Core Coprocessor Interface Specification

Document Number: MD00068
Revision 02.11
July 8, 2009

MIPS Technologies, Inc.
955 East Arques Avenue
Sunnyvale, CA 94085-4521

Template: nB1.03, Built with tags: 2B

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This document describes the Coprocessor Interface standard supported by various MIPS® processor cores. The Coprocessor Interface is designed to enable coprocessors, such as FPUs and Graphics Engines, to be tightly coupled to an integer processor core. Such coprocessors can be internally developed by MIPS Technologies or externally developed by customers or third party design teams.

Note: For clarity, the term *integer processor core* describes the MIPS processor core to which a coprocessor attaches. The integer processor core can do more than integer processing, however. In fact, it can have an internal FPU (and use the Coprocessor Interface for COP2). By the same token, the coprocessor can itself do any kind of processing, including integer calculations.

The Coprocessor Interface has the following features:

- The interface is easy to understand. By keeping the interface as simple as possible, designers can concentrate on the coprocessor's functionality rather than its interface.
- Performance is not compromised. The Coprocessor Interface is compatible with the high-performance features of MIPS microprocessor cores.
- Minimal interface logic is required, which reduces area and power overhead.
- The interface is highly configurable:
  - 32-bit or 64-bit data transfers
  - COP1 and/or COP2 supported
  - From 0 to 7 out-of-order data transfers
  - Single issues up to eight issues supported
  - Support for multithreading
- A coprocessor built for a low-performance integer processor core can be connected to higher performance integer processor cores. Furthermore, a high-performance coprocessor can be connected to a lower-performance integer processor core.

This document contains the following sections:

- Chapter 2, "Coprocessor Instructions" on page 9 describes the specific instructions supported by the Coprocessor Interface.
- Chapter 3, "Signal Descriptions" on page 13 describes the signals that make up the interface.
- Chapter 4, "Configurations" on page 21 describes the configuration options available with the Coprocessor Interface.

- Chapter 5, “Interface Protocols” on page 27 describes the cycle-by-cycle behavior of the signals.

Chapter 2

Coprocessor Instructions

The Coprocessor Interface supports all coprocessor instructions currently defined in the MIPS32®, MIPS64®, and MIPS-3D® architecture specifications.

These coprocessor instructions are divided into three classes.

- Instructions that perform arithmetic operations (called *Arithmetic COP Ops*)
- Instructions that move data into the Coprocessor (called *To COP Ops*)
- Instructions that move data out of the Coprocessor (called *From COP Ops*)

The explicit classification of the opcodes is given below. For a detailed description of these instructions, refer to the MIPS ISA definition or to the Software User's Manual of the appropriate integer processor core.

Arithmetic COP Ops:

- COP1 arithmetic instructions (including COP1X and MDMX instructions)
  - IR[31:26] = 010001 AND IR[25] = 1
  - IR[31:26] = 010011 AND IR[5:4] != 00
  - IR[31:26] = 011110
- COP2 arithmetic instructions
  - IR[31:26] = 010010 AND IR[25] = 1
- COP1 branch instructions (BC1 instructions)
  - IR[31:26] = 010001 AND IR[25:24] = 01
- COP2 branch instructions (BC2 instructions)
  - IR[31:26] = 010010 AND IR[25:24] = 01
- Conditional COP1 movement instructions (MOVF, MOVT instructions)
  - IR[31:26] = 000000 AND IR[5:0] = 000001

The following COP1 arithmetic instructions test coprocessor condition bits:

- BC1, BC2, MOVF and MOVT (as defined above)

Following COP1 arithmetic instructions test integer processor core registers:

- ALNV.PS
  - IR[31:26]=010011 AND IR[5:0]=011110
- ALNV.OB ALNV.QH
  - IR[31:26]=011110 AND IR[5:2]=0110 AND IR[0]=1
- MOVN.S MOVZ.S MOVN.D MOVZ.D MOVN.PS MOVZ.PS

- IR[31:26]=010001 AND IR[25:21]=10000 AND IR[5:1]=01001
- IR[31:26]=010001 AND IR[25:21]=10001 AND IR[5:1]=01001
- IR[31:26]=010001 AND IR[25:21]=10110 AND IR[5:1]=01001

For the remainder of this document, the terms “Arithmetic COP Op” and “arithmetic instruction” are used interchangeably.

From COP Ops:

- COP1 From instructions (including COP1X instructions)
  - IR[31:26] = 111001
  - IR[31:26] = 111101
  - IR[31:26] = 010001 AND IR[25:23] = 000
  - IR[31:26] = 010011 AND IR[5:3] = 001 AND IR[2:0] !=111
  - IR[31:26] = 010000 AND IR[25:21] = 01000 AND IR[5] = 1 AND IR[2:1] = 01
- COP2 From instructions
  - IR[31:26] = 111010
  - IR[31:26] = 111110
  - IR[31:26] = 010010 AND IR[25:23] = 000
  - IR[31:26] = 010000 AND IR[25:21] = 01000 AND IR[5] = 1 AND IR[2:1] = 10

Of the above defined *From COP Ops*, following are 32-bit instructions

- MFC1, CFC1, SWC1, SWXC1
  - IR[31:26] = 010001 AND IR[25:23]=000 AND IR[21]=0
  - IR[31:26]=111001
  - IR[31:26]= 010011 AND IR[5:0]=001000
- MFHC1 (MIPS32 Release 2 only)
  - IR[31:26] = 010001 AND IR[25:21]=00011
- MFC2, CFC2, SWC2
  - IR[31:26] = 010010 AND IR[25:23]=000 AND IR[21]=0
  - IR[31:26]=111010
- MFHC2 (MIPS32 Release 2 only)
  - IR[31:26] = 010010 AND IR[25:21]=00011
- MFTR (MT-ASE only)
  - IR[31:26] = 010000 AND IR[25:21] = 01000 AND IR[5] = 1 AND IR[2:1] = 01
  - IR[31:26] = 010000 AND IR[25:21] = 01000 AND IR[5] = 1 AND IR[2:1] = 10

Of the above defined *From COP Ops*, the following are 64-bit instructions

- DMFC1, SDC1, SDXC1, SUXC1
  - IR[31:26] = 010001 AND IR[25:21]=00001
  - IR[31:26]=111101
  - IR[31:26]= 010011 AND IR[5:3]=001 AND IR[1:0]=01
- DMFC2, SDC2

- IR[31:26] = 010010 AND IR[25:21]=00001
- IR[31:26]=111110

The remaining instructions are reserved opcodes.

To COP Ops:

- COP1 To instructions (including COP1X instructions)
  - IR[31:26] = 110001
  - IR[31:26] = 110101
  - IR[31:26] = 010001 AND IR[25:23] = 001
  - IR[31:26] = 010011 AND IR[5:3] = 000
  - IR[31:26] = 010000 AND IR[25:21] = 01100 AND IR[5] = 1 AND IR[2:1] = 01
- COP2 To instructions
  - IR[31:26] = 110010
  - IR[31:26] = 110110
  - IR[31:26] = 010010 AND IR[25:23] = 001
  - IR[31:26] = 010000 AND IR[25:21] = 01100 AND IR[5] = 1 AND IR[2:1] = 10

Of the above defined *To COP Ops*, the following are 32-bit instructions

- MTC1, CTC1, LWC1, LWXC1
  - IR[31:26] = 010001 AND IR[25:23]=001 AND IR[21]=0
  - IR[31:26]=110001
  - IR[31:26]= 010011 AND IR[5:0]=000000
- MTHC1 (MIPS32 Release 2 only)
  - IR[31:26] = 010001 AND IR[25:21]=00111
- MTC2, CTC2, LWC2
  - IR[31:26] = 010010 AND IR[25:23]=001 AND IR[21]=0
  - IR[31:26]=110010
- MTHC2 (MIPS32 Release 2 only)
  - IR[31:26] = 010010 AND IR[25:21]=00111
- MTTR (MT-ASE only)
  - IR[31:26] = 010000 AND IR[25:21] = 01000 AND IR[5] = 1 AND IR[2:1] = 01
  - IR[31:26] = 010000 AND IR[25:21] = 01000 AND IR[5] = 1 AND IR[2:1] = 10

Of the above defined *To COP Ops*, the following are 64-bit instructions

- DMTC1, LDC1, LDXC1, LUXC1
  - IR[31:26] = 010001 AND IR[25:21]=00101
  - IR[31:26]=110101
  - IR[31:26]= 010011 AND IR[5:3]=000 AND IR[1:0]=01
- DMTC2, LDC2
  - IR[31:26] = 010010 AND IR[25:21]=00101

- IR[31:26]=110110

The remaining instructions are reserved opcodes.

Chapter 3

Signal Descriptions

Table 3.5, Table 3.6, and Table 3.7 describe all of the Coprocessor Interface signals. Note that the signals are grouped according to their logical function rather than alphabetically or by their expected physical location. The interactions of signals within these functional groups are described in Chapter 5, “Interface Protocols” on page 27.

A separate clock signal is not included in the Coprocessor Interface. All signals are synchronous to the input clock of the integer processor core.

The following tables describe the various attributes of the signals. Table 3.1 shows the direction of the I/O signal relative to the integer processor core. Table 3.2 describes how the prefix of a signal determines whether it is required for COP1, COP2, or both. Table 3.3 and Table 3.4 describe issue group attributes. For details about the concept of issue groups, see Section 4.4, "Multi-Issue Support" on page 23.

Table 3.1 Signal Direction Key

Dir	Description
In	Input to the integer processor core.
Out	Output of the integer processor core.
SIn	Static Input to the integer processor core. These signals are normally tied to either power or ground.
SOut	Static Output of the integer processor core. These signals are normally tied to either power or ground.

Table 3.2 Signal Coprocessor Category

Prefix	Description
CP_	Required for both COP1 and COP2. Note: These signals may change name to CP1_ or CP2_ when used in certain configurations, refer to sections 4.1.1 through 4.1.4 on page 22.
CP1_	Required for only COP1.
CP2_	Required only for COP2.

Table 3.3 Issue Group Key

Issue Group	Description
Comb	Signal is part of Combined issue groups.
Arith	Signal is part of Arithmetic issue groups.
TF	Signal is part of To/From issue groups.
NONE	Signal is not part of any issue groups.

