To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: “Standard”, “High Quality”, and “Specific”. The recommended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as “Specific” without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as “Specific” or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is “Standard” unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

“Standard”:	Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.
“High Quality”:	Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.
“Specific”:	Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.

8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

User's Manual

Phase-out/Discontinued

VR10000 Series™

64-/32-bit Microprocessor

μPD30700 (VR10000™)
μPD30700L (VR10000L™)
μPD30710 (VR12000™)
μPD30710A (VR12000A™)
μPD30710L (VR12000L™)

Document No. U10278EJ4V0UM00 (4th edition)
Date Published March 2001 N CP(K)

© NEC Corporation 1995, 2001
© MIPS Technologies, Inc. 1994
Printed in Japan

[MEMO]

NOTES FOR CMOS DEVICES

① PRECAUTION AGAINST ESD FOR SEMICONDUCTORS

Note:

Strong electric field, when exposed to a MOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop generation of static electricity as much as possible, and quickly dissipate it once, when it has occurred. Environmental control must be adequate. When it is dry, humidifier should be used. It is recommended to avoid using insulators that easily build static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work bench and floor should be grounded. The operator should be grounded using wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions need to be taken for PW boards with semiconductor devices on it.

② HANDLING OF UNUSED INPUT PINS FOR CMOS

Note:

No connection for CMOS device inputs can be cause of malfunction. If no connection is provided to the input pins, it is possible that an internal input level may be generated due to noise, etc., hence causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using a pull-up or pull-down circuitry. Each unused pin should be connected to V_{DD} or GND with a resistor, if it is considered to have a possibility of being an output pin. All handling related to the unused pins must be judged device by device and related specifications governing the devices.

③ STATUS BEFORE INITIALIZATION OF MOS DEVICES

Note:

Power-on does not necessarily define initial status of MOS device. Production process of MOS does not define the initial operation status of the device. Immediately after the power source is turned ON, the devices with reset function have not yet been initialized. Hence, power-on does not guarantee out-pin levels, I/O settings or contents of registers. Device is not initialized until the reset signal is received. Reset operation must be executed immediately after power-on for devices having reset function.

VR3000, VR4400, VR5000, VR10000, VR10000L, VR10000 Series, VR12000, VR12000A, and VR12000L are trademarks of NEC Corporation.
R2000, R3000, and R6000 are trademarks of MIPS Computer Systems Inc.
MIPS is a registered trademark of MIPS Technologies, Inc. in the United States.
R4400, R8000, R10000, and R12000 are trademarks of MIPS Technologies, Inc.
UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

Exporting this product or equipment that includes this product may require a governmental license from the U.S.A. for some countries because this product utilizes technologies limited by the export control regulations of the U.S.A.

- **The information in this document is current as of January, 2001. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC's data sheets or data books, etc., for the most up-to-date specifications of NEC semiconductor products. Not all products and/or types are available in every country. Please check with an NEC sales representative for availability and additional information.**
- No part of this document may be copied or reproduced in any form or by any means without prior written consent of NEC. NEC assumes no responsibility for any errors that may appear in this document.
- NEC does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC semiconductor products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC or others.
- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of customer's equipment shall be done under the full responsibility of customer. NEC assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC endeavours to enhance the quality, reliability and safety of NEC semiconductor products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC semiconductor products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment, and anti-failure features.
- NEC semiconductor products are classified into the following three quality grades: "Standard", "Special" and "Specific". The "Specific" quality grade applies only to semiconductor products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of a semiconductor product depend on its quality grade, as indicated below. Customers must check the quality grade of each semiconductor product before using it in a particular application.

  "Standard": Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots

  "Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support)

  "Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

  The quality grade of NEC semiconductor products is "Standard" unless otherwise expressly specified in NEC's data sheets or data books, etc. If customers wish to use NEC semiconductor products in applications not intended by NEC, they must contact an NEC sales representative in advance to determine NEC's willingness to support a given application.

  (Note)

  (1) "NEC" as used in this statement means NEC Corporation and also includes its majority-owned subsidiaries.

  (2) "NEC semiconductor products" means any semiconductor product developed or manufactured by or for NEC (as defined above).

M8E 00. 4

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC product in your application, please contact the NEC office in your country to obtain a list of authorized representatives and distributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and other legal issues may also vary from country to country.

NEC Electronics Inc. (U.S.)
Santa Clara, California
Tel: 408-588-6000
800-366-9782
Fax: 408-588-6130
800-729-9288

NEC Electronics (Germany) GmbH
Duesseldorf, Germany
Tel: 0211-65 03 02
Fax: 0211-65 03 490

NEC Electronics (UK) Ltd.
Milton Keynes, UK
Tel: 01908-691-133
Fax: 01908-670-290

NEC Electronics Italiana s.r.l.
Milano, Italy
Tel: 02-66 75 41
Fax: 02-66 75 42 99

NEC Electronics (Germany) GmbH
Benelux Office
Eindhoven, The Netherlands
Tel: 040-2445845
Fax: 040-2444580

NEC Electronics (France) S.A.
Velizy-Villacoublay, France
Tel: 01-3067-5800
Fax: 01-3067-5899

NEC Electronics (France) S.A.
Madrid Office
Madrid, Spain
Tel: 091-504-2787
Fax: 091-504-2860

NEC Electronics (Germany) GmbH
Scandinavia Office
Taeby, Sweden
Tel: 08-63 80 820
Fax: 08-63 80 388

NEC Electronics Hong Kong Ltd.
Hong Kong
Tel: 2886-9318
Fax: 2886-9022/9044

NEC Electronics Hong Kong Ltd.
Seoul Branch
Seoul, Korea
Tel: 02-528-0303
Fax: 02-528-4411

NEC Electronics Singapore Pte. Ltd.
Novena Square, Singapore
Tel: 253-8311
Fax: 250-3583

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-2719-2377
Fax: 02-2719-5951

NEC do Brasil S.A.
Electron Devices Division
Guarulhos-SP, Brasil
Tel: 11-6462-6810
Fax: 11-6462-6829

J01.2

Main Revision in This Edition

Page	Description
Throughout	Addition of description about V_R12000A
p. 59	Modification and addition in **Table 3-4 Test Interface Signals**
p. 60	Addition of description in **Unused Inputs** of **3.4 Test Interface Signals**
p. 67	Modification in **Figure 4-6 Organization of Secondary Cache**
p. 96	Addition of description in **6.2 System Interface Frequencies**
p. 99	Modification in **Figure 6-2 System Interface Connections for Uniprocessor System**
p. 126	Modification in **Figure 6-6 Arbitration Signals for Uniprocessor System**
p. 158	Modification in **Table 6-27 Action Taken for External Coherency Requests that Target the R10000 Processor**
p. 176	Addition of description in **8.1 Initialization of Logical Registers**
p. 181	Modification and addition in **Table 8-1 Mode Bits**
p. 204	Addition of description in **10.1 Test Access Port (TAP)**
p. 204	Modification and addition of description in **TAP Controller (Input)** of **10.1 Test Access Port (TAP)**
pp. 217 to 240 in the previous edition	Deletion of **11. Electrical Specifications** and **12. Packaging**
p. 213	Modification of description in **Table 11-4 Description of EntryLo Registers' Fields**
p. 221	Addition of footnote in **Figure 11-11 Status Register**
p. 224	Modification of **Figure 11-12 Diagnostic Status Field**
p. 224	Addition of description in **Table 11-11 Status Register Diagnostic Status Bits**
p. 230	Modification in **Table 11-15 Config Register Field Definitions**
p. 230	Modification in **Figure 11-16 Config Register Format**
p. 235	Modification in **Figure 11-21 Diagnostic Register Format**
p. 236	Addition of description and **Table 11-18 Diagnostic Register Fields** in **11.19 Diagnostic Register (22)**
p. 247	Addition of **R12000 Implementation** in **11.20 Performance Counter Registers (25)**
p. 345	Deletion of description about Config register bits 22:23 in **Appendix B Differences between R10000 and R12000**
p. 346	Modification in **Table B-1 Mode Bits 12:9 (SysClkDiv)**
p. 346	Modification in **Table B-2 Mode Bits 21:19 (SCClkDiv)**
p. 347	Modification of description in **B.2 DSD (Delay Speculative Dirty)**
p. 356	Addition of **Appendix C Differences between R12000 and R12000A**

The mark ★ shows major revised points.

PREFACE

Readers

This manual targets users who intends to understand the functions of the VR10000 and VR12000, and to design application systems using this microprocessor.

Purpose

This manual introduces the architecture and hardware functions of the VR10000, VR12000 to users, following the organization described below.

Organization

This manual consists of the following contents:
- Introduction
- Cache
- Hardware
- Coprocessor 0
- Floating-point unit
- Memory management system
- Exception processing
- Instruction set details

How to read this manual

It is assumed that the reader of this manual has general knowledge in the fields of electric engineering, logic circuits, and microcomputers.

The R3000™ in this manual represents the VR3000™.
The R4400™ in this manual represents the VR4400™.
The R10000™ in this manual represents the VR10000 and VR10000L.
The R12000™ in this manual represents the VR12000, VR12000A, and VR12000L.
To learn about detailed function of a specific instruction.
→ Read **Chapter 12 Floating-Point Unit, Chapter 14 CPU Exceptions,** or refer to **VR5000™,VR10000 INSTRUCTION User's Manual** which is separately available.

To learn about the overall functions of the VR10000 and VR12000
→ Read this manual in sequential order.

To learn about electrical specifications,
→ Refer to **Data Sheet** which is separately available.

Unless otherwise specified, the R10000 is treated as the representative model throughout this document.

Legend

Data significance:	Higher on left and lower on right
Active low:	XXX*
Numeric representation:	binary ... XXXX or $XXXX_2$ decimal ... XXXX hexadecimal ... 0xXXXX
Important information	Underlined

Related Documents

The related documents indicated here may include preliminary version. However, preliminary versions are not marked as such.

- Data Sheet

μPD30700, 30700L, 30710 (VR10000, VR12000)	Data Sheet	U12703E

- User's Manual

VR5000, VR10000 INSTRUCTION User's Manual	U12754E

CONTENTS

1. Introduction to the R10000 Processor

- 1.1 MIPS Instruction Set Architecture (ISA) 18
- 1.2 What is a Superscalar Processor? 19
  - Pipeline and Superpipeline Architecture 19
  - Superscalar Architecture 19
- 1.3 What is an R10000 Microprocessor? 20
  - R10000 Superscalar Pipeline 21
  - Instruction Queues 22
  - Execution Pipelines 22
    - Load/store dependency is speculatively ignored (R12000) 22
    - 64-bit Integer ALU Pipeline 22
    - Load/Store Pipeline 23
    - 64-bit Floating-Point Pipeline 23
  - Functional Units 25
    - Increase in pre-decode buffering (R12000) 25
  - Primary Instruction Cache (I-cache) 25
  - Primary Data Cache (D-cache) 25
  - Branch Target Address Cache (R12000) 26
  - Instruction Decode And Rename Unit 26
  - Branch Unit 26
  - External Interfaces 27
    - Additional cycles for System Interface transactions (R12000) 27
- 1.4 Instruction Queues 28
    - FP and Integer-Queue Issue Policy (R12000) 28
  - Integer Queue 28
    - Address calculation for load/store instructions uses integer queue (R12000) 28
  - Floating-Point Queue 29
  - Address Queue 29
- 1.5 Program Order and Dependencies 31
  - Instruction Dependencies 31
  - Execution Order and Stalling 31
  - Branch Prediction and Speculative Execution 32
  - Resolving Operand Dependencies 32
  - Resolving Exception Dependencies 33
  - Strong Ordering 33
    - An Example of Strong Ordering 34
- 1.6 R10000 Pipelines 35
  - Stage 1 35
  - Stage 2 35
  - Stage 3 36
  - Stages 4-6 36
    - Floating-Point Multiplier (3-stage Pipeline) 36
    - Floating-Point Divide and Square-Root Units 36
    - Floating-Point Adder (3-stage Pipeline) 36
    - Integer ALU1 (1-stage Pipeline) 36
    - Integer ALU2 (1-stage Pipeline) 36
    - Address Calculation and Translation in the TLB 37
- 1.7 Implications of R10000 Microarchitecture on Software 38
  - Superscalar Instruction Issue 38
  - Speculative Execution 39
    - Side Effects of Speculative Execution 39
  - Nonblocking Caches 43

1.8 Performance 44
  User Instruction Latency and Repeat Rate 45
  Other Performance Issues 47
  Cache Performance 47

2. System Configurations

2.1 Uniprocessor Systems 50
2.2 Multiprocessor Systems 51
  Multiprocessor Systems Using Dedicated External Agents 51
  Multiprocessor Systems Using a Cluster Bus 52

3. Interface Signal Descriptions

3.1 Power Interface Signals 54
3.2 Secondary Cache Interface Signals 55
3.3 System Interface Signals 57
3.4 Test Interface Signals 59
  Unused Inputs 60

4. Cache Organization and Coherency

4.1 Primary Instruction Cache 62
4.2 Primary Data Cache 64
    DCache set locking relaxed (R12000) 65
4.3 Secondary Cache 67
  <R12000> 69
4.4 Cache Algorithms 70
  Descriptions of the Cache Algorithms 71
    Uncached 71
    Cacheable Noncoherent 71
    Cacheable Coherent Exclusive 71
    Cacheable Coherent Exclusive on Write 71
    Uncached Accelerated 72
4.5 Relationship Between Cached and Uncached Operations 73
4.6 Cache Algorithms and Processor Requests 74
4.7 Cache Block Ownership 75

5. Secondary Cache Interface

5.1 Tag and Data Arrays 77
5.2 Secondary Cache Interface Frequencies 78
5.3 Secondary Cache Indexing 79
  Indexing the Data Array 79
  Indexing the Tag Array 80
5.4 Secondary Cache Way Prediction Table 81
    Increased the Way Prediction Table (MRU table) to 16K single-bit entries 82
    Direct Cache Test Mode 82
5.5 Secondary Cache Tag 83
  SCTag(25:4), Physical Tag 83
  SCTag(3:2), PIdx 84

SCTag(1:0), Cache Block State 84
5.6 Read Sequences 85
4-Word Read Sequence 86
8-Word Read Sequence 87
16 or 32-Word Read Sequence 88
Tag Read Sequence 89
5.7 Write Sequences 90
4-Word Write Sequence 91
8-Word Write Sequence 92
16 or 32-Word Write Sequence 93
Tag Write Sequence 94

6. System Interface Operations

6.1 Request and Response Cycles 96
6.2 System Interface Frequencies 96
6.3 Register-to-Register Operation 96
6.4 System Interface Signals 97
6.5 Master and Slave States 97
6.6 Connecting to an External Agent 97
6.7 Cluster Bus 98
6.8 System Interface Connections 99
Uniprocessor System 99
Multiprocessor System Using Dedicated External Agents 100
Multiprocessor System Using the Cluster Bus 101
6.9 System Interface Requests and Responses 102
Processor Requests 102
External Responses 103
External Requests 103
Processor Responses 103
Outstanding Requests and Request Numbers 103
Request and Response Relationship 104
6.10 System Interface Buffers 105
Cluster Request Buffer 105
Cached Request Buffer 105
Incoming Buffer 106
Outgoing Buffer 107
Uncached Buffer 108
6.11 System Interface Flow Control 109
Processor Write and Eliminate Request Flow Control 109
Processor Read and Upgrade Request Flow Control 109
Processor Coherency Data Response Flow Control 109
External Request Flow Control 109
External Data Response Flow Control 109
6.12 System Interface Block Data Ordering 110
External Block Data Responses 110
Processor Coherency Data Responses 110
Processor Block Write Requests 110
6.13 System Interface Bus Encoding 111
SysCmd[11:0] Encoding 111
SysCmd[11] Encoding 111
SysCmd[10:0] Address Cycle Encoding 111
SysCmd[10:0] Data Cycle Encoding 115

SysCmd[11:0] Map ... 117
SysAD[63:0] Encoding ... 118
SysAD[63:0] Address Cycle Encoding ... 118
SysAD[63:0] Data Cycle Encoding ... 120
SysState[2:0] Encoding ... 120
SysResp[4:0] Encoding ... 121
6.14 Interrupts ... 121
Hardware Interrupts ... 121
Software Interrupts ... 122
Timer Interrupt ... 122
Nonmaskable Interrupt ... 122
6.15 Protocol Abbreviations ... 123
6.16 System Interface Arbitration ... 124
System Interface Arbitration Rules ... 125
Uniprocessor System ... 126
Multiprocessor System Using Cluster Bus ... 127
6.17 System Interface Request and Response Protocol ... 128
Processor Request Protocol ... 128
Processor Block Read Request Protocol ... 129
Processor Double/Single/Partial-Word Read Request Protocol ... 131
Processor Block Write Request Protocol ... 133
Processor Double/Single/Partial-Word Write Request Protocol ... 135
Processor Upgrade Request Protocol ... 137
Processor Eliminate Request Protocol ... 139
Processor Request Flow Control Protocol ... 141
External Response Protocol ... 143
External Block Data Response Protocol ... 143
External Double/Single/Partial-Word Data Response Protocol ... 145
External Completion Response Protocol ... 146
External Request Protocol ... 148
External Intervention Request Protocol ... 149
External Allocate Request Number Request Protocol ... 150
External Invalidate Request Protocol ... 151
External Interrupt Request Protocol ... 152
Processor Response Protocol ... 153
Processor Coherency State Response Protocol ... 154
Processor Coherency Data Response Protocol ... 155
6.18 System Interface Coherency ... 157
External Intervention Shared Request ... 157
External Intervention Exclusive Request ... 157
External Invalidate Request ... 157
External Coherency Request Action ... 158
Coherency Conflicts ... 159
Internal Coherency Conflicts ... 159
External Coherency Conflicts ... 160
External Coherency Request Latency ... 162
SysGblPerf* Signal ... 164
6.19 Cluster Bus Operation ... 164
6.20 Support for I/O ... 168
6.21 Support for External Duplicate Tags ... 168
6.22 Support for a Directory-Based Coherency Protocol ... 169
6.23 Support for Uncached Attribute ... 169
6.24 Support for Hardware Emulation ... 170

7. Clock Signals

7.1 System Interface Clock and Internal Processor Clock Domains 172
7.2 Secondary Cache Clock 173
7.3 Phase-Locked-Loop 174

8. Initialization

8.1 Initialization of Logical Registers........176
8.2 Power-On Reset Sequence 176
8.3 Cold Reset Sequence........178
8.4 Soft Reset Sequence........179
8.5 Mode Bits........180

9. Error Protection and Handling

9.1 Correctable Errors 185
9.2 Uncorrectable Errors 186
9.3 Propagation of Uncorrectable Errors 187
9.4 Cache Error Exception 188
9.5 CP0 CacheErr Register EW Bit 189
9.6 CP0 Status Register DE Bit........189
9.7 CACHE Instruction........189
9.8 Error Protection Schemes Used by R10000........190
    Parity........190
    Sparse Encoding 190
    ECC 190
9.9 Primary Instruction Cache Error Protection and Handling........191
    Error Protection 191
    Error Handling........191
9.10 Primary Data Cache Error Protection and Handling........192
    Error Protection 192
    Error Handling........192
9.11 Secondary Cache Error Protection and Handling 193
    Error Protection 193
    Error Handling........193
        Data Array 193
        Tag Array 196
9.12 System Interface Error Protection and Handling 197
    Error Protection 197
    Error Handling........198
        SysCmd(11:0) Bus 198
        SysAD(63:0) Bus 199
        SysState(2:0) Bus 201
        SysResp(4:0) Bus 201
    Protocol Observation........202

10. JTAG Interface Operation

10.1 Test Access Port (TAP)........204
    TAP Controller (Input)........204
10.2 Instruction Register 205

10.3 Bypass Register 205
10.4 Boundary Scan Register 206

11. Coprocessor 0

11.1 Index Register (0) 211
11.2 Random Register (1) 212
11.3 EntryLo0 (2) and EntryLo1 (3) Registers 213
11.4 Context Register (4) 215
11.5 PageMask Register (5) 216
11.6 Wired Register (6) 217
11.7 BadVAddr Register (8) 218
11.8 Count and Compare Registers (9 and 11) 218
11.9 EntryHi Register (10) 219
11.10 Status Register (12) 220
    Status Register Fields 222
    Diagnostic Status Field 223
    Coprocessor Accessibility 225
11.11 Cause Register (13) 226
11.12 Exception Program Counter (14) 228
11.13 Processor Revision Identifier (PRId) Register (15) 229
11.14 Config Register (16) 230
11.15 Load Linked Address (LLAddr) Register (17) 231
11.16 WatchLo (18) and WatchHi (19) Registers 232
11.17 XContext Register (20) 233
11.18 FrameMask Register (21) 234
11.19 Diagnostic Register (22) 235
11.20 Performance Counter Registers (25) 238
    R10000 Implementation 238
        Details of Counting Events 241
    R12000 Implementation 247
        Details of Counting Events 257
11.21 ECC Register (26) 262
11.22 CacheErr Register (27) 263
    CacheErr Register Format for Primary Instruction Cache Errors 263
    CacheErr Register Format for Primary Data Cache Errors 264
    CacheErr Register Format for Secondary Cache Errors 265
    CacheErr Register Format for System Interface Errors 266
11.23 TagLo (28) and TagHi (29) Registers 267
    CacheOp is Index Load/Store Tag 267
        Primary Instruction Cache Operation 268
        Primary Data Cache Operation 268
        Secondary Cache Operation 270
    CacheOp is Index Load/Store Data 271
        Primary Instruction Cache Operation 271
        Primary Data Cache Operation 272
        Secondary Cache Operation 272
11.24 ErrorEPC Register (30) 273

12. Floating-Point Unit

12.1 Floating-Point Unit Operations 275

12.2 Floating-Point Unit Control 276
    Eliminate traps for Denorm/NaN FP inputs (R12000) 276
12.3 Floating-Point General Registers (FGRs) 277
  32- and 64-Bit Operations 277
  Load and Store Operations 278
12.4 Floating-Point Control Registers 281
  Floating-Point Implementation and Revision Register 281
  Floating-Point Status Register (FSR) 282
    Bit Descriptions of the FSR 283
    Loading the FSR 284

13. Memory Management

13.1 Processor Modes 286
  Processor Operating Modes 286
  Addressing Modes 287
13.2 Virtual Address Space 287
  User Mode Operations 288
    32-bit User Mode (useg) 289
    64-bit User Mode (xuseg) 289
  Supervisor Mode Operations 290
    32-bit Supervisor Mode, User Space (suseg) 290
    32-bit Supervisor Mode, Supervisor Space (sseg) 291
    64-bit Supervisor Mode, User Space (xsuseg) 291
    64-bit Supervisor Mode, Current Supervisor Space (xsseg) 291
    64-bit Supervisor Mode, Separate Supervisor Space (csseg) 291
  Kernel Mode Operations 292
    32-bit Kernel Mode, User Space (kuseg) 293
    32-bit Kernel Mode, Kernel Space 0 (kseg0) 293
    32-bit Kernel Mode, Kernel Space 1 (kseg1) 293
    32-bit Kernel Mode, Supervisor Space (ksseg) 293
    32-bit Kernel Mode, Kernel Space 3 (kseg3) 293
    64-bit Kernel Mode, User Space (xkuseg) 294
    64-bit Kernel Mode, Current Supervisor Space (xksseg) 294
    64-bit Kernel Mode, Physical Spaces (xkphys) 294
    64-bit Kernel Mode, Kernel Space (xkseg) 296
    64-bit Kernel Mode, Compatibility Spaces (ckseg1:0, cksseg, ckseg3) 296
  Address Space Access Privilege Differences Between the R4400 and R10000 296
13.3 Virtual Address Translation 298
  Virtual Pages 298
  Virtual Page Size Encodings 298
  Using the TLB 299
  Cache Algorithm Field 299
  Format of a TLB Entry 299
  Address Translation 300
  Address Space Identification (ASID) 300
  Global Processes (G) 300
  Avoiding TLB Conflict 300

14. CPU Exceptions

14.1 Causing and Returning from an Exception 302
14.2 Exception Vector Locations 302

14.3 TLB Refill Vector Selection 303
  Priority of Exceptions....305
  Cold Reset Exception 306
  Soft Reset Exception 307
  NMI Exception 309
  Address Error Exception 310
  TLB Exceptions....311
    TLB Refill Exception 312
    TLB Invalid Exception....313
    TLB Modified Exception 314
  Cache Error Exception 315
  Virtual Coherency Exception 315
  Bus Error Exception 316
  Integer Overflow Exception 317
  Trap Exception 318
  System Call Exception 319
  Breakpoint Exception 320
  Reserved Instruction Exception....321
  Coprocessor Unusable Exception....322
  Floating-Point Exception....323
  Watch Exception 324
  Interrupt Exception....325
14.4 MIPSIV Instructions 326
14.5 COP0 Instructions....327
14.6 COP1 Instructions....327
14.7 COP2 Instructions....327

15. Cache Test Mode

15.1 Interface Signals....329
15.2 System Interface Clock Divisor 329
15.3 Entering Cache Test Mode....330
15.4 Exit Sequence....331
15.5 SysAD(63:0) Encoding....332
15.6 Cache Test Mode Protocol....333
  Normal Write Protocol 333
  Auto-Increment Write Protocol....334
  Normal Read Protocol 335
  Auto-Increment Read Protocol....336

Appendix A Glossary

A.1 Superscalar Processor....338
A.2 Pipeline....338
A.3 Pipeline Latency....338
A.4 Pipeline Repeat Rate 338
A.5 Out-of-Order Execution 338
A.6 Dynamic Scheduling 339
A.7 Instruction Fetch, Decode, Issue, Execution, Completion, and Graduation 339
A.8 Active List....339
A.9 Free List and Busy Registers....340
A.10 Register Renaming....340

A.11 Nonblocking Loads and Stores 341
A.12 Speculative Branching 342
A.13 Logical and Physical Registers 343
A.14 Register Files........ 343
A.15 ANDES Architecture 344

Appendix B Differences between R10000 and R12000

B.1 Mode bits changed in R12000........ 346
B.2 DSD (Delay Speculative Dirty) 347
B.3 Changes in the Branch Diag Register 348
B.4 Eliminate traps for Denorm/NaN FP inputs........ 349
B.5 Increase in pre-decode buffering........ 350
B.6 Increased penalty for indirect branches........ 350
B.7 Addition of a Branch Target Address Cache 350
B.8 Use of global history in branch-prediction........ 351
B.9 Increase in branch prediction table size 351
B.10 Address calculation for load/store instructions uses integer queue 351
B.11 Load/store dependency is speculatively ignored........ 351
B.12 DCache set locking relaxed........ 352
B.13 SC refill blocking reduced 352
B.14 Increased the Way Prediction Table (MRU table) to 16K single-bit entries 352
B.15 Additional cycles for System Interface transactions........ 352
B.16 FP and Integer-Queue Issue Policy........ 353
B.17 Active List entries are increased to 48 353
B.18 Cache Error inhibits graduation 353
B.19 Changed Spare(1, 3) pins to NC (No Connection) 353
B.20 CacheOp Index Write Back Invalidate(D) also clears Primary Tag........ 353
B.21 Summary of the differences 354

★ Appendix C Differences between R12000 and R12000A

C.1 Mode bits changed in R12000A........ 357
C.2 Changes in the Performance Counter Registers........ 358
C.3 Summary of the differences 358

Appendix D Index

1. Introduction to the R10000 Processor

This user's manual describes the R10000 superscalar microprocessor for the system designer, paying special attention to the external interface and the transfer protocols.

This chapter describes the following:

- MIPS™ ISA
- what makes a generic superscalar microprocessor
- specifics of the R10000 superscalar microprocessor
- implementation-specific CPU instructions

1.1 MIPS Instruction Set Architecture (ISA)

MIPS has defined an instruction set architecture (ISA), implemented in the following sets of CPU designs:

- MIPS I, implemented in the R2000™ and R3000
- MIPS II, implemented in the R6000™
- MIPS III, implemented in the R4400
- MIPS IV, implemented in the R8000™ and R10000

The original MIPS I CPU ISA has been extended forward three times, as shown in Figure 1-1; each extension is backward compatible. The ISA extensions are inclusive; each new architecture level (or version) includes the former levels.[†]

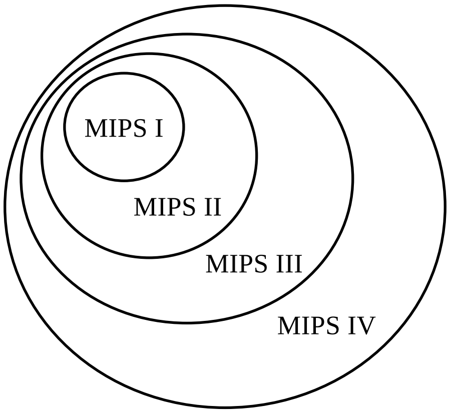


Figure 1-1 MIPS ISA with Extensions

The practical result is that a processor implementing MIPS IV is also able to run MIPS I, MIPS II, or MIPS III binary programs without change.

[†] For more ISA information, please refer to the *MIPS IV Instruction Set Architecture*, published by MIPS Technologies, and written by Charles Price. Contact information is provided both in the *Preface*, and inside the front cover, of this manual.

1.2 What is a Superscalar Processor?

A superscalar processor is one that can fetch, execute and complete more than one instruction in parallel.

Pipeline and Superpipeline Architecture

Previous MIPS processors had linear pipeline architectures; an example of such a linear pipeline is the R4400 superpipeline, shown in Figure 1-2. In the R4400 superpipeline architecture, an instruction is executed each cycle of the pipeline clock (PCycle), or each *pipe stage*.

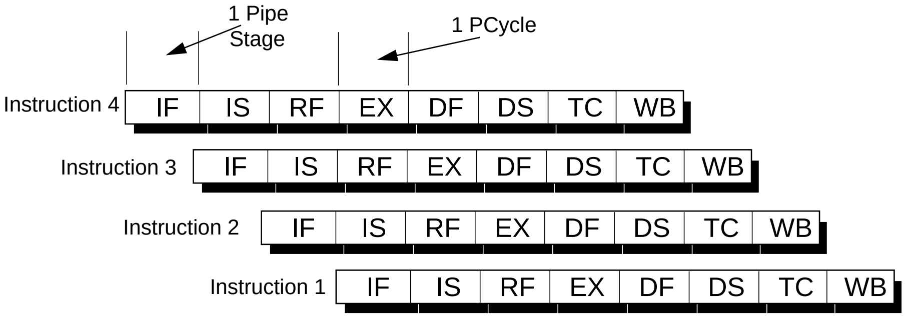

1 Pipe Stage
1 PCycle
Instruction 4
Instruction 3
Instruction 2
Instruction 1
IF
IS
RF
EX
DF
DS
TC
WB

Figure 1-2 R4400 Pipeline

Superscalar Architecture

The structure of 4-way superscalar pipeline is shown in Figure 1-3. At each stage, four instructions are handled in parallel. Note that there is only one EX stage for integers.

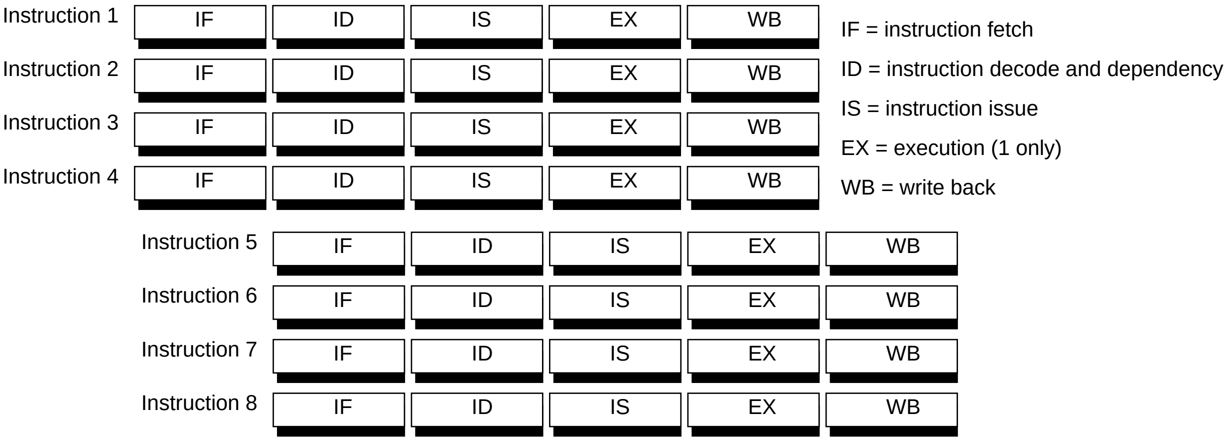

Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Instruction 6
Instruction 7
Instruction 8
IF
ID
IS
EX
WB
IF = instruction fetch
ID = instruction decode and dependency
IS = instruction issue
EX = execution (1 only)
WB = write back

Figure 1-3 4-Way Superscalar Pipeline

1.3 What is an R10000 Microprocessor?

The R10000 processor is a single-chip superscalar RISC microprocessor that is a follow-on to the MIPS RISC processor family that includes, chronologically, the R2000, R3000, R6000, R4400, and R8000.

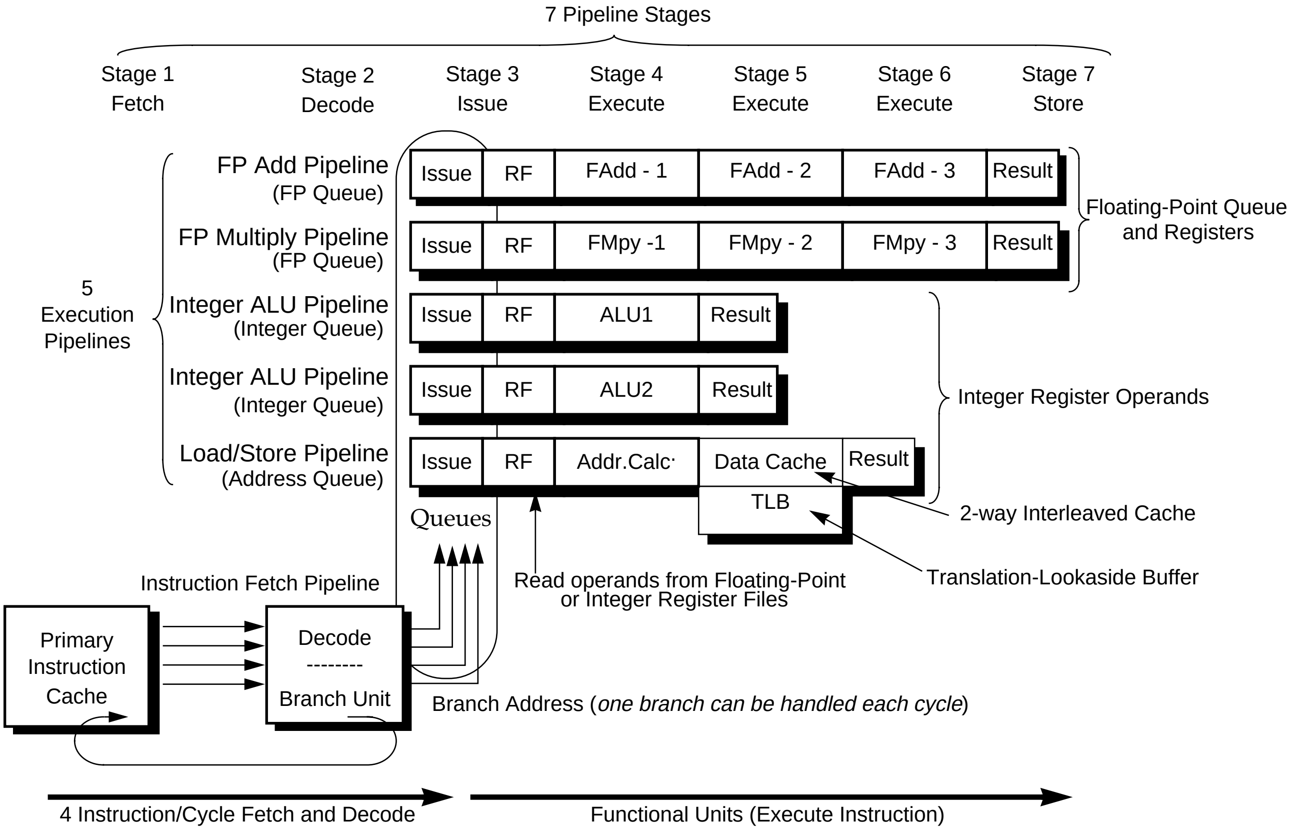
The R10000 processor uses the MIPS ANDES architecture, or *Architecture with Non-sequential Dynamic Execution Scheduling*.

The R10000 processor has the following major features (terms in **bold** are defined in the Glossary):

- it implements the 64-bit MIPS IV instruction set architecture (ISA)
- it can decode four instructions each pipeline cycle, appending them to one of three *instruction queues*
- it has five *execution pipelines* connected to separate internal integer and floating-point execution (or *functional*) units
- it uses **dynamic instruction scheduling** and **out-of-order execution**
- it uses speculative instruction issue (also termed “**speculative branching**”)
- it uses a precise exception model (exceptions can be traced back to the instruction that caused them)
- it uses **non-blocking caches**
- it has separate on-chip 32-Kbyte primary instruction and data caches
- it has individually-optimized secondary cache and System interface ports
- it has an internal controller for the external secondary cache
- it has an internal System interface controller with multiprocessor support

R10000 Superscalar Pipeline

The R10000 superscalar processor fetches and decodes four instructions in parallel each cycle (or pipeline stage). Each pipeline includes stages for fetching (stage 1 in Figure 1-4), decoding (stage 2) issuing instructions (stage 3), reading register operands (stage 3), executing instructions (stages 4 through 6), and storing results (stage 7).


7 Pipeline Stages
Stage 1 Fetch
Stage 2 Decode
Stage 3 Issue
Stage 4 Execute
Stage 5 Execute
Stage 6 Execute
Stage 7 Store
5 Execution Pipelines
FP Add Pipeline (FP Queue)
FP Multiply Pipeline (FP Queue)
Integer ALU Pipeline (Integer Queue)
Integer ALU Pipeline (Integer Queue)
Load/Store Pipeline (Address Queue)
Issue
RF
FAdd - 1
FAdd - 2
FAdd - 3
Result
FMpy -1
FMpy - 2
FMpy - 3
ALU1
ALU2
Addr.Calc·
Data Cache
TLB
Floating-Point Queue and Registers
Integer Register Operands
2-way Interleaved Cache
Translation-Lookaside Buffer
Queues
Read operands from Floating-Point or Integer Register Files
Instruction Fetch Pipeline
Primary Instruction Cache
Decode

Branch Unit
Branch Address (one branch can be handled each cycle)
4 Instruction/Cycle Fetch and Decode
Functional Units (Execute Instruction)

Figure 1-4 Superscalar Pipeline Architecture in the R10000

Instruction Queues

As shown in Figure 1-4, each instruction decoded in stage 2 is appended to one of three instruction *queues*:

- integer queue
- address queue
- floating-point queue

Execution Pipelines

The three instruction queues can issue (see the Glossary for a definition of *issue*) one new instruction per cycle to each of the five execution pipelines:

- the integer queue issues instructions to the two integer ALU pipelines
- the address queue issues one instruction to the Load/Store Unit pipeline
- the floating-point queue issues instructions to the floating-point adder and multiplier pipelines

A sixth pipeline, the fetch pipeline, reads and decodes instructions from the instruction cache.

Load/store dependency is speculatively ignored (R12000)

When a load follows a store in program-order, and the address of the load is known to the Address Queue (AQ) before the address of the store, then the AQ may speculatively issue the load to tag-check and data access. When the address of the store is determined, the AQ can undo the effects of the load through the use of the “soft-exception” mechanism. Since almost all loads which are actually dependent on previous stores use the same registers to form their addresses, normally either the two instructions are independent, or their addresses are resolved in program order, so the soft-exception should occur rarely.

64-bit Integer ALU Pipeline

The 64-bit integer pipeline has the following characteristics:

- it has a 16-entry integer instruction queue that dynamically issues instructions
- it has a 64-bit 64-location integer physical register file, with seven read and three write ports (32 logical registers; see *register renaming* in the Glossary)
- it has two 64-bit arithmetic logic units:
    - ALU1 contains an arithmetic-logic unit, shifter, and integer branch comparator
    - ALU2 contains an arithmetic-logic unit, integer multiplier, and divider

Load/Store Pipeline

The load/store pipeline has the following characteristics:

- it has a 16-entry address queue that dynamically issues instructions, and uses the integer register file for base and index registers
- it has a 16-entry address stack for use by non-blocking loads and stores
- it has a 44-bit virtual address calculation unit
- it has a 64-entry fully associative **Translation-Lookaside Buffer** (TLB), which converts virtual addresses to physical addresses, using a 40-bit physical address. Each entry maps two pages, with sizes ranging from 4 Kbytes to 16 Mbytes, in powers of 4.

64-bit Floating-Point Pipeline

The 64-bit floating-point pipeline has the following characteristics:

- it has a 16-entry instruction queue, with dynamic issue
- it has a 64-bit 64-location floating-point physical register file, with five read and three write ports (32 logical registers)
- it has a 64-bit parallel multiply unit (3-cycle pipeline with 2-cycle latency) which also performs move instructions
- it has a 64-bit add unit (3-cycle pipeline with 2-cycle latency) which handles addition, subtraction, and miscellaneous floating-point operations
- it has separate 64-bit divide and square-root units which can operate concurrently (these units share their issue and completion logic with the floating-point multiplier)

A block diagram of the processor and its interfaces is shown in Figure 1-5, followed by a description of its major logical blocks.

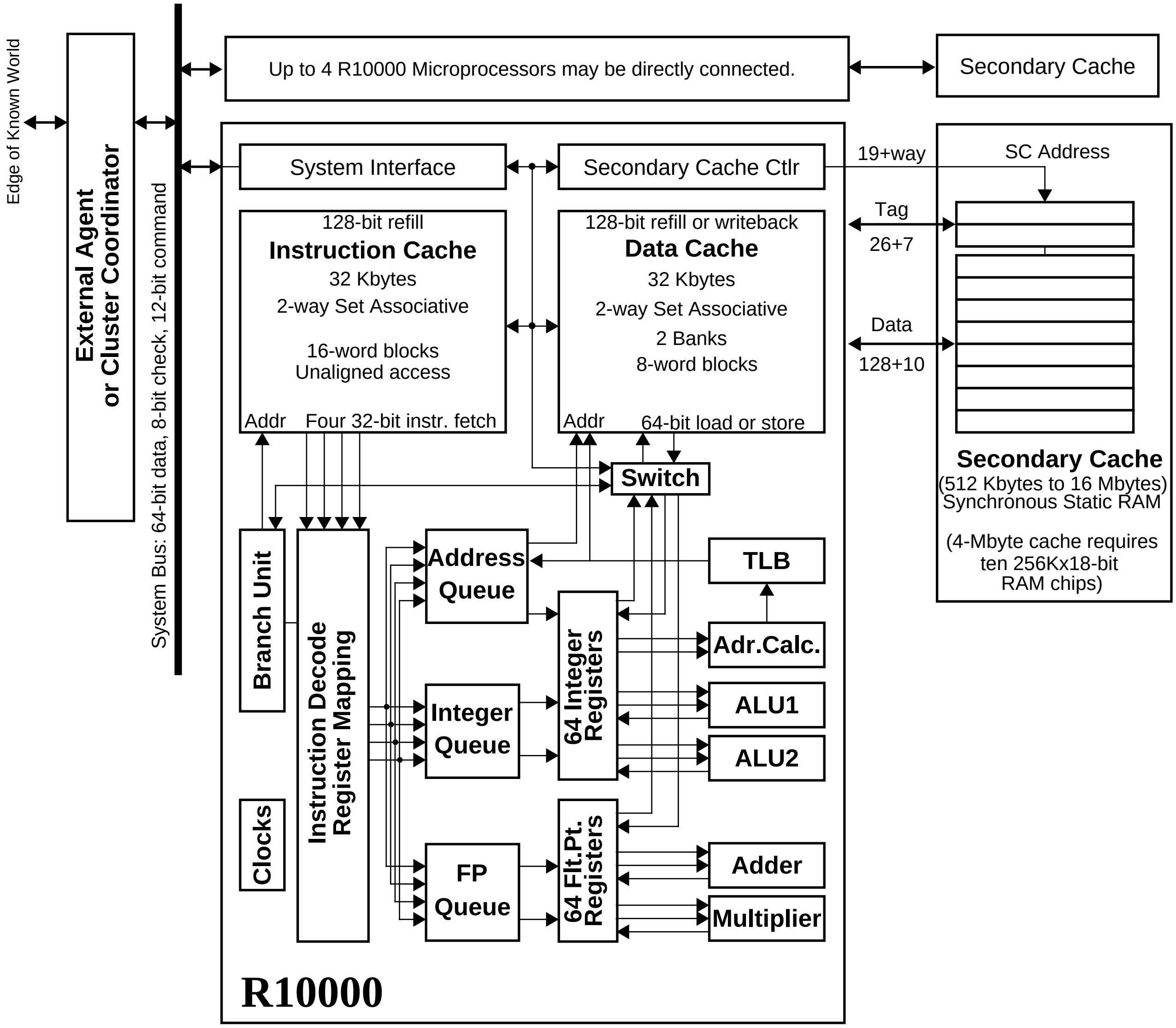

Edge of Known World
External Agent or Cluster Coordinator
System Bus: 64-bit data, 8-bit check, 12-bit command
Up to 4 R10000 Microprocessors may be directly connected.
Secondary Cache
System Interface
Secondary Cache Ctlr
19+way
SC Address
Tag
26+7
Data
128+10
128-bit refill
Instruction Cache
32 Kbytes
2-way Set Associative
16-word blocks
Unaligned access
Addr
Four 32-bit instr. fetch
128-bit refill or writeback
Data Cache
32 Kbytes
2-way Set Associative
2 Banks
8-word blocks
Addr
64-bit load or store
Switch
Secondary Cache
(512 Kbytes to 16 Mbytes)
Synchronous Static RAM
(4-Mbyte cache requires ten 256Kx18-bit RAM chips)
Branch Unit
Instruction Decode Register Mapping
Address Queue
TLB
Adr.Calc.
64 Integer Registers
Integer Queue
ALU1
ALU2
Clocks
FP Queue
64 Flt.Pt. Registers
Adder
Multiplier
R10000

Figure 1-5 Block Diagram of the R10000 Processor

Functional Units

The five execution pipelines allow overlapped instruction execution by issuing instructions to the following five functional units:

- two integer ALUs (ALU1 and ALU2)
- the Load/Store unit (address calculate)
- the floating-point adder
- the floating-point multiplier

There are also three “iterative” units to compute more complex results:

- Integer multiply and divide operations are performed by an Integer Multiply/ Divide execution unit; these instructions are issued to ALU2. ALU2 remains busy for the duration of the divide.
- Floating-point divides are performed by the Divide execution unit; these instructions are issued to the floating-point multiplier.
- Floating-point square root are performed by the Square-root execution unit; these instructions are issued to the floating-point multiplier.

Increase in pre-decode buffering (R12000)

Up to 12 instruction may be buffered before being decoded. This should normally be invisible to the end user, but can be important when debugging systems in uncached-mode, since fetch and decode are now further de-coupled.

Primary Instruction Cache (I-cache)

The primary instruction cache has the following characteristics:

- it contains 32 Kbytes, organized into 16-word blocks, is 2-way set associative, using a **least-recently used** (LRU) replacement algorithm
- it reads four consecutive instructions per cycle, beginning on any word boundary within a cache block, but cannot fetch across a block boundary.
- its instructions are predecoded, its fields are rearranged, and a 4-bit unit select code is appended
- it checks parity on each word
- it permits non-blocking instruction fetch

Primary Data Cache (D-cache)

The primary data cache has the following characteristics:

- it has two interleaved arrays (two 16 Kbyte ways)
- it contains 32 Kbytes, organized into 8-word blocks, is 2-way set associative, using an LRU replacement algorithm.
- it handles 64-bit load/store operations
- it handles 128-bit refill or write-back operations
- it permits non-blocking loads and stores
- it checks parity on each byte

Branch Target Address Cache (R12000)

This 32-entry two-way set-associative cache holds the target addresses of previously-taken branches. When a branch is executed a hit in the BTAC eliminates the one-cycle fetch bubble with the R10000 experiences for every taken branch. However, if a branch which hits in the BTAC is actually predicted not-taken, then a one cycle fetch bubble is introduced where none was present before. Performance simulations indicate that the BTAC is a net win, but because of its “mixed-blessing” nature, a mechanism has been provided to disable it via software. (See description of changes to diag register).

Instruction Decode And Rename Unit

The instruction decode and rename unit has the following characteristics:

- it processes 4 instructions in parallel
- it replaces logical register numbers with physical register numbers (register renaming)
    - it maps integer registers into a 33-word-by-6-bit mapping table that has 4 write and 12 read ports
    - it maps floating-point registers into a 32-word-by-6-bit mapping table that has 4 write and 16 read ports
- it has a 32-entry active list of all instructions within the pipeline.

Branch Unit

The branch unit has the following characteristics:

- it allows one branch per cycle
- conditional branches can be executed speculatively, up to 4-deep
- it has a 44-bit adder to compute branch addresses
- it has a 4-quadword branch-resume buffer, used for reversing mispredicted speculatively-taken branches
- the Branch Return Cache contains four instructions following a subroutine call, for rapid use when returning from leaf subroutines
- it has program trace RAM that stores the program counter for each instruction in the pipeline

External Interfaces

The external interfaces have the following characteristics:

- a 64-bit System interface allows direct-connection for 2-way to 4-way multiprocessor systems. 8-bit ECC Error Check and Correction is made on address and data transfers.
- a secondary cache interface with 128-bit data path and tag fields. 9-bit ECC Error Check and Correction is made on data quadwords, 7-bit ECC is made on tag words. It allows connection to an external secondary cache that can range from 512 Kbytes to 16 Mbytes, using external static RAMs. The secondary cache can be organized into either 16- or 32-word blocks, and is 2-way set associative.

