MIPS
by Imagination

MIPS32® microAptiv™ UP Processor Core Family Integrator's Guide

Document Number: MD00941
Revision 01.01
July 30, 2014

Imagination Technologies Ltd.
955 East Arques Avenue
Sunnyvale, CA 94085-4521

microMIPS™ MIPS Verified™ Aptiv™

Template: nDb1.03, Built with tags: 1D

Table of Contents

List of Figures

List of Tables

Chapter 1

Overview

This document is targeted for the ASIC designer who is integrating a version of the MIPS32® microAptiv™ UP processor core into the system ASIC. This document is applicable both to 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 Descriptions" on page 11 describes the pins of the core.
- Chapter 3, "AHB-Lite Interface" on page 29 describes the AHB-Lite interface protocol used by the core.
- Chapter 4, "Interrupt Interface" on page 39 describes the signalling in different interrupt modes.
- Chapter 5, "EJTAG Interface" on page 45 discusses the EJTAG interface used by the core, including the optional EJTAG TAP controller and the trace interface.
- Chapter 6, "Coprocessor Interface" on page 57 describes the Coprocessor 2 interface and protocol used by the core.
- Chapter 7, "Scratchpad RAM Interface" on page 75 describes the Scratchpad RAM interface that may optionally be present on the core.
- Chapter 8, "Clocking, Reset, and Power" on page 99 covers issues related to handling the clock insertion delay of the microAptiv UP core. Additionally, the hardware reset requirements of the core, as well as power management techniques, are discussed.
- Chapter 9, "Design For Test Features" on page 107 discusses general DFT features which may be present on the microAptiv UP 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 `MIPS_HOME`, `MIPS_CORE` and `MIPS_PROJECT` environment variables. See the "Release Deliverables and Installation" chapter of the System Package & Simulation Flow User's Manual [5] for more information on defining required environment variables.

1.1.1 microAptiv UP Deliverables

All of the microAptiv UP deliverables packages include the following:

- Cycle-exact model: An encrypted, cycle-exact version of the RTL model is generated using VMC from Synopsys and included in every release. This model is intended mainly for hard-core customers who do not receive the source RTL, but is available for soft-core customers if desired. The VMC model is also used within the supplied verification testbench. Use of the cycle-accurate model is currently limited to x86 RedHat Linux platforms.

- Functional simulation: The testbench code is written in Verilog. The simulation environment includes support for the following Verilog simulators: NC-Verilog from Cadence, VCS from Synopsys, and ModelSim from Mentor Graphics.

Soft core deliverables packages will also include support for the following

- RTL: The core RTL code is written in Verilog. The simulation environment includes support for the Verilog simulators listed above. Simulation is supported at both the RTL and gate levels.

- Implementation scripts: synthesis, timing analysis, power analysis, scan insertion, ATPG, equivalence checking, physical design. The MIPS® Physical Design Guide [1]describes the scripts as well as the tools and versions that are supported.

No tool requirements are dictated for the back-end EDA tools that may be used to create a physical implementation of the microAptiv UP core.

1.2 Other Documents

Other documents available from MIPS cover additional aspects of an microAptiv UP core, including the software view of the core, programming guidelines, and general details about certain sub-interfaces. If these other documents are referenced within this *Integrator's Guide*, they are listed in Appendix A, "References" on page 113.

Chapter 2

Signal Descriptions

This chapter describes the signals on a MIPS32 microAptiv UP 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 "Top-level Hierarchy"
- Section 2.3 "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 never change state.

The names of interface signals present on an microAptiv UP core are prefixed with a unique string, according to their primary function. Table 2.2 defines the prefixes used for microAptiv UP core interface signals.

Table 2.2 Signal Prefix Key

Prefix	Description
*H_*	Signals directly related to the AHB-Lite interface.
*SI_*	General system interface signals, which are not part of theAHB-Lite interface.
*EJ_*	Signals related to the EJTAG interface.
*TC_*	Signals related to the EJTAG Trace interface.
*CP2_*	Signals related to the Coprocessor 2 interface.

Table 2.2 Signal Prefix Key (Continued)

Prefix	Description
*UDI_*	Signals related to the CorExtend user-defined instruction interface (Pro Series™ cores only).
*{I,D}SP_*	Instruction/Data ScratchPad RAM interfaces
*PM_*	Performance monitoring signals
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 Top-level Hierarchy

An microAptiv UP processor core has two options for the top-level module when the core is implemented. The choice of top-level module depends on implementation trade-offs when the core is synthesized or hardened, and the choice affects the top-level pinout visible when the core is integrated into the chip.

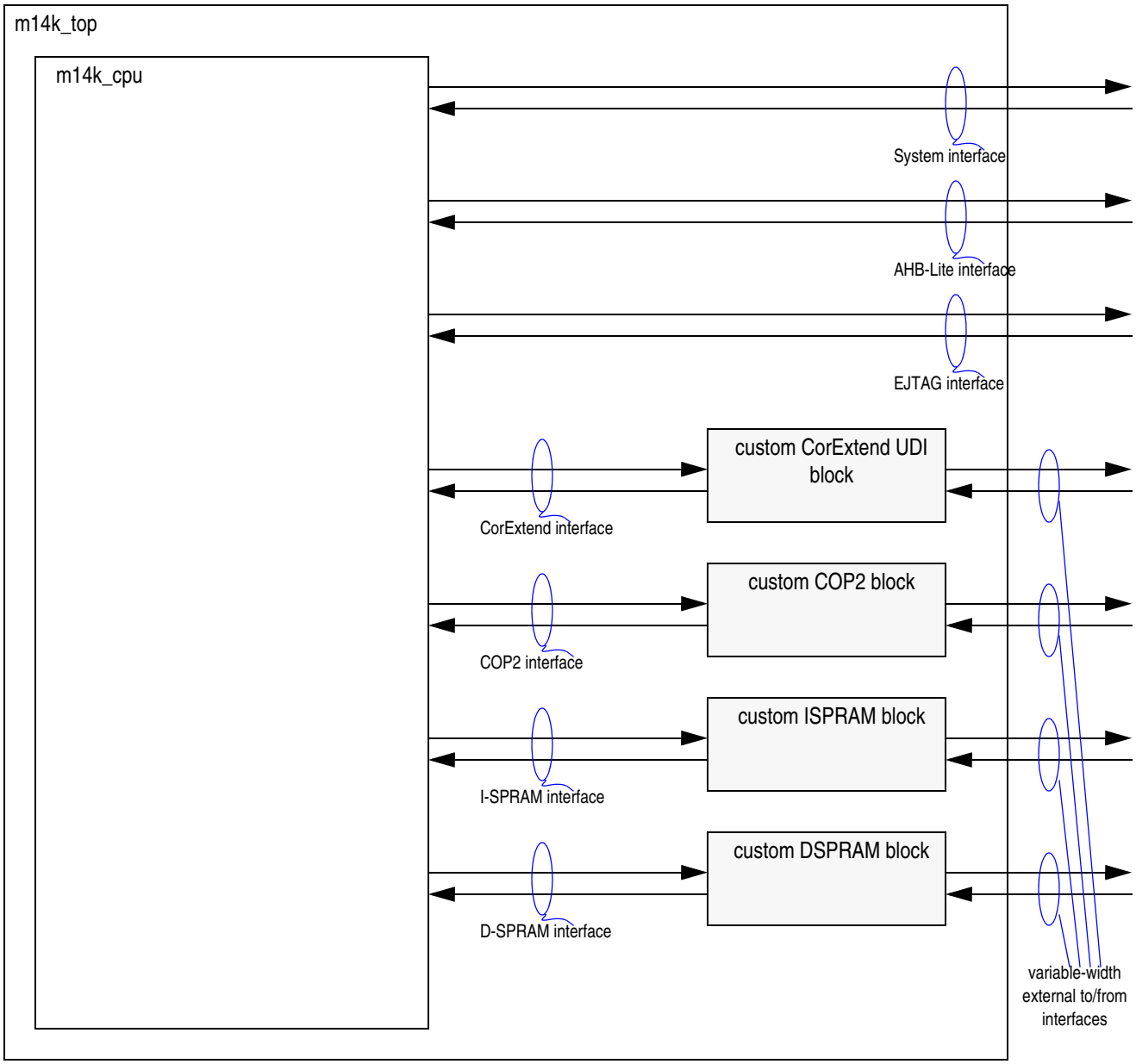
The top-level hierarchy is shown in Figure 2.1. Either module `m14k_cpu` or module `m14k_top` can be chosen as the top-level module at build time. Module `m14k_cpu` represents the direct processor core. Module `m14k_top` encapsulates `m14k_cpu` as well as some user-definable modules that interact tightly with `m14k_cpu` and can be customized by a user.

TheAHB-Lite, system, EJTAG, trace and testability interfaces exist on both the `m14k_cpu` and `m14k_top` levels, with the expectation that any logic interacting with these interfaces lies in system logic above `m14k_top`. Thes-cratchpad RAM, Coprocessor 2 and CorExtend interfaces are present on the `m14k_cpu` level only. These interfaces are closely coupled to the processor pipeline. Custom logic connected to these interfaces can be combined with the processor core itself at synthesis time. In this case, synthesis can be performed at the `m14k_top` level to minimize the need for defining detailed constraints between the CPU and the custom logic, and to allow synthesis to better optimize the interfaces. External system logic can still interact with the custom blocks via the configurable width to/from busses shown in Figure 2.1, with no direct changes needed to the `m14k_top` module RTL.

In some designs, however, the custom logic may not be known when the core is built and will be added later. For this situation, synthesis at the `m14k_cpu` level is appropriate. The custom interfaces need to be constrained for their expected use, but custom logic can be added later.

When integrating a previously hardened microAptiv UP core, consult with the provider of the core to determine whether it was built with `m14k_top` or `m14k_cpu` as the top level.

Figure 2.1 Top-level RTL Hierarchy



2.3 Detailed Signal Descriptions

All core signals at the `m14k_cpu` level are listed in Table 2.3 below. The following table, Table 2.4, lists the variable width to/from signals on the `m14k_top` level that allow external access to internal custom blocks for thescratchpad RAM, Coprocessor 2 and CorExtend interfaces.

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.

2.3.1 Signals at m14k_cpu Level

Table 2.3 describes the signals at the m14k_cpu level of hierarchy.

Table 2.3 Signal Descriptions for m14k_cpu Level

Signal Name	Type	Description
System Interface: Refer to Chapter 8, "Clocking, Reset, and Power" on page 99 for more details		
Clock Signals: Refer to 8.1 "Clocking" on page 99 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 free running 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 8.4 "Power Management" on page 104 for a description of the various types of reset.		
SI_BootExcISAMode	AIn	When set to '0', boot up from MIPS32 mode or, set to '1' to boot in microMIPS mode
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 8.4 "Power Management" on page 104 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_NESTERL	Out	This signal reflects the state of the ERL bit (2) in the CP0 *NestedExc* register.
SI_NESTEXL	Out	This signal reflects the state of the EXL bit (1) in the CP0 *NestedExc* register.
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.
Break Status Signals:		
SI_Ibs[7:0]	Out	Reflects state of breakpoint status (BS) field in the Instruction Breakpoint Status (IBS) register. These bits are set when the corresponding break condition has matched, for breaks enabled as either a breakpoints or trigger points. If fewer than 6 instruction breakpoints exist, the unimplemented bits are tied to 0.
SI_Dbs[3:0]	Out	Reflects state of breakpoint status (BS) field in the Data Breakpoint Status (DBS) register. These bits are set when the corresponding break condition has matched, for breaks enabled as either a breakpoints or trigger points. If fewer than 2 data breakpoints exist, the unimplemented bits are tied to 0.
Interrupt Signals:		

Table 2.3 Signal Descriptions for m14k_cpu Level (Continued)

Signal Name	Type	Description
SI_EICPresent	SIn	Indicates whether an external interrupt controller is present. Value is visible to software in the $Config3_{VEIC}$ register field.
SI_EICVector[5:0]	In	Provides the vector number for an interrupt request in External Interrupt Controller (EIC) mode. (Note: This input decouples the interrupt priority from the vector offset. For compatibility with earlier Release 2 cores in EIC mode, connect *SI_Int[7:0]* and *SI_EICVector[5:0]* together.)
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[7:0]* pins into the $Cause_{RIPL}$ field (overlaid with $Cause_{IP9..IP2}$), 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[7:0]	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 8 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 an 8-bit encoded value in the range of 0..255. • When the core starts the interrupt exception, signaled by the assertion of *SI_IAck*, it loads the value of the *SI_Int[7:0]* pins into the $Cause_{RIPL}$ field (overlaid with $Cause_{IP9..IP2}$). The interrupt controller can then signal another interrupt.
SI_ION[17:1]	Out	Interrupt Offset Number. Indicates the current interrupt offset number that is being serviced. The offset number was captured from *SI_Offset[17:1]* when *SI_Int* was asserted to request an interrupt exception. Depending on the configuration of the EIC, *SI_ION[17:1]* may be updated when *SI_IAck* is asserted, .
SI_IPL[7:0]	Out	Current interrupt priority level from the $Cause_{IPL}$ register field, provided for use by an external interrupt controller. This value is updated whenever *SI_IAck* is asserted.

Table 2.3 Signal Descriptions for m14k_cpu Level (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></tr>
<tr><td>SI_IPTI[2:0]</td><td>SIn</td><td>This input indicates which IP number the timer interrupt is combined with in the core. The value of this bus is visible to software in the $IntCtl_{IPTI}$ register field.

<table><tr><th>SI_IPTI</th><th>Combined w/ SI_Int</th></tr><tr><td>0-1</td><td>None</td></tr><tr><td>2</td><td>SI_Int[0]</td></tr><tr><td>3</td><td>SI_Int[1]</td></tr><tr><td>4</td><td>SI_Int[2]</td></tr><tr><td>5</td><td>SI_int[3]</td></tr><tr><td>6</td><td>SI_Int[4]</td></tr><tr><td>7</td><td>SI_Int[5]</td></tr></table></td></tr>
<tr><td>SI_IVN[5:0]</td><td>Out</td><td>Interrupt Vector Number is to indicate the current interrupt vector number that is being serviced. This vector number was captured from SI_EICVector[5:0] when SI_Int is asserted to request for an interrupt exception. Depends on the EIC configuration, SI_IVN[5:0] is updated when SI_IAck is asserted.</td></tr>
<tr><td>SI_NMITaken</td><td>Out</td><td>NMI Taken reflects the value of CP0 register STATUS.NMITaken.</td></tr>
<tr><td>SI_Offset[17:1]</td><td>In</td><td>Offset for interrupt vector</td></tr>
<tr><td>SI_SWInt[1:0]</td><td>Out</td><td>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.</td></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:
Traditionally, SI_TimerInt is fed back into the core through one of the interrupt pins. However, this is no longer needed, as the core will internally route the interrupt to the IP number set by the IntCtl.IPTI field.

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/Outputs:</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>
</table>

Table 2.3 Signal Descriptions for m14k_cpu Level (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></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_SRSDisable[3:0]</td><td>SIn</td><td>Disable use of some shadow register sets.
<table>
<tr><th>SI_SRSDisable[3:0]</th><th>Register Sets</th></tr>
<tr><td>0000</td><td>Use all register sets</td></tr>
<tr><td>1000</td><td>Use 8 register sets</td></tr>
<tr><td>1100</td><td>Use 4 register sets</td></tr>
<tr><td>1110</td><td>Use 2 register sets</td></tr>
<tr><td>1111</td><td>Use 1 register set</td></tr>
</table></td></tr>
<tr><td>SI_TraceDisable</td><td>SIn</td><td>Set to ‘1’ to disable the trace hardware.</td></tr>
<tr><td colspan="3">Fast Debug Channel:</td></tr>
<tr><td>SI_IPFDCI[2:0]</td><td>In</td><td>This input indicates which IP number the FDC interrupt is combined with internally in the core.</td></tr>
<tr><td>SI_FDCInt</td><td>Out</td><td>FDC interrupt indication. This signal indicates RX FIFO full or TX FIFO or probe interrupt. Probe interrupt only enabled when RX int is available.
For External Interrupt Controller (EIC) mode:
The SI_FDCInt signal is provided to the external interrupt controller, which then prioritizes the FDC 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_IPFDCI pins are not meaningful in EIC mode.</td></tr>
<tr><td colspan="3">Performance Counters:</td></tr>
<tr><td>SI_IPPCI[2:0]</td><td>In</td><td>This input indicates which IP number the PCI interrupt is combined with internally in the core.</td></tr>
<tr><td>SI_PCInt</td><td>Out</td><td>PC interrupt indication.
For External Interrupt Controller (EIC) mode:
The SI_PCInt signal is provided to the external interrupt controller, which then prioritizes the performance counter 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_IPPCI pins are not meaningful in EIC mode.</td></tr>
<tr><td colspan="3">AHB-Lite Interface Refer to Chapter 3, “AHB-Lite Interface” on page 29 for more details.</td></tr>
<tr><td>HADDR[31:0]</td><td>Out</td><td>The 32-bit system address bus</td></tr>
<tr><td>HBURST[2:0]</td><td>Out</td><td>The burst type indicates if the transfer is a single transfer or forms part of a burst. Fixed length bursts of 4, 8, and 16 beats are spec’ed but not all are supported. The burst can be incrementing or wrapping. Only Single or WRAP4 are supported in the microAptiv UP core.</td></tr>
</table>

Table 2.3 Signal Descriptions for m14k_cpu Level (Continued)

Signal Name	Type	Description
HCLK	Out	The bus clock times all bus transfers. All signal timings are related to the rising edge of *HCLK*. It is a reference clock from the gated main clock *SI_ClkIn*.
HMASTLOCK	Out	When HIGH, this signal indicates that the current transfer is part of a locked sequence. It has the same timing as the address and control signals. In the microAptiv UP core when atomic instruction access uncached space through AHB-Lite, assert *HMASTLOCK* until the atomic write transaction is broadcast on the AHB-Lite bus. Typically the locked transfer is used to maintain the integrity of a semaphore.
HPROT[3:0]	Out	The protection control signals provide additional information about a bus access and are primarily intended for use by any module that wants to implement some level of protection. *HPROT[3:1]* is set default to b001 because the microAptiv UP core can not provide the accuracy of all protection information. But *HPROT[0]* is used to distinguish between Opcode fetch and Data access. *HPROT*=4'b0010 for instruction fetches; 4'b0011 for data loads and stores
HRDATA[31:0]	In	Read Data
HREADY	In	When HIGH, the HREADY signal indicates that a transfer has finished on the bus. This signal can be driven LOW to extend a transfer.
HRESETn	Out	The bus reset signal is active LOW and resets the system and the bus. This is the only active LOW AHB-Lite signal.
HRESP	In	The transfer response. When LOW, the HRESP signal indicates that the transfer status is OKAY. When HIGH, the HRESP signal indicates that the transfer status is ERROR.
HSIZE[2:0]	Out	Indicates the size of the transfer, that is typically byte, halfword, or word. The protocol allows for larger transfer sizes up to a maximum of 1024 bits.
HTRANS[1:0]	Out	Indicates the transfer type of the current transfer. This can be: • IDLE • BUSY—Not implemented in M14Kc • NONSEQUENTIAL • SEQUENTIAL
HWDATA[31:0]	Out	The write data bus transfers data from the master to the slaves during write operations.
HWRITE	Out	Indicates the transfer direction. When HIGH this signal indicates a write transfer and when LOW a read transfer.
SI_AHBStb	In	Enable AHB input and output interface signals to be registered and to suppress HCLK high pulses so that the AHB bus can run at a lower clock ratio than the CPU core clock. This signal is registered by the core prior to use.
CorExtend™ Interface: On Pro Series™ cores, there is an interface to an external CorExtend user-defined instruction block. Refer to the *MIPS32® Pro Series™ CorExtend™ Instruction Integrator's Guide* [12] for more details on these signals.		
EJTAG Interface: Refer to Chapter 5, "EJTAG Interface" on page 45 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.

Table 2.3 Signal Descriptions for m14k_cpu Level (Continued)

Signal Name	Type	Description
EJ_TDI	In	Test Data Input (*TDI*) for the EJTAG TAP.
EJ_TDO	Out	Test Data Output (*TDO*) for the EJTAG TAP. Driven on the negative edge of *EJ_TCK*.
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 microAptiv UP core outputs this signal to drive an external 3-state buffer. Driven on the negative edge of *EJ_TCK*.
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.
EJ_ECREjtagBrk	Out	Output the current state of the *ECREjtagBrk* bit. The probe can set this bit to cause a debug exception. For systems that may shut down the main core clock in sleep mode, this signal allow the clocks to be restarted so the debugger initiated exception can be taken. Driven on the negative edge of *EJ_TCK.*
Debug Mode Control / Indication		
EJ_DisableProbeDebug	In	Must be held constant during all operation modes of the core. When asserted: • ProbEn=0 • ProbTrap=0 • EJTagBrk is disabled • EJTAGBOOT is disabled • PC/DA Sampling is disabled
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 8.4 "Power Management" on page 104 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.		
EJ_ManufID[10:0]	SIn	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.
EJ_PartNumber[15:0]	SIn	Value of the *Device ID*$_{PartNumber}$ register field.
EJ_Version[3:0]	SIn	Value of the *Device ID*$_{Version}$ register field.