Table 3.4 Signal Issue Group Number

Suffix	Description
_m	*m* determines to which issue group a signal belongs ($0 \leq m \leq 7$).

Table 3.5 Interface Signal Descriptions (Required for both COP1 and COP2)

Signal Name	Dir	Issue Group	Description
Instruction Dispatch			
CP_ir_m[31:0]	Out	Comb, Arith, TF	**Coprocessor Instruction Word.** This 32-bit bus contains the coprocessor instruction. It is available in the cycle before *CP1_as_m*, *CP2_as_m*, *CP1_ts_m*, *CP2_ts_m*, *CP1_fs_m*, or *CP2_fs_m* is asserted.
CP_tcid_m[7:0]	Out	Comb, Arith, TF	**Coprocessor Instruction TC ID.** This bus indicates which TC the instruction on *CP_ir_m* is for. It is available in the cycle before *CP1_as_m*, *CP2_as_m*, *CP1_ts_m*, *CP2_ts_m*, *CP1_fs_m*, or *CP2_fs_m* is asserted.
CP_vpeid_m[3:0]	Out	Comb, Arith, TF	**Coprocessor Instruction VPE ID.** This bus indicates which VPE the instruction on *CP_ir_m* is for. It is available in the cycle before *CP1_as_m*, *CP2_as_m*, *CP1_ts_m*, *CP2_ts_m*, *CP1_fs_m*, or *CP2_fs_m* is asserted.
CP_targtcid_m[7:0]	Out	Comb, Arith, TF	**Coprocessor Instruction Target TC ID.** This bus indicates which TC the instruction on *CP_ir_m* is accessing. This is valid for MFTR/MTTR instructions which access registers of a TC different from the one executing the instruction. It is available in the cycle before *CP1_as_m*, *CP2_as_m*, *CP1_ts_m*, *CP2_ts_m*, *CP1_fs_m*, or *CP2_fs_m* is asserted.
CP_irenable_m	Out	Comb, Arith, TF	**Enable Instruction Registering.** When this signal is deasserted, no instruction strobes are asserted in the following cycle. When this signal is asserted, there can be an instruction strobe asserted in the following cycle. Instruction strobes include *CP1_as_m*, *CP1_ts_m*, *CP1_fs_m*, *CP2_as_m*, *CP2_ts_m*, *CP2_fs_m*.
CP_order_m[2:0]	Out	Comb, Arith, TF	**Coprocessor Dispatch Order.** This signal signifies the program order of instructions when more than one instruction is issued in a single cycle. Each instruction dispatched has an order value associated with it. There must always be one instruction whose order value is 0. Order values must increment by 1 when more than one instruction is issued in a cycle. This signal is valid when *CP1_as_m*, *CP2_as_m*, *CP1_ts_m*, *CP2_ts_m*, *CP1_fs_m*, or *CP2_fs_m* is asserted.
CP_adisable_m	SIn	Comb, Arith	**Inhibit Arithmetic Dispatch.** When this signal is asserted, the integer processor core is prevented from dispatching an arithmetic instruction using this issue group.
CP_tfdisable_m	SIn	Comb, TF	**Inhibit To/From Dispatch.** When this signal is asserted, the integer processor core is prevented from dispatching a To/From instruction using this issue group.
CP_inst32_m	Out	Comb, Arith, TF	**MIPS32 Compatibility Mode – Instructions.** When this signal is 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. This signal is valid the cycle before *CP1_as_m*, *CP2_as_m*, *CP1_fs_m*, *CP2_fs_m*, *CP1_ts_m*, or *CP2_ts_m* is asserted.

Table 3.5 Interface Signal Descriptions (Required for both COP1 and COP2) (Continued)

<table>
<tr><th>Signal Name</th><th>Dir</th><th>Issue Group</th><th>Description</th></tr>
<tr><td>CP_endian_m</td><td>Out</td><td>Comb, Arith, TF</td><td>Byte Ordering. When this signal is asserted, the processor is using big-endian byte ordering for the dispatched instruction. When this signal is deasserted, the processor is using little-endian byte ordering. This signal is valid the cycle before CP1_as_m, CP2_as_m, CP1_fs_m, CP2_fs_m, CP1_ts_m, or CP2_ts_m is asserted.</td></tr>
<tr><td colspan="4">To Coprocessor Data (For all To COP Ops)</td></tr>
<tr><td>CP_tds_m</td><td>Out</td><td>Comb, TF</td><td>Coprocessor To Data Strobe. This signal is asserted when To COP Op data is available on CP_tdata_m. This signal must not be asserted in the same cycle CP_tdds_m is asserted.</td></tr>
<tr><td>CP_tdk_m</td><td>Out</td><td>Comb, TF</td><td>Coprocessor To Data Kill. This signal is valid when CP_tds_m is asserted. If CP_tdk_m is asserted, the To COP Op was killed and the coprocessor should not writeback CP_tdata_m. If CP_tdk_m is deasserted, the To COP Data transfer completes normally. When CP_tds_m is asserted, CP_tdk_m and CP_tdd_m may not both be asserted at the same time.</td></tr>
<tr><td>CP_tdd_m</td><td>Out</td><td>Comb, TF</td><td>Coprocessor To Data Delayed. This signal is valid when CP_tds_m is asserted. If CP_tdd_m is asserted, the To COP Op data transfer is delayed and CP_tdata_m is invalid. Furthermore, this indicates that a To COP Data (Delayed) transfer will happen. When CP_tds_m is asserted, CP_tdk_m and CP_tdd_m may not both be asserted at the same time.</td></tr>
<tr><td>CP_torder_m[2:0]</td><td>Out</td><td>Comb, TF</td><td>Coprocessor To Order. This signal specifies for which outstanding To COP Op the data is. This signal is valid only when CP_tds_m is asserted.
<table>
<tr><th>CP_torder_m[2:0]</th><th>Order</th></tr>
<tr><td>3’b000</td><td>Oldest outstanding To COP Op data transfer</td></tr>
<tr><td>3’b001</td><td>2nd oldest To COP Op data transfer</td></tr>
<tr><td>3’b010</td><td>3rd oldest To COP Op data transfer</td></tr>
<tr><td>3’b011</td><td>4th oldest To COP Op data transfer</td></tr>
<tr><td>3’b100</td><td>5th oldest To COP Op data transfer</td></tr>
<tr><td>3’b101</td><td>6th oldest To COP Op data transfer</td></tr>
<tr><td>3’b110</td><td>7th oldest To COP Op data transfer</td></tr>
<tr><td>3’b111</td><td>8th oldest To COP Op data transfer</td></tr>
</table></td></tr>
<tr><td>CP_tordlim_m[2:0]</td><td>SIn</td><td>Comb, TF</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 CP_torder_m[2:0].</td></tr>
<tr><td>CP_tdata_m[63:0]</td><td>Out</td><td>Comb, TF</td><td>To Coprocessor Data. Data to be transferred to the coprocessor. For single-word transfers, data is available on CP_tdata_m[31:0]. This bus is valid when CP_tds_m is asserted, CP_tdk_m is deasserted and CP_tdd_m is deasserted. It is also valid when CP_tdds_m is asserted and CP_tddk_m is deasserted.

Note: In 32-bit data transfer size configurations, this bus is reduced to CP_tdata_m[31:0].</td></tr>
<tr><td colspan="4">To Coprocessor Data (Delayed) (For all Delayed To COP Ops)</td></tr>
<tr><td>CP_tdds_m</td><td>Out</td><td>Comb, TF</td><td>Coprocessor To Data (Delayed) Strobe. This signal is asserted when delayed To COP Op data is available on CP_tdata_m. This signal must not be asserted in the same cycle CP_tds_m is asserted.</td></tr>
</table>

Table 3.5 Interface Signal Descriptions (Required for both COP1 and COP2) (Continued)

<table>
<tr><th>Signal Name</th><th>Dir</th><th>Issue Group</th><th>Description</th></tr>
<tr><td>CP_tddk_m</td><td>Out</td><td>Comb, TF</td><td>Coprocessor To Data (Delayed) Kill. This signal is valid when CP_tdds_m is asserted. If CP_tddk_m is asserted, the To COP Op was killed and the coprocessor should not writeback CP_tdata_m. If CP_tddk_m is deasserted, the To COP Data transfer completes normally.</td></tr>
<tr><td>CP_tddtcid_m[7:0]</td><td>Out</td><td>Comb, TF</td><td>Coprocessor To Data (Delayed) TCID. This signal specifies the TC that the To COP Data (Delayed) transfer applies to. This signal is valid only when CP_tdds_m is asserted.</td></tr>
<tr><td colspan="4">From Coprocessor Data (For all From COP Ops)</td></tr>
<tr><td>CP_fds_m</td><td>In</td><td>Comb, TF</td><td>Coprocessor From Data Strobe. This signal is asserted when From COP Op data is available on CP_fdata_m.</td></tr>
<tr><td>CP_forder_m[2:0]</td><td>In</td><td>Comb, TF</td><td>Coprocessor From Order. This signal specifies for which outstanding From COP Op the data is. This signal is valid only when CP_fds_m is asserted.
<table>
<tr><th>CP_forder_m</th><th>Order</th></tr>
<tr><td>3’b000</td><td>Oldest outstanding From COP Op data transfer</td></tr>
<tr><td>3’b001</td><td>2nd oldest From COP Op data transfer</td></tr>
<tr><td>3’b010</td><td>3rd oldest From COP Op data transfer</td></tr>
<tr><td>3’b011</td><td>4th oldest From COP Op data transfer</td></tr>
<tr><td>3’b100</td><td>5th oldest From COP Op data transfer</td></tr>
<tr><td>3’b101</td><td>6th oldest From COP Op data transfer</td></tr>
<tr><td>3’b110</td><td>7th oldest From COP Op data transfer</td></tr>
<tr><td>3’b111</td><td>8th oldest From COP Op data transfer</td></tr>
</table></td></tr>
<tr><td>CP_fordlim_m[2:0]</td><td>SOut</td><td>Comb, TF</td><td>From Coprocessor Data Out-of-Order Limit. This signal forces the coprocessor to limit how much it can reorder From COP Data. The value on this signal corresponds to the maximum allowed value to be used on CP_forder_m[2:0].</td></tr>
<tr><td>CP_fdata_m[63:0]</td><td>In</td><td>Comb, TF</td><td>From Coprocessor Data. This 64-bit bus contains data to be transferred from coprocessor. For single-word transfers, data must be duplicated on both CP_fdata_m[63:32] and CP_fdata_m[31:0]. This bus is valid when CP_fds_m is asserted.