Bit definitions are given in Chapter 3.

Additional cycles for System Interface transactions (R12000)

All transactions which go through the system interface unit (in particular, SCache refills and writebacks) have one additional CPU-clock of latency added to them.

1.4 Instruction Queues

The processor keeps decoded instructions in three instruction queues, which dynamically issue instructions to the execution units. The queues allow the processor to fetch instructions at its maximum rate, without stalling because of instruction conflicts or dependencies.

Each queue uses instruction tags to keep track of the instruction in each execution pipeline stage. These tags set a *Done* bit in the active list as each instruction is completed.

FP and Integer-Queue Issue Policy (R12000)

The integer and floating-point queues are altered so that they are now composed of two 8-entry banks. Instructions are issued into the two banks in an alternating fashion. Each bank independently nominates instructions for the functional units. For each FU, the banks nominate the oldest instruction they contain which is ready to execute. If both banks nominate an instruction for a given FU, a winner is chosen by a priority bit which alternates between the two banks on each cycle.

Integer Queue

The integer queue issues instructions to the two integer arithmetic units: ALU1 and ALU2.

The integer queue contains 16 instruction entries. Up to four instructions may be written during each cycle; newly-decoded integer instructions are written into empty entries in no particular order. Instructions remain in this queue only until they have been issued to an ALU.

Branch and shift instructions can be issued only to ALU1. Integer multiply and divide instructions can be issued only to ALU2. Other integer instructions can be issued to either ALU.

The integer queue controls six dedicated ports to the integer register file: two operand read ports and a destination write port for each ALU.

Address calculation for load/store instructions uses integer queue (R12000)

When load, store, cacheop, or prefetch instructions are decoded, they are sent to both the AQ and IQ units. The IQ treats the address-calculate unit as a third “ALU” and issues instructions to it. When an instruction completes address calculation, the results are forwarded to the AQ. Unlike previously, if an address instruction must be retried for any reason, address calculation is not redone. If the address queue is full, but the integer queue has free entries at the time a load/store instruction is decoded, the load/store is sent only to the integer queue. When the address queue has an available entry the calculated address is forwarded to that entry and the remainder of the load/store execution continues.

Floating-Point Queue

The floating-point queue issues instructions to the floating-point multiplier and the floating-point adder.

The floating-point queue contains 16 instruction entries. Up to four instructions may be written during each cycle; newly-decoded floating-point instructions are written into empty entries in random order. Instructions remain in this queue only until they have been issued to a floating-point execution unit.

The floating-point queue controls six dedicated ports to the floating-point register file: two operand read ports and a destination port for each execution unit.

The floating-point queue uses the multiplier's issue port to issue instructions to the square-root and divide units. These instructions also share the multiplier's register ports.

The floating-point queue contains simple sequencing logic for multiple-pass instructions such as Multiply-Add. These instructions require one pass through the multiplier, then one pass through the adder.

Address Queue

The address queue issues instructions to the load/store unit.

The address queue contains 16 instruction entries. Unlike the other two queues, the address queue is organized as a circular **First-In First-Out** (FIFO) buffer. A newly decoded load/store instruction is written into the next available sequential empty entry; up to four instructions may be written during each cycle.

The FIFO order maintains the program's original instruction sequence so that memory address dependencies may be easily computed.

Instructions remain in this queue until they have graduated; they cannot be deleted immediately after being issued, since the load/store unit may not be able to complete the operation immediately.

The address queue contains more complex control logic than the other queues. An issued instruction may fail to complete because of a memory dependency, a cache miss, or a resource conflict; in these cases, the queue must continue to reissue the instruction until it is completed.

The address queue has three issue ports:

- First, it issues each instruction once to the address calculation unit. This unit uses a 2-stage pipeline to compute the instruction's memory address and to translate it in the TLB. Addresses are stored in the address stack and in the queue's dependency logic. This port controls two dedicated read ports to the integer register file. If the cache is available, it is accessed at the same time as the TLB. A tag check can be performed even if the data array is busy.

- Second, the address queue can re-issue accesses to the data cache. The queue allocates usage of the four sections of the cache, which consist of the tag and data sections of the two cache banks. Load and store instructions begin with a tag check cycle, which checks to see if the desired address is already in cache. If it is not, a refill operation is initiated, and this instruction waits until it has completed. Load instructions also read and align a doubleword value from the data array. This access may be either concurrent to or subsequent to the tag check. If the data is present and no dependencies exist, the instruction is marked *done* in the queue.
- Third, the address queue can issue store instructions to the data cache. A store instruction may not modify the data cache until it graduates. Only one store can graduate per cycle, but it may be anywhere within the four oldest instructions, if all previous instructions are already completed.

The access and store ports share four register file ports (integer read and write, floating-point read and write). These shared ports are also used for Jump and Link and Jump Register instructions, and for move instructions between the integer and register files.

1.5 Program Order and Dependencies

From a programmer's perspective, instructions appear to execute sequentially, since they are fetched and graduated in program order (the order they are presented to the processor by software). When an instruction stores a new value in its destination register, that new value is immediately available for use by subsequent instructions.

Internal to the processor, however, instructions are executed dynamically, and some results may not be available for many cycles; yet the hardware must behave as if each instruction is executed sequentially.

This section describes various conditions and dependencies that can arise from them in pipeline operation, including:

- instruction dependencies
- execution order and stalling
- branch prediction and speculative execution
- resolving operand dependencies
- resolving exception dependencies

Instruction Dependencies

Each instruction depends on all previous instructions which produced its operands, because it cannot begin execution until those operands become valid. These dependencies determine the order in which instructions can be executed.

Execution Order and Stalling

The actual execution order depends on the processor's organization; in a typical pipelined processor, instructions are executed only in program order. That is, the next sequential instruction may begin execution during the next cycle, if all of its operands are valid. Otherwise, the pipeline stalls until the operands do become valid.

Since instructions execute in order, stalls usually delay all subsequent instructions.

A clever compiler can improve performance by re-arranging instructions to reduce the frequency of these stall cycles.

- In an *in-order superscalar processor*, several consecutive instructions may begin execution simultaneously, if all their operands are valid, but the processor stalls at any instruction whose operands are still busy.
- In an *out-of-order superscalar processor*, such as the R10000, instructions are decoded and stored in queues. Each instruction is eligible to begin execution as soon as its operands become valid, independent of the original instruction sequence. In effect, the hardware rearranges instructions to keep its execution units busy. This process is called *dynamic issuing*.

Branch Prediction and Speculative Execution

Although one or more instructions may begin execution during each cycle, each instruction takes several (or many) cycles to complete. Thus, when a branch instruction is decoded, its branch condition may not yet be known. However, the R10000 processor can *predict* whether the branch is taken, and then continue decoding and executing subsequent instructions along the predicted path.

When a branch prediction is wrong, the processor must back up to the original branch and take the other path. This technique is called *speculative execution*. Whenever the processor discovers a mispredicted branch, it aborts all speculatively-executed instructions and restores the processor's state to the state it held before the branch. However, the cache state is not restored (see the section titled "Side Effects of Speculative Execution").

Branch prediction can be controlled by the CP0 *Diagnostic* register. Branch Likely instructions are always predicted as taken, which also means the instruction in the delay slot of the Branch Likely instruction will always be speculatively executed. Since the branch predictor is neither used nor updated by branch-likely instructions, these instructions do not affect the prediction of "normal" conditional branches.

Resolving Operand Dependencies

Operands include registers, memory, and condition bits. Each operand type has its own dependency logic. In the R10000 processor, dependencies are resolved in the following manner:

- register dependencies are resolved by using *register renaming* and the associative comparator circuitry in the queues
- memory dependencies are resolved in the Load/Store Unit
- condition bit dependencies are resolved in the active list and instruction queues

Resolving Exception Dependencies

In addition to operand dependencies, each instruction is implicitly dependent upon any previous instruction that generates an exception. Exceptions are caused whenever an instruction cannot be properly completed, and are usually due to either an untranslated virtual address or an erroneous operand.

The processor design implements *precise exceptions*, by:

- identifying the instruction which caused the exception
- preventing the exception-causing instruction from graduating
- aborting all subsequent instructions

Thus, all register values remain the same as if instructions were executed singly. Effectively, all previous instructions are completed, but the faulting instruction and all subsequent instructions do not modify any values.

Strong Ordering

A multiprocessor system that exhibits the same behavior as a uniprocessor system in a multiprogramming environment is said to be *strongly ordered*.

The R10000 processor behaves as if strong ordering is implemented, although it does not actually execute all memory operations in strict program order.

In the R10000 processor, store operations remain pending until the store instruction is ready to graduate. Thus, stores are executed in program order, and memory values are precise following any exception.

For improved performance however, cached load operations my occur in any order, subject to memory dependencies on pending store instructions. To maintain the appearance of strong ordering, the processor detects whenever the reordering of a cached load might alter the operation of the program, backs up, and then re-executes the affected load instructions. Specifically, whenever a primary data cache block is invalidated due to an external coherency request, its index is compared with all outstanding load instructions. If there is a match and the load has been completed, the load is prevented from graduating. When it is ready to graduate, the entire pipeline is flushed, and the processor is restored to the state it had before the load was decoded.

An uncached or uncached accelerated load or store instruction is executed when the instruction is ready to graduate. This guarantees strong ordering for uncached accesses.

Since the R10000 processor behaves as if it implemented strong ordering, a suitable system design allows the processor to be used to create a shared-memory multiprocessor system with strong ordering.

An Example of Strong Ordering

Given that locations X and Y have no particular relationship—that is, they are not in the same cache block—an example of strong ordering is as follows:

- Processor A performs a store to location X and later executes a load from location Y.
- Processor B performs a store to location Y and later executes a load from location X.

The two processors are running asynchronously, and the order of the above two sequences is unknown.

For the system to be strongly ordered, either processor A must load the new value of Y, or processor B must load the new value of X, or both processors A and B must load the new values of Y and X, respectively, under all conditions.

If processors A and B both load old values of Y and X, respectively, under any conditions, the system is not strongly ordered.

New Value		**Strongly Ordered**
Processor A	**Processor B**	
No	No	No
Yes	No	Yes
No	Yes	Yes
Yes	Yes	Yes

1.6 R10000 Pipelines

This section describes the stages of the superscalar pipeline.

Instructions are processed in six partially-independent pipelines, as shown in Figure 1-4. The Fetch pipeline reads instructions from the instruction cache†, decodes them, renames their registers, and places them in three instruction queues. The instruction queues contain integer, address calculate, and floating-point instructions. From these queues, instructions are dynamically issued to the five pipelined execution units.

Stage 1

In stage 1, the processor fetches four instructions each cycle, independent of their alignment in the instruction cache — except that the processor cannot fetch across a 16-word cache block boundary. These words are then aligned in the 4-word *Instruction* register.

If any instructions were left from the previous decode cycle, they are merged with new words from the instruction cache to fill the *Instruction* register.

Stage 2

In stage 2, the four instructions in the *Instruction* register are decoded and renamed. (Renaming determines any dependencies between instructions and provides precise exception handling.) When renamed, the *logical* registers referenced in an instruction are mapped to *physical* registers. Integer and floating-point registers are renamed independently.

A logical register is mapped to a new physical register whenever that logical register is the destination of an instruction. Thus, when an instruction places a new value in a logical register, that logical register is renamed (mapped) to a new physical register, while its previous value is retained in the old physical register.

As each instruction is renamed, its logical register numbers are compared to determine if any dependencies exist between the four instructions decoded during this cycle. After the physical register numbers become known, the Physical Register Busy table indicates whether or not each operand is valid. The renamed instructions are loaded into integer or floating-point instruction queues.

Only one branch instruction can be executed during stage 2. If the instruction register contains a second branch instruction, this branch is not decoded until the next cycle.

The branch unit determines the next address for the Program Counter; if a branch is taken and then reversed, the branch resume cache provides the instructions to be decoded during the next cycle.

---

† The processor checks only the instruction cache during an instruction fetch; it does not check the data cache.

Stage 3

In stage 3, decoded instructions are written into the queues. Stage 3 is also the start of each of the five execution pipelines.

Stages 4-6

In stages 4 through 6, instructions are executed in the various functional units. These units and their execution process are described below.

Floating-Point Multiplier (3-stage Pipeline)

Single- or double-precision multiply and conditional move operations are executed in this unit with a 2-cycle latency and a 1-cycle repeat rate. The multiplication is completed during the first two cycles; the third cycle is used to pack and transfer the result.

Floating-Point Divide and Square-Root Units

Single- or double-precision division and square-root operations can be executed in parallel by separate units. These units share their issue and completion logic with the floating-point multiplier.

Floating-Point Adder (3-stage Pipeline)

Single- or double-precision add, subtract, compare, or convert operations are executed with a 2-cycle latency and a 1-cycle repeat rate. Although a final result is not calculated until the third pipeline stage, internal bypass paths set a 2-cycle latency for dependent add or multiply instructions.

Integer ALU1 (1-stage Pipeline)

Integer add, subtract, shift, and logic operations are executed with a 1-cycle latency and a 1-cycle repeat rate. This ALU also verifies predictions made for branches that are conditional on integer register values.

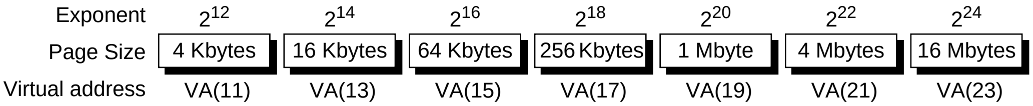
Integer ALU2 (1-stage Pipeline)

Integer add, subtract, and logic operations are executed with a 1-cycle latency and a 1-cycle repeat rate. Integer multiply and divide operations take more than one cycle.

Address Calculation and Translation in the TLB

A single memory address can be calculated every cycle for use by either an integer or floating-point load or store instruction. Address calculation and load operations can be calculated out of program order.

The calculated address is translated from a 44-bit virtual address into a 40-bit physical address using a translation-lookaside buffer. The TLB contains 64 entries, each of which can translate two pages. Each entry can select a page size ranging from 4 Kbytes to 16 Mbytes, inclusive, in powers of 4, as shown in Figure 1-6.



Exponent	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}
Page Size	4 Kbytes	16 Kbytes	64 Kbytes	256 Kbytes	1 Mbyte	4 Mbytes	16 Mbytes
Virtual address	VA(11)	VA(13)	VA(15)	VA(17)	VA(19)	VA(21)	VA(23)

Figure 1-6   TLB Page Sizes

Load instructions have a 2-cycle latency if the addressed data is already within the data cache.

Store instructions do not modify the data cache or memory until they graduate.

1.7 Implications of R10000 Microarchitecture on Software

The R10000 processor implements the MIPS architecture by using the following techniques to improve throughput:

- superscalar instruction issue
- speculative execution
- non-blocking caches

These microarchitectural techniques have special implications for compilation and code scheduling.

Superscalar Instruction Issue

The R10000 processor has parallel functional units, allowing up to four instructions to be fetched and up to five instructions to be issued or completed each cycle. An ideal code stream would match the fetch bandwidth of the processor with a mix of independent instructions to keep the functional units as busy as possible.

To create this ideal mix, every cycle the hardware would select one instruction from each of the columns below. (Floating-point divide, floating-point square root, integer multiply and integer divide cannot be started on each cycle.) The processor can look ahead in the code, so the mix should be kept close to the ideal described below.

Column A	Column B	Column C	Column D	Column E
FPadd	FP mul	FPload	add/sub	add/sub
	FPdiv	FPstore	shift	mul
	FPsqrt	load	branch	div
		store	logical	logical

Data dependencies are detected in hardware, but limit the degree of parallelism that can be achieved. Compilers can intermix instructions from independent code streams.

Speculative Execution

Speculative execution increases parallelism by fetching, issuing, and completing instructions even in the presence of unresolved conditional branches and possible exceptions. Following are some suggestions for increasing program efficiency:

- Compilers should reduce the number of branches as much as possible
- “Jump Register” instructions should be avoided.
- Aggressive use of the new integer and floating-point conditional move instructions is recommended.
- Branch prediction rates may be improved by organizing code so that each branch goes the same direction most of the time, since a branch that is taken 50% of the time has higher average cost than one taken 90% of the time. The MIPS IV conditional move instructions may be effective in improving performance by replacing unpredictable branches.

Side Effects of Speculative Execution

To improve performance, R10000 instructions can be speculatively fetched and executed. Side-effects are harmless in cached coherent operations; however there are potential side-effects with non-coherent cached operations. These side-effects are described in the sections that follow.

Speculatively fetched instructions and speculatively executed loads or stores to a cached address initiate a *Processor Block Read Request* to the external interface if it misses in the cache. The speculative operation may modify the cache state and/or data, and this modification may not be reversed even if the speculation turns out to be incorrect and the instruction is aborted.

Speculative Processor Block Read Request to an I/O Address

Accesses to I/O addresses often cause side-effects. Typically, such I/O addresses are mapped to an uncached region and uncached reads and writes are made as double/single/partial-word reads and writes (non-block reads and writes) in R10000. Uncached reads and writes are guaranteed to be non-speculative.

However, if R10000 has a “garbage” value in a register, a speculative block read request to an unpredictable physical address can occur, if it speculatively fetches data due to a Load or Jump Register instruction specifying this register. Therefore, speculative block accesses to load-sensitive I/O areas can present an unwanted side-effect.

Unexpected Write Back Due to Speculative Store Instruction

When a Store instruction is speculated and the target address of the speculative Store instruction is missing in the cache, the cache line is refilled and the state is marked to be *Dirty*. However the refilled data may not be actually changed in the cache if this *store* instruction is later aborted. This could present a side-effect in cases such as the one described below:

- The processor is storing data sequentially to memory area A, using a code-loop that includes Store and Cond.branch instructions.
- A DMA write operation is performed to memory area B.
- DMA area B is contiguous to the sequential storage area A.
- The DMA operation is noncoherent.
- The processor does not cache any lines of DMA area B.

If the processor and the DMA operations are performed in sequence, the following could occur:

1. Due to speculative execution at the exit of the code-loop, the line of data beyond the end of the memory area A — that is, the starting line of memory area B — is refilled to the cache. This cache line is then marked *Dirty*.
2. The DMA operation starts writing noncoherent data into memory area B.
3. A cache line replacement is caused by later activities of the processor, in which the cache line is written back to the top of area B. Thus, the first line of the DMA area B is overwritten by old cache data, resulting in incorrect DMA operation and data.

The OS can restrict the writable pages for each user process and so can prevent a user process from interfering with an active DMA space. The kernel, on the other hand, retains *xkphys* and *kseg0* addresses in registers. There is no write protection against the speculative use of the address values in these registers. User processes which have pages mapped to physical spaces not in RAM may also have side-effects. These side-effects can be avoided if DMA is coherent.

Speculative Instruction Fetch

The change in a cache line’s state due to a speculative instruction fetch is not reversed if the speculation is aborted. This does not cause any problems visible to the program except during a noncoherent memory operation. Then the following side-effect exists: if a noncoherent line is changed to *Clean Exclusive* and this line is also present in noncoherent space, the noncoherent data could be modified by an external component and the processor would then have stale data.

Workarounds for Noncoherent Cached Systems

The suggestions presented below are not exhaustive; the solutions and trade-offs are system dependent. Any one or more of the items listed below might be suitable in a particular system, and testing and simulations should be used to verify their efficacy.

1. The external agent can reject a *processor block read request* to any I/O location in which a speculative load would cause an undesired affect. Rejection is made by returning an external *NACK completion response*.
2. A *serializing* instruction such as a *cache barrier* or a *CP0 instruction* can be used to prevent speculation beyond the point where speculative stores are allowed to occur. This could be at the beginning of a *basic block* that includes instructions that can cause a store with an unsafe pointer. (*Stores* to addresses like *stack-relative, global-pointer-relative* and pointers to non-I/O memory might be safe.) Speculative loads can also cause a side-effect. To make sure there is no stale data in the cache as a result of undesired speculative loads, portions of the cache referred by the address of the DMA read buffers could be flushed after every DMA transfer from the I/O devices.
3. Make references to appropriate I/O spaces uncached by changing the cache coherency attribute in the TLB.
4. Generally, arbitrary accesses can be controlled by mapping selected addresses through the TLB. However, references to an unmapped cached *xkphys* region could have hazardous affects on I/O. A solution for this is given below:

   First of all, note that the *xkphys* region is hard-wired into cached and uncached regions, however the cache attributes for the *kseg0* region are programmed through the *Config* register. Therefore, clear the *KX* bit (to a zero) and set (to ones) the *SX* and *UX* bits in the *Status* register. This disables access to the *xkphys* region and restricts access to only the User and Supervisor portions of the 64-bit address space.

   In general, the system needs either a coherent or a noncoherent protocol — but not both. Therefore these cache attributes can be used by the external hardware to filter accesses to certain parts of the *kseg0* region. For instance, the cache attributes for the *kseg0* address space might be defined in the *Config* register to be *cache coherent* while the cache attributes in the TLB for the rest of virtual space are defined to be *cached-noncoherent* or *uncached*. The external hardware could be designed to reject all *cache coherent* mode references to the memory except to that prior-defined *safe* space in *kseg0* within which there is no possibility of an I/O DMA transfer. Then before the DMA read process and before the cache is flushed for the DMA read buffers, the cache attributes in the TLB for the I/O buffer address space are changed from *noncoherent* to *uncached*. After the DMA read, the access modes are returned to the *cached-noncoherent* mode.
5. Just before load/store instruction, use a *conditional move* instruction which tests for the reverse condition in the speculated branch, and make all aborted branch assignments *safe*. An example is given below:

```
        bne     r1,     r0, label
        -----
        -----
        -----
        -----
        movn    ra,     r0, r1    # test to see if r1 != 0; if r1 != 0 then branch
                                  # is mispredicted; move safe address (r0)
                                  # into ra

        ld      r4,     0 (ra)    # Without the previous movn, this lld
                                  # could create damaging read.
        -----
        -----
label:  -----
        -----
        -----
```

In the above example, without the MOVN the read to the address in register *ra* could be speculatively executed and later aborted. It is possible that this load could be premature and thus damaging. The MOVN guarantees that if there is a misprediction (*r1* is not equal to 0) *ra* will be loaded with an address to which a read will not be damaging.

6. The following is similar to the conditional-move example given above, in that it protects speculation only for a single branch, but in some instances it may be more efficient than either the conditional move or the cache barrier workarounds.

   This workaround uses the fact that branch-likely instructions are always predicted as taken by the R10000. Thus, any incorrect speculation by the R10000 on a branch-likely always occurs on a taken path. Sample code is:

```
        beql    rx, r1, label
        nop
        sw      r2, 0x0(r1)
label:  -----
        -----
```

The store to *r1* will never be to an address referred to by the content of *rx*, because the store will never be executed speculatively. Thus, the address referred to by the content of *rx* is protected from any spurious write-backs.

This workaround is most useful when the branch is often taken, or when there are few instructions in the protected block that are not memory operations. Note that no instructions in a block following a branch-likely will be initiated by speculation on that branch; however, in the case of a *serial instruction* workaround, only memory operations are prevented from speculative initiation. In the case of the *conditional-move* workaround, speculative initiation of all instructions continues unimpeded. Also, similar to the *conditional-move* workaround, this workaround only protects fall-through blocks from speculation on the immediately preceding branch. Other mechanisms must be used to ensure that no other branches speculate into the protected block. However, if a block that *dominates*[†] the fall-through block can be shown to be protected, this may be sufficient. Thus, if block (a) dominates block (b), and block (b) is the fall-through block shown above, and block (a) is the immediately previous block in the program (i.e., only the single conditional branch that is being replaced intervenes between (a) and (b)), then ensuring that (a) is protected by *serial instruction* means a branch-likely can safely be used as protection for (b).

Nonblocking Caches

As processor speed increases, the processor's data latency and bandwidth requirements rise more rapidly than the latency and bandwidth of cost-effective main memory systems. The memory hierarchy of the R10000 processor tries to minimize this effect by using large set-associative caches and higher bandwidth cache refills to reduce the cost of loads, stores, and instruction fetches. Unlike the R4400, the R10000 processor does not stall on data cache misses, instead defers execution of any dependent instructions until the data has been returned and continues to execute independent instructions (including other memory operations that may miss in the cache). Although the R10000 allows a number of outstanding primary and secondary cache misses, compilers should organize code and data to reduce cache misses. When cache misses are inevitable, the data reference should be scheduled as early as possible so that the data can be fetched in parallel with other unrelated operations.

As a further antidote to cache miss stalls, the R10000 processor supports prefetch instructions, which serve as hints to the processor to move data from memory into the secondary and primary caches when possible. Because prefetches do not cause dependency stalls or memory management exceptions, they can be scheduled as soon as the data address can be computed, without affecting exception semantics. Indiscriminate use of prefetch instructions can slow program execution because of the instruction-issue overhead, but selective use of prefetches based on compiler miss prediction can yield significant performance improvement for dense matrix computations.

[†] In compiler parlance, block (a) *dominates* block (b) if and only if every time block (b) is executed, block (a) is executed first. Note that block (a) does not have to immediately precede block (b) in execution order; some other block may intervene.

1.8 Performance

As it executes programs, the R10000 superscalar processor performs many operations in parallel. Instructions can also be executed out of order. Together, these two facts greatly improve performance, but they also make it difficult to predict the time required to execute any section of a program, since it often depends on the instruction mix and the critical dependencies between instructions.

The processor has five largely independent execution units, each of which are individualized for a specific class of instructions. Any one of these units may limit processor performance, even as the other units sit idle. If this occurs, instructions which use the idle units can be added to the program without adding any appreciable delay.

User Instruction Latency and Repeat Rate

Table 1-1 shows the latencies and repeat rates for all user instructions executed in ALU1, ALU2, Load/Store, Floating-Point Add and Floating-Point Multiply functional units (definitions of *latency* and *repeat rate* are given in the Glossary). Kernel instructions are not included, nor are control instructions not issued to these execution units.

Table 1-1   Latencies and Repeat Rates for User Instructions

Instruction Type	**Execution Unit**	**Latency**	**Repeat Rate**	**Comment**
Integer Instructions				
Add/Sub/Logical/Set	ALU 1/2	1	1	
MF/MT HI/LO	ALU 1/2	1	1	
Shift/LUI	ALU 1	1	1	
Cond. Branch Evaluation	ALU 1	1	1	
Cond. Move	ALU 1	1	1	
MULT	ALU 2	5/6	6	Latency relative to Lo/Hi
MULTU	ALU 2	6/7	7	Latency relative to Lo/Hi
DMULT	ALU 2	9/10	10	Latency relative to Lo/Hi
DMULTU	ALU 2	10/11	11	Latency relative to Lo/Hi
DIV/DIVU	ALU 2	34/35	35	Latency relative to Lo/Hi
DDIV/DDIVU	ALU 2	66/67	67	Latency relative to Lo/Hi
Load (not include loads to CP1)	Load/Store	2	1	Assuming cache hit
Store	Load/Store	-	1	Assuming cache hit
Floating-Point Instructions				
MTC1/DMTC1	ALU 1	3	1	
Add/Sub/Abs/Neg/Round/Trunc/Ceil/Floor/C.cond	FADD	2	1	
CVT.S.W/CVT.S.L	FADD	4	2	Repeat rate is on average
CVT (others)	FADD	2	1	
Mul	FMPY	2	1	
MFC1/DMFC1	FMPY	2	1	
Cond. Move/Move	FMPY	2	1	
DIV.S/RECIP.S	FMPY	12	14	
DIV.D/RECIP.D	FMPY	19	21	
SQRT.S	FMPY	18	20	
SQRT.D	FMPY	33	35	
RSQRT.S	FMPY	30	20	
RSQRT.D	FMPY	52	35	
MADD	FADD+FMPY	2/4	1	Latency is 2 only if the result is used as the operand specified by *fr* of another MADD
LWC1/LDC1/LWXC1/LDXC1	LoadStore	3	1	Assuming cache hit

Please note the following about Table 1-1:

- For integer instructions, conditional trap evaluation takes a single cycle, like conditional branches.
- Branches and conditional moves are not conditionally issued.
- The repeat rate above for Load/Store does not include Load Link and Store Conditional.
- Prefetch instruction is not included here.
- The latency for multiplication and division depends upon the next instruction.
- An instruction using register *Lo* can be issued one cycle earlier than one using *Hi*.
- For floating-point instructions, CP1 branches are evaluated in the Graduation Unit.
- CTC1 and CFC1 are not included in this table.
- The repeat pattern for the CVT.S.(W/L) is “I I x x I I x x ...”; the repeat rate given here, 2, is the average.
- The latency for MADD instructions is 2 cycles if the result is used as the operand specified by *fr* of the second MADD instruction.
- Load Linked and Store Conditional instructions (LL, LLD, SC, and SCD) do not implicitly perform SYNC operations in the R10000. Any of the following events that occur between a Load Linked and a Store Conditional will cause the Store Conditional to fail: an exception; execution of an ERET, a load, a store, a SYNC, a CacheOp, a prefetch, or an external intervention/invalidation on the block containing the linked address. Instruction cache misses do not cause the Store Conditional to fail.
- Up to four branches can be evaluated at one cycle.[†]

For more information about implementations of the LL, SC, and SYNC instructions, please see the section titled, R10000-Specific CPU Instructions, in this chapter.

[†] Only one branch can be decoded at any particular cycle. Since each conditional branch is predicted, the real direction of each branch must be “evaluated.” For example,

```
beq  r2,r3,L1
nop
```

A comparison of r2 and r3 is made to determine whether the branch is taken or not. If the branch prediction is correct, the branch instruction is graduated. Otherwise, the processor must back out of the instruction stream decoded after this branch, and inform the IFetch to fetch the correct instructions. The evaluation is made in the ALU for integer branches and in the Graduation Unit for floating-point branches. A single integer branch can be evaluated during any cycle, but there may be up to 4 condition codes waiting to be evaluated for floating-point branches. Once the condition code is evaluated, all dependant FP branches can be evaluated during the same cycle.

Other Performance Issues

Table 1-1 shows execution times within the functional units only. Performance may also be affected by instruction fetch times, and especially by the execution of conditional branches.

In an effort to keep the execution units busy, the processor predicts branches and speculatively executes instructions along the predicted path. When the branch is predicted correctly, this significantly improves performance: for typical programs, branch prediction is 85% to 90% correct. When a branch is mispredicted, the processor must discard instructions which were speculatively fetched and executed. Usually, this effort uses resources which otherwise would have been idle, however in some cases speculative instructions can delay previous instructions.

Cache Performance

The execution of load and store instructions can greatly affect performance. These instructions are executed quickly if the required memory block is contained in the primary data cache, otherwise there are significant delays for accessing the secondary cache or main memory. Out-of-order execution and non-blocking caches reduce the performance loss due to these delays, however.

The latency and repeat rates for accessing the secondary cache are summarized in Table 1-2. These rates depend on the ratio of the secondary cache's clock to the processor's internal pipeline clock. The best performance is achieved when the clock rates are equal; slower external clocks add to latency and repeat times.

The primary data cache contains 8-word blocks, which are refilled using 2-cycle transfers from the quadword-wide secondary cache. Latency runs to the time in which the processor can use the addressed data.

The primary instruction cache contains 16-word blocks, which are refilled using 4-cycle transfers.

Table 1-2 Latency and Repeat Rates for Secondary Cache Reads

SCClkDiv Mode	**Latency[‡] (PClk Cycles)**	**Repeat Rate[*] (PClk Cycles)**
1	6	2 (data cache) 4 (instruction cache)
1.5	8-10[†]	3 (data cache) 6 (instruction cache)
2	9-12[†]	4 (data cache) 8 (instruction cache)

[‡] Assumes the cache way was correctly predicted, and there are no conflicting requests.

[*] Repeat rate = **PClk** cycles needed to transfer 2 quadwords (data cache) or 4 quadwords (instruction cache). Rate is valid for bursts of 2 to 3 cache misses; if more than three cache misses in a row, there can be a 1-cycle "bubble."

[†] Clock synchronization causes variability.

The processor mitigates access delays to the secondary cache in the following ways:

- The processor can execute up to 16 load and store instructions speculatively and out-of-order, using non-blocking primary and secondary caches. That is, it looks ahead in its instruction stream to find load and store instructions which can be executed early; if the addressed data blocks are not in the primary cache, the processor initiates cache refills as soon as possible.
- If a speculatively executed load initiates a cache refill, the refill is completed even if the load instruction is aborted. It is likely the data will be referenced again.
- The data cache is interleaved between two banks, each of which contains independent tag and data arrays. These four sections can be allocated separately to achieve high utilization. Five separate circuits compete for cache bandwidth (address calculate, tag check, load unit, store unit, external interface.)
- The external interface gives priority to its refill and interrogate operations. The processor can execute tag checks, data reads for load instructions, or data writes for store instructions. When the primary cache is refilled, any required data can be streamed directly to waiting load instructions.
- The external interface can handle up to four non-blocking memory accesses to secondary cache and main memory.

Main memory typically has much longer latencies and lower bandwidth than the secondary cache, which make it difficult for the processor to mitigate their effect. Since main memory accesses are non-blocking, delays can be reduced by overlapping the latency of several operations. However, although the first part of the latency may be concealed, the processor cannot look far enough ahead to hide the entire latency.

Programmers may use pre-fetch instructions to load data into the caches before it is needed, greatly reducing main memory delays for programs which access memory in a predictable sequence.

2. System Configurations

The R10000 processor provides the capability for a wide range of computer systems; this chapter describes some of the uni- and multiprocessor alternatives.

2.1 Uniprocessor Systems

In a typical uniprocessor system, the System interface of the R10000 processor connects in a point-to-point fashion with an external agent. Such a system is shown in Figure 2-1. The external agent is typically an ASIC that provides a gateway to the memory and I/O subsystems; in fact, this ASIC may incorporate the memory controller itself.

If hardware I/O coherency is desired, the external agent may use the multiprocessor primitives provided by the processor to maintain cache coherency for interventions and invalidations. External duplicate tags can be used by the external agent to filter external coherency requests.

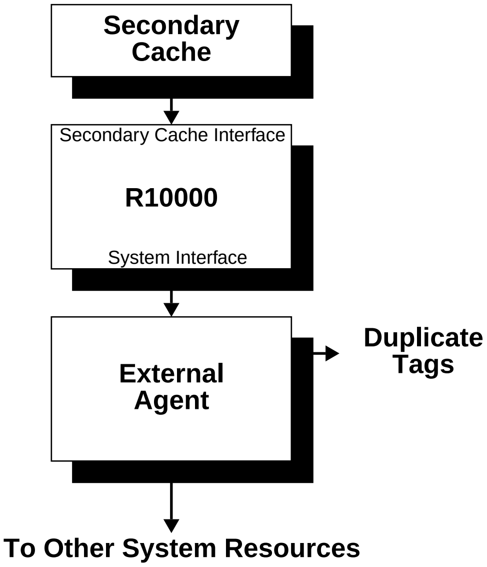

Secondary Cache
Secondary Cache Interface
R10000
System Interface
External Agent
Duplicate Tags
To Other System Resources

Figure 2-1 Uniprocessor System Organization

2.2 Multiprocessor Systems

Two types of multiprocessor systems can be implemented with R10000 processor:

- a dedicated external agent interfaces with each R10000 processor
- up to four R10000 processors and an external agent reside on a cluster bus

Multiprocessor Systems Using Dedicated External Agents

A multiprocessor system may be created with R10000 processors by providing a dedicated external agent for each processor; such a system is shown in Figure 2-2. The external agent provides a path between the processor System interface and some type of coherent interconnect. In such a system, the processor provides support for three coherency schemes:

- snoopy-based
- snoopy-based with external duplicate tags and control
- directory-based with external directory structure and control

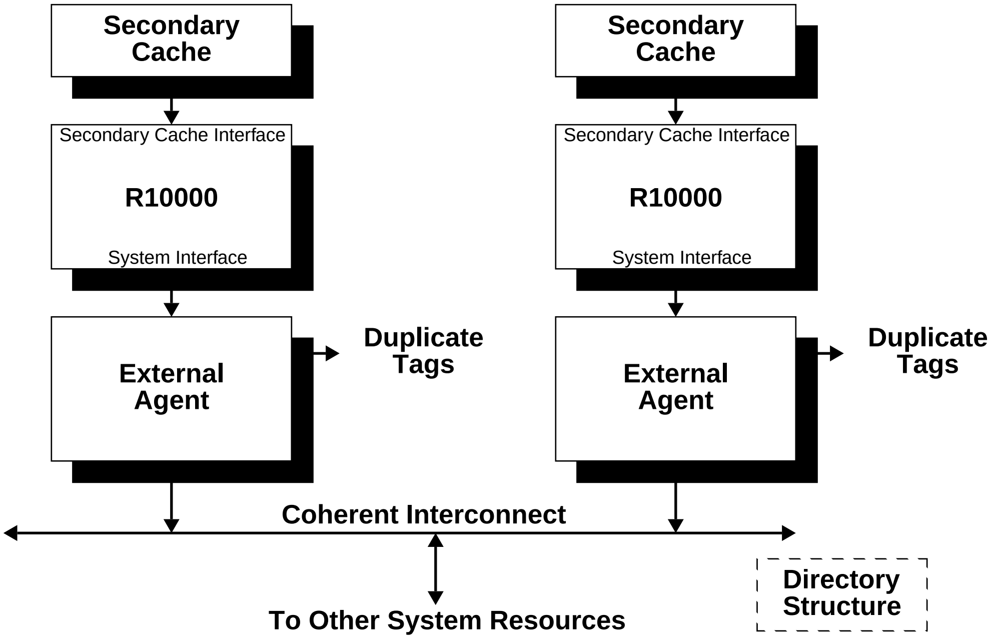

Secondary Cache
Secondary Cache Interface
R10000
System Interface
External Agent
Duplicate Tags
Secondary Cache
Secondary Cache Interface
R10000
System Interface
External Agent
Duplicate Tags
Coherent Interconnect
To Other System Resources
Directory Structure

Figure 2-2 Multiprocessor System Organization using Dedicated External Agents

Multiprocessor Systems Using a Cluster Bus

A multiprocessor system may be created with R10000 processors by using a cluster bus configuration. Such a system is shown in Figure 2-3. A cluster bus is created by attaching the System interfaces of up to four R10000 processors with an external agent (the *cluster coordinator*). The cluster coordinator is responsible for managing the flow of data within the cluster.

This organization can reduce the number of ASICs and the pin count needed for a small multiprocessor systems.

The cluster bus protocol supports three coherency schemes:

- snoopy-based
- snoopy-based with external duplicate tags and control
- directory-based with external directory structure and control

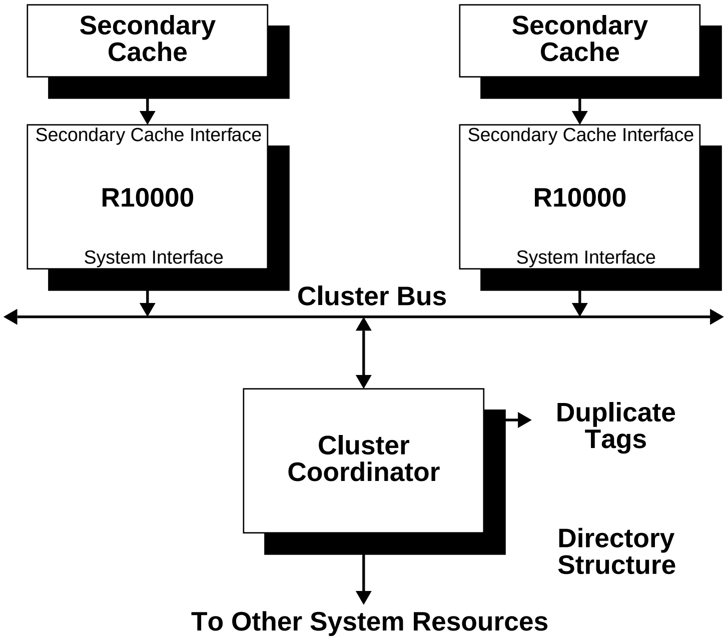

Secondary Cache
Secondary Cache
Secondary Cache Interface
R10000
System Interface
Secondary Cache Interface
R10000
System Interface
Cluster Bus
Cluster Coordinator
Duplicate Tags
Directory Structure
To Other System Resources

Figure 2-3 Multiprocessor System Organization Using the Cluster Bus

3. Interface Signal Descriptions

This chapter gives a list and description of the interface signals.

The R10000 interface signals may be divided into the following groups:

- Power interface
- Secondary Cache interface
- System interface
- Test interface

The following sections present a summary of the external interface signals for each of these groups. An asterisk (*****) indicates signals that are asserted as a logical 0.

3.1 Power Interface Signals

Table 3-1 presents the R10000 processor power interface signals.

Table 3-1 Power Interface Signals

Signal Name	Description	Type
Vcc	**Vcc** core **Vcc** for the core circuits.	Input
VccQSC	**Vcc** output driver secondary cache **Vcc** for the secondary cache interface output drivers.	Input
VccQSys	**Vcc** output driver system **Vcc** for the System interface output drivers.	Input
VrefSC	Voltage reference secondary cache Voltage reference for the secondary cache interface input receivers.	Input
VrefSys	Voltage reference system Voltage reference for the System interface input receivers.	Input
VrefByp	Voltage reference bypass This pin must be tied to **Vss** (preferably) or **VrefSys**, through at least a 100 ohm resistor.	Input
Vss	**Vss** **Vss** for the core circuits and output drivers.	Input
VccPa	**Vcc** PLL analog **Vcc** for the PLL analog circuits.	Input
VssPa	**Vss** PLL analog **Vss** for the PLL analog circuits.	Input
VccPd	**Vcc** PLL digital **Vcc** for the PLL digital circuits.	Input
VssPd	**Vss** PLL digital **Vss** for the PLL digital circuits.	Input
DCOk	DC voltages are OK The external agent asserts these two signals when **Vcc**, **VccQ[SC,Sys]**, **Vref[SC,Sys]**, **Vcc[Pa,Pd]**, and **SysClk** are stable.	Input

3.2 Secondary Cache Interface Signals

Table 3-2 presents the R10000 processor secondary cache interface signals.

Table 3-2 Secondary Cache Interface Signals

Signal Name	Description	Type
SSRAM[‡] Clock Signals		
SCClk(5:0) SCClk*(5:0)	Secondary cache clock Duplicated complementary secondary cache clock outputs.	Output
SSRAM Address Signals		
SCAAddr(18:0) SCBAddr(18:0)	Secondary cache address bus SCBAddr is complementary SCAAddr 19-bit bus, which specifies the set address of the secondary cache data and tag SSRAM that is to be accessed.	Output
SCTagLSBAddr	Secondary cache tag LSB address Signal that specifies the least significant bit of the address for the secondary cache tag SSRAM.	Output
SSRAM Data Signals		
SCADWay SCBDWay	Secondary cache data way Duplicated signal that indicates the way of the secondary cache data SSRAM that is to be accessed.	Output
SCData(127:0)	Secondary cache data bus 128-bit bus to read/write cache data from/to secondary cache data SSRAM.	Bidirectional
SCDataChk(9:0)	Secondary cache data check bus A 10-bit bus used to read/write ECC and even parity from/to the secondary cache data SSRAM.	Bidirectional
SCADOE* SCBDOE*	Secondary cache data output enable Duplicated signal that enables the outputs of the secondary cache data SSRAM.	Output
SCADWr* SCBDWr*	Secondary cache data write enable Duplicated signal that enables writing the secondary cache data SSRAM.	Output
SCADCS* SCBDCS*	Secondary cache data chip select Duplicated signal that enables the secondary cache data SSRAM.	Output

‡ All cache static RAM (SRAM) are synchronous SRAM (SSRAM).

Table 3-2 (cont.) Secondary Cache Interface Signals

Signal Name	Description	Type
	SSRAM Tag Signals	
SCTWay	Secondary cache tag way Signal indicating the way of the secondary cache tag SSRAM to be accessed.	Output
SCTag(25:0)	Secondary cache tag bus A 26-bit bus to read/write cache tags from/to the secondary cache tag SSRAM.	Bidirectional
SCTagChk(6:0)	Secondary cache tag check bus A 7-bit bus used to read/write ECC from/to the secondary cache tag SSRAM.	Bidirectional
SCTOE*	Secondary cache tag output enable A signal that enables the outputs of the secondary cache tag SSRAM.	Output
SCTWr*	Secondary cache tag write enable A signal that enables writing the secondary cache tag SSRAM.	Output
SCTCS*	Secondary cache tag chip select A signal which enables the secondary cache tag SSRAM.	Output

3.3 System Interface Signals

Table 3-3 presents the R10000 processor System interface signals.

Table 3-3 System Interface Signals

Signal Name	Description	Type
System Clock Signals		
SysClk SysClk*	System clock Complementary system clock input.	Input
SysClkRet SysClkRet*	System clock return Complementary system clock return output used for termination of the system clock.	Output
System Arbitration Signals		
SysReq*	System request The processor asserts this signal when it wants to perform a processor request and it is not already master of the System interface.	Output
SysGnt*	System grant The external agent asserts this signal to grant mastership of the System interface to the processor.	Input
SysRel*	System release The master of the System interface asserts this signal for one **SysClk** cycle to indicate that it will relinquish mastership of the System interface in the following **SysClk** cycle.	Bidirectional
System Flow Control Signals		
SysRdRdy*	System read ready The external agent asserts this signal to indicate that it can accept processor read and upgrade requests.	Input
SysWrRdy*	System write ready The external agent asserts this signal to indicate that it can accept processor write and eliminate requests.	Input
System Address/Data Bus Signals		
SysCmd(11:0)	System command A 12-bit bus for transferring commands between processor and the external agent.	Bidirectional
SysCmdPar	System command bus parity Odd parity for the system command bus.	Bidirectional
SysAD(63:0)	System address/data bus A 64-bit bus for transferring addresses and data between R10000 and the external agent.	Bidirectional

Table 3-3 (cont.) System Interface Signals

Signal Name	Description	Type
System State Bus Signals		
SysADChk(7:0)	System address/data check bus An 8-bit ECC bus for the system address/data bus.	Bidirectional
SysVal*	System valid The master of the System interface asserts this signal when it is driving valid information on the system command and system address/data buses.	Bidirectional
SysState(2:0)	System state bus A 3-bit bus used for issuing processor coherency state responses and also additional status indications.	Output
SysStatePar	System state bus parity Odd parity for the system state bus.	Output
SysStateVal*	System state bus valid The processor asserts this signal for one **SysClk** cycle when issuing a processor coherency state response on the system state bus.	Output
System Response Bus Signals		
SysResp(4:0)	System response bus A 5-bit bus used by the external agent for issuing external completion responses.	Input
SysRespPar	System response bus parity Odd parity for the system response bus.	Input
SysRespVal*	System response bus valid The external agent asserts this signal for one **SysClk** cycle when issuing an external completion response on the system response bus.	Input
System Miscellaneous Signals		
SysReset*	System reset The external agent asserts this signal to reset the processor.	Input
SysNMI*	System non-maskable interrupt The external agent asserts this signal to indicate a non-maskable interrupt.	Input
SysCorErr*	System correctable error The processor asserts this signal for one **SysClk** cycle when a correctable error is detected and corrected.	Output
SysUncErr*	System uncorrectable error The processor asserts this signal for one **SysClk** cycle when an uncorrectable tag error is detected.	Output
SysGblPerf*	System globally performed The external agent asserts this signal to indicate that all processor requests have been globally performed with respect to all external agents.	Input
SysCyc*	System cycle The external agent may use this signal to define a virtual System interface clock in a hardware emulation environment.	Input

3.4 Test Interface Signals

Table 3-4 presents the R10000 processor test interface signals.

Table 3-4   Test Interface Signals

Signal Name	Description	Type
JTAG Signals		
JTDI	JTAG serial data input Serial data input.	Input
JTDO	JTAG serial data output Serial data output.	Output
JTCK	JTAG clock Clock input.	Input
JTMS	JTAG mode select Mode select input.	Input
★ JTRST	JTAG reset input (active low) Asynchronous reset input (R12000A only)	Input
Miscellaneous Test Signals		
TCA	Testability control A (for manufacturing test only) This signal must be tied to **Vss**, through a 100 ohm resistor.	Input
TCB	Testability control B (for manufacturing test only) This signal must be tied to **Vss**, through a 100 ohm resistor.	Input
PLLDis	PLL disable (for manufacturing test only) This signal must be tied to **Vss** through a 100 ohm resistor.	Input
PLLRC	PLL Control Node (for manufacturing test only) There must be no connection made to this signal.	
PLLSpare(1:4)	These four pins must be tied to **Vss**.	
Spare(1,3)[†]	These two pins must be tied to **Vss**, through a 100 ohm resistor.	
★ TriState	3-state Control The system asserts this signal to 3-state all outputs and input/output pads except for **SCClk**, **SCClk***, and **JTDO**.	Input
SelDVCO	Select differential VCO (for manufacturing test only) This signal must be tied to Vcc.	Input

[†] The **Spare (1, 3)** are used in R12000 for diagnostic purpose and thus for R12000 should not be connected to anything.

Unused Inputs

Several input pins are unused during normal system operation, and should be tied to **V**cc through resistors:

- **JTDI**
- **JTCK**
- **JTMS**
- ★ **JTRST (for R12000A only)**

Several input pins are unused during normal system operation, and should be tied to **Vss** through 100 ohm resistors:

- **TCA, TCB**
- **PLLDis**
- **Spare1, Spare3** (for R10000)

Several input pins are unused during normal system operation, and should be tied to **Vss**:

- **PLLSpare1, PLLSpare2, PLLSpare3, PLLSpare4**
- **SelDVCO**

★ The following input pins are unused during normal system operation, and should be left open:

- **Spare1, Spare3** (for R12000)

The following input pins may be unused in certain system configurations, and each of them should be tied to **VccQSys**, preferably, through a resistor of 100 ohms or greater value:

- **SysNMI***

The following input pins may be unused in certain system configurations, and each of them should be tied to **Vss**, preferably, through a resistor of 100 ohms or greater value:

- **SysRdRdy***
- **SysWrRdy***
- **SysGblPerf***
- **SysCyc***

The following input pins may be unused in certain system configurations, and each of them should be tied (preferably) to **Vss**, or **VccQSys**, through a resistor of 100 ohms or greater value:

- **SysADChk(7:0)**

4. Cache Organization and Coherency

The processor implements a two-level cache structure consisting of separate primary instruction and data caches and a joint secondary cache.