Table 2.3 Signal Descriptions for m14k_cpu Level (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></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 connected to the Trace Capture Block (TCB) inside the core. 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_Valid</td><td>Out</td><td>Asserted when a new trace word is started on the TC_Data[63:0] signals.</td></tr>
<tr><td>TC_ClockRatio[2:0]</td><td>Out</td><td>Clock ratio. The table below shows the encoded values for clock ratio.
With the iFlowtrace mechanism, the clock ratio is set via the OfClk bit in the ITCB control/status register and the only possible values are 100 and 101.
<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_Data[63:0]</td><td>Out</td><td>Trace word data. With the iFlowtrace mechanism, the 64b data value is held constant until it is acknowledged by the deassertion of TC_Stall.</td></tr>
<tr><td>TC_Stall</td><td>In</td><td>Stall request. In the iFlowtrace scheme, the deassertion of this signal acknowledges that the previous value on TC_Data has been entirely consumed and the next value can be presented.</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 colspan="3">Coprocessor 2 Interface: Refer to Chapter 6, “Coprocessor Interface” on page 57 for more details.</td></tr>
<tr><td colspan="3">Instruction Dispatch: These signals are used to transfer an instruction for the microAptiv UP core to the COP2 coprocessor.</td></tr>
<tr><td>CP2_ir_0[31:0]</td><td>Out</td><td>Coprocessor Arithmetic and To/From Instruction Word: Valid in the cycle before CP2_as_0, CP2_ts_0 or CP2_fs_0 is asserted.</td></tr>
<tr><td>CP2_irenable_0</td><td>Out</td><td>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].</td></tr>
</table>

Table 2.3 Signal Descriptions for m14k_cpu Level (Continued)

Signal Name	Type	Description
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.
CP2_endian_0	Out	**Big Endian Byte Ordering**: When asserted, the processor is using big endian byte ordering for the dispatched instruction. When deasserted, the processor is using little-endian byte ordering. Valid the cycle before *CP2_as_0*, *CP2_fs_0* or *CP2_ts_0* is asserted.
CP2_inst32_0	SOut	**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 microAptiv UP core is a MIPS32 core, and will only issue MIPS32 instructions. Thus *CP2_inst32_0* is tied high.
CP2_kd_mode_0	Out	**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.
To Coprocessor Data: These signals are used when data is sent from the microAptiv UP core to the COP2 coprocessor, as part of completing a To Coprocessor instruction.		
CP2_tds_0	Out	**Coprocessor To Data Strobe**: Asserted when To COP Op data is available on *CP2_tdata_0[31:0]*.

Table 2.3 Signal Descriptions for m14k_cpu Level (Continued)

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

Table 2.3 Signal Descriptions for m14k_cpu Level (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></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 microAptiv UP 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 microAptiv UP 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>
<tr><td colspan="3">Instruction Nullification: These signals are used by the microAptiv UP 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 microAptiv UP core is signalling that the instruction is not nullified. When asserted, the microAptiv UP 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 microAptiv UP 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>
</table>

Table 2.3 Signal Descriptions for m14k_cpu Level (Continued)

<table>
<tr><th>Signal Name</th><th>Type</th><th>Description</th></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">CorExtend UDI Interface: These signals can be used to interface between CorExtend Block and the core. Refer to MD00324 “MIPS Pro Series CorExtend Instruction Integrator’s Guide” for more details.</td></tr>
<tr><td>UDI_ir_e[31:0]</td><td>Out</td><td>This is the complete instruction word. Although the module also gets rs and rt source operands, the full instruction is provided so all or part of the source register fields may be used to hold immediate values. Note that the implementer is responsible for decoding the Opcode and Function fields.</td></tr>
<tr><td>UDI_irvalid_e</td><td>Out</td><td>Indicates whether the value of the instruction word (UDI_ir_e) is valid or not.</td></tr>
<tr><td>UDI_rs_e[31:0]</td><td>Out</td><td>Source operand rs after the bypass mux.</td></tr>
<tr><td>UDI_rt_e[31:0]</td><td>Out</td><td>Source operand rt after the bypass mux.</td></tr>
<tr><td>UDI_endianb_e</td><td>Out</td><td>Indicates that this instruction is executing in Big Endian mode. This signal is generally not needed unless a) the UDI instruction works on sub-word data that is endian dependent, and b) the UDI block is designed to be bi-endian</td></tr>
<tr><td>UDI_kd_mode_e</td><td>Out</td><td>Indicates that the instruction is executing in kernel or debug mode. This can be used to prevent certain UDI instructions from being executed in user mode.</td></tr>
<tr><td>UDI_kill_m</td><td>Out</td><td>Late arriving kill signal due to an exception generated by an earlier instruction. This signal may optionally be used to deassert the UDI_stall_m output for improved interrupt latency on multi-cycle UDIs whose results won’t be used.</td></tr>
<tr><td>UDI_start_e</td><td>Out</td><td>This is the mpc_run_ie signal coming from the core pipeline control logic.</td></tr>
<tr><td>UDI_run_m</td><td>Out</td><td>This is the mpc_run_m signal used to qualify UDI_kill_m.</td></tr>
<tr><td>UDI_greset</td><td>Out</td><td>Reset signal to be used to reset any state machines.</td></tr>
<tr><td>UDI_gclk</td><td>Out</td><td>Clock input.</td></tr>
<tr><td>UDI_gscanenable</td><td>Out</td><td>Global scan enable.</td></tr>
</table>

Table 2.3 Signal Descriptions for m14k_cpu Level (Continued)

Signal Name	Type	Description
UDI_ri_e	In	A one-bit signal which when high indicates that the SPECIAL2 instruction currently being executed is illegal (i.e., reserved). This signal is used by the Master Pipeline Control (MPC) block within the core to signal an illegal instruction, however, this signal is sampled by MPC only if the current instruction is within the SPECIAL2 range of user-defined instructions (bits [5:4] of the instruction are 2'b01).
UDI_rd_m[31:0]	In	The 32-bit result of the executed instruction available in the M stage.
UDI_wrreg_e[4:0]	In	Register to write the result from the execution of this user-defined instruction. This value is also passed on to mpc.
UDI_stall_m	In	Signals that the UDI block is processing a multicycle instruction and needs to stall the pipeline since the outputs need to be written into the register file. Should be set to 0 for single cycle instructions. This is an M stage signal.
UDI_present	In	Static signal that denotes whether any UDI support is available.
UDI_honor_cee	In	Indicates whether the core should honor the CorExtend Enable (CEE) bit contained in the *Status* register. When this signal is asserted, *Status.CEE* is deasserted, and a UDI operation is attempted, the core will take a CorExtend Unusable Exception.
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 7, "Scratchpad RAM Interface" on page 75 for further details.		
DSP_TagAddr[19:2]	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_Lock	Out	Indicates a lock access of a RMW sequence caused by an atomic instruction accessing DSPRAM.
DSP_ParityEn	Out	Indicate Parity is enabled, based on the value of the *ErrCtl.PE* (CP0) bit.
DSP_WPar[3:0]	Out	Parity Bits for Write operation. Only valid when parity is implemented.
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}

Table 2.3 Signal Descriptions for m14k_cpu Level (Continued)

Signal Name	Type	Description
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_ParPresent	In	Indicate that Parity is present
DSP_RPar[3:0]	In	Read Parity bits. Ignored when parity is not implemented.
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.
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. This is the value to be written to the data array.
ISP_DataWrStr	Out	Data Write Strobe - Indicates that the data array should be written.
ISP_ParityEn	Out	Indicate Parity is enabled, based on the value of the ErrCtl.PE (CP0) bit.
ISP_WPar[3:0]	Out	Parity Bits for Write operation. Only valid when parity is implemented.
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_ParPresent	In	Indicates that Parity is present.
ISP_RPar[3:0]	In	Read Parity bits. Ignored when parity is not implemented.
ISP_Present	SIn	Present - Indicates that a SPRAM array is connected to this port.
Performance Monitoring Interface: These signals can be used to implement performance counters, which can be used to monitor hardware/software performance.		
PM_InstnComplete	Out	This signal is asserted each time an instruction completes in the pipeline.
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:0]	In	This signal is input to a scan chain. (x may be an integer greater than or equal to 0.)
gscanout[x:0]	Out	This signal is output from a scan chain. (x may be an integer greater than or equal to 0.)

Table 2.3 Signal Descriptions for m14k_cpu Level (Continued)

Signal Name	Type	Description
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 9, "Design For Test Features" on page 107 for more details about the use of this interface.		
gmb_dc_algorithm[7:0]	In	Algorithm selection for I$ BIST controller. *gmb_dc_algorithm[0]*=1 means IFA13; *gmb_dc_algorithm[0]*=0 means March C+; *gmb_dc_algorithm[5:1]* : IFA13 retention delay.
gmb_ic_algorithm[7:0]	In	Algorithm selection for I$ BIST controller. *gmb_ic_algorithm[0]*=1 means IFA13; *gmb_ic_algorithm[0]*=0 means March C+; *gmb_ic_algorithm[5:1]*: IFA13 retention delay.
gmb_isp_algorithm[7:0]	In	Algorithm selection for ISPRAM BIST controller. *gmb_isp_algorithm[0]*=1 means IFA13; *gmb_isp_algorithm[0]*=0 means March C+; *gmb_isp_algorithm[5:1]* : IFA13 retention delay.
gmb_sp_algorithm[7:0]	In	Algorithm selection for DSPRAM BIST controller. *gmb_sp_algorithm[0]*=1 means IFA13; *gmb_sp_algorithm[0]*=0 means March C+; *gmb_sp_algorithm[5:1]* : IFA13 retention delay.
gmbinvoke	In	Enable signal for integrated BIST controllers.
gmbdone	Out	Common completion indicator for all integrated BIST sequences.
gmbddfail	Out	When high, indicates that the integrated BIST test failed on the data cache data array.
gmbispfail	Out	When high, indicates that the integrated BIST test failed on the ISPRAM array.
gmbspfail	Out	When high, indicates that the integrated BIST test failed on the DSPRAM 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.

2.3.2 External Interface Signals on m14k_top Level to Custom Blocks

Table 2.4 lists external interface signals on `m14k_top` that allow external system access to custom blocks that may reside inside `m14k_top`.

Note that Table 2.4 does not contain the complete signal list for the `m14k_top` level. Rather, the fullscratchpad RAM, Coprocessor 2 and CorExtend interfaces, described in Table 2.3, are replaced with variable-width to/from busses that allow external access to the custom blocks that may reside inside `m14k_top`. The width of these signals is

configured by the user when the m14k_top block is built. All other interfaces described in Table 2.3 are simply propagated from m14k_cpu and also reside on the m14k_top level.

Table 2.4 Signals on m14k_top for External Interface to Custom Blocks

Signal Name	Type	Description
CorExtend™ External Interface:		
UDI_toudi[x-1:0]	In	Variable-width external input to a custom CorExtend block.
UDI_fromudi[x-1:0]	Out	Variable-width external output from a custom CorExtend block.
Coprocessor 2 External Interface:		
CP2_tocp2[x-1:0]	In	Variable-width external input to a custom coprocessor 2 block.
CP2_fromcp2[x-1:0]	Out	Variable-width external output from a custom coprocessor 2 block.
ScratchPad RAM External Interface:		
ISP_toisp[x-1:0]	In	Variable-width external input to a custom instruction SPRAM block.
ISP_fromisp[x-1:0]	Out	Variable-width external output from a custom instruction SPRAM block.
DSP_todsp[x-1:0]	In	Variable-width external input to a custom data SPRAM block.
DSP_fromdsp[x-1:0]	Out	Variable-width external output from a custom data SPRAM block.

Chapter 3

AHB-Lite Interface

This chapter describes the AHB-Lite™ interface, which is present on the MIPS32® microAptiv™ UP processor core. The AHB-Lite interface is generally described in AMBA 3 AHB-Lite Protocol. The rest of this chapter discusses the specific microAptiv UP implementation of the AHB-Lite interface.

This chapter contains the following major sections:

- Section 3.1, "Interface Transactions"
- Section 3.2, "Clock Ratios"
- Section 3.3, "Write Buffer"
- Section 3.4, "Merging Control"

3.1 Interface Transactions

The microAptiv UP implement 32-bit unidirectional address and data buses: *HADDR[31:0]* for address, *HRData[31:0]* for read operations, and *HWData[31:0]* for write operations. All single timings are related to the rising edge of *HCLK*. The *HCLK* is a reference clock from the gated main clock *SI_ClkIn*, i.e, *HCLK* will be gated off via execution of the WAIT instruction. *HCLK* is also a primary output of the microAptiv UP.

The following sections describe the bus transactions:

- Section 3.1.1, "Basic Transfers"
- Section 3.1.2, "Transfer Types"
- Section 3.1.3, "Transfer Size"
- Section 3.1.4, "Burst Operation"
- Section 3.1.5, "Waited Transfers"
- Section 3.1.7, "Locked Transfers"

3.1.1 Basic Transfers

An AHB-Lite basic read and write transfer consists of two phases:

- *Address*: The Address phase lasts for a single *HCLK* cycle unless it is extended by the previous bus transfer.
- *Data*: The Data phase might require several *HCLK* cycles. It uses the *HREADY* signal to control the number of clock cycles required to complete the transfer.

The simplest transfer is one with no wait states, so the transfer consists of one address cycle and one data cycle. Figure 3.1 shows a simple read transfer, and Figure 3.2 shows a simple write transfer.

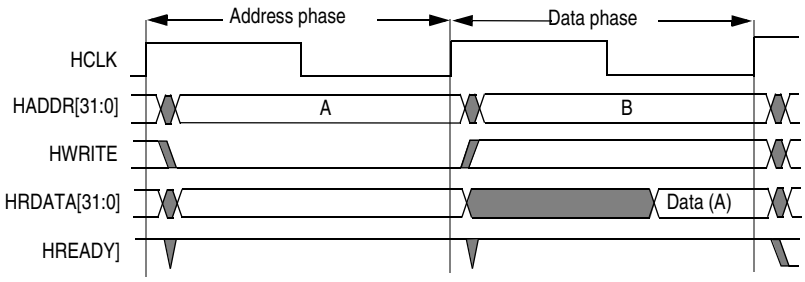


Figure 3.1 Read Transfer with no Wait States

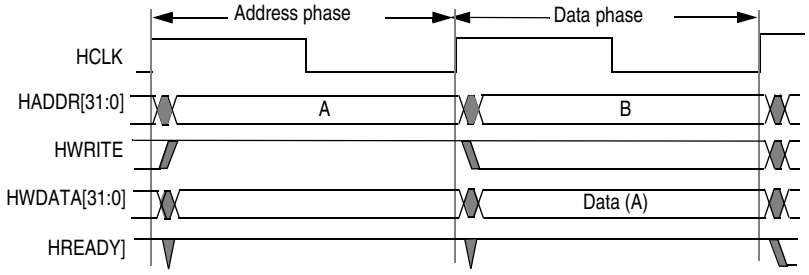


Figure 3.2 Write Transfer with no Wait States

A slave device can insert wait states into any transfer to enable additional time for completion. Figure 3.3 shows a read transfer with two wait states.

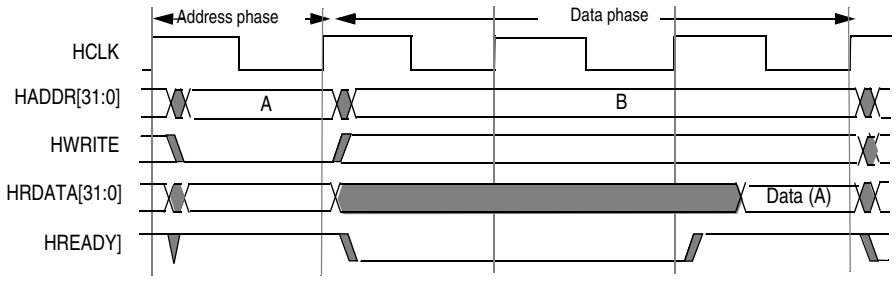


Figure 3.3 Read Transfer with Two Wait States

Figure 3.4 shows a write transfer with one wait state.

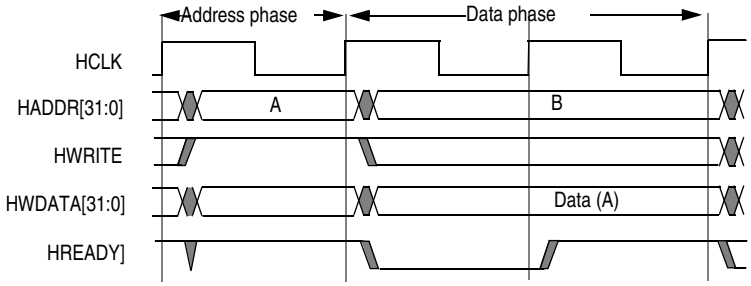


Figure 3.4 Write Transfer with One Wait State

The address phase of any transfer occurs during the data phase of the previous transfer. This overlapping of address and data is fundamental to the pipelined nature of the bus. When a transfer is extended, it has the side-effect of extending the address phase of the next transfer. For write operations, the master holds *HWDATA* stable throughout the extended cycles. For read transfers, the slave does not have to provide valid data on *HRDATA* until the transfer is about to complete. Figure 3.5 shows three transfers to unrelated address, A, B and C with an extended address phase for address C.

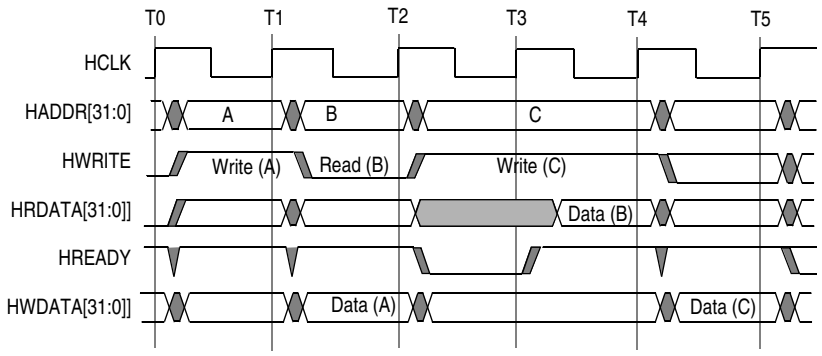


Figure 3.5 Multiple Transfers

3.1.2 Transfer Types

The microAptiv UP core implements three of the four types of transfer defined in the AHB-Lite Protocol, namely, IDLE, NONSEQ, and SEQ. The type of transfer is signified by the *HTRANS[1:0]* signal, as shown in Table 3.1.

Table 3.1 Transfer Types

Type	HTRANS[1:0]	Description
IDLE	2'b 00	Indicates that no data transfer is required
BUSY	2'b 01	Not supported
NONSEQ	2'b 10	Indicates a single transfer or the first transfer of a burst

Table 3.1 Transfer Types

Type	HTRANS[1:0]	Description
SEQ	2'b 11	The remaining transfers in a burst are Sequential and the address is related to the previous transfer

3.1.3 Transfer Size

The microAptiv UP core has 32-bit wide unidirectional read data and write data bus separately. The *HSIZE[2:0]* signal is used to indicate the size of a data transfer. The microAptiv UP core supports three types of transfer size as shown in Table 3.2.

Table 3.2 Transfer Size

Size (Bits)	HSIZE[2:0]	Description
8	3'b 000	Byte Transfer
16	3'b 001	Halfword Transfer
32	3'b 010	Word Transfer

3.1.4 Burst Operation

The *HBURST[2:0]* signal is used to indicate the burst type. The microAptiv UP core supports two types of burst operations, SINGLE and WRAP4, as shown in Table 3.3. A SINGLE burst operation will transfer one word of data. WRAP4 transfers four words of data of the same four-word line, in wrap-around fashion, starting from the addressed word. Figure 3.4 shows the possible sequence order of the words based on different start address.

Table 3.3 Burst Operation Types

Burst Operation	HBURST[2:0]	Description
SINGLE	3'b 000	Single Burst
INCR	3'b 001	Not supported
WRAP4	3'b 010	4 - Beat Wrapping Burst
INCR4	3'b 011	Not supported
WRAP8	3'b 100	Not supported
INCR8	3'b 101	Not supported
WRAP16	3'b 110	Not supported
INCR16	3'b 111	Not supported

Table 3.4 Sequence Order for 4-beat wrapping burst of word

Start Addr (1st Beat)	*HADDR[3:0]* Sequence			
0	0	4	8	C
4	4	8	C	0
8	8	C	0	4
C	C	0	4	8

Figure 3.6 shows a write transfer using a four-beat wrapping burst with a wait state added in the first transfer.