Note: In 32-bit data transfer size configurations, this bus is reduced to CP_fdata_m[31:0].</td></tr>
<tr><td colspan="4">Coprocessor Condition Code Check (Only for BC1, MOVF, MOVT, BC2 Ops)</td></tr>
<tr><td>CP_cccs_m</td><td>In</td><td>Comb, Arith</td><td>Coprocessor Condition Code Check Strobe. This signal is asserted when condition code check results are available on CP_ccc_m.</td></tr>
<tr><td>CP_ccc_m</td><td>In</td><td>Comb, Arith</td><td>Coprocessor Condition Code Check. This signal is valid when CP_cccs_m is asserted. When this signal is asserted, the instruction checking the condition code should proceed with its execution (branch or move data). When this signal is deasserted, the instruction should not execute its conditional operation (do not branch and do not move data).</td></tr>
<tr><td colspan="4">Coprocessor Exceptions</td></tr>
<tr><td>CP_excs_m</td><td>In</td><td>Comb, Arith, TF</td><td>Coprocessor Exception Strobe. This signal is asserted when coprocessor exception signalling is available on CP_exc_m.</td></tr>
</table>

Table 3.5 Interface Signal Descriptions (Required for both COP1 and COP2) (Continued)

<table>
<tr><th>Signal Name</th><th>Dir</th><th>Issue Group</th><th>Description</th></tr>
<tr><td>CP_exc_m</td><td>In</td><td>Comb, Arith, TF</td><td>Coprocessor Exception. When this signal is deasserted, the coprocessor is not causing an exception. When this signal is asserted, the coprocessor is causing an exception. The type of exception is encoded on the signal CP_exccode_m[4:0]. This signal is valid when CP_excs_m is asserted.</td></tr>
<tr><td>CP_exccode_m[4:0]</td><td>In</td><td>Comb, Arith, TF</td><td>Coprocessor Exception Code. This signal is valid when CP_excs_m is asserted and CP_exc_m is asserted.
<table>
<tr><th>CP_exccode_m[4:0]</th><th>Exception</th></tr>
<tr><td>5’b01010</td><td>Reserved Instruction Exception</td></tr>
<tr><td>5’b01111</td><td>Floating-Point Exception</td></tr>
<tr><td>5’b10000</td><td>Available for implementation-specific use</td></tr>
<tr><td>5’b10010</td><td>COP2 Exception</td></tr>
<tr><td>other values</td><td>Reserved.

If other values are signalled, the operation of the integer processor core is UNDEFINED.</td></tr>
</table></td></tr>
<tr><td colspan="4">Instruction Nullification</td></tr>
<tr><td>CP_nulls_m</td><td>Out</td><td>Comb, Arith, TF</td><td>Coprocessor Null Strobe. This signal is asserted when a nullification signal is available on CP_null_m.</td></tr>
<tr><td>CP_null_m</td><td>Out</td><td>Comb, Arith, TF</td><td>Nullify Coprocessor Instruction. When this signal is deasserted, the integer processor core is signalling that the instruction is not nullified. When this signal is asserted, the integer processor core is signalling that the instruction is nullified. This signal is valid when CP_nulls_m is asserted.</td></tr>
<tr><td colspan="4">Instruction Killing</td></tr>
<tr><td>CP_kills_m</td><td>Out</td><td>Comb, Arith, TF</td><td>Coprocessor Kill Strobe. This signal is asserted when kill signalling is available on CP_kill_m.</td></tr>
<tr><td>CP_kill_m[1:0]</td><td>Out</td><td>Comb, Arith, TF</td><td>Kill Coprocessor Instruction. This signal indicates whether or not a coprocessor instruction is killed. It is valid when CP_kills_m is asserted.
<table>
<tr><th>CP_kill_m[1:0]</th><th>Type of Kill</th></tr>
<tr><td>2’b00</td><td rowspan="2">Instruction is not killed and can commit its results</td></tr>
<tr><td>2’b01</td></tr>
<tr><td>2’b10</td><td>Instruction is killed (not due to CP_exc_m)</td></tr>
<tr><td>2’b11</td><td>Instruction is killed (due to CP_exc_m)</td></tr>
</table></td></tr>
<tr><td colspan="4">Miscellaneous</td></tr>
<tr><td>CP_reset</td><td>Out</td><td>NONE</td><td>Coprocessor Reset. This signal is asserted when the integer processor core performs a hard or soft reset. At a minimum, this signal is asserted for two cycles.</td></tr>
</table>

Table 3.5 Interface Signal Descriptions (Required for both COP1 and COP2) (Continued)

Signal Name	Dir	Issue Group	Description
CP_idle	In	NONE	**Coprocessor Idle.** This signal is asserted when the coprocessor logic is idle. It enables the integer processor core to go into sleep mode and shut down the internal integer processor core clock. This signal is valid only if *CP1_fppresent*, *CP1_mdmxpresent*, or *CP2_present* is asserted.

Table 3.6 Coprocessor Interface Signal Descriptions (Required only for COP1)

Signal Name	Dir	Issue Group	Description
Instruction Dispatch			
CP1_as_m	Out	Comb, Arith	**Coprocessor 1 Arithmetic Instruction Strobe.** This signal is asserted in the cycle after an Arithmetic COP1 Op instruction is available on *CP_ir_m*. If *CP1_abusy_m* was asserted in the previous cycle, this signal is not asserted. In any cycle, at most one of the following signals can be asserted at a time in a particular issue group: *CP1_as_m*, *CP2_as_m*, *CP1_ts_m*, *CP2_ts_m*, *CP1_fs_m*, *CP2_fs_m*.
CP1_abusy_m	In	Comb, Arith	**Coprocessor 1 Arithmetic Busy.** When this signal is asserted, a Coprocessor 1 arithmetic instruction is not dispatched. *CP1_as_m* is not asserted in the cycle after this signal is asserted.
CP1_ts_m	Out	Comb, TF	**Coprocessor 1 To Strobe.** This signal is asserted in the cycle after a To COP1 Op instruction is available on *CP_ir_m*. If *CP1_tbusy_m* was asserted in the previous cycle, this signal is not asserted. In any cycle, at most 1 of the following signals can be asserted at a time in a particular issue group: *CP1_as_m*, *CP2_as_m*, *CP1_ts_m*, *CP2_ts_m*, *CP1_fs_m*, *CP2_fs_m*.
CP1_tbusy_m	In	Comb, TF	**To Coprocessor 1 Busy.** When this signal is asserted, a To COP1 Op is not dispatched. *CP1_ts_m* is not asserted in the cycle after this signal is asserted.
CP1_fs_m	Out	Comb, TF	**Coprocessor 1 From Strobe.** This signal is asserted in the cycle after a From COP1 Op instruction is available on *CP_ir_m*. If *CP1_fbusy_m* was asserted in the previous cycle, this signal is not asserted. In any cycle, at most one of the following signals can be asserted at a time in a particular issue group: *CP1_as_m*, *CP2_as_m*, *CP1_ts_m*, *CP2_ts_m*, *CP1_fs_m*, *CP2_fs_m*.
CP1_fbusy_m	In	Comb, TF	**From Coprocessor 1 Busy.** When this signal is asserted, a From COP1 Op is not dispatched. *CP1_fs_m* is not asserted in the cycle after this signal is asserted.
CP1_fr32_m	Out	Comb, Arith, TF	**MIPS32-Compatibility Mode – Registers.** When this signal is asserted, the dispatched instruction uses the MIPS32-compatible register file. This signal is valid the cycle before *CP1_as_m*, *CP1_fs_m*, or *CP1_ts_m* is asserted.
GPR Data (Only for ALNV.PS, ALNV.fmt, MOVN.fmt, MOVZ.fmt Arithmetic COP1 Ops)			
CP1_gprs_m	Out	Comb, Arith	**GPR Strobe.** This signal is asserted when additional general-purpose register information is available on *CP1_gpr_m*.

Table 3.6 Coprocessor Interface Signal Descriptions (Required only for COP1) (Continued)

<table>
<tr><th>Signal Name</th><th>Dir</th><th>Issue Group</th><th>Description</th></tr>
<tr><td>CP1_gpr_m[3:0]</td><td>Out</td><td>Comb, Arith</td><td>GPR Data. This bus supplies additional data from the integer general-purpose register file. CP1_gpr_m[2:0] is valid when CP1_gprs_m is asserted and only for ALNV.PS and ALNV.fmt instructions. CP1_gpr_m[3] is valid when CP1_gprs_m is asserted and only for MOVN.fmt and MOVZ.fmt instructions.
<table>
<tr><th>CP1_gpr_m[2:0]</th><th>RS
(Valid only for ALNV.PS, ALNV.fmt)</th></tr>
<tr><td>Binary encoded</td><td>Lower 3 bits of RS register contents</td></tr>
</table>
<table>
<tr><th>CP1_gpr_m[3]</th><th>RT Zero Check
(Valid only for MOVN.fmt, MOVZ.fmt)</th></tr>
<tr><td>0</td><td>RT!= 0</td></tr>
<tr><td>1</td><td>RT==0</td></tr>
</table></td></tr>
<tr><td colspan="4">Miscellaneous</td></tr>
<tr><td>CP1_fppresent</td><td>SIn</td><td>NONE</td><td>COP1 FPU Present. This signal must be asserted when COP1 FPU hardware is connected to the Coprocessor Interface.</td></tr>
<tr><td>CP1_mdmxpresent</td><td>SIn</td><td>NONE</td><td>COP1 MDMX Present. This signal must be asserted when COP1 MDMX hardware is connected to the Coprocessor Interface.</td></tr>
</table>