Each cache is two-way set associative and uses a **write back protocol**; that is, two cache blocks are assigned to each set (as shown in Figure 4-1), and a cache store writes data into the cache instead of writing it directly to memory. Some time later this data is independently written to memory.

A write-invalidate cache coherency protocol (described later in this chapter) is supported through a set of cache states and external coherency requests.

4.1 Primary Instruction Cache

The processor has an on-chip 32-Kbyte primary instruction cache (also referred to simply as the *instruction cache*), which is a subset of the secondary cache. Organization of the instruction cache is shown in Figure 4-1.

The instruction cache has a fixed block size of 16 words and is two-way set associative with a **least-recently-used** (LRU) replacement algorithm.[†]

The instruction cache is indexed with a virtual address and tagged with a physical address.

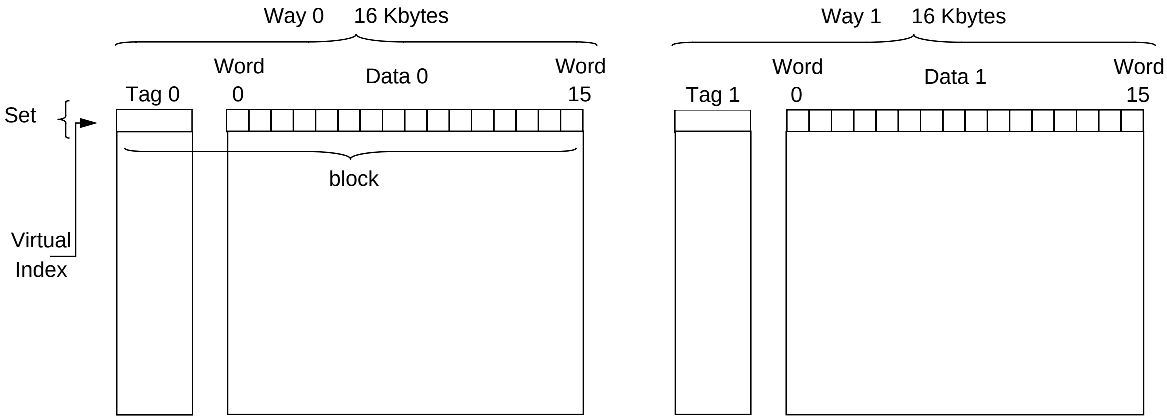

Way 0 16 Kbytes
Way 1 16 Kbytes
Word 0
Data 0
Word 15
Word 0
Data 1
Word 15
Tag 0
Tag 1
Set
block
Virtual Index

Figure 4-1 Organization of Primary Instruction Cache

Each instruction cache block is in one of the following two states:

- *Invalid*
- *Valid*

[†] The precise implementation of the LRU algorithm is affected by the speculative execution of instructions.

An instruction cache block can be changed from one state to the other as a result of any one of the following events:

- a primary instruction cache read miss
- subset property enforcement
- any of various CACHE instructions
- external intervention exclusive and invalidate requests

These events are illustrated in Figure 4-2, which shows the primary instruction cache state diagram.

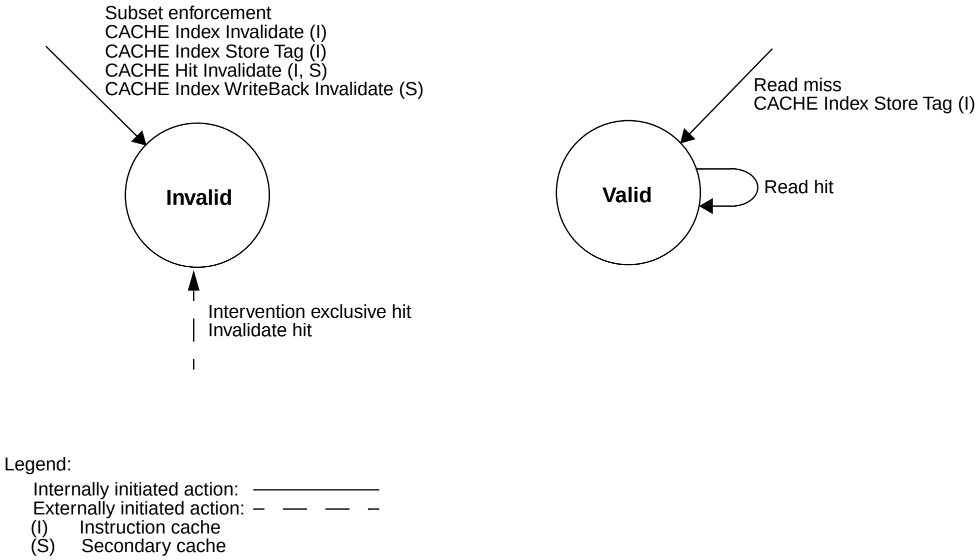

Subset enforcement
CACHE Index Invalidate (I)
CACHE Index Store Tag (I)
CACHE Hit Invalidate (I, S)
CACHE Index WriteBack Invalidate (S)
Read miss
CACHE Index Store Tag (I)
Invalid
Valid
Read hit
Intervention exclusive hit
Invalidate hit
Legend:
Internally initiated action:
Externally initiated action:
(I) Instruction cache
(S) Secondary cache

Figure 4-2 Primary Instruction Cache State Diagram

4.2 Primary Data Cache

The processor has an on-chip 32-Kbyte primary data cache (also referred to simply as the *data cache*), which is a subset of the secondary cache. The data cache uses a fixed block size of 8 words and is two-way set associative (that is, two cache blocks are assigned to each set, as shown in Figure 4-3) with an LRU replacement algorithm.[†]

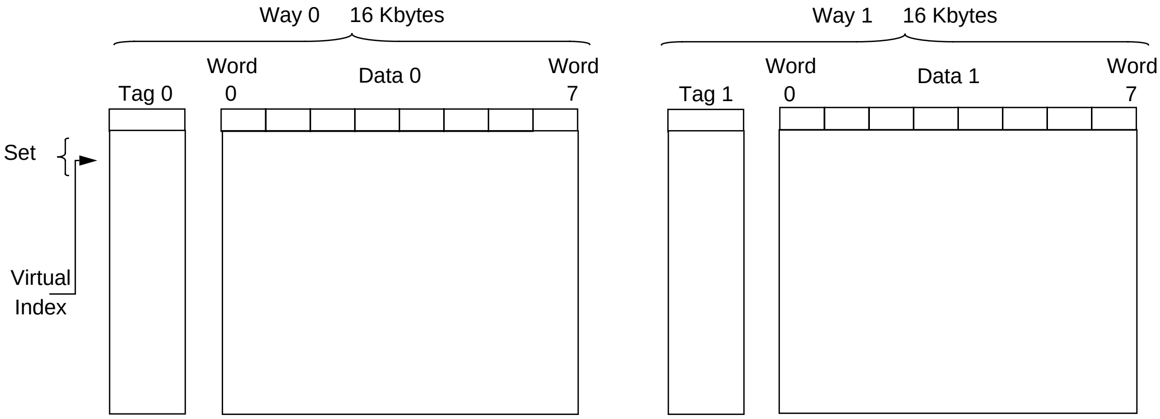

Way 0 16 Kbytes
Way 1 16 Kbytes
Tag 0
Word 0
Data 0
Word 7
Tag 1
Word 0
Data 1
Word 7
Set
Virtual Index

Figure 4-3 Organization of Primary Data Cache

The data cache uses a write back protocol, which means a cache store writes data into the cache instead of writing it directly to memory. Sometime later this data is independently written to memory, as shown in Figure 4-4.

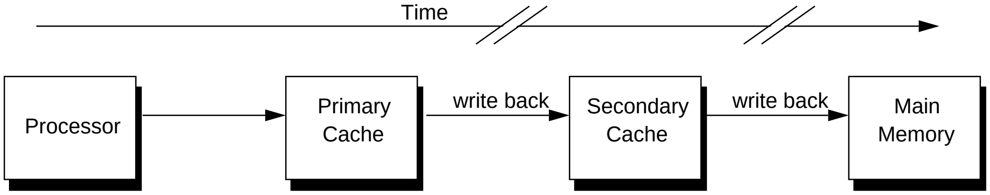

Time
Processor
Primary Cache
write back
Secondary Cache
write back
Main Memory

Figure 4-4 Write Back Protocol

Write back from the primary data cache goes to the secondary cache, and write back from the secondary cache goes to main memory, through the system interface. The primary data cache is written back to the secondary cache before the secondary cache is written back to the system interface.

[†] The precise implementation of the LRU algorithm is affected by the speculative execution of instructions.

The data cache is indexed with a virtual address and tagged with a physical address. Each primary cache block is in one of the following four states:

- *Invalid*
- *CleanExclusive*
- *DirtyExclusive*
- *Shared*

A primary data cache block is said to be *Inconsistent* when the data in the primary cache has been modified from the corresponding data in the secondary cache. The primary data cache is maintained as a subset of the secondary cache where the state of a block in the primary data cache always matches the state of the corresponding block in the secondary cache.

A data cache block can be changed from one state to another as a result of any one of the following events:

- primary data cache read/write miss
- primary data cache write hit
- subset enforcement
- a CACHE instruction
- external intervention shared request
- intervention exclusive request
- invalidate request

These events are illustrated in Figure 4-5, which shows the primary data cache state diagram.

DCache set locking relaxed (R12000)

In R10000, when an AQ entry accesses a DCache line, that line is locked into the cache until the entry graduates, so that the entry will not be removed from the cache until the access completes. If another entry which needs to access exactly the same line arrives in the AQ before the first completes, the two may share the lock. In this way, a line is locked in the cache until all access to it complete. In order to prevent a deadlock from arising, whenever a cache line is locked in this way, only the oldest AQ entry can obtain a lock on the other “way” of the same cache set, thus ensuring that forward progress can be made. This algorithm can cause problems, because often the oldest entry in the AQ is the one which already owns the lock on the first way - thus ensuring that no other entries can access the second way of the cache for that set index. For some algorithms, most notably FFT’s, this can cause severe performance degradation. R12000 allows an entry to obtain the lock on the second way of a set if it is the oldest entry which does not already own a lock. Thus, any entries which have already acquired a lock, including those locking the first way, will not prevent another, younger, entry from accessing that second way.

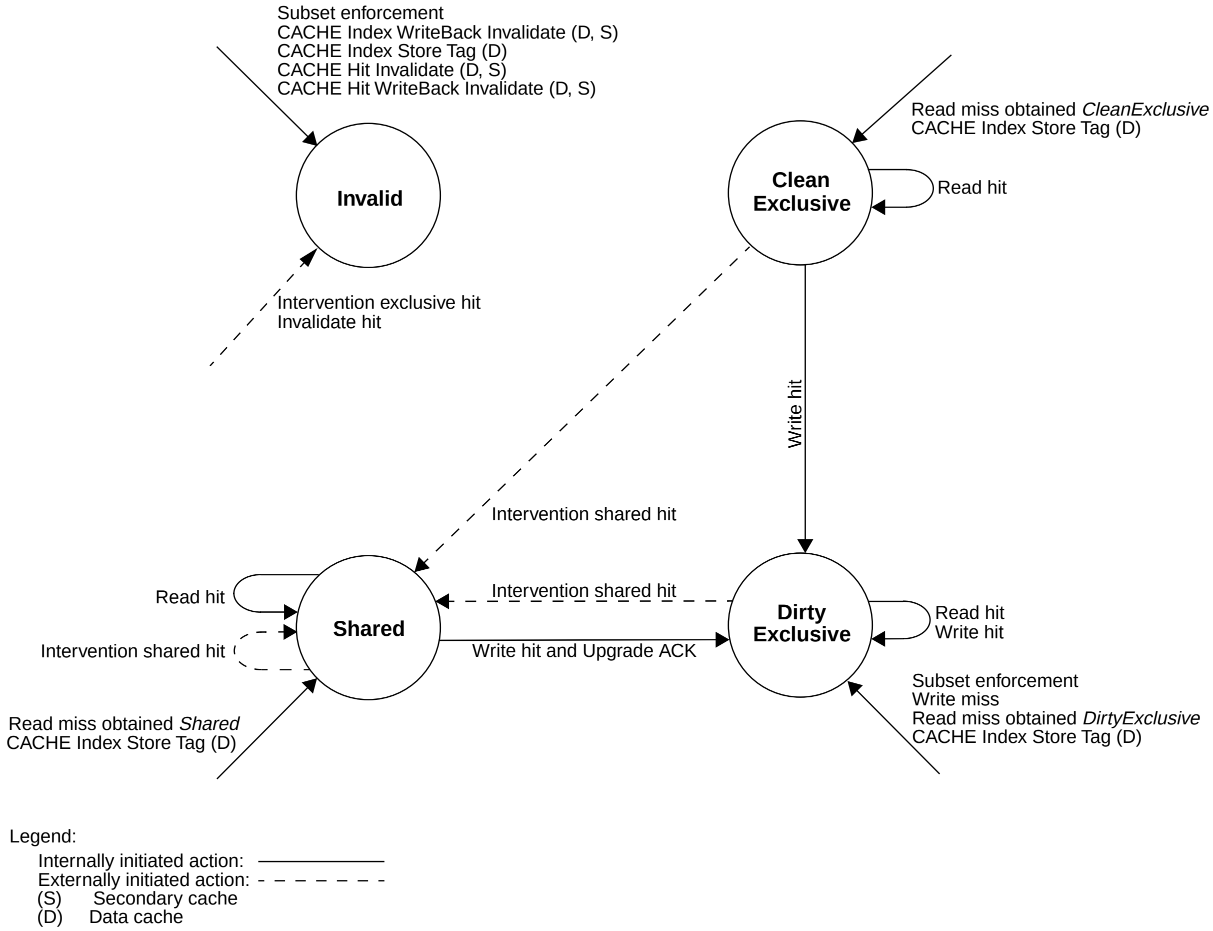

Subset enforcement
CACHE Index WriteBack Invalidate (D, S)
CACHE Index Store Tag (D)
CACHE Hit Invalidate (D, S)
CACHE Hit WriteBack Invalidate (D, S)
Invalid
Intervention exclusive hit
Invalidate hit
Read miss obtained *CleanExclusive*
CACHE Index Store Tag (D)
Clean Exclusive
Read hit
Write hit
Intervention shared hit
Read hit
Intervention shared hit
Shared
Intervention shared hit
Write hit and Upgrade ACK
Dirty Exclusive
Read hit
Write hit
Read miss obtained *Shared*
CACHE Index Store Tag (D)
Subset enforcement
Write miss
Read miss obtained *DirtyExclusive*
CACHE Index Store Tag (D)
Legend:
Internally initiated action:
Externally initiated action:
(S) Secondary cache
(D) Data cache

Figure 4-5 Primary Data Cache State Diagram

4.3 Secondary Cache

The R10000 processor must have an external secondary cache, ranging in size from 512 Kbytes to 16 Mbytes, in powers of 2, as set by the **SCSize** mode bit. The **SCBlkSize** mode bit selects a block size of either 16 or 32 words.

The secondary cache is two-way set associative (that is, two cache blocks are assigned to each set, as shown in Figure 4-6) with an LRU replacement algorithm.†

The secondary cache uses a write back protocol, which means a cache store writes data into the cache instead of writing it directly to memory. Some time later this data is independently written to memory.

The secondary cache is indexed with a physical address and tagged with a physical address.

★

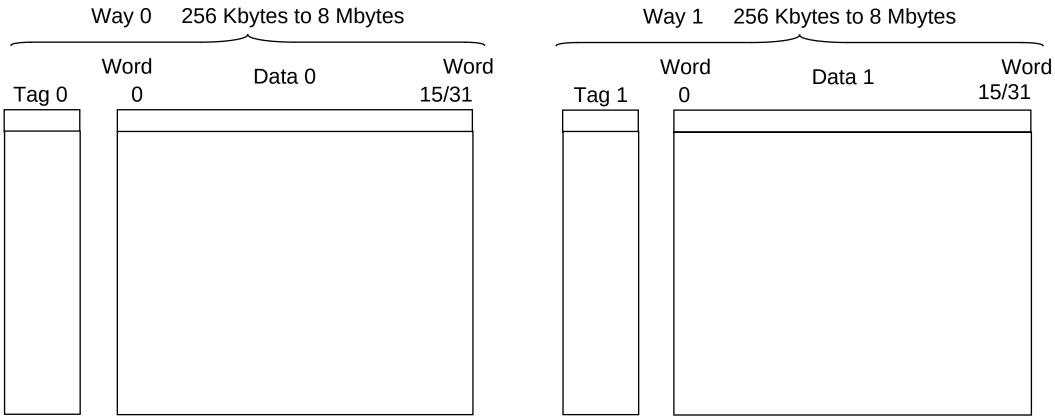


Figure 4-6 Organization of Secondary Cache

Each secondary cache block is in one of the following four states:

- *Invalid*
- *CleanExclusive*
- *DirtyExclusive*
- *Shared*

---

† The precise implementation of the LRU algorithm is affected by the speculative execution of instructions.

A secondary cache block can be changed from one state to another as a result of any of the following events:

- primary cache read/write miss
- primary cache write hit to a *Shared* or *CleanExclusive* block
- secondary cache read miss
- secondary cache write hit to a *Shared* or *CleanExclusive* block
- a CACHE instruction
- external intervention shared request
- intervention exclusive request
- invalidate request

These events are illustrated in Figure 4-7, which shows the secondary cache state diagram.

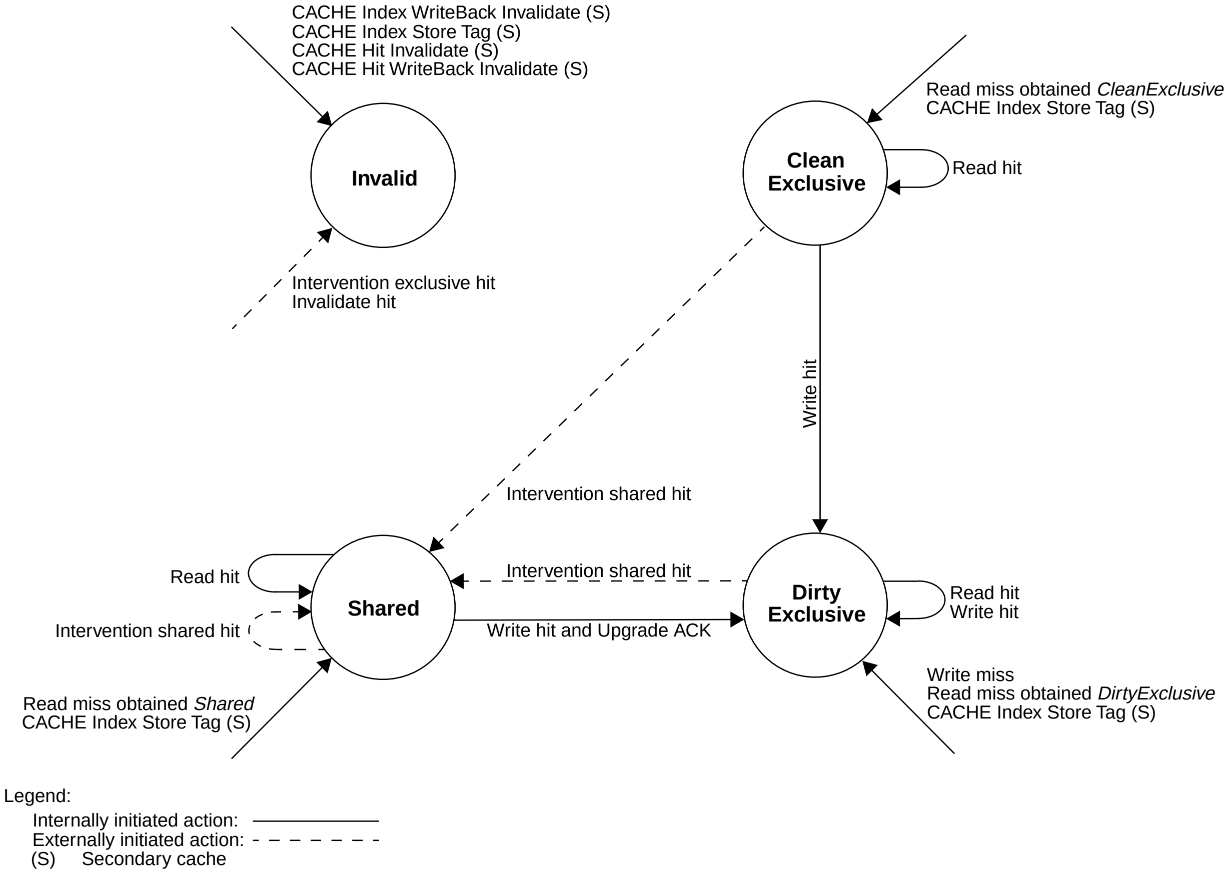

CACHE Index WriteBack Invalidate (S)
CACHE Index Store Tag (S)
CACHE Hit Invalidate (S)
CACHE Hit WriteBack Invalidate (S)
Invalid
Intervention exclusive hit
Invalidate hit
Read miss obtained *CleanExclusive*
CACHE Index Store Tag (S)
Clean Exclusive
Read hit
Write hit
Intervention shared hit
Read hit
Shared
Intervention shared hit
Read miss obtained *Shared*
CACHE Index Store Tag (S)
Intervention shared hit
Write hit and Upgrade ACK
Dirty Exclusive
Read hit
Write hit
Write miss
Read miss obtained *DirtyExclusive*
CACHE Index Store Tag (S)
Legend:
Internally initiated action:
Externally initiated action:
(S) Secondary cache

Figure 4-7 Secondary Cache State Diagram

<R12000>

Pad-ring clock slowed

The clock used to drive data to/from SC around the pad-ring has been slowed to a 2:3 clock divisor, thus sometimes adding an additional cycle of latency to secondary-cache accesses.

SC refill blocking reduced

In R10000, during the time that an SCache line is being refilled from system interface via the “incoming buffer (IB)”, no other accesses to the SCache are allowed. If the external interface sees an ACK to a line that is being refilled before the last words of the SCache line are received by R10000, this means that several cycles can elapse during which SCache access is blocked. By breaking the SCache refill transaction into 64-byte blocks, and allowing other requests to proceed during breaks between the blocks, this effect could be reduced. R12000 pulls in SCache lines with two “pause points.” This first occurs when R12000 receives the ACK for a request. If the first two quad-words are already valid in the Incoming Buffer at that time, then R12000 will proceed to refill the SCache with those two, and forward the results to the DCache or ICache at the same time as normal. The next two quad-words will be refilled as they return, thus continuing to block any other access to the SCache just as today. If however, when the initial ACK is received, the first two are not valid (i.e., either 0 or 1 quad-words are valid at that time) then R12000 will “pause” the SCache refill and wait for both of them to be brought in to the IB. Once the first half is filled in to the SCache, R12000 will again check the IB to see if an additional 3 quad-words are valid (thus 7 out of the 8 quad-words in the SCache line should have arrived into the IB).

Until that is the case, R12000 will again “pause” the SCache refill and allow other accesses to reach the SCache. These two pauses allow for other requests to slip in during an SCache refill. Using only two pauses both simplifies the logic and reduces bus turnarounds.

DCache writebacks never piggyback

In R10000 when a DCache line is written back to SCache, the following line in the DCache might be written back in a “piggybacked” manner. In order for this to occur the following line must have the same tag as the initially-written line, and must be in the “dirty inconsistent” state. This feature is being dropped form R12000.

DCache writebacks never bypass

In R10000 when a DCache line is written back to SCache, if the SCache interface is not otherwise occupied when the writeback begins, the writeback is bypassed directly to the SCache interface, avoiding the cycles required to write the data into the writeback buffer. This feature is being dropped form R12000.

4.4 Cache Algorithms

The behavior of the processor when executing load and store instructions is determined by the cache algorithm specified for the accessed address. The processor supports five different cache algorithms:

- uncached
- cacheable noncoherent
- cacheable coherent exclusive
- cacheable coherent exclusive on write
- uncached accelerated

Cache algorithms are specified in three separate places, depending upon the access:

- the cache algorithm for the mapped address space is specified on a per-page basis by the 3-bit cache algorithm field in the TLB
- the cache algorithm for the *kseg0* address space is specified by the 3-bit *K0* field of the CP0 *Config* register
- the cache algorithm for the *xkphys* address space is specified by **VA[61:59]**

Table 4-1 presents the encoding of the 3-bit cache algorithm field used in the TLB; *EntryLo0* and *EntryLo1* registers; CP0 *Config* register *K0* field for the *kseg0* address space; and **VA[61:59]** for the *xkphys* address space.

Table 4-1 Cache Algorithm Field Encodings

Value	Cache Algorithm
0	Reserved
1	Reserved
2	Uncached
3	Cacheable noncoherent
4	Cacheable coherent exclusive
5	Cacheable coherent exclusive on write
6	Reserved
7	Uncached accelerated

Descriptions of the Cache Algorithms

This section describes the cache algorithms listed in Table 4-1.

Uncached

Loads and stores under the *Uncached* cache algorithm bypass the primary and secondary caches. They are issued directly to the System interface using processor double/single/partial-word read or write requests.

Cacheable Noncoherent

Under the *Cacheable noncoherent* cache algorithm, load and store secondary cache misses result in processor noncoherent block read requests. External agents containing caches need not perform a coherency check for such processor requests.

Cacheable Coherent Exclusive

Under the *Cacheable coherent exclusive* cache algorithm, load and store secondary cache misses result in processor coherent block read exclusive requests. Such processor requests indicate to external agents containing caches that a coherency check must be performed and that the cache block must be returned in an *Exclusive* state.

Cacheable Coherent Exclusive on Write

The *Cacheable coherent exclusive on write* cache algorithm is similar to the *Cacheable coherent exclusive* cache algorithm except that load secondary cache misses result in processor coherent block read shared requests. Such processor requests indicate to external agents containing caches that a coherency check must be performed and that the cache block may be returned in either a *Shared* or *Exclusive* state.

Store hits to a *Shared* block result in a processor upgrade request. This indicates to external agents containing caches that the block must be invalidated.

Uncached Accelerated

The R10000 processor implements a new cache algorithm, *Uncached accelerated*. This allows the kernel to mark the TLB entries for certain regions of the physical address space, or certain blocks of data, as uncached while signalling to the hardware that data movement optimizations are permissible. This permits the hardware implementation to gather a number of uncached writes together, either a series of writes to the same address or sequential writes to all addresses in the block, into an uncached accelerated buffer and then issue them to the system interface as processor block write requests. The *uncached accelerated* algorithm differs from the *uncached* algorithm in that block write gathering is not performed.

There is no difference between an uncached accelerated load and an uncached load. Only word or doubleword stores can take advantage of this mode.

Stores under the *Uncached accelerated* cache algorithm bypass the primary and secondary caches. Stores to identical or sequential addresses are gathered in the uncached buffer, described in Chapter 6, the section titled “Uncached Buffer.”

Completely gathered uncached accelerated blocks are issued to the System interface as processor block write requests. Incompletely gathered uncached accelerated blocks are issued to the System interface using processor double/single-word write requests; this is also described in Chapter 6, the section titled “Uncached Buffer.”

4.5 Relationship Between Cached and Uncached Operations

Uncached and uncached accelerated load and store instructions are executed in order, and non-speculatively. Such accesses are buffered in the uncached buffer by the processor until they can be issued to the System interface.

All uncached and uncached accelerated accesses retain program order within the uncached buffer. The processor continues issuing cached accesses while uncached accesses are queued in the uncached buffer.

NOTE: Cached accesses do not probe the uncached buffer for conflicts.

Buffered uncached stores prevent a SYNC instruction from graduating. However buffered uncached accelerated stores do not prevent a SYNC instruction from graduating. The processor continues issuing cached accesses speculatively and out of order beyond a SYNC instruction that is waiting to graduate.

An uncached load may be used to guarantee that the uncached buffer is flushed of all uncached and uncached accelerated accesses.

A SYNC instruction and the **SysGblPerf*** signal may be used to guarantee that all cache accesses and uncached stores have been globally performed as described in Chapter 6, the section titled “SysGblPerf* Signal.”

An uncached load followed by a SYNC instruction may be used to guarantee that all cache accesses, uncached accesses, and uncached accelerated accesses have been globally performed.

4.6 Cache Algorithms and Processor Requests

The cache algorithm determines the type of processor request generated for secondary cache load misses, secondary cache store misses, and store hits.
Table 4-2 presents the relationship between the cache algorithm and processor requests.

Table 4-2 Cache Algorithms and Processor Requests

Cache Algorithm	**Load Miss**	**Store Miss**	**Store Hit**
Uncached	Double/single/partial-word read	Double/single/partial-word write	NA
Cacheable noncoherent	Noncoherent block read	Noncoherent block read	Upgrade if *Shared*[‡]
Cacheable coherent exclusive	Coherent block read exclusive	Coherent block read exclusive	Upgrade if *Shared*[‡]
Cacheable coherent exclusive on write	Coherent block read shared	Coherent block read exclusive	Upgrade if *Shared*
Uncached accelerated	Double/single/partial-word read	Gather identical or sequential double/single-word stores in the uncached buffer. Block write for completely gathered blocks. Double/single-word write for incompletely gathered blocks. Partial-word write for partial-word stores.	NA

[‡] Should not occur under normal circumstances. Most systems return the *Exclusive* state for a cacheable noncoherent line; therefore, the *Shared* state is not normal.

4.7 Cache Block Ownership

The processor requires cache blocks to have a single owner at all times. The owner is responsible for providing the current contents of the cache block to any requestor.

The processor uses the following ownership rules:

- The processor assumes ownership of a cache block if the state of the cache block becomes *DirtyExclusive*. For a processor block read request, the processor assumes ownership of the block after receiving the last doubleword of a *DirtyExclusive* external block data response and an external ACK completion response. For a processor upgrade request, the processor assumes ownership of the block after receiving an external ACK completion response.
- The processor gives up ownership of a cache block if the state of the cache block changes to *Invalid*, *CleanExclusive*, or *Shared*.
- *CleanExclusive* and *Shared* cache blocks are always considered to be owned by memory.

5. Secondary Cache Interface

The processor supports a mandatory secondary cache by providing an internal secondary cache controller with a dedicated secondary cache port.

The cache's tag and data arrays each consist of an external bank of industry-standard synchronous SRAM (SSRAM). This SSRAM must have registered inputs and outputs, asynchronous output enables, and use the late write protocol (data is expected one cycle after the address).

5.1 Tag and Data Arrays

The secondary cache consists of a 138-bit wide data array (128 data bits + 9 ECC bits + 1 parity bit) and a 33-bit wide tag array (26 tag bits + 7 ECC bits), as shown in Figure 5-1. ECC is supported for both the data and tag arrays to improve data integrity.

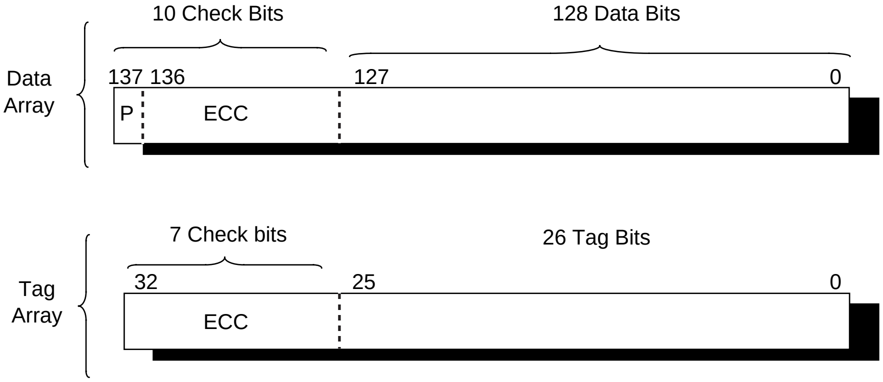


Figure 5-1 Secondary Cache Data and Tag Array

The secondary cache is implemented as a two-way set associative, combined instruction/ data cache, which is physically addressed and physically tagged, as described in Chapter 4, the section titled “Cache Organization and Coherency.”

The **SCSize** mode bits specify the secondary cache size; minimum secondary cache size is 512 Kbytes and the maximum secondary cache size is 16 Mbytes, in power of 2 (512 Kbytes, 1 Mbyte, 2 Mbytes, etc.).

The **SCBlkSize** mode bit specifies the secondary cache block size. When negated, the block size is 16 words, and when asserted, the block size is 32 words.

5.2 Secondary Cache Interface Frequencies

The secondary cache interface operates at the frequency of **SCClk**, which is derived from **PClk**. The **SCClkDiv** mode bits select a **PClk** to **SCClk** divisor of 1, 1.5, 2, 2.5, or 3, using the formula described in Chapter 7, the section titled “Secondary Cache Clock.”

Synchronization between the **PClk** and **SCClk** is performed internally and is invisible to the system. The processor supplies six complementary copies of the secondary cache clock on **SCClk(5:0)** and **SCClk(5:0)***.

The outputs and inputs at this interface are triggered by an internal **SCClk**. The relationship between the internal **SCClk** and the external **SCClk[5:0]/SCClk[5:0]*** can be programmed during boot time by setting the **SCClkTap** mode bits (see the section titled “Mode Bits” in Chapter 8 for detail on mode bits).

5.3 Secondary Cache Indexing

The secondary cache data array width is one quadword, and therefore **PA(3:0)**, which specify a byte within a quadword, are unused by the Secondary Cache interface.

Indexing the Data Array

Since the maximum secondary cache size is 16 Mbytes (8 Mbytes per way), each way requires a maximum of 23 bits to index a byte within a selected way, or 19 bits to index a quadword within a way. Consequently, the processor supplies **PA(22:4)** on **SC(A,B)Addr(18:0)** to index a quadword within a way. The processor selects a secondary cache data way with the **SC(A,B)DWay** signal.

Table 5-1 presents the secondary cache data array index for each secondary cache size; for instance, a 4 Mbyte cache uses the 17 address bits, **PA(20:4)** on **SC(A,B)Addr(16:0)**, concatenated with the way bit, **SC(A,B)DWay**, to index a quadword within a 2 Mbyte way.

Table 5-1 Secondary Cache Data Array Index

SCSize Mode Bits	Secondary Cache Size	Secondary Cache Data Array Index	Physical Address Bits Used
0	512 Kbyte	**SC(A,B)DWay || SC(A,B)Addr(13:0)**	**PA(17:4)**
1	1 Mbyte	**SC(A,B)DWay || SC(A,B)Addr(14:0)**	**PA(18:4)**
2	2 Mbyte	**SC(A,B)DWay || SC(A,B)Addr(15:0)**	**PA(19:4)**
3	4 Mbyte	**SC(A,B)DWay || SC(A,B)Addr(16:0)**	**PA(20:4)**
4	8 Mbyte	**SC(A,B)DWay || SC(A,B)Addr(17:0)**	**PA(21:4)**
5	16 Mbyte	**SC(A,B)DWay || SC(A,B)Addr(18:0)**	**PA(22:4)**

Indexing the Tag Array

The processor supplies the secondary cache tag array's least significant index bit on **SCTagLSBAddr** to support two block sizes without system hardware changes. This signal functions normally as a least significant index bit when the secondary cache block size is 16 words. However, when the secondary cache block size is 32 words, this signal is always negated, since only half as many tags are required. The processor supplies the secondary cache tag way on **SCTWay**.

Table 5-2 presents the secondary cache tag array index for each secondary cache size; it shows each index is composed of a physical address loaded onto **SC(A,B)Addr()**, concatenated with **SCTWay** and **SCTagLSBAddr**.

Table 5-2 Secondary Cache Tag Array Index

SCSize Mode Bits	Secondary Cache Size	Secondary Cache Tag Array Index
0	512 Kbyte	**SCTWay || SC(A,B)Addr(13:3) || SCTagLSBAddr**
1	1 Mbyte	**SCTWay || SC(A,B)Addr(14:3) || SCTagLSBAddr**
2	2 Mbyte	**SCTWay || SC(A,B)Addr(15:3) || SCTagLSBAddr**
3	4 Mbyte	**SCTWay || SC(A,B)Addr(16:3) || SCTagLSBAddr**
4	8 Mbyte	**SCTWay || SC(A,B)Addr(17:3) || SCTagLSBAddr**
5	16 Mbyte	**SCTWay || SC(A,B)Addr(18:3) || SCTagLSBAddr**

For a system design that only supports a secondary cache block size of 32 words, the secondary cache tag array need not use **SCTagLSBAddr** as an index bit.

5.4 Secondary Cache Way Prediction Table

The primary and secondary caches are two-way set associative. However, the implementation of the secondary cache is different than the primary caches.

The primary caches read simultaneously from two separate tag arrays, corresponding to each way in the cache, and then select the data based on the result of two parallel tag compares.

The secondary cache does not use this implementation because it would either require too many pins to read in two full copies of the data and tags, or add latency to externally multiplex two banks of memory. Instead, a way prediction table is used to determine which way to read from first.

The way prediction table is internal to the processor and has 8K one-bit entries, each entry corresponding to a pair of secondary cache blocks. The bit entry indicates which way of the addressed set has been most-recently used (MRU). When the secondary cache is accessed, this prediction bit is used as an address bit; thus the two ways in the secondary cache are shared in the same SSRAM bank.

The secondary cache way prediction table is indexed with a subset of 11 to 13 bits of the physical address, based on both the secondary cache block size, and the secondary cache size, as shown in Table 5-3. “0 |” indicates a zero bit concatenated to the address to pad the index out to a full 13-bits.

Table 5-3 Secondary Cache Way Prediction Table Index

SCSize Mode Bits	Secondary Cache Size	SCBlkSize Mode Bit	Secondary Cache Block Size	Secondary Cache Way Prediction Table Index
0	512 Kbyte	0	16-word	0 || PA(17:6)
		1	32-word	0 || 0 || PA(17:7)
1	1 Mbyte	0	16-word	PA(18:6)
		1	32-word	0 || PA(18:7)
2 to 5	2M to 16 Mbyte	0	16-word	PA(18:6)
		1	32-word	PA(19:7)

Three states are possible in the way prediction table:

- the desired data is in the predicted way
- the desired data is in the non-predicted way
- the desired data is not in the secondary cache

The tags for both ways are read “underneath” the data access cycles in order to discern as rapidly as possible which of these states are valid. This reading is possible because it takes two accesses to read a primary data block (8 words) and 4 cycles to read a primary instruction block (16 words); thus the bandwidth needed to read the tag array twice exists in all cases. Only an extra address pin to the tag array is needed to make this operation parallel and this is implemented by the **SCTWay** pin.

The three possible states are handled in the following manner:

- If, after reading the tags for both ways, it is discovered that the data exists in the predicted way, the processor continues normally.
- If the data exists in the non-predicted way, the processor accesses this non-predicted way in the secondary cache and updates the way prediction table to point to this way.
- If the access misses in both ways of the secondary cache, the data is fetched from the system interface. If the state of the predicted way is found to be *invalid*, the fetched data is placed in it and the MRU is unchanged. However, if the state of the predicted way is found to be *valid* then the fetched data is placed into the non-predicted way, and the way prediction table is updated to point to this way since it is now the most-recently-used.

The way prediction table can cover up to a 2 Mbyte secondary cache when the secondary cache block size is 32 words. If the secondary cache exceeds this size, the accuracy of the way prediction table diminishes slightly. However, the extremely large performance gain made by making the secondary cache larger far outstrips any performance loss in the way prediction table.

Increased the Way Prediction Table (MRU table) to 16K single-bit entries

The size of the table has been increased to 16K entries, so that 4MB caches with 128B lines or 2MB caches with 64B lines can be fully mapped.

Direct Cache Test Mode

Due to the increase size of the Way Prediction Table, Direct Cache Test Mode have been modified for testing the Way Prediction Table.

5.5 Secondary Cache Tag

The secondary cache tag, transferred on the **SCTag(25:0)** bus, is divided into three fields, as shown in Figure 5-2 below.

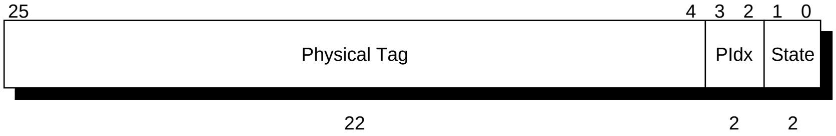


Figure 5-2 Secondary Cache Tag Fields

SCTag(25:4), Physical Tag

The minimum secondary cache size is 512 Kbytes (256 Kbytes per way), so a minimum of 18 bits are required to index a data byte within a selected way. Since the processor supports 40 physical bits, a maximum of 22 bits are required for the physical tag:

```
40 physical address bits - 18 minimum required = 22
```

Consequently, the processor supplies the 22 physical address bits, **PA(39:18)**, on **SCTag(25:4)** for the physical tag.

When the secondary cache is larger than the minimum size, the secondary cache tag array must still maintain the full physical tag supplied by the processor, even though some bits are redundant.

SCTag(3:2), PIdx

Bits **SCTag(3:2)** of the secondary cache tag contain the primary cache index, *PIdx*.

The *PIdx* field contains VA(13:12), which are the two lowest virtual address bits above the minimum 4 Kbyte page size. This field is written into the secondary cache tag during a secondary cache refill. For each processor-initiated secondary cache access, the virtual address bits are compared with the *PIdx* field of the secondary cache tag. If a mismatch occurs, a virtual coherency condition exists and the value of the *PIdx* field is used by internal control logic to purge primary cache locations, so that all primary cache blocks holding valid data have indices known to the secondary cache. This mechanism, unlike that of the R4400 processor, is implemented in hardware. It helps preserve the integrity of cached accesses to a physical address using different virtual addresses, an occurrence called **virtual aliasing**. For each external coherency request, the *PIdx* field of the secondary cache tag provides a mechanism to locate subset lines in the primary caches.

SCTag(1:0), Cache Block State

The lower two bits of the secondary cache tag, **SCTag(1:0)**, contain the cache block state, which can be *Invalid, Shared, CleanExclusive*, or *DirtyExclusive* as shown in Table 5-4.

Table 5-4 Secondary Cache Tag State Field Encoding

SCTag(1:0)	State
0	*Invalid*
1	*Shared*
2	*CleanExclusive*
3	*DirtyExclusive*

Since the secondary cache tags are updated immediately for stores to the primary data cache, and all caches use a write back protocol, the data in the secondary cache may not always be consistent with data in the primary cache even though the tags always reflect the correct state of a secondary cache block.

5.6 Read Sequences

There are five basic read sequences:

- a 4-word read
- an 8-word read
- a 16-word read
- a 32-word read
- a tag read

The **SCClk** referred in the secondary cache read and write timing diagrams is an internal **SCClk**. The relationship between this internal **SCClk** and the external **SCClk[5:0]/ SCClk[5:0]*** can be programmed during boot time by setting the **SCClkTap** mode bits (see the section titled “Mode Bits” in Chapter 8 for detail on mode bits).

4-Word Read Sequence

A 4-word read sequence is performed by a CACHE Index Load Data (S) instruction to read a doubleword of data and 10 check bits from the secondary cache data array.

Figure 5-3 depicts a secondary cache 4-word read sequence. A quadword is read from the index specified by **PA(23:6)**, and the way specified by **VA(0)** of the CACHE instruction.

The doubleword specified by **VA(3)** is then stored into the CP0 *TagHi* and *TagLo* registers, and the corresponding check bits are stored into the CP0 *ECC(9:0)* register. The data may be examined by copying the CP0 *TagHi*, *TagLo*, and *ECC* registers to the general registers with the MTC0 instruction.

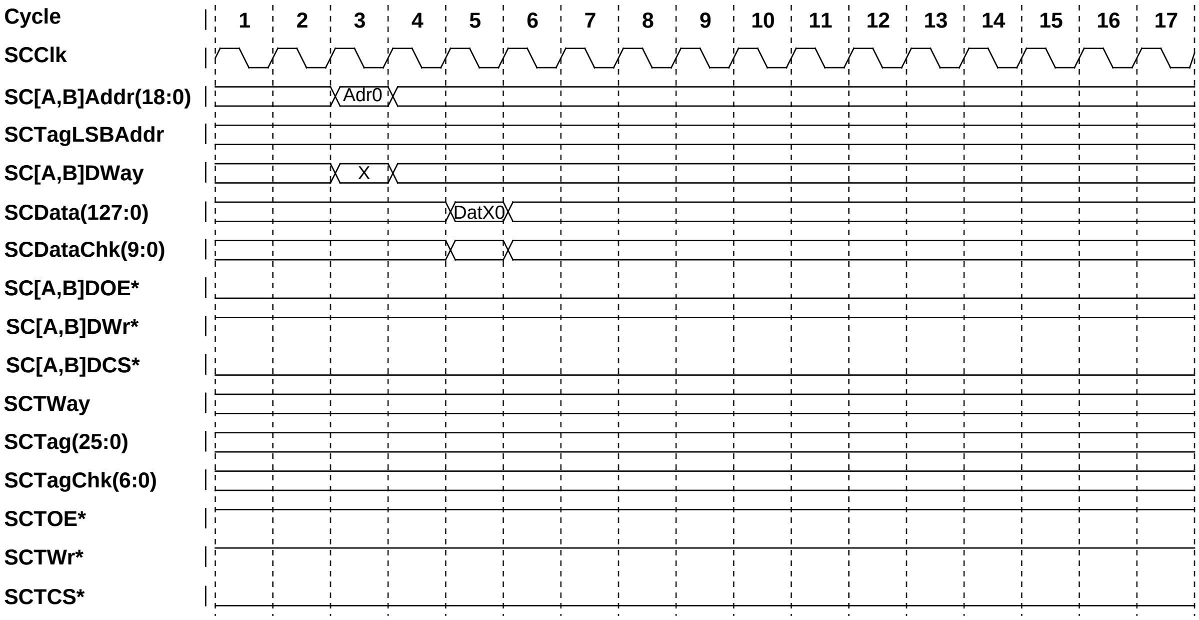

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SCClk
SC[A,B]Addr(18:0)
Adr0
SCTagLSBAddr
SC[A,B]DWay
X
SCData(127:0)
DatX0
SCDataChk(9:0)
SC[A,B]DOE*
SC[A,B]DWr*
SC[A,B]DCS*
SCTWay
SCTag(25:0)
SCTagChk(6:0)
SCTOE*
SCTWr*
SCTCS*

Figure 5-3   4-Word Read Sequence

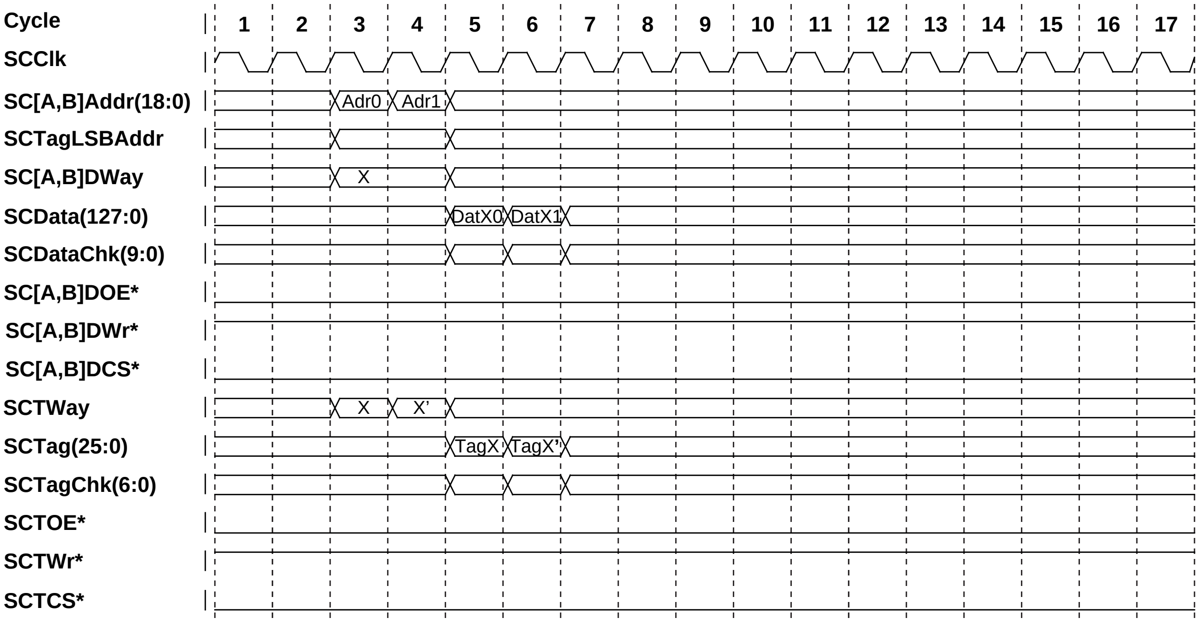
8-Word Read Sequence

An 8-word read sequence refills the primary data cache from the secondary cache after a primary data cache miss.

Figure 5-4 depicts a secondary cache 8-word read sequence. In it, **SC(A,B)DWay** and **SCTWay** are driven with value X on the first address cycle, which is obtained from the way prediction table.

On the next address cycle, **SCTWay** is complemented in order to read the tag from the non-predicted way of the addressed set. **SC(A,B)DWay** is not changed since it is assumed that the way prediction table is correct and the read is likely to hit in the predicted way.

The tag for the non-predicted way is returned to the processor in the same cycle as the second quadword of data. Reads that miss in the predicted way, but hit in the non-predicted way, are noted by the internal control logic and reissued to the secondary cache as soon as possible.



Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SCClk
SC[A,B]Addr(18:0)
Adr0
Adr1
SCTagLSBAddr
SC[A,B]DWay
X
SCData(127:0)
DatX0
DatX1
SCDataChk(9:0)
SC[A,B]DOE*
SC[A,B]DWr*
SC[A,B]DCS*
SCTWay
X
X'
SCTag(25:0)
TagX
TagX'
SCTagChk(6:0)
SCTOE*
SCTWr*
SCTCS*

Figure 5-4 8-Word Read Sequence

16 or 32-Word Read Sequence

A 16-word read sequence refills the primary instruction cache from the secondary cache after a primary instruction cache miss. A 16-word read sequence is also performed when the secondary cache block size is 16 words, and a *DirtyExclusive* secondary cache block must be written back to the System interface.

A 32-word read sequence is performed when the secondary cache block size is 32 words, and a *DirtyExclusive* secondary cache block must be written back to the System interface.

Figure 5-5 depicts a secondary cache 16 or 32-word read sequence. This is similar to an 8-word read sequence except that more addresses must be issued, in order to read the appropriate number of quadwords.

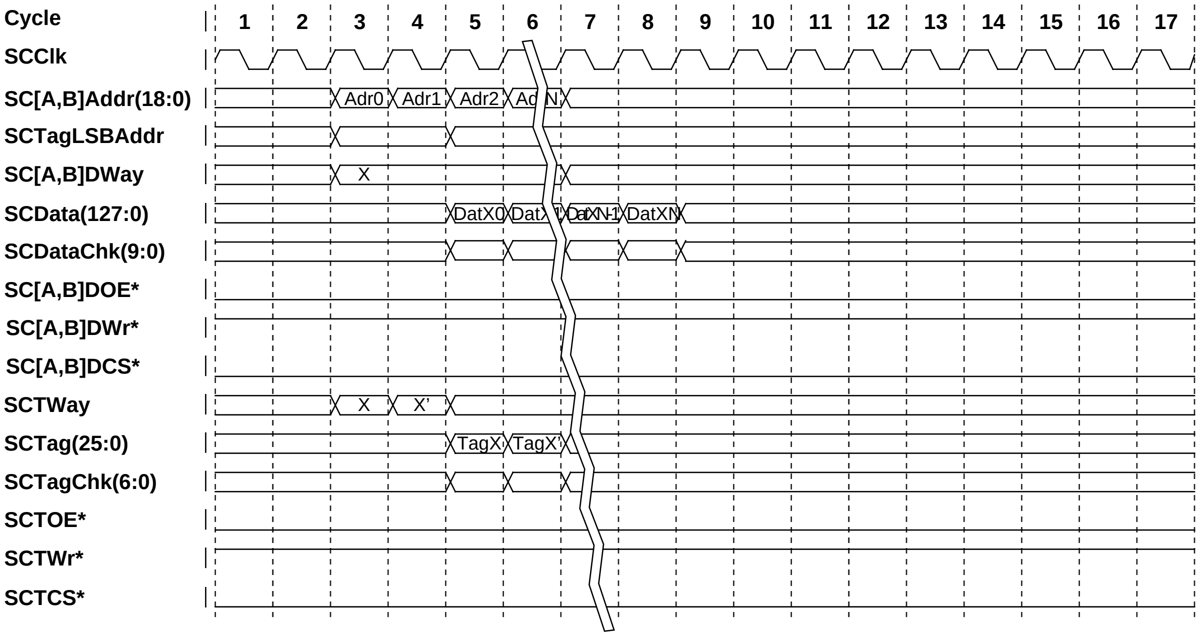

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SCClk
SC[A,B]Addr(18:0)
Adr0
Adr1
Adr2
Ad
N
SCTagLSBAddr
SC[A,B]DWay
X
SCData(127:0)
DatX0
DatX1
DatXN-1
DatXN
SCDataChk(9:0)
SC[A,B]DOE*
SC[A,B]DWr*
SC[A,B]DCS*
SCTWay
X
X'
SCTag(25:0)
TagX
TagX'
SCTagChk(6:0)
SCTOE*
SCTWr*
SCTCS*

Figure 5-5 16 or 32-Word Read Sequence

Tag Read Sequence

A tag read sequence is performed when the state of a secondary cache block is required, but it is not necessary to access the data array. This sequence is used for the CACHE Index Load Tag (S) instruction.

Figure 5-6 depicts a secondary cache tag read sequence.

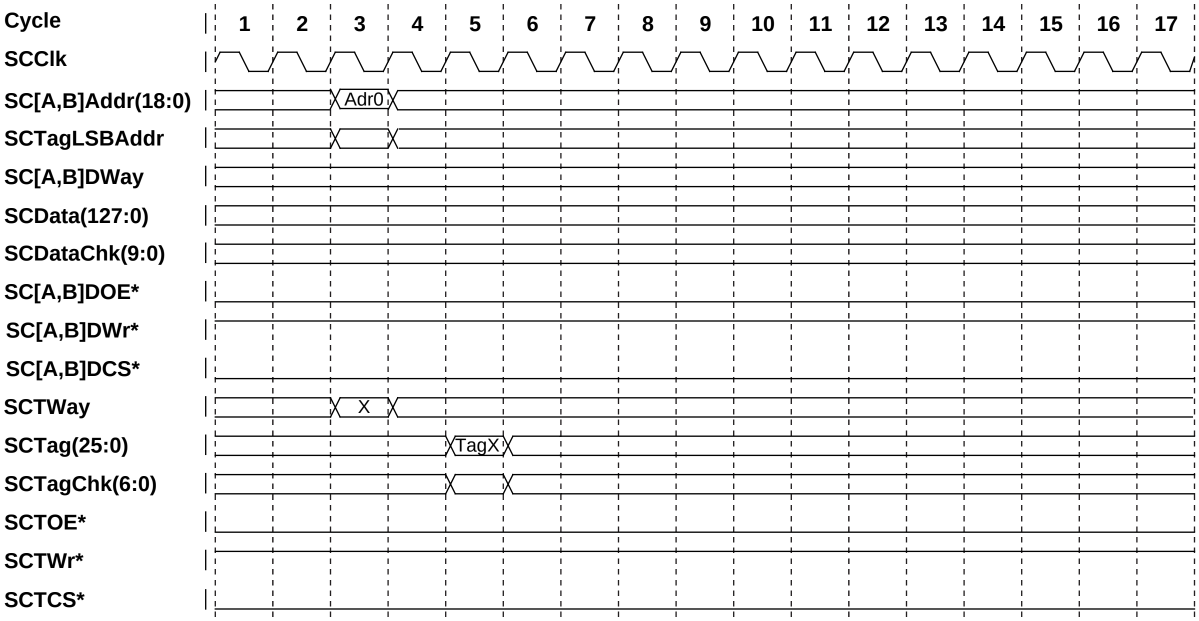

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SCClk
SC[A,B]Addr(18:0)
Adr0
SCTagLSBAddr
SC[A,B]DWay
SCData(127:0)
SCDataChk(9:0)
SC[A,B]DOE*
SC[A,B]DWr*
SC[A,B]DCS*
SCTWay
X
SCTag(25:0)
TagX
SCTagChk(6:0)
SCTOE*
SCTWr*
SCTCS*

Figure 5-6 Tag Read Sequence

5.7 Write Sequences

There are five basic write sequences:

- a 4-word write.
- an 8-word write
- a 16-word write
- a 32-word write
- a tag write

The **SCClk** referred in the secondary cache read and write timing diagrams is an internal **SCClk**. The relationship between this internal **SCClk** and the external **SCClk[5:0]/ SCClk[5:0]*** can be programmed during boot time by setting the **SCClkTap** mode bits (see the section titled “Mode Bits” in Chapter 8 for detail on mode bits).

4-Word Write Sequence

A 4-word write sequence is performed by a CACHE Index Store Data (S) instruction to store a quadword of data and 10 check bits into the secondary cache data array.

Figure 5-7 depicts a secondary cache 4-word write sequence. A quadword is written to the index specified by **PA(23:6)**, and the way specified by **VA(0)** of the CACHE instruction.

A doubleword specified by **VA(3)** is obtained from the CP0 *TagHi* and *TagLo* registers, and the other half of the doubleword is padded to zeros. Normal ECC and parity generation is bypassed and the check field of the data array is written with the contents of the CP0 *ECC(9:0)* register.

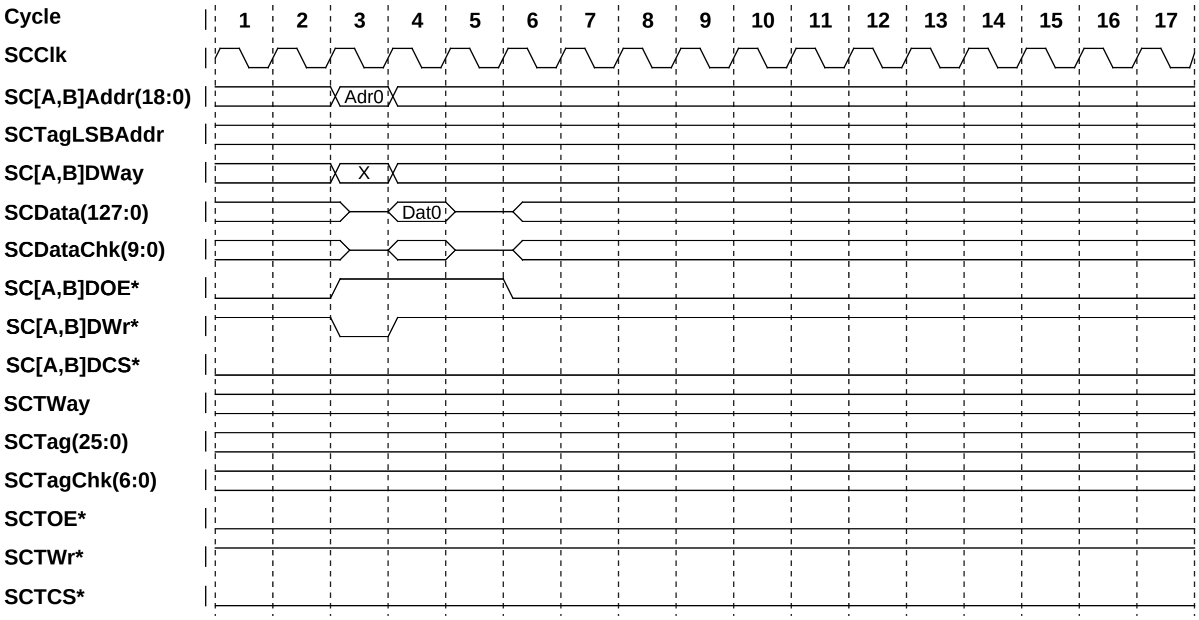

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SCClk
SC[A,B]Addr(18:0)
Adr0
SCTagLSBAddr
SC[A,B]DWay
X
SCData(127:0)
Dat0
SCDataChk(9:0)
SC[A,B]DOE*
SC[A,B]DWr*
SC[A,B]DCS*
SCTWay
SCTag(25:0)
SCTagChk(6:0)
SCTOE*
SCTWr*
SCTCS*

Figure 5-7 4-Word Write Sequence

8-Word Write Sequence

An 8-word write sequence writes back a dirty block from the primary data cache to the secondary cache.

Figure 5-8 depicts a secondary cache 8-word write sequence. **SC(A,B)DWay** are driven with the way bit obtained from the primary data cache tag. The secondary cache tag is not written since it was previously updated when the primary data cache block was modified.

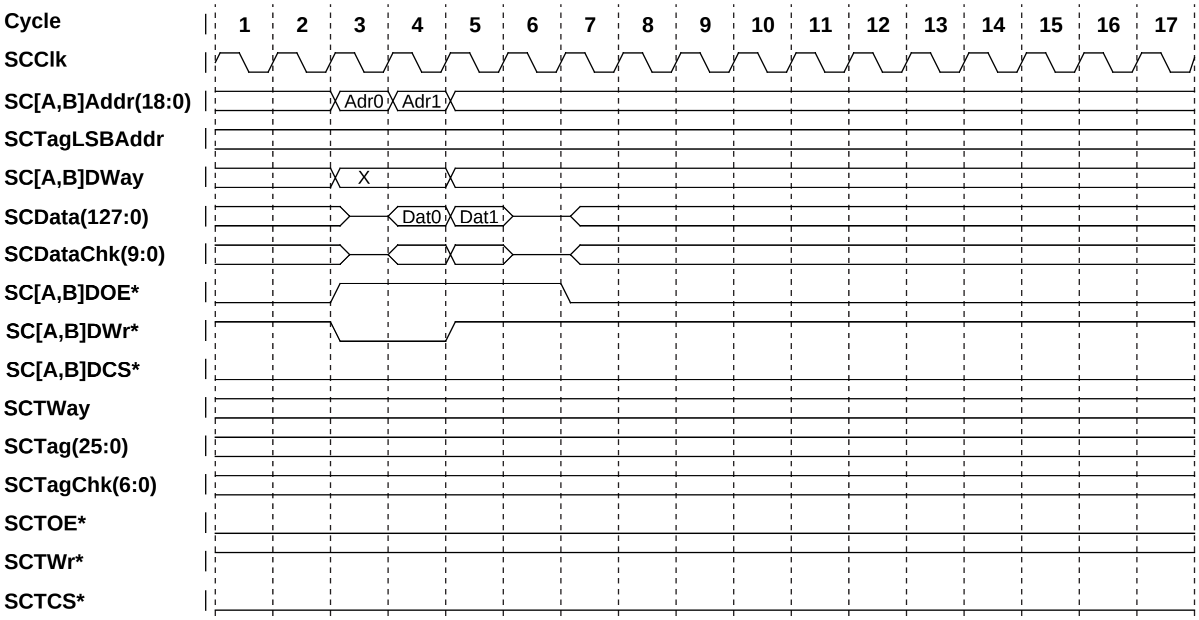

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SCClk
SC[A,B]Addr(18:0)
Adr0
Adr1
SCTagLSBAddr
SC[A,B]DWay
X
SCData(127:0)
Dat0
Dat1
SCDataChk(9:0)
SC[A,B]DOE*
SC[A,B]DWr*
SC[A,B]DCS*
SCTWay
SCTag(25:0)
SCTagChk(6:0)
SCTOE*
SCTWr*
SCTCS*

Figure 5-8 8-Word Write Sequence

16 or 32-Word Write Sequence

A 16- or 32-word write sequence refills a secondary cache block from the System interface after a secondary cache miss. A 16-word write sequence is performed when the secondary cache block size is 16 words, and a 32-word write sequence is performed when the secondary cache block size is 32 words.

Figure 5-9 depicts a secondary cache 16 or 32-word write sequence.

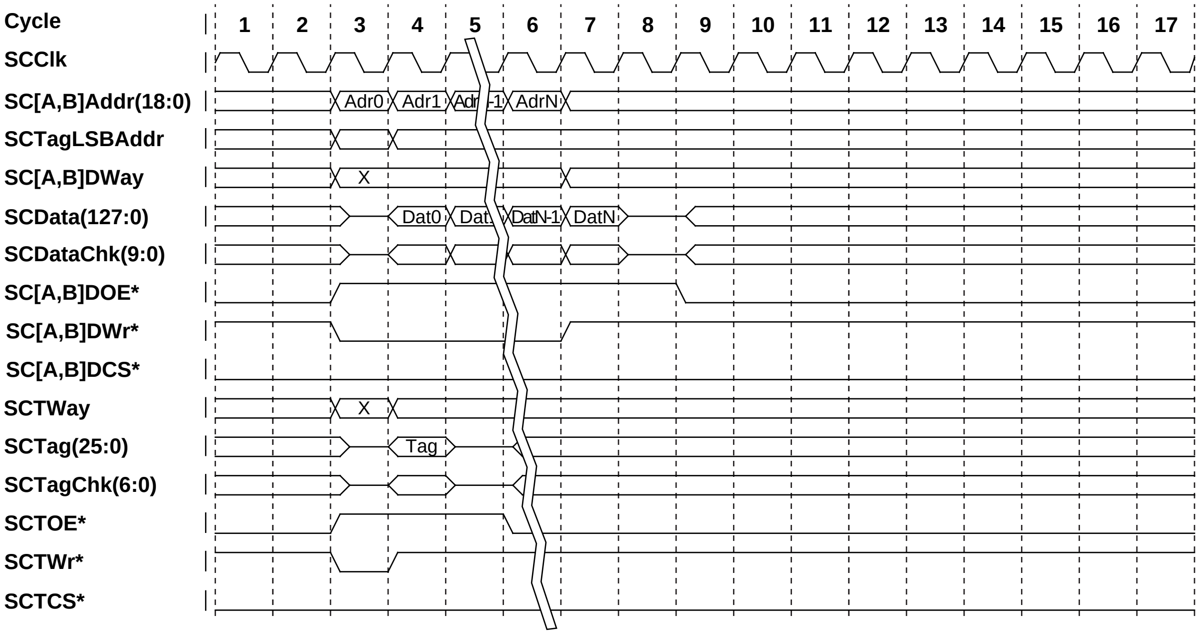

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SCClk
SC[A,B]Addr(18:0)
Adr0
Adr1
AdrN-1
AdrN
SCTagLSBAddr
SC[A,B]DWay
X
SCData(127:0)
Dat0
Dat1
DatN-1
DatN
SCDataChk(9:0)
SC[A,B]DOE*
SC[A,B]DWr*
SC[A,B]DCS*
SCTWay
X
SCTag(25:0)
Tag
SCTagChk(6:0)
SCTOE*
SCTWr*
SCTCS*

Figure 5-9 16/ 32-Word Write Sequence

Tag Write Sequence

A tag write sequence updates the secondary cache tag array without affecting the data array. This sequence is used for the following:

- to reflect primary cache state changes in the secondary cache
- for external coherency requests
- for the CACHE Index Store Tag (S) instruction

Figure 5-10 depicts the secondary cache tag write protocol.

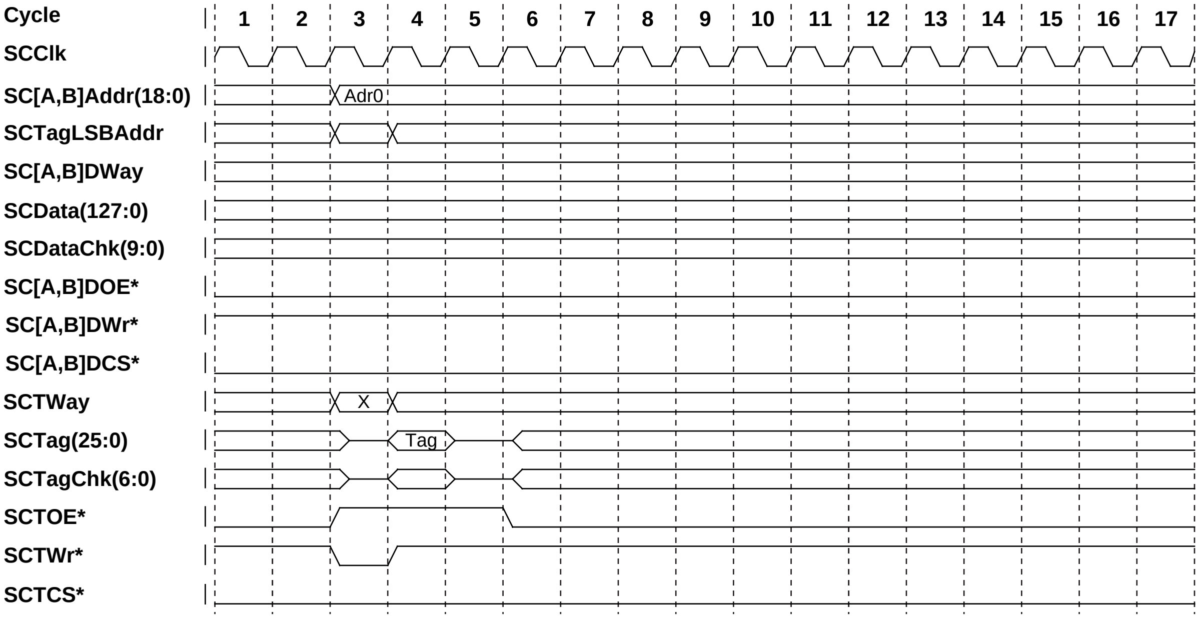

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SCClk
SC[A,B]Addr(18:0)
Adr0
SCTagLSBAddr
SC[A,B]DWay
SCData(127:0)
SCDataChk(9:0)
SC[A,B]DOE*
SC[A,B]DWr*
SC[A,B]DCS*
SCTWay
X
SCTag(25:0)
Tag
SCTagChk(6:0)
SCTOE*
SCTWr*
SCTCS*

Figure 5-10 Tag Write Sequence

6. System Interface Operations

The R10000 System interface provides a gateway between processor, with its associated secondary cache, and the remainder of the computer system.

For convenience, any device communicating with the processor through the System interface is referred to as the **external agent**.

6.1 Request and Response Cycles

The System interface supports the following request and response cycles:

- **Processor requests** are generated by the processor, when it requires a system resource.
- **External responses** are supplied by an external agent in response to a processor request.
- **External requests** are generated by an external agent when it requires a resource within the processor.
- **Processor responses** are supplied by the processor in response to an external request.

6.2 System Interface Frequencies

The System interface operates at **SysClk** frequency, supplied by the external agent. The internal processor clock, **PClk**, is derived from this same **SysClk**.

★ The **SysClkDiv** mode bits select a **PClk** to **SysClk** divisor using the formula described in Chapter 7, the section titled “System Interface Clock and Internal Processor Clock Domains.” The selectable divisors are 1, 1.5, 2, 2.5, 3, 3.5, and 4 in the R10000, or 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, and 6 in the R12000 (7 is also selectable in the R12000A only).

6.3 Register-to-Register Operation

The System interface is designed to operate in the following register-to-register fashion with the external agent:

- all System interface outputs are sourced directly from registers clocked on the rising edge of **SysClk**
- all System interface inputs directly feed registers that are clocked on the rising edge of **SysClk**

This allows the System interface to run at the highest possible clock frequency.

6.4 System Interface Signals

The R10000 System interface is composed of:

- **3 arbitration signals**
- **2 flow-control input signals**
- **a bidirectional 12-bit command bus**
- **a bidirectional 64-bit multiplexed address/data bus**
- **a 3-bit state output bus**
- **a 5-bit response input bus**

6.5 Master and Slave States

At any time, the System interface is either in *master* or *slave* state.

In **master** state, the processor drives the bidirectional System interface signals and is permitted to issue processor requests to the external agent.

In **slave** state, the processor tristates the bidirectional System interface signals and accepts external requests from the external agent.

6.6 Connecting to an External Agent

In a uni- or multiprocessor system using dedicated external agents, the System interface connects to a single external agent.

In a multiprocessor system using the cluster bus (see below), the system can connect up to four R10000 processors to an external agent. This external agent is referred to as the **cluster coordinator**.

6.7 Cluster Bus

In a multiprocessor system using the cluster bus, the cluster coordinator performs the cluster bus arbitration and data flow management. The arbitration scheme assures that either one of the processors or the cluster coordinator is master at any given time, while the remaining devices are slave.

A processor request issued by the master processor is observed as an external request by all slave R10000 processors, as shown in Figure 6-1. Similarly, a processor coherency data response issued by a master processor is observed as an external data response by the slave processors.

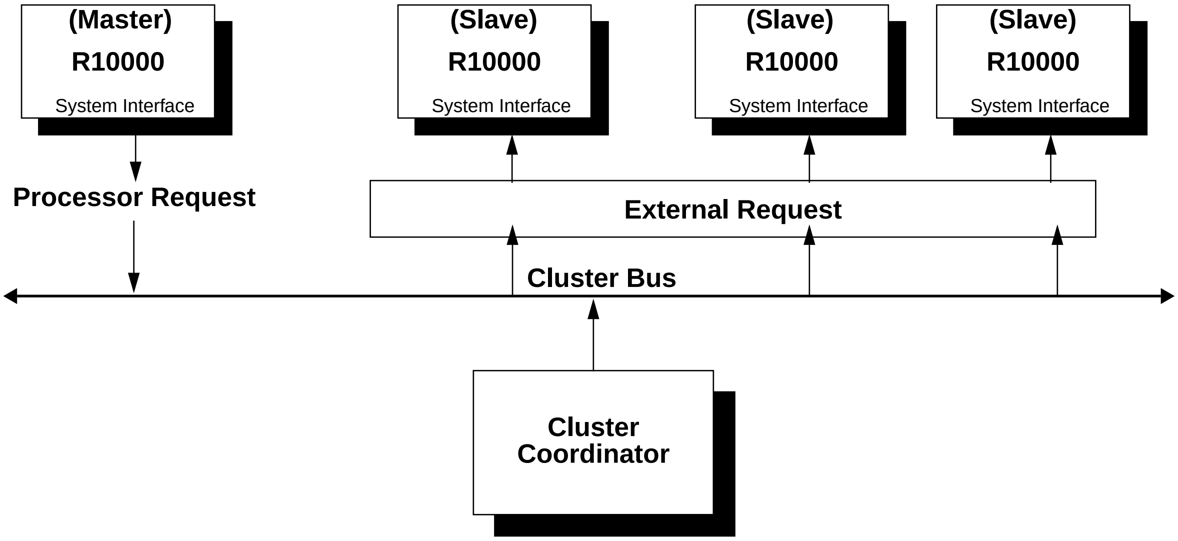

(Master)
R10000
System Interface
(Slave)
R10000
System Interface
(Slave)
R10000
System Interface
(Slave)
R10000
System Interface
Processor Request
External Request
Cluster Bus
Cluster Coordinator

Figure 6-1 Processor Request Master/Slave Status

In a multiprocessor system using the cluster bus, a mode bit specifies whether processor coherent requests are to target the external agent only, or all processors and the external agent. This allows systems with efficient snoopy, duplicate tag, or directory-based coherency protocols to be created.

6.8 System Interface Connections

The major System interface connections required for various system configurations are presented in this section.

Uniprocessor System

Figure 6-2 shows the major System interface connections required for a typical uniprocessor system.

★

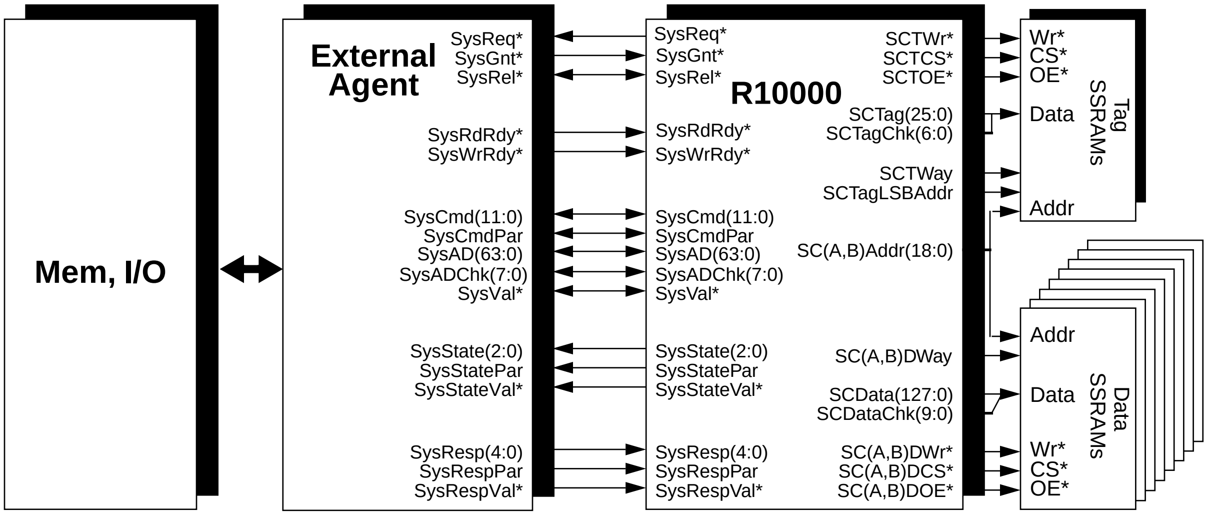

Mem, I/O
External Agent
SysReq*
SysGnt*
SysRel*
SysRdRdy*
SysWrRdy*
SysCmd(11:0)
SysCmdPar
SysAD(63:0)
SysADChk(7:0)
SysVal*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*
R10000
SysReq*
SysGnt*
SysRel*
SysRdRdy*
SysWrRdy*
SysCmd(11:0)
SysCmdPar
SysAD(63:0)
SysADChk(7:0)
SysVal*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*
SCTWr*
SCTCS*
SCTOE*
SCTag(25:0)
SCTagChk(6:0)
SCTWay
SCTagLSBAddr
SC(A,B)Addr(18:0)
SC(A,B)DWay
SCData(127:0)
SCDataChk(9:0)
SC(A,B)DWr*
SC(A,B)DCS*
SC(A,B)DOE*
Wr*
CS*
OE*
Data
Addr
Tag SSRAMs
Addr
Data
Wr*
CS*
OE*
Data SSRAMs

Figure 6-2 System Interface Connections for Uniprocessor System

Multiprocessor System Using Dedicated External Agents

Figure 6-3 shows the major System interface connections required for a typical multiprocessor system using dedicated external agents.

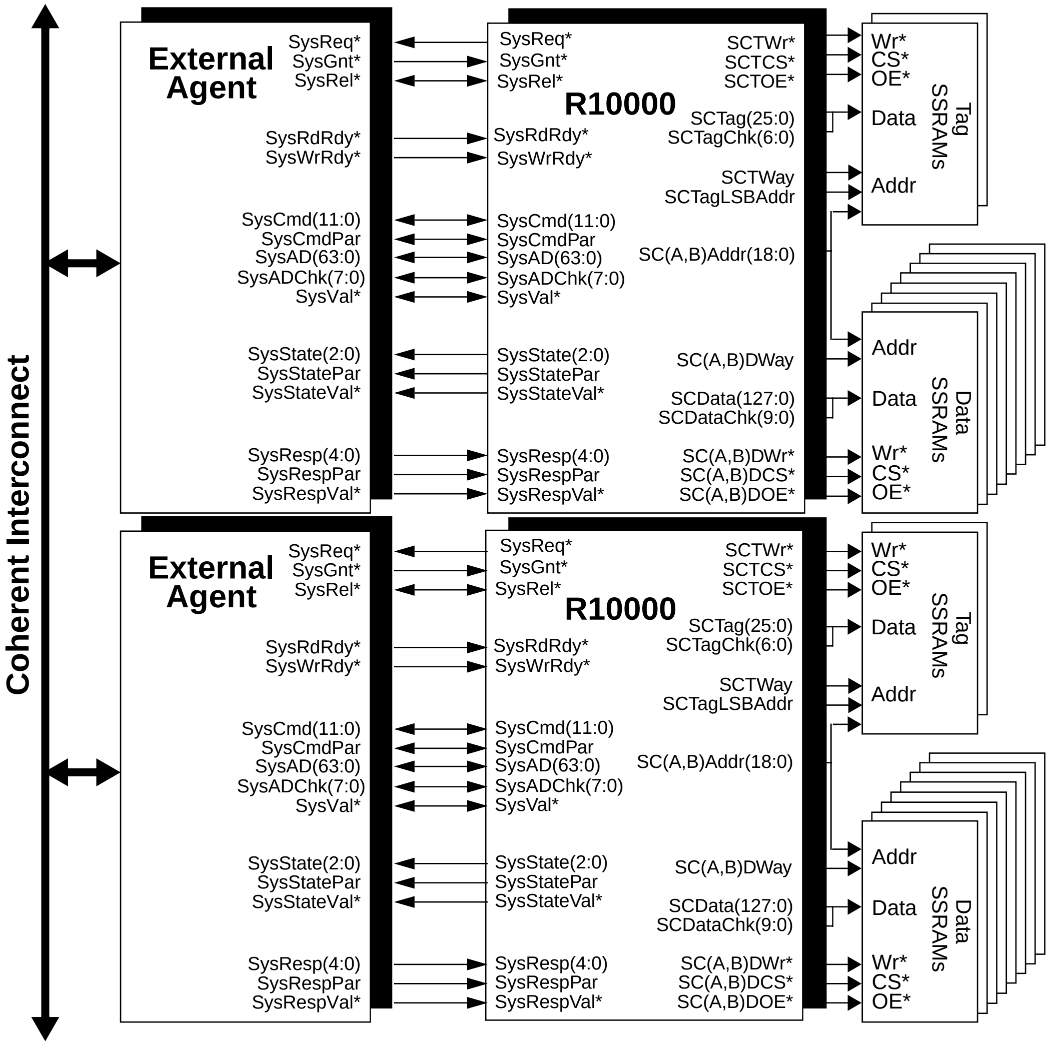

Coherent Interconnect
External Agent
SysReq*
SysGnt*
SysRel*
SysRdRdy*
SysWrRdy*
SysCmd(11:0)
SysCmdPar
SysAD(63:0)
SysADChk(7:0)
SysVal*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*
R10000
SCTWr*
SCTCS*
SCTOE*
SCTag(25:0)
SCTagChk(6:0)
SCTWay
SCTagLSBAddr
SC(A,B)Addr(18:0)
SC(A,B)DWay
SCData(127:0)
SCDataChk(9:0)
SC(A,B)DWr*
SC(A,B)DCS*
SC(A,B)DOE*
Wr*
CS*
OE*
Data
Addr
Tag SSRAMs
Data SSRAMs

Figure 6-3 System Interface Connections for Multiprocessor using Dedicated External Agents

Multiprocessor System Using the Cluster Bus

Figure 6-4 presents the major System interface connections required for a typical multiprocessor system using the cluster bus.

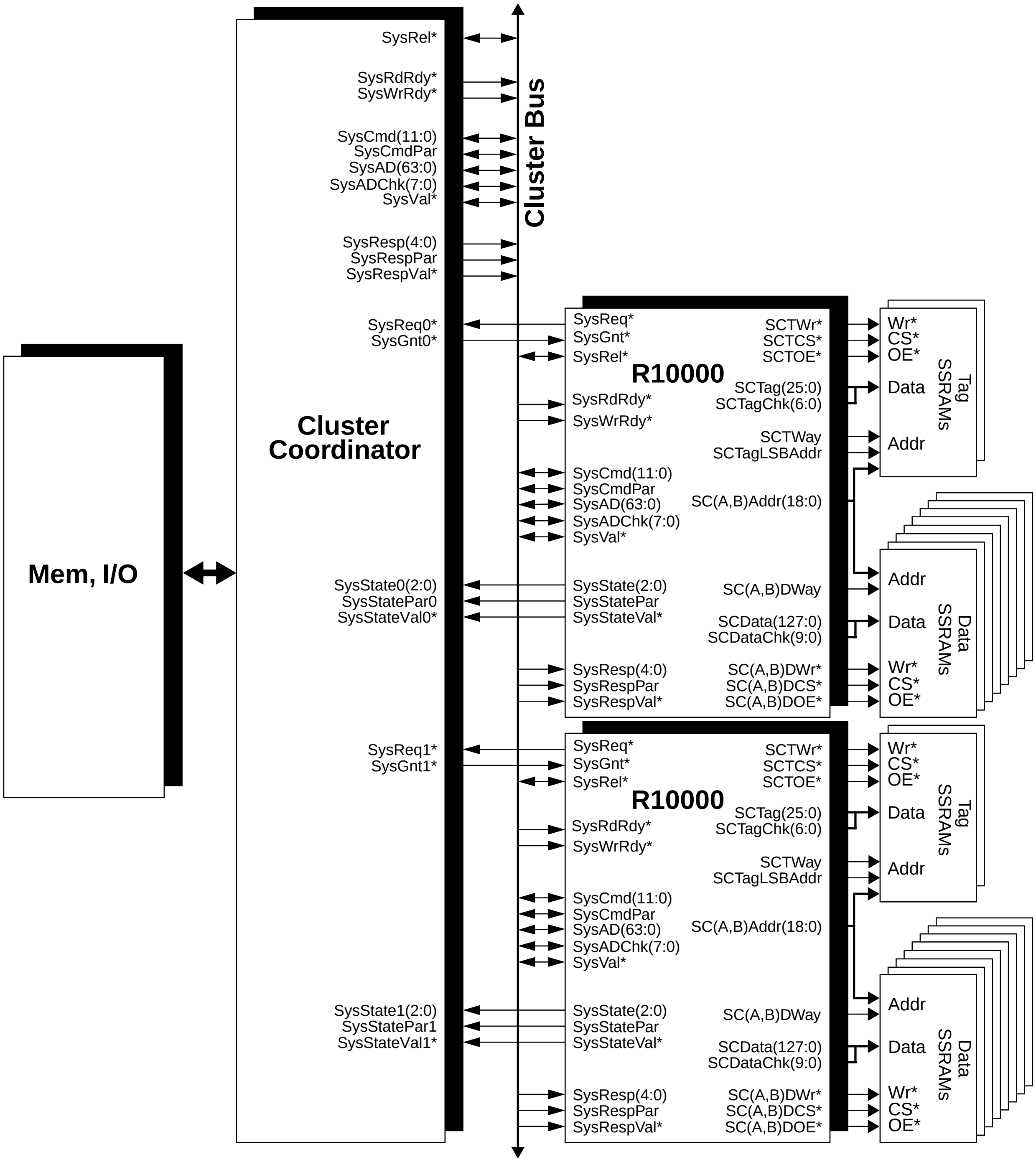

Cluster Bus
Cluster Coordinator
Mem, I/O
SysRel*
SysRdRdy*
SysWrRdy*
SysCmd(11:0)
SysCmdPar
SysAD(63:0)
SysADChk(7:0)
SysVal*
SysResp(4:0)
SysRespPar
SysRespVal*
SysReq0*
SysGnt0*
SysState0(2:0)
SysStatePar0
SysStateVal0*
SysReq1*
SysGnt1*
SysState1(2:0)
SysStatePar1
SysStateVal1*
R10000
SysReq*
SysGnt*
SysRel*
SysRdRdy*
SysWrRdy*
SysCmd(11:0)
SysCmdPar
SysAD(63:0)
SysADChk(7:0)
SysVal*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*
SCTWr*
SCTCS*
SCTOE*
SCTag(25:0)
SCTagChk(6:0)
SCTWay
SCTagLSBAddr
SC(A,B)Addr(18:0)
SC(A,B)DWay
SCData(127:0)
SCDataChk(9:0)
SC(A,B)DWr*
SC(A,B)DCS*
SC(A,B)DOE*
Wr*
CS*
OE*
Data
Addr
Tag SSRAMs
Data SSRAMs

Figure 6-4 System Interface Connections for Multiprocessor Using the Cluster Bus

6.9 System Interface Requests and Responses

The System interface supports the following:

- processor request
- external response
- external request
- processor response

The following sections describe these request and response types, and their operations.

Processor Requests

Processor requests are generated by the processor when it requires a system resource. The following processor requests are supported:

- coherent block read shared request
- coherent block read exclusive request
- noncoherent block read request
- double/single/partial-word read request
- block write request
- double/single/partial-word write request
- upgrade request
- eliminate request

Processor write and eliminate requests do not require or expect a response by the external agent. However, if an external agent detects an error in a processor write or eliminate request, it may use an interrupt to signal the processor. It is not possible to generate precise exceptions for processor write and eliminate requests for which an external agent detects an error.

Processor read and upgrade requests require some type of response by the external agent.

External Responses

External responses are supplied by an external agent or another processor in response to a processor request. The following external responses are supported:

- block data response
- double/single/partial-word data response
- completion response

External Requests

External requests are generated by an external agent when it requires a resource within the processor. The following external requests are supported:

- intervention shared request
- intervention exclusive request
- allocate request number request
- invalidate request
- interrupt request

External intervention and invalidate requests require some type of response by the processor.

Processor Responses

Processor responses are supplied by the processor in response to an external request. The following processor responses are supported:

- coherency state response
- coherency data response

Outstanding Requests and Request Numbers

The processor allows requests and corresponding responses to be split transactions, which enables additional processor and external requests to be issued while waiting for a prior response. The System interface supports a request number field to link requests with their corresponding responses, so responses can be returned out of order.

The processor allows a maximum of eight outstanding requests on the System interface through a 3-bit request number. These outstanding requests may be composed of any mix of processor and external requests.

An individual processor (as opposed to the System interface, above) supports a maximum of four outstanding processor requests at any given time.

Request and Response Relationship

The relationship between processor and external requests, and their acceptable responses, is presented in Table 6-1. The data in this table is given with respect to a single processor, in either a uni- or multiprocessor system (independent of cluster/non-cluster configuration).

Table 6-1 Request and Response Relationship

Request	Acceptable Response Sequences
Processor block read request	External NACK or ERR completion response
	0 or more external block data responses followed by a final external block data response with a coincidental or subsequent external ACK, NACK, or ERR completion response
Processor double/single/partial-word read request	External NACK or ERR completion response
	0 or more external double/single/partial-word data responses followed by a final external double/single/partial-word data response with a coincidental or subsequent external ACK, NACK, or ERR completion response
Processor block write request	None
Processor double/single/partial-word write request	None
Processor upgrade request	External ACK, NACK, or ERR completion response
	0 or more external block data responses followed by a final external block data response with a coincidental or subsequent external ACK, NACK, or ERR completion response
Processor eliminate request	None
External intervention request	Processor coherency state response followed by processor coherency data response (if *DirtyExclusive)* with a coincidental or subsequent external ACK, NACK, or ERR completion response‡
External allocate request number request	External ACK, NACK, or ERR completion response‡
External invalidate request	Processor coherency state response followed by external ACK, NACK, or ERR completion response‡
External interrupt request	None

‡ External completion response is required to free the request number.

6.10 System Interface Buffers

The processor contains the following five buffers to enhance the performance of the System interface and to simplify the system design:

- cluster request buffer
- cached request buffer
- incoming buffer
- outgoing buffer
- uncached buffer

These buffers are described in the following sections.

Cluster Request Buffer

The System interface contains an 8-entry cluster request buffer. This buffer maintains the status of the eight possible outstanding requests on the System interface. When the System interface is in master state, and it issues the address cycle of processor read or upgrade request, the processor places an entry into the cluster request buffer. When the System interface is in slave state, and an external agent issues an external coherency or allocate request number request, it places an entry into the cluster request buffer.

Once an entry is placed into the cluster request buffer, the associated request number transitions from *free* to *busy.* An entry remains busy until the processor receives an external completion response. Processor requests that are ready to be issued to the System interface bus probe the cluster request buffer to detect conflict conditions.

Cached Request Buffer

The System interface contains a four-entry cached request buffer. This buffer holds the status of the four possible outstanding processor cached requests, including processor block read and upgrade requests. The relative order of the requests is maintained in the cached request buffer.

External coherency requests probe the cached request buffer to detect conflict conditions.

Incoming Buffer

The System interface contains an incoming buffer for external block and double/single/partial-word data responses. The four 32-word entries of the incoming buffer correspond to the four possible outstanding processor requests. Block data in each entry of the incoming buffer is stored in subblock order, beginning with a quadword-aligned address.

The incoming buffer eliminates the need for the processor to flow-control the external agent that is providing the external data responses. Regardless of the cache bandwidth or internal resource availability, the external agent may supply external data response data for all outstanding read and upgrade requests at the maximum System interface data rate.

The external agent may issue any number of external data responses for a particular request number before issuing a corresponding external completion response. An external data response remains in the incoming buffer until a corresponding external completion response is received. A former buffered external data response for a particular request number is over-written by a subsequent external data response for the same request number.

An external ACK completion response frees buffered data to be forwarded to the caches and other internal resources while an external NACK or ERR completion response purges any corresponding buffered data. For minimum latency, the external agent should issue an external ACK completion response coincident with the first doubleword of an external data response.

External coherency requests that target blocks residing in the incoming buffer are stalled until the incoming buffer data is forwarded to the secondary cache, and the instruction that caused the secondary miss is satisfied.

Each doubleword of the incoming buffer has an Uncorrectable Error flag. When an external data response provides a doubleword, the processor asserts the corresponding incoming buffer Uncorrectable Error flag if the data quality indicator, **SysCmd[5]**, is asserted, or if an uncorrectable ECC error is encountered on the system address/data bus and the ECC check indication on **SysCmd[0]** is asserted.

When the processor forwards block data from an incoming buffer entry after receiving an external ACK completion response, the associated incoming buffer Uncorrectable Error flags are checked, and if any are asserted, a single Cache Error exception is posted. When the processor forwards double/single/partial-word data from an incoming buffer entry after receiving an external ACK completion response, the associated incoming buffer Uncorrectable Error flag is checked, and if asserted, a Bus Error exception is posted.

Outgoing Buffer

The System interface contains a five-entry outgoing buffer to provide buffering for the following:

- DirtyExclusive blocks that are cast out of the secondary cache because of a block replacement
- various CACHE instructions
- an external intervention request.

Four 32-word *typical* entries are associated with the four possible outstanding processor cached requests allowed by the processor. One 32-word *special* entry is reserved for external intervention requests only. The data is stored in each entry of the outgoing buffer in sequential order, beginning with a secondary cache block-aligned address.

An instruction or data access that misses in the secondary cache but targets an entry in the outgoing buffer is stalled until the outgoing buffer entry is issued as a processor block write request or coherency data response to the System interface bus.

External coherency requests probe the four typical outgoing buffer entries, with the following results:

- If an external intervention request hits a typical entry, that entry is converted from a processor block write request to a processor coherency data response.
- If an external invalidate request hits a typical outgoing buffer entry, that entry is deleted.
- If an external intervention request does not hit a typical outgoing buffer entry, but hits a *DirtyExclusive* block in the secondary cache, the special outgoing buffer entry is used to buffer the processor coherency data response.

A typical outgoing buffer entry containing a block write is ready for issue to the System interface bus when the first quadword is received from the secondary cache. The processor allows data to stream from the secondary cache to the System interface bus through the outgoing buffer.

An outgoing buffer entry containing a coherency data response is ready for issue to the System interface bus when the quadword specified by the corresponding external intervention request is received from the secondary cache. The processor then allows the data to stream from the secondary cache to the System interface bus through the outgoing buffer.

Each quadword of the outgoing buffer maintains an Uncorrectable Error flag. If an uncorrectable error is encountered while a block is being cast out of the secondary cache, the associated outgoing buffer quadword Uncorrectable Error flag is asserted. When the processor empties an outgoing buffer entry by issuing a processor block write or coherency data response, the outgoing buffer Uncorrectable Error flags are reflected by the data quality indication on **SysCmd[5]**.

Uncached Buffer

The System interface contains an uncached buffer to provide buffering for uncached and uncached accelerated load and store operations. All operations retain program order within the uncached buffer.

The uncached buffer is organized as a 4-entry FIFO followed by a 2-entry gatherer. Each gathered entry has a capacity of 16 or 32 words, as specified by the **SCBlkSize** mode bit.

The uncached buffer begins gathering when an uncached accelerated double or singleword block-aligned store is executed. Gathering continues if the subsequent uncached operation executed is an uncached accelerated double or singleword store to a sequential or identical address. Once a second uncached accelerated store is gathered, the gathering mode is determined to be sequential or identical. Gathering continues until one of the following conditions occurs:

- a complete block is gathered
- an uncached or uncached accelerated load is executed
- an uncached or uncached accelerated partial-word store is executed
- an uncached store is executed
- a change in the current gathering mode is observed
- a change in the uncached attribute is observed

When gathering terminates, the data is ready for issue to the System interface bus. A processor uncached accelerated block write request is used to issue a completely gathered uncached accelerated block. One or more disjoint processor uncached accelerated double or singleword write requests are used to issue an incompletely gathered uncached accelerated block.

When gathering in an identical mode, uncached accelerated double or singleword stores may be freely mixed. The uncached buffer packs the associated data into the gatherer. When gathering in sequential mode, uncached accelerated singleword stores must occur in pairs, to prevent an address error exception. For instance, SW, SW, SD, SW, SW is legal. SD, SW, SD, is not.

External coherency requests have no effect on the uncached buffer.

CACHE instructions have no effect on the uncached buffer. SYNC instructions are prevented from graduating if an uncached store resides in the uncached buffer.

6.11 System Interface Flow Control

The System interface supports a maximum *request rate* of one request per **SysClk** cycle, and a maximum *data rate* of one doubleword per **SysClk** cycle.

Various flow control mechanisms are provided to limit these rates, as described below.

Processor Write and Eliminate Request Flow Control

The processor can only issue a processor write or eliminate request if:

- the System interface is in master state
- **SysWrRdy*** was asserted two **SysClk** cycles previously

Processor Read and Upgrade Request Flow Control

The processor can only issue a processor read or upgrade request if:

- the System interface is in master state
- **SysRdRdy*** was asserted two **SysClk** cycles previously
- the maximum number of outstanding processor requests specified by the **PrcReqMax** mode bits is not exceeded
- there is a free request number

Processor Coherency Data Response Flow Control

The processor can only issue a processor coherency data response if:

- the System interface is in master state
- **SysWrRdy*** was asserted two **SysClk** cycles previously

External Request Flow Control

When the System interface is in *Slave* state, it is capable of accepting external requests. An external agent may issue external requests in adjacent **SysClk** cycles.

External Data Response Flow Control

Since the processor has an incoming buffer, an external agent may supply external data response data in adjacent **SysClk** cycles, without regard to cache bandwidth or internal resource availability.

6.12 System Interface Block Data Ordering

During block data transfers on the System interface **SysAD[63:0]** bus, even doublewords (Dat0, Dat2,...) always correspond to **SCData[127:64]**, and odd doublewords (Dat1, Dat3,...) always correspond to **SCData[63:0]**.

External Block Data Responses

During the address cycle of processor block read and upgrade requests, the processor specifies a quadword-aligned address. The processor expects the external block data response to be supplied in a subblock order sequence, beginning at the specified quadword-aligned address.

Processor Coherency Data Responses

The address of external intervention requests are internally aligned by the processor to a quadword address. If the processor determines that it must issue a processor coherency data response, it supplies the data in a subblock order sequence beginning at the quadword-aligned address specified by the corresponding external coherency request.