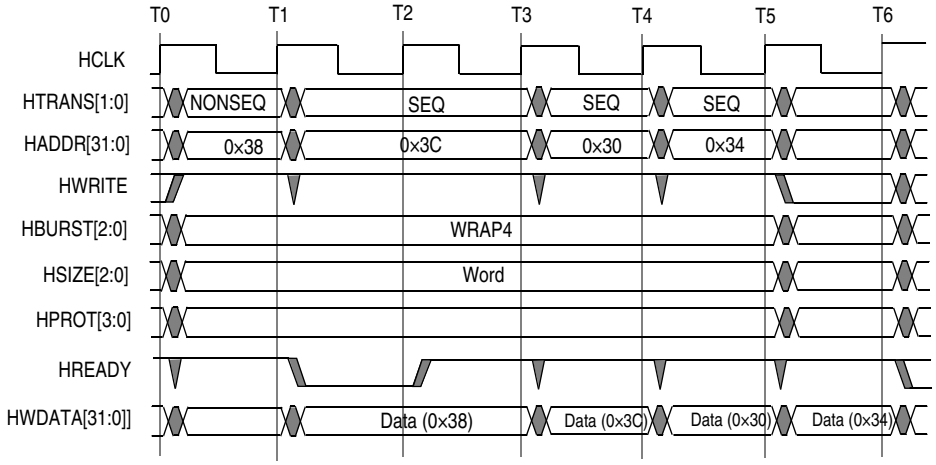


Figure 3.6 Four-Beat Wrapping Burst of Write Transfer

Figure 3.7 shows a read transfer using a four-beat wrapping burst, with a wait state added for the first transfer.

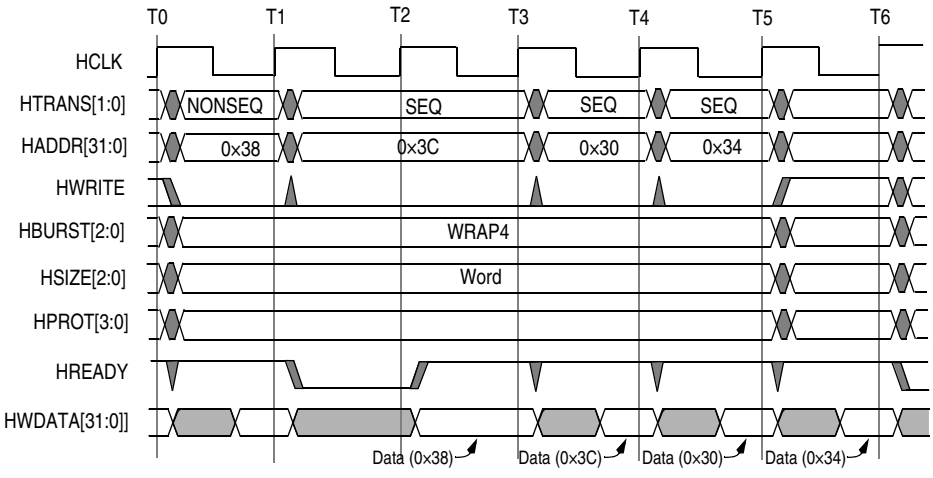


Figure 3.7 Four-Beat Wrapping Burst of Read Transfer

Early Burst Termination

A burst operation can be terminated by the slave device with an ERROR response on the *HRESP* signal. When a slave device issues an ERROR response, the core discontinues the current transfer, and then issues a new burst transaction or issues single transactions for the remaining incomplete data transfers of the previous burst. Figure 3.8 shows a waited transfer, with the address changing, followed by an ERROR response from the slave device. The ERROR response to the access of the address 0x24 will raise a bus error exception in the core.

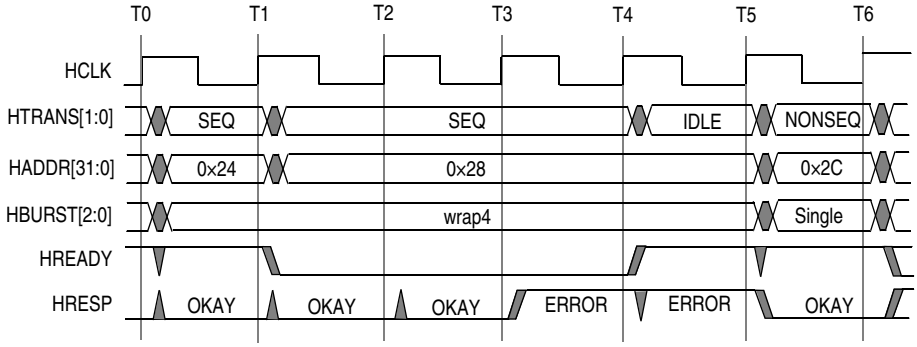


Figure 3.8 Address Changes During a Waited Transfer After an ERROR

If the ERROR response is the last beat of a burst, and the next terminated access is the first beat of a new burst (single or WRAP), this new burst will be restarted on the bus. An example is shown in Figure 3.9.

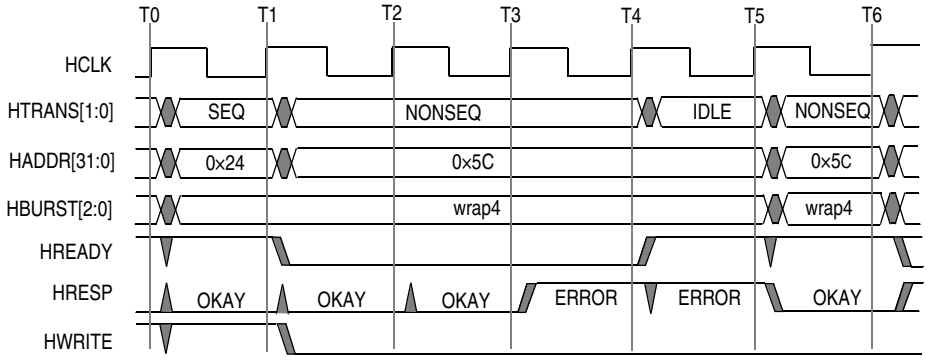


Figure 3.9 Error Response Terminates the First Beat of a Read burst

3.1.5 Waited Transfers

This section describes transfer-type changes during wait states and address changes during Waited states.

IDLE Transfer

During a waited transfer, the master is permitted to change the transfer type from IDLE to NONSEQ and also change the address.

Figure 3.10 shows a waited transfer for a SINGLE burst, with a transfer-type change from IDLE to NONSEQ.

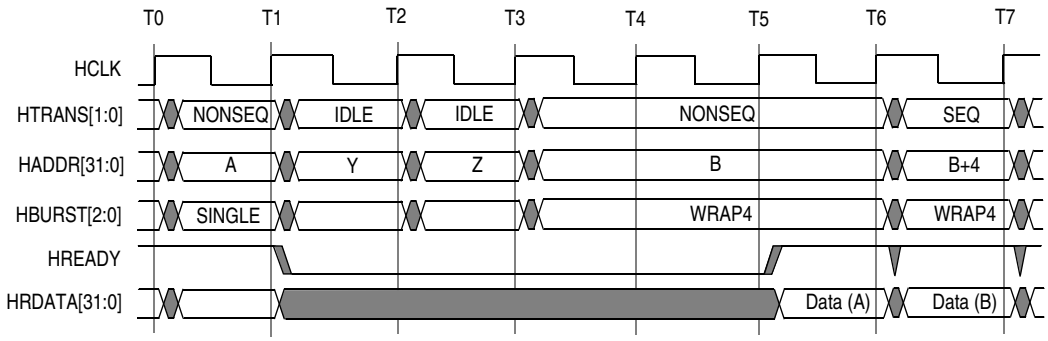


Figure 3.10 Waited Transfer, IDLE to NONSEQ

3.1.6 Protection Control

The bus interface unit of the microAptiv UP core does not generate all protection information on the *HPROT[3:0]* signal. The upper three bits of *HRPOT* defaults to 3'b001, while the lsb is used to distinguish between an opcode fetch and a data access, as shown in Table 3.5.

Table 3.5 Protection Control

HPROT[3:0]	Description
4'b 0010	Opcode Fetch
4'b 0011	Data Access

3.1.7 Locked Transfers

In the microAptiv UP core, *HMASTLOCK* is used to lock access of a RMW sequence raised by an atomic instruction accessing uncached space. Figure 3.11 shows a locked transfer.

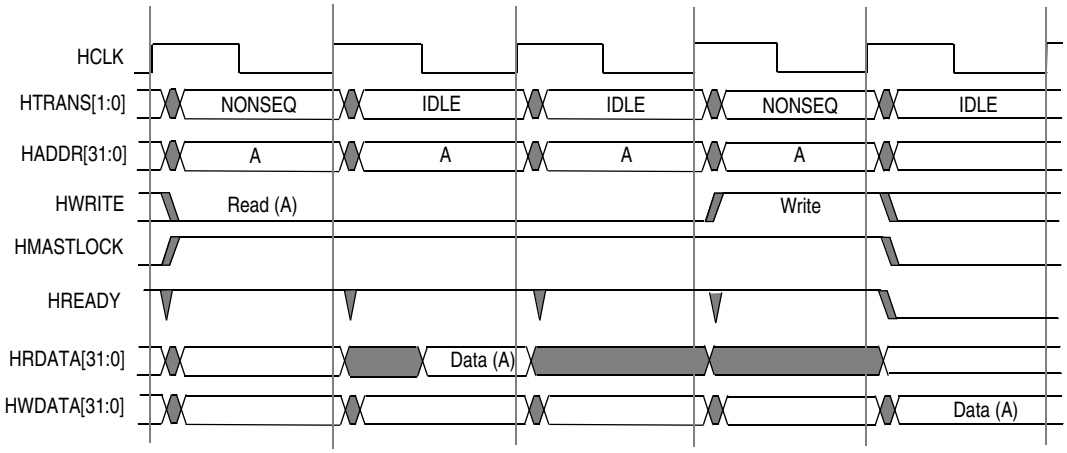


Figure 3.11 Locked Transfer

Between the locked read and write transactions, there must be IDLE cycles inserted to ensure there is enough time for data calculation and data transmission. Other transactions are not allowed to be inserted during these IDLE cycles. *HMASTLOCK* will be held until the write transaction is completed on the bus. The lock sequence would be terminated when the Read part of the RMW sequence gets an ERROR response as shown in Figure 3.12. The *HMASTLOCK* would be de-asserted right after the second phase of the ERROR response of the READ.

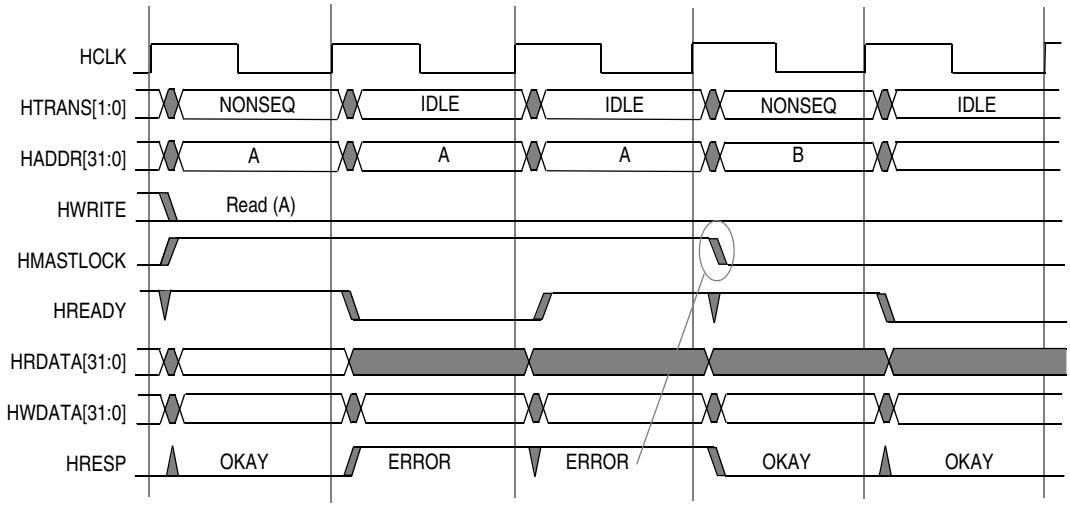


Figure 3.12 HMASTLOCK was deasserted by the ERROR response of Read Sequence

3.2 Clock Ratios

The AHB interface of the microAptiv UP core can be connected to a device that is running at a lower clock rate. The core I/O registers are clocked by the primary core clock, but can be selectively enabled so that outputs are driven and inputs are sampled in the appropriate cycles for communicating with a lower speed device. For devices running at the

same core frequency, the core registers will be enabled every cycle. See Chapter 8, “Clocking, Reset, and Power” on page 99 for more details about the clocking.

3.3 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 AHB-Lite bus under one of the following conditions:

- When a store is attempted by the core to a 16-byte block that is different from the block that 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 16-byte 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.

3.4 Merging Control

All microAptiv UP 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. When merging is disabled, the buffer will still attempt to gather an entire 16-byte 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.

Chapter 4

Interrupt Interface

This chapter describes the interrupt signalling on the MIPS32® microAptiv™ UP processor core. It is divided into the following sections:

- Section 4.1 "Introduction"
- Section 4.2 "Compatibility and Vectored Interrupt Modes"
- Section 4.3 "External Interrupt Controller Mode"

4.1 Introduction

The core I/O for interrupts is treated differently depending on the interrupt mode in which the core is operating. Refer to the *Software User's Manual* [3] for more details on the interrupt modes.

4.2 Compatibility and Vectored Interrupt Modes

In these modes, the interrupt pins are treated as individual requests. Each pin indicates a separate interrupt or group of interrupts. The interrupt pins can be driven asynchronously. They are continually sampled, reflected in the $Cause_{IP}$ field. and made available to the exception generation logic.

In compatibility mode, the 8 interrupt pins and the 2 software interrupt bits are treated equally by hardware. If any one of them is asserted and enabled by software and the operating mode of the processor, an interrupt exception will be taken. The exception vector used is the same for all interrupts and software must check the $Cause_{IP}$ field to determine what interrupts are active and how to service them.

In vectored interrupt mode, the interrupt pins and software interrupt bits are prioritized by the hardware. *SI_Int[7]* is the highest priority, followed by [6],[5]...[0] and then the two software interrupts. The highest priority interrupt that is active and enabled when an interrupt exception is taken will determine which interrupt vector will be used. The software at each vector can be specialized to only handle the particular interrupt associated with it.

In both of these modes, processor generated interrupts (Timer and Performance Counter Interrupts) are output from the core and must be brought back in on the *SI_Int* pins. Traditionally, this has been done on bit[5], but this can be tailored to the particular interrupt scheme in the system. The *SI_IPTI* core input is used to indicate to software which interrupt pin this interrupt is being tied to. This input is shifted to reflect the bit position within the entire $Cause_{IP}$ field - if Int[5] is used, it should be tied to 3'h7, while Int[0] would be signaled as 3'h2. Table 4.1describes how the

various interrupt signals are used in these modes. A number of signals are specific to EIC mode and can be tied to 0 (inputs) or left unconnected (outputs) if an external interrupt controller is not present.

Table 4.1 Interrupt Signals in Compatibility and Vectored Modes

Signal Name	Description
SI_EICPresent	Indicates whether an external interrupt controller is present. This value is visible to software in the $Config3_{VEIC}$ register field.
SI_EICVector	Unused
SI_EISS[3:0]	Unused
SI_IAck	Unused
SI_Int[7:0]	Active high Interrupt pins. These signals are driven by external logic and when asserted indicate an interrupt exception to the core. The *SI_Int* signals go through synchronization logic and can be asserted asynchronously to *SI_ClkIn.* The interrupt pins are level-sensitive and should remain asserted until the interrupt has been serviced. In Release 1 Interrupt Compatibility mode: • All 8 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.
SI_ION[17:0]	Unused
SI_IPL[7:0]	Unused
SI_IPTI[2:0]	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. The value driven on this signal and stored in the register indicates the bit position within the entire $Cause_{IP}$ field. Thus a value of 0x7 would indicate *SI_Int[5]* and 0x2 would indicate *SI_Int[0].*
SI_IVN[5:0]	Unused
SI_SWInt[1:0]	Unused
SI_TimerInt	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.

4.3 External Interrupt Controller Mode

In this mode, the interrupt pins are treated as a bus with an encoded interrupt priority level from 0 to 255 (with 0 indicating no interrupt pending). Because the bits are read as a bus, they must be driven synchronously to the *SI_Clk* clock to ensure that valid values are seen by the processor.

The requested interrupt priority level signaled on *SI_Int[7:0]* is compared with the interrupt priority level that is currently being processed as set by software in $StatusI_{PL}$. If the requested interrupt is a higher priority and the core is in an operating mode where interrupts are allowed, an interrupt exception will be taken, and the interrupt vector used will be a function of the *SI_EICVector* input. When shadow register sets are present, the EIC can also specify which

register set a particular interrupt should use. This value is driven on *SI_EISS*. If different values are used for different interrupts, the requested shadow set should change at the same time as the requested priority level.

When an interrupt exception is taken, the core will assert *SI_IAck* for one *SI_Clk* cycle. At this time, *SI_IPL[7:0], SI_IVN[5:0],* and *SI_ION[17:1]* will also be updated and reflect the priority, vector number, and offset number of the interrupt that was taken respectively. This information may be useful to the interrupt controller.

In EIC mode, processor-generated Timer Interrupts as well as Software Interrupts are output from the core. The interrupt controller should take these values and prioritize them relative to other system interrupts.

Note that the processor core is always in compatibility mode following reset. The interrupt controller should be aware of the signal behavior in that mode as described above, or the software should switch to EIC mode prior to enabling any interrupts.

Figure 4.1shows some example waveforms of interrupt signals. *SI_Int* is changing frequently based on the state of interrupts coming into the EIC. The different interrupts want to use different shadow register sets, so *SI_EISS* is changing at the same time as *SI_Int*. In cycle 5, the core asserts *SI_IAck*, indicating that it has taken an interrupt. The value on *SI_IPL[7:0], SI_IVN[5:0]* and *SI_ION[17:1]* are updated and show which interrupt was taken. At this point, the currently requested interrupt is a higher priority, so when software re-enables interrupts, the core will take the higher priority interrupt.

Figure 4.1 EIC Interrupt Signals

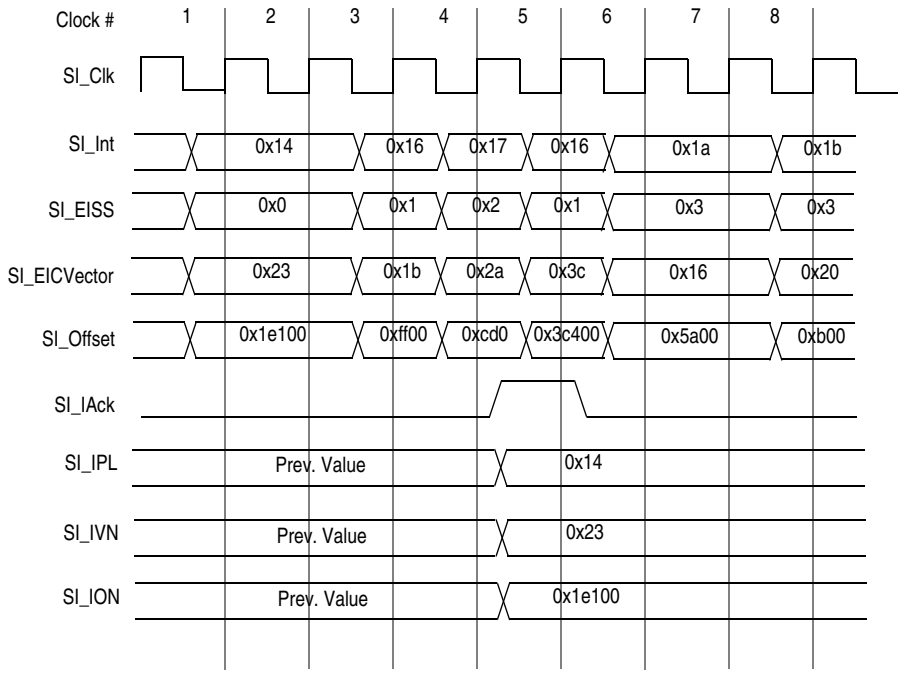


Table 4.2 describes the EIC mode behavior of the interrupt signals.

Table 4.2 Interrupt Signals in EIC Mode

Signal Name	Description
Interrupt Signals:	
SI_EICPresent	Must be set to 1 to enable EIC mode.
SI_EICVector[5:0]	Specifies the interrupt vector to be used for the currently requested interrupt. If the interrupt is taken, the exception processing will begin at this interrupt vector. This allows the vector number to be different than the priority level. If this functionality is not needed, this input should be driven to the same value as *SI_Int*. This value will be output on the *SI_IVN* pins and *SI_IAck* will be asserted for one cycle.
SI_EISS[3:0]	General purpose register shadow set number to be used when servicing an interrupt in EIC interrupt mode.
SI_IAck	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[7:0]* pins into the $Cause_{RIPL}$ field (overlaid with $Cause_{IP9..IP2}$), and signals the external interrupt controller to notify it that the interrupt request is being serviced.
SI_Int[7:0]	Active high Interrupt pins. These signals are driven by external logic and when asserted indicate an interrupt exception to the core. In External Interrupt Controller mode, the interrupt pins are interpreted as an encoded value, so they must be asserted synchronously to the *SI_Clk* clock 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. • An external block prioritizes its various interrupt requests and produces a vector number of the highest priority interrupt to be serviced. • The priority level is driven on the *SI_Int* pins and is treated as an 8-bit encoded value in the range of 0..255. • When the core starts the interrupt exception, it loads the sampled value of the *SI_Int[7:0]* pins into the $Cause_{RIPL}$ field (overlaid with $Cause_{IP9..IP2}$). This value will be output on the *SI_IPL* pins, and *SI_IAck* will be asserted for one cycle.
SI_ION[17:1]	Current interrupt offset number, provided for use by an external interrupt controller. This value is updated whenever *SI_IAck* is asserted.
SI_IPL[7:0]	Current interrupt priority level from the $Cause_{IPL}$ register field, provided for use by an external interrupt controller. This value is updated whenever *SI_IAck* is asserted.
SI_IVN[5:0]	Current interrupt vector number, provided for use by an external interrupt controller. This value is updated whenever *SI_IAck* is asserted.
SI_IPTI[2:0]	Unused
SI_Offset[17:1]	This signal will be used when “Send entire vector offset along with RIPL during EIC mode” is selected during configuration. Otherwise, it will ignored. This specifies the vector offset to be used for the currently requested interrupt. If the interrupt is taken, the exception processing will begin at this vector offset. This allows the vector offset to be directly used by the processor.This value will be output on *SI_ION*, and *SI_IAck* will be asserted for one cycle.