Table 3.7 Coprocessor Interface Signal Descriptions (Required only for COP2)

Signal Name	Dir	Issue Group	Description
Arithmetic Dispatch			
CP2_as_m	Out	Comb, Arith	**Coprocessor 2 Arithmetic Instruction Strobe.** This signal is asserted in the cycle after an Arithmetic COP2 Op instruction is available on *CP_ir_m*. If *CP2_abusy_m* was asserted in the previous cycle, this signal is not asserted. In any cycle, at most one of the following signals can be asserted at a time in a particular issue group: *CP1_as_m*, *CP2_as_m*, *CP1_ts_m*, *CP2_ts_m*, *CP1_fs_m*, *CP2_fs_m*.
CP2_abusy_m	In	Comb, Arith	**Coprocessor 2 Arithmetic Busy.** When this signal is asserted, a Coprocessor 2 arithmetic instruction is not dispatched. *CP2_as_m* is not asserted in the cycle after this signal is asserted.
CP2_ts_m	Out	Comb, TF	**Coprocessor 2 To Strobe.** This signal is asserted in the cycle after a To COP2 Op instruction is available on *CP_ir_m*. If *CP2_tbusy_m* was asserted in the previous cycle, this signal is not asserted. In any cycle, at most one of the following signals can be asserted at a time in a particular issue group: *CP1_as_m*, *CP2_as_m*, *CP1_ts_m*, *CP2_ts_m*, *CP1_fs_m*, *CP2_fs_m*.
CP2_tbusy_m	In	Comb, TF	**To Coprocessor 2 Busy.** When this signal is asserted, a To COP2 Op is not dispatched. *CP2_ts_m* is not asserted in the cycle after this signal is asserted.
CP2_fs_m	Out	Comb, TF	**Coprocessor 2 From Strobe.** This signal is asserted in the cycle after a From COP2 Op instruction is available on *CP_ir_m*. If *CP2_fbusy_m* was asserted in the previous cycle, this signal is not asserted. In any cycle, at most 1 of the following signals can be asserted at a time in a particular issue group: *CP1_as_m*, *CP2_as_m*, *CP1_ts_m*, *CP2_ts_m*, *CP1_fs_m*, *CP2_fs_m*.
CP2_fbusy_m	In	Comb, TF	**From Coprocessor 2 Busy.** When this signal is asserted, a From COP2 Op is not dispatched. *CP2_fs_m* is not asserted in the cycle after this signal is asserted.

Table 3.7 Coprocessor Interface Signal Descriptions (Required only for COP2) (Continued)

Signal Name	Dir	Issue Group	Description
CP2_kd_mode_m	Out	Comb, Arith, TF	**Kernel/Debug Mode Indication.** When this signal is asserted the dispatched instruction is executed in either Kernel or Debug mode. This signal is valid the cycle before *CP2_as_m*, *CP2_fs_m*, or *CP2_ts_m* is asserted.
Miscellaneous			
CP2_present	SIn	NONE	**COP2 Present.** This signal must be asserted when COP2 hardware is connected to the Coprocessor Interface.
CP2_tx32	SIn	NONE	**COP2 32-bit Transfers.** When this signal is asserted, the integer unit must cause an RI exception for 64-bit COP2 TF instructions. This static input must always be valid.

Chapter 4

Configurations

The Coprocessor Interface allows a coprocessor to be connected to a MIPS integer processor core. An integer processor core can implement various options of the Coprocessor Interface as described in this section. These configuration options impact the Coprocessor Interface in two ways: the signals required to be implemented and the width of the bus signals.

Figure 4.1 shows a simple block diagram of how the Coprocessor Interface connects a single coprocessor to an integer processor core.

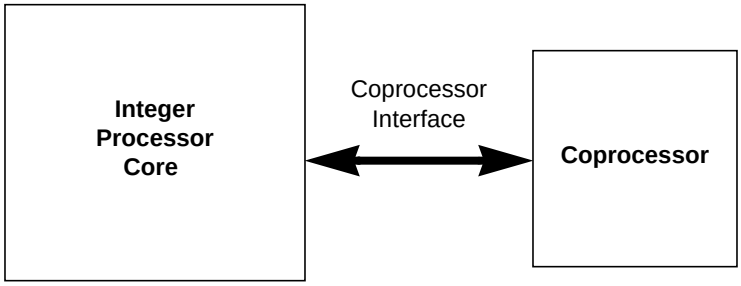

Figure 4.1 Coprocessor Interface Block Diagram

4.1 Types of Coprocessors

This section lists the different kinds of coprocessors that can be supported by an integer processor core. The integer processor core supports one or more of these options.

Each configuration option described below includes a description of which of the three signal categories is required. The signals of the Coprocessor Interface are divided into three categories:

- Signals that are required for both COP1 and COP2 implementations are named “CP_*”.
- Signals that are required only for COP1 implementations are named “CP1_*”.
- Signals that are required only for COP2 implementations are named “CP2_*”.

Note: Depending on the implementation of this interface on the integer processor core and the coprocessor, some signals while present are unused. Unused input signals on a particular implementation must be connected to their inactive states.

4.1.1 Single Coprocessor 1

COP1 is reserved for a floating-point coprocessor in the MIPS architecture. The Coprocessor Interface supports all COP1, COP1X, MDMX, and MIPS-3D instructions as defined by the MIPS ISA.

- Signals named “CP_*” are required to be implemented and must be renamed “CP1_*”.
- Signals named “CP1_*” are required to be implemented.

- 64-bit data transfers are required to be implemented.

Signals not included in the implemented interface can be ignored in the signal descriptions. For instance, *CP_idle* is renamed to *CP1_idle*. *CP1_idle* is valid only when *CP1_fppresent* or *CP1_mdmxpresent* is asserted.

4.1.2 Single Coprocessor 2

The function of Coprocessor 2 is user-definable and is intended to allow special-purpose engines, such as graphics accelerators, to be integrated into the architecture.

- Signals named “CP_*” are required to be implemented and must be renamed “CP2_*”.
- Signals named “CP2_*” are required to be implemented.

Signals not included in the implemented interface can be ignored in the signal descriptions. For instance, *CP_idle* is renamed to *CP2_idle*. *CP2_idle* is valid only when *CP2_present* is asserted.

4.1.3 Single Coprocessor 1 and 2

A user-defined coprocessor can be designed that implements functionality from both COP1 and COP2.

- All signals are required to be implemented. No renaming will take place as “CP_*” signals are shared for COP1 and COP2 functionality.
- 64-bit data transfers are required to be implemented.

4.1.4 Dual Coprocessors using Separate Interfaces

An integer processor core can feature two independent Coprocessor Interfaces: one for COP1 and one for COP2. In this case, each interface is functionally independent of the other. Each requires a full set of I/O signals as described in Section 4.1.1, "Single Coprocessor 1" and Section 4.1.2, "Single Coprocessor 2".

4.1.5 No Coprocessors

If a Coprocessor Interface is unused then all inputs must be tied to their inactive state, which is logic zero.

4.2 Data Transfer Widths

An integer processor core can support 64-bit or 32-bit data transfer sizes.

4.2.1 64-bit Transfer Width

An integer processor core that implements COP1 must support 64-bit data transfers. A processor that supports COP2 can optionally support 64-bit data transfers. For the remainder of this document, this configuration option is assumed.

An integer processor core that supports 64-bit data transfers can be connected to COP2 coprocessors designed for 32-bit transfers. The coprocessor must assert *CP2_tx32*. Furthermore the *CP_fdata_m[31:0]* output from the coprocessor must be connected to *CP_fdata_m[31:0]* and *CP_fdata_m[63:32]* of the integer processor core.

Note: When *CP2_tx32* is asserted, instructions that transfer 64 bits of data cause the integer processor core to signal a reserved instruction exception. These instructions include DMFC2, DMTC2, LDC2, and SDC2.

4.2.2 32-bit Transfer Width (Cop2 only)

An integer processor core that supports only COP2 can optionally support only 32-bit transfers. In this configuration, the use of instructions that transfer 64 bits of data causes a reserved instruction exception from the integer processor core.

With this configuration, the following restrictions apply:

- The integer processor core must signal Reserved Instruction exception for DMFC2 (MIPS64), DMTC2 (MIPS64), LDC2 (MIPS32), and SDC2 (MIPS32) instructions.
- *CP2_tx32* cannot be implemented. A 32-bit integer processor core always works as if *CP2_tx32* is asserted, thus the signal is not needed.
- 32-bit buses are required to be implemented:
  - *CP_tdata_m[63:0]* is reduced to *CP_tdata_m[31:0]*.
  - *CP_fdata_m[63:0]* is reduced to *CP_fdata_m[31:0]*.

4.3 Out-of-Order Data Transfers

An integer processor core can support a configurable degree of out-of-order data transfers on both the To COP Data and From COP Data transfer interfaces. The Coprocessor Interface includes handshake signals that allow any integer processor core to work with any coprocessor.

For To COP Data, an integer processor core can reorder data for up to eight instructions. However, it must limit this out of order data transfer according to *CP_tordlim_m[2:0]*. This signal allows the coprocessor to limit reordering to only as much as it can handle.

Similarly for From COP Data, a coprocessor can return data for up to eight instructions out of order. The integer processor core can limit this reordering using the *CP_fordlim_m[2:0]* static output. This signal works the same way as *CP_tordlim_m*.

4.4 Multi-Issue Support

The Coprocessor Interface is extensible to support single-issue to multi-issue integer processor cores and coprocessors. Furthermore, it enables compatibility between any integer processor core and any coprocessor without glue logic.

Multi-issue support is easily achieved by duplicating certain signals of the Coprocessor Interface. This section specifies in detail exactly what needs to be duplicated for the different configuration options. In general, the following rules apply:

- Signals are grouped together to form an “issue group”.
- There are three types of issue groups: Combined, Arithmetic, and To/From.
  - The Combined issue group includes all signals used for both arithmetic and To/From instructions.
  - The To/From issue group includes all signals used for To/From instructions.
  - The Arithmetic issue group includes all signals used for arithmetic instructions.

