenever there is a data cache hit.</td></tr>
<tr><td>PM_DCacheMiss</td><td>Out</td><td>This signal is asserted whenever there is a data cache miss.</td></tr>
<tr><td>PM_DTLBHit</td><td>Out</td><td>This signal is asserted whenever there is a hit in the data TLB. This signal is valid only on the 4KEc™ core and should be ignored when using the 4KEp™ and 4KEm™ cores.</td></tr>
<tr><td>PM_DTLBMiss</td><td>Out</td><td>This signal is asserted whenever there is a miss in the data TLB. This signal is valid only on the 4KEc core and should be ignored when using the 4KEp and 4KEm cores.</td></tr>
<tr><td>PM_ICacheHit</td><td>Out</td><td>This signal is asserted whenever there is an instruction cache hit.</td></tr>
<tr><td>PM_ICacheMiss</td><td>Out</td><td>This signal is asserted whenever there is an instruction cache miss.</td></tr>
<tr><td>PM_InstComplete</td><td>Out</td><td>This signal is asserted each time an instruction completes in the pipeline.</td></tr>
</table>

Table 2-3 Signal Descriptions (Continued)

Signal Name	Type	Description
PM_ITLBHit	Out	This signal is asserted whenever there is an instruction TLB hit. This signal is valid only on the 4KEc core and should be ignored when using the 4KEp and 4KEm cores.
PM_ITLBMiss	Out	This signal is asserted whenever there is an instruction TLB miss. This signal is valid only on the 4KEc core and should be ignored when using the 4KEp and 4KEm cores.
PM_JTLBHit	Out	This signal is asserted whenever there is a joint TLB hit. This signal is valid only on the 4KEc core and should be ignored when using the 4KEp and 4KEm cores.
PM_JTLBMiss	Out	This signal is asserted whenever there is a joint TLB miss. This signal is valid only on the 4KEc core and should be ignored when using the 4KEp and 4KEm cores.
PM_WTBMerge	Out	This signal is asserted whenever there is a successful merge in the write through buffer.
PM_WTBNoMerge	Out	This signal is asserted whenever a non-merging store is written to the write through buffer.
ScratchPad RAM interface: This interface allows a ScratchPad RAM (SPRAM) array to be connected in parallel with the cache arrays, enabling fast access to data. There are independent interfaces for Instruction and Data ScratchPads. Note: In order to achieve single cycle access, the ScratchPad interface is not registered, unlike the other core interfaces. This requires more careful timing considerations. Refer to Chapter 6, “Scratchpad RAM Interface,” on page 59 for further details.		
DSP_TagAddr[19:4]	Out	Virtual index into the SPRAM used for tag reads and writes.
DSP_TagRdStr	Out	Tag Read Strobe - Hit, Stall, TagRdValue use this strobe.
DSP_TagWrStr	Out	Tag Write Strobe - If SPRAM tag is software configurable, this signal will indicate when to update the tag value.
DSP_TagCmpValue[23:0]	Out	Tag Compare Value - This bus is used for both tag comparison and tag write value. For tag comparison, the bus usage is {PA[31:10], 2’b0} and contains the address to determine hit/miss. For tag writes, the bus contains {PA[31:10], Lock, Valid} from the *TagLo* register.
DSP_DataAddr[19:2]	Out	Virtual index into the SPRAM used for data reads and writes.
DSP_DataWrValue[31:0]	Out	Data Write Value - Data value to be written to the data array.
DSP_DataRdStr	Out	Data Read Strobe - Indicates that the data array should be read.
DSP_DataWrStr	Out	Data Write Strobe - Indicates that the data array should be written.
DSP_DataWrMask[3:0]	Out	Data Write Mask - Byte enables for a data write.
DSP_DataRdValue[31:0]	In	Data Read Value - Data value read from the data array.
DSP_TagRdValue[23:0]	In	Tag Read Value - Tag value read from the tag array. Written to *TagLo* register on a CACHE instruction. Read value maps into these *TagLo* fields: {PA[31:10], Lock, Valid}
DSP_Hit	In	Hit - Indicates that this read was to an address covered by the SPRAM.
DSP_Stall	In	Stall - Indicates that the read has not yet completed.
DSP_Present	SIn	Present - Indicates that a SPRAM array is connected to this port.
ISP_Addr[19:2]	Out	Virtual index into the SPRAM used for both reads and writes of tag and data.

Table 2-3 Signal Descriptions (Continued)

Signal Name	Type	Description
ISP_RdStr	Out	Read Strobe - indicates a read of the tag and data arrays. Hit and Stall signals are also based off of this strobe.
ISP_TagWrStr	Out	Tag Write Strobe - If SPRAM tag is software configurable, this signal will indicate when to update the tag value.
ISP_DataTagValue[31:0]	Out	Write/Compare Data For data writes, this is the value to be written to the data array. For tag writes the bus contains the {8'b0, PA[31:10], Lock, Valid} from the *TagLo* register. For tag comparison, the bus has the address to be used for hit/miss determination in the format {8'b0, PA[31:10], Uncacheable, 1'b0}. When high, the Uncacheable bit indicates that the physical address bits (PA[31:10]) are to an uncacheable address; when the Uncacheable bit is low, the physical address is to a cacheable address.
ISP_DataWrStr	Out	Data Write Strobe - Indicates that the data array should be written.
ISP_DataRdValue[31:0]	In	Data Read Value - Data value read from the data array.
ISP_TagRdValue[23:0]	In	Tag Read Value - Tag value read from the tag array. Written to *TagLo* register on a CACHE instruction. Read value maps into these *TagLo* fields: {PA[31:10], Lock, Valid}
ISP_Hit	In	Hit - Indicates that this read was to an address covered by the SPRAM.
ISP_Stall	In	Stall - Indicates that the read has not yet completed.
ISP_Present	SIn	Present - Indicates that a SPRAM array is connected to this port.
Scan Test Interface: These signals provide the interface for testing the core. The use and configuration of these pins are implementation-dependent.		
gscanenable	In	This signal should be asserted while scanning vectors into or out of the core. The *gscanenable* signal must be deasserted during normal operation and during capture clocks in test mode.
gscanmode	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.
gscanramwr	In	This signal will optionally provide direct control over the read and write strobes of the RAM arrays in the core. This control will only occur if *gscanmode* is also asserted, and if this feature was selected when the core was built. *gscanramwr* is recommended to be held low during normal (non-scan) 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
Integrated Memory BIST Interface: These signals provide an interface to integrated memory BIST features present within the core for testing the internal cache SRAM arrays. Refer to Chapter 10, "Design For Test Features," on page 101 for more details about the use of this interface.		
gmbinvoke	In	Enable signal for integrated BIST controllers.
gmbdone	Out	Common completion indicator for all integrated BIST sequences.

Table 2-3 Signal Descriptions (Continued)

Signal Name	Type	Description
gmbddfail	Out	When high, indicates that the integrated BIST test failed on the data cache data array.
gmbtdfail	Out	When high, indicates that the integrated BIST test failed on the data cache tag array.
gmbwdfail	Out	When high, indicates that the integrated BIST test failed on the data cache way select array.
gmbdifail	Out	When high, indicates that the integrated BIST test failed on the instruction cache data array.
gmbtifail	Out	When high, indicates that the integrated BIST test failed on the instruction cache tag array.
gmbwifail	Out	When high, indicates that the integrated BIST test failed on the instruction cache way select array.

EC Interface

This chapter describes the EC interface, which is present on all MIPS32 4KE processor cores. The EC interface is generally described in the companion document, titled *EC Interface Specification* [2]. The rest of this chapter discusses the specific 4KE implementation of the EC interface.

This chapter contains the following major sections:

3.1 Interface Transactions

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

3.1.1 Fastest Read Transaction

The core allows data to be returned in the same clock that the address is driven onto the bus. This is the fastest type of read cycle as shown in Figure 3-1 on page 20.

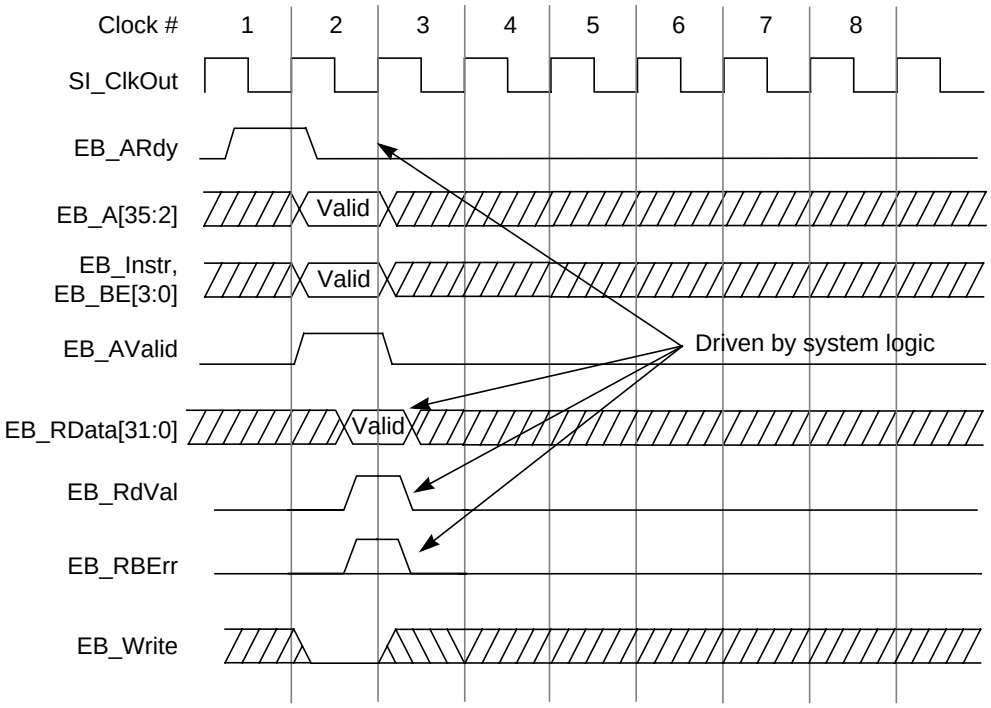


Figure 3-1 Fastest Read Cycle

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

3.1.2 Single Read with Wait States

Figure 3-2 on page 21 shows a single read transaction with one address wait state and one data wait state. The core drives address onto *EB_A[35:2]* and byte enable information onto *EB_BE[3:0]*. To maximize performance, the bus interface does not define a maximum number of outstanding bus cycles. Instead the interface provides the *EB_ARdy* input signal; this signal is driven by external logic and controls the generation of addresses on the bus. Current versions of the 4KE cores can only have a maximum of 8 reads and 8 writes outstanding, but future versions may have a larger number of outstanding transactions.

The core drives an address whenever it becomes available, regardless of the state of *EB_ARdy*. However, the core always continues to drive the address until the clock after *EB_ARdy* is sampled asserted. For example, on the rising edge of clock 2 in Figure 3-2 on page 21, the *EB_ARdy* signal is sampled low, indicating that external logic is not ready to accept the new address; however, the core still drives *EB_A[35:2]* in this clock as shown. On the rising edge of clock 3 the core samples *EB_ARdy* asserted and continues to drive the address until the rising edge of clock 4.

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

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

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

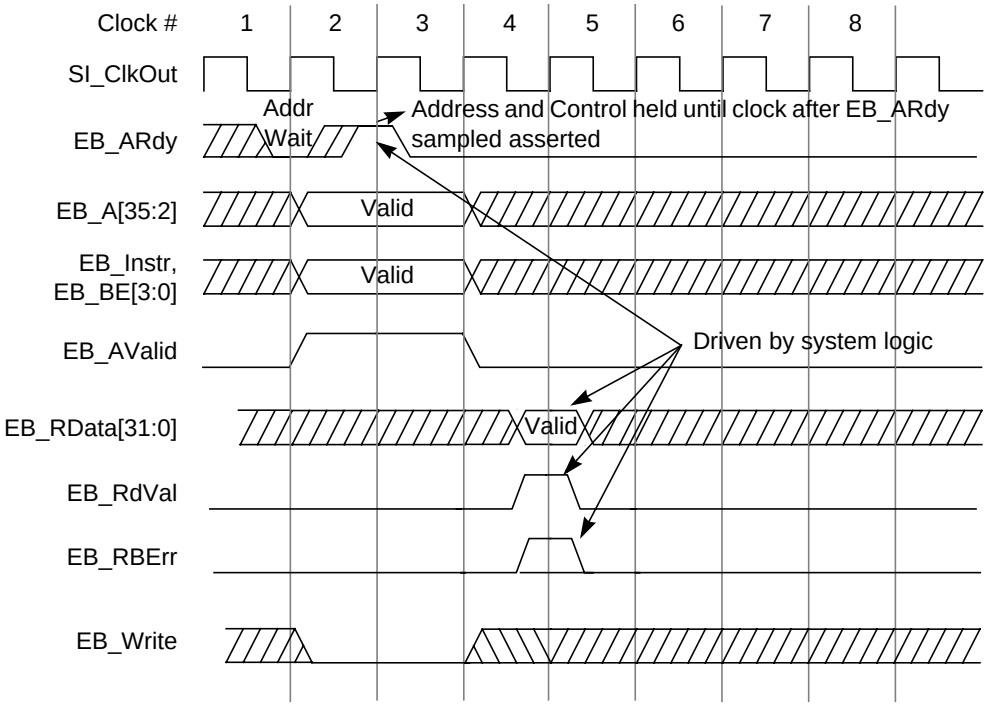


Figure 3-2 Single Read Transaction with Wait States

3.1.3 Fastest Write Transaction

The core allows the *EB_WDRdy* signal to be driven active in the same clock that address and data are driven onto the bus. This is the fastest type of write cycle as shown in Figure 3-3 on page 22.

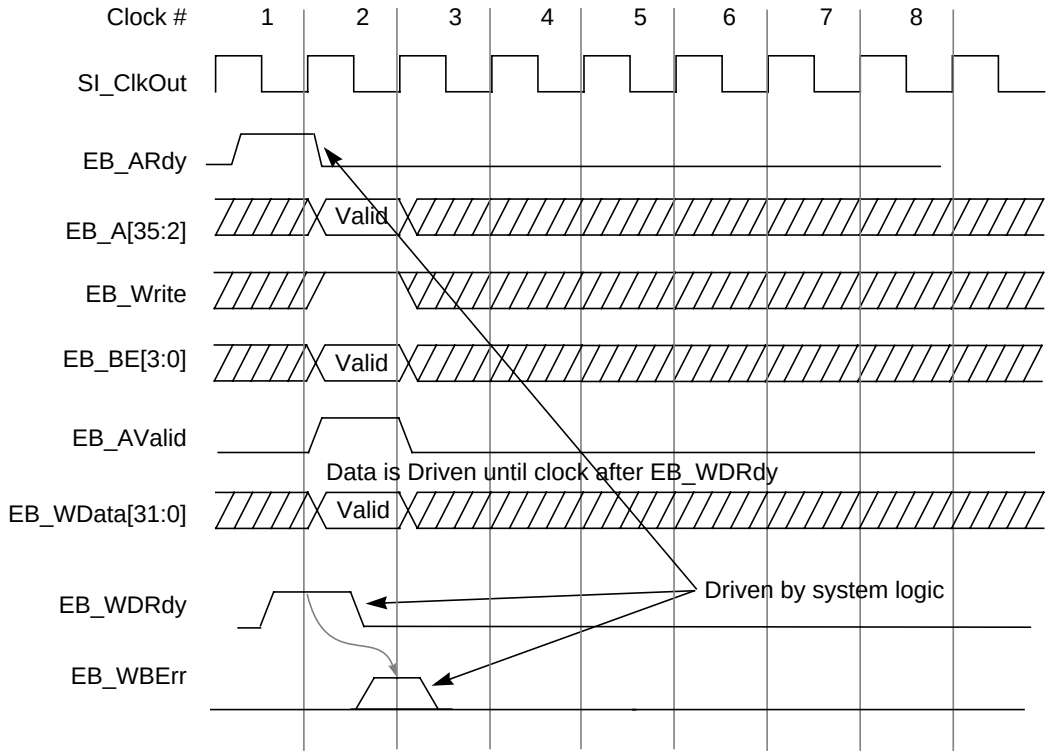


Figure 3-3 Fastest Write Transaction

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

3.1.4 Single Write with Wait States

Figure 3-4 on page 23 shows a write transaction with one address wait state and two data wait states. The core drives address and control information onto the *EB_A[35:2]* and *EB_BE[3:0]* signals on the rising edge of clock 2. As in the single read cycle with wait states, these signals remain active until the clock edge after the *EB_ARdy* signal is sampled asserted. The core asserts the *EB_Write* signal to indicate that a valid write cycle is on the bus, and asserts *EB_AValid* to indicate that a valid address is on the bus.

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

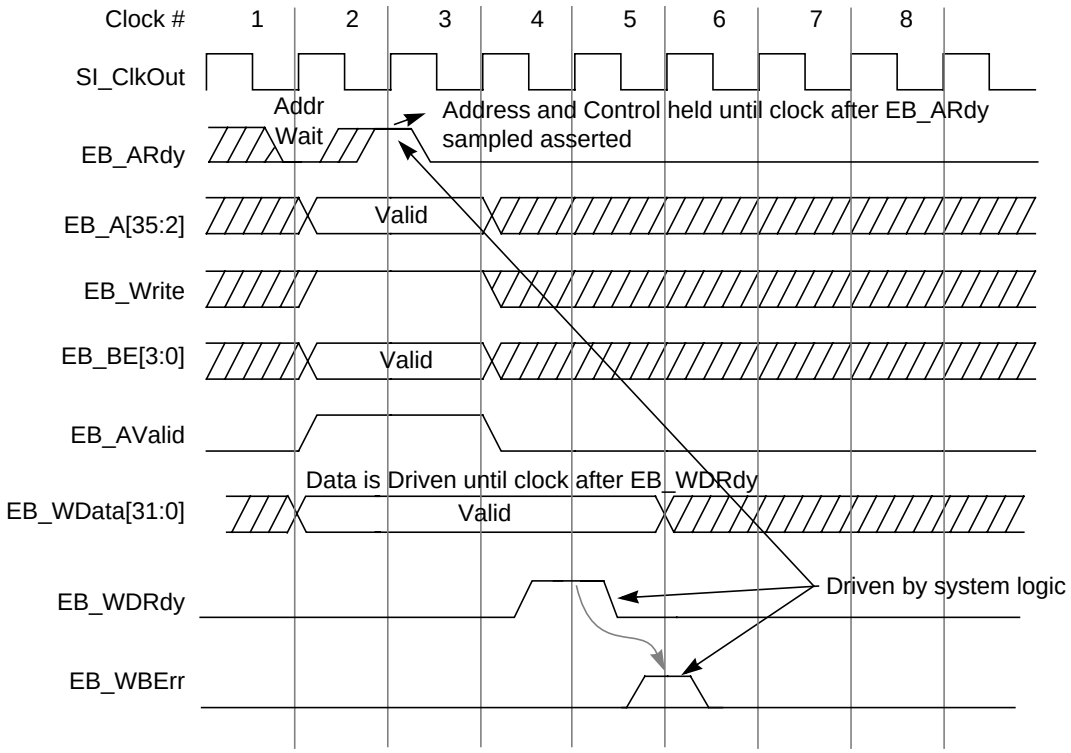


Figure 3-4 Single Write Transaction with Wait States

3.1.5 Burst Read

The core is capable of generating burst transactions on the bus. A burst transaction is used to transfer multiple related data items. Redundant signals are provided so that system logic can treat a burst transaction either as one transaction or as multiple independent transactions.

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

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

Table 3-1 Sequential Burst Order

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

Table 3-2 SubBlock Burst Order

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

Figure 3-5 on page 25 shows an example of a burst read transaction. The core drives address and control information onto the *EB_A[35:2]* and *EB_BE[3:0]* signals on the rising edge of clock 2. As in the single read cycle, these signals remain active until the clock edge after the *EB_ARdy* signal is sampled asserted. The core continues to drive *EB_AValid* as long as a valid address is on the bus.

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

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

Note that on the rising edge of clock 6 in Figure 3-5 on page 25 the *EB_RdVal* signal is sampled deasserted, causing a wait state between *Data 2* and *Data 3*. External logic asserts the *EB_RBErr* signal in the same clock as data if a bus error occurs during that data transfer.

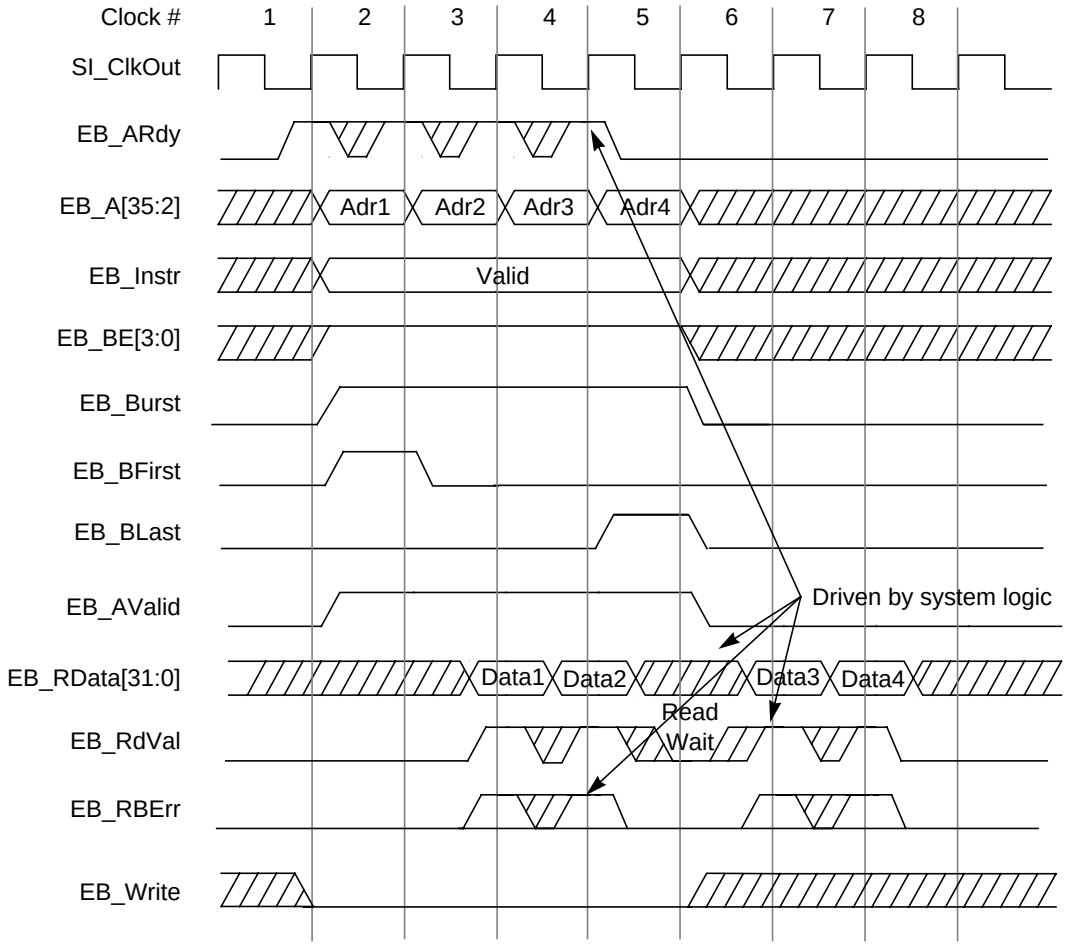


Figure 3-5 Burst Read Transaction Timing Diagram

3.1.6 Burst Write

Burst write transactions are used to empty one of the write buffers. A burst transaction is only performed if the write buffer contains 16 bytes of data associated with the same aligned memory block; otherwise individual write transactions are performed. Figure 3-6 on page 26 shows a timing diagram of a burst write transaction. Unlike the read burst, a write burst always begins with EB_A[3:2] equal to 00b.