Table 4.2 Interrupt Signals in EIC Mode (Continued)

Signal Name	Description
SI_SWInt[1:0]	Software interrupt request.These signals represent the value in the *IP[1:0]* field of the *Cause* register. They are provided for use by an external interrupt controller.
SI_TimerInt	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.

Chapter 5

EJTAG Interface

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

This chapter contains the following sections:

- Section 5.1 "EJTAG versus JTAG"
- Section 5.2 "How to Connect EJ_* Pins"
- Section 5.3 "cJTAG Interface"
- Section 5.4 "Multi-Core Implementations"
- Section 5.5 "Trace Capability"
- Section 5.6 "SecureDebug"

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

Reference to the general *EJTAG Specification* [6] 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.

5.1 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® microAptiv™ UP Processor CoreSoftware User's Manual* [3] to learn more about the software debugging capabilities of EJTAG.

5.1.1 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.

5.1.2 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 microAptiv UP core, however. The microAptiv UP 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.

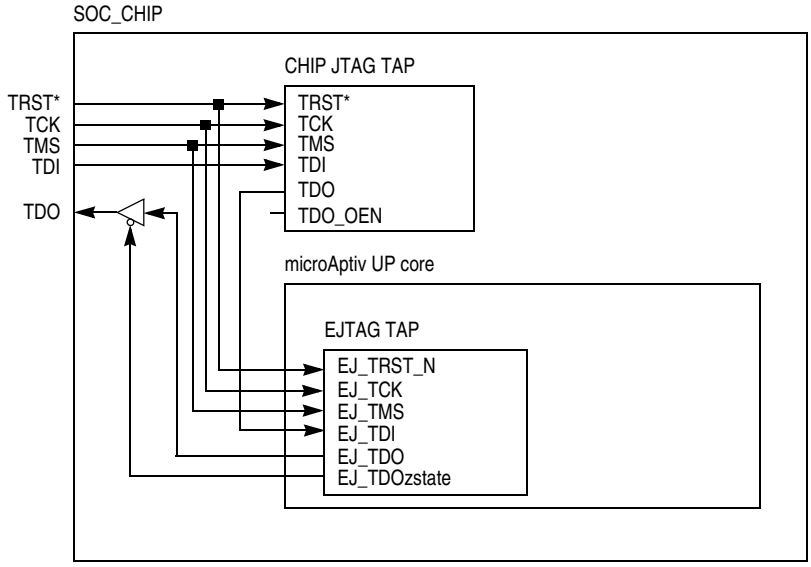
5.1.2.1 Daisy-Chained TDI-TDO

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 5.1 shows the serial *TDI-TDO* chain setup with parallel control of the TAP controllers.

Figure 5.1 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)

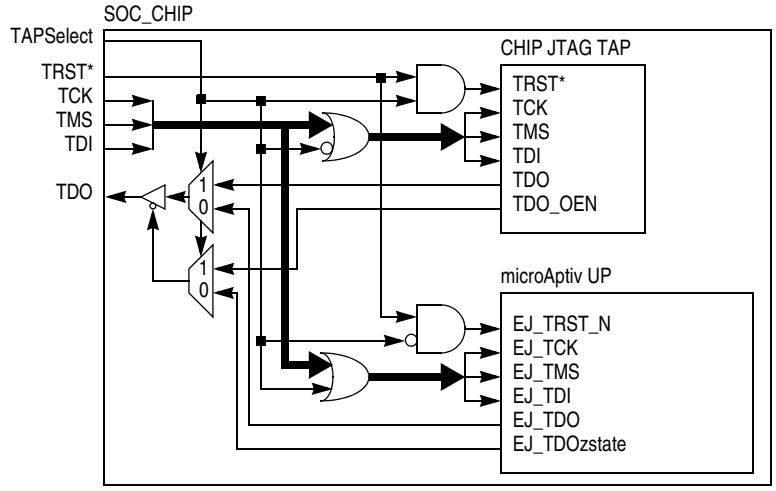
This will enable the debugger to always select the bypass register within the JTAG TAP controller during EJTAG access, and compensate for the bypass register length.

The main concern is the presence of the serial EJTAG TAP controller in the JTAG TAP path; automatic JTAG test-benches may not like the visibility of another TAP controller inside the chip. If considering the JTAG-EJTAG daisy-chained approach, ensure any debug tools that will be connected to the JTAG or EJTAG controllers are capable of placing the other controller(s) in bypass and the additional shift length can be accommodated by the tools.

5.1.2.2 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 5.2, a setup which anticipates the existence of *TRST** on the "CHIP JTAG TAP" controller is shown.

Figure 5.2 Multiplexing Between JTAG and EJTAG TAP Controllers



TAPSelect in Figure 5.2 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 microAptiv UP core (and thus probably the entire SOC_CHIP) is held in reset. During reset, EJTAG commands can initialize the microAptiv UP core to leave the reset state in Debug Mode, and thus the debug interface can control the microAptiv UP 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 5.2.

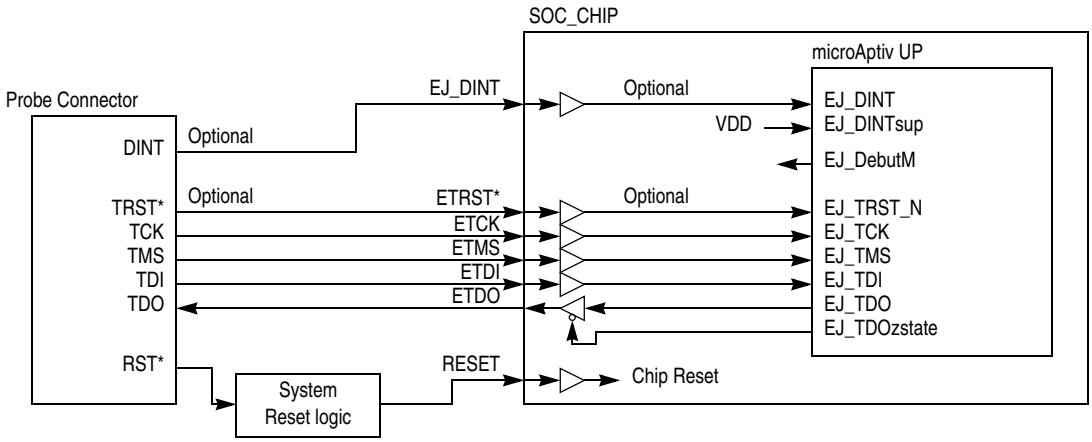
5.2 How to Connect EJ_* Pins

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

5.2.1 EJTAG Chip-Level Pins

The EJTAG TAP signals on the microAptiv UP 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 5.3 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 5.3 EJTAG Chip-Level Pin Connection



Note: Probe connections should include pull-up, pull-down, or series resistors. See the EJTAG Specification fo rmore details.

AC timing characteristics for the *ETDO* driver and the input buffers can be found in Section 8.2, "AC Timing Characteristics", of the *EJTAG Specification* [6]. In particular notice that all the probe pins must have pull-up, pull-down, or series resistors attached; see Section 8.5, "Target System PCB Design" of the *EJTAG Specification*. As shown in Figure 5.3, 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 5.4 for further details on EJTAG-related reset circuitry.

5.2.1.1 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 microAptiv UP 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.

5.2.1.2 Optional EJ_DINT Pin

The *EJ_DINT* input pin is also optional. An assertion of *EJ_DINT* in the microAptiv UP core triggers a Debug Interrupt Exception. This will stop the normal program flow within the microAptiv UP 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 *EJ_DINT* pin can be removed from the chip.

EJ_DINT on the microAptiv UP core may also be connected to on-chip logic, such as a Multi-Core Breakpoint Unit (see Figure 5.6 for more details). The *EJ_DINTsup* (EJTAG Debug Interrupt Pin Supported) input on an microAptiv UP 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 microAptiv UP core to an interrupt source is not connected, then both *EJ_DINT* and *EJ_DINTsup* must be deasserted by connecting them to logic zero.

5.2.2 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]*.

5.2.2.1 EJ_ManufID[10:0]

EJ_ManufID[10:0] must be a compressed form of a JEDEC standard manufacturer's identification code. See "5.2.2 "EJTAG Device ID Input Pins" on page 50".

5.2.2.2 EJ_PartNumber[15:0]

EJ_PartNumber[15:0] is recommended to be a manufacturer-specific number identifying this core as a MIPS32 microAptiv UP 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.

5.2.2.3 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.

5.2.3 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 microAptiv UP core, accessible in Debug Mode).

5.2.3.1 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 5.4 shows one possible implementation for the use of *EJ_PrRst*.

5.2.3.2 EJ_PerRst Signal

Peripheral Reset can be used as a soft reset of the peripherals surrounding the microAptiv UP core. The effect of an asserted *EJ_PerRst* is implementation-dependent; however, it should never result in a reset of the microAptiv UP core itself. Figure 5.4 shows one possible implementation of the use of *EJ_PerRst*.

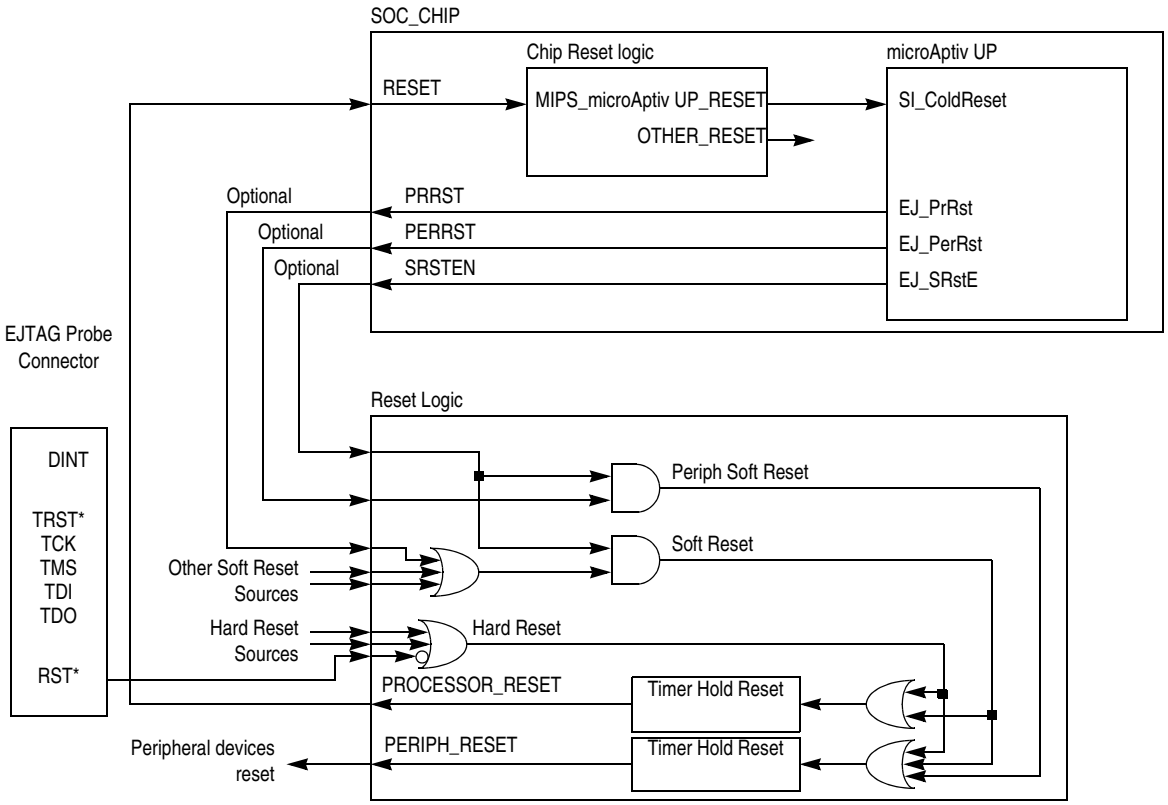
5.2.3.3 EJ_SRstE Signal

As described earlier, this signal can be used to control one or more Soft Reset sources in the system reset logic. See Figure 5.4 for a possible implementation.

5.2.3.4 A Reset Logic Implementation

Figure 5.4 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 microAptiv UP core. This requires 3 extra output signals, but this need not be the case.

Figure 5.4 Reset Circuitry Implementation



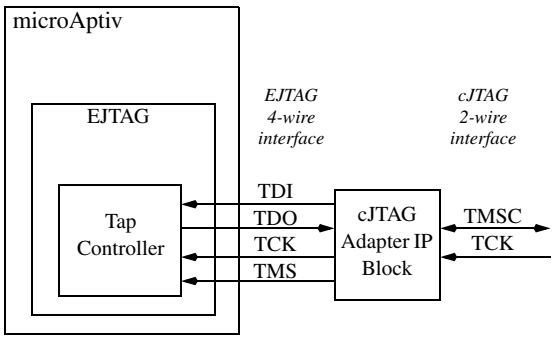
Note: The *RST** input to the Reset Logic from the Probe Connector is a required connection when implementing EJTAG into the system.

5.3 cJTAG Interface

One of the enhancements in the updated IEEE 1149.7 standard is the reduction in the number of external signals required on the EJTAG interface from four to two. This 2-pin interface, also known as the cJTAG interface, provides the capability of debugging with only two wires when pin count is critical..

MIPS provides a cJTAG Adapter IP block on MIPS softcores to convert between the 2-pin interface in 1149.7 and the 4-pin interface in 1149.1. The adapter IP resides outside the core and is treated as a separate IP block from the point of view of integration. The implementation is shown in Figure 5.5.

Figure 5.5 cJTAG Interface



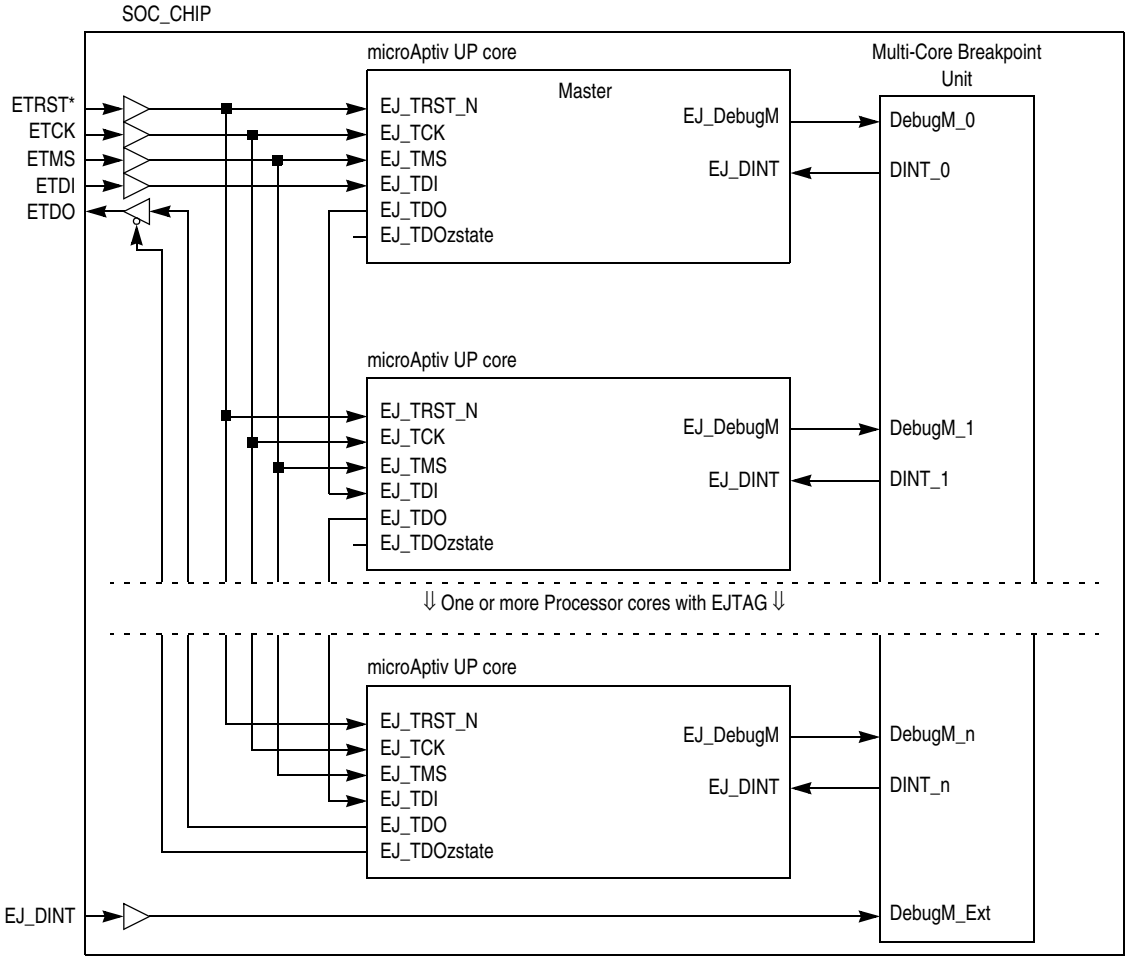
MIPS debug probes have been enhanced to support cJTAG—a single probe can support both legacy EJTAG and the new cJTAG. EJTAG and cJTAG use the same 14-pin connector specified in the MIPS EJTAG specification, but when connected to cJTAG, the TDI and TDO signals are not used.

The 1149.7 specification is complex and much more flexible than is needed in normal applications. The provided cJTAG Adapter IP implemented is a subset of 1149.7 specifcation. Refer to the *cJTAG Adapter User's Manual* [7] for more details.

5.4 Multi-Core Implementations

In a chip configuration with multiple microAptiv UP 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 5.6.

Figure 5.6 Multi-Core Implementation



5.4.1 TDI/TDO Daisy-Chain Connection

In a Multi-Core implementation, one of the processor cores is often be the Master. In Figure 5.6, 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.

5.4.2 Multi-Core Breakpoint Unit

The Multi-Core Breakpoint Unit (MCBU) shown to the right in Figure 5.6 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 microAptiv UP core ignores any subsequent Debug Interrupts on *EJ_DINT*.

5.5 Trace Capability

An microAptiv UP core can support MIPS iFlowtrace™ features. The iFlowtrace mechanism is an option that provides tracing of the Program Counter and special events. The iFlowtrace logic is included as a build-time option. Four basic options are possible:

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

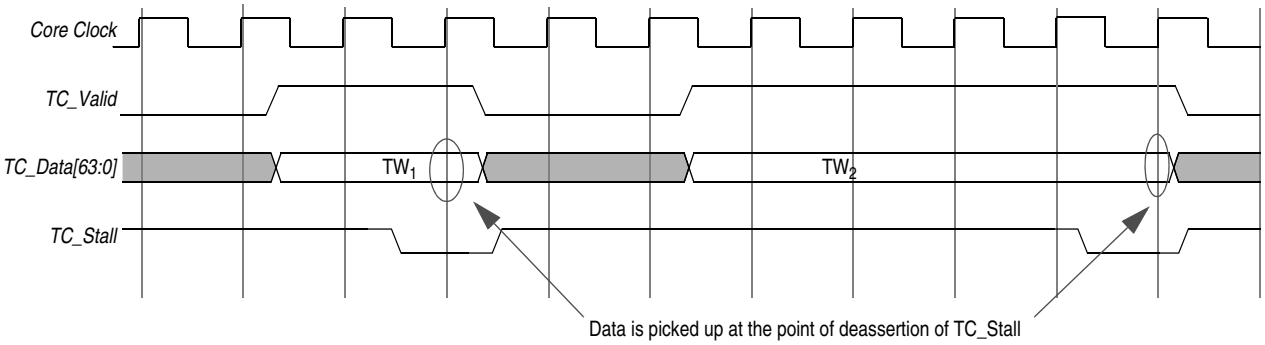
If options 1 or 2 are present, then the *TC_* output pins on the core will be statically driven to zero, and all the *TC_* inputs are ignored. With option 2, access to the trace features and on-chip trace memory occurs through DRSEG registers, namely ITCBTW, ITCBRDP and ITCBWRP.

For the remaining options, the TCtrace Interface on the microAptiv UP 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* [6] which structured the whole connection logic to support the complete MIPS Trace feature. But for iFlowtrace, it is relative simpler. There are only 5 useful TCtrace ports, namely *TC_DATA*, *TC_Valid*, *TC_Stall*, *TC_ClockRatio* and *TC_Pibpresent*. *TR_DATA* is fixed at 4 bits wide, *TR_CLK* is always one half or one fourth of the core cycle.