- A particular issue group is delineated by a unique suffix of the form "_*m*" where *m* is an integer that signifies the "issue group" for those signals. The value of *m* must be between 0 and 7, inclusive. Because *CP_order_m* has only three bits, there cannot be more than eight issue groups.
- Signals that are not associated with an issue group do not have the "_*m*" suffix.
- An integer processor core must have at least one Combined issue group. This group must be assigned as issue group 0 (*m* = 0). The integer processor core can have up to seven additional issue groups of any type.

Note: Depending on the implementation of this interface on the integer processor core and the coprocessor, some signals while present are unused. Unused input signals on a particular implementation must be connected to their inactive states.

4.4.1 Single-Issue Support

An integer processor core that supports only single issues will implement a single Combined issue group as follows:

- This group is Issue Group 0 (*m* = 0).
- *CP_adisable_0* and *CP_tfdisable_0* cannot be implemented. Because this is the only issue group, these instructions can never be disabled.
- *CP_order_0[2:0]* cannot be implemented. Because there is only one issue group, dispatch order is not needed.

An integer processor core with this configuration can be used with a coprocessor with more issue groups. In this case, the Combined issue group of the coprocessor is connected to the integer processor core and the other issue groups of the coprocessor are tied inactive.

4.4.2 Limited Dual-Issue Support

An integer processor core that supports limited dual issues supports dual issuing of instructions only, where one is an arithmetic coprocessor instruction and the other is a To/From coprocessor instruction. With this option, two issue groups are implemented - one combined (Issue Group 0) and one arithmetic (Issue Group 1).

- If *CP_adisable_1* is asserted, the integer processor core must dispatch arithmetic instructions using Issue Group 0. If *CP_adisable_1* is deasserted, the integer processor core must dispatch arithmetic instructions using Issue Group 1.
- *CP_adisable_0* and *CP_tfdisable_0* cannot be implemented. *CP_tfdisable_0* is not needed because this is the only issue group for To/From instructions; these instructions cannot be disabled. *CP_adisable_0* is not needed because the integer processor core only uses the combined interface for arithmetic instructions if the coprocessor is already asserting *CP_adisable_1* for the Arithmetic Issue group.

The above rules allow a single-issue coprocessor to be used with a limited dual-issue integer processor core by simply connecting the combined issue groups together and asserting *CP_adisable_1*.

Coprocessors with more multi-issue support can be connected to a limited dual-issue integer processor core by tying off unused issue groups on the coprocessor.

4.4.3 Dual Arithmetic Issues

An integer processor core that supports full dual issues supports all the cases of limited dual issues, plus it can issue two arithmetic instructions or two To/From instructions. With this option, two combined issue groups are implemented.

A single-issue coprocessor can be used with a dual-issue integer processor core by simply connecting the combined issue groups together and asserting *CP_adisable_m* and *CP_tfdisable_m* for the second combined issue group of the integer processor core.

A limited dual-issue coprocessor can be used with a dual-issue integer processor core by connecting the coprocessor combined issue group to one of the integer processor core's combined issue groups and asserting *CP_adisable_m* for that issue group. Then connect the coprocessor's arithmetic issue group to the remaining combined issue group of the integer processor core and assert *CP_tfdisable_m* for that issue group.

4.4.4 Additional Multi-Issue Support

The rules explained in the previous section can be easily extrapolated for up to eight simultaneously dispatched instructions from the integer processor core.

4.5 Multithreading Support

Support for multithreading is supplied by three instruction dispatch signals and the To COP Data (Delayed) transfer. At instruction dispatch, *CP_tcid_m*, *CP_vpeid_m* and *CP_targtcid_m* indicate the additional TC and VPE information needed by a coprocessor to execute instructions.

The To COP Data (Delayed) transfer is required to enable blocking loads to remain outstanding while instructions from other TCs continue to execute. When a To COP Data (Delayed) transfer is pending, no other coprocessor instructions for that TC will be completed.

In a non-multithreaded coprocessor, a Kill transfer kills the instruction and all instructions behind it that have been dispatched. In a multithreaded coprocessor, a Kill transfer still kills the instruction, but it also only kills subsequent instructions from the same TC. Instructions from other TCs are not killed. This rule enables a pipelined coprocessor to work efficiently because data dependent instructions will be killed if the source instruction of the dependency is killed.

Note: Because instructions from different TCs are not always killed together, it is recommended that coprocessor instructions that access other TCs registers not be pipelined together. Data bypassed from an instruction that was subsequently killed is not valid and the recipient instruction must be restarted or otherwise get the original data.

Chapter 5

Interface Protocols

This section describes the different types of transfers that occur over the Coprocessor Interface. It also describes the function of specific signals including hardware and idle indicators and reset.

5.1 Overview of Transfers

The Coprocessor Interface is composed of several simple transfers:

- **Instruction Dispatch** - Starts coprocessor instructions.
- **To COP Data** - Transfers data to the coprocessor.
- **To COP Data (Delayed)** - Transfers data to the coprocessor for blocking loads.
- **From COP Data** - Transfers data from the coprocessor.
- **Coprocessor Condition Code Checking** - Transfers the coprocessor condition check result to the integer processor core.
- **GPR Data** - Transfers additional data from the integer processor core's general-purpose register file to the coprocessor.
- **Coprocessor Exceptions** - Notifies the integer processor core if any coprocessor exceptions happened for an instruction.
- **Instruction Nullification** - Notifies the coprocessor whether instructions are nullified or not.
- **Instruction Killing** - Notifies the coprocessor when instructions can commit state or not.

All transfers use the following protocol:

1. 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. Do not use it to clock registers.
2. No handshake confirmation of transfer.
3. No flow control except for instruction dispatches.
4. Out-of-order transfers are not allowed except for To/From COP data transfers. All transfers of a given type, except To/From COP data transfers, in the same issue group must be in dispatch order.
5. 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 integer processor core. The additional information and the transfers required are summarized in Table 5.1.

Note: For each dispatch type given in the table, all listed transfers are *required* to be done. No transfers are optional. However, after an instruction is killed or nullified, any transfers that have not already happened will not happen. In other words, once an instruction is killed or nullified, no further transfers for that instruction can happen.

Table 5.1 Transfers Required for Each Dispatch

Dispatch Type	Required Transfers	Direction Core <—> COP
To COP Op	• Instruction nullification	—>
	• To Coprocessor data transfer	—>
	• Coprocessor exceptions	<—
	• Instruction killing	—>
From COP Op	• Instruction nullification	—>
	• From Coprocessor data transfer	<—
	• Coprocessor exceptions	<—
	• Instruction killing	—>
Arithmetic COP Op	• Instruction nullification	—>
	• Coprocessor exceptions	<—
	• Instruction killing	—>
Additionally for BC1[1] BC2[1] MOVF[1] MOVT[1]	• Condition code check results	<—
Additionally for MOVZ.fmt[1] MOVN.fmt[1] ALNV.PS[1] ALNV.fmt[1]	• GPR Data	—>

1. For a description of this instruction, refer to the MIPS ISA definition.

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

The Coprocessor Interface operates with coprocessors of any pipeline structure and latency. If the integer processor core requires a specific transfer by a certain cycle, the integer processor core must stall until the transfer has completed. However, if an exceptional instruction (CpU, MDMX, RI) was dispatched then the integer processor core cannot expect that the coprocessor is able to return any transfers. In that case the integer processor core must release any stalls on the instruction and send an instruction kill transfer.

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 two cycles, then two transfers occur. For instruction dispatches, the strobe signal is asserted in the cycle after the instruction is dispatched in order to insulate the signals from poor timing.

Figure 5.1 shows examples of the transfer of nullification information. However, all non-dispatch transfers follow the same protocol.

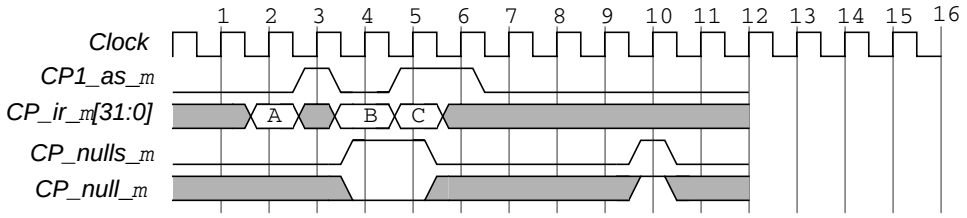

Figure 5.1 General Transfer Example

On edge 4, *CP_nulls_m* is asserted, signifying the null transfer for Instruction A. Since *CP_null_m* 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. Because it is the cycle after dispatch, this is the earliest possible time any transfer for Instruction B can happen. Instruction C is dispatched at edge 5. However, the nullification transfer is delayed for some reason until edge 10.

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. However, 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.

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 can occur, but they are ignored.

The integer processor core is typically pipelined internally. This may imply bindings on what transfers the integer processor core requires before returning other transfers. For instance, an integer processor core implementation may require that the coprocessor returns From data before the kill transfer can happen. Such requirements should be described in the documentation for the integer processor core.

5.2 Instruction Dispatch Transfer

The instruction dispatch transfer signals the coprocessor to start executing 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 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.

CP1_as_m, *CP2_as_m*, *CP1_ts_m*, *CP2_ts_m*, *CP1_fs_m*, and *CP2_fs_m* are asserted in the cycle after the instruction is driven. These signals are delayed strobe signals; 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 be timing-critical.

Because the above instruction strobes are delayed, the coprocessor is normally required to register *CP_ir_m* 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 *CP_irenable_m* signal compensates for this effect by enabling the coprocessor to avoid registering instructions that will never be dispatched to it.

Only one of the instruction strobes in an issue group can ever be asserted at the same time: *CP1_as_m*, *CP2_as_m*, *CP1_ts_m*, *CP2_ts_m*, *CP1_fs_m*, and *CP2_fs_m*. However, if multiple enabled issue groups exist, more than one

instruction can be dispatched per cycle. When two instructions are dispatched at the same time, the coprocessor must know their program order to properly calculate dependencies. This information is output on *CP_order_m[2:0]*. For the first instruction, *CP_order_m* is 0. For the next instruction *CP_order_m* is 1, and so on.

By asserting *CP_adisable_m* or *CP_tfdisable_m* appropriately, coprocessors that do not support superscalar operation can disable it.