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

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

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

In Figure 3-6 on page 26 the first data word (*Data1*) is driven in clocks 2 and 3 due to the *EB_WDRdy* signal being sampled deasserted on the rising edge of clock 2, causing one wait state cycle. When *EB_WDRdy* is sampled asserted on the rising edge of clock 3, the core responds by driving the second word (*Data2*) on the rising edge of clock 4.

External logic drives the EB_WBErr signal one clock after the corresponding assertion of EB_WDRdy if a bus error has occurred as shown by the arrows in Figure 3-6 on page 26.

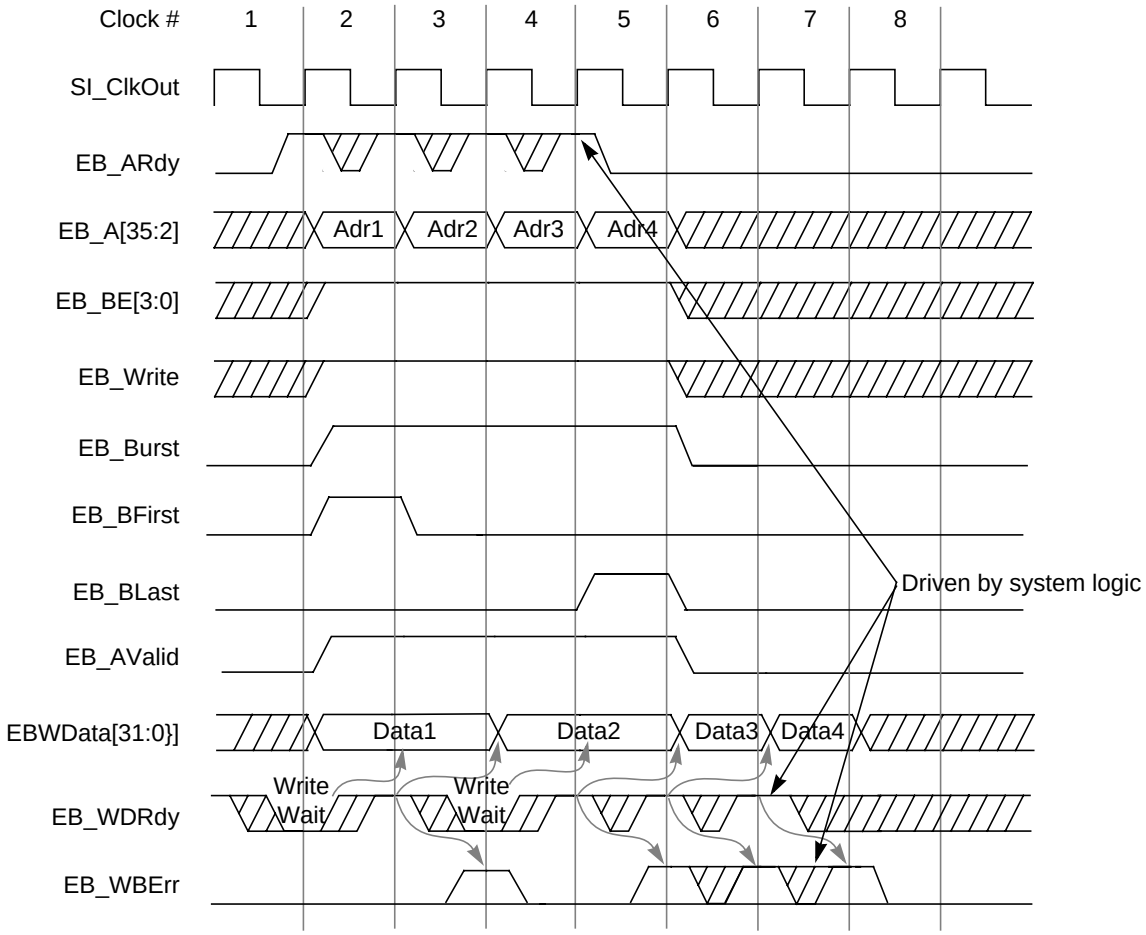


Figure 3-6 Burst Write Transaction Timing Diagram

3.1.7 Back-to-Back Reads

Figure 3-7 on page 27 shows the basic timing relationships of signals during a back-to-back read transaction. During a back-to-back read cycle, the core drives addresses for both read cycles onto *EB_A[35:2]* and byte enable information onto *EB_BE[3:0]*. Note that unlike the 4K cores, the 4KE cores do not necessarily leave a dead clock between address transactions.

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

An address is driven by the core whenever it becomes available, regardless of the state of *EB_ARdy*; however, the core always continues to drive the address until the clock after *EB_ARdy* is sampled asserted. For example, on the rising edge of clock 2 in Figure 3-7 on page 27 the *EB_ARdy* signal is sampled low, indicating that external logic is not ready to accept the new address. However, the core still drives *EB_A[35:2]* in this clock as shown. on the rising edge of clock 3, the core samples *EB_ARdy* asserted and continues to drive the address until the rising edge of clock 4.

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

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

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

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

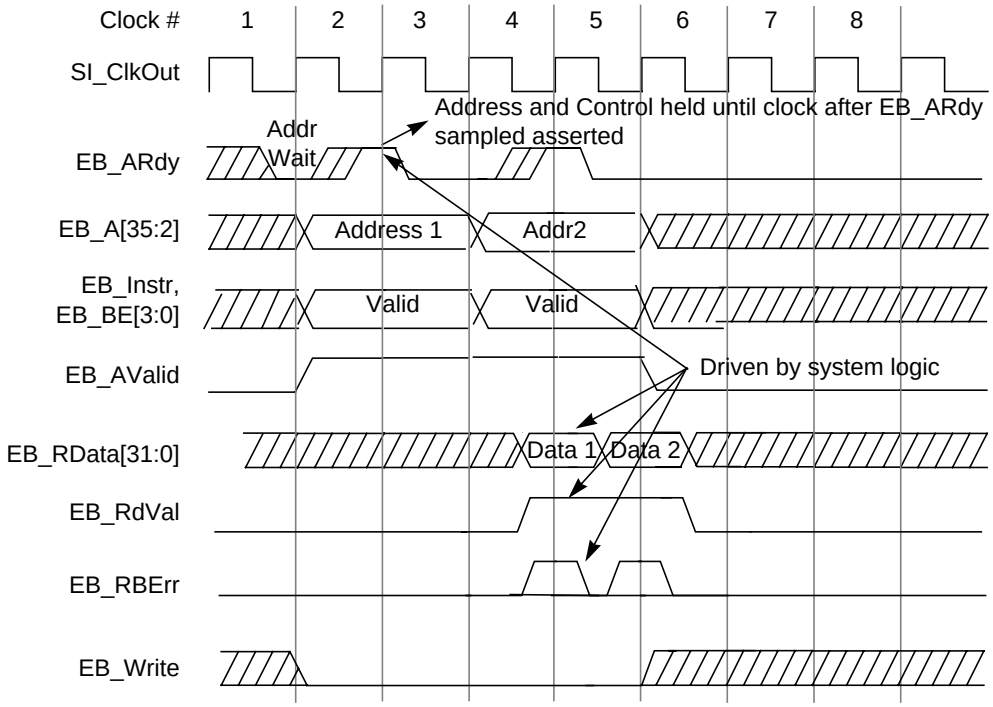


Figure 3-7 Back-to-Back Read Transaction Timing Diagram

3.1.8 Back-to-Back Writes

Figure 3-8 on page 28 shows a timing diagram of a back-to-back write operation. In any bus transaction, the core drives address, control, and data information as they becomes available, regardless of the state of *EB_ARdy*. If the *EB_ARdy* signal is asserted at the time that the address is driven by the processor, indicating that the external agent can accept another address, then the processor can drive a new address on the following clock.

In Figure 3-8 on page 28, address, control, and data (*Write1/Data1*) become available and are driven onto the bus by the core during clock 2. *EB_ARdy* is sampled deasserted on the rising edge of clock 2, indicating that the external agent is not ready to accept a new address. This causes one address wait state for the *Write1* address. The processor continues to drive the bus with the *Write1* address and control until the clock after *EB_ARdy* is sampled asserted. In this case, *EB_ARdy* is sampled asserted on the rising edge of clock 3, allowing the processor to drive new address and control on the rising edge of clock 4.

The *EB_WDRdy* signal is driven by the external agent to indicate that it has accepted the data on the bus. In this example, the *EB_WDRdy* signal is sampled deasserted by the core on the rising edge of clock 3, causing the core to hold the data (*Data1*) during the following cycle, clock 4. The external agent asserts *EB_WDRdy* during clock 3, which is sampled by the core on the rising edge of clock 4, indicating that is has accepted the data on the bus. The core continues to drive data until one clock after *EB_WDRdy* is sampled asserted, so the core drives *Data1* until the rising edge of clock 5. Note that

EB_WDRdy will never be sampled earlier than the rising edge in which the associated *EB_ARdy* is sampled asserted. If *EB_WDRdy* was asserted on the rising edge of clock 2 (one cycle before the *EB_ARdy*), then it would have been ignored.

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

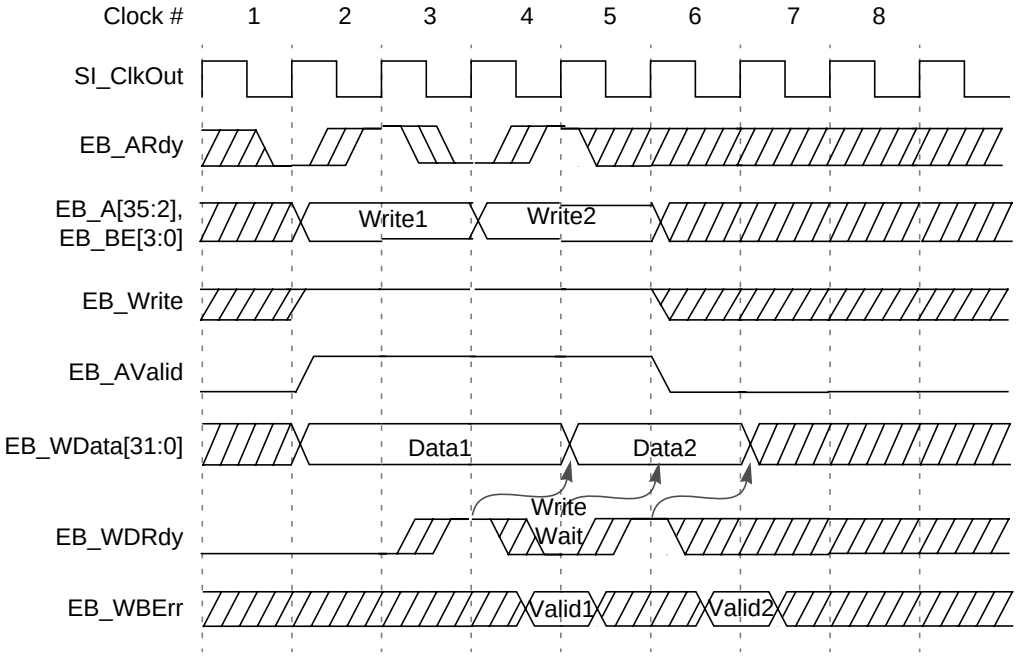


Figure 3-8 Back-to-Back Write Transactions

3.1.9 Read Followed by Write with Reordering

Figure 3-9 on page 29 shows a timing diagram of a read followed by a write operation with the operations being completed out of order. Since data is transferred for read and write operations on independent unidirectional buses (and their corresponding ready indicators), the bus interface allows read and write operations to complete out of order with respect to how the read and write addresses were initiated.

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

Address and control for the read operation become available and are driven onto the bus by the core on the rising edge of clock 2. The external agent has *EB_ARdy* asserted so there are no address wait states for the read. The processor continues to drive the bus until one clock after *EB_ARdy* is sampled asserted. The write address and control information are driven on the rising edge of clock 4(1 clock after it could have been sent). The external agent asserts *EB_ARdy* for an additional clock, which is sampled by the core on the rising edge of clock 4, so the core could have driven a new address (not shown) on the rising edge of clock 5.

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

Read data is driven onto the bus during clock 5, one clock after the write operation has completed. The core samples the *EB_RdVal* signal asserted on the rising edge of clock 6, causing the processor to latch the data and terminate the read

cycle. Note that it is the responsibility of the external agent to ensure the correct data is returned when re-ordering data transactions.

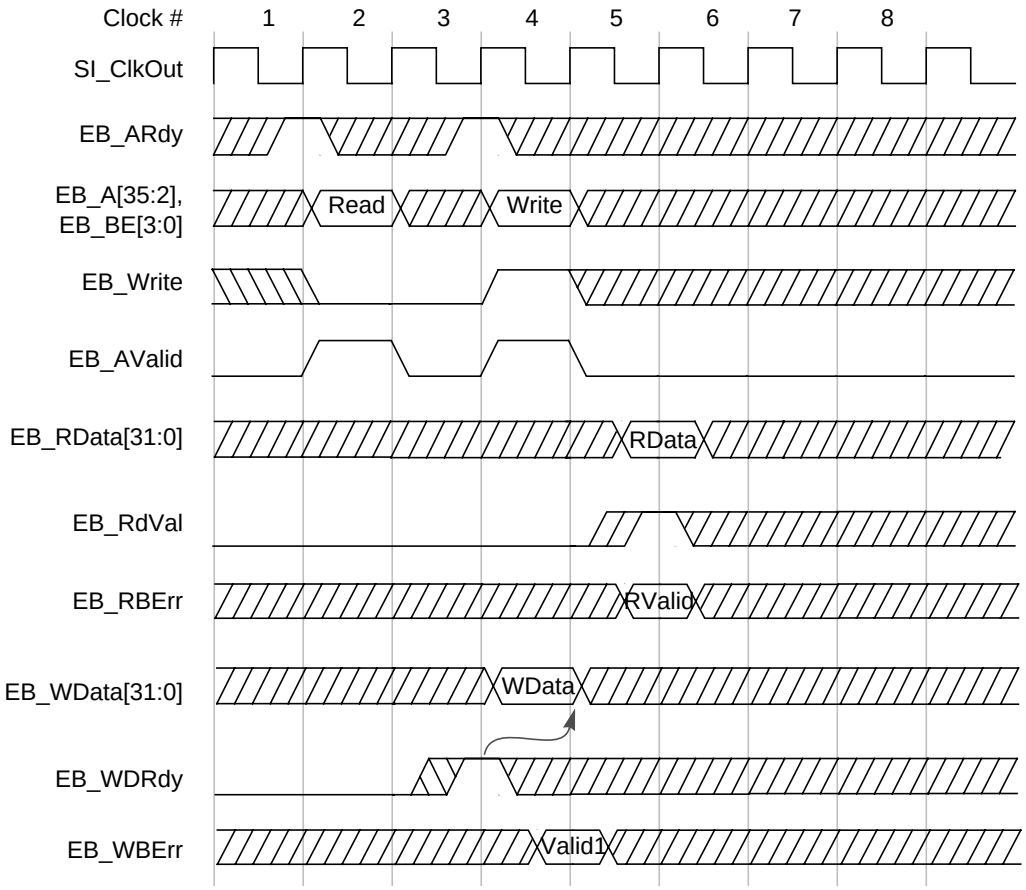


Figure 3-9 Read Followed by Write Transaction with Reordering

3.1.10 Write Followed by Read with Reordering

Figure 3-10 on page 30 shows a timing diagram of a write followed by a read operation with the operations being completed out of order.

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

Address, control, and data for the write operation become available and are driven by the core on the rising edge of clock 2. The processor continues to drive the address and control buses until *EB_ARdy* is sampled asserted. Since *EB_ARdy* was not sampled asserted on the rising edge of clock 2, an address wait state results. The assertion of *EB_ARdy* is sampled on the rising edge of clock 3, causing the processor to drive the write address and control until the rising edge of clock 4. The read address and control are then initiated on the bus on the rising edge of clock 5 (the read could have started in clock 4). Note that a new address (not shown) could have been driven on the bus on the rising edge of clock 6.

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

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

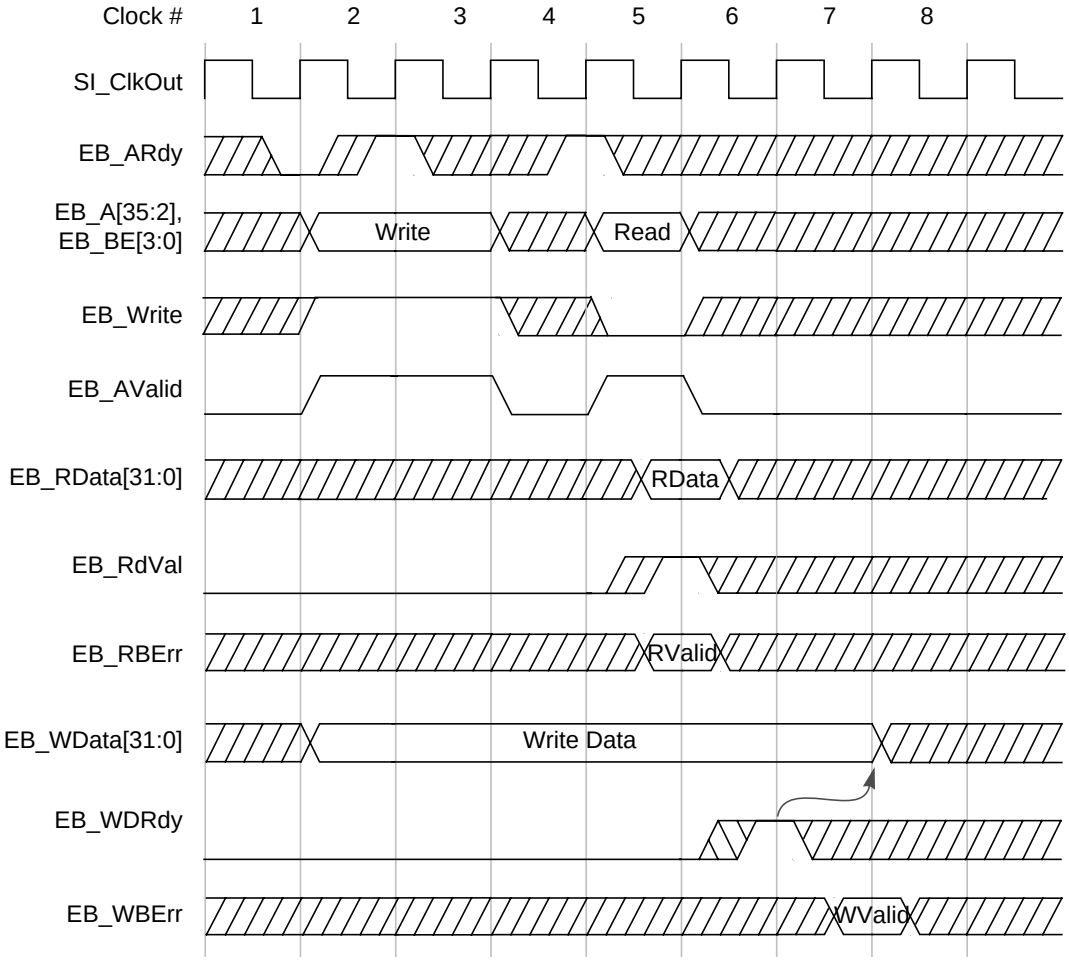


Figure 3-10 Write Followed by Read Transaction with Reordering

3.2 Outstanding Transactions

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

The bus interface implementation of the 4KE core contains enough buffering to allow a maximum of 16 transactions to be active simultaneously, as follows:

- Four bursted instruction reads
- Four bursted data reads
- Eight writes to a single 16B line

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

3.3 Sequential Transactions

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

Unlike the 4K processor cores, the 4KE cores will not add a dead clock between address transactions. For example, an instruction read address can come onto the bus immediately after a data read address.

3.4 Write Buffer

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

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

- When a store is attempted from the core to a different 16-byte block than is currently being accumulated.
- **SYNC** instruction. The **CACHE** instruction also performs an implicit **SYNC**.
- Store to a valid word in the buffer if merging is disabled.
- Any store to uncached memory.
- A load to the line being merged.
- A complete 16B line has been gathered for a burst write and the bus is idle.

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

3.5 Merging Control

All 4KE cores implement two 16 byte collapsing write buffers that allow byte, halfword, tri-byte, or word writes from the core to be accumulated in the buffer into a 16 byte value before bursting the data out onto the bus in word format. This buffer also gathers dirty cache lines during an eviction. Note that writes to uncached areas are never merged.

Merging can be disabled. If merging is disabled, the buffer will still attempt to gather an entire 16B line to generate a bursted write. If a store is attempted to a word that is already valid in the write buffer, the buffer will be flushed and the two stores will not merge.

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

3.6 SimpleBE Mode

The merging write buffer and even individual load and store instructions can generate bus 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 EC 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 3-3.

Table 3-3 Allowable Byte Enables in SimpleBE Mode

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

The only load instruction that attempts to generate a complex byte enable combination is an uncached LWL/LWR instruction requesting a tri-byte from memory. In SimpleBE mode, this transaction will be turned into a word request on the bus. When the full word is returned, the core will only use the appropriate 3 bytes. In normal mode, load operations to uncached space are always for the exact bytes requested. In SimpleBE mode, however, uncached tri-byte loads are turned into a full word request, the memory system must be capable of tolerating an uncached request to the fourth byte which won't actually be used by the core.