Figure 5.7 shows the timing relationship between *TC_Valid* and *TC_Stall*, when *TC_Data* changes value depending on *TC_Stall*.

Figure 5.7 *TC_Valid* and *TC_Stall* Timing



5.6 SecureDebug

The SecureDebug debug feature is optional and it is used to provide a controllable method to disable EJTAG access so that an EJTAG probe cannot be used to control a target processor, place it into debug mode, insert instructions, access memory, breakpoint, or single step.

An input signal to the core, *EJ_DisableProbeDebug*, when asserted has the following effects:

1. It forces ProbEn = 0
2. It forces ProbTrap = 0
3. EjtagBrk is disabled.
4. EJTAGBOOT is disabled.
5. PC Sampling is disabled.
6. DINT signal is ignored .

Suggested implementation of the *EJ_DisableProbeDebug* signal is for a microcontroller to provide a bit within non-volatile memory (outside the core) that is pre-programmed to set or clear this control signal.

Note that cJTAG is implemented by converting the EJTAG signals to two cJTAG signals. If the SecureDebug feature is implemented, cJTAG is similarly secured.

For more detail information, refer to the *Disabling EJTAG Debugging* section in the *EJTAG Debug Support in the microAptiv™ UP Core* chapter of the *MIPS32® microAptiv™ UP Processor Core Software User's Manual*.

Chapter 6

Coprocessor Interface

This chapter describes the MIPS Core Coprocessor Interface supported by the MIPS32 microAptiv UP processor core. The MIPS Core Coprocessor Interface is described in the companion document, titled *Core Coprocessor Interface Specification* [9]. The Core Coprocessor Interface is an optional feature in an microAptiv UP core. If the microAptiv UP core does not contain the Core Coprocessor Interface logic, then this chapter is irrelevant. This chapter discusses the specific microAptiv UP implementation of the Core Coprocessor Interface, in the following sections:

6.1 Introduction

The microAptiv UP 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 microAptiv UP 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 an microAptiv UP 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 microAptiv UP processor core.
- Fully compliant to the MIPS Core Coprocessor Interface standard.

6.2 Coprocessor Instructions

An microAptiv UP core supports all MIPS32-compliant COP2 instructions, except the load double (LDC2) and store double (SDC2) instructions. Table 6.1 lists all the supported instructions and how they are decoded.

Table 6.1 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[1].
SWC2	IR[31:26] = 111010_2	Store Word to memory from a Coprocessor 2 register. COP2 register number = IR[20:16], sub-select = 0[1].
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][2].
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][3].
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][2].
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][3].
BC2F BC2FL	IR[31:26] = 010010_2 & IR[25:23] = 010_2 & IR[16] = 0_2	Branch on Coprocessor 2 condition false (likely)[4]. 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)[4]. 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].

1. The LWC2 and SWC2 instructions has no room to specify a sub-select COP2 register value. sub-select 0 must be assumed.
2. 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.
3. 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.
4. 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 6.1 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 microAptiv UP core (see 6.4.5 “Coprocessor Exceptions”).

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

6.3 Signal Configuration

The microAptiv UP 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 microAptiv UP 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 for a listing of all the microAptiv UP core signals).

The *CP2_inst32_0* output is tied high (logic 1). The microAptiv UP 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 microAptiv UP core never sends data out of order. The coprocessor attached to an microAptiv UP core does not need to limit the use of out-of-orderness. 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-orderness of the data received from the coprocessor. If data is sent to the microAptiv UP 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 microAptiv UP 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.

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

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 microAptiv UP 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).

6.4 Interface Protocols

Refer to Table 2.3 for a complete listing of all the pins of the microAptiv UP 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 microAptiv UP core.
- **Coprocessor Exceptions** - Notifies the microAptiv UP 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 microAptiv UP processor core. The additional information and the transfers required are summarized in Table 6.2.

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 6.2 Transfers Required for Each Dispatch

Dispatch Type	Required Transfers
To COP Op (LWC2/ MTC2/ CTC2)	• Instruction nullification or not[1] • To Coprocessor data transfer • Coprocessor exceptions or not • Instruction killing or not
From COP Op (SWC2/ MFC2/ CFC2)	• Instruction nullification or not[1] • From Coprocessor data transfer • Coprocessor exceptions or not • Instruction killing or not
Arithmetic Op (COP2[2])	• Instruction nullification or not[1] • Coprocessor exceptions or not • Instruction killing or not
Arithmetic Op, Branch (BC2[2])	• Instruction nullification[1] • Condition code check results • Coprocessor exceptions or not • Instruction killing or not

1. The microAptiv UP core will always signal not-nullified on all instructions.
2. 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 6.1 General Transfer Example

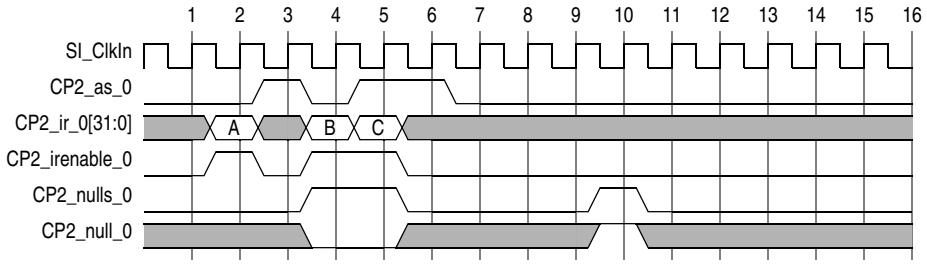


Figure 6.1 above shows examples of the transfer of nullification information. All non-dispatch transfers follow the same protocol.

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 microAptiv UP core, also the nullify strobe is always send in the cycle after dispatch on the microAptiv UP 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 microAptiv UP 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 6.3. 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 6.3 Allowable Interface Latencies from a Coprocessor to the microAptiv UP Core

From	To	Stage Needed	Min (cycles)	Max (cycles)	microAptiv UP 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[1]	1	∞	-1[2]

1. The microAptiv UP 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.
2. 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 microAptiv UP core does not generate all allowable latencies for transfers from the integer unit to the coprocessor. Table 6.4 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 dis-

patch that the integer unit will send the data. The “Max” column shows the longest time after dispatch that the data could be sent.

Table 6.4 Interface Latencies from the microAptiv UP Core to a Coprocessor

From	To	Stage Sent	Min (cycles)	Max
Instruction Dispatch	Instruction Nullification	E+1	1[1]	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)

1. 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.

6.4.1 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 an microAptiv UP 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 microAptiv UP 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 microAptiv UP 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 microAptiv UP 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 microAptiv UP 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 6.2 Instruction Dispatch Waveforms

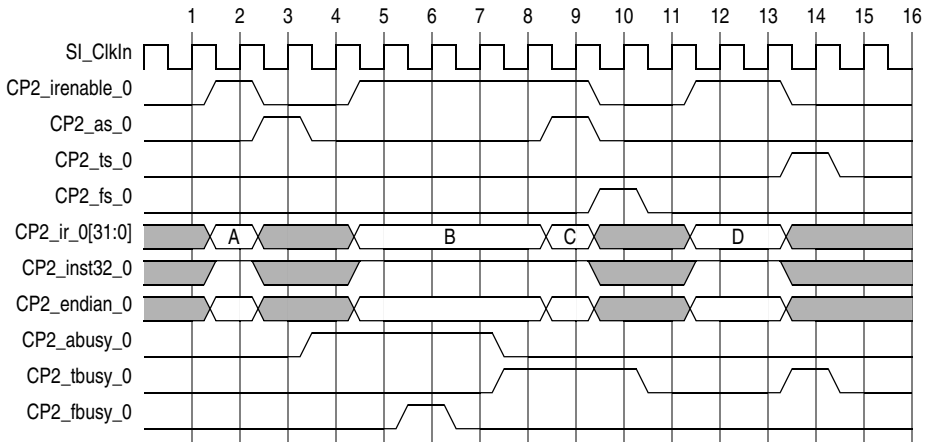


Figure 6.2 shows example waveforms of four instruction dispatches.

- 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 microAptiv UP 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 microAptiv UP 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 microAptiv UP 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

6.4.2 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 microAptiv UP from reordering To COP Data by driving *CP2_tordlim_0[2:0]* to 000_2.

Note: The microAptiv UP 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 6.3 To Coprocessor Data Waveforms

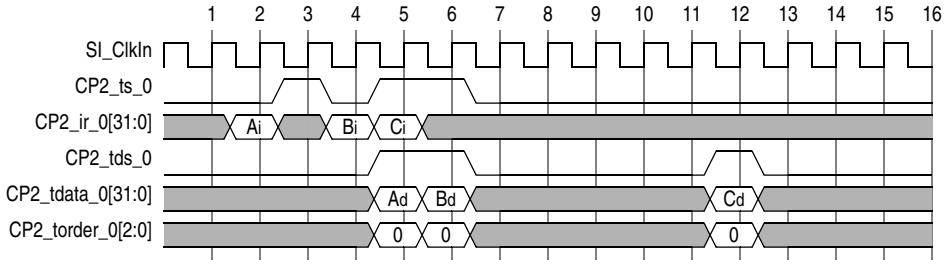


Figure 6.3 shows waveforms for 3 To COP Op instructions and the data transfer associated with this instruction. On edges 2, 4 and 5 the To COP Op instructions A, B and C respectively are dispatched to the coprocessor. Because they are To COP Ops, the *CP2_ts_0* strobe is used to strobe the instruction 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 microAptiv UP 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 microAptiv UP core internal reasons. The Coprocessor must support any cycle delay from instruction dispatch to data transmit on To COP Ops.

6.4.3 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 microAptiv UP 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 microAptiv UP 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 6.4 From Coprocessor Data Waveforms

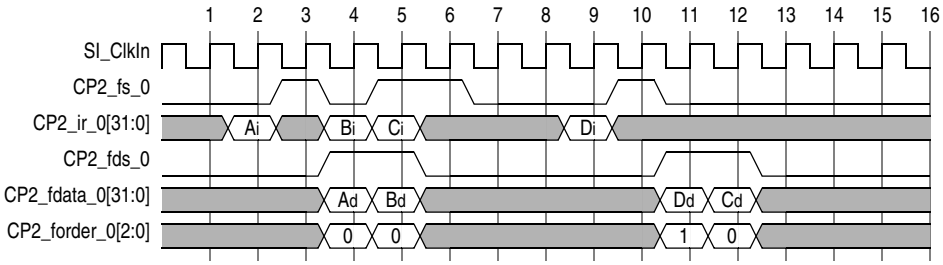


Figure 6.1 shows example waveforms for 4 From COP Op instructions, and the data transfer associated with these instructions. On edge 2, 4, 5 and 9 the From COP Ops A, B, C and D respectively are dispatched from the integer core. They are all From COP Ops, thus *CP2_fs_0* is used to strobe the instruction.

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 microAptiv UP core. The data for instruction A is one cycle delayed, causing one stall cycle in the microAptiv UP 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 1001_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.

6.4.4 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 microAptiv UP 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 microAptiv UP core. The values are not available until the cycle after return from the coprocessor.

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

Figure 6.5 Condition Code Check Waveforms

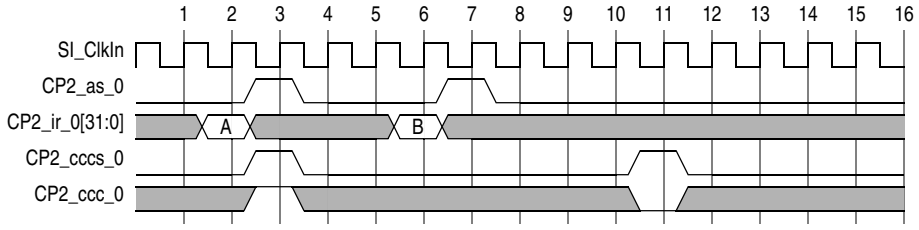


Figure 6.5 shows an example waveform for two BC2 instructions. BC2 instructions belong to the arithmetic COP Op group of instructions and the dispatch is thus strobed using the CP2_as_0 strobe.

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 microAptiv UP 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.

6.4.5 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 microAptiv UP 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 an implementation-specific exception code (IS1). This exception code 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 exceptions also apply to IS1 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[2].

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

The microAptiv UP 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]*).

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

Figure 6.6 Exception Waveforms

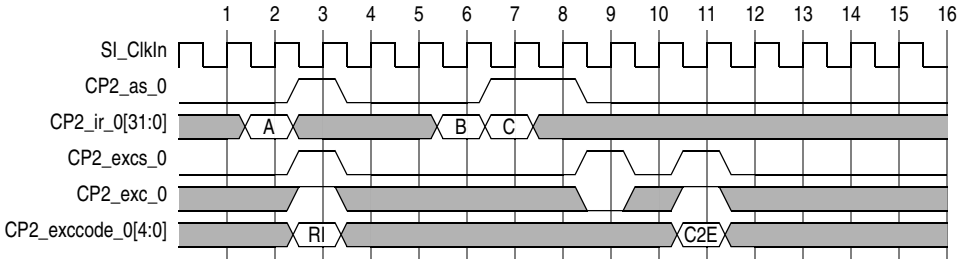


Figure 6.6 shows example waveforms for an exception return from three coprocessor instructions. In this example, the exception returns are all arithmetic COP Ops, and *CP2_as_0* is used to strobe the dispatch.

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 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.

6.4.6 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 microAptiv UP core never nullifies an instruction. No nullify is always transferred in the cycle after dispatch.

Nullification transfers follow the generic example given in Figure 6.1.

6.4.7 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 6.7 Instruction Killing Waveforms

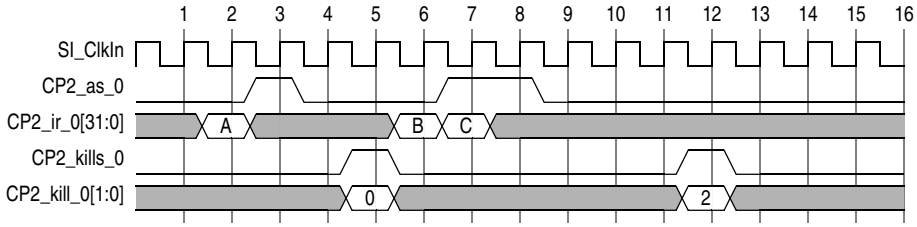


Figure 6.7 shows example waveforms for instruction killing.

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.

6.5 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 microAptiv UP core and the coprocessor.

6.5.1 No Coprocessor Present

If a hard-core version of the microAptiv UP 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 microAptiv UP 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 microAptiv UP core low (0) if there is no coprocessor attached to the integer core.

6.5.2 How to Use CP2_idle

CP2_idle is an input to the microAptiv UP 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 microAptiv UP core. When **WAIT** is executed, the microAptiv UP 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*.

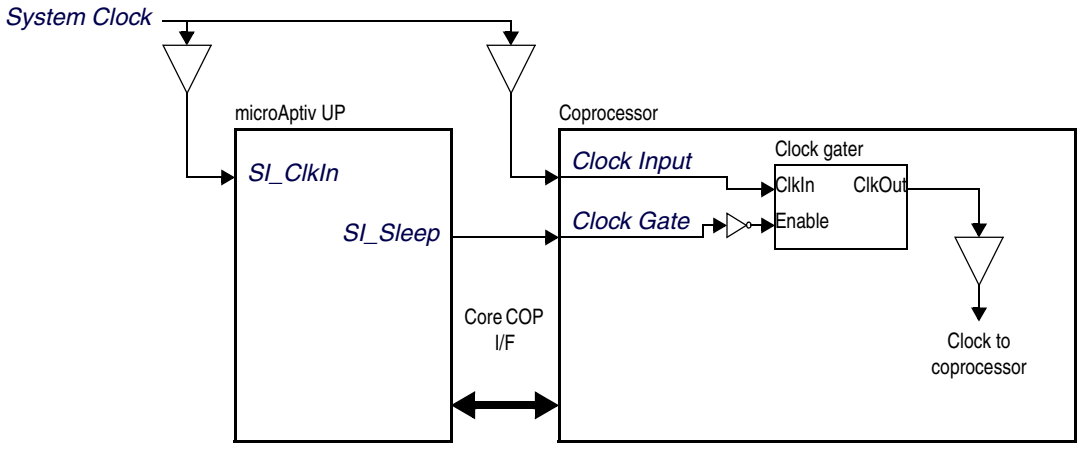
While the WAIT instruction ensures that no new instructions go down the pipe in the integer core, nothing is implicitly done to tell the coprocessor to prepare for a possible stopping of its clock. This is where the *CP2_idle* signal is used. The coprocessor must assert this signal high whenever no instruction 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 microAptiv UP 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 microAptiv UP core. After the **WAIT** instruction has actively stopped the main clock to most of the microAptiv UP 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 microAptiv UP core.

6.5.3 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 microAptiv UP core has an output, *SI_Sleep,* which indicates when the internal clock in the integer core is stopped. Figure 6.8 shows an example of how to implement and control a top-level clock gater in the coprocessor.

Figure 6.8 Use of SI_Sleep for Clock-Gating in the Coprocessor



6.5.4 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 6.9 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 6.9 Clock-Gating of To Data Registers in Coprocessor

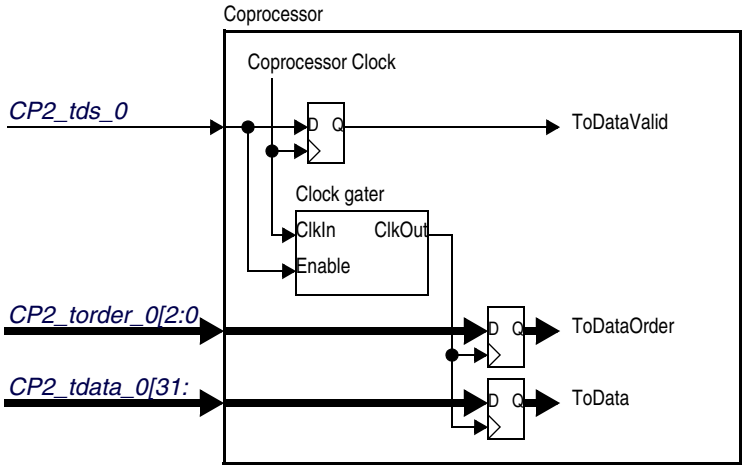
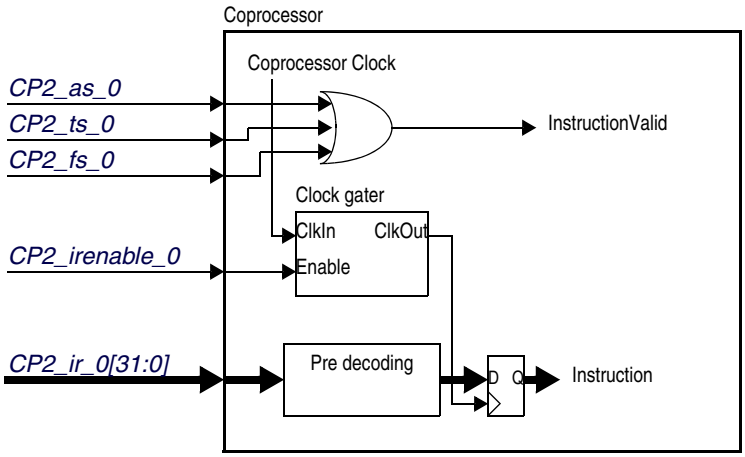


Figure 6.10 shows the intended use of *CP2_irenable_0*. *CP2_irenable_0 is used* only as a gated-clock enabling signal when the clock-gating on the capture of the instruction word is introduced. For all other purposes, the *CP2_as_0*, *CP2_ts_0* and *CP2_fs_0* are the true qualifiers for a valid instruction.

Figure 6.10 Clock Gating of Instruction Registers in Coprocessor



The Pre-decoding block in Figure 6.10 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 microAptiv UP core is to be maintained. Refer to Table 6.4, for information on allowable latencies to maintain stall-free operation.

6.6 Template for Coprocessor Modules

A template for coprocessor 2 modules is included in the soft core release. This template provides a simple implementation of Cop2 interface logic. It can be used as is for many coprocessor designs or can be used as a reference for designing coprocessor interface logic. There is an application note, *Core Coprocessor 2 Module Template Application Note* [11] in the `$MIPS_PROJECT/doc` directory as well as RTL in the `$MIPS_PROJECT/design/modules/user/cop2_syn` directory.

Chapter 7

Scratchpad RAM Interface

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

The pin list associated with the SPRAM interface was introduced in Chapter 2, “Signal Descriptions” on page 11. This chapter contains further details about the use of the SPRAM interface in a system and is specific to the microAptiv UP core, organized into the following major sections:

7.1 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.
- An SPRAM port logically replaces one way of a cache. If both SPRAM and cache are present, then the maximum cache associativity is 3.