The integer processor core is allowed to dispatch an instruction within the To/From/Arithmetic groups even though the instruction is exceptional due to RI, MDMX and CpU exceptions. If such an instruction is dispatched, the integer processor core must subsequently kill the instruction (refer to Section 5.9, "Instruction Killing Transfer") without expecting that the coprocessor returns any transactions. Note that RI on Arithmetic instructions must be signalled by the coprocessor whereas RI on To/From instructions must be signalled by the integer processor core.

However, it is not allowed to dispatch instructions to not present hardware when using the configuration described in Section 4.1.3, "Single Coprocessor 1 and 2". This restriction does not apply to the other proposed configurations.

The above two paragraphs imply that a coprocessor that does not support all instructions must be able to recognize all possible instructions for the purpose of the interface transfers. For instance, if MDMX or paired single is unimplemented in a COP1 the coprocessor should not by mistake use a GPR transfer for the ALNV.fmt/ALNV.PS instructions for the COP1 MOVN.fmt/MOVZ.fmt instructions.

When the processor is operating in MIPS32-compatibility mode according to the User/Supervisor/Kernel/Debug mode (*PX*, *SX*, and *UX* bits of the CP0 *Status* register), the *CP_inst32_m* signal is asserted during dispatch. The coprocessor must signal a Reserved Instruction exception for any arithmetic instruction that is not MIPS32 compatible. Refer to the MIPS ISA documentation for more details on MIPS32-compatibility modes in integer processor cores.

CP1_fr32_m can be asserted during dispatch to notify the coprocessor that MIPS32-compatible floating-point registers are enabled. Normally, the coprocessor would then change the behavior of some instructions to correctly operate using the MIPS32-compatible register file. *CP1_fr32_m* is asserted according to the *FR* bit in the CP0 *Status* register.

The *CP_endian_m* signal is asserted during dispatch to notify the coprocessor of the proper byte-ordering mode to use. This indication is needed for the ALNV.fmt and ALNV.PS instructions.

The *CP2_kd_mode_m* signal is asserted during dispatch to notify the coprocessor that the instruction is executed in Kernel or Debug mode. This allows for implementation of COP2 coprocessor instructions which cannot be executed outside Kernel or Debug mode.

Figure 5.2 shows waveforms for an example Coprocessor 1 dispatch. Dispatch of Coprocessor 2 instructions is the same, with different signal names.

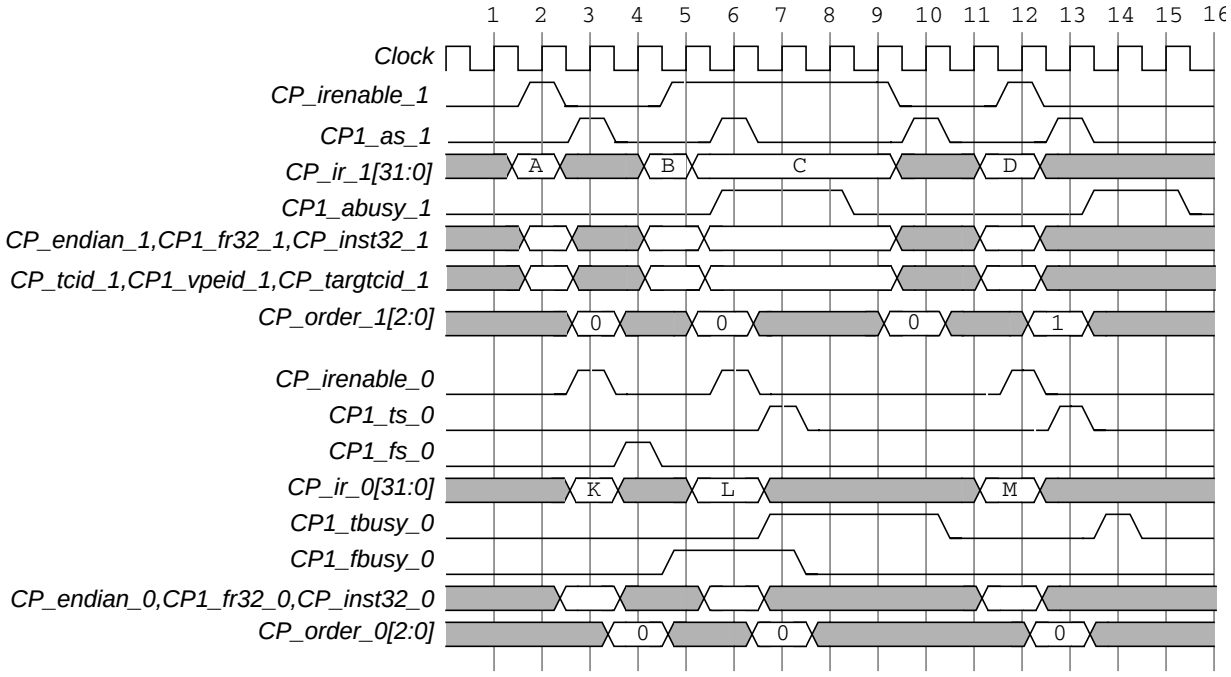

Figure 5.2 Arithmetic Coprocessor Dispatch Waveform

On edge 2, Instruction A is dispatched. On edge 3, *CP1_as_1* is asserted, validating the previous cycle's dispatch. *CP1_as_1* is always asserted in the cycle after the instruction word is driven. On edge 3, Instruction K is dispatched. *CP1_fs_0* is asserted on edge 4.

On edge 5, Instruction B is dispatched. On edge 6, Instruction C is driven onto *CP_ir_1*, and Instruction L is driven onto *CP_ir_0*. Instruction C is not dispatched because *CP1_abusy_1* was asserted. But Instruction L was dispatched. For Instruction C, the integer processor core does not assert *CP1_as_1* until the coprocessor can accept it (when *CP1_abusy_1* is deasserted). Instruction C is finally dispatched on edge 9.

On edge 12, both Instructions D and M are dispatched at the same time. *CP_order_0* and *CP_order_1* are valid on edge 13 and indicate that Instruction M was functionally before Instruction D.

5.3 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, the coprocessor must assert *CP1_tbusy_m* or *CP2_tbusy_m*, as appropriate, to prevent dispatch.

In some processors, a To COP Op can be killed after the instruction has passed the point in the integer pipeline where both the nullification and killing transfers would normally occur. To enable subsequent non-dependent instructions to continue on the COP interface, the *CP_tdk_m* signal is available to allow the processor to kill the To COP Op after the nullification and killing transfers have been completed. When the coprocessor sees *CP_tds_m* asserted, if *CP_tdk_m* is asserted, this indicates that *CP_tdata_m* should be dropped and not affect the state of the coprocessor.

In order to exploit parallelism in a multithreaded system, delaying To COP Data is sometimes needed. By delaying the data transfer, the interface enables an in-order coprocessor to have a pending transfer for one TC while continuing to

transfer To COP Data for other TCs. A given TC can have at most 1 To COP Data transfer delayed at a time. Integer cores are expected to use this transfer for blocking load data. A delayed transfer is indicated by the assertion of *CP_tdd_m* during the normal To COP Data transfer. When this signal is asserted, the coprocessor should ignore *CP_tdata_m* and instead expect a subsequent To COP Data (Delayed) transfer.

A delayed kill signal (*CP_tddk_m*) is also supplied for delayed To COP Data transfers. Integer cores are expected to use this kill when blocking loads are killed due to a TC being halted. When the coprocessor sees *CP_tddk_m* asserted, this indicates that *CP_tdata_m* should be dropped and not affect the state of the coprocessor. A late kill is possible because no subsequent dependent instructions will have been started. Thus, no state has been committed and it is still possible to precisely kill the instruction.

The Coprocessor Interface allows out-of-order data transfers; that is, 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 *CP_torder_m[2:0]* signal is also sent. This signal tells the coprocessor whether the data word is for the oldest outstanding To COP data transfer, the second oldest, or the third oldest, etc. The Coprocessor Interface allows up to eight transfers to be outstanding while returning data for the next transfer. The coprocessor can limit the extent of this reordering to match what its hardware supports using the *CP_tordlim_m[2:0]* signal.

The type of instruction dispatched determines which bits on the bus are valid:

- 32-bit transfer: The 32-bit data word is driven on *CP_tdata_m[31:0]*.
- 64-bit transfer: The 64-bit data word is driven on *CP_tdata_m[63:0]*.

Figure 5.3 shows waveforms for an example To Coprocessor data transfer. Three instructions are dispatched: A, B, C and D on edges 2, 4, 6 and 8, respectively. Data for Instruction A is sent on edge 6. At that time, it is the oldest outstanding transfer, so *CP_torder_m* is set to 0. On edge 10, data for Instruction C is sent. Because it is the second oldest outstanding transfer, *CP_torder_m* is set to 1. In the following cycle, data for Instruction B is finally transferred. That instruction is now the oldest outstanding instruction, so *CP_torder_m* is again set to 0. On edge 13, the data transfer for Instruction D is completed.

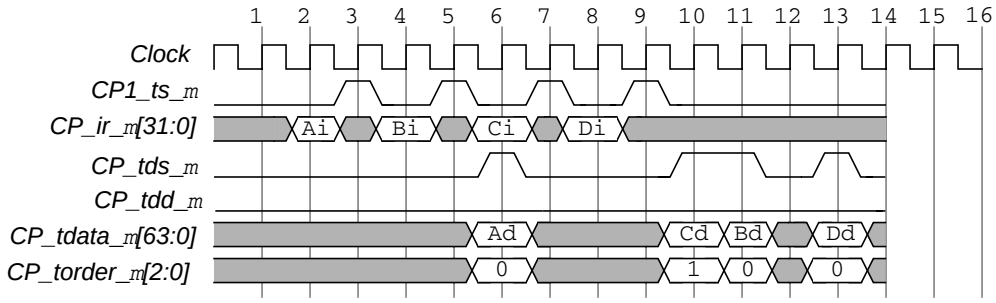

Figure 5.3 To Coprocessor Data Transfer Waveforms

Figure 5.4 shows waveforms for an example To Coprocessor data (Delayed) transfer. Three instructions are dispatched: A, B, and C on edges 2, 4, and 6, respectively. At edge 6, the To COP Data for Instruction A is delayed since *CP_tds_m* and *CP_tdd_m* are asserted. *CP_tdata_m* is therefore invalid at this edge. Note however, that *CP_torder_m* does still indicate the relative ordering for this To COP Data transfer. Data for Instruction C and D is transferred out of order on edges 10 and 11 similar to the example given above. Note here that the To COP Data transfer is completed as far as *CP_torder_m* is concerned. At edge 13, the delayed data for Instruction A is finally transferred. In this example, the data is valid because *CP_tddk_m* was deasserted. However, the transfer could have been killed if *CP_tddk_m* were asserted at edge 13.