Merging stores or SWL/SWR instructions can also attempt to generate complex byte enable combinations. When a write transaction with complex byte enables is detected internally, the core will split the write into two transactions on the bus. Each transaction meets one of the byte enable combinations shown in Table 3-3: one with the upper two byte enables deasserted and one with the lower two deasserted.

The setting of *SI_SimpleBE* is independent of the value for *SI_MergeMode*. For example, the full merging option could be chosen for *SI_MergeMode*, while SimpleBE mode is selected for *SI_SimpleBE*.

3.7 External Write Buffers

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

In most systems, the *EB_EWBE signal* is tied high. Just using the *EB_WWBE* signals does not ensure coherency. If a write is in the external write buffer, then the core can generate a read request to the given address without asserting

EB_WWBE (because the core has no knowledge of the external write buffers). Therefore, any write buffers in the system must maintain coherency with reads.

The *EB_WWBE/EB_EWBE* interface can be used to make **SYNC**s "harder" by forcing the flush of the external write buffers. This is a system/SW design issue - a decision must be made in determining what the system does when a **SYNC** instruction is executed (and the same will be done for other synchronizing instructions such as **CACHE**).

In order to minimize the delay when the write buffers are already empty (or the *EB_EWBE* signal is not used and just tied high), *EB_EWBE* can be sampled before *EB_WWBE* is seen externally. This will only happen if the internal write buffers are empty and no writes are on the EC interface. If there are writes already on the bus or new writes caused by the **SYNC**, *EB_EWBE* will not be sampled until the cycle that the final write is accepted.

EJTAG Interface

This chapter discusses chip-level integration details for the EJTAG-related signals on a MIPS32™ 4KE™ 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 4KE cores within a single chip are discussed.

This chapter contains the following sections:

An EJTAG TAP controller is an optional feature in a 4KE core. If the 4KE core under use does not contain the EJTAG TAP controller, then much of this chapter is irrelevant.

Reference to the general *EJTAG Specification* [3] 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™ 4KE™ Processor Core Family 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.

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 4KE core, however. The 4KE 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 36 shows the serial *TDI-TDO* chain setup with parallel control of the TAP controllers.

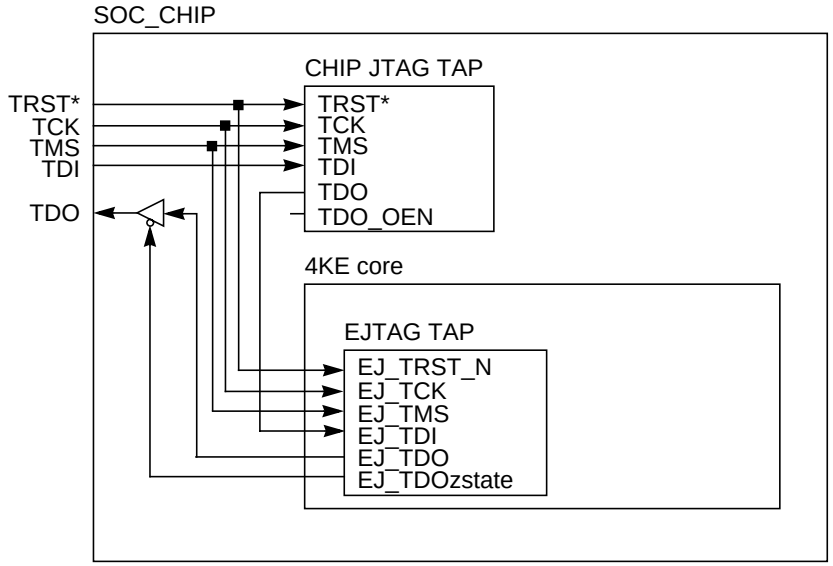


Figure 4-1 Daisy-Chained *TDI-TDO* Between JTAG and EJTAG TAP Controllers

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.

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 36 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 37, a setup which anticipates the existence of *TRST** on the "CHIP JTAG TAP" controller is shown.

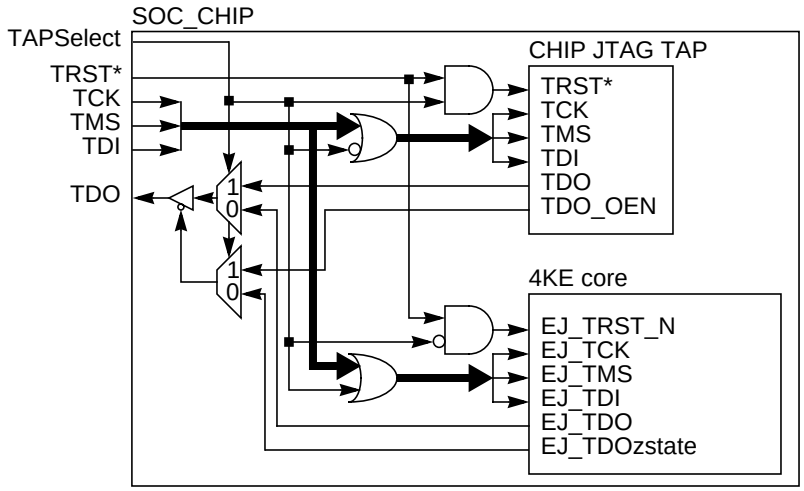


Figure 4-2 Multiplexing Between JTAG and EJTAG TAP Controllers

TAPSelect in Figure 4-2 on page 37 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 4KE core (and thus probably the entire SOC_CHIP) is held in reset. During reset, EJTAG commands can initialize the 4KE core to leave the reset state in Debug Mode, and thus the debug interface can control the 4KE 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 37.

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 4KE core will require separate pins on the chip. This section will discuss how to connect all the *EJ_** pins in the chip.

4.2.1 EJTAG Chip-Level Pins

The EJTAG TAP signals on the 4KE 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 38 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.

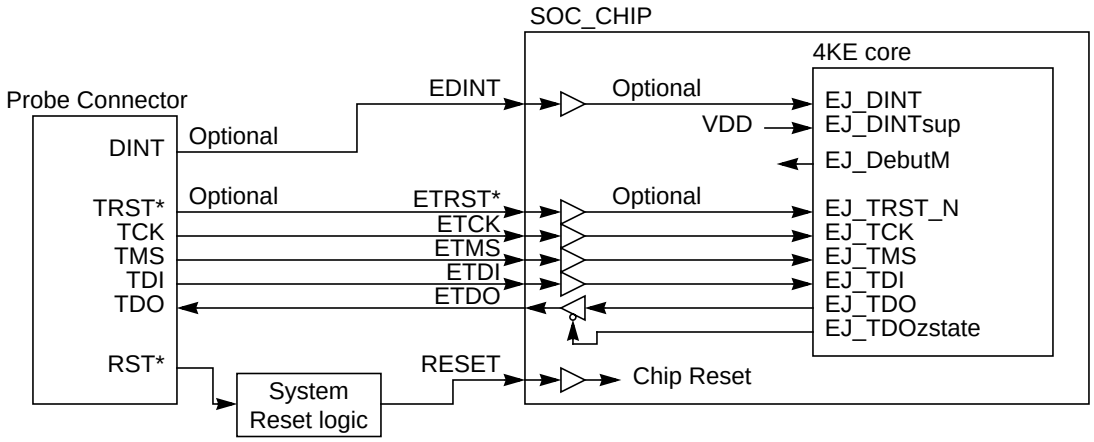


Figure 4-3 EJTAG Chip-Level Pin Connection

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 38, 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 40 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 4KE 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 4KE core triggers a Debug Interrupt Exception. This will stop the normal program flow within the 4KE 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 4KE core may also be connected to on-chip logic, such as a Multi-Core Breakpoint Unit (see Figure 4-5 on page 41 for more details). The *EJ_DINTsup* (EJTAG Debug Interrupt Pin Supported) input on a 4KE 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.

If *EJ_DINT* on the 4KE 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 39".

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 4KE 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 4KE 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 40 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 4KE core. The effect of an asserted *EJ_PerRst* is implementation-dependent; however, it should never result in a reset of the 4KE core itself. Figure 4-4 on page 40 shows one possible implementation of the use of *EJ_PerRst*.

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 40 for a possible implementation.

4.2.3.4 A Reset Logic Implementation

Figure 4-4 on page 40 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 4KE core. This requires 3 extra output signals, but this need not be the case.

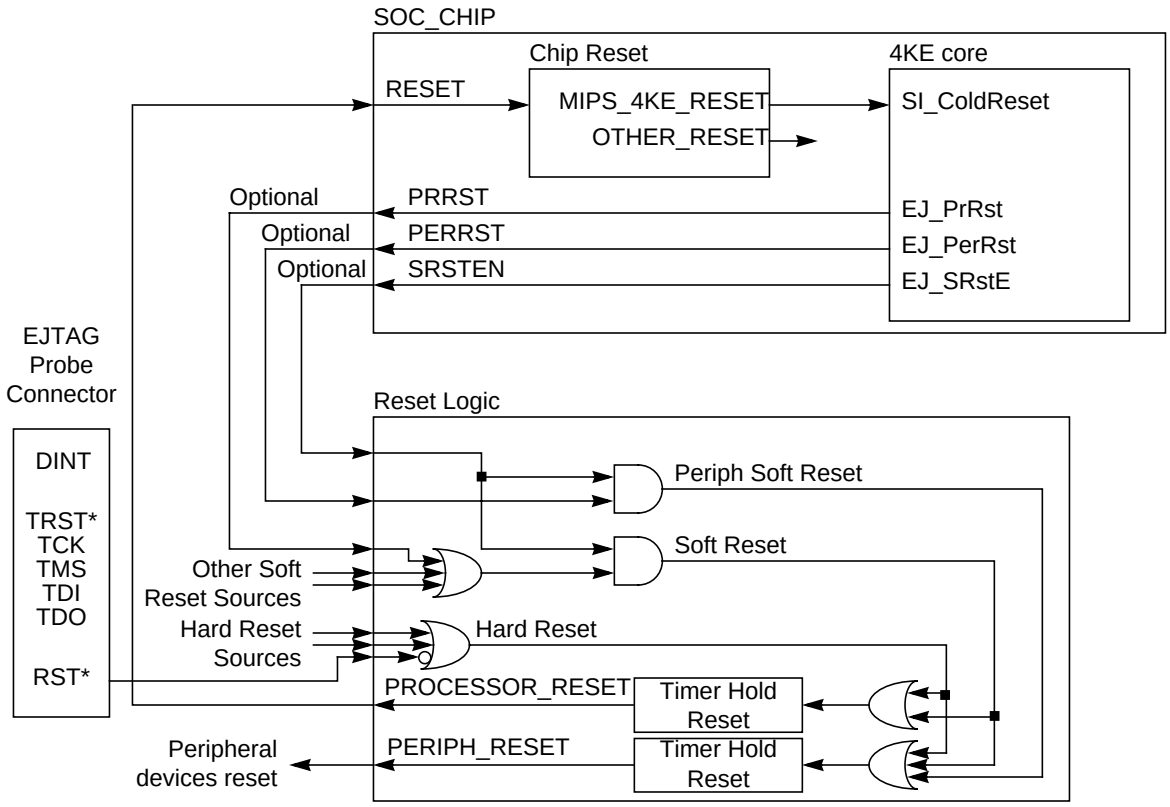


Figure 4-4 Reset Circuitry Implementation

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 4KE 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 41.

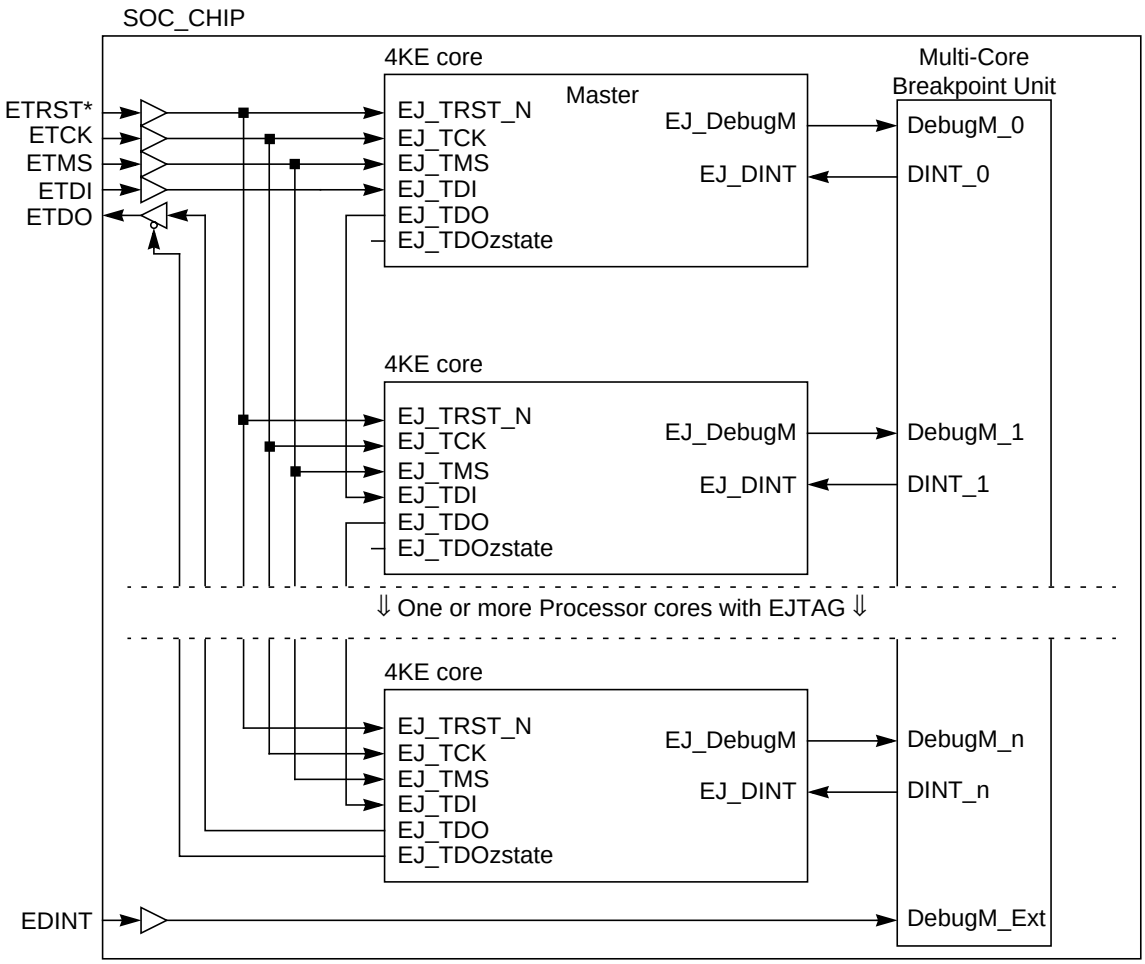


Figure 4-5 Multi-Core Implementation

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 41, 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 41 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).

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 4KE core ignores any subsequent Debug Interrupts on *EJ_DINT.*

4.4 EJTAG Trace

A 4KE 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 4KE 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* [4].

Coprocessor Interface

This chapter describes the MIPS Core Coprocessor Interface supported by the MIPS32™ 4KE™ processor core. The MIPS Core Coprocessor Interface is described in the companion document, titled *Core Coprocessor Interface Specification* [5]. The Core Coprocessor Interface is an optional feature in a 4KE core. If the 4KE core does not contain the Core Coprocessor Interface logic, then this chapter is irrelevant. This chapter discusses the specific 4KE 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"

5.1 Introduction

The 4KE 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 4KE 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 4KE 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 4KE processor core family.
- Fully compliant to the MIPS Core Coprocessor Interface standard.

5.2 Coprocessor Instructions

A 4KE 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.

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 4KE core (see Section 5.4.5, "Coprocessor Exceptions").

The 4KE core only dispatches instructions to the coprocessor if the CU2 bit in the CP0 *Status* register is set. Refer to the *MIPS32 4KE Processor Core Family Software User's Manual* for details on Coprocessor 2 instructions and CP0 registers.

5.3 Signal Configuration

The 4KE 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 4KE 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 4 for a listing of all the 4KE core signals).

The *CP2_inst32_0* output is tied high (logic 1). The 4KE 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 4KE core never sends data out of order. The coprocessor attached to a 4KE 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 4KE 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 4KE 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 4KE 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.

5.4 Interface Protocols

Refer to Table 2-3 on page 4 for a complete listing of all the pins of the 4KE 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 4KE core.
- **Coprocessor Exceptions** - Notifies the 4KE 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 4KE 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

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[b])	• Instruction nullification[a] • Condition code check results • Coprocessor exceptions or not • Instruction killing or not
Note: [a] The 4KE 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.

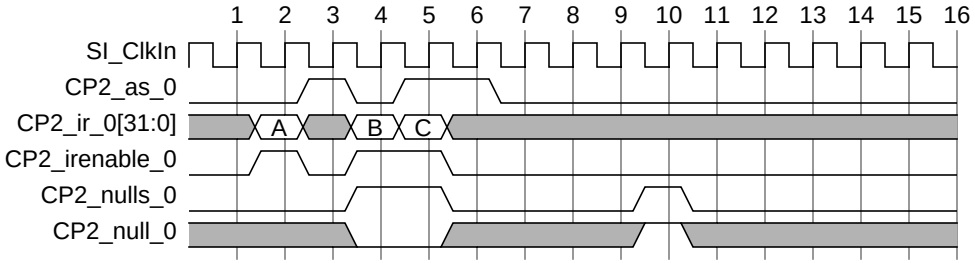


Figure 5-1 General Transfer Example

Figure 5-1 on page 47 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 4KE core, also the nullify strobe is always send in the cycle after dispatch on the 4KE 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.

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 4KE 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 4KE Core

From	To	Stage Needed	Min (cycles)	Max (cycles)	4KE 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 4KE 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 4KE 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 4KE 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.

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 4KE 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 4KE 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 4KE 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 4KE 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 4KE 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.

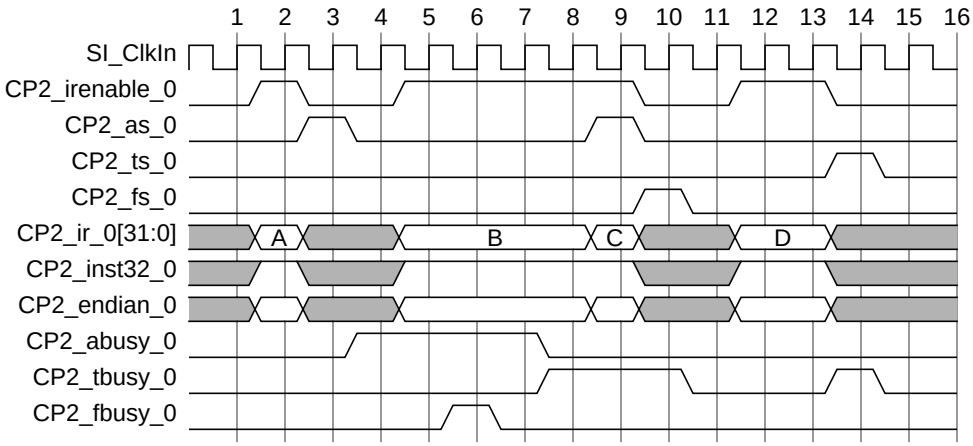


Figure 5-2 Instruction Dispatch Waveforms

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 4KE 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 4KE 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 4KE 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 4KE from reordering To COP Data by driving *CP2_tordlim_0[2:0]* to 000_2.

Note: The 4KE 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.

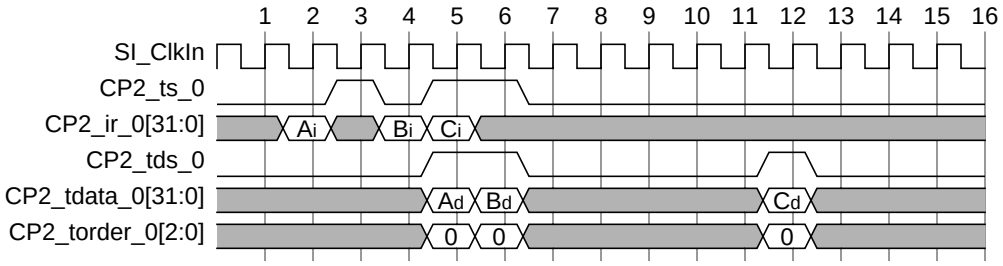


Figure 5-3 To Coprocessor Data Waveforms

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 4KE 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 4KE 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 4KE 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 4KE 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.

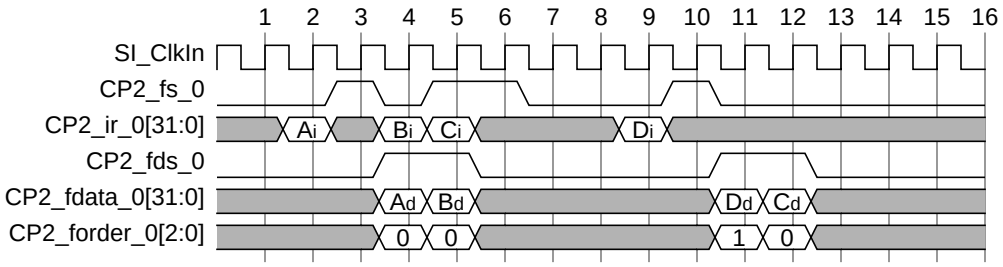


Figure 5-4 From Coprocessor Data Waveforms

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 4KE core. The data for instruction A is one cycle delayed, causing one stall cycle in the 4KE 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 4KE 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 4KE core. The values are not available until the cycle after return from the coprocessor.

Note: The 4KE core always stalls for a minimum of 2 cycles in E-stage for any BC2 instruction sent to the coprocessor.

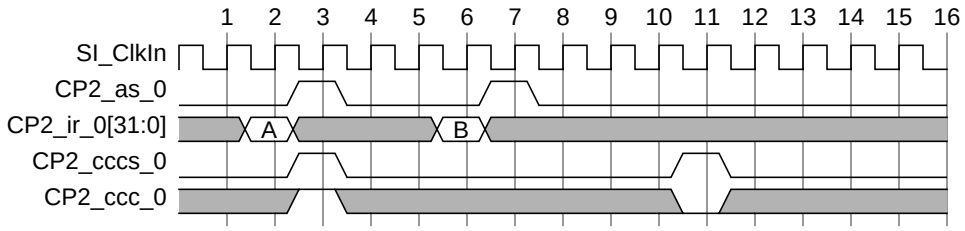


Figure 5-5 Condition Code Check Waveforms

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 4KE 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 4KE 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.

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 4KE 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]*).