Processor Block Write Requests

During the address cycle of processor block write requests, the processor specifies a cache block-aligned address. During the subsequent data cycles for typical processor block write requests, the processor supplies the data in sequence, beginning with the secondary cache block-aligned address.

6.13 System Interface Bus Encoding

This section presents the encoding of the following four System interface buses:

- **SysCmd[11:0]**
- **SysAD[63:0]**
- **SysState[2:0]**
- **SysResp[4:0]**

SysCmd[11:0] Encoding

This section describes address and data cycle encodings for the system command bus, **SysCmd[11:0]**.

SysCmd[11] Encoding

When **SysVal*** is asserted, **SysCmd[11]** indicates whether the **SysAD[63:0]** bus represents an address or a data cycle, as shown in Table 6-2.

Table 6-2 Encoding of **SysCmd[11]**

SysCmd[11]	Data/Address Cycle Indication
0	**SysAD[63:0]** address cycle
1	**SysAD[63:0]** data cycle

SysCmd[10:0] Address Cycle Encoding

During the address cycle of processor read and upgrade requests, **SysCmd[10:8]** contain the request number, as shown in Table 6-3. The request number provides a mechanism to associate an external response with the corresponding processor request.

Table 6-3 Encoding of **SysCmd[10:8]** *for Processor Read and Upgrade Requests*

SysCmd[10:8]	Request Number

During the address cycle of processor requests, **SysCmd[7:5]** contain the command, as shown in Table 6-4.

Table 6-4 Encoding of **SysCmd[7:5]** *for Processor Requests*

SysCmd[7:5]	Command
0	Coherent block read shared
1	Coherent block read exclusive
2	Noncoherent block read
3	Double/single/partial-word read
4	Block write
5	Double/single/partial-word write
6	Upgrade
7	Special

During the address cycle of processor read requests, **SysCmd[4:3]** contain the read cause indication, as shown in Table 6-5. This information is useful in handling the associated external response.

Table 6-5 Encoding of **SysCmd[4:3]** *for Processor Read Requests*

SysCmd[4:3]	Read Cause Indication
0	Instruction access
1	Data typical access
2	Data LL/LLD access
3	Data prefetch access

During the address cycle of processor write requests, **SysCmd[4:3]** contain the write cause indication, as shown in Table 6-6. This information is useful in handling the associated write data.

Table 6-6 Encoding of **SysCmd[4:3]** *for Processor Write Requests*

SysCmd[4:3]	Write Cause Indication
0	Reserved
1	Data typical access
2	Data uncached accelerated sequential access
3	Data uncached accelerated identical access

During the address cycle of processor upgrade requests, **SysCmd[4:3]** contain the upgrade cause indication, as shown in Table 6-7. This information useful in handling the associated external response.

Table 6-7 Encoding of **SysCmd[4:3]** *for Processor Upgrade Requests*

SysCmd[4:3]	Upgrade Cause Indication
0	Reserved
1	Data typical access
2	Data SC/SCD access
3	Data prefetch access

During the address cycle of processor special requests, **SysCmd[4:3]** contain the processor special cause indication, as shown in Table 6-8. This information differentiates between the various processor special requests.

Table 6-8 Encoding of **SysCmd[4:3]** *for Processor Special Requests*

SysCmd[4:3]	Special Cause Indication
0	Reserved
1	Eliminate
2	Reserved
3	Reserved

During the address cycle of processor block read, typical block write, upgrade, and eliminate requests, **SysCmd[2:1]** contain the secondary cache block former state, as shown in Table 6-9. This information may be useful for system designs implementing a duplicate tag or a directory-based coherency protocol.

Table 6-9 Encoding of **SysCmd[2:1]** *for Processor Block Read/Write, Upgrade, Eliminate Requests*

SysCmd[2:1]	Secondary Cache Block Former State
0	Invalid
1	Shared
2	CleanExclusive
3	DirtyExclusive

During the address cycle of processor double/single/partial-word read and write requests, **SysCmd[2:0]** contain the data size indication, as shown in Table 6-10.

Table 6-10 Encoding of **SysCmd[2:0]** *for Processor Double/Single/Partial-Word Read/Write Requests*

SysCmd[2:0]	Data Size Indication
0	One byte valid (Byte)
1	Two bytes valid (Halfword)
2	Three bytes valid (Tribyte)
3	Four bytes valid (Word)
4	Five bytes valid (Quintibyte)
5	Six bytes valid (Sextibyte)
6	Seven bytes valid (Septibyte)
7	Eight bytes valid (Doubleword)

During the address cycle of external intervention and invalidate requests, **SysCmd[10:8]** contain the request number, as shown in Table 6-11. The request number provides a mechanism to associate a potential processor coherency data response with the corresponding external coherency request.

Table 6-11 Encoding of **SysCmd[10:8]** *for External Intervention and Invalidate Requests*

SysCmd[10:8]	Request Number

During the address cycle of external requests, **SysCmd[7:5]** contain the command, as shown in Table 6-12.

Table 6-12 Encoding of **SysCmd[7:5]** *for External Requests*

SysCmd[7:5]	Command
0	Intervention shared
1	Intervention exclusive
2	Allocate request number
3	Allocate request number
4	NOP
5	NOP
6	Invalidate
7	Special

During the address cycle of external special requests, **SysCmd[4:3]** contain the external special cause indication, as shown in Table 6-13. This information is used to differentiate between the various external special requests.

Table 6-13 Encoding of **SysCmd[4:3]** *for External Special Requests*

SysCmd[4:3]	Special Cause Indication
0	Reserved
1	NOP
2	Interrupt
3	Reserved

During external address cycles, **SysCmd[0]** specifies whether ECC checking and correcting is to be performed for the **SysAD[63:0]** bus, as shown in Table 6-14. During the address cycle of processor block read, data typical block write, upgrade, and eliminate requests, the processor asserts **SysCmd[0]**. Consequently, in a multiprocessor system using the cluster bus, ECC checking and correcting is enabled for external coherency requests resulting from processor coherent block read and upgrade requests.

Table 6-14 Encoding of **SysCmd[0]** *for External Address Cycles*

SysCmd[0]	ECC check indication
0	ECC checking and correcting disable
1	ECC checking and correcting enable

SysCmd[10:0] Data Cycle Encoding

During the data cycles of an external data response or a processor coherency data response, **SysCmd[10:8]** contain the request number associated with the original request, as shown in Table 6-15.

Table 6-15 Encoding of **SysCmd[10:8]** *for Data Responses*

SysCmd[10:8]	Request Number

During data cycles, **SysCmd[5]** indicates the data quality, as shown in Table 6-16.

Table 6-16 Encoding of **SysCmd[5]** *for Data Cycles*

SysCmd[5]	Data quality indication
0	Data is good
1	Data is bad

During data cycles, **SysCmd[4:3]** indicate the data type, as shown in Table 6-17. Processor block write and double/single/partial-word write requests use request data and request last data type indications. External data and processor coherency data responses use response data and response last data type indications.

Table 6-17 Encoding of **SysCmd[4:3]** *for Data Cycles*

SysCmd[4:3]	Data type Indication
0	Request data
1	Response data
2	Request last
3	Response last

During data cycles of an external block data response or processor coherency data response, **SysCmd[2:1]** contain the state of the cache block, as shown in Table 6-18.

Table 6-18 Encoding of **SysCmd[2:1]** *for Block Data Responses*

SysCmd[2:1]	Cache Block State
0	Reserved
1	*Shared*
2	*CleanExclusive*
3	*DirtyExclusive*

During data cycles, **SysCmd[0]** specifies whether ECC checking and correcting is to be performed for the **SysAD[63:0]** bus, as shown in Table 6-19. During processor data cycles, the processor asserts **SysCmd[0]**. Consequently, in a multiprocessor system using the cluster bus, ECC checking and correcting will be enabled for external block data responses resulting from processor coherency data responses.

Table 6-19 Encoding of **SysCmd[0]** *for External Data Cycles*

SysCmd[0]	ECC check indication
0	ECC checking and correcting disable
1	ECC checking and correcting enable

SysCmd[11:0] Map

Table 6-20 presents a map for the **SysCmd[11:0]** bus.

Table 6-20 SysCmd[11:0] Map

Cycle Type	Command		SysCmd[11:0] Bit											
			11	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Processor address cycles	Coherent block read shared		0	Request number			0	0	0	Read cause		Block state		1
	Coherent block read exclusive						0	0	1					
	Noncoherent block read						0	1	0					
	Double/single/partial-word read						0	1	1			Data size		
	Block write			0			1	0	0	Write cause		Block state		1
	Double/single/partial-word write						1	0	1			Data size		
	Upgrade			Request number			1	1	0	Upgrade cause		Block state		1
	Special	Reserved		Reserved			1	1	1	0	0	Reserved		
		Eliminate		0						0	1	Block state		1
		Reserved		Reserved						1	0	Reserved		
										1	1			
Processor data cycles	Double/single/partial-word write		1	0			0		0	Data type		0		1
	Block write								Data quality			Block state		
	Coherency data response			Request number										
External address cycles	Intervention shared		0	Request number			0	0	0	X				ECC
	Intervention exclusive						0	0	1					
	Allocate request number						0	1	0					X
							0	1	1					
	NOP			X			1	0	0					
							1	0	1					
	Invalidate			Request number			1	1	0					ECC
	Special	NOP		X			1	1	1	0	0	X		X
										0	1			
		Interrupt								1	0			ECC
		NOP								1	1			X
External data cycles	Block data response		1	Request number			X		Data quality	Data type		Block state		ECC
	Double/single/partial-word data response											X		

SysAD[63:0] Encoding

This section describes the system address/data bus encoding.

SysAD[63:0] Address Cycle Encoding

Table 6-21 presents the encoding of the **SysAD[63:0]** bus for address cycles.

Table 6-21 Encoding of **SysAD[63:0]** *for Address Cycles*

SysAD[63:60]	Target Indication
SysAD[63]	Target processor with **DevNum** = 3
SysAD[62]	Target processor with **DevNum** = 2
SysAD[61]	Target processor with **DevNum** = 1
SysAD[60]	Target processor with **DevNum** = 0
SysAD[59:58]	Uncached attribute
SysAD[57]	Secondary cache block way indication
SysAD[56:40]	Reserved
SysAD[39:0]	Physical address

SysAD[63:60]

During the address cycle of processor noncoherent block read, double/single/partial-word read, block write, double/single/partial-word write, and eliminate requests, the processor always drives a target indication of 0 on **SysAD[63:60]**. This indicates that the request targets the external agent only. When the **CohPrcReqTar** mode bit is negated, during the address cycle of processor coherent block read and upgrade requests, the processor also drives a target indication of 0 on **SysAD[63:60]**. However, when the **CohPrcReqTar** mode bit is asserted, during the address cycle of processor coherent block read and upgrade requests, the processor drives a target indication of 0xF on **SysAD[63:60]**. This indicates that the request targets all processors, together with the external agent, on the cluster bus. In multiprocessor systems using the cluster bus, the **CohPrcReqTar** mode bit is asserted for a snoopy-based coherency protocol, and negated for a duplicate tag or directory-based coherency protocol.

When the processor is in slave state, an external agent uses the target indication field to specify which processors are targets of an external request.

SysAD[59:58] Uncached Attribute

During the address cycle of processor double/single/partial-word read and write requests and during the address cycle of processor *Uncached accelerated* block write requests, the processor drives the uncached attribute onto **SysAD[59:58]**. See the section titled, Support for Uncached Attribute, in this chapter for more information.

SysAD[57]

During the address cycle of processor block read, typical block write, upgrade, and eliminate requests, **SysAD[57]** contains the secondary cache block way indication. This information may be useful for system designs implementing a duplicate tag or a directory-based coherency protocol.

SysAD[56:40]

When processor is in master state, it drives **SysAD[56:40]** to zero during address cycles.

SysAD[39:0]

During the address cycle of processor and external requests, **SysAD[39:0]** contain the physical address.

Table 6-22 presents the processor request address cycle address alignment.

Table 6-22 Processor Request Address Cycle Alignment

Processor Request Type	Address Alignment	Address Bits Which Are Driven to 0
Block read	Quadword	3:0
Doubleword read/write	Doubleword	2:0
Singleword read/write	Singleword	1:0
Halfword read/write	Halfword	0
Byte, tribyte, quintibyte, sextibyte, septibyte read/write	Byte	-
Block write	Secondary cache block	5:0 (**SCBlkSize** = 0) 6:0 (**SCBlkSize** = 1)
Upgrade	Quadword	3:0
Eliminate	Secondary cache block	5:0 (**SCBlkSize** = 0) 6:0 (**SCBlkSize** = 1)

Table 6-23 presents the external coherency request address cycle address alignment.

Table 6-23 External Coherency Request Address Cycle Alignment

External Request Type	Address Alignment	Address Bits Which Are Ignored
Intervention	Quadword	3:0
Invalidate	Secondary cache block	5:0 (**SCBlkSize** = 0) 6:0 (**SCBlkSize** = 1)

SysAD[63:0] Data Cycle Encoding

During System interface data cycles, when less than a doubleword is transferred on the **SysAD[63:0]** bus, the valid byte lanes depend on the request address and the **MemEnd** mode bit.

For example, consider the data cycle for a byte request whose address modulo 8 is 1. When **MemEnd** is negated (little endian), the **SysAD[15:8]** byte lane is valid. When **MemEnd** is asserted (big endian), the **SysAD[55:48]** byte lane is valid.

SysState[2:0] Encoding

The processor provides a processor coherency state response by driving the targeted secondary cache block tag quality indication on **SysState[2]**, driving the targeted secondary cache block former state on **SysState[1:0]** and asserting **SysStateVal*** for one **SysClk** cycle. Table 6-24 presents the encoding of the **SysState[2:0]** bus when **SysStateVal*** is asserted.

Table 6-24 Encoding of **SysState[2:0]** *when* **SysStateVal*** *Asserted*

SysState[2]	Secondary cache block tag quality indication
0	Tag is good
1	Tag is bad
SysState[1:0]	**Secondary cache block former state**
0	*Invalid*
1	*Shared*
2	*CleanExclusive*
3	*DirtyExclusive*

When **SysStateVal*** is negated, **SysState[0]** indicates if a processor coherency data response is ready for issue. Table 6-25 presents the encoding of the **SysState[2:0]** bus when **SysStateVal*** is negated.

Table 6-25 Encoding of **SysState[2:0]** *When* **SysStateVal*** *Negated*

SysState[2:1]	Reserved
0	Reserved
1	
2	
3	
SysState[0]	**Processor coherency data response indication**
0	Not ready for issue
1	Ready for issue

SysResp[4:0] Encoding

An external agent issues an external completion response by driving the request number associated with the corresponding request on **SysResp[4:2]**, driving the completion indication on **SysResp[1:0]**, and asserting **SysRespVal*** for one **SysClk** cycle. Table 6-26 presents the encoding of the **SysResp[4:0]** bus.

Table 6-26 Encoding of **SysResp[4:0]**

SysResp[4:2]	Request number
SysResp[1:0]	**Completion indication**
0	Acknowledge (ACK)
1	Error (ERR)
2	Negative acknowledge (NACK)
3	Reserved

6.14 Interrupts

The processor supports five hardware, two software, one timer, and one nonmaskable interrupt. The Interrupt exception is described in Chapter 15, the section titled “Interrupt Exception.”

Hardware Interrupts

Five hardware interrupts are accessible to an external agent via external interrupt requests.

An external interrupt request consists of a single address cycle on the System interface. During the address cycle, **SysAD[63:60]** specify the target indication, which allows an external agent to define the target processors of the external interrupt request. If a processor determines it is an external interrupt request target, **SysAD[20:16]** are the write enables for the five individual *Interrupt* register bits and **SysAD[4:0]** are the values to be written into these bits, as shown in Figure 6-5. This allows any subset of the *Interrupt* register bits to be set or cleared with a single external interrupt request.

The *Interrupt* register is an architecturally transparent, level-sensitive register that is directly readable as bits 14:10 of the *Cause* register. Since it is level-sensitive, an interrupt bit must remain asserted until the interrupt is taken, at which time the interrupt handler must cause a second external interrupt request to clear the bit.

The processor clears the *Interrupt* register during any of the reset sequences.

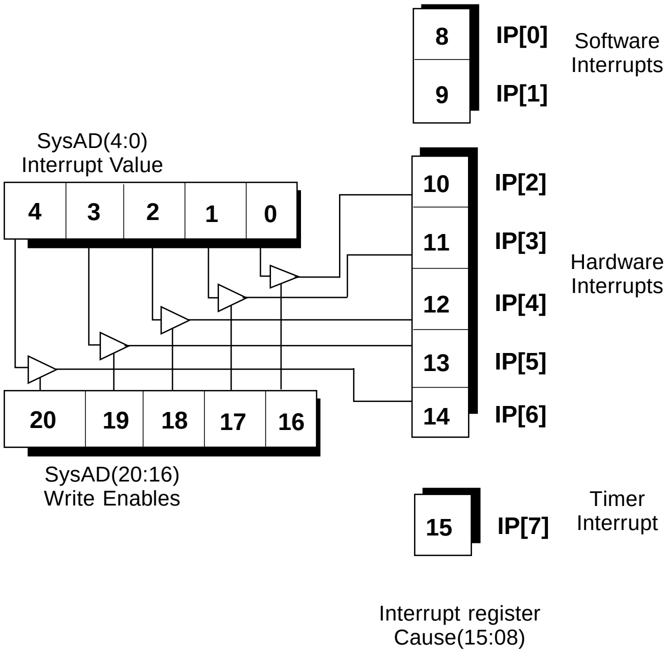

8
9
IP[0]
IP[1]
Software Interrupts
SysAD(4:0) Interrupt Value
4
3
2
1
0
10
11
12
13
14
IP[2]
IP[3]
IP[4]
IP[5]
IP[6]
Hardware Interrupts
20
19
18
17
16
SysAD(20:16) Write Enables
15
IP[7]
Timer Interrupt
Interrupt register Cause(15:08)

Figure 6-5   Hardware Interrupts

Software Interrupts

The two software interrupts are accessible as bits 9:8 of the *Cause* register, as shown in Figure 6-5. An MTC0 instruction is used to write these bits.

Timer Interrupt

The timer interrupt is accessible as bit 15 of the *Cause* register, **IP[7]**, as shown in Figure 6-5. This bit is set when one of the following occurs:

- **the *Count* register is equal to the *Compare* register**
- **either one of the two performance counters overflows**

Nonmaskable Interrupt

A nonmaskable interrupt is accessible to an external agent as the **SysNMI*** signal. To post a nonmaskable interrupt, an external agent asserts **SysNMI*** for at least one **SysClk** cycle.

The processor recognizes the nonmaskable interrupt on the first **SysClk** cycle that **SysNMI*** is asserted. After the nonmaskable interrupt is serviced, an external agent may post another nonmaskable interrupt by first negating **SysNMI*** for at least one **SysClk** cycle, and reasserting **SysNMI*** for at least one **SysClk** cycle.

6.15 Protocol Abbreviations

The following abbreviations are used in the System interface protocols:

SysCmd[11:0] Abbreviations

Cmd	Unspecified command
BlkRd	Block read request command
RdShd	Coherent block read shared request command
RdExc	Coherent block read exclusive request command
DSPRd	Double/single/partial-word read command
BlkWr	Block write request command
DSPWr	Double/single/partial-word write request command
Ugd	Upgrade request command
Elm	Eliminate request command
IvnShd	Intervention shared request command
IvnExc	Intervention exclusive request command
Alc	Allocate request number command
Ivd	Invalidate request command
Int	Interrupt request command
ExtCoh	External coherency request command
ReqDat	Request data
RspDat	Response data
ReqLst	Request last
RspLst	Response last
Empty	Empty; **SysCmd(11:0)** and **SysAD(63:0)** are undefined

SysAD[63:0] Abbreviations

Adr	Physical address
Dat	Unspecified data
Dat<n>	Doubleword n of a block

SysState[2:0] Abbreviations

State	Unspecified state
Ivd	*Invalid*
Shd	*Shared*
ClnExc	*CleanExclusive*
DrtExc	*DirtyExclusive*

SysResp[4:0] Abbreviations

Rsp	Unspecified completion response
ACK	Acknowledge completion response
ERR	Error completion response
NACK	Negative acknowledge completion response

Master Abbreviations

EA	External agent
Pn	R10000 processor whose device number is *n*
-	Dead cycle

6.16 System Interface Arbitration

The processor supports a simple System interface arbitration protocol, which relies on an external arbiter. This protocol is used in uniprocessor systems, multiprocessor systems using dedicated external agents, and multiprocessor systems using the cluster bus. System interface arbitration is handled by the **SysReq***, **SysGnt***, and **SysRel*** signals (request, grant, and release).

As described earlier in this chapter, the System interface resides in either master or slave state; the processor enters slave state during all of the reset sequences.

When mastership of the System interface changes, there is always one dead **SysClk** cycle during which the bidirectional signals are not driven; the processor ignores all bidirectional signals during this dead **SysClk** cycle.

The protocol supports overlapped arbitration which allows arbitration to occur in parallel with requests and responses. This results in fewer wasted cycles when mastership of the System interface changes.

Grant parking is also supported, allowing a device to retain mastership of the System interface as long as no other device requests the System interface.

In multiprocessor systems using the cluster bus, the external arbiter typically implements a round-robin priority scheme.

System Interface Arbitration Rules

The rules for the System interface arbitration are listed below:

- If the System interface is in slave state, and a processor request or coherency data response is ready for issue, and the required resources are available (e.g. a free request number, **SysRdRdy*** asserted, etc.), the processor asserts **SysReq***. The processor will not assert **SysReq*** unless all of the above conditions are met.
- The processor waits for the assertion of **SysGnt***.
- When the processor observes the assertion of **SysGnt*** it negates **SysReq*** two **SysClk** cycles later. Once the processor asserts **SysReq***, it does not negate **SysReq*** until the assertion of **SysGnt***, even if the need for the System interface bus is contravened by an external coherency request.
- When the processor observes the assertion of **SysRel***, it enters master state two **SysClk** cycles later, and begins to drive the System interface bus. **SysRel*** may be asserted coincidentally with or later than **SysGnt***.
- Once in master state, the processor does not relinquish mastership of the System interface until it observes the negation of **SysGnt***.
- The processor indicates it is relinquishing mastership of the System interface bus by asserting **SysRel*** for one **SysClk** cycle, two or more **SysClk** cycles after the negation of **SysGnt***. The processor may issue any type of processor request or coherency data response in the two **SysClk** cycles following the negation of **SysGnt***. This may delay the assertion of **SysRel***.

Uniprocessor System

Figure 6-6 shows how the System interface arbitration signals are used in a uniprocessor system. Note that this same configuration would be used in a multiprocessor system using dedicated external agents.

★

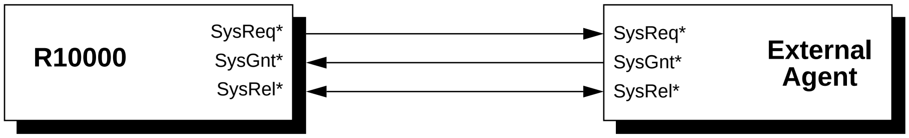

R10000
SysReq*
SysGnt*
SysRel*
SysReq*
SysGnt*
SysRel*
External Agent

Figure 6-6 Arbitration Signals for Uniprocessor System

Figure 6-7 is an example of the operation of the System interface arbitration in a uniprocessor system. The *Master* row in the following figures indicates which device is driving the System interface bidirectional signals (P_0 and EA in Figure 6-7). When this row contains a dash (-), as shown in Cycle 12 of Figure 6-7, mastership of the System interface is changing and no device is driving the System interface bidirectional signals for this one dead **SysClk** cycle.

The external agent generally asserts the **SysGnt*** signal, which allows the processor to issue requests at any time.

When the external agent needs to return an external data response, it negates **SysGnt*** for a minimum of one cycle, waits for the processor to assert **SysRel***, and then begins driving the System interface bus after one dead **SysClk** cycle.

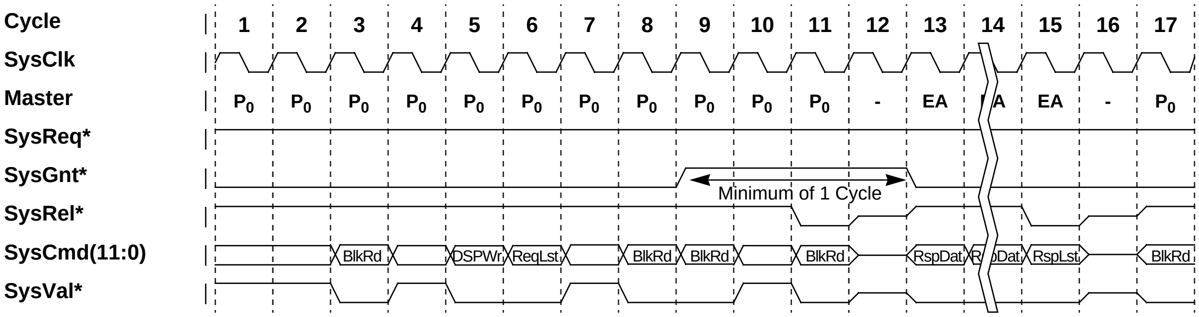

Cycle
SysClk
Master
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
SysVal*
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
P0 P0 P0 P0 P0 P0 P0 P0 P0 P0 P0 - EA EA EA - P0
Minimum of 1 Cycle
BlkRd
DSPWr
ReqLst
BlkRd
BlkRd
BlkRd
RspDat
RspDat
RspLst
BlkRd

Figure 6-7 Arbitration Protocol for Uniprocessor System

Multiprocessor System Using Cluster Bus

Figure 6-8 shows how the System interface arbitration signals are used in a four-processor system using the cluster bus.

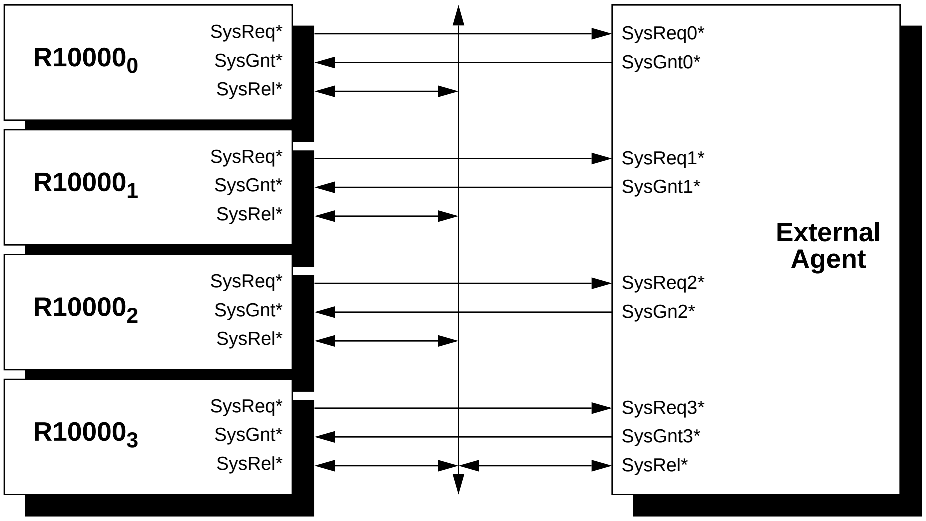

R10000₀
SysReq*
SysGnt*
SysRel*
R10000₁
SysReq*
SysGnt*
SysRel*
R10000₂
SysReq*
SysGnt*
SysRel*
R10000₃
SysReq*
SysGnt*
SysRel*
SysReq0*
SysGnt0*
SysReq1*
SysGnt1*
External Agent
SysReq2*
SysGn2*
SysReq3*
SysGnt3*
SysRel*

Figure 6-8 Arbitration Signals for Multiprocessor System Using the Cluster Bus

Figure 6-9 is an example of the System interface arbitration in a four-processor system using the cluster bus.

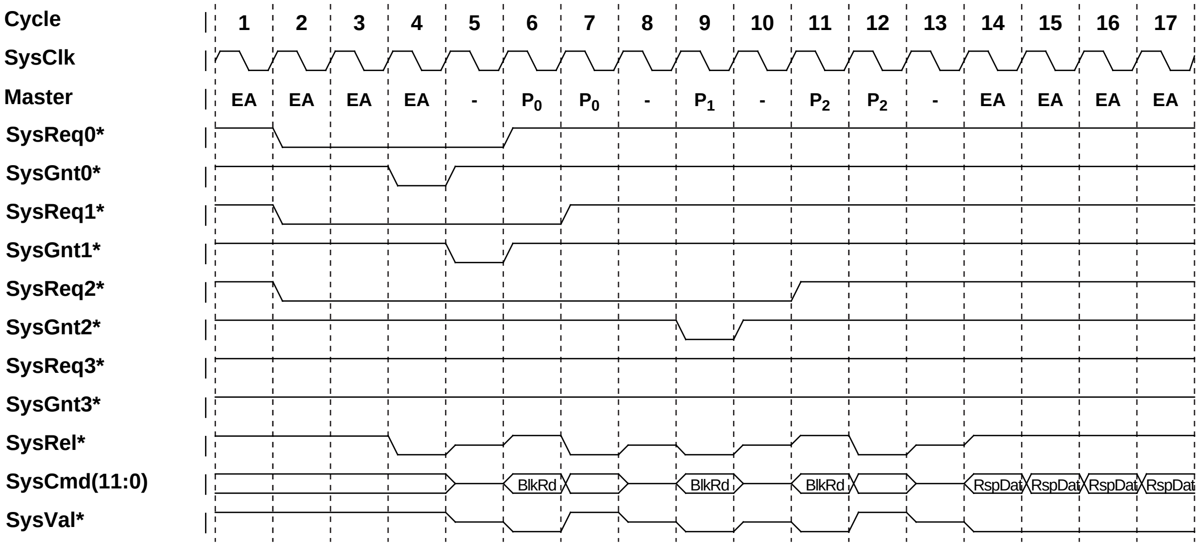

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
EA
EA
EA
EA
-
P₀
P₀
-
P₁
-
P₂
P₂
-
EA
EA
EA
EA
SysReq0*
SysGnt0*
SysReq1*
SysGnt1*
SysReq2*
SysGnt2*
SysReq3*
SysGnt3*
SysRel*
SysCmd(11:0)
BlkRd
BlkRd
BlkRd
RspDat
RspDat
RspDat
RspDat
SysVal*

Figure 6-9 Arbitration Protocol for Multiprocessor System Using the Cluster Bus

6.17 System Interface Request and Response Protocol

The following sections detail the System interface request and response protocol. A 32-word secondary cache block size is assumed in the examples below.

Processor Request Protocol

A processor request is generated when the R10000 processor requires a system resource.

The processor may only issue a processor request when the System interface is in master state. If the System interface is in master state, the processor may issue a processor request immediately. Processor requests may occur in adjacent **SysClk** cycles. If the System interface is not in master state, the processor must first assert **SysReq***, and then wait for the external agent to relinquish mastership of the System interface bus by asserting **SysGnt*** and **SysRel***.

When multiple, nonconflicting processor requests and/or coherency data responses are ready and meet all issue requirements, the processor uses the following priority:

- block read and upgrade requests have the highest priority, followed by
- processor coherency data responses,
- processor eliminate and typical block write requests,
- processor double/single/partial-word read/write and uncached accelerated block write requests, which have the lowest priority.

Processor Block Read Request Protocol

A processor block read request results from a cached instruction fetch, load, store, or prefetch that misses in the secondary cache. Before issuing a processor block read request, the processor changes the secondary cache state to *Invalid*. Additionally, if the secondary cache block former state was *DirtyExclusive*, a write back is scheduled. Note that if the processor block read request receives an external NACK or ERR completion response, the secondary cache block state remains *Invalid*.

The processor issues a processor block read request with a single address cycle. The address cycle consists of the following:

- negating **SysCmd[11]**
- driving a free request number on **SysCmd[10:8]**
- driving the block read command on **SysCmd[7:5]**
- driving the read cause indication on **SysCmd[4:3]**
- driving the secondary cache block former state on **SysCmd[2:1]**
- asserting **SysCmd[0]**
- driving the target indication on **SysAD[63:60]**
- driving the secondary cache block way on **SysAD[57]**
- driving the physical address on **SysAD[39:0]**
- asserting **SysVal***

The processor may only issue a processor block read request address cycle when the following are true:

- the System interface is in master state
- **SysRdRdy*** was asserted two **SysClk** cycles earlier
- there is no conflicting entry in the outgoing buffer
- the maximum number of outstanding processor requests specified by the **PrcReqMax** mode bits is not exceeded
- there is a free request number
- the processor is not the target of a conflicting outstanding external coherency request

A single processor may have as many as four processor block read requests outstanding on the System interface at any given time.

Figure 6-10 depicts four processor block read requests. Since the System interface is initially in slave state, the processor must first assert **SysReq*** and then wait until the external agent relinquishes mastership of the System interface by asserting **SysGnt*** and **SysRel***.

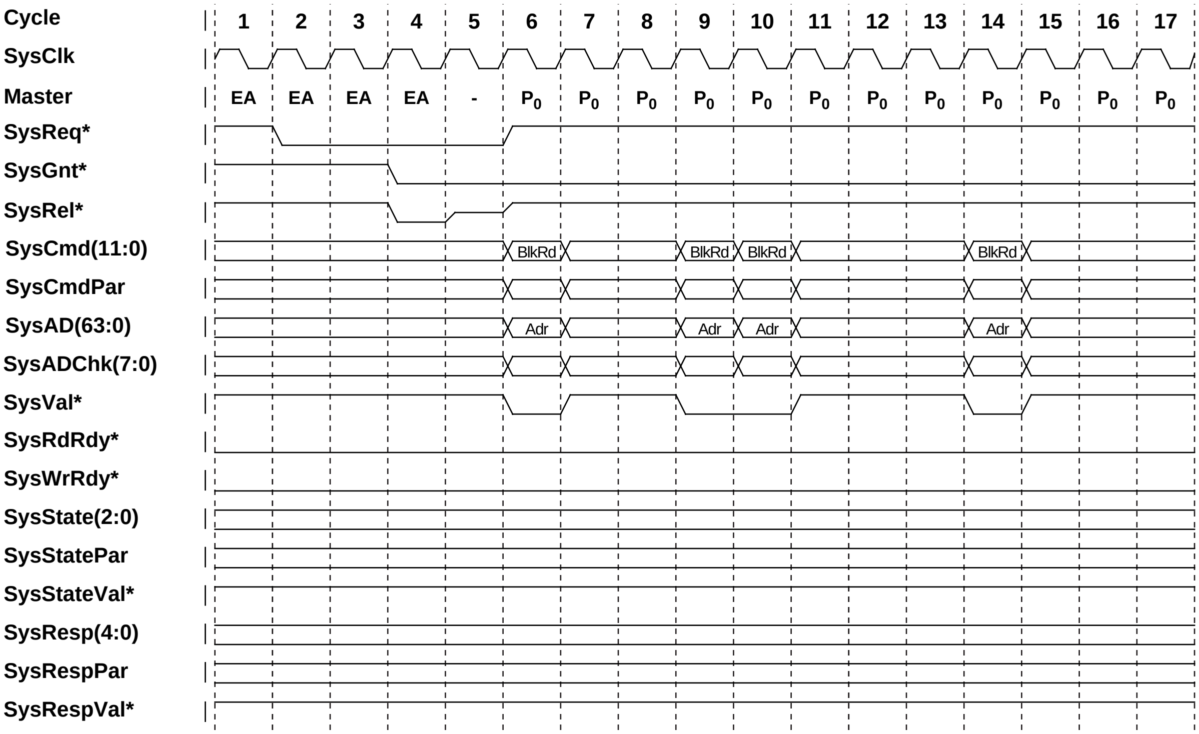

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
EA
EA
EA
EA
-
P0
P0
P0
P0
P0
P0
P0
P0
P0
P0
P0
P0
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
BlkRd
BlkRd
BlkRd
BlkRd
SysCmdPar
SysAD(63:0)
Adr
Adr
Adr
Adr
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*

Figure 6-10 Processor Block Read Request Protocol

Processor Double/Single/Partial-Word Read Request Protocol

A processor double/single/partial-word read request results from an uncached instruction fetch or load.

The processor issues a processor double/single/partial-word read request with a single address cycle. The address cycle consists of:

- negating **SysCmd[11]**
- driving a free request number on **SysCmd[10:8]**
- driving the double/single/partial-word read command on **SysCmd[7:5]**
- driving the read cause indication on **SysCmd[4:3]**
- driving the data size indication on **SysCmd[2:0]**
- driving the target indication on **SysAD[63:60]**
- driving the uncached attribute on **SysAD[59:58]**
- driving the physical address on **SysAD[39:0]**
- asserting **SysVal***

The processor may only issue a processor double/single/partial-word read request address cycle when:

- the System interface is in master state
- **SysRdRdy*** was asserted two **SysClk** cycles previously
- the maximum number of outstanding processor requests specified by the **PrcReqMax** mode bits is not exceeded
- there is a free request number

A single processor may have a maximum of one processor double/single/partial-word read request outstanding on the System interface at any given time.

Figure 6-11 depicts a processor double/single/partial-word read request. Since the System interface is initially in slave state, the processor must first assert **SysReq*** and then wait until the external agent gives up mastership of the System interface by asserting **SysGnt*** and **SysRel***.

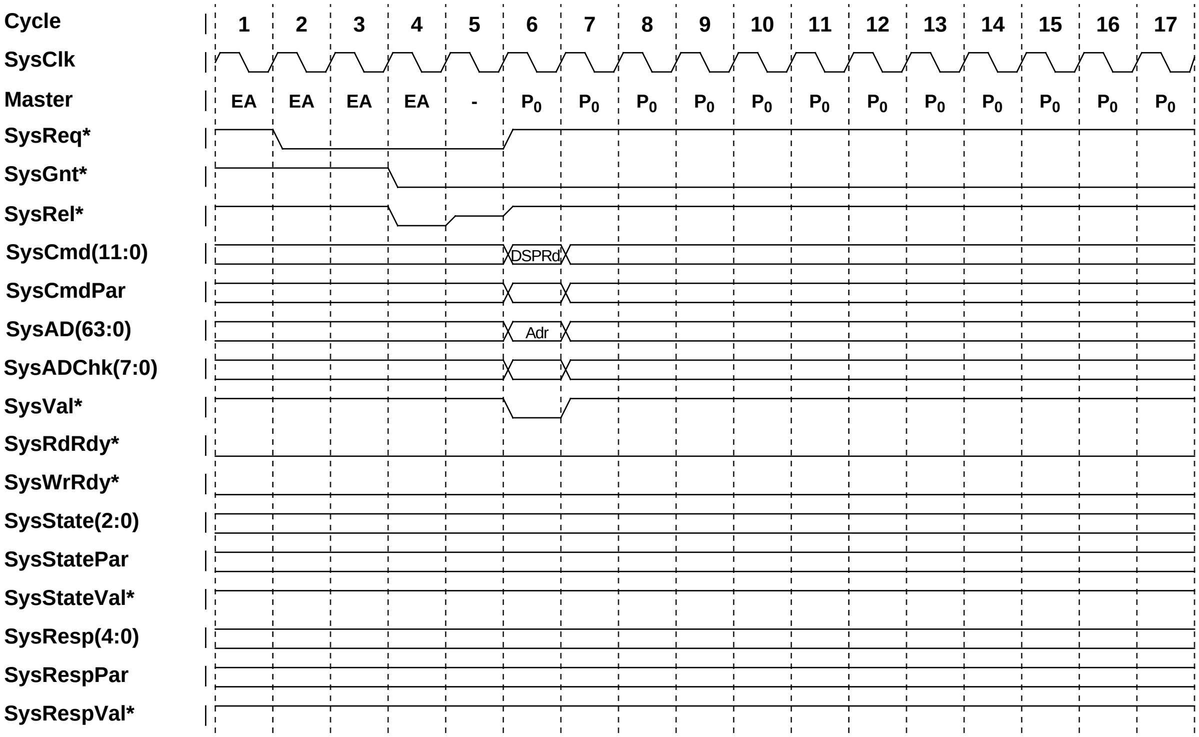

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
EA
EA
EA
EA
-
P_0
P_0
P_0
P_0
P_0
P_0
P_0
P_0
P_0
P_0
P_0
P_0
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
DSPRd
SysCmdPar
SysAD(63:0)
Adr
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*

Figure 6-11 Processor Double/Single/Partial-Word Read Request Protocol

Processor Block Write Request Protocol

A processor block write request results from the following:

- replacement of a *DirtyExclusive* secondary cache block due to a load, store, or prefetch secondary cache miss
- a CACHE Index WriteBack Invalidate (S) or Hit WriteBack Invalidate (S) instruction
- a completely gathered uncached accelerated block

As shown in Figure 6-12, the processor issues a processor block write request with a single address cycle followed by 8 or 16 data cycles.

The address cycle consists of the following:

- negating **SysCmd[11]**
- driving the block write command on **SysCmd[7:5]**
- driving the write cause indication on **SysCmd[4:3]**
- driving the target indication on **SysAD[63:60]**
- driving the physical address on **SysAD[39:0]**
- asserting **SysVal***

If the processor block write request results from the writeback of a secondary cache block, the *Dirty Exclusive* secondary cache block former state is driven on **SysAD[2:1**], the secondary cache block way is driven on **SysAD[57]** and **SysCmd[0]** is asserted.

If the processor block write request results from a completely gathered uncached accelerated block, the uncached attribute is driven on **SysAD[59:58]** and **SysCmd[0]** is negated.

Each data cycle consists of the following:

- asserting **SysCmd[11]**
- driving the data quality indication on **SysCmd[5]**
- driving the data type indication on **SysCmd[4:3]**
- driving the data on **SysAD[63:0]**
- asserting **SysVal***

The first 7 or 15 data cycles have a request data type indication, and the last data cycle has a request last data type indication.

The processor may negate **SysVal*** between data cycles of a processor block write request only if the **SCClk** frequency is less than half of the **SysClk** frequency.

The processor may only issue a processor block write request address cycle when the following are true:

- the System interface is in master state
- **SysWrRdy*** was asserted two **SysClk** cycles previously
- the processor is not the target of a conflicting outstanding external coherency request

Figure 6-12 depicts two adjacent processor block write requests. Since the System interface is initially in slave state, the processor must first assert **SysReq*** and then wait until the external agent relinquishes mastership of the System interface by asserting **SysGnt*** and **SysRel***.

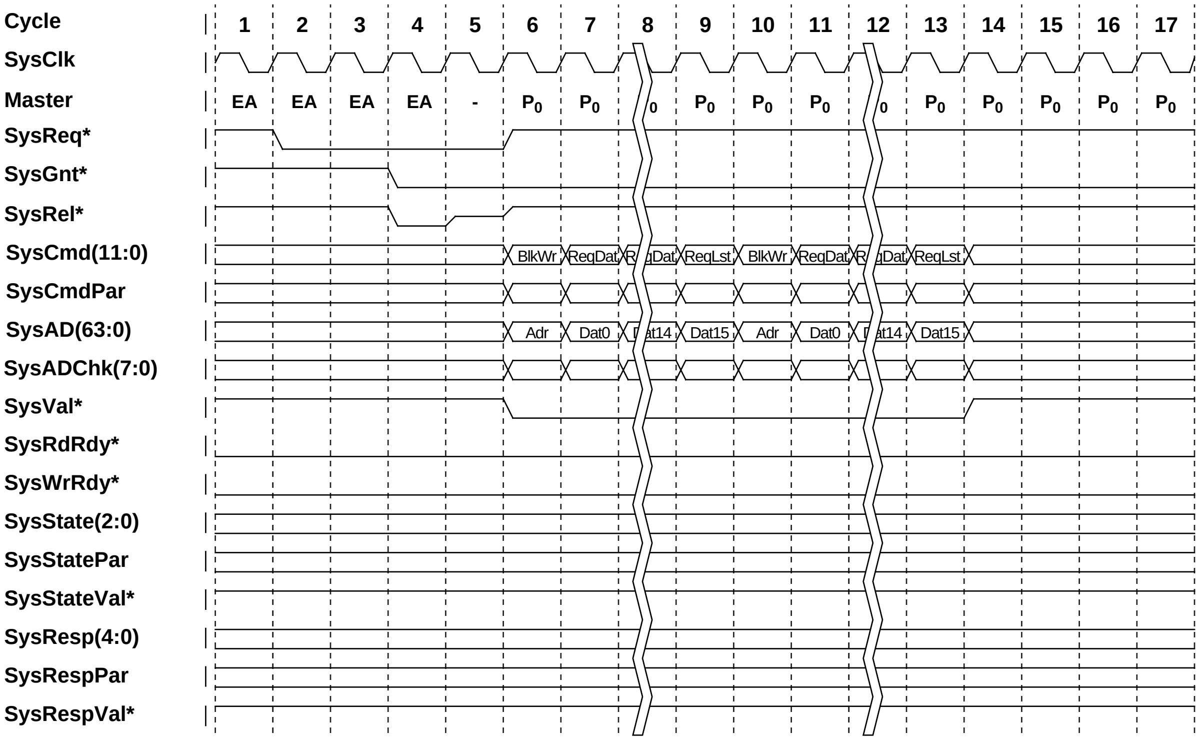

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
EA
EA
EA
EA
-
P0
P0
P0
P0
P0
P0
P0
P0
P0
P0
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
BlkWr
ReqDat
ReqLst
SysCmdPar
SysAD(63:0)
Adr
Dat0
Dat14
Dat15
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*

Figure 6-12 Processor Block Write Request Protocol

Processor Double/Single/Partial-Word Write Request Protocol

A processor double/single/partial-word write request results from an uncached store or incompletely gathered uncached accelerated block.

As shown in Figure 6-13, the processor issues a processor double/single/partial-word write request with a single address cycle immediately followed by a single data cycle.

The address cycle consists of the following:

- negating **SysCmd[11]**
- driving the double/single/partial-word write command on **SysCmd[7:5]**
- driving the write cause indication on **SysCmd[4:3]**
- driving the data size indication on **SysCmd[2:0]**
- driving the target indication on **SysAD[63:60]**
- driving the uncached attribute on **SysAD[59:58]**
- driving the physical address on **SysAD[39:0]**
- asserting **SysVal***

The data cycle consists of the following:

- asserting **SysCmd[11]**
- driving the request last data type indication on **SysCmd[4:3]**
- driving the write data on **SysAD[63:0]**
- asserting **SysVal***

The processor may only issue a processor double/single/partial-word write request address cycle when the System interface is in master state and **SysWrRdy*** was asserted two **SysClk** cycles previously.

Figure 6-13 depicts three processor double/single/partial write requests. Since the System interface is initially in slave state, the processor must first assert **SysReq*** and then wait until the external agent relinquishes mastership of the System interface by asserting **SysGnt*** and **SysRel***.

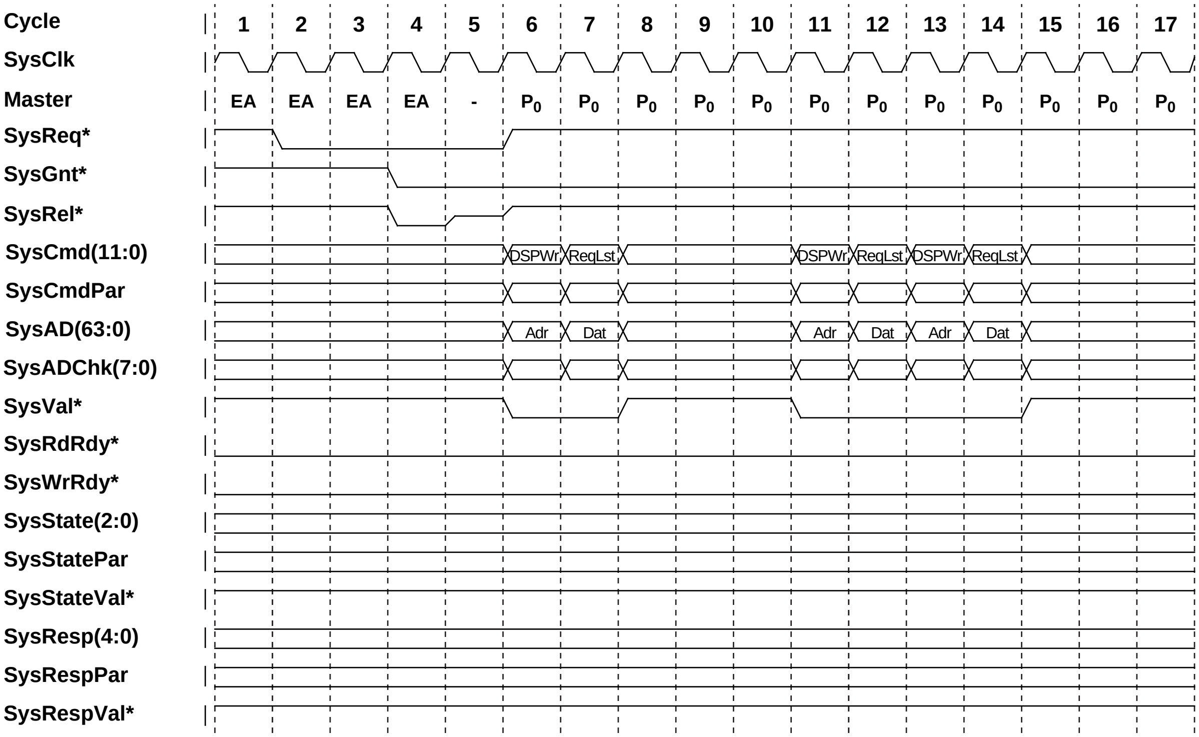

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
EA
EA
EA
EA
-
P0
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
DSPWr
ReqLst
DSPWr
ReqLst
DSPWr
ReqLst
SysCmdPar
SysAD(63:0)
Adr
Dat
Adr
Dat
Adr
Dat
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*

Figure 6-13 Processor Double/Single/Partial-Word Write Request Protocol

Processor Upgrade Request Protocol

A processor upgrade request results from a store or prefetch exclusive that hits a *Shared* block in the secondary cache.