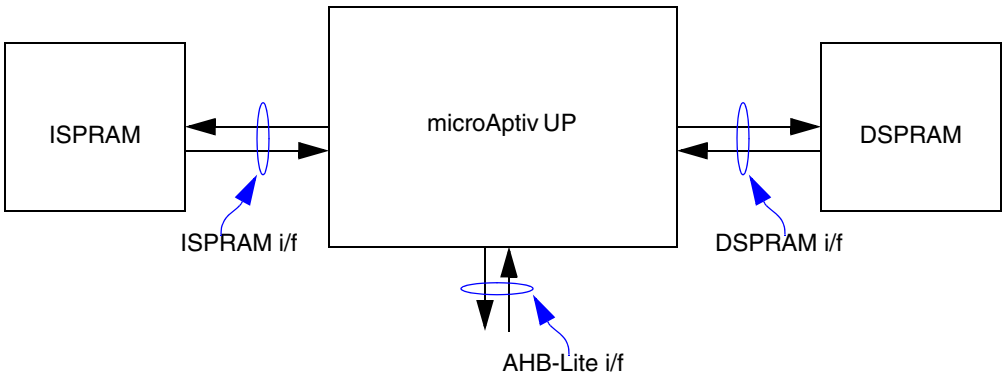
- SPRAM can be mapped to either cached or uncached space. The address decode and comparison for SPRAM will be performed regardless of the cacheability attribute. When Instruction SPRAM can service uncached references, enabling processor boot with no AHB-Lite interface accesses.
- Backstalling. The SPRAM 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.
- Optional parity protection is supported for SPRAM. For DSPRAM, one parity bit is implemented for every 8 bits of data. For ISPRAM, one parity bit is implemented for every 8 bits of instruction opcode.
- The microAptiv UP core provides a single-cycle latency SPRAM implementation with BIST support.

7.2 SPRAM Overview

A block diagram of a basic microAptiv UP system with SPRAM functionality is shown in Figure 7.1.

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 microAptiv UP core utilizing the other port(s).

Figure 7.1 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 microAptiv UP 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 microAptiv UP core). Virtual aliasing occurs when a single physi-

cal 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.

7.2.1 SPRAM Differences From 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.

7.2.2 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 7.2.4, "Delayed Stores".

The data scratchpad port can accommodate either no parity or one bit of parity per byte. There is a 4-bit parity bus between the scratchpad RAM and the processor; when no parity is implemented or the parity is disabled, the parity busses are ignored.

Many of the signals on the SPRAM interface apply to only one of the tag/data accesses, while others apply to both. Table 7.1 shows which signals are related to tag access, data access, or both and when they are logically valid.

Table 7.1 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

Table 7.1 SPRAM Interface Cycle Timing (Continued)

Signal Name	Port	Dir.	Typical Timing, as % of min. cycle	Validity relative to strobes/stalls
{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_DataWrStr may be asserted with all 0's on DSP_DataWrMask and no write should occur.
DSP_DataWrMask	Data	Out	80	Valid when DataWrStr is asserted.
{I,D}SP_ParityEn	Data	Out	Static	Static configuration output.
ISP_WPar	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 in which Stall is deasserted
DSP_Lock	Both	Out	90	Asserted to indicate a lock access of a RMW sequence raised by an atomic instruction to DSPRAM space. The lock signal is valid since DSP_TagRdStr and DSP_DataRdStr are asserted and will be held until DSP_DataWrStr is also asserted. The signal will also be deasserted when Read operation is missed in DSPRAM.
DSP_WPar	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 in which Stall is deasserted
{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.
{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.

Table 7.1 SPRAM Interface Cycle Timing (Continued)

Signal Name	Port	Dir.	Typical Timing, as % of min. cycle	Validity relative to strobes/stalls
{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
{I,D}SP_ParPresent	Data	In	Static	Static configuration input
{I,D}SP_RPar	Data	In	60	For single-cycle accesses, parity for read data should be returned the cycle after DataRdStr/RdStr is asserted. For multi-cycle accesses, parity for read data should be returned in the same cycle that stall is deasserted.

7.2.3 Timing Considerations

The SPRAM interface, unlike the other external interfaces on an microAptiv UP 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 7.1 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 7.1 can be relaxed.

7.2.4 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 7.2.2, "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.

7.2.5 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.

7.2.6 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 7.1 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.

7.2.7 Access Granularity

The widths of the data bus for read and write requests to SPRAM are shown in Table 7.2. 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 microMIPS instruction fetches.

Table 7.2 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 7.3; when a bit in *DSP_DataWrMask* is high, the corresponding byte from *DSP_DataWrValue* should be written to the array.

Table 7.3 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]

7.2.8 Write Strobe with 0 Write Mask

On the DSPRAM interface, the write strobe must be qualified with the write mask. It is possible for the write strobe to be asserted with a value of all 0's on the write mask. Not qualifying the write strobe allows it to come out a little earlier in the cycle. With typical byte-write SRAMs, it is legal to send a write strobe with a 0 mask, but it may consume additional power. To minimize power consumption or connect to atypical devices, these signals can be gated within the SPRAM module.

7.2.9 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 7.2. 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.

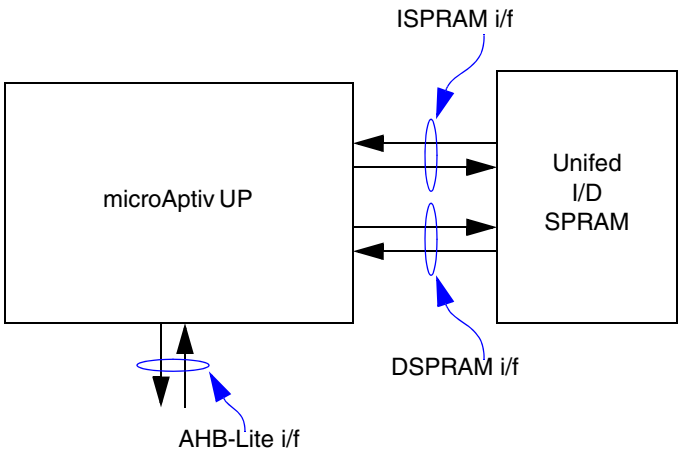


Figure 7.2 Unified I/D SPRAM Block Diagram

7.2.10 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.

7.2.11 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 implementor must ensure that the behavior of the I/O devices meets the same requirements as the SPRAM. In particular, 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 7.2.10, "Restartability of SPRAM Accesses".

7.2.12 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.

7.3 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 7.4 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 7.4 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 7.6, "Using the Same Design for ISPRAM and DSPRAM" on page 94.

7.3.1 Single Read

Figure 7.3 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 microAptiv UP 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.

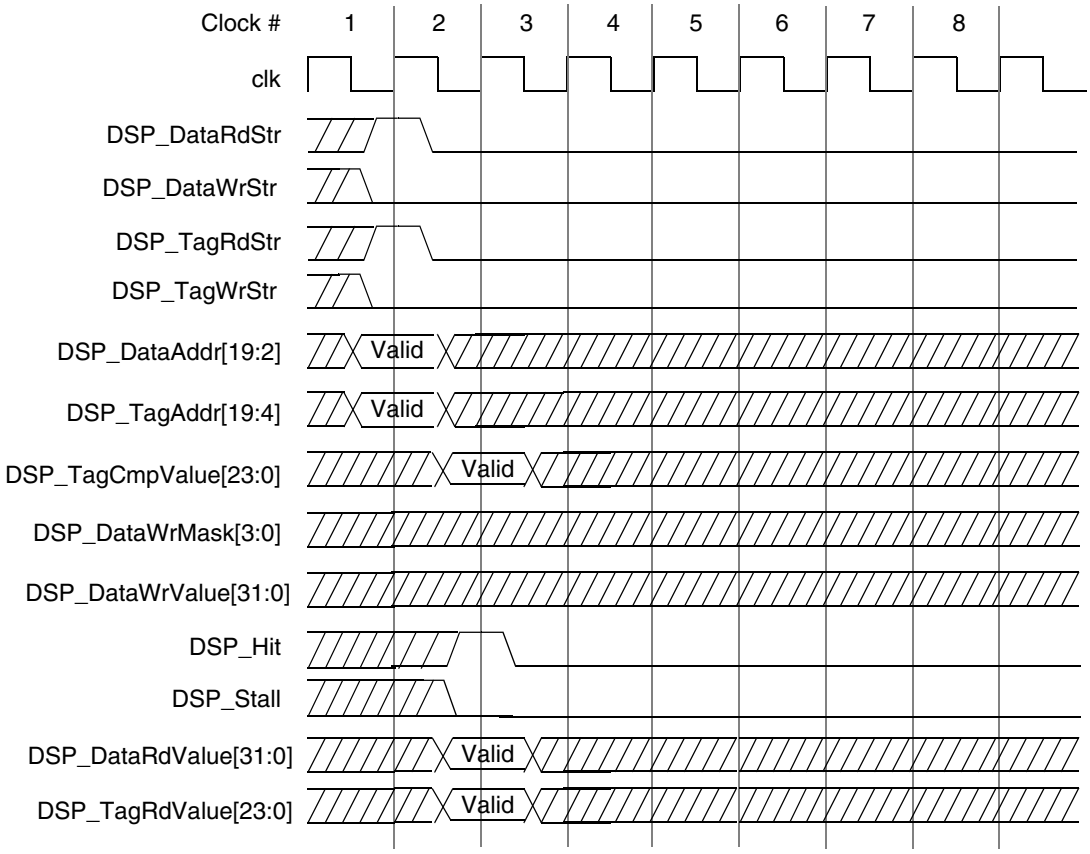


Figure 7.3 Single DSPRAM Read

7.3.2 Single Multi-Cycle Read

Figure 7.4 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 7.3.1, "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.

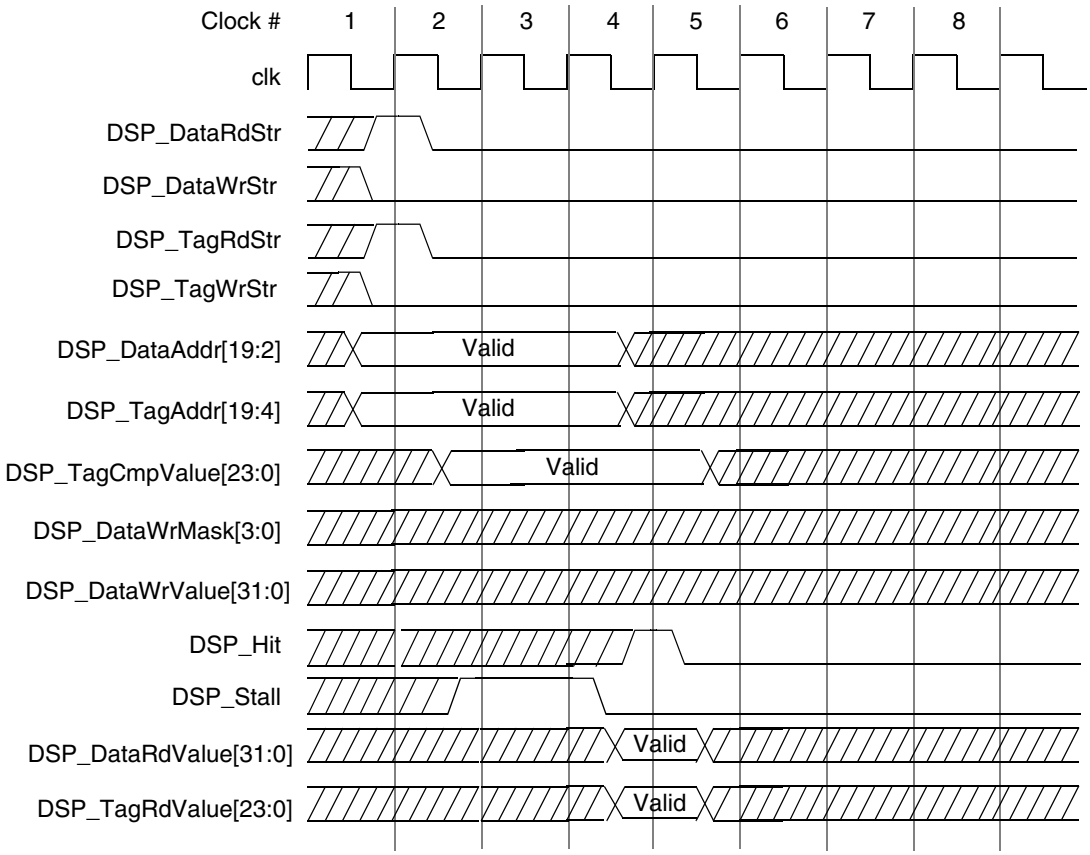


Figure 7.4 Single Multi-Cycle DSPRAM Read

7.3.3 Single Write

Figure 7.3 shows a timing diagram for a single store to the DSPRAM. In cycle 1, a tag lookup is initiated to determine if the store is writing to the DSPRAM. In cycle 2, the DSPRAM indicates that the access did hit via *DSP_Hit*. The store data is held in an internal store buffer and is written to the DSPRAM at a later time when the data port is free. This can occur as soon as cycle 2, but could also be delayed for any number of cycles due to other activity. In this waveform, *DSP_DataWrStr* becomes active in cycle 3 to write the data into the SPRAM array.

This shows a write due to a store instruction. If the write was caused by a CACHE instruction, the data write in cycle 3 could occur without the initial tag read.

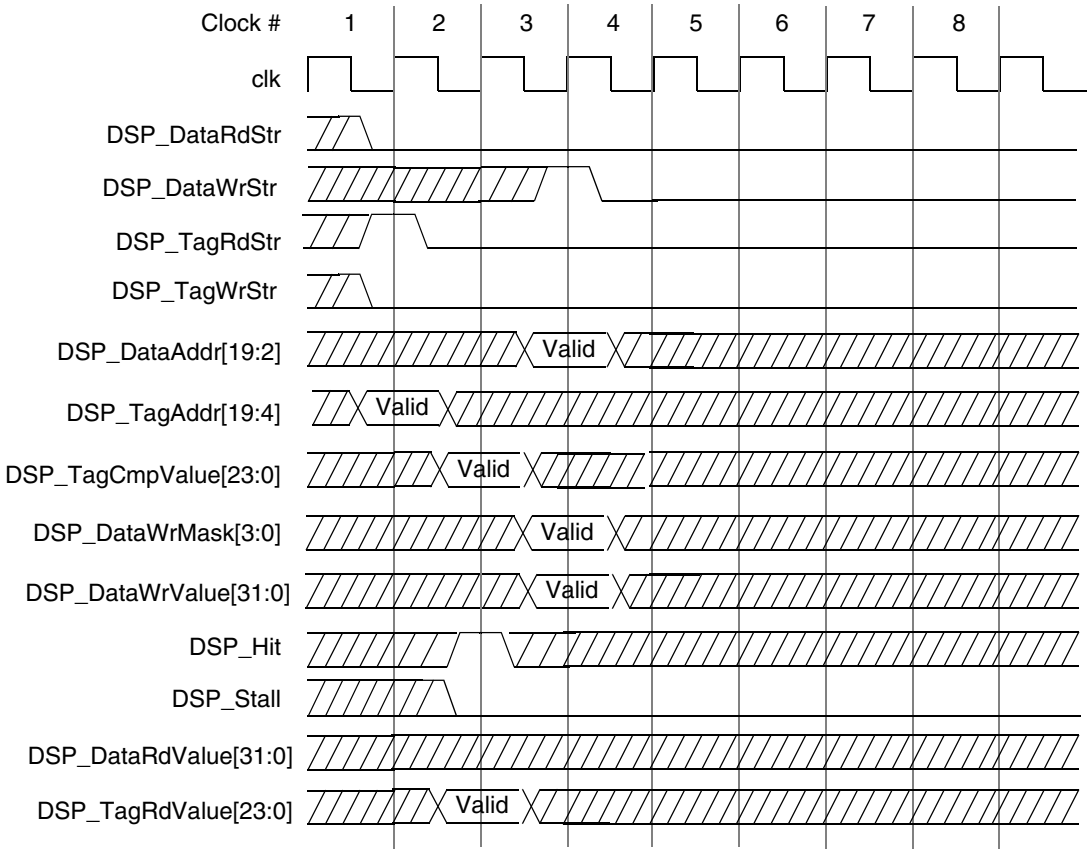


Figure 7.5 Single DSPRAM Write

7.3.4 Single Multi-Cycle Write

Figure 7.6 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 7.3.3, "Single Write", but now the external SPRAM logic was unable to immediately service either the tag read or the data write associated with the write request.

The core initiates the tag read in cycle 1, by driving the tag read strobe and the address. In cycle 2, the core drives the tag compare data, but the external DSPRAM logic is unable to complete the lookup, so it asserts *DSP_Stall*. The tag lookup continues until *DSP_Stall* is deasserted in cycle 4. A hit is indicated and the core immediately begins the data write phase by driving the data write strobe, index address, store data, and byte mask. Again, the external DSPRAM logic is unable to process the write during cycle 5, 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 7, the write could proceed, and the external logic then deasserts stall to complete the write transaction.

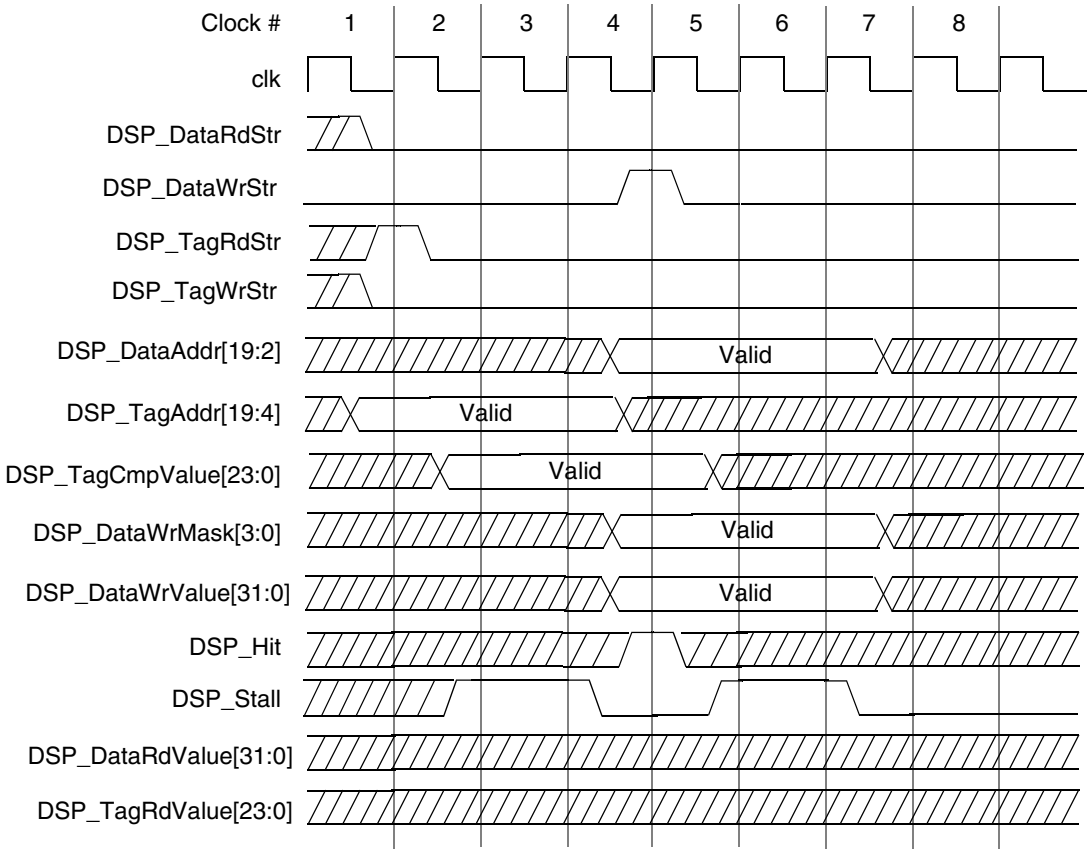


Figure 7.6 Single Multi-Cycle DSPRAM Write

7.3.5 Simultaneous Tag Read and Data Write

Table 7.7 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 7.3.3, "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.