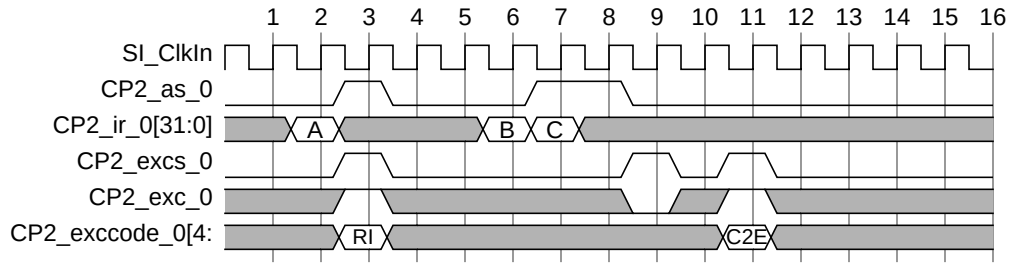


Figure 5-6 Exception Waveforms

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 4 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.

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 4KE 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 47.

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.

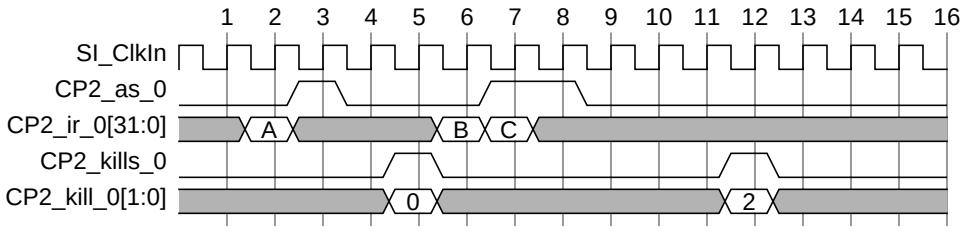


Figure 5-7 Instruction Killing Waveforms

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 4KE core and the coprocessor.

5.5.1 No coprocessor Present

If a hard-core version of a 4KE 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 4KE 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 4KE 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 4KE 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 4KE core. When **WAIT** is executed, the 4KE core will stall the front of the pipe, and wait for all older instruction and pending bus activity to complete. Once this is detected,

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 4KE 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 4KE core. After the **WAIT** instruction has actively stopped the main clock to most of the 4KE 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 4KE 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 4KE 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.

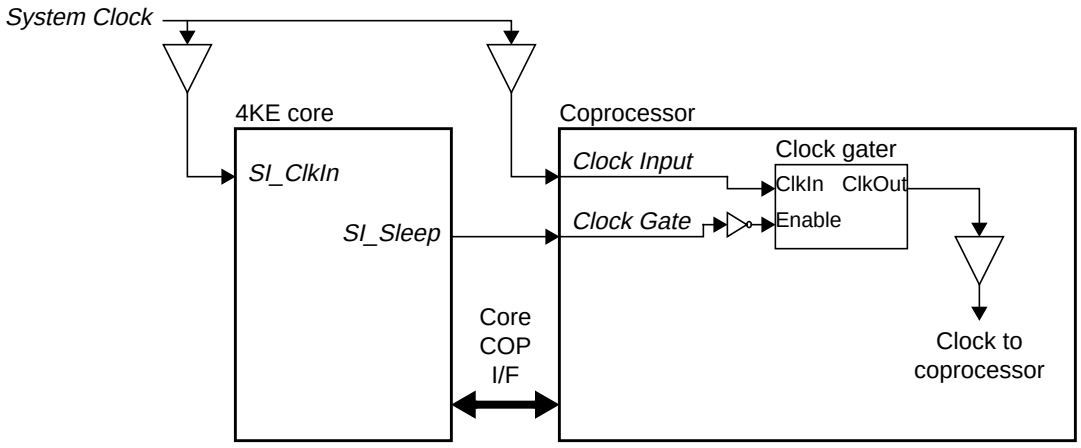


Figure 5-8 Use of SI_Sleep for Clock-Gating in the Coprocessor

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 58 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.

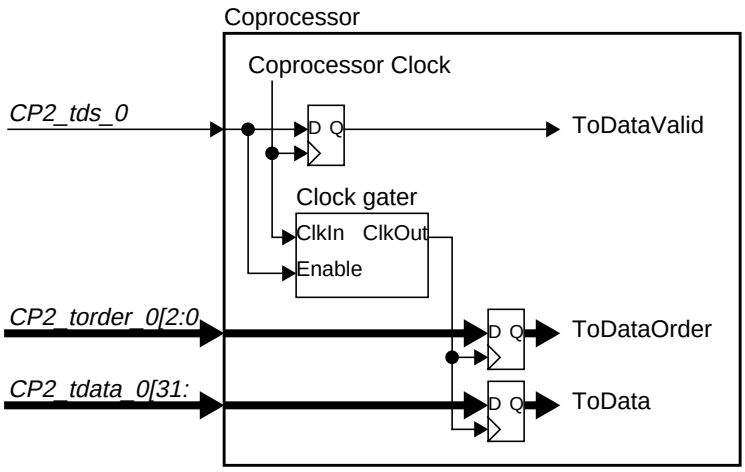


Figure 5-9 Clock-Gating of To Data Registers in Coprocessor

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.

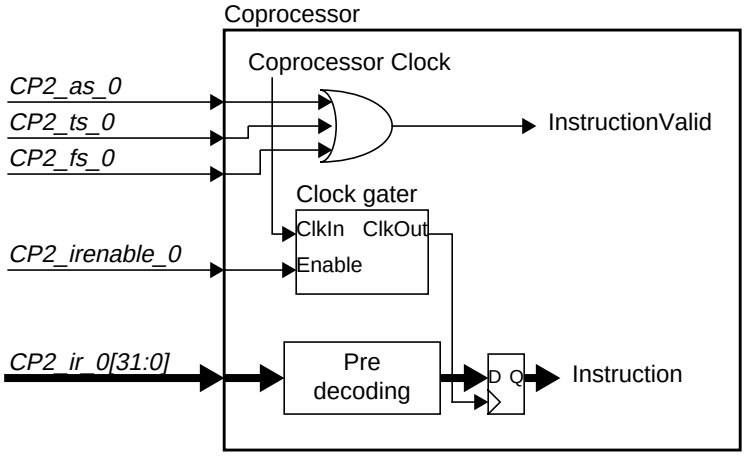


Figure 5-10 Clock Gating of Instruction Registers in Coprocessor

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 4KE core is to be maintained. Refer to Table 5-4 on page 48, for information on allowable latencies to maintain stall-free operation.

Scratchpad RAM Interface

The Scratchpad RAM (SPRAM) option on a MIPS32™ 4KE™ core is designed to provide low-latency access to on-chip memories. SPRAM is supported for both instruction and data references. The SPRAM port is accessed in parallel with the caches. This saves a number of cycles that would normally be required going through the BIU and the EC interface.

The pin list associated with the SPRAM interface was introduced in Chapter 2, “Signal Description,” on page 3. This chapter contains further details about the use of the interface in a system. The chapter contains the following major sections:

- Section 6.1, "SPRAM Features"
- Section 6.2, "SPRAM Overview"
- Section 6.3, "SPRAM Interface Transactions"
- Section 6.4, "External Access to Scratchpad Memory"
- Section 6.5, "SPRAM Initialization"
- Section 6.6, "Using the same design for ISPRAM and DSPRAM"
- Section 6.9, "Reference Design"

6.1 SPRAM Features

SPRAM combines some features of main memory and caches. SPRAM has the following features:

- A SPRAM data array can be up to 1MB in size, much larger than the maximum 64KB cache size.
- There are separate interfaces to instruction SPRAM (ISPRAM) and data SPRAM (DSPRAM). The presence of SPRAM on the I-side or D-side can be independently configured.
- The ISPRAM and DSPRAM interfaces are not completely symmetric. There are no stores to the ISPRAM, so this asymmetry saves some pins.
- A full tag array is not needed for SPRAM. The equivalent tag functionality is normally replaced by a simple decode of the physical address to determine hit or miss.
- The cache way-select (WS) array is not needed for SPRAM.
- A SPRAM port logically replaces one way of a cache. If both SPRAM and cache are present, then the maximum cache associativity is 3.
- Stores to a data SPRAM only go to the scratchpad, and are never written to main memory. For local data, this will reduce the bus bandwidth associated with store traffic, as compared to the write-through or write-back protocols employed by stores to a data cache.
- Instruction SPRAM can service uncached references, enabling processor boot with no EC interface accesses.
- Backstalling. The SPRAM port can stall the core if the SPRAM array was busy the previous cycle or if data is not ready. This can enable other sources to access the SPRAM without the need for dual-porting the array. This is useful, for example, if there is a DMA engine filling the SPRAM or if a unified I/D SPRAM is desired. A cache, in contrast, has fixed single-cycle timing.

6.2 SPRAM Overview

A block diagram of a basic 4KE system with SPRAM functionality is shown in Figure 6-1.

The SPRAM interface is designed to be flexible enough to work with a variety of system designs. A variety of memory devices can be connected to the SPRAM interface: SRAM, ROM, flash, etc. If desired, memory-mapped functions can also be connected, as long as the interface protocol is met. Multi-ported devices can also be used; in this case, the ISPRAM or DSPRAM interface is logically connected to just one of the ports, with other system logic unrelated to the 4KE core utilizing the other port(s).

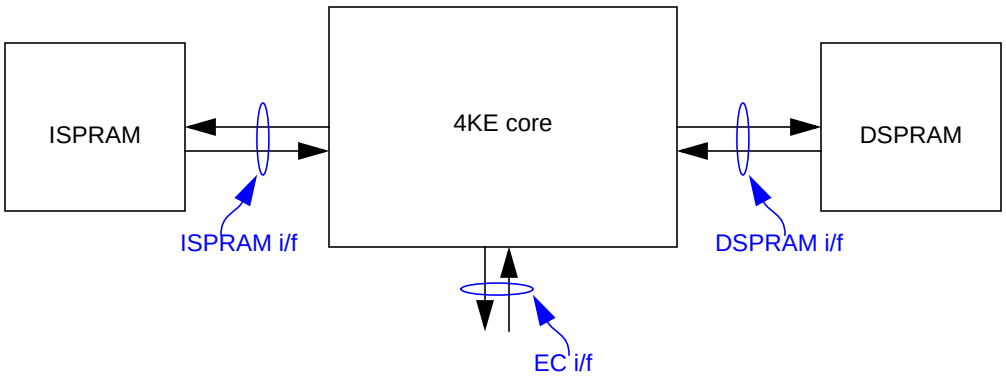


Figure 6-1 Basic SPRAM Block Diagram

The SPRAM array effectively replaces a cache way and is always located at the last cache way. A SPRAM array can be used with or without caches. If caches are present in conjunction with SPRAM, then the maximum cache associativity is 3. The existence of an ISPRAM or DSPRAM interface must be selected at build time for the 4KE core. Even if selected at build time, an SPRAM device need not be connected to the interface. In this case, the SPRAM-related core input pins should be tied off to 0.

The SPRAM array, like the cache arrays, is indexed with a virtual address and the “tag” comparison (really just decode logic for an SPRAM) is performed using a physical address. Note that because the SPRAM “way” can be larger than the 1KB minimum page size, it is possible to have virtual aliasing in the SPRAM. (The potential aliasing issue exists only in TLB-mapped regions with the 4KEc™ core; with the fixed MMU in the 4KEm™ and 4KEp™ cores, the effective minimum page size is 512MB). Virtual aliasing occurs when a single physical address is accessed via two different virtual addresses that can simultaneously be resident in memory. This is not handled by the hardware and programmers must be aware of it.

During normal operation, it will be impossible for a reference to hit in both the SPRAM and cache. If this error condition does occur via manipulation of the cache or SPRAM tags, the cache overrides the SPRAM and the SPRAM hit indication is ignored.

6.2.1 SPRAM Differences versus a Cache

SPRAM behaves much like a cache way, with a few exceptions:

- Software must ensure a SPRAM entry has been initialized before it is read, to avoid reading spurious data.
- ISPRAM never refills automatically. To move instructions into the SPRAM, software must use the CACHE instruction.
- DSPRAM does not fill automatically, either. It should normally be initialized with stores to the address range.

- Store operations which hit in the DSPRAM do not produce writes to main memory, unlike write-through stores that hit in the cache and write to main memory.
- The SPRAM array is not required to hold the last read value.

6.2.2 Independent Tag/Data accesses

The D-side SPRAM interface has independent tag and data ports. This is done to aid the efficiency of stores. A store must perform a lookup to determine if/where to write the data, then the actual data must be written. Because the lookup does not need to access the data array, these operations can occur in parallel if the data writes are buffered within the core, as described further in Section 6.2.4, "Delayed Stores".

Many of the signals on the SPRAM interface apply to only one of the tag/data accesses, while others apply to both. Table 6-1 shows which signals are related to tag access, data access, or both and when they are logically valid.

Table 6-1 SPRAM Interface Cycle Timing

Signal Name	Port	Dir.	Typical Timing, as % of min. cycle	Validity relative to strobes/stalls
ISP_Addr	Both	Out	80	This is valid during the cycles that RdStr, TagWrStr, or DataWrStr are asserted. If Stall is asserted, this value will be held until the cycle that Stall is deasserted.
ISP_RdStr	Both	Out	90	Asserted when tag and data lookups are being performed.
DSP_TagAddr	Tag	Out	80	This is valid during the cycle that TagRdStr or TagWrStr is asserted. If Stall is asserted, this value will be held until the cycle that Stall is deasserted.
DSP_TagRdStr	Tag	Out	90	Asserted when a tag lookup is being performed
{I,D}SP_TagWrStr	Tag	Out	90	Asserted when a CACHE instn is writing the tag - note: this will never be asserted in the cycle after TagRdStr/RdStr to avoid a conflict on TagCmpValue
DSP_TagCmpValue	Tag	Out	40	For reads, this is valid the cycle after TagRdStr/RdStr. If Stall is asserted, this value will be held until the cycle after Stall is deasserted. For writes, this is valid the same cycle as TagWrStr.
DSP_DataAddr	Data	Out	80	This is valid during the cycle that DataRdStr or DataWrStr is asserted. If Stall is asserted the following clock, this value will be held until the cycle that Stall is deasserted.
DSP_DataWrValue	Data	Out	80	This is valid in the same cycle that DataWrStr is asserted. If Stall is asserted the following clock, this value will be held until the cycle that Stall is deasserted.
ISP_DataTagValue	Data	Out	40	This is valid in the same cycle that DataWrStr is asserted. If Stall is asserted the following clock, this value will be held until the cycle that Stall is deasserted.
	Tag	Out	40	For reads, this is valid the cycle after TagRdStr/RdStr. If Stall is asserted, this value will be held until the cycle after Stall is deasserted. For tag writes, this is valid the same cycle as TagWrStr.
{I,D}SP_DataRdStr	Data	Out	90	Asserted when a data read is being performed - this will never be asserted unless TagRdStr is also asserted.
{I,D}SP_DataWrStr	Data	Out	90	Asserted when a data write is being performed.
DSP_DataWrMask	Data	Out	80	Valid when DataWrStr is asserted
{I,D}SP_DataRdValue	Data	In	60	For single cycle access, read data should be returned the cycle after DataRdStr/RdStr is asserted. For multi-cycle accesses, read data should be returned in the same cycle that stall is deasserted.

Table 6-1 SPRAM Interface Cycle Timing (Continued)

Signal Name	Port	Dir.	Typical Timing, as % of min. cycle	Validity relative to strobes/stalls
{I,D}SP_TagRdValue	Tag	In	70	For single cycle access, tag value should be returned the cycle after TagRdStr/RdStr is asserted. For multi-cycle accesses, tag value should be returned in the same cycle that stall is deasserted.
{I,D}SP_Hit	Tag	In	60	For single cycle access, this should be valid the cycle after TagRdStr/RdStr is asserted. For multi-cycle accesses, this should be valid in the same cycle that Stall is deasserted.
{I,D}SP_Stall	Both	In	40	The Stall signal can be related to either Tag or Data access. Because both Tag and Data accesses can occur at the same time, the input should be the OR of both Tag and Data stall sources.
	Tag			Should be asserted in the cycle after TagRdStr/RdStr if hit determination cannot be returned or tag value is not available. Remains asserted until the lookup can be completed. It is not possible to stall a tag write.
	Data			Should be asserted in the cycle after DataRdStr/RdStr if read data cannot be returned. Remains asserted until the read data is available. Should be asserted in the cycle after DataWrStr if the data write has not been completed.
{I,D}SP_Present	Both	In	Static	Static configuration input

6.2.3 Timing Considerations

The SPRAM interface, unlike the other external interfaces on a 4KE core, is not fully registered; however, all signals are synchronous to the rising edge of the primary core clock, *SI_ClkIn*. Outputs on the SPRAM 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. 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 SPRAM interface depends on many factors, such as maximum target frequency, process technology, standard cell library characteristics, setup and access times for the SPRAM array, etc., so it is difficult to provide a generic set of timing guidelines that will apply in all situations. The “Typical Timing” column in Table 6-1 shows the timing of SPRAM 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 SPRAM 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 **_Hit* and especially **_Stall* signals are critical inputs to the core. Care must be taken in the amount of logic performed by the external SPRAM block when driving these signals. For example, stall generation based on the decoding of the physical address (*DSP_TagCmpValue* or *ISP_DataTagValue*) is probably not possible if maximum frequency is desired. For lower target frequencies, of course, the timing constraints shown in Table 6-1 can be relaxed.

6.2.4 Delayed Stores

A store buffer exists within the core for holding the last store data. Due to the separate tag and data accesses described in Section 6.2.2, "Independent Tag/Data accesses", the store data written to the DSPRAM data array is actually for the previous store, while the “tag” address is for the current store. This means that the DSPRAM data array for a specific store is not guaranteed to be written until the *next* store is executed in the pipeline. During cycles in which the DSPRAM is otherwise idle, pending store data can be written with no corresponding tag access. If the store buffer is empty when the current store is processed, then only the “tag” transaction will occur.

6.2.5 Tag Reads and Writes

The interface allows for “tag” values to be read and written. This capability is not used in normal operation. The tag values are read/written by the CACHE instruction. This can optionally provide a mechanism for software to determine the SPRAM configuration and change it. The reference design shows one possible use for this interface - software can probe the SPRAM to determine the base address and whether it is enabled. These values are also write-able, allowing software to dynamically configure the SPRAM parameters. A more complex SPRAM could use tag values at multiple indexes to encode even more configuration information.

6.2.6 Uncacheable References to SPRAM

Normally, only cacheable addresses can be serviced by the caches on a 4KE core; uncacheable references ignore the cache data and go directly to the EC bus interface. The I-side SPRAM interface has been enhanced to optionally allow uncacheable references to occur to the ISPRAM. This feature enables booting directly from an ISPRAM that has been pre-loaded with the boot sequence (booting always begins in an uncacheable region of memory, as dictated by the MIPS32™ Architecture).

The uncacheable access is handled by essentially providing an extra physical address bit to the ISPRAM, via *ISP_DataTagValue[1]*. When high, this bit indicates that the reference is uncacheable, while a low value denotes a cacheable address. An ISPRAM implementation may choose to use this additional information to qualify the generation of *ISP_Hit* as desired.

The uncacheable support only applies to the ISPRAM, and references to the D-side SPRAM must always be cacheable. The DSPRAM interface does not contain the “extra” physical address bit on the *DSP_TagCmpValue* bus. Because the cacheability information may not be determined until after the cache/SPRAM access has started, uncacheable references will still be presented to the DSPRAM, but the data will not be used and an external bus request will be initiated. The DSPRAM does not need to directly support uncacheable references, since software can ensure that references intended for the DSPRAM are always specified within a cacheable region of memory.

6.2.7 Backstalling the SPRAM interface

The normal cache interface has fixed single-cycle timing. Both the I- and D-side SPRAM interfaces allow the SPRAM to backstall the core if it is busy, via assertion of the *{I,D}SP_Stall* signal. This mechanism may be used to support multi-cycle timing on the SPRAM interface. For example, the backstall mechanism could allow a single-port SRAM to arbitrate between the core access and an external interface.

The following considerations should be noted when using the backstalling capability:

- When *{I,D}SP_Stall* is asserted, the *{I,D}SP_Hit* signal is ignored by the core.
- The *{I,D}SP_Stall* signal is a timing-critical input to the core. Care should be taken when creating the *{I,D}SP_Stall* signal, as it feeds into the main pipeline stall logic and must be valid approximately halfway into the cycle. The stall signal is also used asynchronously by the core to prevent the next access from occurring, and to conditionally hold some interface signals valid from the prior request. For these reasons, the *{I,D}SP_Stall* timing is generally more critical than *{I,D}SP_Hit*. In low-frequency applications, the stall signal may be generated combinationally, based on the physical address presented on the *DSP_TagCmpValue* or *ISP_DataTagValue* buses; for timing reasons, however, this is not recommended when maximum core frequency is desired. For max. frequency, it is recommended to derive the stall signal from the strobes and index address asserted in the previous cycle, as well as the fact that some external device is using the SPRAM array in the current cycle.
- If *{I,D}SP_Stall* is asserted for an address that hits in the cache, the cache hit is preserved but the core pipeline will be needlessly stalled as long as *{I,D}SP_Stall* is asserted.
- Refer to Table 6-1 for a description of how core outputs behave when stall is asserted. The strobe signals are not held asserted by the core during a stall. The address and write values are held valid during a stall, for the related tag or data

port that is active. For example, if a read transaction is occurring on the tag port but the data port is idle, then the address and/or write value associated with the tag port will be held valid during a stall, but the addresses or write value on the data port is “don’t care” data and may change during the stall sequence.

6.2.8 Access Granularity

The widths of the data bus for read and write requests to SPRAM are shown in Table 6-2. A read always returns a word (32 bits) of data. The core internally handles any alignment necessary for sub-word read requests, like byte loads or MIPS16 instruction fetches.

Table 6-2 Read and Write Width for SPRAM Arrays

Array	Max Read Width (bits)	Min Read Width (bits)	Max Write Width (bits)	Write Granularity (bits)
ISPRAM	32	32	32	32
DSPRAM	32	32	32	8

Writes to ISPRAM are always a full word. The maximum width of DSPRAM writes is a word, but one, two, or three bytes can be written as well. The byte lanes to be written to DSPRAM are controlled by the *DSP_DataWrMask[3:0]* bus, as shown in Table 6-3; when a bit in *DSP_DataWrMask* is high, the corresponding byte from *DSP_DataWrValue* should be written to the array.

Table 6-3 Byte Control for DSPRAM Writes

DSP_DataWrMask bit asserted	*DSP_DataWrValue* bits to be written
[0]	[7:0]
[1]	[15:8]
[2]	[23:16]
[3]	[31:24]

6.2.9 Unified I/D SPRAM

Separate interfaces are provided from the core to I- and D-side SPRAM. It is possible to create a shared I/D SPRAM, if desired. A unified SPRAM could allow a system to dynamically share the same memory array between the needs of instruction and data, as compared to the build-time partitioning which must be done for the separate Harvard-style interfaces.