As shown in Figure 6-14, the processor issues a processor upgrade request with a single address cycle. This address cycle consists of the following:

- negating **SysCmd[11]**
- driving a free request number on **SysCmd[10:8]**
- driving the upgrade command on **SysCmd[7:5]**
- driving the upgrade cause indication on **SysCmd[4:3]**
- driving the secondary cache block former state on **SysCmd[2:1]**
- asserting **SysCmd[0]**
- driving the target indication on **SysAD[63:60]**
- driving the secondary cache block way on **SysAD[57]**
- driving the physical address on **SysAD[39:0]**
- asserting **SysVal***

The processor may only issue a processor upgrade request address cycle when the following are true:

- the System interface is in master state
- **SysRdRdy*** was asserted two **SysClk** cycles previously
- the maximum number of outstanding processor requests specified by the **PrcReqMax** mode bits is not exceeded
- there is a free request number
- the processor is not the target of a conflicting outstanding external coherency request

A single processor may have as many as four processor upgrade requests outstanding on the System interface at any given time.

Figure 6-14 depicts four processor upgrade requests. Since the System interface is initially in slave state, the processor must first assert **SysReq*** and then wait until the external agent relinquishes mastership of the System interface by asserting **SysGnt*** and **SysRel***.

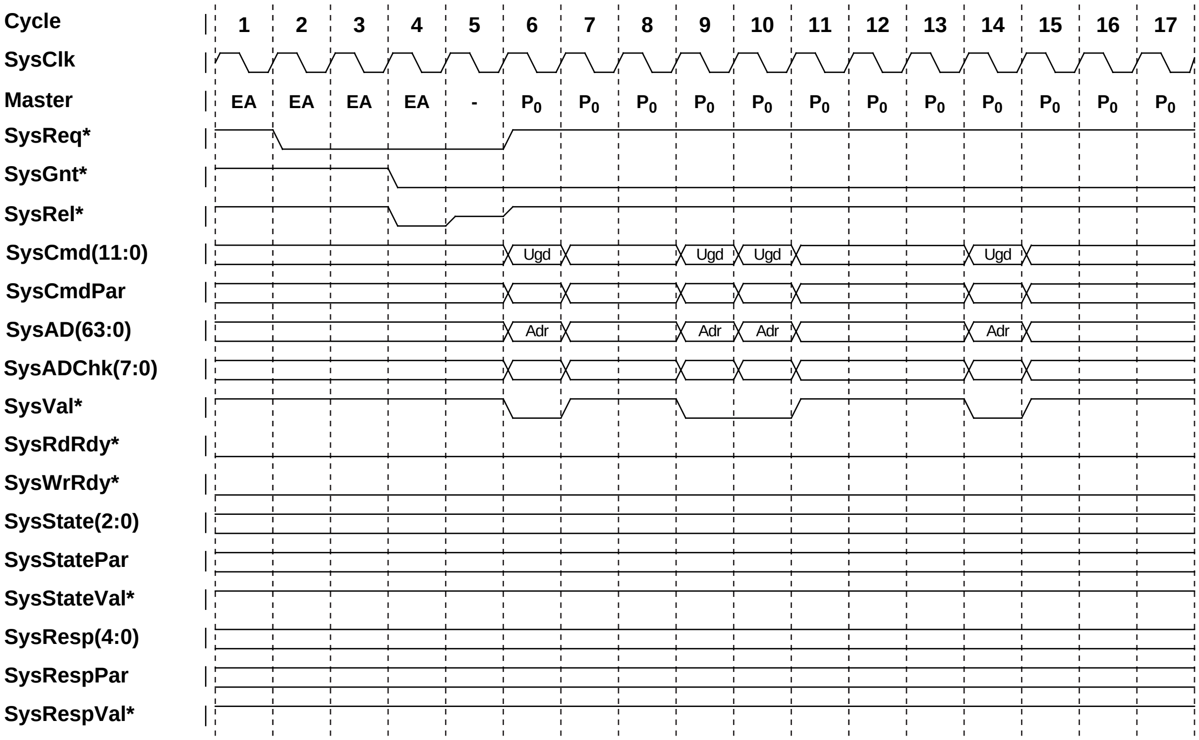

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
EA
EA
EA
EA
-
P0
P0
P0
P0
P0
P0
P0
P0
P0
P0
P0
P0
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
Ugd
Ugd
Ugd
Ugd
SysCmdPar
SysAD(63:0)
Adr
Adr
Adr
Adr
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*

Figure 6-14 Processor Upgrade Request Protocol

Processor Eliminate Request Protocol

A processor eliminate request results from the following:

- a cached instruction fetch, load, store, or prefetch that misses in the secondary cache and forces the replacement of a *Shared* or *CleanExclusive* secondary cache block
- a CACHE Index WriteBack Invalidate (S), Hit Invalidate (S), or Hit WriteBack Invalidate (S) instruction that forces the invalidation of a *Shared* or *CleanExclusive* secondary cache block
- a CACHE Hit Invalidate (S) instruction that forces the invalidation of a *DirtyExclusive* secondary cache block.

A processor eliminate request notifies the external agent that a *Shared*, *CleanExclusive*, or *DirtyExclusive* block has been eliminated from the secondary cache. Such requests are useful for systems implementing a directory-based coherency protocol, and are enabled by asserting the **PrcElmReq** mode bit.

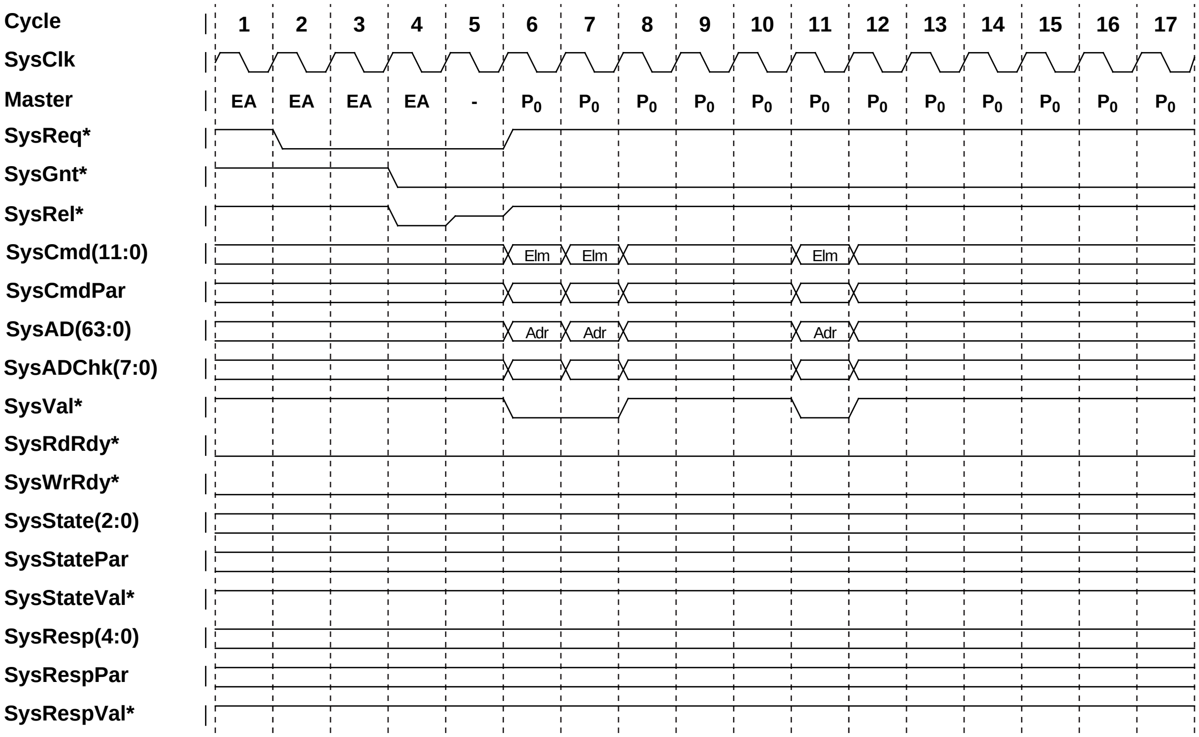
The processor issues a processor eliminate request with a single address cycle. This address cycle consists of the following:

- negating **SysCmd[11]**
- driving the special command on **SysCmd[7:5]**
- driving the eliminate special cause indication on **SysCmd[4:3]**
- driving the secondary cache block former state on **SysCmd[2:1]**
- asserting **SysCmd[0]**
- driving the target indication on **SysAD[63:60]**
- driving the secondary cache block way on **SysAD[57]**
- driving the physical address of the eliminated secondary cache block on **SysAD[39:0]**
- asserting **SysVal***

The processor may only issue a processor eliminate request address cycle when the following are true:

- the System interface is in master state
- **SysWrRdy*** was asserted two **SysClk** cycles previously
- the **PrcElmReq** mode bit is asserted
- the processor is not the target of a conflicting outstanding external coherency request

Figure 6-15 depicts three processor eliminate requests. Since the System interface is initially in slave state, the processor must first assert **SysReq*** and then wait until the external agent relinquishes mastership of the System interface by asserting **SysGnt*** and **SysRel***.



Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
EA
EA
EA
EA
-
P0
P0
P0
P0
P0
P0
P0
P0
P0
P0
P0
P0
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
Elm
Elm
Elm
SysCmdPar
SysAD(63:0)
Adr
Adr
Adr
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*

Figure 6-15 Processor Eliminate Request Protocol

Processor Request Flow Control Protocol

The processor provides the signals **SysRdRdy*** and **SysWrRdy*** to allow an external agent to control the flow of processor requests. **SysRdRdy*** controls the flow of processor read and upgrade requests whereas **SysWrRdy*** controls the flow of processor write and eliminate requests.

The processor can only issue a processor read or upgrade request address cycle to the System interface if **SysRdRdy*** was asserted two **SysClk** cycles previously. Similarly, the processor can only issue the address cycle of a processor write or eliminate request to the System interface if **SysWrRdy*** was asserted two **SysClk** cycles previously.

To determine the processor request buffering requirements for the external agent, note that the processor can issue any combination of processor requests in adjacent **SysClk** cycles. Also, since the System interface operates register-to-register with the external agent, a round trip delay of four **SysClk** cycles occurs between a processor request address cycle which prompts the external agent for flow control, and the flow control actually preventing any additional processor request address cycles from occurring. Consequently, if the maximum number of outstanding processor requests specified by the **PrcReqMax** mode bits is four, the external agent must be able to accept at least four processor read or upgrade requests. Also, the external agent must be able to accept at least four processor eliminate requests, two processor double/single/partial-word write requests, or one processor block write request.

Figure 6-16 depicts three processor double/single/partial-word write requests and four processor block read requests. After sensing the first processor double/single/partial-word write request, the external agent negates **SysWrRdy***. The external agent must have buffering sufficient for one additional processor write request before the flow control takes effect.

The external agent negates **SysRdRdy*** upon observing the first processor read request. The external agent must have buffering sufficient for three additional processor read requests before the flow control takes effect.

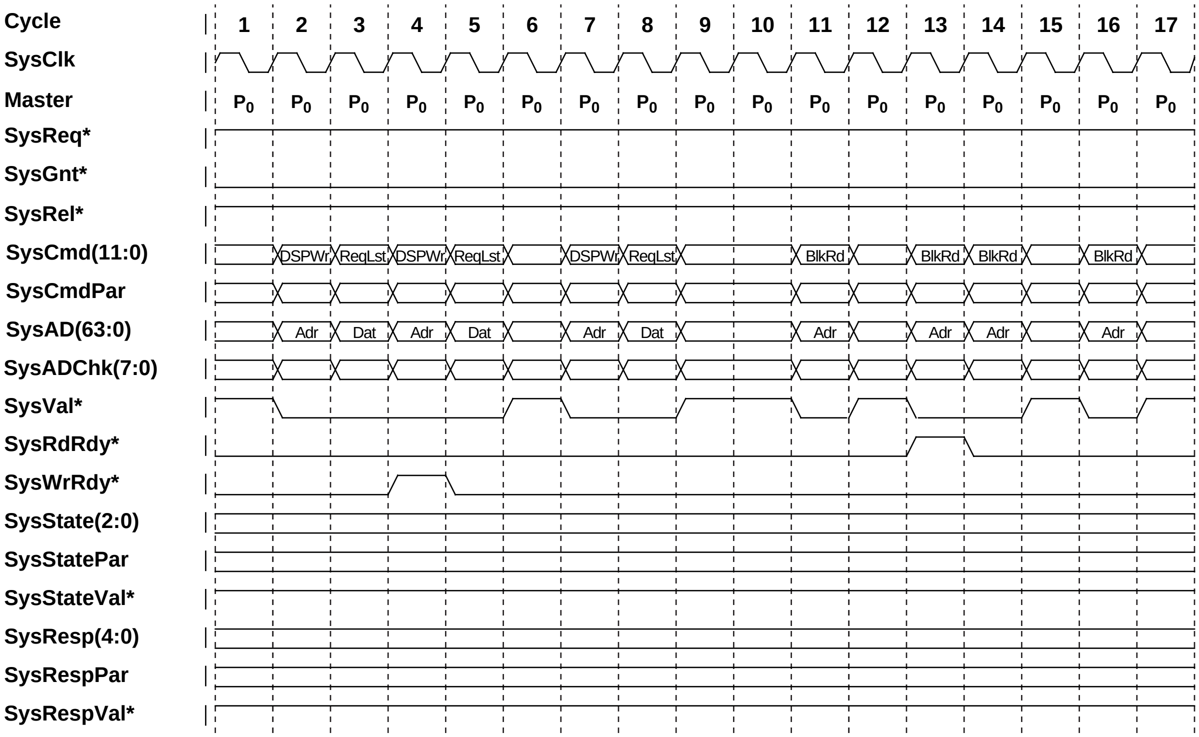

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
P0
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
DSPWr
ReqLst
DSPWr
ReqLst
DSPWr
ReqLst
BlkRd
BlkRd
BlkRd
BlkRd
SysCmdPar
SysAD(63:0)
Adr
Dat
Adr
Dat
Adr
Dat
Adr
Adr
Adr
Adr
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*

Figure 6-16 Processor Request Flow Control Protocol

External Response Protocol

The processor supports two classes of external responses:

- external data responses provide a double/single/partial-word of data or provide a block of data using the **SysAD[63:0]** bus
- external completion responses provide an acknowledge, error, or negative acknowledge indication using the **SysResp[4:0]** bus

An external agent may only issue an external data response to the processor when the System interface is in slave state. If the System interface is not already in slave state, the external agent must first negate **SysGnt*** and then wait for the processor to assert **SysRel***. If the System interface is already in slave state, the external agent may issue an external data response immediately.

External data responses may be accepted by the processor in adjacent **SysClk** cycles and in arbitrary order, relative to corresponding processor requests.

An external agent may issue an external completion response when the System interface is in either master or slave state. External completion responses may be accepted by the processor in adjacent **SysClk** cycles and in arbitrary order, relative to the corresponding processor requests.

External Block Data Response Protocol

An external agent may issue an external block data response in response to a processor block read or upgrade request.

An external agent issues an external block data response with 8 or 16 data cycles. Each data cycle consists of the following:

- asserting **SysCmd[11]**
- driving the request number associated with the corresponding processor request on **SysCmd[10:8]**
- driving the data quality indication on **SysCmd[5]**
- driving the data type indication on **SysCmd[4:3]**
- driving the cache block state on **SysCmd[2:1]**
- driving the ECC check indication on **SysCmd[0]**
- driving the data on **SysAD[63:0]**
- asserting **SysVal***

The first 7 or 15 data cycles have a response data type indication, and the last data cycle has a response last data type indication. The external agent may negate **SysVal*** between data cycles of an external block data response.

External block data response data must be supplied in subblock order, beginning with the quadword-aligned address specified by the corresponding processor request.

External block data responses for processor coherent block read shared or noncoherent block read requests may indicate a state of *Shared*, *CleanExclusive*, or *DirtyExclusive*. External block data responses for processor coherent block read exclusive or upgrade requests may indicate a state of *CleanExclusive* or *DirtyExclusive*.

Figure 6-17 depicts two processor block read requests and the corresponding external block data responses.

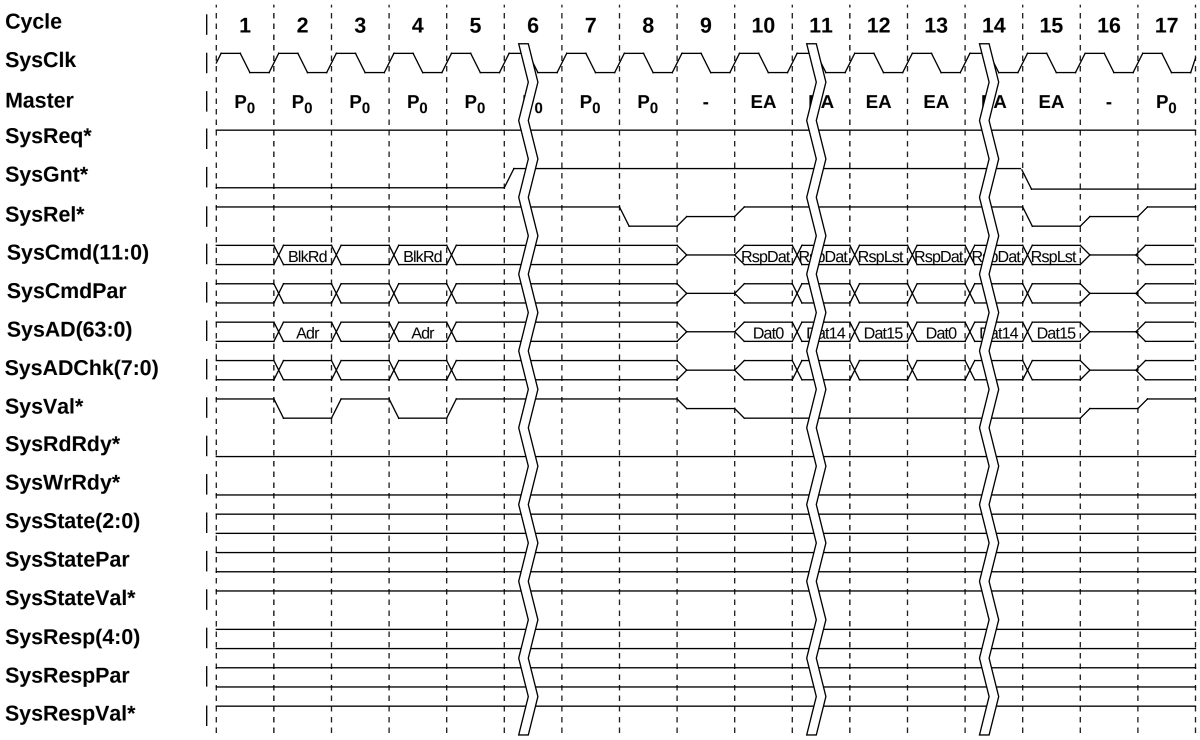

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
P0
P0
P0
P0
P0
0
P0
P0
-
EA
A
EA
EA
A
EA
-
P0
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
BlkRd
BlkRd
RspDat
RspLst
RspDat
RspLst
SysCmdPar
SysAD(63:0)
Adr
Adr
Dat0
Dat14
Dat15
Dat0
Dat14
Dat15
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*

Figure 6-17 External Block Data Response Protocol

External Double/Single/Partial-Word Data Response Protocol

An external agent may issue an external double/single/partial-word data response in response to a processor double/single/partial-word read request.

An external agent issues an external double/single/partial-word data response with a single data cycle; the data cycle consists of:

- asserting **SysCmd[11]**
- driving the request number associated with the corresponding processor request on **SysCmd[10:8]**
- driving the data quality indication on **SysCmd[5]**
- driving the response last data type indication on **SysCmd[4:3]**
- driving the ECC check indication on **SysCmd[0]**
- driving the data on **SysAD[63:0]**
- asserting **SysVal***

Figure 6-18 depicts a processor double/single/partial-word read request and the corresponding external double/single/partial-word data response.

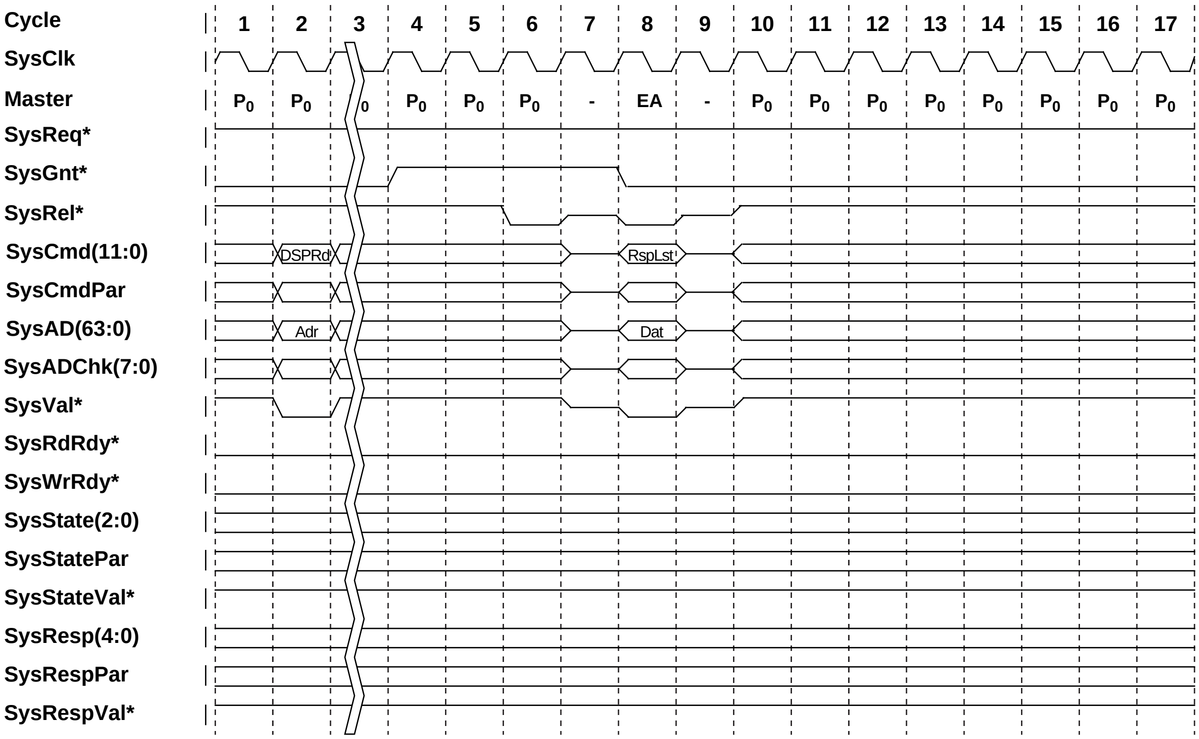

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
P0
P0
0
P0
P0
P0
-
EA
-
P0
P0
P0
P0
P0
P0
P0
P0
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
DSPRd
RspLst
SysCmdPar
SysAD(63:0)
Adr
Dat
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*

Figure 6-18 External Double/Single/Partial-Word Data Response Protocol

External Completion Response Protocol

An external agent issues an external completion response to provide an acknowledge, error, or negative acknowledge to an outstanding request, and to free the associated request number.

An external agent issues an external completion response by driving the response on **SysResp[4:0]** and asserting **SysRespVal*** for one **SysClk** cycle. **SysResp[4:2]** contains the request number associated with the corresponding outstanding request and **SysResp[1:0]** contains an acknowledge, error, or negative acknowledge indication, as described below:

- The external agent issues an external ACK completion response for a processor read or upgrade request to indicate that the request was successful. An external ACK completion response may only be issued for a processor read request if a corresponding external data response is coincidentally or previously issued.
- The external agent issues an external ERR completion response for a processor read or upgrade request to indicate that the request was unsuccessful. Upon receiving an external ERR completion response, the processor takes a Bus Error exception on the associated instruction. If the processor read or upgrade request was caused by a PREFETCH instruction, no exception is taken. Also, if the request was caused by a speculative instruction, no exception is taken.
- The external agent issues an external NACK completion response for a processor read or upgrade request to indicate that the request was not accepted. Upon receiving an external NACK completion response, the processor re-evaluates the associated instruction. Due to the speculative nature of the R10000 processor, the re-evaluation may or may not result in the reissue of a similar processor request.

An external ERR or NACK completion response issued in response to an external intervention, allocate request number, or invalidate has no affect on the processor except to free the request number.

Figure 6-19 depicts a processor upgrade request and a corresponding external completion response.

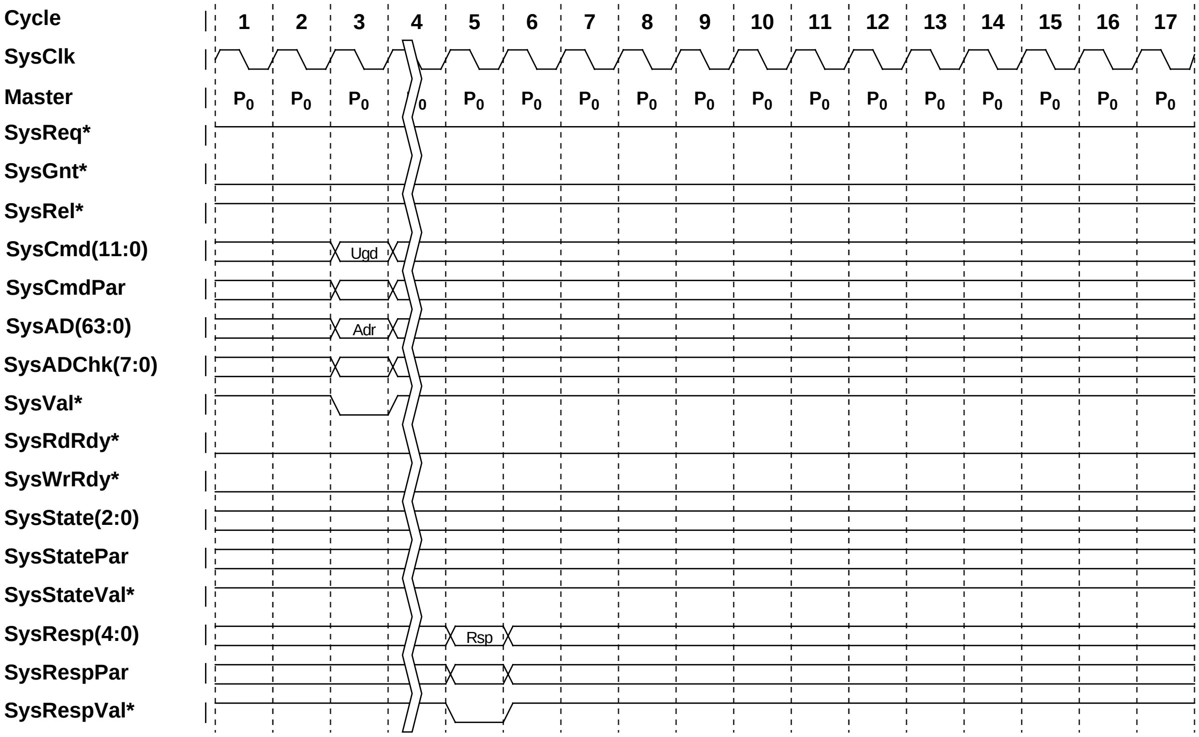

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
P0
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
Ugd
SysCmdPar
SysAD(63:0)
Adr
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
Rsp
SysRespPar
SysRespVal*

Figure 6-19 External Completion Response Protocol

External Request Protocol

An external agent issues an external request when it requires a resource within the processor. The external agent refers to any device attached to the processor system interface. It may be memory interface or cluster coordinator ASIC, or another processor residing on the cluster bus.

An external agent may only issue an external request to the processor when the System interface is in slave state. If the System interface is not already in slave state, the external agent must first negate **SysGnt*** and then wait for the processor to assert **SysRel***. If the System interface is already in slave state, the external agent may issue an external request immediately. The total number of outstanding external requests, including interventions, allocate request numbers, and invalidates, cannot exceed eight.

External requests may be accepted by the processor in adjacent **SysClk** cycles. External intervention and invalidate requests are considered external coherency requests.

External Intervention Request Protocol

An external agent issues an external intervention request to obtain a *Shared* or *Exclusive* copy of a secondary cache block.

An external agent issues an external intervention request with a single address cycle; this address cycle consists of the following:

- negating **SysCmd[11]**
- driving a request number on **SysCmd[10:8]**
- driving the intervention command on **SysCmd[7:5]**
- driving the ECC check indication on **SysCmd[0]**
- driving the target indication on **SysAD[63:60]**
- driving the physical address on **SysAD[39:0]**
- asserting **SysVal***

An external agent may only issue an external intervention request address cycle when the System interface is in slave state; typically a free request number is specified. An external agent may have as many as eight external intervention requests outstanding on the System interface at any given time.

Figure 6-20 depicts three external intervention requests. Since the System interface is initially in master state, the external agent must first negate **SysGnt*** and then wait until the processor relinquishes mastership of the System interface by asserting **SysRel***.

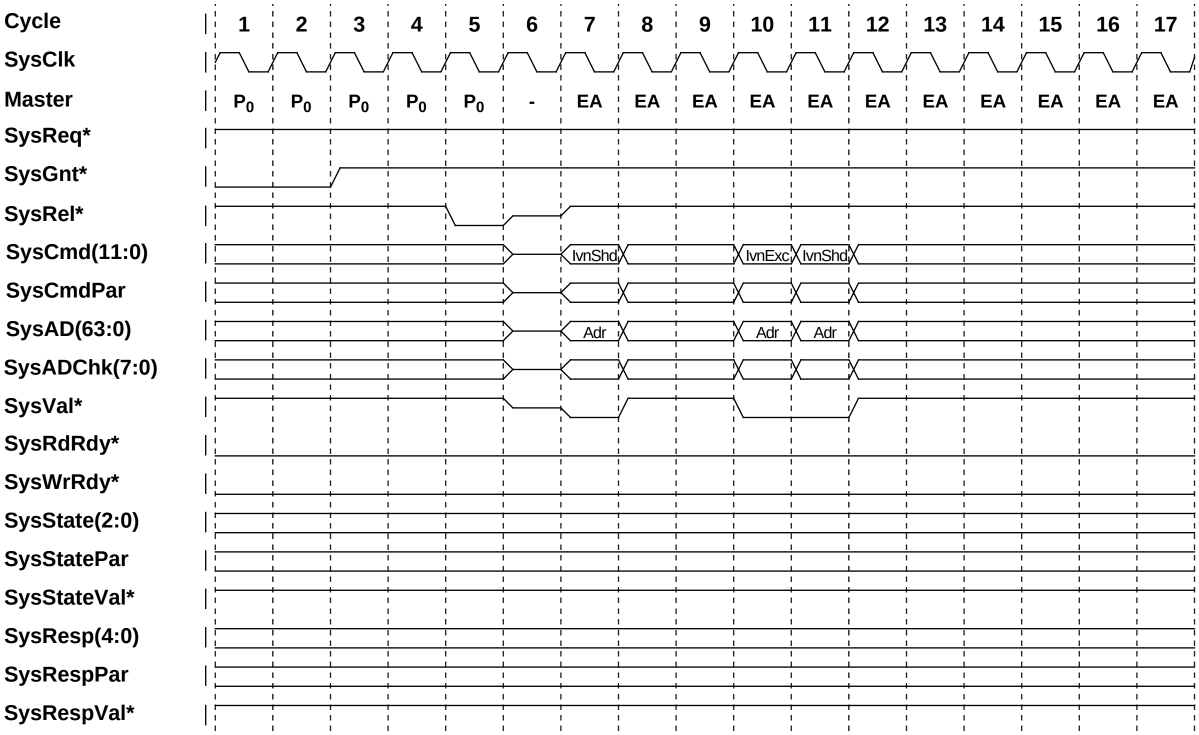

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
P0
P0
P0
P0
P0
-
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
IvnShd
IvnExc
IvnShd
SysCmdPar
SysAD(63:0)
Adr
Adr
Adr
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*

Figure 6-20 External Intervention Request Protocol

External Allocate Request Number Request Protocol

An external agent issues an external allocate request number request to reserve a request number for private use. Once allocated, the processor is prevented from using the request number until an external completion response for the request number is received.

An external agent issues an external allocate request number request with a single address cycle; this address cycle consists of the following:

- negating **SysCmd[11]**
- driving a free request number on **SysCmd[10:8]**
- driving the allocate request number command on **SysCmd[7:5]**
- asserting **SysVal***

An external agent may only issue an external allocate request number request address cycle when the System interface is in slave state and there is a free request number. The external agent may have as many as eight external allocate request number requests outstanding on the System interface at any given time.

Figure 6-21 depicts three external allocate request number requests. Since the System interface is initially in master state, the external agent must first negate **SysGnt*** and then wait until the processor relinquishes mastership of the System interface by asserting **SysRel***.

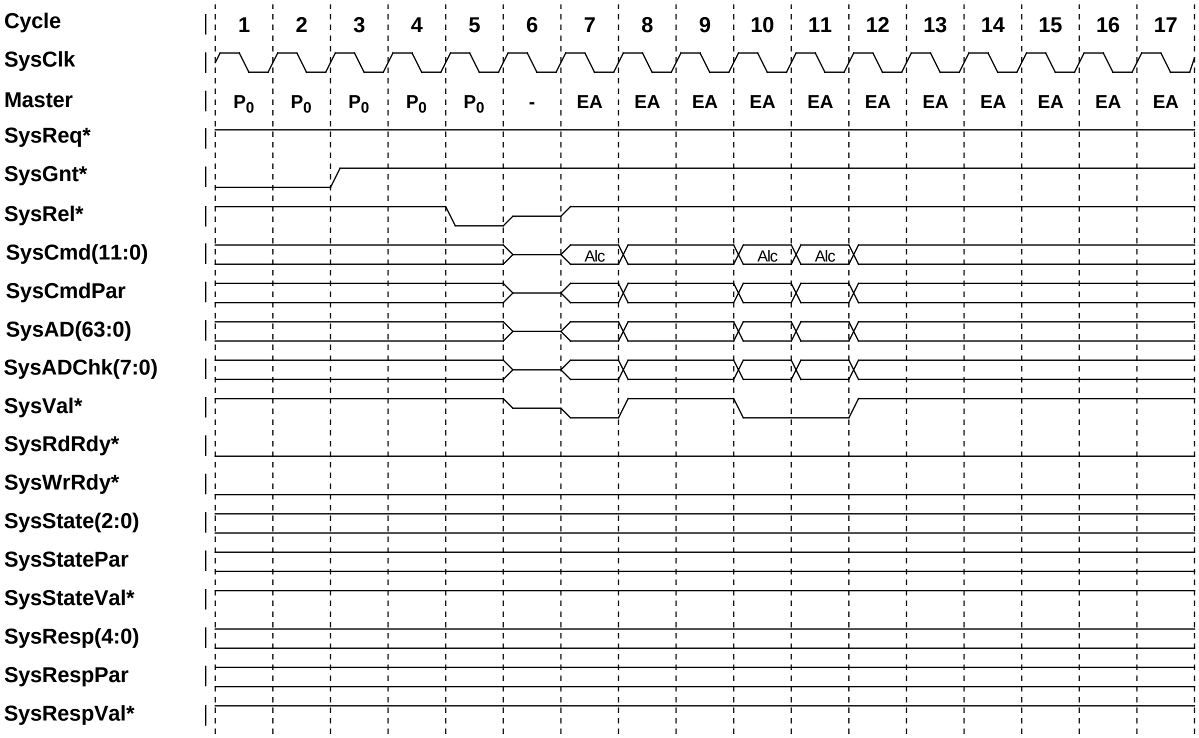

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
P0
P0
P0
P0
P0
-
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
Alc
Alc
Alc
SysCmdPar
SysAD(63:0)
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*

Figure 6-21 External Allocate Request Number Request Protocol

External Invalidate Request Protocol

An external agent issues an external invalidate request to invalidate a secondary cache block.

An external agent issues an external invalidate request with a single address cycle. This address cycle consists of the following:

- negating **SysCmd[11]**
- driving a request number on **SysCmd[10:8]**
- driving the invalidate command on **SysCmd[7:5]**
- driving the ECC check indication on **SysCmd[0]**
- driving the target indication on **SysAD[63:60]**
- driving the physical address on **SysAD[39:0]**
- asserting **SysVal***

An external agent may only issue an external invalidate request address cycle when the System interface is in slave state; typically a free request number is specified. An external agent may have as many as eight external invalidate requests outstanding on the System interface at any given time.

Figure 6-22 depicts three external invalidate requests. Since the System interface is initially in master state, the external agent must first negate **SysGnt*** and then wait until the processor relinquishes mastership of the System interface by asserting **SysRel***.

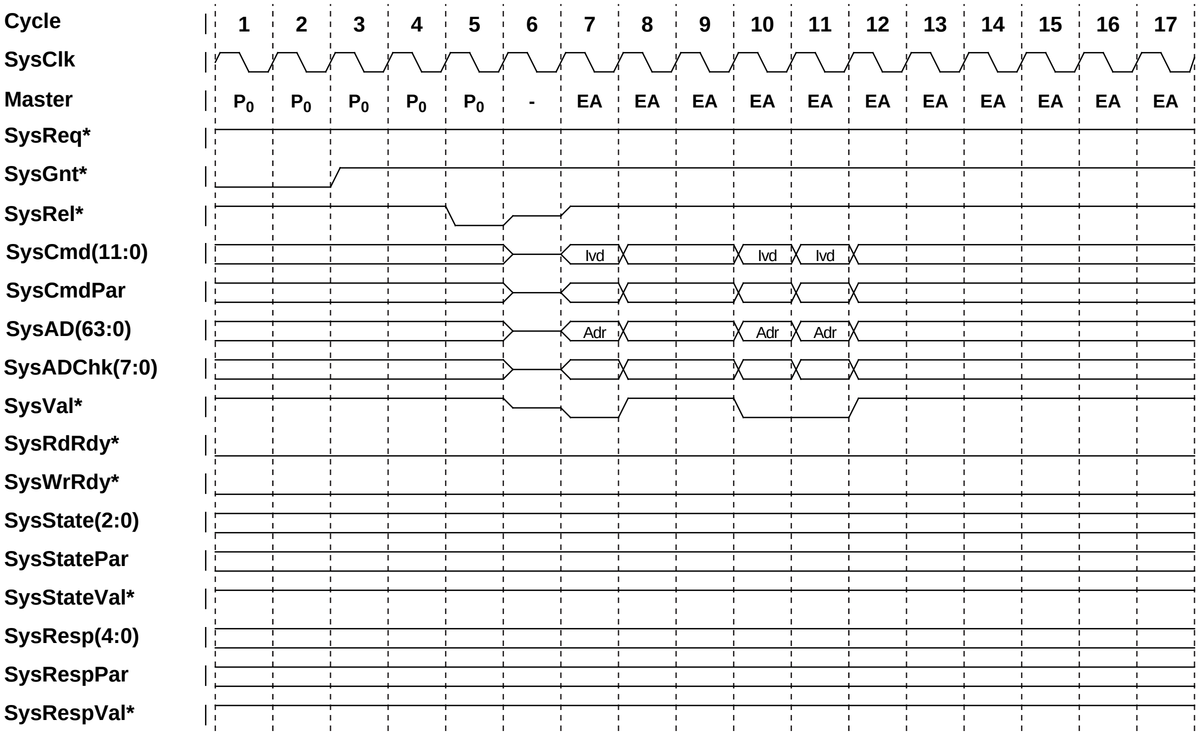

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
P0
P0
P0
P0
P0
-
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
Ivd
Ivd
Ivd
SysCmdPar
SysAD(63:0)
Adr
Adr
Adr
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*

Figure 6-22 External Invalidate Request Protocol

External Interrupt Request Protocol

An external agent issues an external interrupt request to interrupt the normal instruction flow of the processor.

An external agent issues an external interrupt request with a single address cycle. This address cycle consists of the following:

- negating **SysCmd[11]**
- driving the special command on **SysCmd[7:5]**
- driving the interrupt special cause indication on **SysCmd[4:3]**
- driving the ECC check indication on **SysCmd[0]**
- driving the target indication on **SysAD[63:60]**
- driving the *Interrupt* register write enables on **SysAD[20:16]**
- driving the *Interrupt* register values on **SysAD[4:0]**
- asserting **SysVal***

An external agent may only issue an external interrupt request address cycle when the System interface is in slave state.

Figure 6-23 depicts three external interrupt requests. Since the System interface is initially in master state, the external agent must first negate **SysGnt*** and then wait until the processor relinquishes mastership of the System interface by asserting **SysRel***.

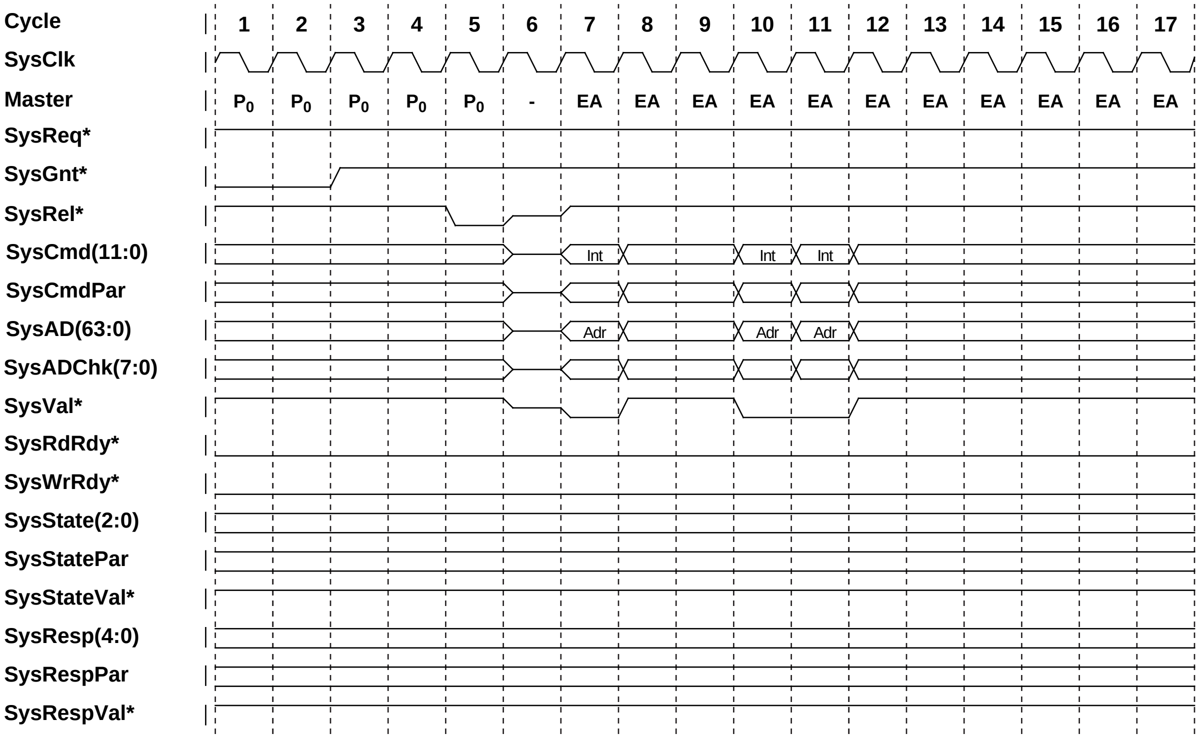

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
P_0
P_0
P_0
P_0
P_0
-
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
Int
Int
Int
SysCmdPar
SysAD(63:0)
Adr
Adr
Adr
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*

Figure 6-23 External Interrupt Request Protocol

Processor Response Protocol

Processor responses are supplied by the processor in response to external coherency requests that target the processor. The R10000 processor issues a processor coherency state response for each external coherency request that targets the processor. The processor issues a processor coherency data response for each external intervention request that targets the processor and hits a *DirtyExclusive* secondary cache block.

Processor coherency state responses are issued by the processor in the same order that the corresponding external coherency requests are received. Processor coherency state and data responses may occur in adjacent **SysClk** cycles.

Processor Coherency State Response Protocol

A processor coherency state response results from an external coherency request that targets the processor.

The processor issues a processor coherency state response by driving the secondary cache block tag quality indication on **SysState[2]**, driving the secondary cache block former state on **SysState[1:0]**, and asserting **SysStateVal*** for one **SysClk** cycle. The processor coherency state responses are issued in an order designated by the external coherency requests and will always be issued before an associated processor coherency data response. Note that processor coherency state responses can be pipelined ahead of the associated processor coherency data responses, and processor coherency data responses can be returned out-of-order. These cases typically arise from external coherency requests hitting outgoing buffer entries.

Figure 6-24 depicts two external coherency requests and the resulting processor coherency state responses.

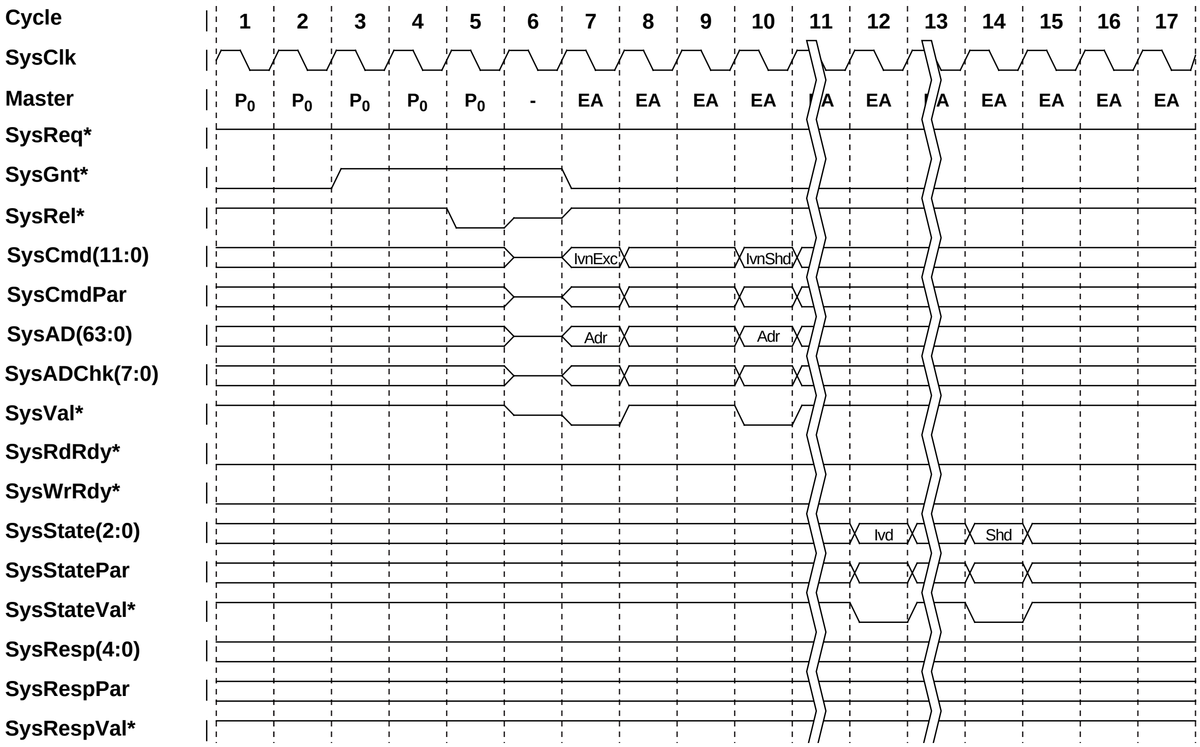

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
P_0
P_0
P_0
P_0
P_0
-
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
IvnExc
IvnShd
SysCmdPar
SysAD(63:0)
Adr
Adr
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
Ivd
Shd
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*

Figure 6-24 Processor Coherency State Response Protocol

Processor Coherency Data Response Protocol

A processor coherency data response results from an external intervention request that targets the processor and hits a *DirtyExclusive* secondary cache block.

The processor issues a processor coherency data response with a single empty cycle followed by either 8 or 16 data cycles. The empty cycle consists of negating **SysVal*** for a single **SysClk** cycle. The data cycles consist of the following:

- asserting **SysCmd[11]**
- driving the request number associated with the corresponding external coherency request on **SysCmd[10:8]**
- driving the data quality indication on **SysCmd[5]**
- driving the data type indication on **SysCmd[4:3]**
- driving the state of the cache block on **SysCmd[2:1]**
- asserting **SysCmd[0]**
- driving the data on **SysAD[63:0]**,
- asserting **SysVal***

The first 7 or 15 data cycles have a response data type indication, and the last data cycle has a response last data indication. The processor may negate **SysVal*** between data cycles of a processor coherency data response only if the **SCClk** frequency is less than half of the **SysClk** frequency.

The processor may only issue a processor coherency data response when the System interface is in master state and **SysWrRdy*** was asserted two **SysClk** cycles previously. Note that the empty cycle is considered the issue cycle for a processor coherency data response. If the System interface is not already in master state, the processor must first assert **SysReq***, and then wait for the external agent to relinquish mastership of the System interface bus by asserting **SysGnt*** and **SysRel***. If the System interface is already in master state, the processor may issue a processor coherency data response immediately.

When **SysStateVal*** is negated, **SysState[0]** provides the processor coherency data response indication. The processor asserts the processor coherency data response indication when there are one or more processor coherency data responses pending issue in the outgoing buffer. Once asserted, the indication is negated when the first doubleword of the last pending issue processor coherency data response is issued to the system interface bus. The processor coherency data response indication is not affected by **SysWrRdy***. However, as previously noted the processor may only issue a processor coherency data response when **SysWrRdy*** was asserted two **SysClk** cycles previously.

Processor coherency data response data is supplied in subblock order, beginning with the quadword-aligned address specified by the corresponding external coherency request. Processor coherency data responses are not necessarily issued in the same order as the external coherency requests; however each processor coherency data response always follows the corresponding processor coherency state response. Note that more than one processor coherency state response may be pipelined ahead of the corresponding processor coherency data responses.