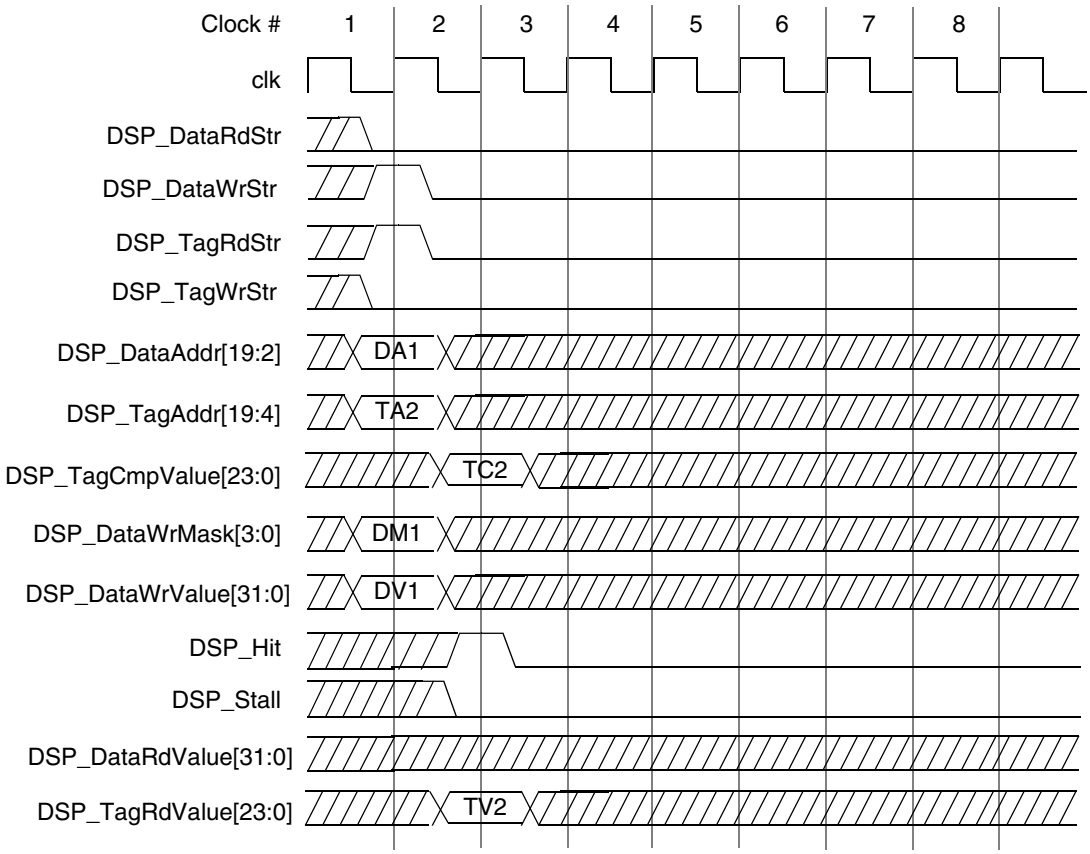


Figure 7.7 Combined DSPRAM Tag Read and Data Write

7.3.6 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 7.8 shows two back-to-back read transactions. Each individual transaction looks like the single-cycle read introduced in Section 7.3.1, "Single Read".

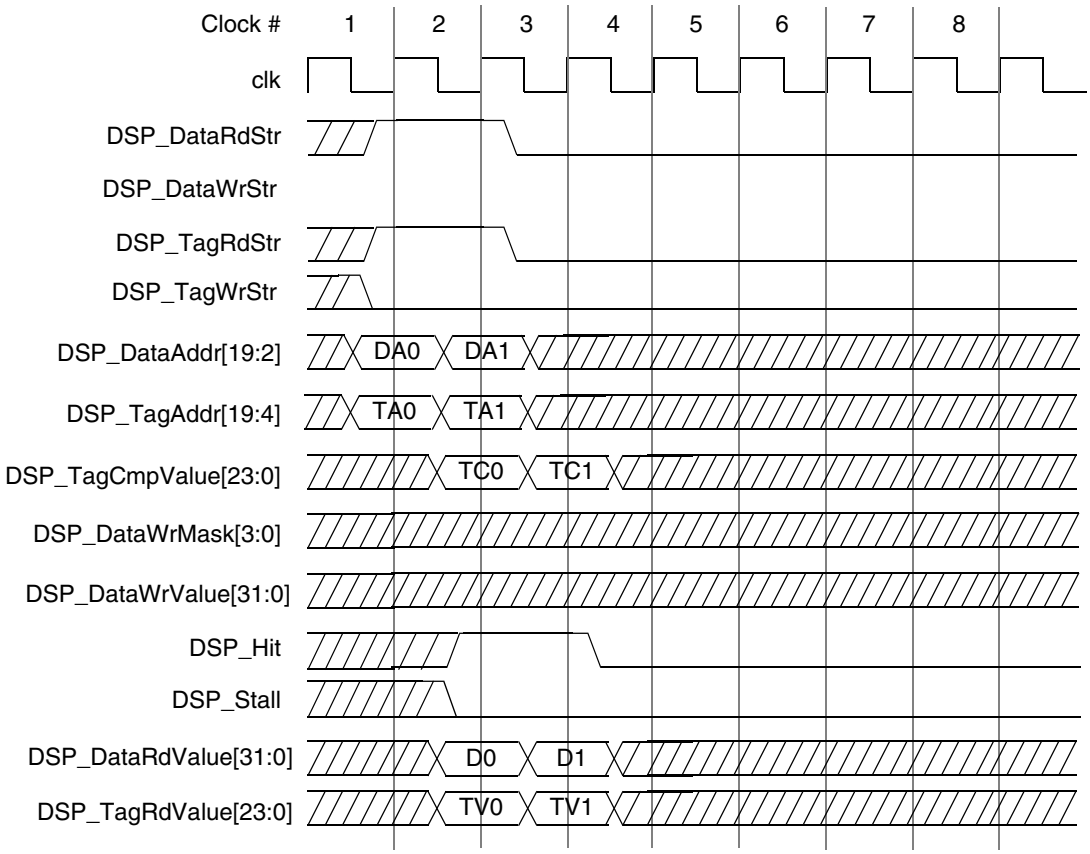


Figure 7.8 Consecutive DSPRAM Reads

7.3.7 Read-Write-Read Sequence

Figure 7.9 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 7.3.1, "Single Read". The data store is derived from the single-cycle transaction introduced in Section 7.3.5, "Simultaneous Tag Read and Data Write", but the completion has been stalled for two additional cycles.

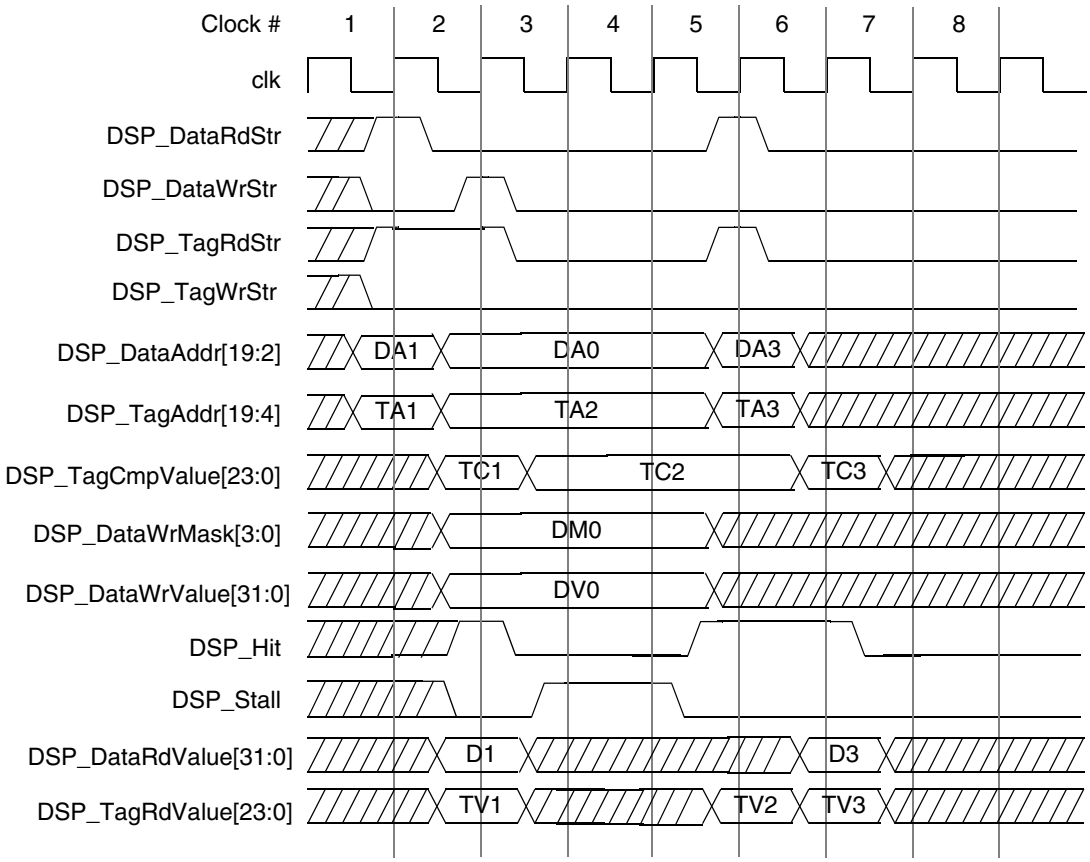


Figure 7.9 Read-Write-Read

7.3.8 Read-Modified-Write Sequence (Locked transfers)

7.3.8.1 RMW Operation Hit in DSPRAM

Figure 7.10 shows a locked transfer for an RMW sequence caused by an atomic instruction accessing DSPRAM. The read request of the atomic instruction is initiated by the core in cycle 1 by driving read strobes and index addresses. In the same cycle, *DSP_Lock* is asserted to lock the DSPRAM access. At cycle 4, the SPRAM array access proceeds, and external logic asserts hit and drives the requested read data. Then the write request of the atomic instruction is sent to the DSPRAM interface by driving the write strobes and index addresses in cycle 5. *DSP_Lock* is held until the write strobe is asserted.

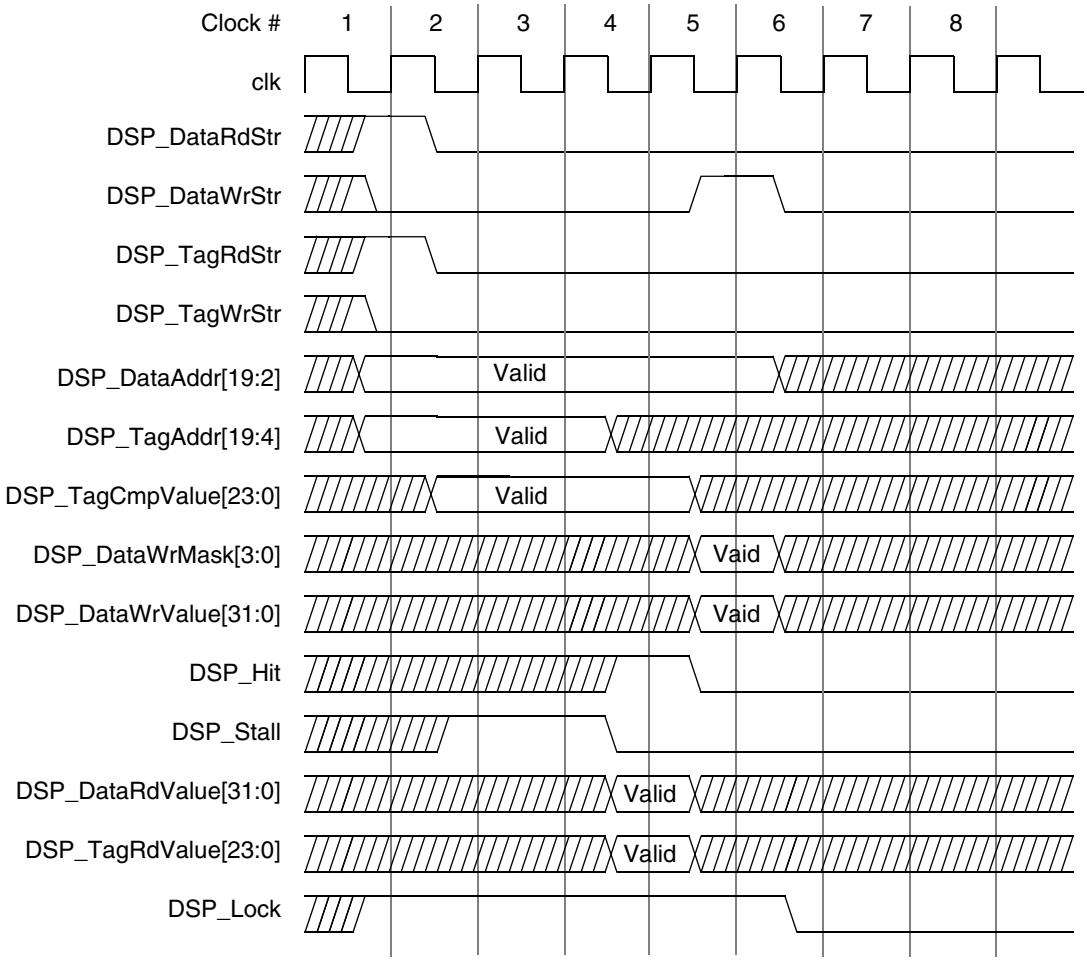


Figure 7.10 A complete RMW operation

7.3.8.2 A Store Operation to DSPRAM in the RMW Sequence

As shown in Figure 7.11, the write operation is normally buffered in the store buffers instead of immediately being sent out on the bus. So it is possible to have a write operation in the middle of a RMW sequence. In cycle 4, a store instruction prior to the atomic instruction has its write access to the DSPRAM in the middle of the RMW sequence.

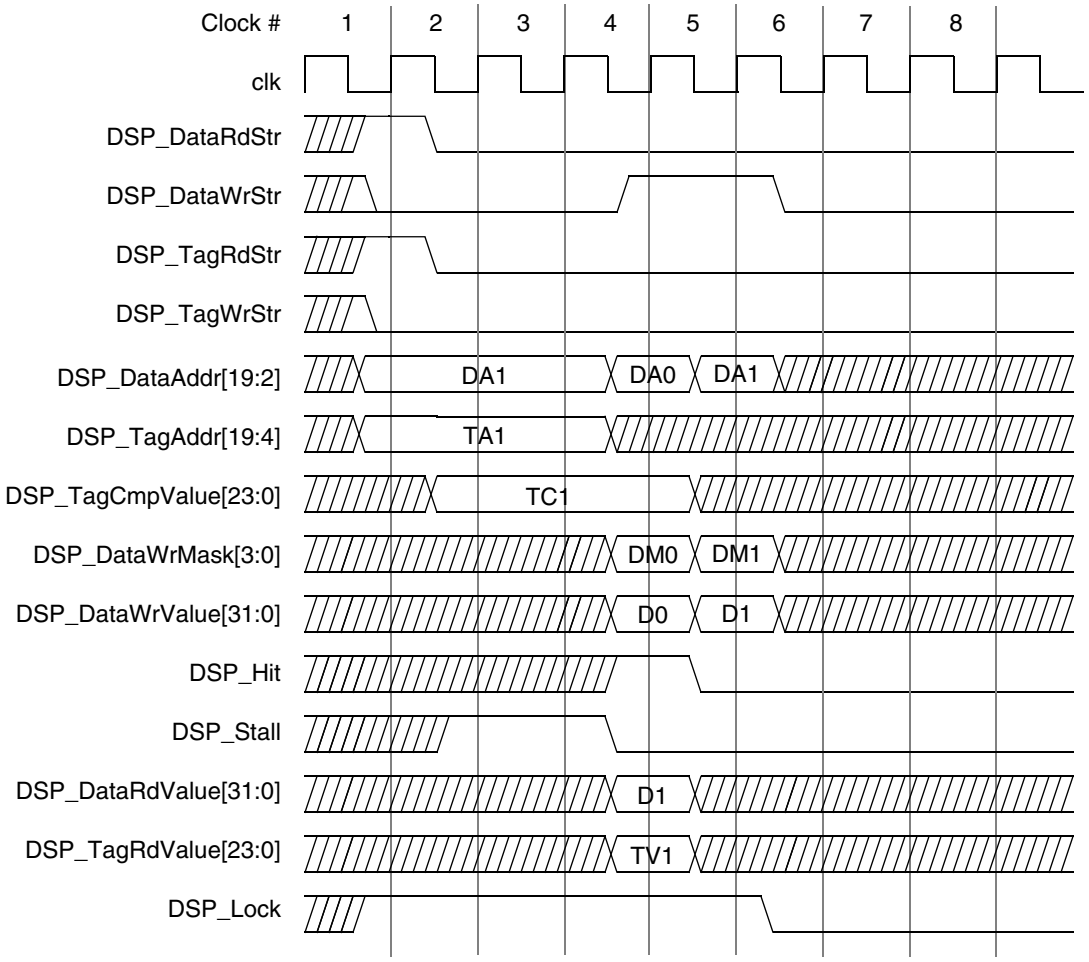


Figure 7.11 A Store operation followed by an atomic operation in DSPRAM access

7.3.8.3 RMW operation does not hit in DSPRAM

The lock sequence is terminated when the read operation of a RMW sequence is missed in the DSPRAM as shown in Figure 7.12. *DSP_Lock* signal is deasserted when DSPRAM deasserts both *DSP_HIT* and *DSP_Stall* in cycle 4.

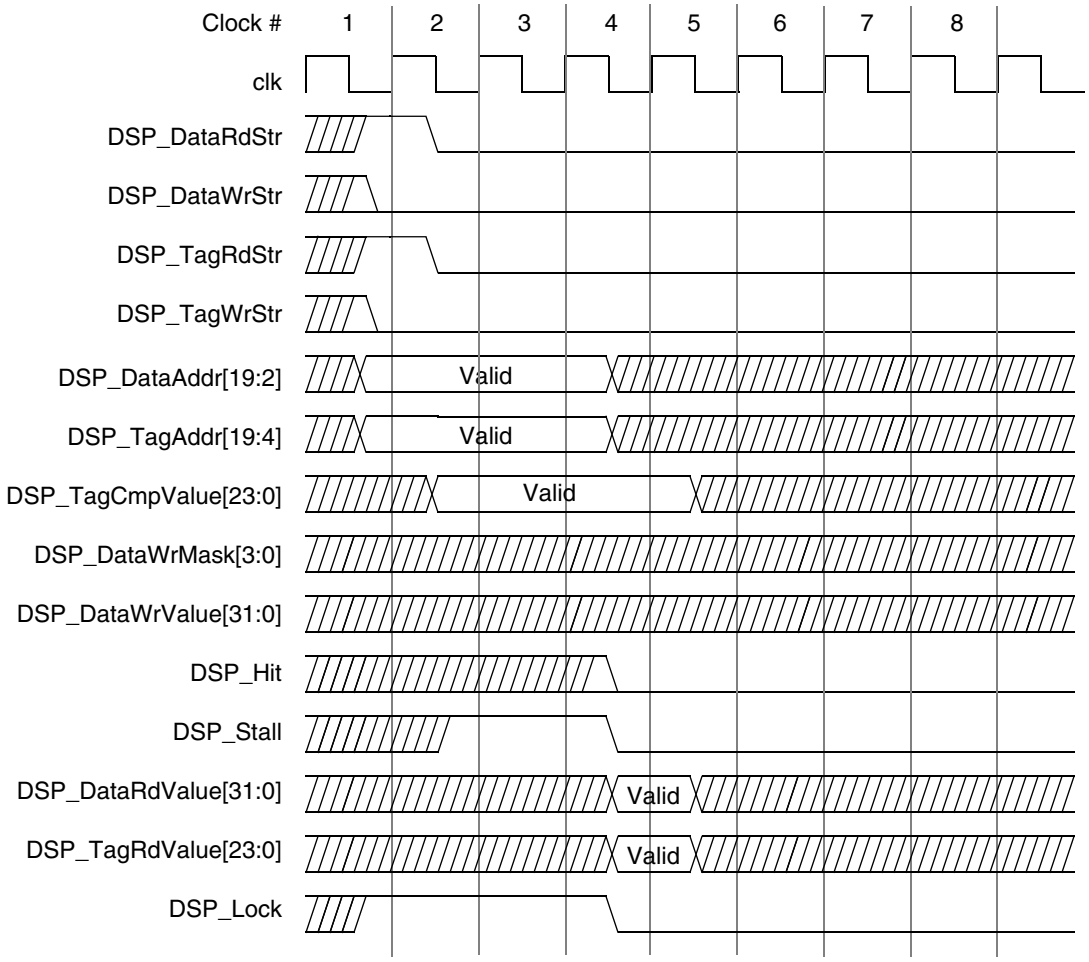


Figure 7.12 RMW Operation does not hit in DSPRAM

7.4 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 back-stalling capability of the SPRAM interface to allow arbitration between the core and a back-door to a single-ported SRAM, as shown in Figure 7.13. The arbitration logic can back-stall the core by asserting *{I,D}SP_Stall* 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.

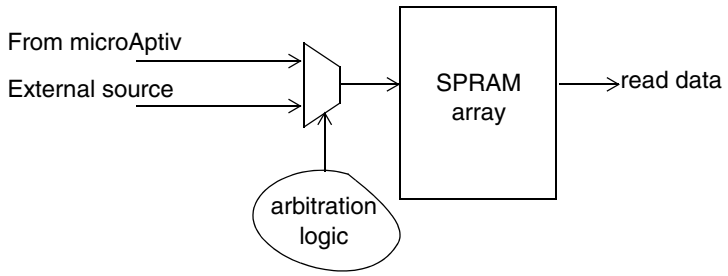


Figure 7.13 External Access to Single-ported SPRAM

7.5 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 AHB-Lite 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.

7.6 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 7.5 shows how ISPRAM signals should be connected to appropriate DSPRAM ports.

Table 7.5 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]	
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.