If a unified SPRAM is desired, the existing I and D SPRAM interfaces on the core would need to be brought into a common external block, as illustrated in Figure 6-2. Since I and D requests can occur in the same cycle, a method to handle simultaneous requests will be required. A dual-ported memory could be used to handle simultaneous I/D requests. With a single-ported memory, the backstalling mechanism described previously is one way the I/D prioritization could be achieved.

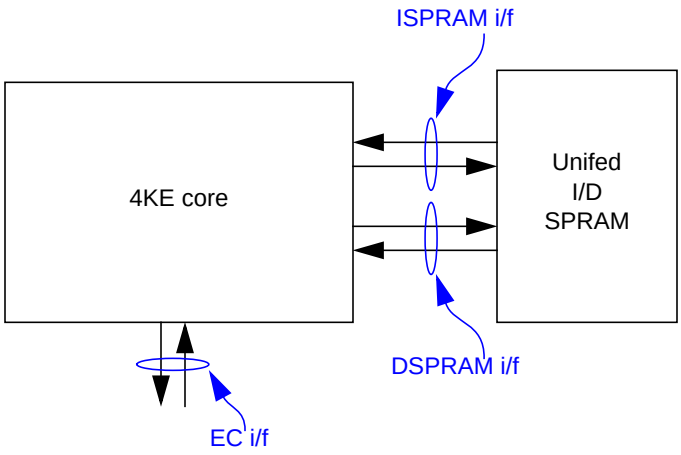


Figure 6-2 Unified I/D SPRAM Block Diagram

6.2.10 SPRAM considerations with MIPS16

If SPRAM is used in conjunction with the MIPS16 capabilities of the 4KE core, a new twist must be considered. MIPS16 includes support for PC-relative loads, in which a data address reads the instruction stream (typically to read constants intermixed with the instruction text segment). For a normal cache-based processor core, there is always backing system memory which is used to service cache misses. If a PC-relative load misses in the D-cache, it will be filled from system memory. Both the I-cache and D-cache could contain copies of the “same” location, but this is generally ok since PC-relative loads are used to access constants located in the instruction memory and are not normally written.

If an ISPRAM is present, however, an issue can arise with servicing the D-side PC-relative load if there is no backing memory in the system. The core itself does not handle this situation, so it will need to be dealt with by the system hardware and/or software, if required.

If a unified I/D SPRAM is employed, then no action will be required since all instruction and data values are already located in the same array. For separate ISPRAM and DSPRAM arrays, the exact method to handle the PC-relative loads will depend on the system. In some cases, the compiler might have the ability to separate text and constant segments, so the code might be loaded into ISPRAM and constants could be loaded into DSPRAM when the arrays are initialized.

If the constants are really located in the ISPRAM array, then when a PC-relative load is presented to the DSPRAM, the DSPRAM block will need to detect the address range(s) where PC-relative load is held, backstall the DSPRAM interface, divert the DSPRAM reference to the ISPRAM (or elsewhere in the system) to get the constant data, and then return the data to the core.

6.2.11 Restartability of SPRAM accesses

The location of the SPRAM interface within the pipeline has some implications related to events which may cause an transaction to be replayed. Exceptions that occur late in the pipeline, after the SPRAM access has already occurred, can cause the instruction which caused the access to be killed and possibly re-executed at a later point in time, depending on the exception handler. Examples of such exceptions include interrupts, bus errors, and EJTAG or Watch breakpoints. These exception are detected after the SPRAM access has occurred, but the exception PC will point to the instruction which caused the access, or perhaps even a preceding instruction. Hence, the SPRAM accesses will generally need to be restartable, so the SPRAM device must be capable of re-playing the read or write after the exception has been processed. Care must be taken for memory-mapped devices which may be attached to the SPRAM interface, so they can handle the potential replay of a read or write access.

6.2.12 Connecting I/O Devices to the Scratchpad Interface

In addition to, or perhaps instead of, an SRAM array, it is possible to connect I/O devices to the SPRAM interface. Connecting I/O devices to the cache interface allows low latency, high throughput access to critical I/O devices in the system. To accomplish this, the implementer must ensure that the behavior of the I/O devices meets the same requirements as the SPRAM. I/O devices connected to the SPRAM port must be capable of re-playing reads and writes with no adverse effects, as described in Section 6.2.11, "Restartability of SPRAM accesses".

6.2.13 Null connection to unused SPRAM interface

The presence of ISPRAM and/or DSPRAM interfaces must be chosen when the core is built. Even if the SPRAM interface is present, there need not be a device connected to it. If the interface is not to be used, then the *{I,D}SP_Present* input signal to the core should be driven low. All other input signals to the core for the unused SPRAM interface should also be tied low, to avoid floating inputs. All output signals from the core related to the unused SPRAM interface can be left unconnected.

6.3 SPRAM Interface Transactions

Strobe signals on the SPRAM interface determine the type of transaction that is active. In general, there are independent interfaces for “tag” accesses and “data” accesses. For some transaction types, both the tag and data interfaces are used to process the same request. In other cases, the tag and data interfaces may process unrelated requests.

Table 6-4 shows the type of transaction indicated by the tag/data read/write strobe signals. All four strobes are present on the DSPRAM interface. On the ISPRAM interface, there are three strobes: a single read strobe and separate tag/data write strobes. Note that some strobe combinations never occur.

Table 6-4 SPRAM Transaction Types

DataWrStr	TagWrStr	TagRdStr	DataRdStr	Transaction Type
0	0	0	0	No access
X	0	0	1	Not possible
0	0	1	0	DSPRAM: Store address lookup with no data write
0	0	1	1	ISPRAM: Instruction fetch or CACHE read (fill or index load tag) DSPRAM: Load or CACHE read (index load tag)
0	1	0	0	CACHE write (index store tag)
0	1	0	1	Not possible
0	1	1	X	Not possible
1	0	0	0	ISPRAM: CACHE write (index store data) DSPRAM: Idle cycle store or CACHE write (index store data)
1	0	1	0	ISPRAM: Not possible DSPRAM: Store lookup with store data write
1	0	1	1	Not possible
1	1	X	X	Not possible

This remainder of this section contains timing diagrams for typical read and write transactions to SPRAM. Since the DSPRAM interface is a superset of the ISPRAM interface, the diagrams only depict DSPRAM transactions. The relationship of interface signals which are only present on the DSPRAM interface to the ISPRAM is discussed in Section 6.6, "Using the same design for ISPRAM and DSPRAM" on page 75.

6.3.1 Single Read

Figure 6-3 shows the timing diagram for a single SPRAM read. This scenario can occur for the following conditions:

- data load to DSPRAM
- instruction fetch to ISPRAM
- CACHE read (index load tag to either SPRAM or a fill lookup to ISPRAM)
- store address lookup to DSPRAM, when no previous store data is pending (in this case *DSP_TagRdStr* will assert, but *DSP_DataRdStr* will not)

The 4KE core initiates the read by asserting read strobe signals (*DSP_TagRdStr*, *DSP_DataRdStr*) and by driving a valid index on the address busses (*DSP_TagAddr*, *DSP_DataAddr*) during cycle 1. Typically, these signals are used by synchronous logic in the external DSPRAM block to perform a read on the rising edge of cycle 2. Also during cycle 2, the physical address for tag comparison (*DSP_TagCmpValue*) is driven by the core.

The external DSPRAM block uses the strobe and address information driven by the core to determine that the address is indeed within the range mapped by the SPRAM array, and that the requested read data can be returned immediately. Thus, the external logic asserts *DSP_Hit* and deasserts *DSP_Stall* in cycle 2, while driving the read data on bus *DSP_DataRdValue*. For the minimum read response, the hit and stall signals must be signalled combinationally after performing a tag comparison on the physical address provided on bus *DSP_TagCmpValue*. The external logic might also return tag read data associated with a CACHE instruction request, on bus *DSP_TagRdValue*, if that is relevant for the SPRAM implementation.

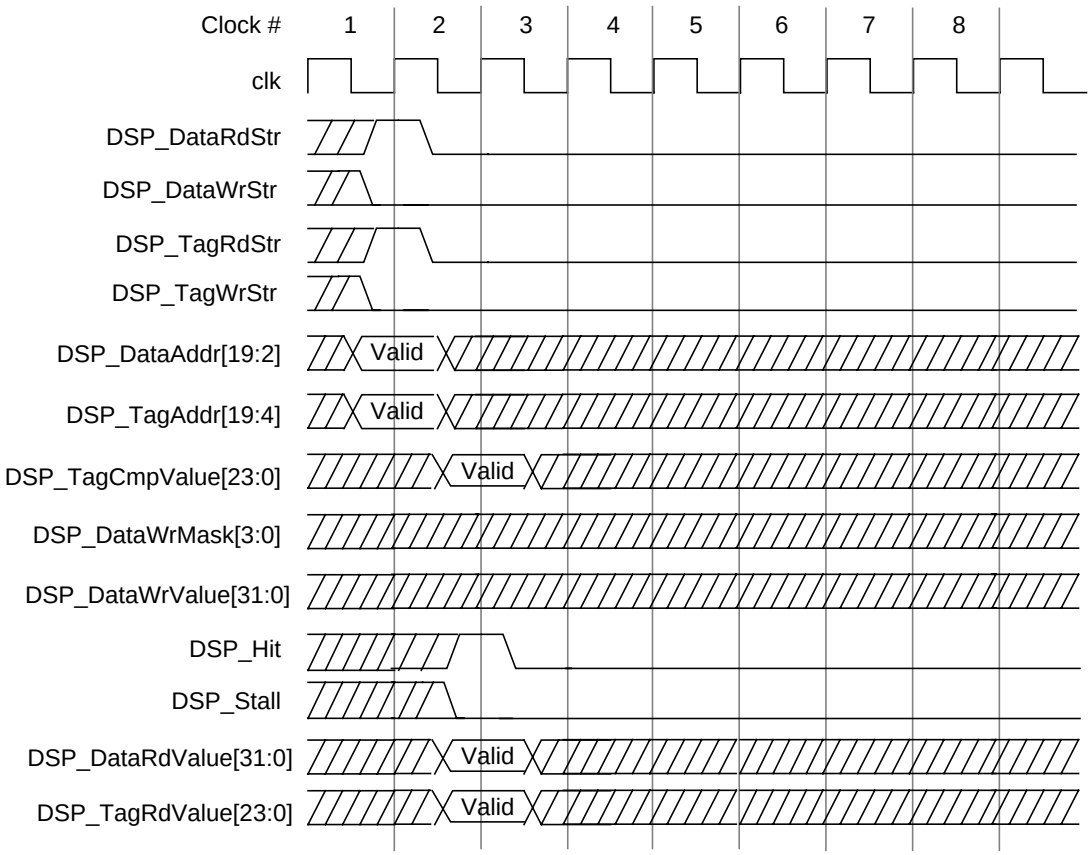


Figure 6-3 Single DSPRAM Read

6.3.2 Single Multi-Cycle Read

Figure 6-4 shows the timing diagram for a single DSPRAM multi-cycle read, and illustrates the back-stalling capability of the interface. This is similar to the single-cycle read case described in Section 6.3.1, "Single Read", but now the external SPRAM logic was unable to immediately service the read request.

The read request is initiated by the core in cycle 1 by driving read strobes and index addresses. In cycle 2, however, the SPRAM access cannot be completed for some reason, so the external logic responds by asserting *DSP_Stall*. The value driven on *DSP_Hit* is ignored by the core whenever stall is asserted. The stall indication is used combinationally by the core to hold the index addresses valid for the original request. In this case, stall is asserted for two cycles, and is finally deasserted in cycle 4. During cycle 4, the SPRAM array access proceeds, and external logic asserts hit and drives the requested read data.

Note that while stall is asserted, the index and tag addresses are held by the core, but the strobe signals are not. The core will never assert another strobe request while stall is asserted.

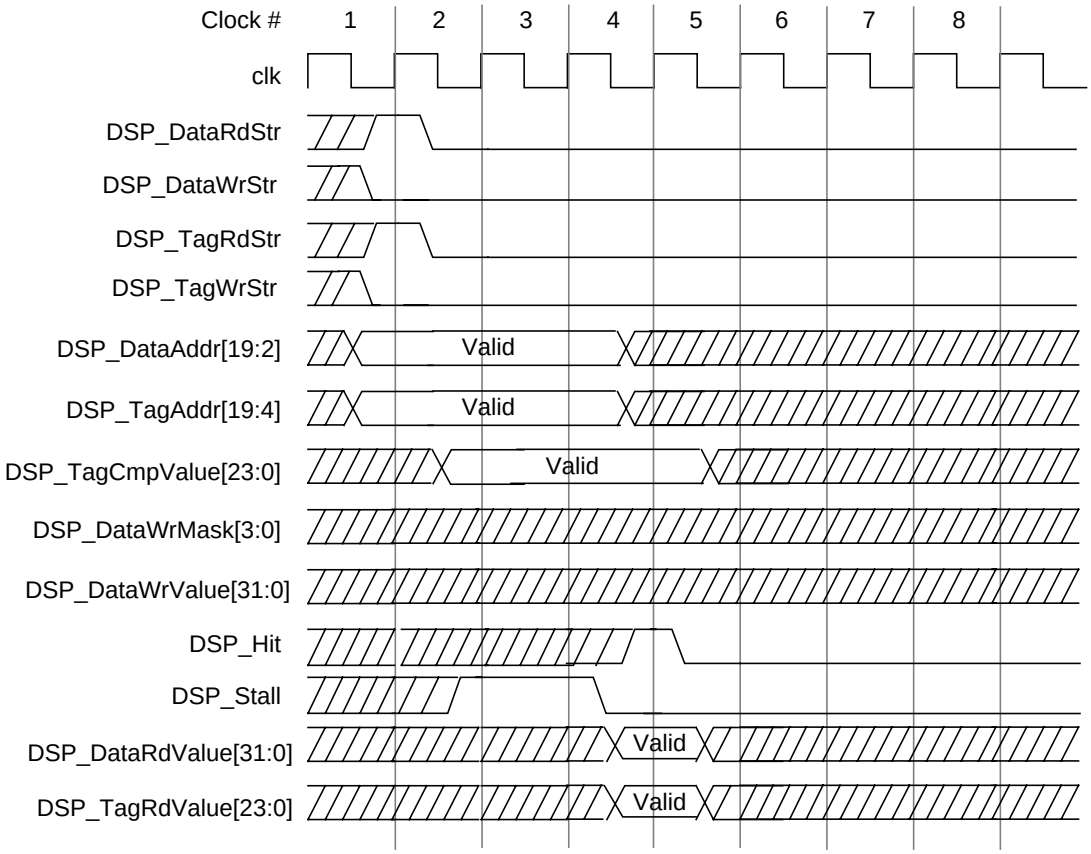


Figure 6-4 Single Multi-Cycle DSPRAM Read

6.3.3 Single Write

Figure 6-3 shows the timing diagram for a single DSPRAM write. Only *DSP_DataWrStr* is active. This is the simplest write scenario and can occur when store data is sent to the DSPRAM during an idle cycle when no other loads or stores are being processed. This case can also occur when a CACHE index store data operation is presented to the SPRAM array.

The core initiates the write in cycle 1, by driving the write strobe (*DSP_DataWrStr*), data index address (*DSP_DataAddr*), write data (*DSP_DataWrValue*), and byte mask (*DSP_DataWrMask*). Since the tag strobes are not active, the tag index address and tag physical address are not valid.

The SPRAM logic is able to complete the write during cycle 2, so the stall signal (*DSP_Stall*) is deasserted and the transaction completes. Note that the hit signal (*DSP_Hit*) is not looked at by the core, since the tag strobe signals are inactive.

For an idle cycle store, the tag address transaction occurred at some earlier point. The core tracks the fact that the previous tag lookup was a DSPRAM hit, and presents only the corresponding index address and store data to the DSPRAM for the transaction here.

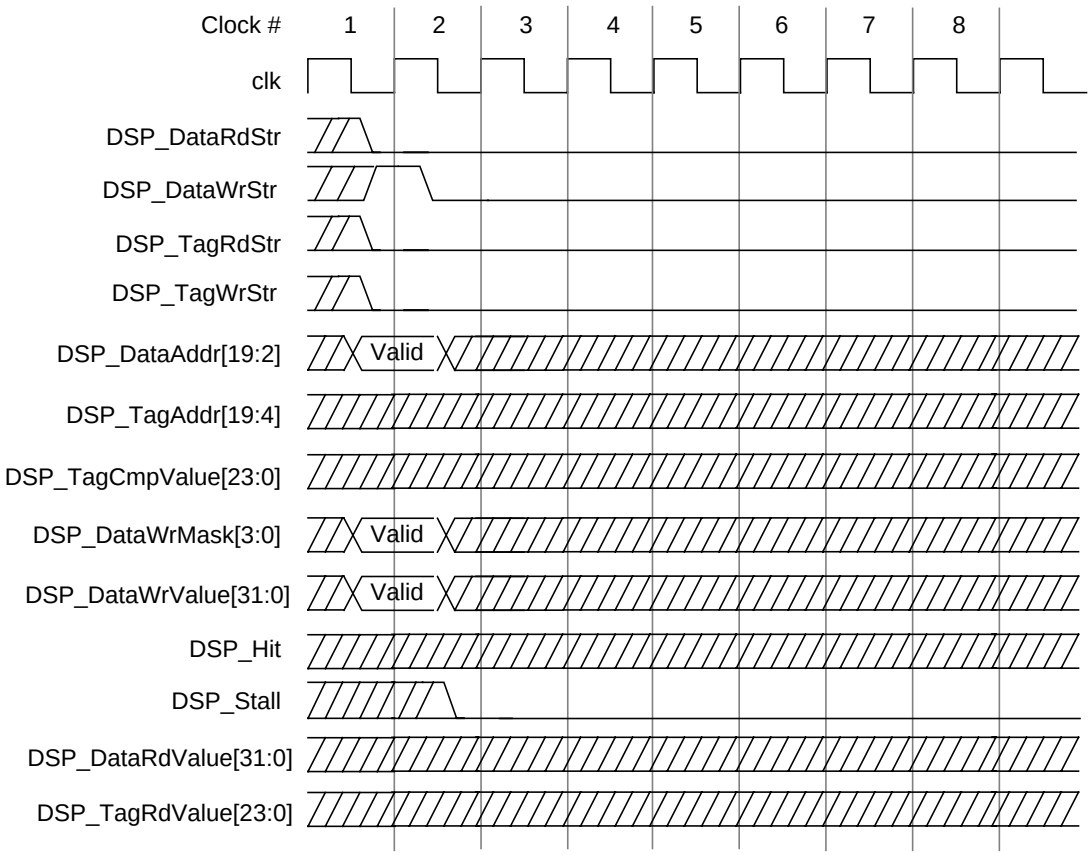


Figure 6-5 Single DSPRAM Write

6.3.4 Single Multi-Cycle Write

Figure 6-6 shows the timing diagram for a single DSPRAM multi-cycle write, and illustrates the back-stalling capability of the interface. This is similar to the single-cycle write case described in Section 6.3.3, "Single Write", but now the external SPRAM logic was unable to immediately service the write request.

The core initiates the write in cycle 1, by driving the data write strobe, index address, store data, and byte mask. But the external DSPRAM logic is unable to process the write during cycle 2, so it responds by asserting DSP_Stall, in this case for two cycles. The core holds the address, store data, and byte mask valid while the stall signal is asserted. During cycle 4, the write could proceed, and the external logic then deasserts stall to complete the write transaction.

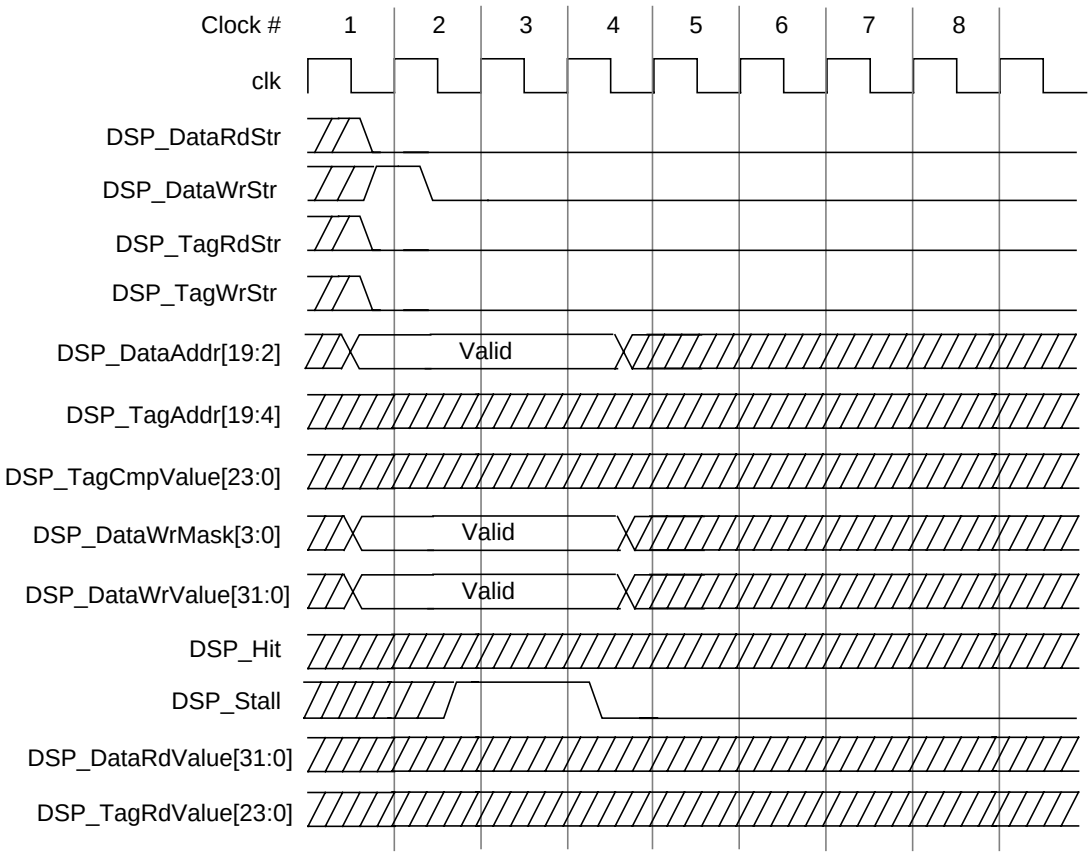


Figure 6-6 Single Multi-Cycle DSPRAM Write

6.3.5 Simultaneous Tag Read and Data Write

Figure 6-7 presents a transaction in which a write to the data interface occurs at the same time as a read to the tag interface. This is an extension to the data write-only transaction discussed in Section 6.3.3, "Single Write", but is a typical occurrence to DSPRAM, when an address transaction occurs for the current store instruction, while store data from a previous DSPRAM hit is simultaneously presented to the data array. This situation never occurs to ISPRAM.