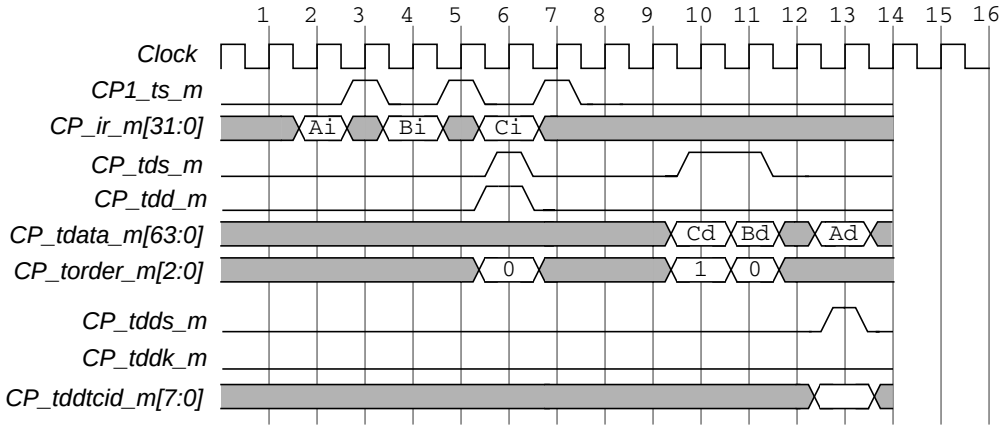

Figure 5.4 To Coprocessor Data (Delayed) Transfer Waveforms

5.4 From Coprocessor Data Transfers

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 integer processor core must have buffers for this data that enable the transfer to occur in 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 *CP_forder_m[2:0]* signal is also sent. This signal tells the integer processor core whether the data word is for the oldest outstanding From COP data transfer, the second oldest, or the third oldest, etc. The Coprocessor Interface allows up to eight transfers to be outstanding while returning the data for the next transfer. The integer processor core can limit the extent of this reordering to match what its hardware supports using the *CP_fordlim_m[2:0]* signal.

For single-word transfers, the coprocessor must drive the 32-bit value on both *CP_fdata_m[31:0]* and *CP_fdata_m[63:32]*, making the transfer independent of the byte ordering (big or little endian).

Note: For integer processor cores that only support 32-bit COP2, From COP Data is always 32 bits wide and is only driven on *CP_fdata_m[31:0]*.

Figure 5.5 shows waveforms for an example From Coprocessor data transfer. The A, B, and C instructions are dispatched on edges 2, 3, and 4, respectively. The coprocessor returns the data for Instruction A on edge 4.

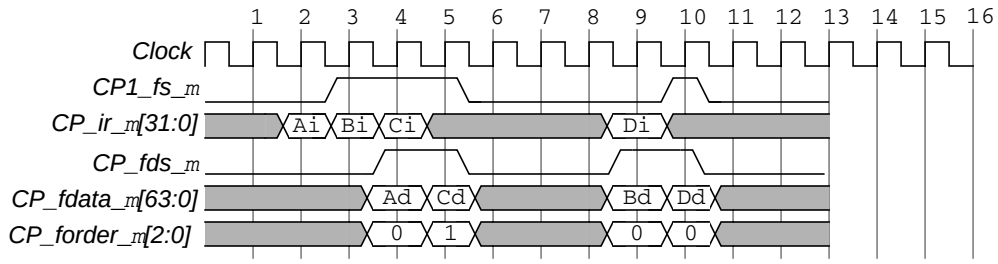

Figure 5.5 From Coprocessor Data Transfer Waveforms

On edge 5, the data for Instruction C is returned. Note that Instruction C's data is returned before the data for Instruction B and is thus out-of-order (indicated on *CP_forder_m* = 3'b1).

Instruction D is dispatched on edge 9. At the same time, the data for Instruction B is sent. On edge 10, data for Instruction D is sent one cycle after dispatch, which is the fastest data return possible.

5.5 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 BC1, BC2, MOVF and MOVT 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 execute that instruction. For branches, the coprocessor tells the integer processor core whether or not to branch. For conditional moves, the coprocessor tells the integer processor core whether or not to do the move. For this reason, the coprocessor must interpret the type of instruction to decide whether or not to execute it. Customer-defined BC1, BC2, MOVF and MOVT instructions are thus possible.

Condition code check transfers follow the generic example given in Figure 5.1 on page 29. The signals *CP_cccs_m* and *CP_ccc_m* are used instead of *CP_nulls_m* and *CP_null_m* as shown in the figure.

5.6 GPR Data Transfers

The integer processor core transfers the results of a check that RT == 64'b0 for the two special arithmetic Coprocessor 1 instructions, MOVN.fmt and MOVZ.fmt. It also transfers the lower three bits of the RS operand for the ALNV.PS and ALNV.fmt Coprocessor 1 instructions. When these instructions are dispatched to the coprocessor, they are also dispatched to the integer pipeline. In this way, the integer processor core can properly bypass RS as well as check the RT value against zero.

GPR data transfers follow the generic example given in Figure 5.1. The signals *CP1_gprs_m* and *CP1_gpr_m[3:0]* are used instead of *CP_nulls_m* and *CP_null_m* as shown in the figure.

5.7 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 for any instruction dispatched to it. However, the coprocessor should only signal FPE exceptions for COP1 and C2E exceptions for COP2. The coprocessor can also signal one of two implementation-specific exception codes. These exception codes can be used to trigger special software exception handling routines.

Note: A coprocessor can signal an exception for To/From COP Ops. Except for CTC1 and CTC2 instructions, this exception cannot depend on the associated data, implying that the integer processor core must transfer the CTCx data before it requires the exception information to prevent a deadlock condition.

Note: An integer processor core cannot expect that a coprocessor will return any additional transfers if it has signalled that an instruction is exceptional. The integer processor core must thus release stalls for that instructions and not wait for e.g. From COP Data or CCC transfers.

Signalling for Reserved Instruction exceptions are 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.
- The coprocessor hardware must signal Reserved Instruction exceptions for all arithmetic coprocessor instructions.

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

If imprecise coprocessor exceptions are allowed, the coprocessor can use the “No exception” signal immediately after dispatch to prevent stalling in the integer pipeline while waiting for precise results. If an exception does occur for that instruction, 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.

Exception transfers follow the generic example given in Figure 5.1. The signals *CP_excs_m*, *CP_exc_m*, and *CP_exccode_m[4:0]* are used instead of *CP_nulls_m* and *CP_null_m* as shown in the figure.

5.8 Instruction Nullification Transfers

All instructions dispatched utilize this transfer. It is used to signal if an instruction was nullified in the integer processor core. This transfer must happen 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 are also killed. An instruction can 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 can 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.

Nullification transfers follow the generic example given in Figure 5.1.

5.9 Instruction Killing Transfer

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

Due to various exceptional conditions, any instruction might 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 might need to perform special operations. For example, if a floating-point instruction is killed because of a Floating-point exception, the coprocessor must update exception status bits in the coprocessor’s *FCSR* register. 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.

When a coprocessor instruction is killed, all subsequent coprocessor arithmetic instructions and To/From COP Ops in the same issue group that have been dispatched from the same TC are also killed. This is necessary because the killed instruction(s) might affect the operation of subsequent instructions (for example, because of bypassing). In the cycle in which an instruction is killed, other transfers can occur, but after that cycle, no further transfers occur for any of the killed instructions. A side-effect 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 the same kill signals kills that instruction as well.

Note: Because instructions from different TCs are not always killed together, it is recommended that coprocessor instructions that access other TCs registers not be pipelined together. Data bypassed from an instruction that was subsequently killed is not valid and the recipient instruction must be restarted or otherwise get the original data.

Killing transfers follow the generic example given in Figure 5.1. The signals *CP_kills_m* and *CP_kill_m[1:0]* are used instead of *CP_nulls_m* and *CP_null_m* as shown in the figure.

5.10 Transfer Example

Figure 5.6 shows an example of a complete transfer sequence on a COP1 coprocessor interface generated by the various types of instructions listed in Table 5.2.

Note that the example does not cover all possible scenarios.