7.7 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 7.14 shows 3 regions within a single memory array. Several of the bits of the VA are fixed for each region. Figure 7.15 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 7.14 Multiple SPRAM Regions

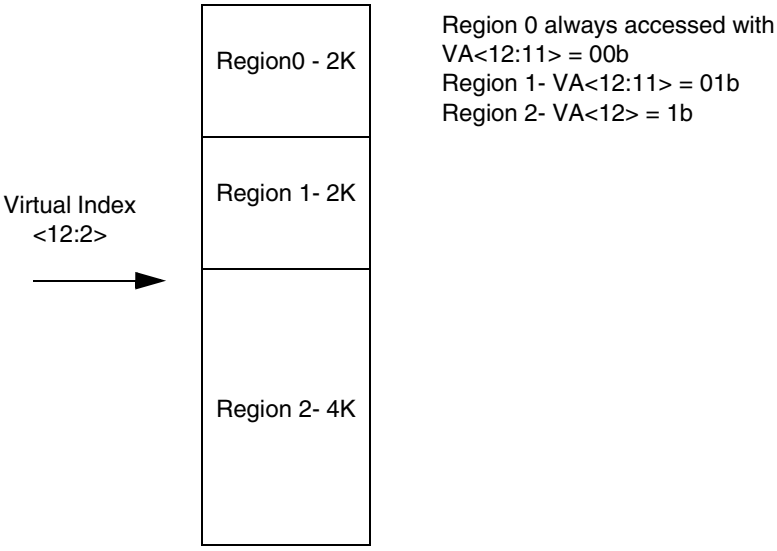
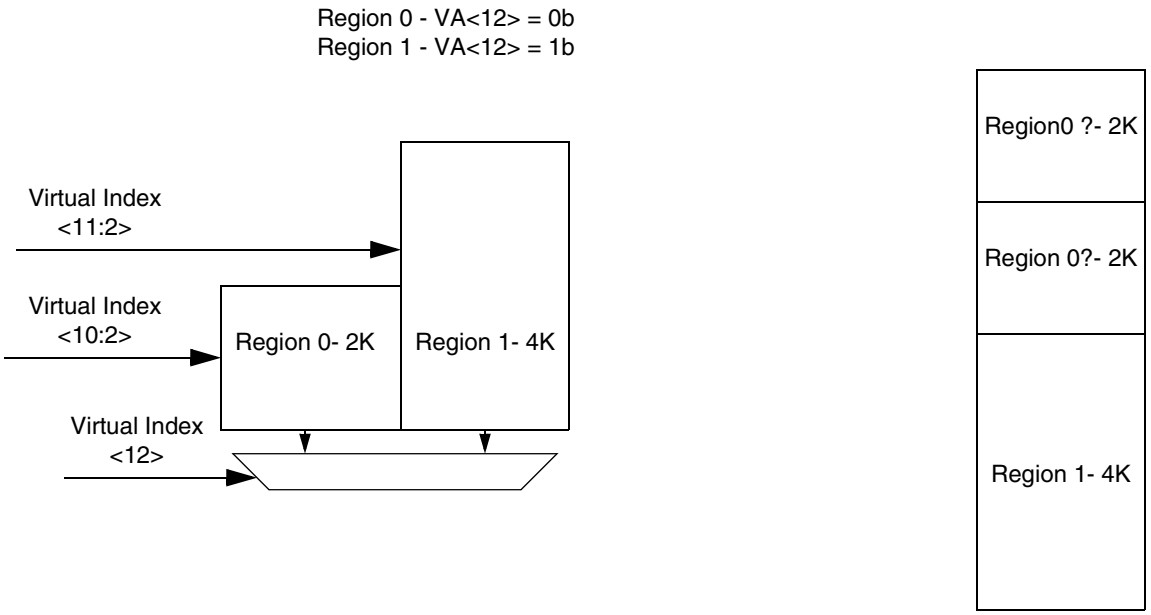


Figure 7.15 Multiple SPRAM Regions in Separate Arrays



7.8 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.

This section provides examples of configuring the SPRAM. To obtain the best optimized results, users must develop an SPRAM design specific to the application .

7.8.1 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
[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)
[0] 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.

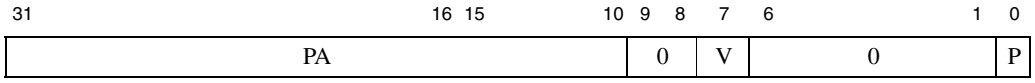
Here's an example showing the tag registers associated with 3 discontiguous SPRAM regions:

```
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**



7.8.2 Region Sizes

Note that the encoding described in Section 7.8.1, "Software-visible Configuration Information" imposes restrictions on the size of memory regions within the SPRAM. Also, if integrated BIST is configured for SPRAM, a minimum of 4KB size SPRAM is required.

7.8.3 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.

7.8.4 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.

7.8.5 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.

7.8.6 SPRAM Parity Support

Parity protection is optionally enabled for SPRAM. A parity error on a SPRAM read will cause a CacheErr exception (for a load or fetch). The CacheErr parity error detection logic resides in the core. For the reference design, if parity is enabled, it must be supported by the instruction cache, data cache and both SPRAM arrays.

From the reference SPRAM module, the outputs DSP_ParPresent and ISP_ParPresent indicate whether each SPRAM array is parity protected. If a custom SPRAM module is built, users might choose not to check parity for SPRAM even though parity checking for instruction and data caches is enabled; in this case, the output should be de-asserted and no parity checking will be done for that SPRAM.

Chapter 8

Clocking, Reset, and Power

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

This chapter contains the following sections:

8.1 Clocking

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

8.1.1 SI_ClkIn Clock

SI_ClkIn is the primary 1x input clock to the microAptiv UP 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 microAptiv UP 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 microAptiv UP core implementation.

8.1.2 EJ_TCK Clock

EJ_TCK is an optional 1x clock input to the microAptiv UP 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.

8.1.3 Handling Clock Insertion Delay

When an microAptiv UP 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 microAptiv UP core implementation.

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

  Similar to *SI_ClkOut*, the other output reference clocks of the microAptiv UP core, *HCLK* (for external AHB-Lite bus interface logic) and *UDI_gclk* (for external UDI logic), have the same de-skew usage and properties. These clocks are gated off from the internal *SI_ClkIn* during "Sleep" mode for power-saving purposes.

8.2 AHB Bus Clock

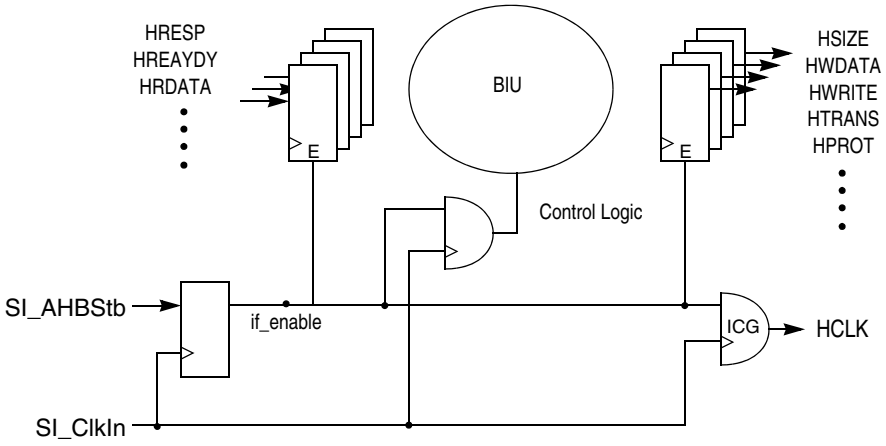
The AHB bus interface on the microAptiv UP core runs synchronously to *SI_ClkIn* but can run at the same or lower clock ratio, such as 2:1, 3:1 and 4:1. An AHB primary input clock is not actually present on the core; instead, AHB interface flops are clocked by *SI_ClkIn* with an enable for driving outputs and sampling inputs on the appropriate *SI_ClkIn* clocks.

8.2.1 SI_AHBStb to enable lower AHB Bus Clock Ratio

The core has one input, *SI_AHBStb,* for controlling the clock ratio. It is registered prior to use as an enable to clock the registers of the AHB input and output interface signals. It enables the core flops of the AHB interface input signals, capturing the values driven from the system. Also it enables the core flops on the AHB interface data output, driving new values on to the bus.

The output clock *HCLK* shows the reference AHB clock used by the microAptiv UP to the SOC designer. Figure 8.1 shows this signal in relation to the overall AHB I/O logic.

Figure 8.1 SI_AHBStb enables AHB bus clock ratio



8.2.2 Waveforms and Timing Requirements for fixed AHB Clock Ratios

The clocking scheme can support a lower AHB bus clock than the core clock, such as 2:1, 3:1 and 4:1. This section includes waveforms showing the behavior and strobing for the 1:1, 2:1, 3:1, and 4:1 ratios.

Figure 8.2 Waveform for 1:1 Clock Ratio

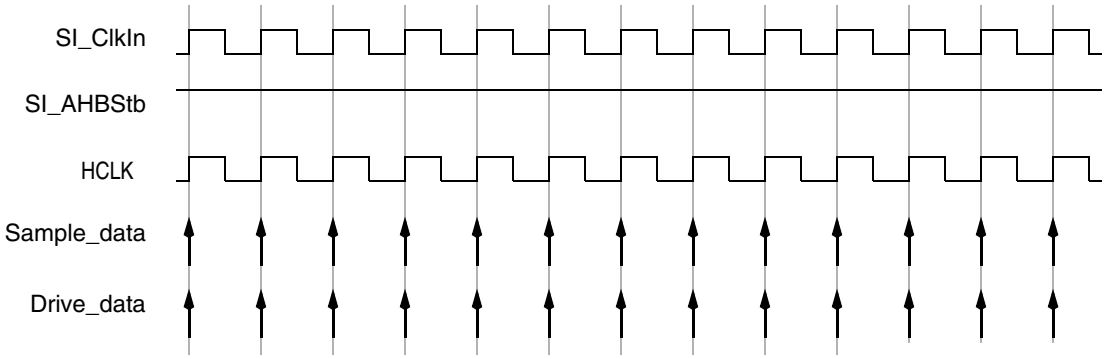


Figure 8.3 Waveform for 2:1 Clock Ratio

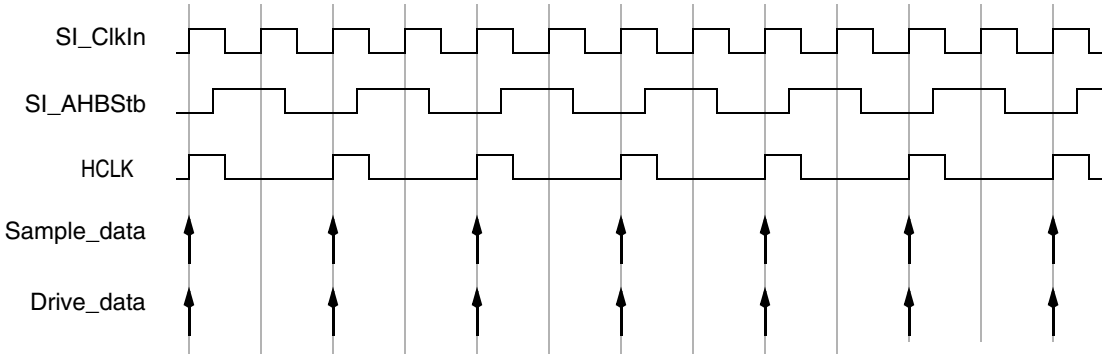


Figure 8.4 Waveform for 3:1 Clock Ratio

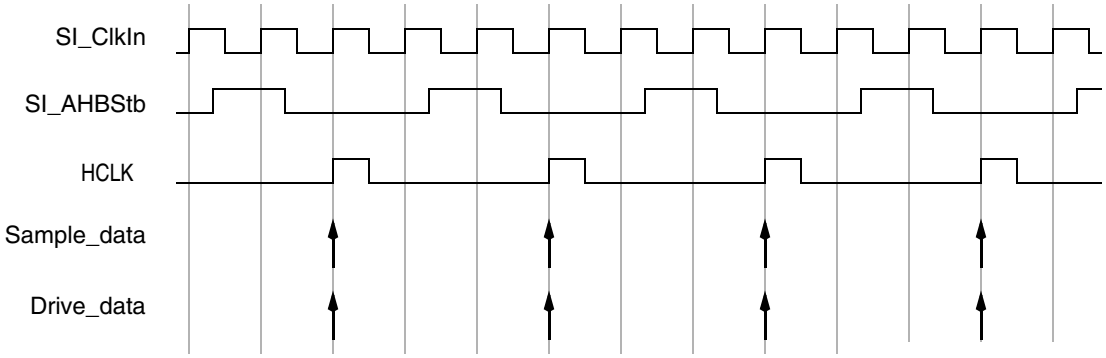
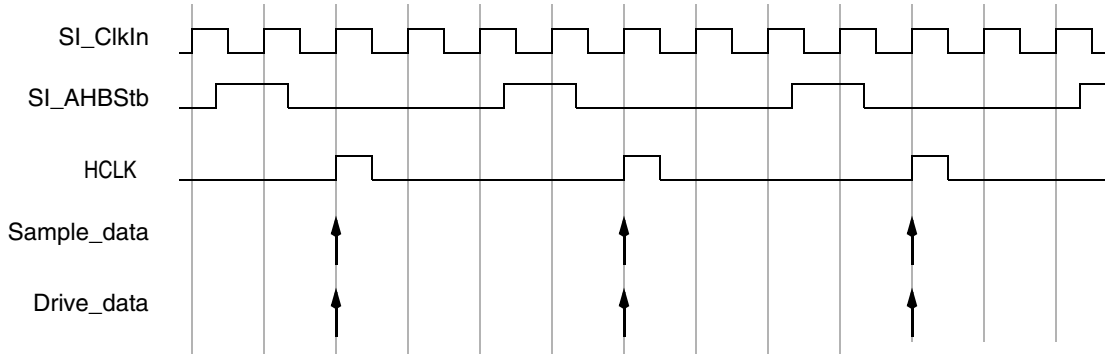


Figure 8.5 Waveform for 4:1 Clock Ratio



8.2.3 System Static Timing Analysis for AHB Clock Domain

The design is always synthesized to meet the tightest timing constraints from the 1:1 clock ratio. This allows it to be used with any clock ratio, and the tight constraints can help meeting system timing requirements.

8.3 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 microAptiv UP 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 microAptiv UP core, and not left floating or indeterminate. Each of the reset-related *SI_** inputs triggers a different type of exception within the microAptiv UP core; the *MIPS32® microAptiv™ UP Processor CoreSoftware User's Manual* [3] describes more details about these exceptions.

The initialization process for an microAptiv UP 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 [3] for a description of the software initialization requirements of an microAptiv UP core.

8.3.1 SI_ColdReset

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

8.3.2 SI_Reset

The high-active *SI_Reset* input is a warm reset input to the microAptiv UP 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 microAptiv UP 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 *StatusSR* field to identify a soft reset exception.

8.3.3 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.

8.3.4 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.

8.4 Power Management

Two primary mechanisms exist for managing system power with an microAptiv UP 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.

8.4.1 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 microAptiv UP core is not in use, if desired by your system logic. The microAptiv UP 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 microAptiv UP 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 microAptiv UP 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 only if the clock frequency is more than 0 MHz (namely, *SI_ERL*/*SI_EXL* do not operate on free-running clocks). *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 when other system logic is available to indicate that the microAptiv UP core is not being used.

8.4.2 Software-Induced Sleep Mode

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

Chapter 9

Design For Test Features

This chapter describes the Design For Test (DFT) features of the MIPS32 microAptiv UP 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 9.1 "Introduction"
- Section 9.2 "Scan Test"
- Section 9.3 "Integrated RAM BIST"
- Section 9.4 "User-Specific RAM BIST"

9.1 Introduction

An implementation of an microAptiv UP 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 9.1 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 9.1 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

Table 9.1 Core Input Values for Major Operating Modes (Continued)

Input Pin	Mode			
	Normal (non-test)	Scan	Integrated BIST	User-specified BIST
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.

9.2 Scan Test

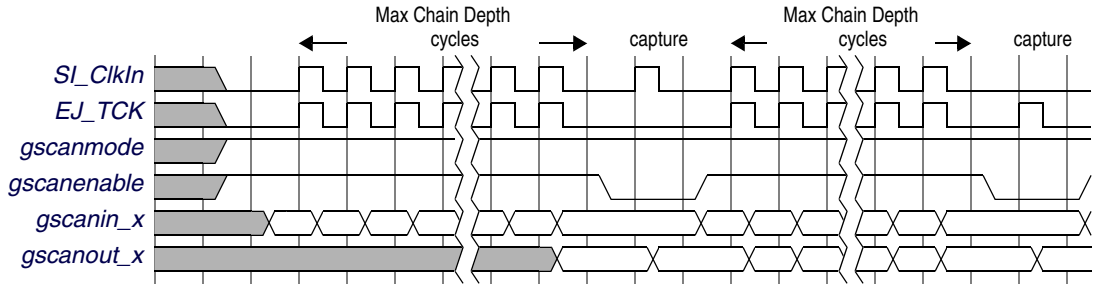
The scan methodology normally used on an microAptiv UP core is muxed scan. The exact scan functionality is dependent on the choices made when the core was created. Specific details about scan operation are therefore implementation-dependent and beyond the scope of this document, but a few general comments 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 microAptiv UP 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 9.1. 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 9.1 Timing Diagram of Typical Scan Chain and Capture Operation



9.3 Integrated RAM BIST

The microAptiv UP 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 selected by an input pin.

9.3.1 RAM BIST-related Interface Signals

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

9.3.1.1 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.

9.3.1.2 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*.

9.3.1.3 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.

9.3.1.4 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.

9.3.1.5 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 9.2. When *gmbinvoke* is deasserted, all fail signals are deasserted in the following cycle.

Table 9.2 Fail Signals

Fail Signals	Instruction Cache				Data Cache			
	Data Memory	Tag Memory	Way-Select Memory	ISPRAM Memory	Data Memory	Tag Memory	Way-Select Memory	SPRAM Memory
gmbdifail	X							
gmbtifail		X						
gmbwifail			X					
gmbispfail				X				
gmbddfail					X			
gmbtdfail						X		
gmbwdfail							X	
gmbspfail								X

9.3.1.6 gscanenable

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

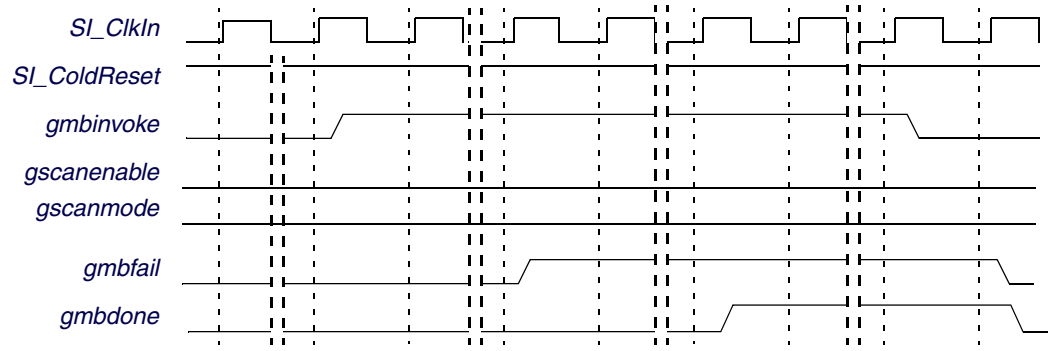
9.3.1.7 gscanmode

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

9.3.2 RAM BIST Signal Waveform for a Memory Test

A diagram with the timing of an integrated memory BIST sequence is shown in Figure 9.2.

Figure 9.2 RAM BIST I/O Signals Timing



9.3.3 Number of Cycles for Memory BIST

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

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

For the March C+ algorithm, NumberofOperations per bit is 14. For the IFA-13 algorithm, NumberofOperations per bit is 16. WaySize is different with and without Parity, so the number of Cylcles will be different too.

9.4 User-Specific RAM BIST

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

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

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

Appendix A

References

This appendix lists other publications 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 microAptiv UP soft or hard core release, or be available on the MIPS web site, under http://www.mips.com/publications/index.html.

1. MIPS® Physical Design Guide
   MIPS document: MD00606

2. MIPS32® microAptiv™ UP Processor Core Family Data Sheet
   MIPS document: MD00939

3. MIPS32® microAptiv™ UP Processor Core Family Software User's Manual
   MIPS document: MD00942

4. MIPS32® microAptiv™ UP Processor Core Family Implementor's Guide
   MIPS Document: MD00940

5. MIPS32® microAptiv™ UP Processor Core Family System Package & Simulation Flow User's Manual
   MIPS document: MD00943

6. EJTAG Specification
   MIPS document: MD00047

7. MIPS® cJTAG Adapter User's Manual
   MIPS Document: MD00862

8. MIPS® iFlowtrace Architecture Specification
   MIPS document: MD00526

9. Core Coprocessor Interface Specification
   MIPS document: MD00068

10. MIPS32® Architecture For Programmers Volume III: The MIPS32® Privileged Resource Architecture
    MIPS document: MD00090

11. Core Coprocessor 2 Module Template Application Note
    MIPS document: MD00130

12. MIPS32® Pro Series® CorExtend® Instruction Integrator's Guide
    MIPS document: MD00324

13. MINI Testbench Specification
    MIPS document: MD00493

14. Security Features of the M14K™ Processor Family

Appendix B

Revision History

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

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

Revision	Date	Description
01.00	July 31, 2013	• Initial 3_0_0 General Availability release.
01.01	July 30, 2014	• Update SP_DataTagValue signal. • Changes to timer interrupt input.