The core initiates the transaction in cycle 1, by asserting the tag read strobe (*DSP_TagRdStr*) and data write strobe (*DSP_DataWrStr*). On the data interface, the data index address, data value, and byte mask, all corresponding to the prior store which hit in the DSPRAM, are also driven in cycle 1. On the tag interface, tag index address for a new store is driven in cycle 1, while the physical address for that store is driven in cycle 2.

The external DSPRAM logic is able to process both the tag and data transactions during cycle 2, so it asserts hit (*DSP_Hit*) based on a successful physical address comparison, and deasserts stall (*DSP_Stall*), thereby completing both the tag and data portions of the transaction. The tag read value (*DSP_TagRdValue*) is also shown as being driven valid in cycle 2. This value is probably not relevant for this type of store transaction, but the external logic may choose to always drive this bus in response to the tag read strobe for simplicity.

Note that the interface does not permit the tag read and data write transactions to be completed independently, since there is a single stall signal. The external DSPRAM block must complete both operations in cycle 2, or assert stall to complete them in a later cycle.

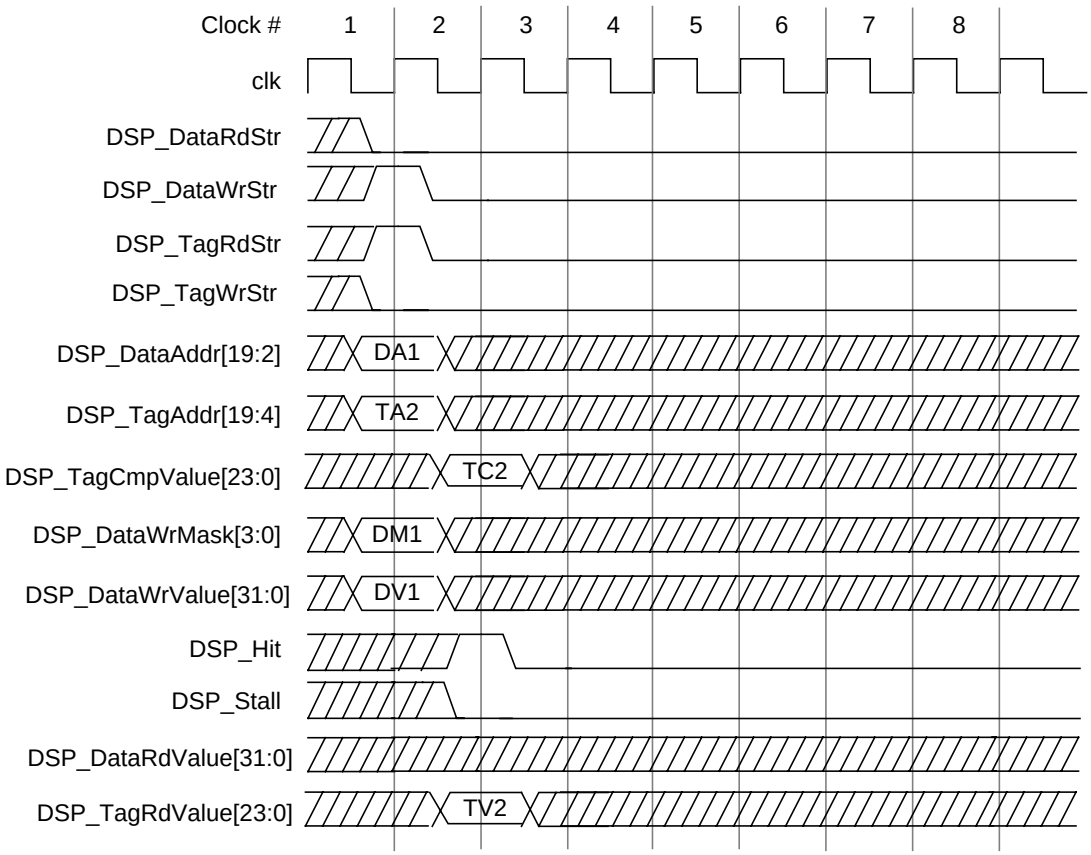


Figure 6-7 Combined DSPRAM Tag Read and Data Write

6.3.6 Back-to-Back Reads

The SPRAM interface is fully pipelined, and any combination of the previously introduced single-transactions can be combined in consecutive cycles. The core will never initiate a new transaction whenever stall is asserted, however.

Figure 6-8 shows two back-to-back read transactions. Each individual transaction looks like the single-cycle read introduced in Section 6.3.1, "Single Read".

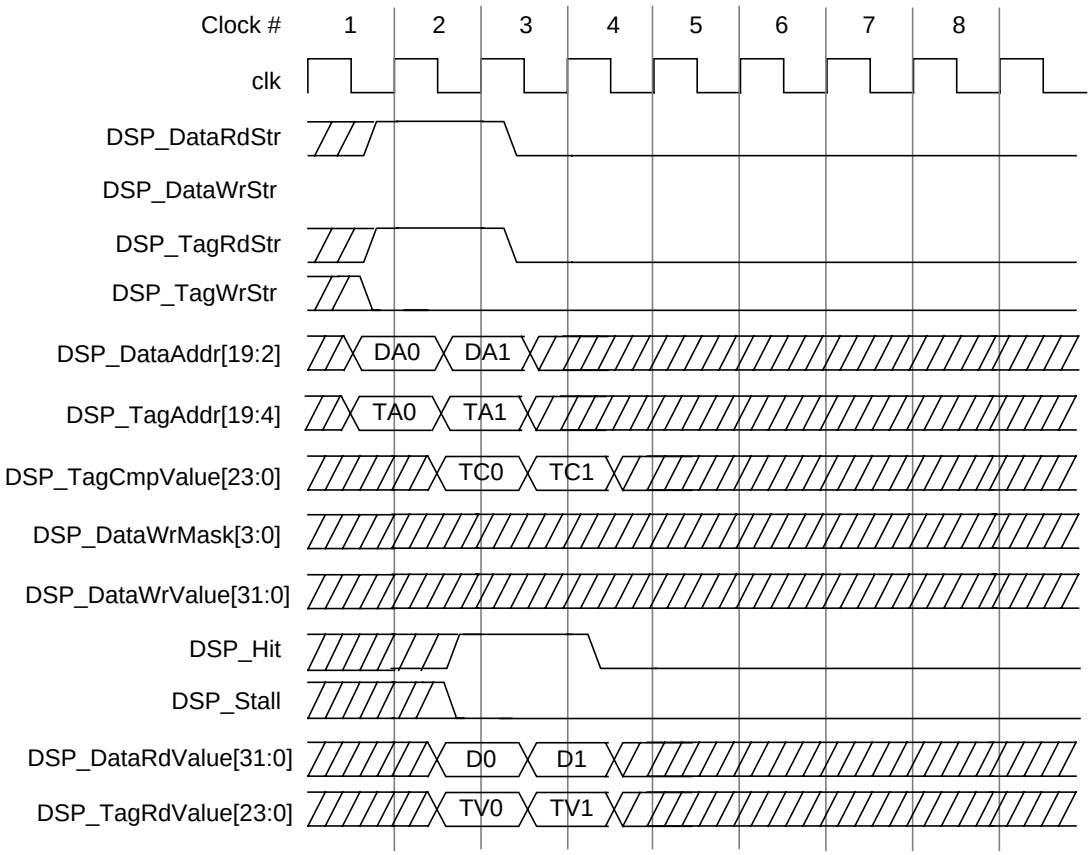


Figure 6-8 Consecutive DSPRAM Reads

6.3.7 Read-Write-Read Sequence

Figure 6-9 depicts a three transaction sequence, consisting of a single-cycle read, followed by a data store (with simultaneous tag read) that is stalled for two cycles, and finally followed by another single-cycle read.

The first and last reads are like single-cycle read described in Section 6.3.1, "Single Read". The data store is derived from the single-cycle transaction introduced in Section 6.3.5, "Simultaneous Tag Read and Data Write", but the completion has been stalled for two additional cycles.

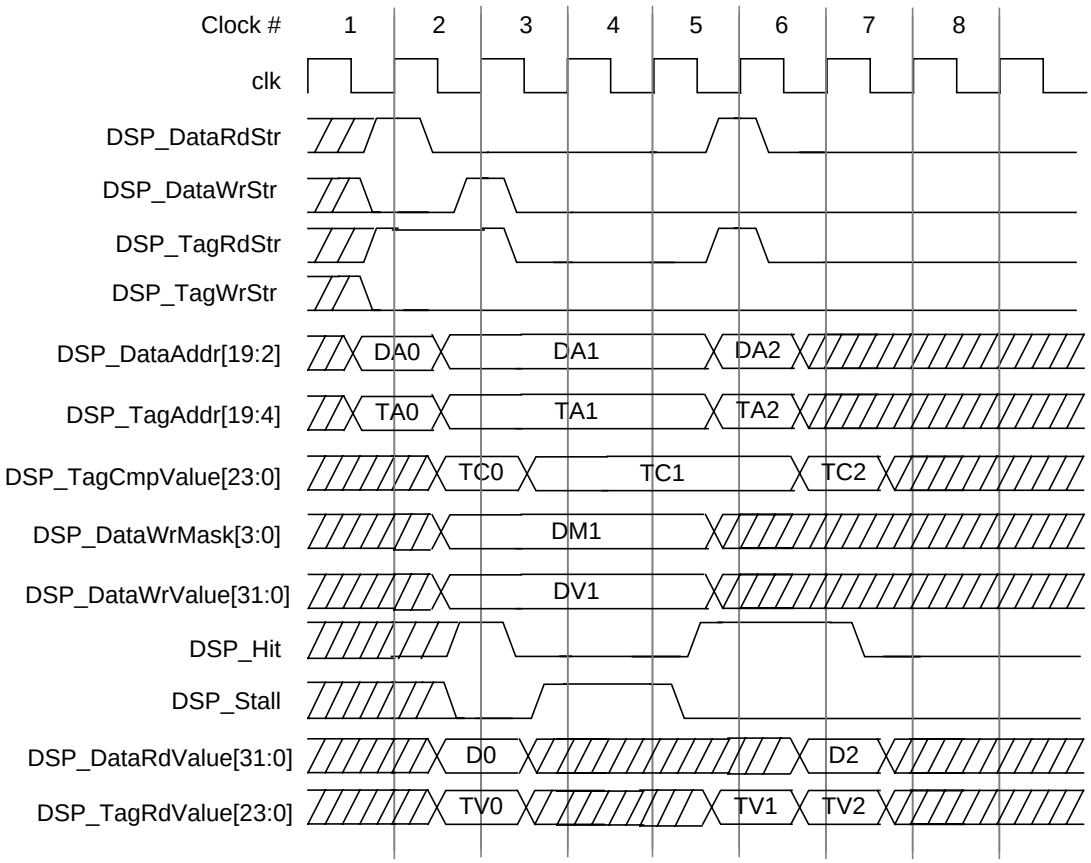


Figure 6-9 Read-Write-Read

6.4 External Access to Scratchpad Memory

A system design may desire access to the SPRAM by a source external to the core, referred to as a *backdoor.* Creating such an external access path quickly becomes a system architecture issue which is beyond the scope of this document, but here are a few methods which could be considered:

1. Use the backstalling capability of the SPRAM interface to allow arbitration between the core and backdoor to a single-ported SRAM, as shown in Figure 6-10. The arbitration logic can backstall the core by asserting the *{I,D}SP_Stall* signal when the core attempts an access at the same time as an external device.
2. Use a true dual-ported SRAM. The core can use one port, and the backdoor can use the other. Software only has to ensure that the same address is not written on both ports at the same time.
3. Split the SPRAM into two or more banks. Under software control, the backdoor could then gain access to one bank, while the core accesses the other(s). This method might also be combined with the backstalling capability, but stalls should be less frequent.

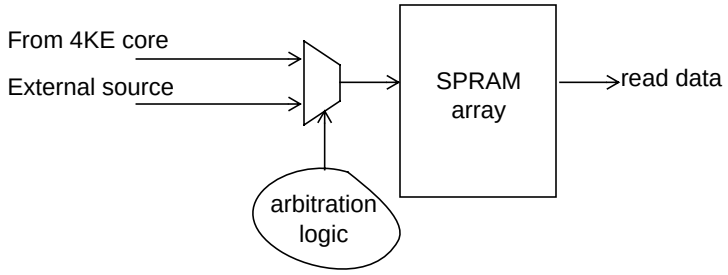


Figure 6-10 External Access to Single-ported SPRAM

6.5 SPRAM Initialization

If the scratchpad is really a RAM-based structure, then it must be initialized with valid data before it can be used. There are several standard mechanisms to handle this. One or more of these options should be available depending on your system.

- CACHE fill instruction: In general, executing the Fill version of the CACHE instruction forces a refill of the cache from main memory. If the reference hits in the cache, the fill will go to the same way to avoid a conflict. This mechanism works for the SPRAM as well: if the reference hits in the SPRAM, the cache controller will try to fill to the SPRAM way. The CACHE Fill instruction is only available for the I-cache (and thus ISPRAM) and it requires backing memory at the SPRAM address, since the fill will be serviced via the EC interface.
- Stores: For DSPRAM, the array can be initialized with normal store instructions that hit in the SPRAM region.
- CACHE Index Store Data instruction: Indexed cache operations can be forced to go to the SPRAM by setting the *SPR* bit in the Coprocessor0 *ErrCtl* register. When this bit is set, it is possible to use the Index Store Data flavor of the CACHE instruction to move data from the *DataLo* Cop0 register into the SPRAM. This mechanism does not require any backing memory and can even be used to load the SPRAM from an EJTAG probe for early system bringup. This method can be used for either the ISPRAM or DSPRAM, although using stores to initialize DSPRAM is much more efficient. It is recommended that all SPRAM implementations support this method in addition to any other loading mechanisms.
- Backdoor port: If there is an external DMA port into the SPRAM, then the system can load data directly into the array. This can be done while holding the core in reset or by backstalling any core references to the SPRAM. This would work for either an I-side or D-side SPRAM.

6.6 Using the same design for ISPRAM and DSPRAM

In order to minimize the number of pins on the external interface, the I-side and D-side SPRAM interfaces are not identical. The I-side is more constrained in the type of possible writes, so several of the busses are shared. For design reuse considerations, it may be desirable to only develop one SPRAM module and use it on both ports. The common module should have all of the ports for the DSPRAM. Table 6-5 shows how ISPRAM signals should be connected to appropriate DSPRAM ports.

Table 6-5 ISPRAM Connection to DSPRAM Ports

DSPRAM port	ISPRAM port	Description
DSP_DataAddr	ISP_Addr[19:2]	The I-side Tag and Data ports share the same address.
DSP_TagAddr	ISP_Addr[19:4]	

Table 6-5 ISPRAM Connection to DSPRAM Ports

DSPRAM port	ISPRAM port	Description
DSP_TagRdStr	ISP_RdStr	Both Tag and Data are always read at the same time on the I-side.
DSP_DataRdStr	ISP_RdStr	
DSP_TagCmpValue	ISP_DataTagValue[23:0]	This bus is shared on the I-side because only one of the following actions can occur in any given cycle: Data write, tag write, or tag compare
DSP_DataWrValue	ISP_DataTagValue[31:0]	
DSP_DataWrMask	4'hf	On an I-side data write, all 4 bytes of the given word will always be written at the same time.

6.7 Multiple SPRAM regions

It is possible to map multiple SPRAM regions into a single SPRAM block. Note, however, that the entire array is indexed with a virtual address. This places constraints on the virtual addresses associated with the given regions. This may in turn place constraints on the physical address of the region.

Figure 6-11 shows 3 regions within a single memory array. Several of the bits of the VA are fixed for each region. Figure 6-12 shows 2 regions built in separate arrays. In this case, only one bit of the virtual address is fixed. For region 0, VA<11> can be either 0 or 1. Using PA<11> in the hit determination will select one of the two spots and leave a hole in the other one.

Figure 6-11 Multiple SPRAM regions

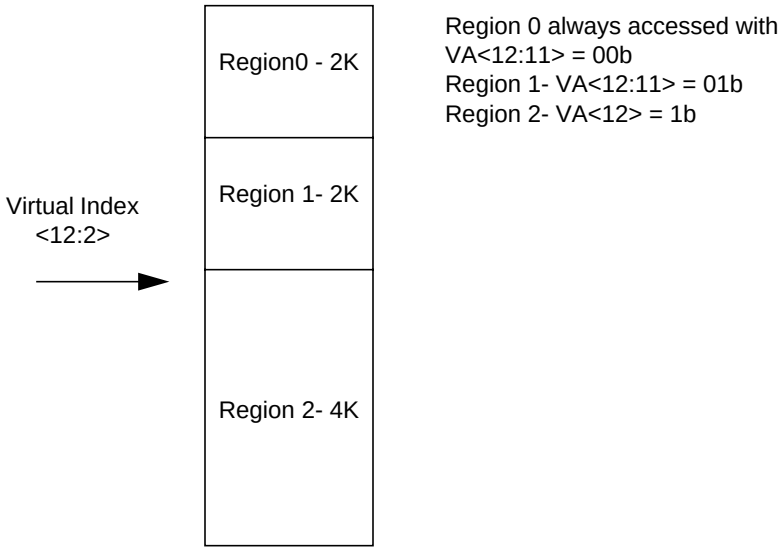
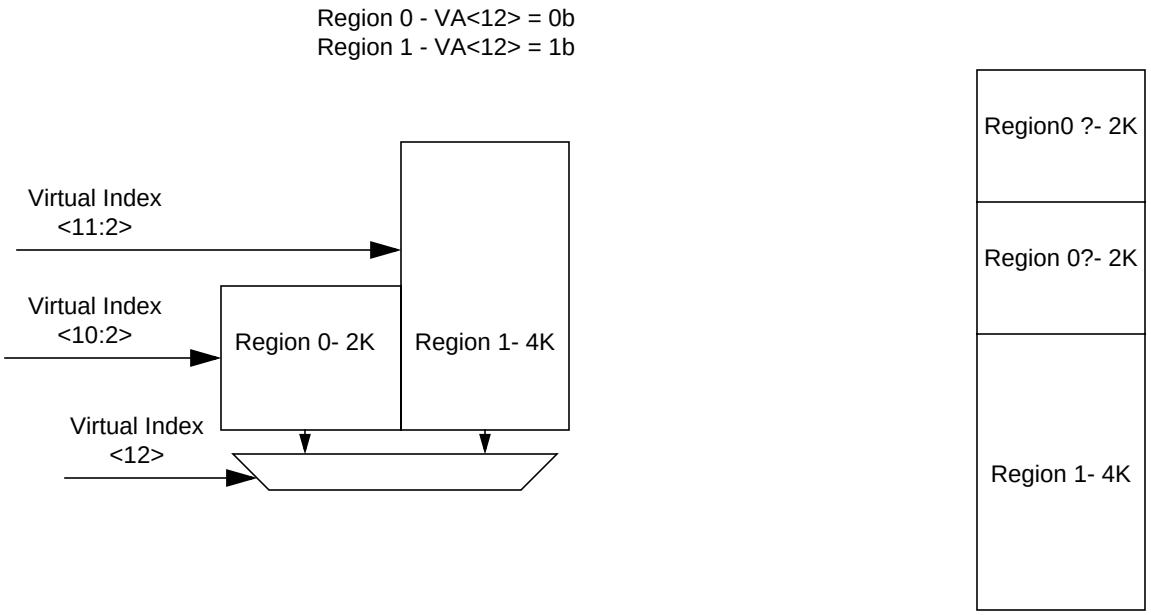


Figure 6-12 Multiple SPRAM regions in separate arrays



6.8 Implementation recommendations

The SPRAM interface provides a great deal of flexibility. That flexibility can make it difficult for standard toolchains and debuggers to work with the SPRAM. By adhering to a few standard features, that interface can be made simpler and may have better tool support.

6.8.1 Software visible configuration information

Using the CACHE instruction, it is possible to read or write the ‘tag’ value associated with the SPRAM. To provide a common software interface, it is recommended that all SPRAM implementations provide some standard configuration information via this mechanism.

If the SPR bit in the ErrCtl register is set, an Index Load Tag CACHE instruction will read the SPRAM tag and place the contents in the TagLo register. The index value (bits 19:4) will be passed to the SPRAM block which is used to select between different configuration registers. These are the recommended read values that will allow identification of a SPRAM consisting of one or more blocks of memory. Additional configuration information can be stored in unused fields or unused indices. If there is a hole in the virtual address space in the SPRAM, other discontiguous regions should have their own ID registers and be marked as not valid/enabled.

The tag read value for the first index in a region should be the following:

```
[23:2] PA - bits [31:10] of base address for memory region
[1] Lock - Indicate whether an ISPRAM block can service uncacheable fetches
             unused for DSPRAM
[0] Valid - memory region is enabled
```

The tag read value for the second index in a region should be:

```
[23:2] PA - size of memory region (number of 16B lines)
[1:0] Lock/Valid - unused
```

Using the size information, software can determine the first index associated with the following memory region. This chain can be followed in a linked list fashion until all memory regions have been identified. The end of the list can be indicated by one of three values in the next set of registers.

1. Size = 0
2. PA/Size = PA/Size of previous region
3. PA/Size = PA/Size of first region

Method one is preferable, but the second and third methods can be used to reduce the amount of hardware required for generation of the tag read values.

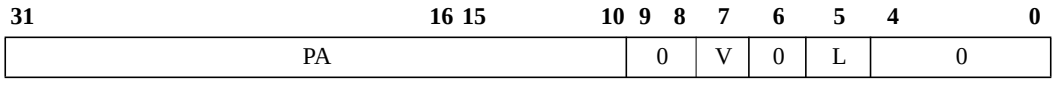
Here’s an example showing the tag registers associated with 3 discontiguous SPRAM regions:

```
16KB region at PA: 0x0000_0000
16KB region at PA: 0x0080_0000
64KB region at PA: 0x0001_0000

Tag 0 - {22’h0, 1’h0, 1’h1}
Tag 1 - {22’h400, 2’h0}
Tag 1024 - {22’h2000, 1’h0, 1’h1}
Tag 1025 - {22’h400, 2’h0}
Tag 2048 - {22’h40, 1’h0, 1’h1}
Tag 2049 - {22’h1000, 2’h0}
Tag 6144 - {24’h0}
Tag 6145 - {24’h0}
```

Note that these bits will be remapped to the format of the TagLo register:

***TagLo* Register Format**



6.8.2 Region sizes

Note that the encoding described in Section 6.8.1, "Software visible configuration information" imposes restrictions on the size of memory regions within the SPRAM. The minimum size is 32B and the size must be a multiple of 16B.