Figure 6-25 depicts one external coherency request and the resulting processor coherency state and data responses.

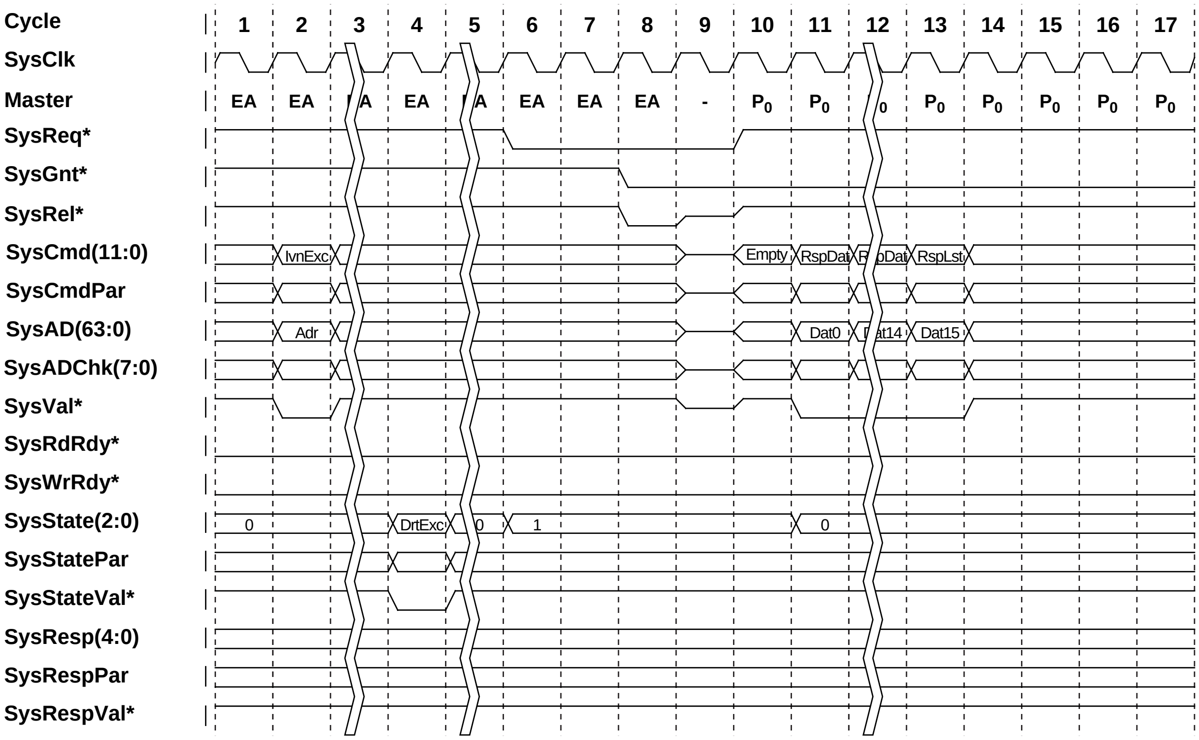

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
EA
EA
EA
EA
EA
EA
-
P0
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
IvnExc
Empty
RspDat
RspLst
SysCmdPar
SysAD(63:0)
Adr
Dat0
Dat14
Dat15
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
0
DrtExc
1
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*

Figure 6-25 Processor Coherency Data Response Protocol

6.18 System Interface Coherency

The System interface supports external intervention shared, intervention exclusive, and invalidate coherency requests. These requests are used by an external agent or other R10000 processors on the cluster bus to maintain cache coherency.

Each external coherency request that targets an R10000 results in a processor coherency state response. Additionally, each external intervention request that targets the R10000 and hits a *DirtyExclusive* secondary cache block results in a processor coherency data response.

External coherency requests and the corresponding processor coherency state responses are handled in FIFO order.

External Intervention Shared Request

An external intervention shared request is used by an external agent to obtain a *Shared* copy of a cache block. If the desired block resides in the processor cache, it is marked *Shared*.

If the secondary cache block's former state was *DirtyExclusive*, the processor issues a processor coherency data response.

External Intervention Exclusive Request

An external intervention exclusive request is used by an external agent to obtain an *Exclusive* copy of a cache block. If the desired block resides in the processor cache, it is marked *Invalid*.

If the secondary cache block's former state was *DirtyExclusive*, the processor issues a processor coherency data response.

External Invalidate Request

An external invalidate request is used by an external agent to invalidate a cache block. If the desired block resides in the processor cache, it is marked *Invalid*.

Under normal circumstances, the secondary cache block former state should not be *CleanExclusive* or *DirtyExclusive*.

External Coherency Request Action

Table 6-27 indicates the action taken for external coherency requests that target the processor.

Table 6-27 Action Taken for External Coherency Requests that Target the R10000 Processor[†]

Secondary Cache Block Former State	**Type of External Request**	**Secondary Cache Block New State**	**Processor Coherency State Response SysState[1:0]**	**Processor Coherency Data Response Required?**	**Processor Coherency Data Response State SysCmd[2:1]**
Invalid	Intervention shared	*Invalid*	0	No	N/A
	Intervention exclusive	*Invalid*	0	No	N/A
	Invalidate	*Invalid*	0	No	N/A
Shared	Intervention shared	*Shared*	1	No	N/A
	Intervention exclusive	*Invalid*	1	No	N/A
	Invalidate	*Invalid*	1	No	N/A
CleanExclusive	Intervention shared	*Shared*	2	No	N/A
	Intervention exclusive	*Invalid*	2	No	N/A
	Invalidate[‡]	*Invalid*	2	No	N/A
★ DirtyExclusive	Intervention shared[*]	*Shared*	3	Yes	*Shared*
	Intervention exclusive[*]	*Invalid*	3	Yes	*DirtyExclusive*
	Invalidate	*Invalid*	3	No	N/A

[‡] This should not occur under normal circumstances.

[*] The processor coherency data response must be written back to memory.

[†] These actions are taken in cases where there are no internal coherency conflicts. For exceptions due to internal coherency conflicts, please refer to Table 6-28.

Coherency Conflicts

Coherency conflicts arise when a processor request and an external request target the same secondary cache block. Coherency conflicts may be categorized as either internal or external, and are described in this section.

Internal Coherency Conflicts

A processor request is considered to be **pending issue** when it is buffered in the processor and has not yet been issued to the System interface bus. Internal coherency conflicts occur when the processor has a processor request pending issue and a conflicting external coherency request is received. Internal coherency conflicts are unavoidable and cannot be anticipated by the external agent since it cannot anticipate when the processor will have processor requests pending issue.

Table 6-28 describes the manner in which the processor resolves internal coherency conflicts.

Table 6-28 Internal Coherency Conflict Resolution

Processor Request Pending Issue	Conflicting External Coherency Request	Resolution
Coherent block read	Intervention shared	The processor allows the conflicting external coherency request to proceed and provides an *Invalid* processor coherency state response. The processor stalls the processor coherent block read request until the conflicting external coherency request has received an external completion response.
	Intervention exclusive	
	Invalidate	
Upgrade	Intervention shared	The processor allows the conflicting external coherency request to proceed and provides a *Shared* processor coherency state response. Once the conflicting external coherency request has received an external completion response, the processor internally NACKs the processor upgrade request that is pending issue.
	Intervention exclusive	
	Invalidate	
Block write	Intervention shared	The processor provides a *DirtyExclusive* processor coherency state response and changes the processor block write request that is pending issue into a *DirtyExclusive* processor coherency data response.
	Intervention exclusive	
	Invalidate	The processor provides a *DirtyExclusive* processor coherency state response and deletes the processor block write request that is pending issue.
Eliminate	Intervention shared	The processor provides a *Shared* or *CleanExclusive* processor coherency state response and deletes the processor eliminate request that is pending issue.[‡]
	Intervention exclusive	
	Invalidate	

[‡] If the processor eliminate request that is pending issue has a *DirtyExclusive* state, a *CleanExclusive* processor coherency state response is provided.

External Coherency Conflicts

A processor request is considered to be **pending response** when it has been issued to the System interface bus but has not yet received an external data or completion response. External coherency conflicts occur when the processor has a processor request that is pending response and a conflicting external coherency request is received. The processor relies on the external agent to detect and resolve external coherency conflicts. If the external agent chooses to issue an external coherency request to the processor which causes an external coherency conflict, the external coherency request must be completed before an external response is given to the conflicting processor request.

External coherency conflicts may be avoided if the point of coherence is the processor System interface bus and only one request is allowed to be outstanding for any given secondary cache block. However, in some system designs external coherency conflicts are unavoidable.

Processor block write and eliminate requests are never pending response, and therefore cannot cause external coherency conflicts.

Table 6-29 describes the manner in which the external agent resolves external coherency conflicts.

Table 6-29 External Coherency Conflict Resolution

Processor Requests that are Pending Response	Conflicting External Coherency Request	Resolution
Coherent block read	Intervention shared	The external agent responds to the external coherency requestor that the block is *Invalid*. At some later time, the external agent supplies an external response to the processor coherent block read request that is pending response.‡
	Intervention exclusive	
	Invalidate	
Upgrade	Intervention shared	The external agent responds to the external coherency requestor that the block is *Shared*. At some later time, the external agent supplies an external response to the processor upgrade request that is pending response.*
	Intervention exclusive	The external agent issues the conflicting external coherency request to the processor. The processor allows the conflicting external coherency request to proceed and supplies a *Shared* processor coherency state response. After observing the processor coherency state response, the external agent provides an external ACK completion response for the conflicting external coherency request. At some later time, the external agent supplies an external response for the processor upgrade request that is pending response. This external response may not be an external ACK completion response unless it is associated with an external block data response.
	Invalidate	

‡ Although it is not required, the external agent may choose to issue the conflicting external coherency request to R10000 and the processor will return an *invalid* processor coherency state response.

* Although it is not required, the external agent may choose to issue the conflicting external coherency request to R10000 and the processor will return a *shared* processor coherency state response.

External Coherency Request Latency

This section describes the R10000 external coherency request latency. Figure 6-26 depicts the following:

- an external coherency request which targets the processor
- the resulting processor coherency state response
- the potential processor coherency data response

Two external coherency request latency parameters are also defined:

- the processor coherency state response latency, t_{pcsr}, specifies the time from external coherency request to processor coherency state response
- the processor coherency data response latency, t_{pcdr}, specifies the time from the external coherency request to the processor coherency data response if a master, or to the assertion of the processor coherency data response indication on **SysState[0]** if a slave.

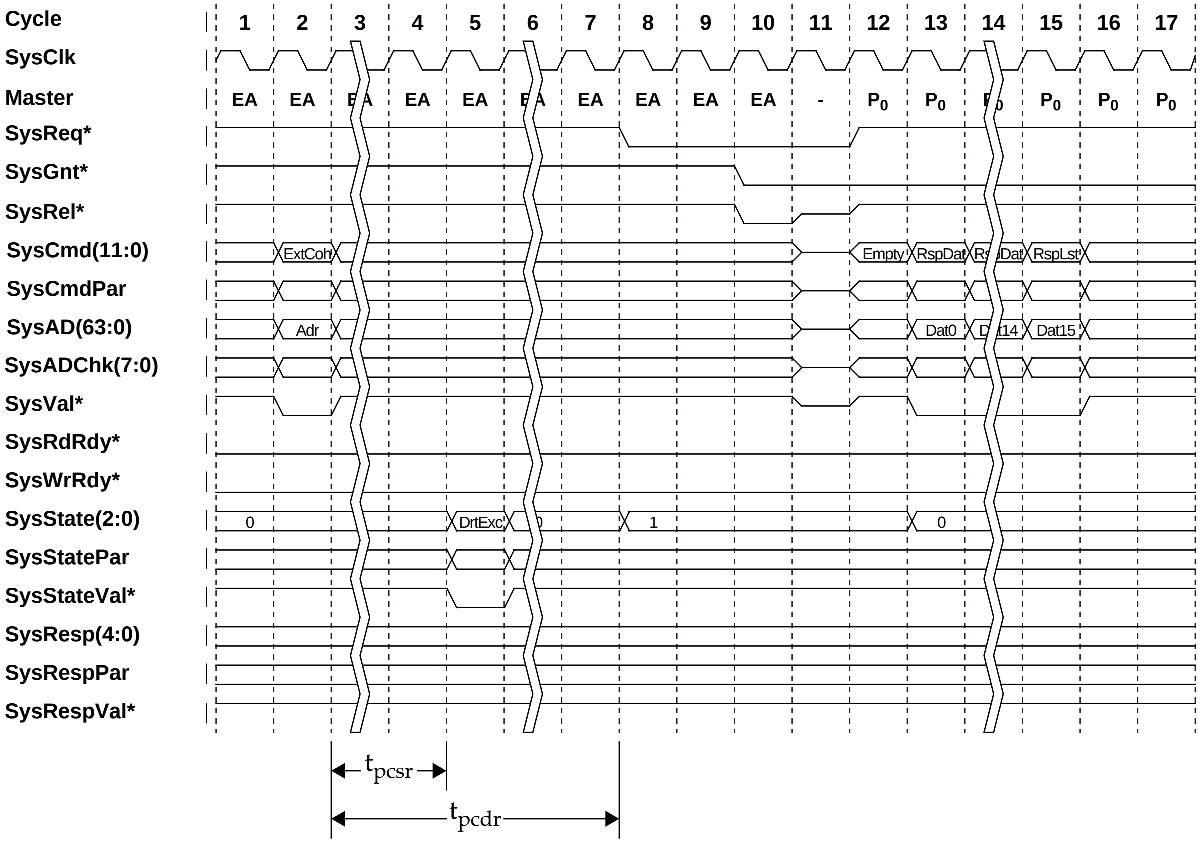

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
EA
-
P0
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
ExtCoh
Empty
RspDat
RspLst
SysCmdPar
SysAD(63:0)
Adr
Dat0
Dat14
Dat15
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
0
DrtExc
1
0
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*
tpcsr
tpcdr

Figure 6-26 External Coherency Request Latency Parameters

The external coherency request latency is presented in Table 6-30.

Table 6-30 External Coherency Request Latency

	Latency‡ (PClk cycles)					
	Processor Coherency State Response (t_{pcsr})			**Processor Coherency Data Response* (t_{pcdr})**		
SCClkDiv	**Min†**	**Typ‡‡**	**Max****	**Min††**	**Typ‡‡‡**	**Max*****
1	5	10	39	8	28	70
1.5	5	13	48	8	33	88
2	5	14	59	8	38	105
2.5	5	16	71	8	43	128
3	5	17	79	8	43	141

‡ This latency assumes no other previously issued external coherency requests are outstanding. 1 to 3 additional **PClk** cycles may be required for synchronization with **SysClk** depending on the **SysClkDiv** mode bits.

* This value assumes a 32-word secondary cache block size.

† This value assumes the external coherency request hits a cached or outgoing buffer entry.

‡‡ This value assumes the external coherency request does not hit a cached or outgoing buffer entry, the secondary cache is not busy, and the external coherency request hits in the MRU way of the secondary cache. If the external coherency request misses in the most-recently used (MRU) way of the secondary cache, 1 to 3 additional **PClk** cycles are required to query the LRU way of the secondary cache, depending on the **SCClkDiv** mode bits.

** This value assumes the external coherency request does not hit a cached or outgoing buffer entry, the secondary cache just commenced an index-conflicting CACHE Hit WriteBack Invalidate (S), and the external coherency request misses in the secondary cache MRU way.

†† This value assumes the external coherency request hits an outgoing buffer entry.

‡‡‡ This value assumes the external coherency request does not hit a cached or outgoing buffer entry, the secondary cache is not busy, the external coherency request hits in the MRU way of the secondary cache, no subset primary data cache blocks are inconsistent, and the external coherency request is secondary cache block-aligned. If the external coherency request misses in the MRU way of the secondary cache, 1 to 3 additional **PClk** cycles are required to query the LRU way of the secondary cache, depending on the **SCClkDiv** mode bits.

*** This value assumes the external coherency request does not hit a cached or outgoing buffer entry, the secondary cache just commenced an index-conflicting CACHE Hit WriteBack Invalidate (S), the external coherency request hits in the LRU way of the secondary cache, all subset primary data cache blocks are inconsistent, and the external coherency request is not secondary cache block-aligned.

SysGblPerf* Signal

The **SysGblPerf*** signal is provided for systems implementing a relaxed consistency memory model. The external agent asserts this signal when all processor requests are globally performed, thereby allowing the processor to graduate SYNC instructions. The external agent negates this signal when some processor requests are not yet globally performed, thereby preventing the processor from graduating SYNC instructions.

To prevent a SYNC instruction from graduating, the external agent must negate the **SysGblPerf*** signal no later than the same **SysClk** cycle in which it issued the external completion response for a processor read or upgrade request which is not yet globally performed. Also, the external agent must negate the **SysGblPerf*** signal no later than two **SysClk** cycles after the address cycle of a processor double/single/partial-word write request which has not yet been globally performed.

The **SysGblPerf*** signal may be permanently asserted in systems implementing a sequential consistency memory model.

6.19 Cluster Bus Operation

A R10000 multiprocessor cluster may be created by directly attaching the System interfaces of 2 to 4 R10000 processors, and providing an external cluster coordinator to handle arbitration and coherency management.

The cluster coordinator arbitrates the multiprocessors using the **SysReq***, **SysGnt***, and **SysRel*** signals.

A processor request issued by an R10000 processor in master state is observed as an external request by any R10000 processors in the slave state on the cluster bus. This is described Table 6-31.

Table 6-31 Relationship Between Processor and External Requests for the Cluster Bus

Processor Request	External Request
Coherent block read shared	Intervention shared
Coherent block read exclusive	Intervention exclusive
Noncoherent block read	Allocate request number
Double/single/partial-word read	Allocate request number
Block write	NOP
Double/single/partial-word write	NOP
Upgrade	Invalidate
Eliminate	NOP

In the same manner, a processor coherency data response issued by a processor in the master state is observed as an external block data response by any processors in the slave state.

External coherency requests that target a processor are handled in FIFO order and result in processor coherency state responses. If an external coherency request that targets a processor hits a *DirtyExclusive* secondary cache block, the processor also provides a processor coherency data response.

Figure 6-27 presents an example of a processor read request with four R10000 processors residing on the cluster bus. The **CohPrcReqTar** mode bit is asserted for a snoopy-based coherency protocol. $R10000_0$ issues a processor coherent read exclusive request. This is observed as an external intervention exclusive request by $R10000_1$, $R10000_2$, and $R10000_3$. $R10000_1$ and $R10000_3$ respond with *Invalid* processor coherency state responses. $R10000_2$ responds with a *DirtyExclusive* processor coherency state response. Based on these processor coherency state responses, the cluster coordinator allows $R10000_2$ to become master of the System interface so that it may provide a processor coherency data response, which will be observed as an external block data response by $R10000_0$. Finally, the cluster coordinator issues an external ACK completion response to forward the external block data response and to free the request number.

Figure 6-28 presents an example of a processor upgrade request with four R10000 processors residing on the cluster bus. The **CohPrcReqTar** mode bit is asserted for a snoopy-based coherency protocol. $R10000_0$ issues a processor upgrade request, observed as an external invalidate request by $R10000_1$, $R10000_2$, and $R10000_3$. $R10000_2$ and $R10000_3$ provide *Shared* processor coherency state responses. $R10000_1$ provides an *Invalid* processor coherency state response. Based on these processor coherency state responses, the cluster coordinator issues an external ACK completion response for the processor upgrade request to indicate that the request was successful and to free the request number.

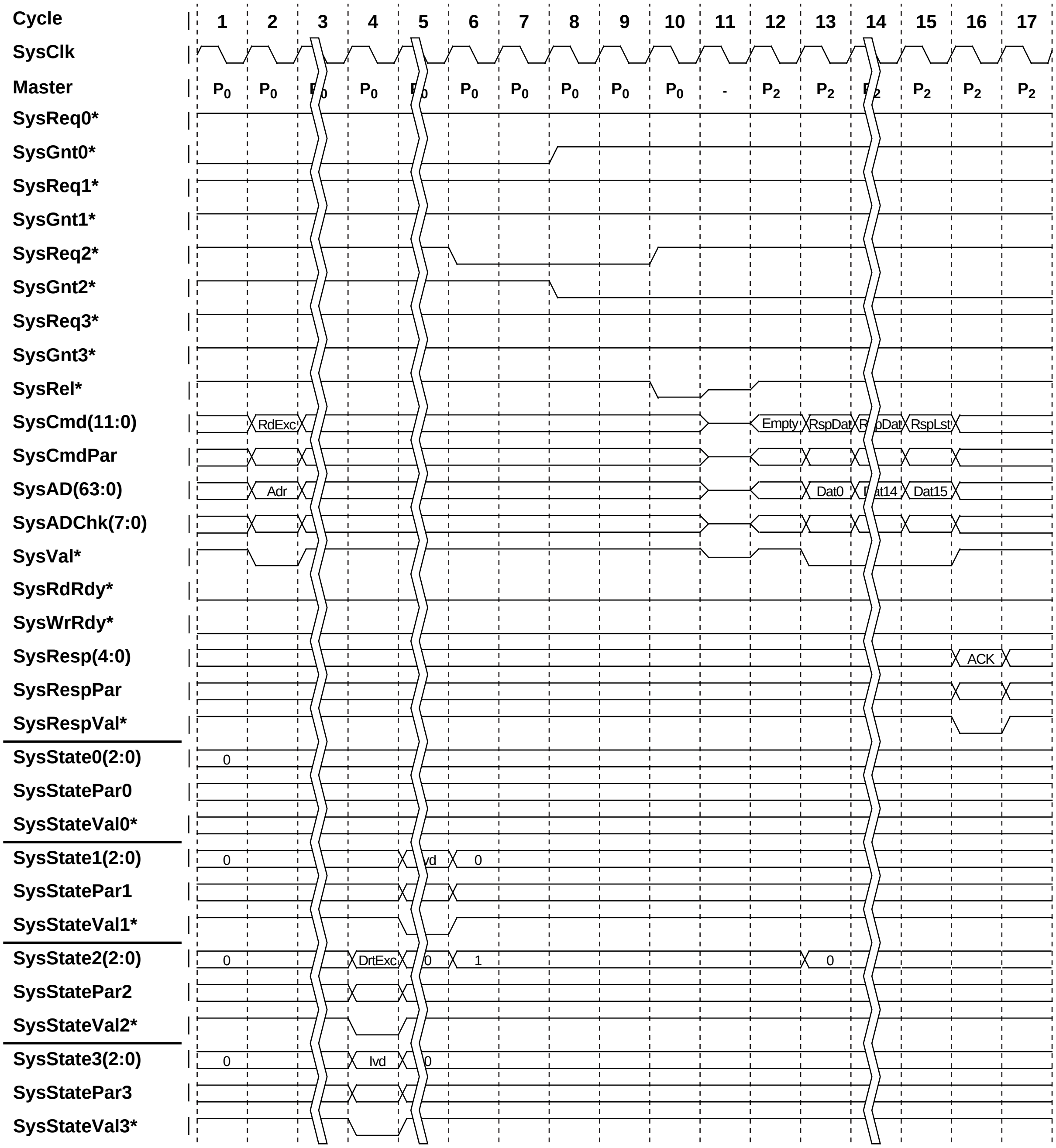

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
P_0
P_0
P_0
P_0
P_0
P_0
P_0
P_0
P_0
P_0
-
P_2
P_2
P_2
P_2
P_2
P_2
SysReq0*
SysGnt0*
SysReq1*
SysGnt1*
SysReq2*
SysGnt2*
SysReq3*
SysGnt3*
SysRel*
SysCmd(11:0)
RdExc
Empty
RspDat
RspDat
RspLst
SysCmdPar
SysAD(63:0)
Adr
Dat0
Dat14
Dat15
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysResp(4:0)
ACK
SysRespPar
SysRespVal*
SysState0(2:0)
0
SysStatePar0
SysStateVal0*
SysState1(2:0)
0
Ivd
0
SysStatePar1
SysStateVal1*
SysState2(2:0)
0
DrtExc
0
1
0
SysStatePar2
SysStateVal2*
SysState3(2:0)
0
Ivd
0
SysStatePar3
SysStateVal3*

Figure 6-27 R10000 Multiprocessor Cluster Processor Read Request Example

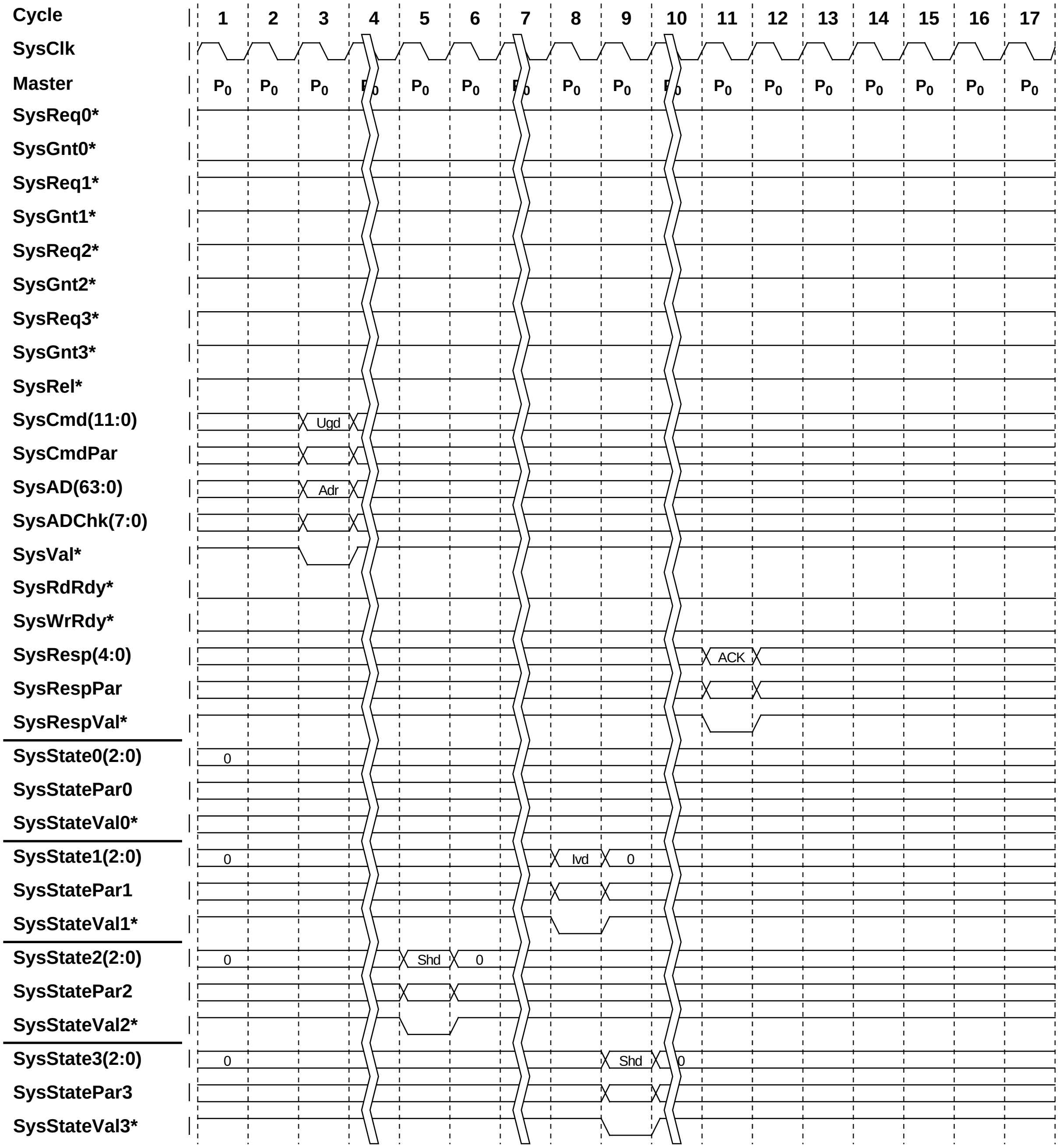

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
P0
SysReq0*
SysGnt0*
SysReq1*
SysGnt1*
SysReq2*
SysGnt2*
SysReq3*
SysGnt3*
SysRel*
SysCmd(11:0)
Ugd
SysCmdPar
SysAD(63:0)
Adr
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysResp(4:0)
ACK
SysRespPar
SysRespVal*
SysState0(2:0)
0
SysStatePar0
SysStateVal0*
SysState1(2:0)
0
Ivd
0
SysStatePar1
SysStateVal1*
SysState2(2:0)
0
Shd
0
SysStatePar2
SysStateVal2*
SysState3(2:0)
0
Shd
0
SysStatePar3
SysStateVal3*

Figure 6-28 R10000 Multiprocessor Cluster Processor Upgrade Request Example

6.20 Support for I/O

The processor assumes a memory-mapped I/O model. Consequentially, no special System interface encodings are provided, or required to designate I/O accesses. It is left to the programmer to ensure that I/O addresses have the appropriate TLB mappings.

The processor supports system designs utilizing hardware or software for coherent I/O. The external coherency requests are useful for creating systems with hardware I/O coherency, and the CACHE instruction is sufficient for creating a system with software I/O coherency.

6.21 Support for External Duplicate Tags

Some system designs implement an external duplicate copy of the secondary cache tags to reduce the coherency request latency and also filter out unnecessary external coherency requests made to the R10000 processor.

For such systems, it must be remembered that blocks may reside in either the secondary cache or in the outgoing buffer. During the address cycle of processor block read requests, the secondary cache block former state is provided. The external agent may use this information to maintain the external duplicate tags.

Typically, in a multiprocessor system using the cluster bus, the cluster coordinator specifies a free request number for an external coherency request. However, in a system using a duplicate-tag or directory-based coherency protocol, where the **CohPrcReqTar** mode bit is negated, the cluster coordinator may specify a busy request number for an external coherency request, providing each targeted R10000 processor has the request number busy due to an outstanding processor coherency request from another processor.

For example, suppose the processor in master state issues a processor coherent block read or upgrade request. The processors in slave state observe the processor request as an external coherency request that targets the external agent only, causing the associated request number to become busy. The cluster coordinator checks the duplicate tag or directory structure to determine if the block resides in the cache of one of the processors that was in slave state. If necessary, the cluster coordinator issues an external coherency request targeted at one or more of the processors that were in slave state. By using the same request number as the original processor request, this external coherency request does not consume a free request number, and allows a potential processor coherency data response to be supplied as an external block data response to the original processor request.

6.22 Support for a Directory-Based Coherency Protocol

Some system designs implement a directory-based coherency protocol.

For such systems, the processor provides the processor eliminate request cycle. If the **PrcElmReq** mode bit is asserted, the processor issues a processor eliminate request whenever it intends to eliminate a *Shared*, *CleanExclusive*, or *DirtyExclusive* block from the secondary cache. During the address cycle of the processor eliminate request, the physical address and the secondary cache block former state are provided. The external agent may then use this information to maintain an external directory structure.

6.23 Support for Uncached Attribute

The processor supports a 2-bit user-defined *Uncached Attribute*, which is driven on **SysAD[59:58]** during the address cycle of the following:

- processor double/single/partial-word read requests
- double/single/partial-word write requests
- block write requests resulting from completely gathered uncached accelerated blocks

For unmapped accesses, the uncached attribute is sourced from **VA[58:57]**.

For mapped accesses, the uncached attribute is sourced from the TLB *Uncached Attribute* field. The TLB *Uncached Attribute* field may be initialized in 64-bit mode using bits 63:62 of the CP0 *EntryLo0* and *EntryLo1* registers.

6.24 Support for Hardware Emulation

When using the R10000 processor in hardware emulation, it is desirable to operate the System interface at a relative low frequency (typically 1 MHz or below). Since the R10000 processor contains dynamic circuitry, an external agent cannot simply provide low frequency **SysClk**, so a **SysCyc*** input to the processor allows an external agent to define a virtual system clock, and yet supply a **SysClk** within the acceptable operating range. The assertion of **SysCyc*** in a particular **SysClk** cycle creates a virtual system clock pulse four **SysClk** cycles later. **SysCyc*** may be asserted aperiodically.

In a normal system environment, the **SysCyc*** input should be permanently asserted.

Figure 6-29 depicts the use of **SysCyc*** to create a virtual **SysClk** of one-third the normal **SysClk** frequency.

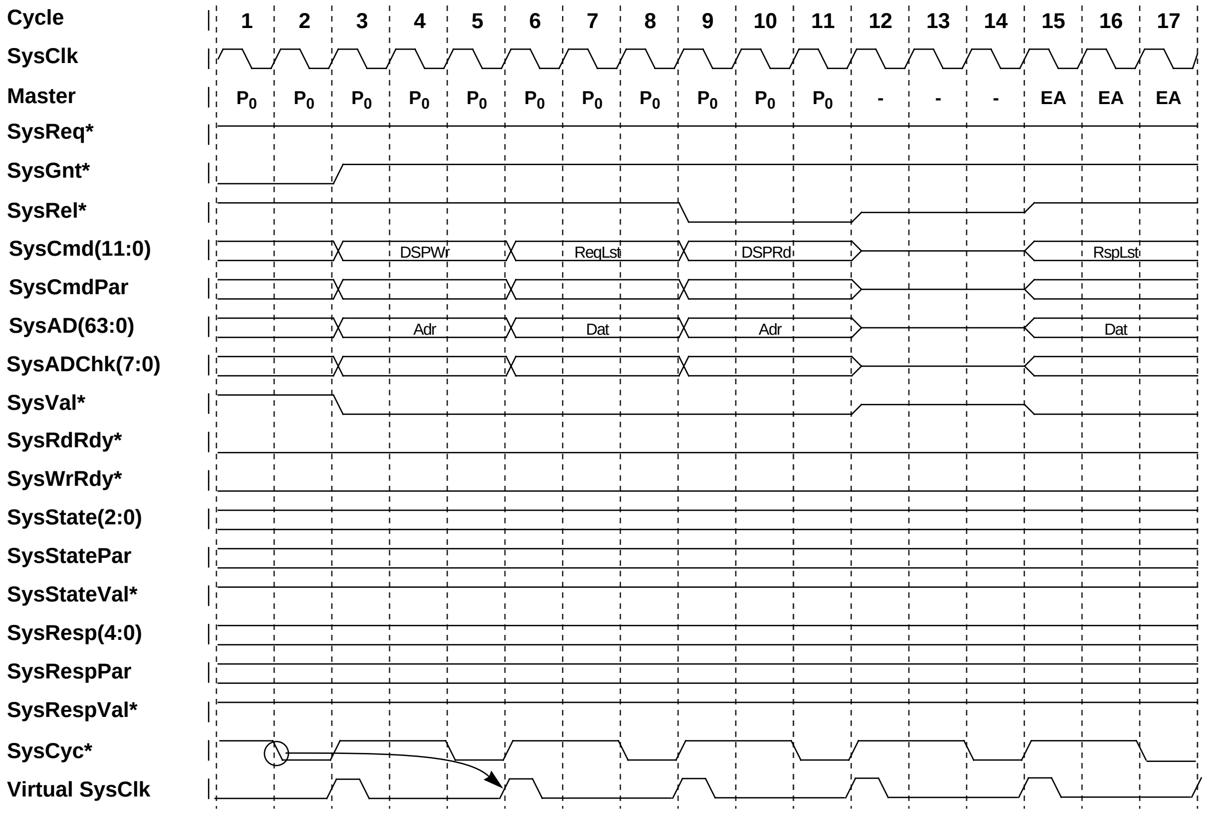

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
P0 P0 P0 P0 P0 P0 P0 P0 P0 P0 P0 - - - EA EA EA
SysReq*
SysGnt*
SysRel*
SysCmd(11:0)
DSPWr
ReqLst
DSPRd
RspLst
SysCmdPar
SysAD(63:0)
Adr
Dat
Adr
Dat
SysADChk(7:0)
SysVal*
SysRdRdy*
SysWrRdy*
SysState(2:0)
SysStatePar
SysStateVal*
SysResp(4:0)
SysRespPar
SysRespVal*
SysCyc*
Virtual SysClk

Figure 6-29 Hardware Emulation Protocol

7. Clock Signals

The R10000 processor has differential PECL clock inputs, **SysClk** and **SysClk***, from which all processor internal clock signals and secondary cache clock signals are derived.

Three major clock domains are in the processor:

- the **System interface clock domain**, which operates at the system clock frequency and controls the System interface signals
- the **internal processor clock domain**, which controls the processor core logic
- the **secondary cache clock domain**, which controls signals communicating with the external secondary cache synchronous SRAM

These domains are described in this chapter.

7.1 System Interface Clock and Internal Processor Clock Domains

In high performance systems, PECL-level differential clocks are routinely used to minimize system clock skews. The R10000 processor receives differential system clock signals at the **SysClk** and **SysClk*** pins; two additional pins, **SysClkRet** and **SysClkRet***, are the return paths for termination of these signals.

SysClk and **SysClk*** are used to drive an on-chip phase-locked loop (PLL), which multiplies the system clock to create an internal processor clock, **PClk**.

The R10000 processor always communicates with the system at the **SysClk** frequency, and **PClk** always runs at a frequency-multiple of **SysClk**, according to the following formula:

```
PClk = SysClk*(SysClkDiv+1)/2
```

For example, in a 50 MHz system with **SysClkDiv** = 7 and **SCClkDiv**=2,
PClk= 50*8/2 = 200 MHz.

> **NOTE:** It is preferred that the R10000 processor uses a differential PECL clock input. However, in a less-aggressive system, a CMOS/TTL single-ended clock can be used to drive the processor, provided its complementary clock input, **SysClk***, is tied to an appropriate reference voltage (1.4V for TTL, Vcc/2 for CMOS). In any case, the reference voltage applied to SysClk* should not be less than 1.2V.

7.2 Secondary Cache Clock

The processor uses registered synchronous SRAMs for its secondary cache, to allow pipelined accesses.

The processor provides 6 pairs of differential clock outputs, **SCClk(5:0)** and **SCClk*(5:0)**, to be used by the secondary cache synchronous SRAMs. These outputs swing between **VccQSC** and **Vss**. The **SCClkTap** mode bits (Mode bits are described in Chapter 8, the section titled “Mode Bits.”) specify the alignment of **SCClk(5:0)** and **SCClk*(5:0)** relative to the internal secondary cache clock. Note that the output buffer delay is not included.

The secondary cache interface clock is generated by dividing down the internal processor clock, **PClk**.

SCClk is related to **SysClk** according to the following formula:

```
SCClk = SysClk*(SysClkDiv+1)/(SCClkDiv+1)
```

For example, in a 50 MHz system with **SysClkDiv**=7 and **SCClkDiv**=2,
SCClk = 50*8/3 = 133 MHz.

7.3 Phase-Locked-Loop

The processor uses the internal PLL for clock generation and multiplication as shown in Figure 7-1.

Values of the termination resistors for the **SysClkRet/SysClkRet*** signals are system-dependent. The system designer must select a value based upon the characteristic impedance of the board, therefore it is beyond the scope of this manual to specify values for these termination resistors.

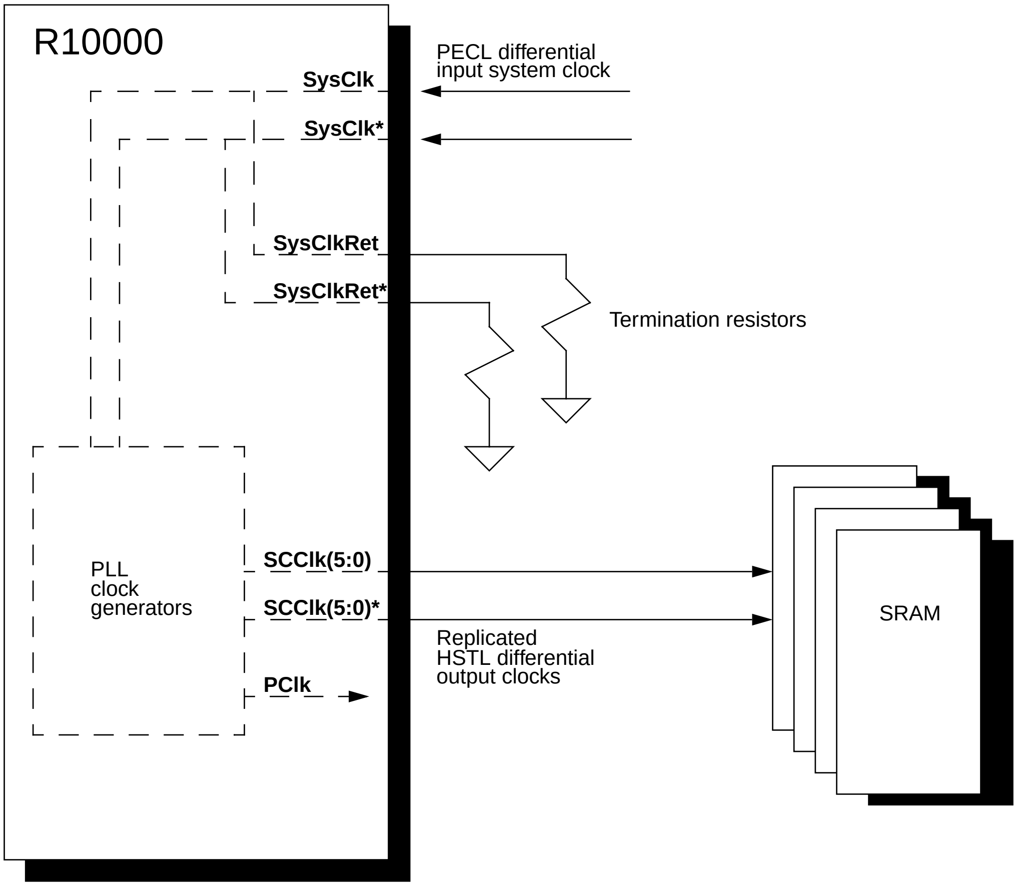

R10000
SysClk
SysClk*
SysClkRet
SysClkRet*
PECL differential input system clock
Termination resistors
PLL clock generators
SCClk(5:0)
SCClk(5:0)*
PClk
Replicated HSTL differential output clocks
SRAM

Figure 7-1 R10000 System and Secondary Cache Clock Interface

8. Initialization

This section describes initialization of the R10000 processor, including initialization of logical registers.

Initialization of the processor occurs during a reset sequence. The processor supports three separate reset sequences:

- Power-on reset
- Cold reset
- Soft reset

These sequences are described in this chapter.

Also described are the mode bits.

8.1 Initialization of Logical Registers

After a power-on or cold reset sequence, all logical registers (both in the integer and the floating-point register files) must be written before they can be read. Failure to write any of these registers before reading from them will have an unpredictable result.

★ **NOTE:** On the initialization of the FPU after a power-on or cold reset, a write for initialization of the busy-bit table can be performed by using MTC1 instruction with FR=1 (during the initialization only) or DMTC1 instruction.

8.2 Power-On Reset Sequence

The Power-on Reset sequence is used to reset the processor after the initial power-on, or whenever power or **SysClk** are interrupted.

The Power-on Reset sequence is as follows:

- The external agent negates **DCOk**.
- The external agent asserts **SysReset***.
- The external agent negates **SysGnt***.
- The external agent negates **SysRespVal***.
- Once **Vcc**, **VccQ[SC,Sys]**, **Vref[SC,Sys]**, **Vcc[Pa,Pd]**, and **SysClk** stabilize, the external agent waits at least 1ms and then asserts **DCOk**.
- At this time, the System interface resides in slave state and all internal state is initialized.
- The **SysClkDiv** mode bits default to divide-by-1.
- The **SCClkDiv** mode bits default to divide-by-3.
- After waiting at least 100 ms for the internal clocks to stabilize, the external agent loads the mode bits into the processor by driving the mode bits on **SysAD[63:0]**, waiting at least two **SysClk** cycles, and then asserting **SysGnt*** for at least one **SysClk** cycle.
- After waiting at least another 100 ms for the internal clocks to restabilize, the external agent synchronizes all clocks internal to the processor. This is performed by asserting **SysRespVal*** for one **SysClk** cycle.
- After waiting at least 100 ms for the internal clocks to again restabilize, (a third 100 ms restabilization period) the external agent negates **SysReset***.
- The external agent must retain mastership of the System interface, refrain from issuing external requests or nonmaskable interrupts, and ignore the system state bus until the processor asserts **SysReq***. The assertion of **SysReq*** indicates the processor is ready for operation. In a cluster arrangement, all processors must assert **SysReq***, indicating they are ready for operation.

NOTE: If the **virtual SysClk** is used during the reset sequence, the mode bits, **SysGnt***, **SysRespVal***, and **SysReset*** should all be referenced to the virtual **SysClk** that is created with **SysCyc***. This approach will cause the R10000 to come out of reset synchronously with the **virtual SysClk**, which will allow repeatable and lock-step operation (see Chapter 6, the section titled "Support for Hardware Emulation," for description of **virtual SysClk** operation).

During a Power-on Reset sequence, all internal state is initialized. A Power-on Reset sequence causes the processor to start with the Reset exception.

Figure 8-1 shows the Power-on Reset sequence.

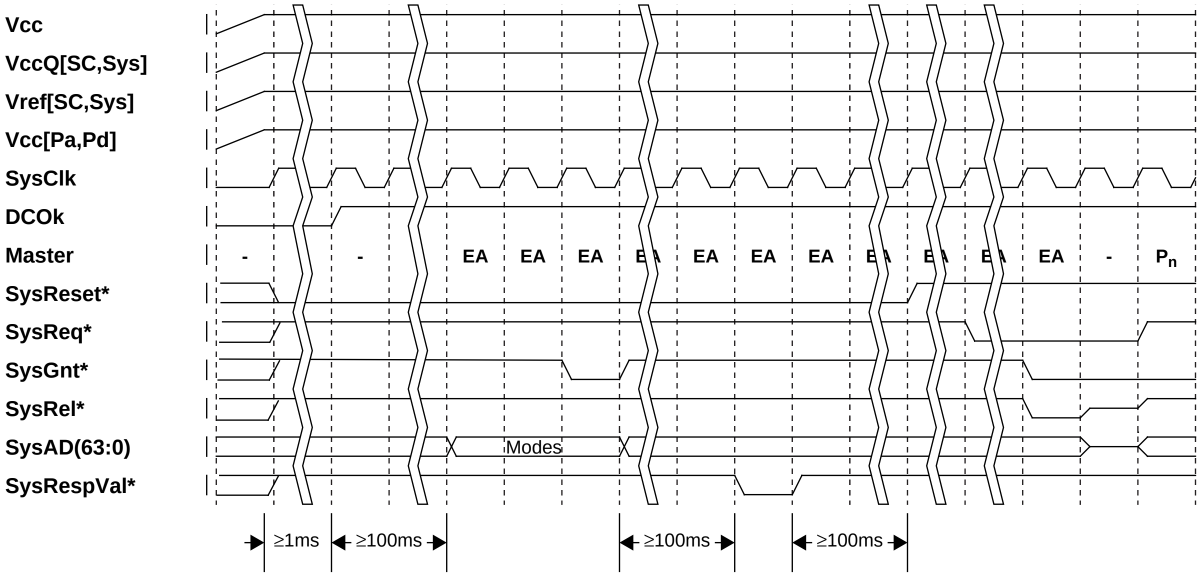

Vcc
VccQ[SC,Sys]
Vref[SC,Sys]
Vcc[Pa,Pd]
SysClk
DCOk
Master
SysReset*
SysReq*
SysGnt*
SysRel*
SysAD(63:0)
SysRespVal*
-
-
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
-
P_n
Modes
≥1ms
≥100ms
≥100ms
≥100ms

Figure 8-1 Power-On Reset Sequence

8.3 Cold Reset Sequence

The Cold Reset sequence is used to reset the entire processor, and possibly alter the mode bits while power and **SysClk** are stable.

The Cold Reset sequence is as follows:

- The external agent negates **SysGnt*** and **SysRespVal***.
- After waiting at least one **SysClk** cycle, the external agent asserts **SysReset***.
- After waiting at least 100 ms, the external agent loads the mode bits into R10000. This is performed by driving the mode bits on **SysAD[63:0]**, waiting at least two **SysClk** cycles, and then asserting **SysGnt*** for at least one **SysClk** cycle.
- After waiting at least another 100 ms for the internal clocks to restabilize, the external agent synchronizes all processor internal clocks by asserting **SysRespVal*** for one **SysClk** cycle.
- After waiting at least 100 ms for the internal clocks to again restabilize, (a third 100 ms restabilization period) the external agent negates **SysReset***.
- The external agent must retain mastership of the System interface, refrain from issuing external requests or nonmaskable interrupts, and ignore the system state bus until the processor asserts **SysReq***. The assertion of **SysReq*** indicates the processor is ready for operation. In a cluster arrangement, all processors must assert **SysReq***, indicating they are ready for operation.

During a Cold Reset sequence all processor internal state is initialized. A Cold Reset sequence causes the processor to start with a Reset exception.

Figure 8-2 shows the cold reset sequence.