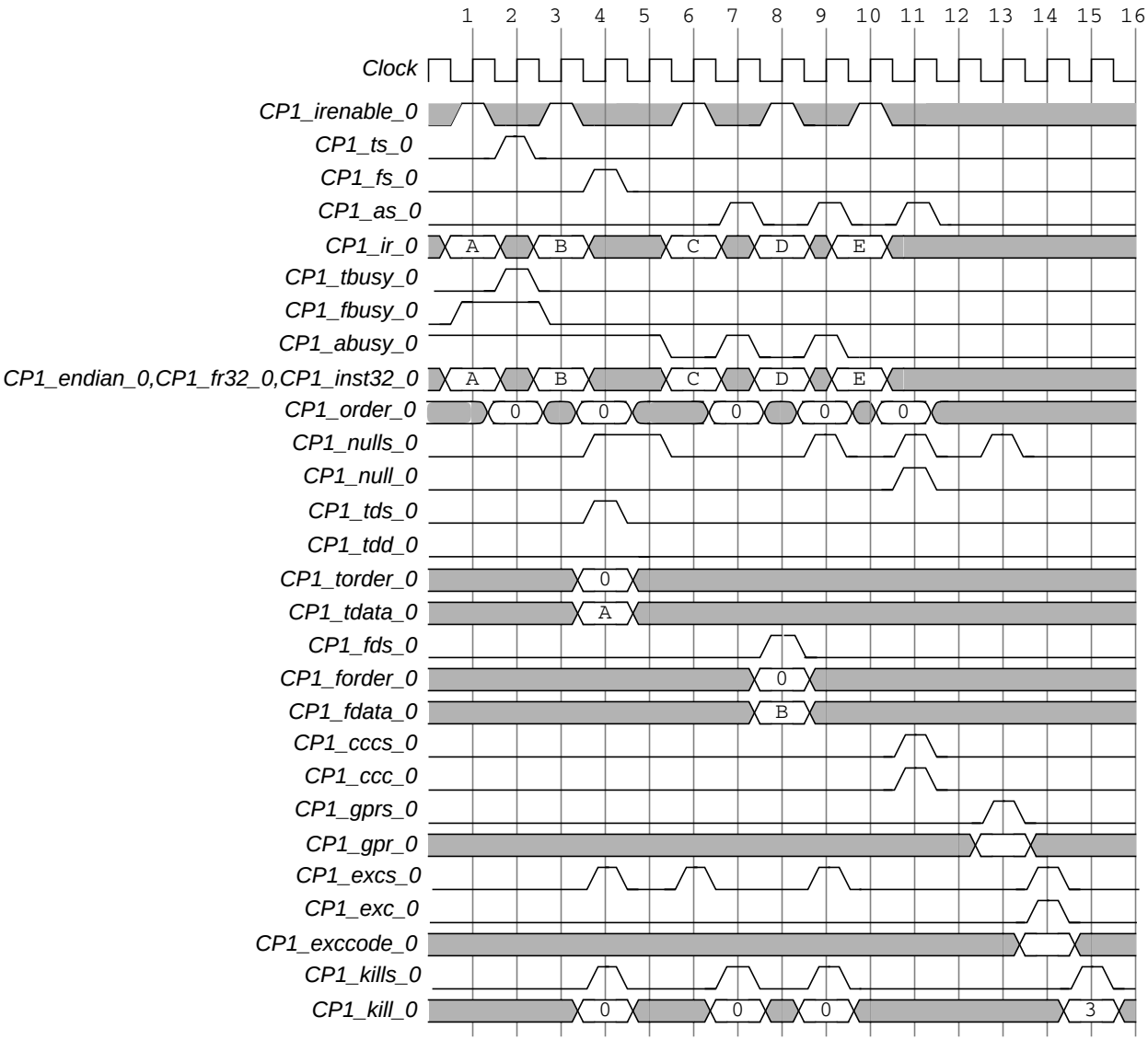

Figure 5.6 Complete COP1 Sequence

Table 5.2 Transfers in Above Waveform (numbers refer to clock cycles)

Inst	Opcode	Dispatch	Null	To Data	From Data	CCC	GPR	Exc	Kill
A	MTC1 / LWC1	1	4	4	-	-	-	4	4
B	MFC1 / SWC1	3	5	-	8	-	-	6	7
C	ADD.s	6	9	-	-	-	-	9	9
D	BC1	8	11[1]	-	-	11	-	-	-
E	MOVZ.s	10	13	-	-	-	13	14	15

1. This transfer nulls instruction D and inhibits further transfers for this instruction.

5.11 Miscellaneous Coprocessor Signals

This section describes the function of the hardware and coprocessor indicators. It also describes the operation of the coprocessor reset signal.

5.11.1 Hardware Present Signaling

Three Coprocessor Interface static inputs (*CP1_fppresent*, *CP1_mdmxpresent*, and *CP2_present*) enable the integer processor core to know what type of hardware is connected to the Coprocessor Interface. If one of these signals is asserted and the respective hardware is not available to handle the instructions, the operation is **UNDEFINED**, and the integer processor core might hang.

The three signals drive the *FP, MD* and *C2* bits of the CP0 *Config1* register, respectively. If either *FP* or *MD* is set, the *CU1* bit in the CP0 *Status* register can be set by software. If *C2* is set, the *CU2* bit in the CP0 *Status* register can be set by software.

If the *CU1* bit in the CP0 *Status* register is cleared the execution of a COP1 instruction will cause the integer processor core to signal a Coprocessor Unusable exception. Likewise, a cleared *CU2* bit in the *Status* register will cause a Coprocessor Unusable exception when executing a COP2 instruction.

If *CP1_mdmxpresent* is deasserted, the execution of an MDMX instruction will cause the integer processor core to signal a Reserved Instruction exception. If *CU1* is deasserted (but the MDMX hardware is present) an MDMX instruction will cause a Coprocessor Unusable exception. Likewise, if the MDMX hardware is present, but the MX bit in CP0 *Status* register is deasserted, then an MDMX Unusable exception will be signalled.

5.11.2 Coprocessor Idle

The Coprocessor Interface includes an idle indication from the coprocessor, *CP_idle*. The coprocessor deasserts this signal whenever it is performing a calculation, and asserts this signal when it has no instructions in progress. When asserted, *CP_idle* allows the integer processor core to enter a low-power mode, potentially shutting down the internal integer processor core clock. *CP_idle* is ignored if no coprocessor is using the Coprocessor Interface (when *CP1_fppresent, CP1_mdmxpresent,* and *CP2_present* are all deasserted).

Since the coprocessor will deassert *CP_idle* when any instruction is in-progress, the integer processor core design must take into account instructions that will not complete before entering power-down mode. If an instruction is dispatched

to the coprocessor, the coprocessor will not assert *CP_idle* until that instruction is completed. In the MIPS architecture, the WAIT instruction enables low-power mode and normally stalls the integer processor pipeline. The integer processor core can solve this problem in several ways:

- Do not dispatch instructions after a WAIT instruction.
- Nullify all instructions that are dispatched after a WAIT instruction.
- Kill all instructions that are dispatched after a WAIT instruction.
- Ignore *CP_idle* after a certain number of cycles.

Unless one of the above solutions or something similar is used, the coprocessor holds *CP_idle* deasserted because dispatched instructions cannot complete due to the WAIT instruction being stalled in the pipeline. The integer processor core will never enter low-power mode due to the fact that *CP_idle* is deasserted.

5.11.3 Reset

When the integer processor core is reset, it asserts *CP_reset*. On reset, the coprocessor must stop all in-progress operations and reset all control state machines to their idle states. When *CP_reset* is asserted, any in-progress protocols are broken, and all transfers immediately stop. All signals must reset to their inactive states by the cycle *CP_reset* is deasserted.

Note: CP_reset can be asserted for as few as two cycles, although longer assertions are legal. Thus the coprocessor must properly reset even when *CP_reset* is asserted for only two cycles.

After *CP_reset* is deasserted, transactions are not started on the Coprocessor Interface for at least four cycles, giving the coprocessor extra time to reset its state machines before a new instruction is dispatched. However, all Coprocessor Interface signals must still be deasserted by the cycle *CP_reset* is deasserted so that both the integer processor core and the coprocessor start transfers cleanly after reset.

Appendix A

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
0.1	May 31, 2000	Initial version.
0.2	June 1, 2000	Added Open issues.
0.3	June 5, 2000	Updated after review.
0.4	June 15, 2000	32-bit dynamic mode - removed *CP_tduw*, changed definitions for single-word transfers.
0.5	June 20, 2000	Updated post-review.
1.0	July 10, 2000	Final post-review edits.
1.1	July 27, 2000	Results from Vidya review.
1.2	October 23, 2000	Added notes of clarification that unused inputs must be connected inactive.
1.3	Nov 17, 2000	• Clarified description of which instructions are killed by a kill signal. • Clarified the fact that coprocessor conditional instructions and instructions that test integer processor core registers are dispatched as arithmetic instructions.
1.4	Nov 29, 2000	Added a note about the term “integer processor core” to Section 1.
1.5	Dec 4, 2000	• Split section 5.9 into three subsections. • Added new section 5.9.3 describing reset behavior.
1.6	Dec 5, 2000	Changed minimum reset length from 1 cycle to 2 cycles.
1.7	Jan 8, 2001	Added a note of clarification about instruction strobes—they can be asserted for additional instructions as long as those instructions are killed.
1.8	Feb 8, 2001	Added section 4.1.5 describing a processor with two Coprocessor Interfaces.
1.9	Feb 13, 2001	Added note to section 5.6 clarifying stalls for exceptional instructions.
1.10	Feb 16, 2001	• Changed CP_tx32 -> CP2_tx32. • Changed CP_fr32_m -> CP1_fr32_m. • Added description for CP_idle relating to integer processor core design and a potential lock-out condition where low-power mode would never be entered.
1.11	March 30, 2001	Converted to new template.

Revision	Date	Description
1.12	June 12, 2001	Explicitly listed all To/From COP Ops in Section 2. Changed CP_* signal names for all configurations except the shared COP1/COP2 option. Clarified how dispatch works around CpU/RI/MDMX exceptions. Corrected section 5.10.1.
1.13	August 31, 2001	Document template updated.
1.14	March 22, 2002	Added MIPS32 Release 2 M{F|T}HC{1|2} instructions (section 2) Added CP2_kd_mode_m signal. Minor clarifications (sections 4.1.4, 5.1, 5.2) Complete transfer example (section 5.10)
1.15	September 25, 2002	Added opcodes for all listed instructions (section 2) Minor clarifications and typos (sections 2, 5.3, Revision History)
1.16	August 20, 2004	Removed Implementation Specific 2 exception code Updated templates
2.00	November 1, 2007	Added *CP_tdk_m* signal Updated templates
2.10	December 19, 2008	Enhanced for Multithreading, This includes new dispatch signals *CP_tcid_m[7:0]*, *CP_vpeid_m[3:0]*, and *CP_targtcid_m[7:0]*. Multithreading also required the addition of To COP Data (Delayed) transfers. This includes new signals *CP_tdds_m*, *CP_tddtcid_m[7:0]*, and *CP_tddata_m[63:0]*. Furthermore, *CP_tdk_m* was renamed to *CP_tddk_m*.
2.11	July 8, 2009	Clarified that a Kill transfer on a multithreaded coprocessor only affects instructions from that TC. Removed *CP_tddata_m*. Delayed To COP Data transfers now utilize *CP_tdata_m* instead. Reinstated *CP_tdk_m*.