6.8.3 Unique addresses

In order to provide a simple programming interface, it is recommended that if ISPRAM and DSPRAM are simultaneously present, they should have unique addresses and do not overlap. If there is backing memory for the SPRAM regions, the same address can exist in both SPRAM and main memory, but otherwise it should not.

6.8.4 Support ISPRAM writes

In a very simple system, the data write port on the ISPRAM seems extraneous. This write port can, however, be used by a CACHE Index Store Data instruction to manipulate the contents of the ISPRAM. One case where this could be helpful is when debug software inserts breakpoints in the instruction stream.

6.8.5 Virtual Aliasing

When placing SPRAMs in an address region that is mapped via the TLB, there is a potential problem with virtual aliasing. The SPRAM is virtually indexed and physically tagged. A virtual address is used to index into the SPRAM and the following cycle, a physical address is presented for the hit determination.

Virtual aliasing is possible. This is the condition where one physical address can exist in different memory locations if it is accessed with different virtual addresses. This can be avoided by using a page size the same size or larger than the SPRAM, or by forcing a 1-1 VA-PA translation on bits used to index the SPRAM.

6.9 Reference Design

A simple example of scratchpad control logic is shown in Section 6.9.1, "Example SPRAM Block". The example module supports a very basic scratchpad implementation that could be used for either ISPRAM or DSPRAM. It is configurable within certain constraints:

- SPRAM size can range from 1KByte to 1MByte in powers of 2.
- A base physical address for the SPRAM location in physical memory needs to be specified. The address range must be naturally aligned (i.e. a 64KB SPRAM’s base address must be on a 64KB boundary).
- The size and address range are incorporated into the SPRAM model via **‘defines** within the module.
- The array always returns data in a single cycle and does not utilize the backstalling capabilities of the general SPRAM interface.
- Booting from the ISPRAM is not supported; hits only occur on cacheable references.

This is intended to be a very basic example of how a SPRAM block might be implemented. If this limited capability is sufficient, it is possible to use this code by instantiating a real SRAM and modifying the constants.

Figure 6-13 on page 82 shows the simple hookup of SPRAM and the hit logic in this example. As shown, the *Hit* logic is very simple. This is the reason for the limitations on size and base address. No real tag array exists, but the base address, masked with the SPRAM index size, is compared to the physical address to determine a hit and generate the *Hit* signal.

6.9.1 Example SPRAM Block

Verilog code for a possible SPRAM implementation is shown below:

```
module m4k_dspram(TagAddr,TagRdStr,TagWrStr,TagCmpValue,DataAddr,
   DataWrValue,DataRdStr,DataWrStr,DataWrMask,Clk,Reset,Present,
   DataRdValue,TagRdValue,Hit,Stall);

`define M4K_DSPRAM_DEPTH 13
`define M4K_DSPRAM_SIZE 32*1024
`define M4K_DSPRAM_BASE 32'h00058000

   parameter DEPTH = `M4K_DSPRAM_DEPTH;     // Bits of index needed for Array
   parameter SIZE = `M4K_DSPRAM_SIZE;       // Size of array in bytes
   parameter BASE_PA = `M4K_DSPRAM_BASE;    // Base Physical Address
   parameter TAG_BIT_BOUND = 2 + DEPTH;   /* Lowest bit of PA that is part of tag */

   input [19:4]TagAddr;// Index into tag array
   input  TagRdStr;// Tag Read Strobe
   input  TagWrStr;// Tag Write Strobe
   input [23:0] TagCmpValue;    // Data for tag compare {PA[31:10], 2'b0}

   input [19:2]DataAddr;       // Index into data array
   input [31:0]DataWrValue;// Data in
   input  DataRdStr;// Data Read Strobe
   input  DataWrStr;// Data Write Strobe
   input [3:0]DataWrMask;

   input  Clk;   // Clock
   input  Reset;          // Reset

   /* Outputs */
   output Present;        // Static output indicating spram is present
   output [31:0]DataRdValue;// Read data
   output [23:0]TagRdValue;// read tag
   output Hit;        // This reference hit and was valid
   output Stall;          // Read has not completed

wire   spram_enable_cond, Stall, Present, Hit, spram_enable, spram_enable_cnxt;
wire [31:10] spram_base_boot, spram_base_cnxt, zeros, ones, spram_base, tag_mask;
wire [31:0] Size, spram_base32;
wire [23:0] TagRdValue;
wire spram_base_cond, in_spram_range, TagAddr4_Reg;

   assign Present = 1'b1;

   /* SRAM Array: */
   sram_array sram_array (
             .clk(Clk),
             .line_idx(DataAddr[(2+DEPTH)-1:2]),
             .rd_str(DataRdStr),
             .wr_str(DataWrStr),
             .wr_mask(DataWrMask),
             .wr_data(DataWrValue),
             .rd_data(DataRdValue)
             );

   /* spram_enable: Master enable signal for the SPRAM.  Writing to
    * index 0 of the 'tag-array' with teh valid bit set will enable SPRAM
    */
```

```
   cregister #(1) _spram_enable(spram_enable,spram_enable_cond, Clk,
spram_enable_cnxt);
   assign spram_enable_cond = Reset || ( TagWrStr && (TagAddr[11:4] == 8'd0));
   assign spram_enable_cnxt = !Reset && ( TagCmpValue[0] == 1'b1 );

   /* tag_mask: Masking bits for Tag-compare address
    * The masking is based on the size of the SPRAM, rounded up to the
    * nesrest power of 2 size.
    */
   assign ones [31:10] = {22{1'b1}};
   assign zeros [31:10] = 22'b0;

   assign tag_mask [31:10] = { ones[31:TAG_BIT_BOUND], zeros[TAG_BIT_BOUND-1:10]};

   /* Two options for configuring SPRAM base address:
    * Hard code it -
    */
   //   spram_base32 [31:0] = BASE_PA;
   //   spram_base [31:10] = spram_base32[31:10] & tag_mask;

   /* Programmable -
    * Offer ability to configure it via Idx Store Tag cacheops
    */
   assign spram_base32 [31:0] = BASE_PA;
   assign spram_base_boot [31:10] = spram_base32[31:10];
   assign spram_base_cond = Reset || ( TagWrStr && (TagAddr[11:4] == 8'd1));
   assign spram_base_cnxt [31:10] = Reset ? (spram_base_boot & tag_mask) :
               (TagCmpValue[23:2] & tag_mask);
   cregister #(22) _spram_base_31_10_(spram_base[31:10],spram_base_cond, Clk,
spram_base_cnxt);

   /* Pseudo Tag-array and Decode Logic */

   // SPRAM Hit signal
   // TagCmpValue[1] is used on I-side to indicate uncached reference
   assign in_spram_range = ~|((TagCmpValue[23:2] ^ spram_base) & (tag_mask));
   assign Hit = in_spram_range && spram_enable && !TagCmpValue[1];

   /* TagRdValue: Tag Return data
    * TagEntry 0 - Base Address
    * TagEntry 1 - Size
    * Software can probe 'tags' to determine size/location
    */
   cregister #(1) _TagAddr4_Reg(TagAddr4_Reg,Clk, TagAddr[4]);

   assign Size [31:0] = `M4K_DSPRAM_SIZE >> 4;

   assign TagRdValue [23:0] = {TagAddr4_Reg ? Size[21:0] : spram_base, 1'b1 /* Lock
*/, spram_enable /*Valid*/};

   assign Stall = 1'b0;

endmodule

module cregister(q, cond, clk, d)
   parameter WIDTH = 1;
   output [WIDTH-1:0] q;
   reg [WIDTH-1:0]    q;
   input     clk;
   input     cond;
```

```
        input [WIDTH-1:0]  d;

        always @(posedge clk)
           if (cond)
              q <= #1 d;

    endmodule
```

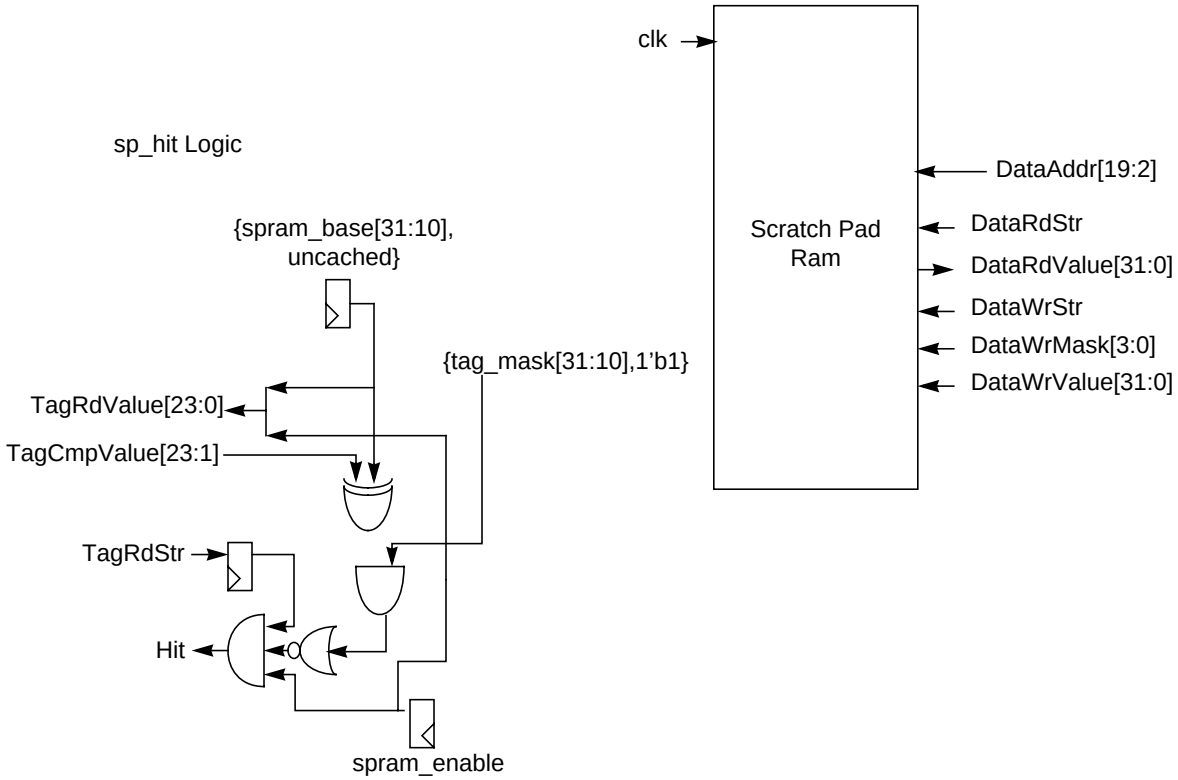


Figure 6-13 SPRAM Hookup and Hit Logic in `m4k_dspram` module

The fixed bit of logic "1" (not shown in Figure 6-13, but in the sample code) and the usage of *spram_enable* on *TagRdValue[1:0]* represents the lock bit and valid bit. The SPRAM behaves like a locked entry. All locations in the SPRAM are per definition always valid, thus the valid bit for each line is always set when the SPRAM is enabled.

In the `m4k_dspram` module, the *spram_enable* register is reset to zero. It must be set to enable the SPRAM. Special software boot-code is required to enable the SPRAM before any attempt to use it is made. To set the *spram_enable* bit in the `m4k_dspram` module, the user must write a tag-entry with the valid bit set, using the CACHE instruction, to index zero of the "virtual" tag entry for the SPRAM.

A register is used to hold the *spram_base* base address. This register is reset to the value defined at the top of the file, but can be overwritten by writing to index 0 of the "virtual" tag entry for the SPRAM. This makes the base address programmable by software, which may prove useful for a more general implementation. Note that special software boot code is required to place the SPRAM in the right address space, before any attempt to use it is made.

Software can probe the tag entries for the SPRAM to learn how it is configured. Using the CACHE Index Load Tag instruction, the tag values can be read. When index 0 of the tag array is read, the base physical address of the SPRAM is returned and placed into *TagLo*. When index 1 of the tag array is read, the size (in number of 16B lines) is returned.

Performance Monitoring Interface

This chapter describes the Performance Monitoring(PM) interface, which is present on a MIPS32 4KE™ processor core. This interface allows a system designer to implement performance counters which can be used for profiling software and hardware performance.

The specific pins on the interface are prefixed with *PM_*, and were introduced in Chapter 2, “Signal Description,” on page 3. This chapter includes further details about the use of the PM interface, and contains the following major sections:

- Section 7.1, "PM Interface versus Performance Counters"
- Section 7.2, "Interface Protocol"

7.1 PM Interface versus Performance Counters

The PM interface is a replacement for the Performance Counters present on some other MIPS processors. Performance counters are typically configurable to allow the counting of a variety of processor events. The counters are included in Coprocessor 0 and can be read and configured by kernel routines. Monitoring software can periodically read the registers, or they can be configured to signal an interrupt when the counter overflows. Frequently, multiple counters are implemented so that the relative frequency of different events can be compared over a large sample set - for example the ratio of D-cache hits to misses, or the ratio of micro TLB misses to instructions completed.

The PM interface was implemented instead of Performance Counters for a number of reasons:

- The area associated with the counters can be avoided if counters are not desired.
- The integrator has more flexibility for choosing the number and type of counters, as well as system access to them.
- The added pin bandwidth required is not a significant cost on a core (assuming that any counters will be implemented on-chip).

More general information about Performance Counters can be found in the MIPS32 architecture document: *MIPS32™ Architecture For Programmers Volume III: The MIPS32™ Privileged Resource Architecture* [6]. Another reference is the User’s Manual of a processor implementing Performance Counters such as the *MIPS64 5Kc™ Processor Core Software User’s Manual* [7].

7.2 Interface Protocol

The PM interface is rather simple. Each of the pins, when asserted high, indicates that a particular event has occurred within the core. A pin will be asserted for one cycle for each event. For example, an instruction cache miss will cause *PM_ICacheMiss* to be asserted for one cycle even if the miss is stalling the core for many cycles. Signals will only be asserted once for each instruction that is executed. Counting a specific event simply requires adding an incrementer of the desired width that updates once per cycle when that signal is high.

It is important to note that the PM signals do not travel down the instruction pipeline This has two implications:

1. The signals associated with a single instruction may come out in different cycles depending on which pipe stage they were detected in.
2. Events from early stages of the pipeline will be reported even if the instruction takes an exception.

7.2.1 Events

The following events are available on the PM interface, independently for the instruction and data sides of the core:

- Cache hit and miss. One of these signals asserts whenever a cacheable access occurs. Uncacheable accesses due not assert these signals.
- Micro-TLB hit and miss. One of these signals asserts whenever a valid micro-TLB access occurs to a mapped address. They will never assert for unmapped addresses, or on cores with no TLB.

Additionally, these general events are also available:

- Instruction complete indication. Asserts whenever an instruction completes at the end of the pipeline. Instructions which are killed due to exceptions will not cause an assertion of this signal, but could result in assertion of the other PM signals since they are generally reported earlier in the pipeline.
- JTLB hit and miss. One of these signals asserts whenever a valid JTLB access occurs. The DTLB and JTLB are always accessed in parallel, so a DTLB hit will always produce an assertion of the JTLB hit signal as well. When either the I or D micro-TLBs miss, the JTLB will be accessed and result in either a JTLB hit or miss assertion. For cores with no TLB, these signals will never assert.
- Write buffer merge or no merge. One of these signals asserts whenever a store is presented to the merging write buffer.

For the hit/miss style PM signals, the hit rate can be expressed as follows, based on counts from the coupled hit/miss events:

```
(No. of hits) / (No. of hits + No. of misses)
```

Similarly, the miss rate can be expressed as:

```
(No. of misses) / (No. of hits + No. of misses)
```

The JTLB ratios may be more meaningful if ITLB hits are included in the sum of events, since an ITLB hit implies that there was a mapped address that did not access the JTLB, but would have hit. So the JTLB hit ratio could be expressed as:

```
(No. of JTLB hits + No. of ITLB hits) / (No. of JTLB hits + No. of JTLB misses +
No. of ITLB hits)
```

Then the JTLB miss ratio is:

```
(No. of JTLB misses) / (No. of JTLB hits + No. of JTLB misses + No. of ITLB hits)
```

Another interesting ratio is events/instruction:

```
(No. of events) / (No. of instns)
```

7.2.2 Example Instruction Sequence

Table 7-1 shows several instructions being executed and the corresponding values on the PM interface. A “-” for the value indicates that the value is dependent on instructions not shown in the sequence. The following sequence of instructions are executed in the example:

1. ADD - hits in ITLB, misses in the I-cache
2. SW - hits in ITLB/I-cache and DTLB/JTLB/D-cache, does not merge in WTB
3. LW - hits in ITLB/I-cache, misses in JTLB/DTLB causing an exception
4. SRL - hits in ITLB/I-cache
5. ADD - hits in ITLB, misses in I-cache
6. First instruction of exception handler, hits in I-cache (no ITLB access)

Note that even though instructions 3-5 are killed by the exception, the PM signals are still asserted for events at the beginning of the pipeline (I-cache/ITLB).

Table 7-1 Performance Monitoring Example

Instn	**Pipe Stage**																
Add	I	I	I	I	E	M	A	W									
SW					I	E	M	A	W								
LW - (takes TLBL exc.)						I	E	M	M	A	W						
SRL							I	E	E	M	A	W					
Add								I	I	I	I						
Exc. Handler												I	E	M	A	W	
Signal	**Value**																
PM_DCacheHit	-	-	-	-	0	0	0	0	1	0	0	0	0	0	0	0	-
PM_DCacheMiss	-	-	-	-	0	0	0	0	0	0	0	0	0	0	0	0	-
PM_ICacheHit	-	-	0	0	0	0	1	1	1	0	0	0	0	1	-	-	-
PM_ICacheMiss	-	-	1	0	0	0	0	0	0	1	0	0	0	0	-	-	-
PM_InstnComplete	-	-	-	-	-	-	-	-	1	1	0	0	0	0	0	0	1
PM_ITLBHit	-	1	0	0	0	1	1	1	1	0	0	0	0	-	-	-	-
PM_ITLBMiss	-	0	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-
PM_DTLBHit	-	-	-	0	0	0	0	1	0	0	0	0	0	0	0	-	-
PM_DTLBMiss	-	-	-	0	0	0	0	0	0	1	0	0	0	0	0	-	-
PM_JTLBHit	-	-	-	0	0	0	0	1	0	0	0	0	0	0	0	-	-
PM_JTLBMiss	-	-	-	0	0	0	0	0	1	1	0	0	0	0	0	-	-
PM_WTBMerge	-	-	-	-	0	0	0	0	0	0	0	0	0	0	0	0	-
PM_WTBNoMerge	-	-	-		0	0	0	0	1	0	0	0	0	0	0	0	-

VMC Simulation Model

This chapter discusses the simulation models included in a MIPS32™ 4KE™ core release. It contains the following section:

- Section 8.1, "Cycle-Exact Simulation Model"

8.1 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 4KE processor core.

8.1.1 Installing the VMC Model

1. The 4KE VMC model is supported under the Sun Solaris UNIX and x86 RedHat Linux platforms.
2. The 4KE 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 4KE 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:
   ```
   % $MIPS_PROJECT/vmc[_sun,_linux]/m4ke_vmc_release/sl_admin.csh
   ```
   1. A dialog box labeled “Install From...” should pop up.
   2. Make sure the text input box points to the package, “m4ke_vmc_release”.
   3. Press “Open” to continue.
   4. Another dialog box is used to select the models that will be installed. Only one choice is available in this release, a model called “m4ke_vmc_model” followed by a version number. Click on that model to bring it into the “Models to Install” window.
   5. Click “Continue” to close this dialog box.
   6. Another dialog box to select the platforms for this model installation will appear. Each release package will only contain the model for one platform and that check box should be selected. The appropriate simulator packages used under the “EDAV Packages” heading also need to be specified. Both Verilog-XL and NC-Verilog are covered by the “Cadence Design Systems” push button. Modelsim and VCS have their own buttons. Multiple EDAV packages can be selected and the packages for all simulators that will be used should be selected. Push the “Install” button to continue.
   7. An “Install complete” message in the main message window is received and then exit from the sl_admin tool.
4. During the installation, a documentation directory will be created at `$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.

5. The 4KE 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
     ```

8.1.2 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 4KE VMC model. Invocation is as follows:

```
% $LMC_HOME/bin/swiftcheck m4ke_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.

8.1.3 SWIFT Template Generation

In order to instantiate the 4KE VMC model in the RTL simulation environment, a SWIFT template of the 4KE 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 m4ke_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 m4ke_vmc_model
```

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

8.1.4 Back-Annotating with SDF Timing

This is not supported.

8.1.5 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 8-1 shows the signals available via register windows.

Table 8-1 Core Signals Visible in VMC model

Name	Bits	Description
RFn	[31:0]	Contents of register n. Entries RF1-RF31 of the register file are available. Entry RF0 is always 0. Contents of shadow registers sets are not available as windowed signals.

Table 8-1 Core Signals Visible in VMC model (Continued)

Name	Bits	Description
CPZ_x	[31:0]	Contents of Coprocessor 0 register xxx. All possible 4KE COP0 registers are included, but TLB-related ones are not valid when using the Fixed Block Address Translation instead of the TLB.
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.
InstnVirtual Address	[31:0]	Virtual Address for the Instruction Fetch.
InstnPA	[31:12]	Physical Address for the Instruction Fetch (bits [11:0] are untranslated and thus the same as the VA).
InstnCacheable	[0]	Indicates whether the Instruction Fetch is a cacheable reference.
ICacheHit	[0]	Indicates that Instruction reference hit in the I$.
InstnData	[31:0]	Instruction Data returned for Instruction Fetch.
Data Virtual Address	[31:0]	Virtual Address for the Load/Store reference.
DataPA	[31:12]	Physical Address for the Load/Store reference (bits [11:0] are untranslated and thus the same as the VA).
DataCacheable	[0]	Indicates whether the Load/Store reference is cacheable.
DCacheHit	[0]	Indicates that Load/Store reference hit in the D$.
LoadData	[31:0]	Load Data returned on a Load.
BusType	[2:0]	Indicates what type of Load/Store operation is occurring. Use to qualify DataVA etc. 0-No operation, 1-load, 2-store, 3-prefetch, 4-sync, 5-ICacheOp, 6-DCacheOp
BIU_LWptr	[3:0]	Bus Interface Unit read transaction tracking. LWptr is bumped every time a read address is accepted by the system. LRptr is bumped every time read data is returned. When LWptr != LRptr, the 4KE core is waiting for read data to be returned. Useful for debugging system problems. Core “hangs” are often the result of a system not returning all requested data.
BIU_LRptr	[3:0]	See above.

8.1.5.1 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_m4ke_vmc_model>”);
```

8.1.5.2 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 8.1.3, "SWIFT Template Generation". The format of the enabling code is:

```
$lm_monitor_vec_map(<verilog_register>, “<instance_path_to_m4ke_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_m4ke_vmc_model>”, “RF1”);
   $lm_monitor_vec_map(RF2, “<instance_path_to_m4ke_vmc_model>”, “RF2”);
```

```
...
end
```

8.1.6 VMC Simulation Configuration

The VMC model is configurable so that all functionally visible features of the 4KE core are visible. The available options are shown in Table 8-2 and include processor type (4KEc, 4KEm, or 4KEp core), cache organization, 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 8-2.

Table 8-2 VMC Configuration Options

Name	Addr (hex)	Description	Legal Values	Default
ICacheAssoc	1	Associativity of the instruction cache.	1,2,3,4	2
ICacheWaySize	2	Size of each way of instruction cache (in KB).	0(no I$), 1, 2, 4	4
DCacheAssoc	3	Associativity of the data cache.	1,2,3,4	2
DCacheWaySize	4	Size of each way of data cache (in KB).	0(no D$), 1, 2, 4	4
InitCaches	5	Magically flush caches at time 0 to avoid simulation cycles for software cache initialization.	0 - No Magic Init 1 - Magic Init	1
BATMMU	6	Use Fixed Block Address Translation instead of TLB.	0 - Use TLB (4KEc core) 1 - Use Fixed MMU (4KEm core / 4KEp core)	0
LITEMDU	7	Choose multiply/divide unit (MDU) type.	0 - Fast, high-performance MDU (4KEc core/4KEm core) 1 - Small, iterative MDU (4KEp core)	0
EJSModule	8	Which EJTAG simple break module should be used.	0 - No SB 1 - 2I/1D SB 2 - 4I/2D SB	2
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

Table 8-2 VMC Configuration Options (Continued)

Name	Addr (hex)	Description	Legal Values	Default
dumpTrace	E	Enables instruction trace to file `vmc[.Inst].trace`.	0 - No tracing 1 - Trace file will be created	0
M16 PreWSModule	F	Use MIPS16 decode before the way-select mux (cannot be set when option h'10 is also set).	0 - No MIPS 16 decode 1 - MIPS 16 decode included	0
M16 PostWSModule	10	Use MIPS16 decode after way-select mux (cannot be set when option h'F is also set).		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
I-side ScratchPad	15	Indicates that an I-side ScratchPad is present. The scratchpad logic is customer defined and not included in the VMC model. This bit only affects the setting of a config bit in a Cop0 register (*Config3.ISP*)	0 - No I-side ScratchPad 1 - I-side ScratchPad	0
D-side ScratchPad	16	Indicates that an D-side ScratchPad is present. The scratchpad logic is customer defined and not included in the VMC model. This bit only affects the setting of a config bit in a Cop0 register (*Config3.DSP*)	0 - No D-side ScratchPad 1 - D-side ScratchPad	0
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
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

Table 8-2 VMC Configuration Options (Continued)

Name	Addr (hex)	Description	Legal Values	Default
ICacheMemoryBist	1D	Selects whether the I-side Integrated Memory BIST is enabled in the VMC model. Only one version of MemBIST module is implemented, for the March C+ algorithm.	0 - No I-side Memory BIST 1 - I-side Memory BIST	1
DCacheMemoryBist	1E	Selects whether the D-side Integrated Memory BIST is enabled in the VMC model. Only one version of MemBIST module is implemented, for the March C+ algorithm.	0 - No D-side Memory BIST 1 - D-side Memory BIST	1
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
Watch Registers	22	Selects the number of watch channels that exist within the core	0-8: N pairs of watch registers	1
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

#DCacheWaySize
4/2;
#ICacheWaySize
2/4;
#LITEMDU
7/0;
#BATMMU
6/0;
#EJSModule
8/2;
#EJTModule
9/1;
#DCacheAssoc
3/4;
#ICacheAssoc
1/4;
#InitCaches
5/0;
#Inst
A/0;
#dispEn
B/1;
#haltIt
C/1;
#bus_trace
D/1;
#dumpTrace
E/1;
#M16PreWS Module
F/1;
```

```
#M16PostWS Module
10/0;
#Cop2 Interface Module
11/1;
```

8.1.7 Trace Files

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

8.1.7.1 EC interface Trace

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

- Idle: Indicates how many idle cycles immediately preceded this transaction on the bus. For bursted transactions, the value for the first beat of the burst is used for all beats of the burst.
- Pipe: Indicates the pipeline depth - how many transactions were outstanding when this transaction started. Again, all beats of a burst reflect the value for the first beat of the burst.
- Type: Transaction type: RI- Instruction read, RD- Data read, W- Data write.
- Beat: Indicates which beat of the burst this is and the total length of the burst. “1 of 1x” indicates a non-bursted transaction. “3 of 4” indicates the 3rd beat of a 4 beat burst.
- EB_A<35:0>: Address value.
- EB_R/WData<31:0>: Read or Write data. The value in parentheses is the valid mask. A zero in any bit position indicates that there was an x in the corresponding bit of the data.
- BE<3:0>: Byte Enables - indicates which byte lanes are active for this transaction.
- Error: Indicates whether or not a bus error was signalled on this transaction.
- A wait states: Indicates the number of address wait states seen by this transaction.
- D wait states: Indicates the number of data wait states seen by this transaction.
- Cycle: Indicates a cycle number when this transaction completed. (Cycles are counted from the falling edge of the first Cold Reset). For bursts, all beats of the burst report the cycle that the burst completed.
-

8.1.7.2 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 4KE cores, only two of the eight CCA values are truly supported. When simulating a 4KE core, the entire CCA is not maintained in the ITLB, so the CCA for mapped instruction addresses will always be reported as 2 (uncacheable) or 3 (cacheable).

c) This field is the instruction opcode and disassembly.

```
[Ins:954 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.

```
[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 missed in the cache and went to memory.
- Cache Read indicates a load that hit in the cache.
- SPRam Read indicates a load that hit in the scratchpad RAM
- Probe Read indicates a load that went to DRSEG in EJTAG space
- Mem Alloc Write indicates a store that missed in the cache and caused a cache fill
- Mem AllocH Write indicates a store that missed in the cache, but matched in the store buffer or write through buffer.
- Mem Write indicates a store that either hit in the cache or did not cause a cache allocation
- SPRam Write indicates a store that hit in the scratchpad RAM
- 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.

```
[Ins:187 0 Cyc:1978 ]Write TLB Entry[15]: PageMask(mask) = 00000000(ffffffff)
[Ins:187 0 Cyc:1978 ]      TLB Entry[15]: EnHi(mask)     = 80000000(ffffffff)
[Ins:187 0 Cyc:1978 ]      TLB Entry[15]: EnLo1(mask)    = 00000000(ffffffff)
[Ins:187 0 Cyc:1978 ]      TLB Entry[15]: EnLo0(mask)    = 00000000(ffffffff)
|<--------a-------->|<------------------------i------------------------->|
```

i) A TLB write is shown on multiple lines indicating the TLB entry number and the source registers for data being written into the entry.

8.1.8 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 4KE 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 8.1.3, "SWIFT Template Generation" for details on how to create a template. There are two sample templates in the `verification` directory: `m4ke_vmc_model.vcs.v` is a template for vcs, and `m4ke_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.

8.1.9 Multiple VMC Instances

It is possible to instantiate multiple 4KE 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 8.1.6, "VMC Simulation Configuration" on page 90.

```
m4ke_vmc_model vmc1 (....);
defparam vmc1.InstanceName = “vmc1”;
defparam vmc1.MemoryFile = “memory1”

m4ke_vmc_model vmc2 (...);
defparam vmc2.InstanceName = “vmc2”;
defparam vmc2.MemoryFile = “memory2”;
```

8.1.10 Assertion Checks

A variety of assertion checks are embedded within the 4KE 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 8.1.6, "VMC Simulation Configuration" on page 90 for more details.

Clocking, Reset and Power

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

This chapter contains the following sections:

9.1 Clocking

There are potentially two input clocks that must be generated and driven to a 4KE core. The main clock input is named *SI_ClkIn*, and exists on every 4KE 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 4KE 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 4KE 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 4KE core that is being integrated before proceeding with system clock design.

9.1.1 *SI_ClkIn* Clock

SI_ClkIn is the primary 1x input clock to the 4KE 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 4KE 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 4KE core implementation.

9.1.2 *EJ_TCK* Clock

EJ_TCK is an optional 1x clock input to the 4KE 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.

9.1.3 Handling Clock Insertion Delay

When a 4KE 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 4KE core implementation.

The clock insertion delay presents an issue that must be managed when the 4KE 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 4KE 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 4KE 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 4KE 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 4KE 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 4KE 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 4KE core. The difference between *SI_ClkIn* and *SI_ClkOut* represents the clock insertion delay of the primary clock used within the 4KE 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.

9.2 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 4KE 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 4KE core, and not left floating or indeterminate. Each of the reset-related *SI_** inputs triggers a different type of exception within the 4KE core; the *MIPS32 4KE™ Processor Core Family Software User's Manual* [1] describes more details about these exceptions.

The initialization process for a 4KE 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 4KE core.

9.2.1 SI_ColdReset

The high-active *SI_ColdReset* input is a hard reset signal that initializes the internal hardware state of the 4KE 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.

9.2.2 SI_Reset

The high-active *SI_Reset* input is a warm reset input to the 4KE 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 4KE 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.

9.2.3 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.

9.2.4 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.

9.3 Power Management

Two primary mechanisms exist for managing system power with a 4KE 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.

9.3.1 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 4KE core is not in use, if desired by your system logic. The 4KE 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 4KE core in a low-power state by reducing the clock frequency. When the RP bit in the Status

register is set by software, system logic can detect the assertion of the *SI_RP* output and choose to place the 4KE 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 4KE core is not being used.

9.3.2 Software-Induced Sleep Mode

Upon execution of the software WAIT instruction, the 4KE core will enter a low-power state once all outstanding bus activity has completed. Most of the clocks in the 4KE 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 4KE 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 4KE core could be placed into a low power state as well.

Design For Test Features

This chapter describes the Design For Test (DFT) features of the MIPS32™ 4KE™ 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 10.1, "Introduction"
- Section 10.2, "Scan Test"
- Section 10.3, "Integrated RAM BIST"
- Section 10.4, "User-Specific RAM BIST"

10.1 Introduction

An implementation of a 4KE 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 integrated controllers
- Memory BIST using a user-specified method
- Other implementation-dependent features

Table 10-1 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.

Table 10-1 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
gscanramwr	0	assert during capture cycles for RAM strobe control	0	0
gmbinvoke	0	0	1	0
BistIn[n:0]	0	0	0	impl-dependent

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

10.2 Scan Test

The scan methodology normally used on a 4KE 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: *gscanramwr, gscanmode* and *gscanenable*. If the scan insertion scripts for Mentor DFTAdvisor, provided with a soft 4KE 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 10-1. 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. The timing of *gscanramwr* is not shown in the figure, but it must be stable around the capture cycle(s) and can be used to control the read and write strobes for cache arrays, if the SRAMs are handled as a bypass flop during scan mode.

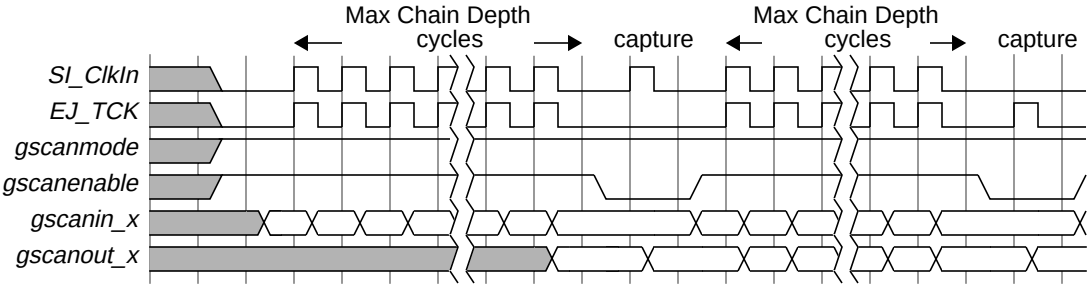


Figure 10-1 Timing Diagram of Typical Scan Chain and Capture Operation

10.3 Integrated RAM BIST

The 4KE core may optionally include an integrated BIST controller to test the cache SRAMs within the core. Some signals present on the core interface, prefixed by *gmb*, are specifically dedicated to integrated RAM BIST. These signals are always present on the core, but whether they are active or not is implementation-dependent. In addition to the *gmb** signals, some other signals are also active when using integrated RAM BIST.

The integrated BIST controller is capable of supporting two algorithms, March C+ or IFA-13. (IFA-13 includes support for retention testing.) The algorithm present (if any) is a build-time option chosen when the core is created.

10.3.1 RAM BIST-related Interface Signals

This section describes the relevant core interface signals for launching an integrated BIST test and reporting the results.

10.3.1.1 Clocking

The clock for integrated memory BIST is provided by the primary clock input, *SI_ClkIn*. *SI_ClkIn* must be running while the *gmbinvoke* and *SI_ColdReset* signals are asserted and for at least the first cycle after *gmbinvoke* is deasserted, in order to perform and complete a BIST test. The *EJ_TCK* input clock is unused for integrated BIST and may be driven to any value.

10.3.1.2 Reset

SI_ColdReset must be asserted while integrated memory BIST is running. This forces the main clock tree derived from *SI_ClkIn* to be running, since it could have been disabled by WAIT-induced sleep mode or unknown at power up. *SI_ColdReset* should be asserted at least 5 *SI_ClkIn* cycles prior to the assertion of *gmbinvoke*, and held asserted for at least one cycle after the deassertion of *gmbinvoke*.

10.3.1.3 Invoke

The primary enable signal to activate integrated memory BIST is *gmbinvoke*. The *gmbinvoke* signal should only be asserted while *SI_ColdReset* is also asserted. After BIST testing is completed and *gmbinvoke* is deasserted, a normal *SI_ColdReset* sequence should be applied to reset the processor for non-BIST operation.

10.3.1.4 Done Indication

When the BIST test is completed, *gmbdone* is asserted. If the memory BIST test is performed for both I-cache and D-cache, *gmbdone* is asserted only when both tests are done. When *gmbinvoke* is deasserted, *gmbdone* is deasserted in the following cycle.

10.3.1.5 Fail Indication

Separate fail signals exist for each sub-array in both the instruction and data caches. If a failure occurs during the test, a fail signal is asserted accordingly: *gmbddfail*, *gmbtdfail*, *gmbwdfail*, *gmbdifail*, *gmbtifail* and/or *gmbwifail*. The fail signals are related to specific cache arrays as shown in Table 10-2. When *gmbinvoke* is deasserted, all fail signals are deasserted in the following cycle.

Table 10-2 Fail Signals

Fail Signals	Instruction Cache			Data Cache		
	Data Memory	**Tag Memory**	**Way-Select Memory**	**Data Memory**	**Tag Memory**	**Way-Select Memory**
gmbdifail	X					
gmbtifail		X				
gmbwifail			X			
gmbddfail				X		
gmbtdfail					X	
gmbwdfail						X

10.3.1.6 gscanenable

The *gscanable* signal enables the scan chain operation. When memory BIST test is running, *gscanenable* must be deasserted low.

10.3.1.7 gscanmode

The *gscanmode* signal enables scan test mode. When memory BIST test is running, *gscanmode* must be deasserted low.

10.3.2 RAM BIST Signal Waveform for a Memory Test

A diagram with the timing of an integrated memory BIST sequence is shown in Figure 10-2.

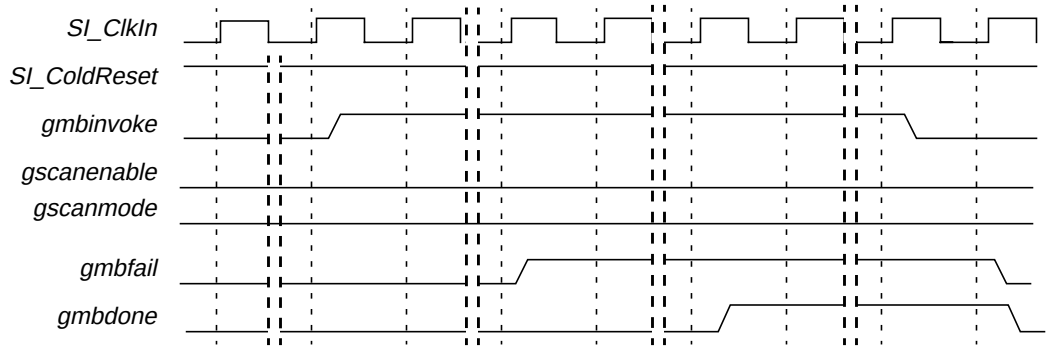


Figure 10-2 RAM BIST I/O Signals Timing

10.3.3 Number of Cycles for Memory BIST

The number of cycles for integrated memory BIST is determined by:

$$\text{Cycles(WithoutSPRAM)} = \text{WaySize(kBytes)} \times (1024 \times 8)\left(\frac{\text{bit}}{\text{kByte}}\right)\left(\frac{\text{cycle}}{\text{bit}}\right) \times \text{Associativity} \times \text{NumofOperations} + 32\text{cycles}$$

$$\text{Cycles(WithSPRAM)} = \text{WaySize(kBytes)} \times (1024 \times 8)\left(\frac{\text{bits}}{\text{kByte}}\right)\left(\frac{\text{cycle}}{\text{bit}}\right) \times \text{MaxAssociativity} \times \text{NumberofOperations} + \text{SPRAMSize(kBytes)} \times (1024 \times 8)\left(\frac{\text{bit}}{\text{kBytes}}\right)\left(\frac{\text{cycle}}{\text{bit}}\right) + 32\text{cycles}$$

For the March C+ algorithm, NumberofOperations per bit is 14. For the IFA-13 algorithm, NumberofOperations per bit is 16.

10.4 User-Specific RAM BIST

User-specific RAM BIST utilizes the top-level *BistIn* and *BistOut* buses to test the cache or on-chip trace SRAM arrays. 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.

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 4KE soft or hard core release, or be available on the MIPS web site, under **http://www.mips.com/publications/index.html**.

1. MIPS32™ 4KE™ Processor Core Family Software User's Manual
   MIPS document: MD00103
2. EC™ Interface Specification
   MIPS document: MD00052
3. EJTAG™ Specification
   MIPS document: MD00047
4. EJTAG Trace Control Block Specification
   MIPS document: MD00148
5. Core Coprocessor Interface Specification
   MIPS document: MD00068
6. MIPS32™ Architecture For Programmers Volume III: The MIPS32™ Privileged Resource Architecture
   MIPS document: MD00090
7. MIPS64™ 5Kc™ Processor Core Software User's Manual
   MIPS document: MD00012

Revision History

Table B-1 Revision History

Revision	Date	Description
00.90	Nov 28, 2000	• Initial revision.
00.91	February 16, 2001	• Added description about running `swiftcheck` to verify the VMC installation. • Reworded some of the explanations about never sending a nullify to any coprocessor instructions. • Changed *MIPS4KHOME* environment variable to *M4KHOME*. • Added *SI_SimpleBE[1:0]* signals to control limited byte enable combinations on the EC interface. • Removed sysad option from *SI_MergeMode[1:0]*.
01.00	March 27, 2001	• Standardized section links at the beginning of each chapter. • Minor updates to SimpleBE description. • Removed *CP_fr32_0* signal, this is a CP1 pin only. • Added memory bits for the VMC module configuration for MIPS16 and CP2 IF modules.
01.01	May 17, 2001	• Added description of interface pins related to integrated memory BIST feature in Table 2-3. • Modified section on coprocessor exception signaling for clarity.
01.02	June 12, 2001	• Added Chapter on DFT features. • Cleaned up some pin descriptions.
01.03	July 13, 2001	• Minor grammar updates.
01.04	August 29, 2001	• Reworded Section 4.1.2.2, "Multiplexed Pin Access" on page 37, to not claim violation of EJTAG spec. as this was not true. • Added EJTAG Trace information: TCtrace core <-> PIB pins to pinlist, and information in Section 4.4, "EJTAG Trace" on page 42. • Added the *gscanramwr* pin to the pinlist. • Added reference appendix. • Included new options for VMC model. • Added availability of VMC model on x86 Linux platform. • Added more details to DFT chapter. • Noted that interrupt pins are level sensitive and not prioritized by HW to answer a F.A.Q.
01.05	October 4, 2001	• Changed confidentiality level for 4KE document to “commercial”; no functional changes.

Table B-1 Revision History

Revision	Date	Description
01.06	December 5, 2001	• Added Mem-BIST and global clock gate VMC switches to Table 8-2. • Fixed documentation errors with VMC option numbering in Table 8-2. • Further description of EB_EWBE signaling. • Added descriptions for external SPRAM signals in Chapter 2, “Signal Description,” on page 3. • Added new chapter describing external SPRAM interface in Chapter 6, “Scratchpad RAM Interface,” on page 59.
01.07	December 10, 2001	• Clarified that SPRAM signals are only held during a stall if a transaction is really outstanding, in Table 6-1. • Corrected error in *ISP_DataTagValue* pin name. Modified references from *ISP_DataWrValue* to *ISP_DataTagValue*, to reflect proper pin name. • Corrected some hyperlinks in Chapter 6, “Scratchpad RAM Interface,” on page 59, which were not showing up as blue in the pdf.
01.08	January 25, 2002	• Added new chapter describing performance monitor pins, Chapter 7, “Performance Monitoring Interface,” on page 83. • Removed references to the filenames of related pdf documents, since those filenames are now explicitly identified by the MIPS MDxxxxx document number.
02.00	November 8, 2002	• Added legal footer to Chapter 7, “Performance Monitoring Interface,” on page 83. • Added *CP2_kd_mode_0* pin. • Clarified description of fastest EC write transaction, in Section 3.1.3, "Fastest Write Transaction". • Corrected use of “maximum” and “minimum” in description of *TC_CRMax* and *TC_CRMin* signals in Table 2-3 on page 4. • Modified wording in Section 9.1.3, "Handling Clock Insertion Delay", since not all SPRAM interface signals are fully registered. • Updated Simulation Models chapter with more recent VMC information • Various updates to describe new MIPS32 Release 2 capabilities, included in version 3.0 or higher core releases.