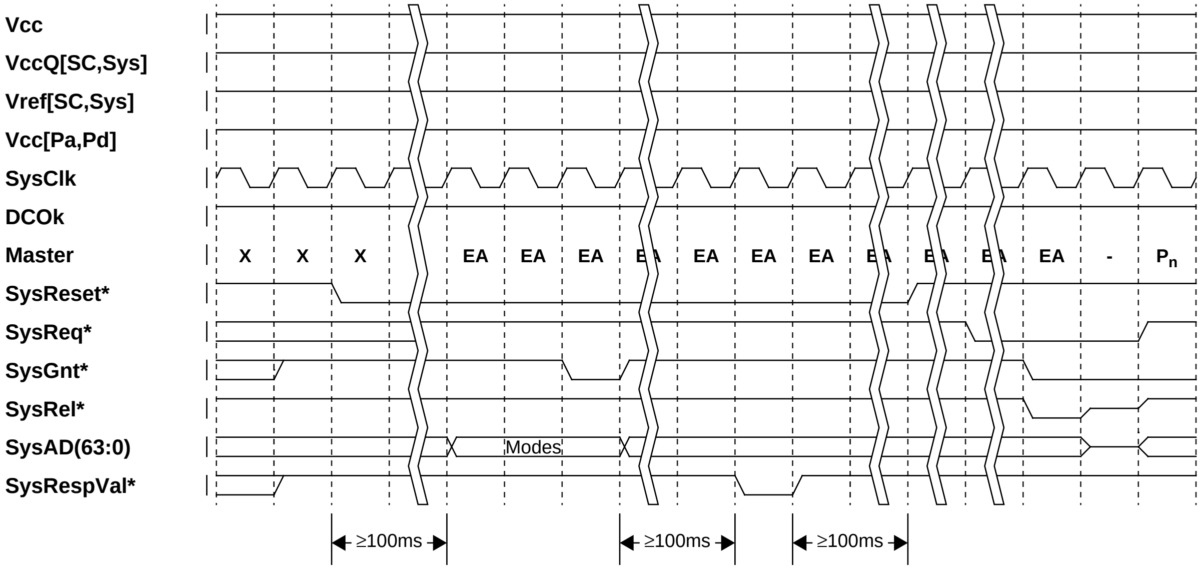


Figure 8-2 Cold Reset Sequence

8.4 Soft Reset Sequence

A Soft Reset sequence is used to reset the external interface of the processor without altering the mode bits while power and **SysClk** are stable.

The Soft Reset sequence is as follows:

- The external agent negates **SysGnt*** and **SysRespVal***.
- After waiting at least one **SysClk** cycle, the external agent asserts **SysReset*** for at least 16 **SysClk** cycles.
- The external agent must retain mastership of the System interface, refrain from issuing external requests or nonmaskable interrupts, and ignore system state bus until the processor asserts **SysReq***. The assertion of **SysReq*** indicates the processor is ready for operation. In a cluster arrangement, all processors must assert **SysReq***, indicating they are ready for operation.

During a Soft Reset sequence, all external interface state is initialized. The internal and secondary cache clocks are not affected by a Soft Reset sequence. The general purpose, CP0, and CP1 registers are preserved, as well as the primary and secondary caches.

A Soft Reset sequence causes a Soft Reset exception, in which the Soft Reset exception handler executes instructions from uncached space and uses CACHE instructions to analyze and dump the contents of the primary and secondary caches. To resume normal operation, a Cold Reset sequence must be initiated.

Figure 8-3 presents the Soft Reset sequence.

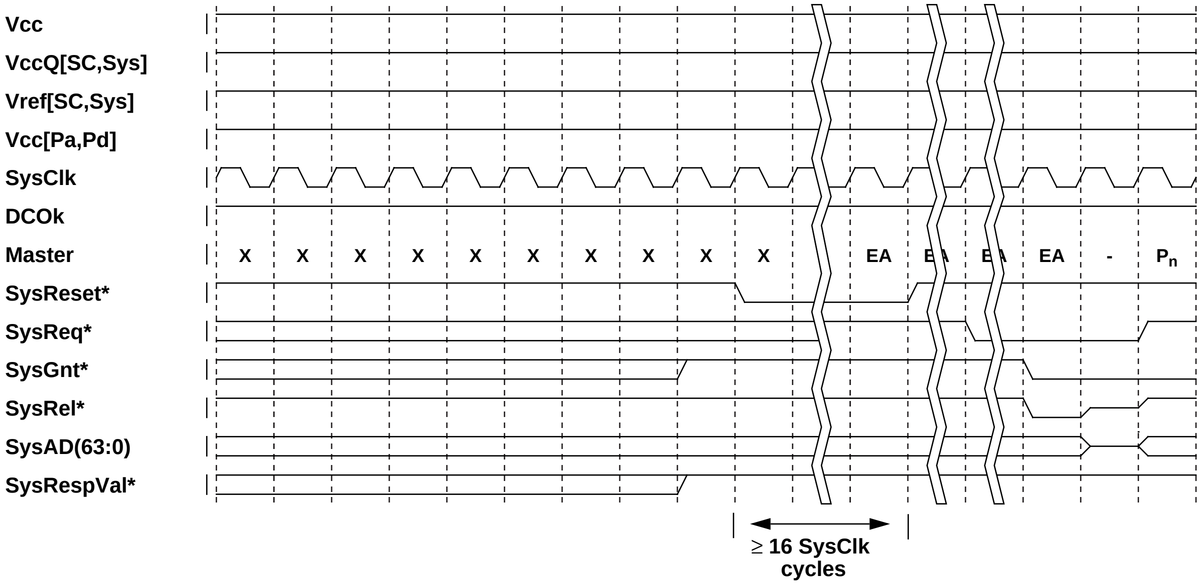


Figure 8-3 Soft Reset Sequence

8.5 Mode Bits

The R10000 processor uses mode bits to configure the operation of the microprocessor. These mode bits are loaded into the processor from the **SysAD[63:0]** bus during a power-on or cold reset sequence while **SysGnt*** is asserted. The **SysADChk[7:0]** bus does not have to contain correct ECC during mode bit initialization. During the reset sequence, the mode bits obtained from **SysAD[24:0]** are written into bits 24:0 of the CP0 *Config* register.

The mode bits are described in Table 8-1.

Table 8-1 Mode Bits

SysAD Bit	Name and Function	Value	Mode Setting	
			R10000	R12000
2:0	**Kseg0CA** Specifies the *kseg0* cache algorithm.	0	Reserved	
		1	Reserved	
		2	Uncached	
		3	Cacheable noncoherent	
		4	Cacheable coherent exclusive	
		5	Cacheable coherent exclusive on write	
		6	Reserved	
		7	Uncached accelerated	
4:3	**DevNum** Specifies the processor device number.	0-3		
5	**CohPrcReqTar** Specifies the target of processor coherent requests issued on the System interface by the processor.	0	External agent only	
		1	Broadcast	
6	**PrcElmReq** Specifies whether to enable processor eliminate requests onto the System interface by the processor.	0	Disable	
		1	Enable	
8:7	**PrcReqMax** Specifies the maximum number of outstanding processor requests allowed on the System interface by the processor.	0	1 outstanding processor request	
		1	2 outstanding processor requests	
		2	3 outstanding processor requests	
		3	4 outstanding processor requests	

★ Table 8-1 (cont.) Mode Bits

SysAD Bit	Name and Function	Value	Mode Setting	
			R10000	R12000
12:9	**SysClkDiv** Sets **PClk** to **SysClk** ratio; determines the System interface clock frequency; see Chapter 7, the section titled “System Interface Clock and Internal Processor Clock Domains.”	0	Reserved	Reserved
		1	Result of division by 1	Reserved
		2	Result of division by 1.5	Reserved
		3	Result of division by 2	Result of division by 2
		4	Result of division by 2.5	Result of division by 2.5
		5	Result of division by 3	Result of division by 3
		6	Result of division by 3.5	Result of division by 3.5
		7	Result of division by 4	Result of division by 4
		8	Reserved	Result of division by 4.5
		9	Reserved	Result of division by 5
		A	Reserved	Result of division by 5.5
		B	Reserved	Result of division by 6
		C	Reserved	Result of division by 7[‡‡]
		D	Reserved	Reserved
		E	Reserved	Reserved
		F	Reserved	Reserved
13	**SCBlkSize** Specifies the secondary cache block size.	0	16-word	
		1	32-word	
14	**SCCorEn** Specifies the method of correcting secondary cache data array ECC errors.	0	Retry access through corrector	
		1	Always access through corrector	
15	**MemEnd** Specifies the memory system endianness.	0	Little endian	
		1	Big endian	
18:16	**SCSize** Specifies the size of the secondary cache.	0	512 Kbyte	
		1	1 Mbyte	
		2	2 Mbyte	
		3	4 Mbyte	
		4	8 Mbyte	
		5	16 Mbyte	
		6	Reserved	
		7	Reserved	
21:19	**SCClkDiv** Sets **PClk** to **SCClk** ratio; determines the secondary cache clock frequency; see Chapter 7, the section titled “System Interface Clock and Internal Processor Clock Domains.”	0	Reserved	Reserved
		1	Result of division by 1	Reserved
		2	Result of division by 1.5	Result of division by 1.5
		3	Result of division by 2	Result of division by 2
		4	Result of division by 2.5	Result of division by 2.5
		5	Result of division by 3	Result of division by 3
		6	Reserved	Reserved
		7	Reserved	Result of division by 4
24:22	Reserved	0		Reserved
		1		Reserved
		2		Reserved
		3		Reserved
		4		Delay Speculative Dirty - fix for speculative store[†]
		5		Reserved
		6		Reserved
		7		Reserved

Table 8-1 (cont.) Mode Bits

SysAD Bit	Name and Function	Value	Mode Setting	
			R10000	**R12000**
28:25	**SCClkTap** Specifies the alignment[††] of **SCClk[5:0]** and **SCClk*[5:0]** relative to the internal secondary cache clock.	0 1 2 3 4 5 6 7 8 9 A B C D E F	**SCClk** same phase as internal clock **SCClk** 1/12 **PClk** period earlier than internal clock **SCClk** 2/12 **PClk** period earlier than internal clock **SCClk** 3/12 **PClk** period earlier than internal clock **SCClk** 4/12 **PClk** period earlier than internal clock **SCClk** 5/12 **PClk** period earlier than internal clock undefined undefined **SCClk** 6/12 **PClk** period earlier than internal clock **SCClk** 7/12 **PClk** period earlier than internal clock **SCClk** 8/12 **PClk** period earlier than internal clock **SCClk** 9/12 **PClk** period earlier than internal clock **SCClk** 10/12 **PClk** period earlier than internal clock **SCClk** 11/12 **PClk** period earlier than internal clock undefined undefined	
29[‡]	Reserved	0		
30[‡]	**ODrainSys** Specifies whether or not to configure select[†††] System interface bidirectional and output signals as open drain.	0 1	Push-pull Open drain	
31	**CTM** Specifies whether or not to enable cache test mode.	0 1	Disable Enable	
63:32	Reserved	0		

[‡‡] For R12000A only. This setting is reserved in the R12000 and R12000L.

★ [†] The Boot Mode bit 24 corresponds to the Config register[24] bit and this controls DSD during user mode. However, the DSD mode can also be enabled in the kernel mode by setting the Status register[24] bit. Config register[24] is read-only and can be set only at boot time.

If the DSD mode is set –

a) R12000 will not set the Dirty bit for a secondary cache block until the store instruction is the oldest in the Active List and is about to be executed. (An interrupt could cause a case where the dirty bit is set (store is no longer speculative), but the store does not immediately graduate. We believe this case should not cause any problem. This mode does prevent speculative stores from setting the dirty bit.)

b) This mode will have slightly lower performance due to the delay in the setting of the Dirty bit. This delay will occur just once per block refill from main memory, when it is necessary to set the dirty bit. Setting the bit requires about ten cycles; but usually the processor will continue to overlap execution of other instructions. Once a block becomes dirty in secondary cache, this mode has no performance effect.

c) In this mode, a miss in secondary cache, due to a store instruction which is not already the oldest in the pipeline, will cause a refill to the "clean exclusive" state. A hit to a shared line will immediately cause an upgrade to "clean exclusive". Thus, bus operations (which are relatively slow) will still begin speculatively.

Independent of the DSD mode, R12000 will delay a "cached, non-coherent" load until it is the oldest instruction. This change is implemented because a speculative load accessing an unmapped "xkphys" address as "cached, non-coherent" might bring data into the secondary cache without the proper coherency checks.

R12000 is doing no changes to prevent it from speculatively refilling cache lines in shared or clean states except the "xkphys" case described above.

†† Does not include the output buffer delay.

†††**SysReq***, **SysRel***, **SysCmd[11:0]**, **SysCmdPar**, **SysAD[63:0]**, **SysADChk[7:0], SysVal***, **SysState[2:0]**, **SysStatePar**, **SysStateVal***, **SysCorErr***, **SysUncErr***

‡ In the R12000A, the Boot Mode bits 30:29 are assigned to HSTL Mode bits as below;

SysAD Bit	Name and Function	Value	Mode Setting
29	**HSTL Mode** Specifies the HSTL class of output pins on the secondary cache interface.	0 1	**HSTL 1** **HSTL 2**
30	**HSTL Mode** Specifies the HSTL class of output pins on the System interface.	0 1	**HSTL 1** **HSTL 2**

9. Error Protection and Handling

This chapter presents the error protection and handling features provided by the R10000 processor.

Two types of errors can occur in an R10000 system:

- correctable
- uncorrectable

The following two sections describe them.

9.1 Correctable Errors

Correctable errors consist of:

- secondary cache tag array correctable ECC errors
- secondary cache data array correctable ECC errors
- System interface address/data bus correctable ECC errors

When the processor detects a correctable error, the error is automatically corrected, and normal operation continues. Secondary cache array scrubbing is not performed.

The processor informs the external agent that a correctable error was detected and then corrected by asserting the **SysCorErr*** signal for one **SysClk** cycle.

9.2 Uncorrectable Errors

Uncorrectable errors consist of:

- Primary instruction cache array parity errors
- Primary data cache array parity errors
- Secondary cache tag array uncorrectable ECC errors
- Secondary cache data array uncorrectable ECC errors
- System interface command bus parity errors
- System interface address/data bus uncorrectable ECC errors
- System interface response bus parity errors

When the processor detects an uncorrectable error, a Cache Error exception is posted. In general, the detection of an uncorrectable error does not disrupt any ongoing operations. However, the instruction fetch and load/store units never use data which contains an uncorrectable error.

To inform the external agent, the processor asserts **SysUncErr*** for one **SysClk** cycle whenever any of the following uncorrectable errors are detected:

- Primary instruction cache tag array parity errors
- Primary data cache tag array parity errors
- Secondary cache tag array uncorrectable ECC errors
- System interface command bus parity errors
- System interface address/data bus external address cycle uncorrectable ECC errors
- System interface response bus parity errors.

The processor informs the external agent that an uncorrectable tag error has been detected by asserting **SysUncErr*** for one **SysClk** cycle.

9.3 Propagation of Uncorrectable Errors

The processor assists the external agent in limiting the propagation of uncorrectable errors in the following manner:

- During external block data response cycles, if the data quality indication on **SysCmd(5)** is asserted, or if an uncorrectable ECC error is encountered on the system address/data bus while the ECC check indication on **SysCmd(0)** is asserted, the processor intentionally corrupts the ECC of the corresponding secondary cache quadword after receiving an external ACK completion response.
- During processor data cycles, the processor asserts the data quality indication on **SysCmd(5)** if the data is known to contain uncorrectable errors. The System interface ECC is never intentionally corrupted; the **SysCmd(5)** bit is used to indicate corrupted data.
- If an uncorrectable cache tag error is detected, the processor asserts **SysUncErr*** for one **SysClk** cycle.
- An external coherency request that detects a secondary cache tag array uncorrectable error asserts the secondary cache block tag quality indication on **SysState(2)** during the corresponding processor coherency state response.
- If an external coherency request requires a processor coherency data response, and a primary data cache tag parity error is encountered during the primary cache interrogation, or a secondary cache tag array uncorrectable error is encountered during the secondary cache interrogation, the processor asserts the data quality indication on **SysCmd(5)** for all doublewords of the corresponding processor coherency data response.

9.4 Cache Error Exception

The processor indicates an uncorrectable error has occurred by asserting a Cache Error exception.

The following four internal units detect and report uncorrectable errors:

- instruction cache
- data cache
- secondary cache
- System interface

Each of these four units maintains a unique local *CacheErr* register.

A Cache Error exception is imprecise; that is, it is not associated with a particular instruction. When any of the four units post a Cache Error exception, completed instructions are graduated before the Cache Error exception is taken. If there are Cache Error exceptions posted from more than one of the units, the exceptions are prioritized in the following order:

1. instruction cache
2. data cache
3. secondary cache
4. System interface.

The corresponding local *CacheErr* register is transferred to the CP0 *CacheErr* register and the CP0 *Status* register *ERL* bit is asserted. Instruction fetching begins from 0xa0000100 or 0xbfc00300, depending on the CP0 *Status* register *BEV* bit. The CP0 *ErrorEPC* register is loaded with the virtual address of the next instruction that has not been graduated, so that execution can resume after the Cache Error exception handler completes.

When *ERL*=1, the user address region becomes a 2-Gbyte uncached space mapped directly to the physical addresses. This allows the Cache Error handler to save registers directly to memory without having to use a register to construct the address.

The processor does not support nested Cache Error exception handling. While the CP0 *Status* register *ERL* bit is asserted, any subsequent Cache Error exceptions are ignored. However, the detection of additional uncorrectable errors is not inhibited, and additional Cache Error exceptions may be posted.†

---

† The hardware does not handle the case of multiple Cache Error exceptions in any special manner; caches are refilled as normal, and data forwarded to the appropriate functional units.

9.5 CP0 *CacheErr* Register *EW* Bit

When a unit detects an uncorrectable error, it records information about the error in its local *CacheErr* register and posts a Cache Error exception. If a subsequent uncorrectable error occurs while waiting for the Cache Error exception to be taken and transfer of the local *CacheErr* register to the CP0 *CacheErr* register to complete, the *EW* bit is set in its local *CacheErr* register. Once the Cache Error exception is taken, the *EW* bit in the CP0 *CacheErr* register is set and the Cache Error exception handler now determines that a second error has occurred.

Once the CP0 *CacheErr* register *EW* bit is set, it can only be cleared by a reset sequence.

9.6 CP0 *Status* Register *DE* Bit

Asserting the CP0 *Status* register *DE* bit suppresses the posting of future Cache Error exceptions. All local *CacheErr* registers are also prevented from being updated. Unlike the R4400 processor architecture, when the *DE* bit is asserted, cache hits are not inhibited when an uncorrectable error is detected. Correctable errors are handled normally when the *DE* bit is set.

> **NOTE:** Be careful when setting this bit, since it may cause erroneous data and/or instructions to be propagated.

9.7 CACHE Instruction

Uncorrectable error protection is suppressed for the Index Load Tag, Index Store Tag, Index Load Data, and Index Store Data CACHE instruction variations. These four variations may be used within a Cache Error exception handler to examine the cache tags and data without the occurrence of further uncorrectable errors.

9.8 Error Protection Schemes Used by R10000

Error protection schemes used in the R10000 processor are:

- parity
- sparse encoding
- ECC

These schemes are described in this section, and listed in Table 9-1.

Table 9-1 Error Protection Schemes Used in the R10000 Processor

Error Detection Used	What is Protected
Parity	Primary caches Secondary cache data System interface buses
Sparse encoding	Primary data cache state mod array
ECC (SECDED)	Secondary cache tag Secondary cache data System interface address/data bus

Parity

Parity is used to protect the primary caches and various System interface buses. The processor uses both odd and even parity schemes:

- in an odd parity scheme, the total number of ones on the protected data and the corresponding parity bit should be odd
- in an even parity scheme, the total number of ones on the protected data and the corresponding parity bit should be even.

Sparse Encoding

A sparse encoding is used to protect the primary data cache state mod array. In such a scheme, valid encodings are chosen so that altering a single bit creates an invalid encoding.

ECC

An error correcting code (ECC) is used to protect the secondary cache tag, the secondary cache data, and the System interface address/data bus. A distinct single-bit error correction and double-bit error detection (SECDED) code is used for each of these three applications.

9.9 Primary Instruction Cache Error Protection and Handling

This section describes error protection and error handling schemes for the primary instruction cache.

Error Protection

The primary instruction cache arrays have the following error protection schemes, as listed in Table 9-2.

Table 9-2 Primary Instruction Cache Array Error Protection

Array	Width	Error Protection
Tag Address	27-bit	Even parity
Tag State	1-bit	Even parity
Data	36-bit	Even parity
LRU	1-bit	None

Error Handling

All primary instruction cache errors are uncorrectable. If an error is detected, the instruction cache unit posts a Cache Error exception and initializes the *D*, *TA*, *TS*, and *PIdx* fields in the local *CacheErr* register (see Chapter 11, the section titled “CacheErr Register (27),” for more information). If an error is detected on the tag address or state array, the processor informs the external agent that an uncorrectable tag error was detected by asserting **SysUncErr*** for one **SysClk** cycle.

9.10 Primary Data Cache Error Protection and Handling

This section describes error protection and error handling schemes for the primary data cache.

Error Protection

The primary data cache arrays have the following error protection schemes, as listed in Table 9-3.

Table 9-3   Primary Data Cache Array Error Protection

Array	Width	Error Protection
Tag Address	28-bit	Even parity
Tag State	3-bit	Even parity
Tag Mod	3-bit	Sparse encoding
Data	8-bit	Even parity
LRU	1-bit	None

Error Handling

All primary data cache errors are uncorrectable. If an error is detected, the data cache unit posts a Cache Error exception and initializes the *EE*, *D*, *TA*, *TS*, *TM*, and *PIdx* fields in the local *CacheErr* register (see Chapter 11, the section titled “CacheErr Register (27),” for more information). If an error is detected on the tag address, state, or mod array, the processor informs the external agent that an uncorrectable tag error was detected by asserting **SysUncErr*** for one **SysClk** cycle.

9.11 Secondary Cache Error Protection and Handling

This section describes error protection and error handling schemes for the secondary cache.

Error Protection

The secondary cache arrays have the following error protection schemes, as listed in Table 9-4.

Table 9-4 Secondary Cache Array Error Protection

Array	Width	Error Protection
Data	128-bit	9-bit ECC + even parity
Tag	26-bit	7-bit ECC
MRU (Way prediction table)	1-bit	None

Error Handling

This section describes error handling for the data array and the tag array. As shown in Table 9-4, errors are not detected for the way prediction table.

Data Array

The 128-bit wide secondary cache data array is protected by a 9-bit wide ECC. An even parity bit for the 128 bits of data is used for rapid detection of correctable (single-bit) errors; when a correctable parity error is detected, the data is sent through the data corrector. The parity bit does not have any logical effect on the processor’s ability to either detect or correct errors.

Whenever the processor writes the secondary cache data array, it drives the proper ECC on **SCDataChk(8:0)** and even parity on **SCDataChk(9)**.

Data Array in Correction Mode

The secondary cache operates in correction mode when the **SCCorEn** mode bit is asserted. Whenever the processor reads the secondary cache data array in correction mode, the data is sent through a data corrector.

If a correctable error is detected, in-line correction is automatically made without affecting latency. The processor informs the external agent that a correctable error was detected and corrected by asserting **SysCorErr*** for one **SysClk** cycle.

If an uncorrectable error is detected, the secondary cache unit posts a Cache Error exception and initializes the *D* and *SIdx* fields in the local *CacheErr* register (see Chapter 11, the section titled “CacheErr Register (27),” for more information).

In correction mode, secondary-to-primary cache refill latency is increased by two **PClk** cycles. Multiple processors, operating in a lock-step fashion, remain synchronized in the presence of secondary cache data array correctable errors.

Table 9-5 presents the ECC matrix for the secondary cache data array.

Table 9-5 ECC Matrix for Secondary Cache Data Array

Check Bit		8	7654	3210																																
Data Bit					1111 2222 7654	1111 2222 3210	1111 1111 9876	1111 1111 5432	1111 1100 1098	1111 0000 7654	1111 0000 3210	9999 9876	9999 5432	9988 1098	8888 7654	8888 3210	7777 9876	7777 5432	7766 1098	6666 7654	6666 3210	5555 9876	5555 5432	5544 1098	4444 7654	4444 3210	3333 9876	3333 5432	3322 1098	2222 7654	2222 3210	1111 9876	1111 5432	11 1098	7654	3210
Number of ones per row	54	1	0000	0000	1111	1111	0000	0011	1111	1111	1110	0011	0000	0110	1111	1111	0000	0111	0010	0001	0110	0010	1101	0000	1011	0000	1101	0000	0000	0000	0010	0000	0010	0000	1000	0000
	53	0	1000	0000	1111	1111	1111	1111	0000	0011	0000	1111	1111	1111	0000	0111	0000	0011	1011	0010	0001	0000	0000	1000	0100	1000	1100	1000	0000	0010	1001	0000	1001	0000	0010	0000
	54	0	0100	0000	1000	0011	1111	1111	1111	1111	0000	0011	0000	0101	0000	0011	1111	1111	0101	1100	1000	0000	1000	0100	0000	0100	1100	0100	0011	1000	0100	1000	0100	1000	0101	0000
	53	0	0010	0000	0100	0010	1000	0000	1000	0000	0001	0010	1111	1100	1111	1011	1111	1100	0011	1111	1111	1101	0110	0010	0000	0010	0010	0010	1000	0101	0000	0100	1100	0100	0000	1000
	53	0	0001	0000	0010	0001	0100	0000	0100	0011	1111	1111	1000	0000	1000	0000	1000	0100	1111	1111	1111	1111	0010	0001	0000	0001	0000	0001	1111	1111	1100	0010	0000	0010	1000	0100
	53	0	0000	1000	0001	0000	0010	0011	0010	0000	1000	0001	0100	0100	0100	0000	0100	0110	1011	1111	1111	1100	0011	1111	1101	1111	0011	1111	0100	1100	0000	0001	0000	0001	0100	0010
	54	0	0000	0100	0000	1010	0001	0010	0001	0010	0001	1000	0010	0011	0010	0000	0010	0001	0000	0000	1011	1010	1111	1111	1100	0000	1010	0000	1101	0000	1111	1111	1111	1111	1100	0001
	53	0	0000	0010	0000	0100	0000	1001	0000	1001	0100	0100	0001	0011	0001	0010	0001	0000	0000	1000	0100	1101	1100	0000	1110	0000	1111	1111	1110	0000	1100	0000	1111	1111	1111	1111
	54	0	0000	0001	0000	0001	0000	0100	0000	0100	0010	0000	0000	1011	0000	1101	0000	1011	0100	0111	0000	0100	1110	0000	1111	1111	0110	0000	1100	0011	1111	1111	1100	0000	1111	1111
Number of ones per column		1	1111	1111	3333	3355	3333	3355	3333	3355	3333	3355	3333	3555	3333	3355	3333	3555	3355	5555	5555	5533	5553	3333	5533	3333	5553	3333	5533	3333	5533	3333	5533	3333	5533	3333

Data Array in Noncorrection Mode

When the **SCCorEn** mode bit is negated, the secondary cache operates in noncorrection mode. Whenever the processor reads the secondary cache data array in noncorrection mode, it checks for even parity on **SCDataChk(9)**. If a parity error is detected, it is assumed that a correctable error has occurred, and the secondary cache block is again read through a data corrector. During this re-read, the processor checks the **SCDataChk(8:0)** bus for the proper ECC.

If a correctable error is detected, correction is automatically performed in-line. To inform the external agent that a correctable error had been detected and corrected, the processor asserts **SysCorErr*** for one **SysClk** cycle.

If an uncorrectable error is detected, the secondary cache unit posts a Cache Error exception and initializes the *D* and *SIdx* fields in the local *CacheErr* register.

Secondary cache data array correctable errors are monitored with Performance Counter 0.

Tag Array

The 26-bit-wide secondary cache tag array is protected by a 7-bit-wide ECC. Table 9-6 presents the ECC matrix for the secondary cache tag array.

Table 9-6 ECC Matrix for Secondary Cache Tag Array

Check Bit			0	12	34	56				
Data Bit		2 5	222 432	22 10	11 98	11 76	1111 5432	11 1098	7654	3210
Number of ones per row	11 13 11	0 0 1	0100 1000 0010	1000 0100 1000	1000 0100 0001	0001 0010 1000	1111 1111 0000	1000 1111 1111	1000 0000 0100	1000 0100 0010
	11 13 12 14	1 0 1 0	0100 1000 0010 1111	0100 0001 0010 1100	0010 1000 0100 1100	0100 1000 0100 1100	1000 0100 0010 0001	0100 0000 0010 0001	1111 1111 0010 0001	0000 1111 1111 0001
Number of ones per column		3	3331	3311	3311	3311	3333	3333	3333	3333

Whenever the processor reads the secondary cache tag array, it checks the **SCTagChk(6:0)** bus for the proper ECC. If a correctable error is detected, correction is automatically performed in-line, without affecting latency. The processor asserts **SysCorErr*** for one **SysClk** cycle to inform the external agent that a correctable error has been detected and corrected. If an uncorrectable error is detected, the secondary cache unit posts a Cache Error exception and initializes the *TA* and *SIdx* fields in the local *CacheErr* register. The processor asserts **SysUncErr*** for one **SysClk** cycle to inform the external agent that an uncorrectable tag error has been detected.

Whenever the processor writes the secondary cache tag array, it drives the proper ECC on the **SCTagChk(6:0)** bus.

9.12 System Interface Error Protection and Handling

This section describes error protection and error handling schemes for the System interface.

Error Protection

The System interface buses have the following error protection schemes, as listed in Table 9-7.

Table 9-7 System Interface Bus Error Protection

Bus	**Width**	**Error Protection**
SysCmd	12-bit	Odd parity
SysAD	64-bit	8-bit ECC
SysState	3-bit	Odd parity
SysResp	5-bit	Odd parity

Error Handling

This section describes error handling on the system command bus, system address/data bus, system state bus, and system response bus.

SysCmd(11:0) Bus

The 12-bit wide system command bus, **SysCmd(11:0)**, is protected by odd parity.

Whenever the processor is in master state and it asserts **SysVal*** to indicate that it is driving valid information on the **SysCmd(11:0)** bus, it also drives odd parity on the **SysCmdPar** signal.

Whenever the processor is in slave state and an external agent asserts **SysVal*** to indicate that it is driving valid information on the **SysCmd(11:0)** bus, the processor checks the **SysCmdPar** signal for odd parity. If a parity error is detected, the processor ignores the **SysCmd(11:0)** and **SysAD(63:0)** buses for one **SysClk** cycle. The System interface unit posts a Cache Error exception and sets the SC bit in the local *CacheErr* register. Additionally, the processor informs the external agent by asserting **SysUncErr*** for one **SysClk** cycle.

> **Caution:** By ignoring the **SysCmd(11:0)** and **SysAD(63:0)** buses, the processor to become unsynchronized with other processors or the external agent on the cluster bus.

SysAD(63:0) Bus

The 64-bit wide system address/data bus, **SysAD(63:0)**, is protected by an 8-bit-wide ECC.

Processor in Master State

Whenever the processor is in master state and it asserts **SysVal*** to indicate it is driving valid information on the **SysAD(63:0)** bus, it also drives the proper ECC on the **SysADChk(7:0)** bus.

Processor in Slave State

Whenever the processor is in slave state, error checking is enabled with the assertion of **SysCmd(0)**, and an external agent asserts **SysVal*** to indicate it is driving valid information on the **SysAD(63:0)** bus, the processor checks the **SysADChk(7:0)** bus for the proper ECC.

Correctable Error Detected

If a correctable error is detected during an external address cycle, or during an external data cycle for a processor read or upgrade request originated by the R10000 processor, correction is automatically performed in-line without affecting latency. The processor asserts **SysCorErr*** for one **SysClk** cycle to inform the external agent that a correctable error has been detected and corrected.

Uncorrectable Error Detected

If an uncorrectable error is detected during an external address cycle, the processor ignores the **SysCmd(11:0)** and **SysAD(63:0)** buses for one **SysClk** cycle, and the System interface unit posts a Cache Error exception and sets the *SA* bit in the local *CacheErr* register. Additionally, the processor informs the external agent by asserting **SysUncErr*** for one **SysClk** cycle.

> **Caution:** By ignoring the **SysCmd(11:0)** and **SysAD(63:0)** buses, this processor may become unsynchronized with other processors or the external agent on the cluster bus.

If an uncorrectable error is detected or the data quality indication on **SysCmd(5)** is asserted during an external data cycle for a processor read or upgrade request originated by the processor, the R10000 asserts the corresponding incoming buffer uncorrectable error flag.

When the processor forwards block data from an incoming buffer entry after receiving an external ACK completion response, the associated incoming buffer uncorrectable error flags are checked, and if any are asserted, the System interface unit posts a single Cache Error exception and initializes the *EE*, *D*, and *SIdx* fields in the local *CacheErr* register.

When the processor forwards double/single/partial-word data from an incoming buffer entry after receiving an external ACK completion response, the associated incoming buffer uncorrectable error flag is checked and, if asserted, the System interface unit posts a Bus Error exception.

Table 9-8 presents the ECC matrix for the System interface address/data bus. This ECC matrix is identical to that used by the R4X00 System interface.

Table 9-8   ECC Matrix for System Interface Address/Data Bus

Check Bit			43		52											70		61	
Data Bit		6666 3210	55 98	5555 7654	55 32	5544 1098	4444 7654	4444 3210	3333 9876	3333 5432	3322 1098	2222 7654	2222 3210	1111 9876	1111 5432	11 10	9876	54	3210
Number of ones per row	27	1111	1100	1100	1000	1000	0000	1111	1111	0000	1000	1000	1000	1000	0000	1010	0100	1000	1000
	27	1111	1000	1000	1000	0100	0000	0000	0000	1111	0100	0100	0100	0100	1111	1100	1100	1010	0100
	27	0000	1000	1100	1010	0010	1111	1111	0000	0000	0010	0010	0010	0010	1111	1000	1000	1100	0010
	27	0000	1010	0100	1100	0001	1111	0000	1111	1111	0001	0001	0001	0001	0000	1000	1100	1000	0001
	27	1000	0101	0011	0100	0000	1000	1000	1000	1000	1111	1111	0000	1111	1000	1100	0001	0100	0000
	27	0100	1100	0010	0101	1111	0100	0100	0100	0100	0000	0000	1111	1111	0100	0100	0011	0100	0000
	27	0010	0100	0011	1100	1111	0010	0010	0010	0010	1111	0000	0000	0000	0010	0100	0010	0101	1111
	27	0001	0100	0001	0100	0000	0001	0001	0001	0001	0000	1111	1111	0000	0001	0101	0011	1100	1111
Number of ones per column		3333	5511	3333	5511	3333	3333	3333	3333	3333	3333	3333	3333	3333	3333	5511	3333	5511	3333

SysState(2:0) Bus

The 3-bit wide system state bus, **SysState(2:0)**, is protected by odd parity. The processor drives odd parity on the **SysStatePar** signal.

SysResp(4:0) Bus

The 5-bit wide system response bus, **SysResp(4:0)**, is protected by odd parity.

Whenever an external agent asserts **SysRespVal*** to indicate it is driving valid information on the **SysResp(4:0)** bus, the processor checks the **SysRespPar** signal for odd parity. If a parity error is detected, the processor ignores the **SysResp(4:0)** bus for one **SysClk** cycle. The System interface unit posts a Cache Error exception and sets the *SR* bit in the local *CacheErr* register. Additionally, the processor informs the external agent by asserting **SysUncErr*** for one **SysClk** cycle.

> **Caution:** If the processor ignores the **SysResp(4:0)** bus, it may become unsynchronized with other processors or the external agent on the cluster bus. Also, the processor will "hang" if a parity error is detected on the **SysResp[4:0]** bus during an external completion response cycle for a processor double/single/partial-word read request originated by the processor. The external agent may initiate a Soft Reset sequence to obtain the contents of the *CacheErr* register, and the *CacheErr* register will indicate a System interface uncorrectable system response bus error.

Protocol Observation

The processor continuously observes the protocol on the System interface.
Table 9-9 presents the supported protocol observations and the associated error handling sequence.

Table 9-9 Protocol Observation

Protocol Observation	Error Handling
External response data cycle with an unexpected request number during an external block data response for a processor block read or upgrade request originated by the processor.	Ignore the external response data cycle
External block data response specifying a *Reserved* cache block state for a processor block read or upgrade request originated by the processor.	Override the cache block state to *CleanExclusive*
External block data response specifying a *Shared* cache block state for a processor coherent block read exclusive or upgrade request originated by the processor.	Override the cache block state to *CleanExclusive*
External completion response specifying a *Reserved* completion indication.	Ignore the external completion response
External ACK completion response for a processor read request originated by the processor that has not received an external data response.	Override the external ACK completion response to a NACK

10. JTAG Interface Operation

The JTAG interface is implemented according to the standard IEEE 1149.1 test access port protocol specifications.

The JTAG interface accesses the JTAG controller and instruction register as well as a boundary scan register. The JTAG operation does not require **DCOk** to be asserted or **SysClk** to be running; however, if **DCOk** is asserted the **SysClk** must run at the specified minimum frequency or the core logic may be damaged.

10.1 Test Access Port (TAP)

The test access port (TAP) consists of four interface signals. These signals are used to control the serial loading and unloading of instructions and test data, as well as to execute tests.

The TAP consists of the following signals:

JTDI: Serial data input (Input signal)
JTDO: Serial data output (Output signal)
JTMS: Mode select (Input signal)
JTCK: Clock (Input signal)
★ **JTRST**: Reset input (Input signal, active low)

The timing and the relationship of the TAP signals follows the IEEE 1149.1 standard protocol.

TAP Controller (Input)

The R10000 processor implements the 16-state TAP controller specified by the IEEE 1149.1 standard in the following manner:

- The **JTMS** signal operates the state machine synchronized by the **JTCK** signal.
- The TAP controller is reset by keeping the **JTMS** signal asserted through five consecutive edges of **JTCK.** This reset condition sets the reset state of the controller.
- ★ In the R12000, the TAP controller is also reset by asserting **SysReset***. This pin must not be asserted while using the boundary scan register.
- ★ In the R12000A, the TAP controller is also reset by asserting **JTRST**. This signal can be asserted asynchronously.

10.2 Instruction Register

The JTAG instruction register is four bits wide, permitting a total of 16 instructions to control the selection of the bypass register, the boundary scan register, and other data registers.

The encoding of the instruction register is given in Table 10-1:

Table 10-1 JTAG Instruction Register Encoding

MSB...LSB	Selected Data Register
0000	Boundary Scan Register
0001	Sample - Preload
0010 to 1110	Data Register (not used)
1111	Bypass Register

The 0001 value is provided to represent sample-preload, but also selects the boundary scan register.

During a reset of the TAP controller, the value 1111 is loaded into the parallel output of the instruction register, thus selecting the bypass register as the default.

During the Shift-IR state of the TAP controller, data is shifted serially into the instruction register from **JTDI**, and the LSB of the instruction register is shifted out onto **JTDO**.

During the Update-IR state, the current state of the instruction register is shifted to its parallel output for decoding.

10.3 Bypass Register

The bypass register is 1 bit wide.

When the bypass register is selected and the TAP controller is in the Shift-DR state, data on **JTDI** is shifted into the bypass register and the output of the bypass register is shifted out onto **JTDO**.

10.4 Boundary Scan Register

The bypass register is 1 bit wide.

The boundary scan data register is selected by loading 0000 into the instruction register. The Shift-DR, Update-DR, and Capture-DR states of the TAP controller are used to operate the boundary scan register according to the IEEE 1149.1 standard specifications.

The boundary scan register provides serial access to each of the processor interface pins, as shown in Figure 10-1. Hence, the boundary scan register can be used to load and observe specific logic values on the processor pins.

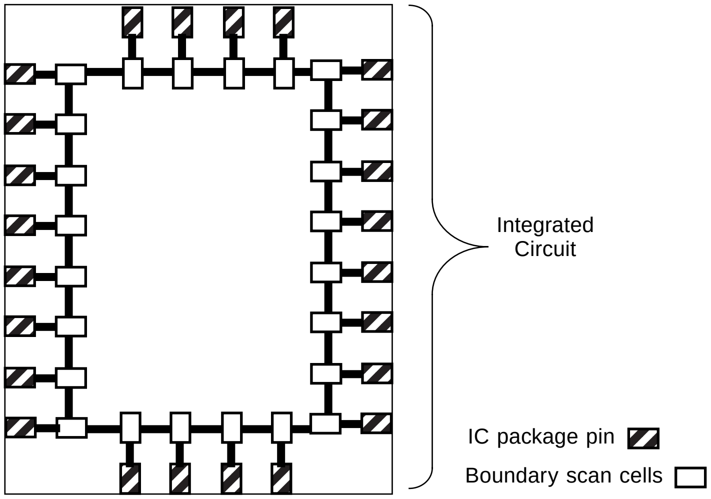


Figure 10-1 JTAG Boundary Scan Cells

The main application of the boundary scan register is board-level interconnect testing.

The use of the boundary scan register for applying data to and capturing data from the internal microprocessor circuitry is not supported.

The boundary scan register list for rev 1.2 of the fab is given in Table 10-2. The **TriState** signal will be eliminated from the BSR in rev 2.0 of the fab, and beyond.

An additional bit is provided in the boundary scan register to control the direction of bidirectional pins. As it is loaded through JTDI, this bit is the first bit in the boundary scan chain. The logic value of this bit is latched during the Update-DR state, and sets the direction of all bidirectional pins as follows:

Value	Direction
0	Input
1	Output

The value is set to 0 during reset, setting all bidirectional pins to *input* prior to any boundary scan operations.

Table 10-2 Boundary Scan Register Pinlist, rev 1.2

Signal	Signal	Signal	Signal	Signal	Signal
1. SCDataChk[1]	2. SCData[63]	3. SCData[62]	4. SCData[61]	5. SCData[60]	6. SCData[59]
7. SCData[58]	8. SCData[57]	9. SCData[56]	10. SCData[55]	11. SCData[54]	12. SCData[53]
13. SCData[52]	14. SCData[51]	15. SCData[50]	16. SCData[49]	17. SCData[48]	18. SCData[47]
19. SCData[46]	20. SCData[45]	21. SCData[44]	22. SCData[43]	23. SCData[42]	24. SCData[41]
25. SCData[40]	26. SCData[39]	27. SCData[38]	28. SCData[37]	29. SCData[36]	30. SCData[35]
31. SCData[34]	32. SCData[33]	33. SCData[32]	34. SysAD[0]	35. SysAD[1]	36. SysAD[2]
37. SysAD[3]	38. SysAD[4]	39. SysAD[5]	40. SysAD[6]	41. SysAD[7]	42. SysAD[8]
43. SysAD[9]	44. SysAD[10]	45. SysAD[11]	46. SysAD[12]	47. SysAD[13]	48. SysAD[14]
49. SysAD[15]	50. SCData[0]	51. SCData[1]	52. SCData[2]	53. SCData[3]	54. SCData[4]
55. SCData[5]	56. SCData[6]	57. SCData[7]	58. SCData[8]	59. SCData[9]	60. SCData[10]
61. SCData[11]	62. SCData[12]	63. SCData[13]	64. SCData[14]	65. SCData[15]	66. SCData[16]
67. SCData[17]	68. SCData[18]	69. SCData[19]	70. SCData[20]	71. SCData[21]	72. SCData[22]
73. SCData[23]	74. SCData[24]	75. SCData[25]	76. SCData[26]	77. SCData[27]	78. SCData[28]
79. SCData[29]	80. SCData[30]	81. SCData[31]	82. SCDataChk[0]	83. SCAAddr[18]	84. SCAAddr[17]
85. SCAAddr[16]	86. SCAAddr[15]	87. SCAAddr[14]	88. SCAAddr[13]	89. SCAAddr[12]	90. SCAAddr[11]
91. SCAAddr[10]	92. SCAAddr[9]	93. SCDataChk[2]	94. SCDataChk[4]	95. SCData[64]	96. SCData[65]
97. SCData[66]	98. SCData[67]	99. SCData[68]	100. SCData[69]	101. SCData[70]	102. SCData[71]
103. SCDataChk[9]	104. SysCyc*	105. SysAD[32]	106. SysAD[33]	107. SysAD[34]	108. SysAD[35]
109. SysAD[36]	110. SysAD[37]	111. SysAD[38]	112. SysAD[39]	113. SysAD[40]	114. SysAD[41]
115. SysAD[42]	116. SysAD[43]	117. SysAD[44]	118. SysAD[45]	119. SysAD[46]	120. SysAD[47]
121. SCData[72]	122. SCData[73]	123. SCData[74]	124. SCData[75]	125. SCData[76]	126. SCData[77]
127. SCData[78]	128. SCData[79]	129. SCAAddr[0]	130. SCAAddr[1]	131. SCAAddr[2]	132. SCAAddr[3]
133. SCAAddr[4]	134. SCAAddr[5]	135. SCAAddr[6]	136. SCAAddr[7]	137. SCAAddr[8]	138. SCADWay
139. SCADCS*	140. SCADOE*	141. SCADWr*	142. SCData[80]	143. SCData[81]	144. SCData[82]
145. SCData[83]	146. SCData[84]	147. SCData[85]	148. SCData[86]	149. SCData[87]	150. SCData[88]
151. SCData[89]	152. SCData[90]	153. SCData[91]	154. SCData[92]	155. SCData[93]	156. SCData[94]
157. SCData[95]	158. SCDataChk[6]	159. SCDataChk[8]	160. Spare1	161. SCTCS*	162. SCTOE*
163. SCTWr*	164. SCTag[25]	165. SCTag[24]	166. SCTag[23]	167. SCTag[22]	168. SCTag[21]
169. SCTag[20]	170. SCTag[19]	171. SCTag[18]	172. SCTag[17]	173. SCTag[16]	174. SCTag[15]
175. SCTag[14]	176. SCTag[13]	177. SCTag[12]	178. SCTag[11]	179. SCTag[10]	180. SCTag[9]
181. SCTag[8]	182. SCTag[7]	183. SCTag[6]	184. SCTag[5]	185. SCTag[4]	186. SCTag[3]
187. SCTag[2]	188. SCTag[1]	189. SCTag[0]	190. SCTagLSBAddr	191. TriState‡	192. SCTWay
193. SCTagChk[6]	194. SCTagChk[5]	195. SCTagChk[4]	196. SCTagChk[3]	197. SCTagChk[2]	198. SCTagChk[1]
199. SCTagChk[0]	200. SysCmd[0]	201. SysCmd[1]	202. SysCmd[2]	203. SysCmd[3]	204. SysCmd[4]
205. SysCmd[5]	206. SysCmd[6]	207. SysCmd[7]	208. SysCmd[8]	209. SysCmd[9]	210. SysCmd[10]
211. SysCmd[11]	212. SysCmdPar	213. SysVal*	214. SysReq*	215. SysRel*	216. SysGnt*
217. SysReset*	218. SysRespVal*	219. SysRespPar	220. SysResp[4]	221. SysResp[3]	222. SysResp[2]
223. SysResp[1]	224. SysResp[0]	225. SysGblPerf*	226. SysRdRdy*	227. SysWrRdy*	228. SysStateVal*
229. SysStatePar	230. SysState[2]	231. SysState[1]	232. SysState[0]	233. SysCorErr*	234. SysUncErr*
235. SysNMI*	236. SCDataChk[7]	237. SCDataChk[5]	238. SCData[127]	239. SCData[126]	240. SCData[125]
241. SCData[124]	242. SCData[123]	243. SCData[122]	244. SCData[121]	245. SCData[120]	246. SCData[119]
247. SCData[118]	248. SCData[117]	249. SCData[116]	250. SCData[115]	251. SCData[114]	252. SCData[113]

‡ Will be eliminated after rev. 1.2.

Table 10-2 (cont.) Boundary Scan Register Pinlist, rev 1.2

Signal	Signal	Signal	Signal	Signal	Signal
253. SCData[112]	254. SCBDWr*	255. SCBDOE*	256. SCBDCS*	257. SCBDWay	258. SCBAddr[8]
259. SCBAddr[7]	260. SCBAddr[6]	261. SCBAddr[5]	262. SCBAddr[4]	263. SCBAddr[3]	264. SCBAddr[2]
265. SCBAddr[1]	266. SCBAddr[0]	267. SCData[111]	268. SCData[110]	269. SCData[109]	270. SCData[108]
271. SCTag[8]	272. SCTag[7]	273. SCTag[6]	274. SCTag[5]	275. SCTag[4]	276. SCTag[3]
277. SCTag[2]	278. SCTag[1]	279. SCTag[0]	280. SCTagLSBAddr	281. TriState[‡]	282. SCTWay
283. SCTagChk[6]	284. SCTagChk[5]	285. SCTagChk[4]	286. SCTagChk[3]	287. SCTagChk[2]	288. SCTagChk[1]
289. SCTagChk[0]	290. SysCmd[0]	291. SysCmd[1]	292. SysCmd[2]	293. SysCmd[3]	294. SysCmd[4]
295. SysCmd[5]	296. SysCmd[6]	297. SysCmd[7]	298. SysCmd[8]	299. SysCmd[9]	300. SysCmd[10]
301. SysCmd[11]	302. SysCmdPar	303. SysVal*	304. SysReq*	305. SysRel*	306. SysGnt*
307. SysReset*	308. SysRespVal*	309. SysRespPar	310. SysResp[4]	311. SysResp[3]	312. SysResp[2]
313. SysResp[1]	314. SysResp[0]	315. SysGblPerf*	316. SysRdRdy*	317. SysWrRdy*	318. SysStateVal*
319. SysStatePar	320. SysState[2]	321. SysState[1]	322. SysState[0]	323. SysCorErr*	324. SysUncErr*
325. SysNMI*	326. SCDataChk[7]	327. SCDataChk[5]	328. SCData[127]	329. SCData[126]	330. SCData[125]
331. SCData[124]	332. SCData[123]	333. SCData[122]	334. SCData[121]	335. SCData[120]	336. SCData[119]
337. SCData[118]	338. SCData[117]	339. SCData[116]	340. SCData[115]	341. SCData[114]	342. SCData[113]
343. SCData[112]	344. SCBDWr*	345. SCBDOE*	346. SCBDCS*	347. SCBDWay	348. SCBAddr[8]
349. SCBAddr[7]	350. SCBAddr[6]	351. SCBAddr[5]	352. SCBAddr[4]	353. SCBAddr[3]	354. SCBAddr[2]
355. SCBAddr[1]	356. SCBAddr[0]	357. SCData[111]	358. SCData[110]	359. SCData[109]	360. SCData[108]
361. SCData[107]	362. SCData[106]	363. SCData[105]	364. SCData[104]	365. SysAD[63]	366. SysAD[62]
367. SysAD[61]	368. SysAD[60]	369. SysAD[59]	370. SysAD[58]	371. SysAD[57]	372. SysAD[56]
373. SysAD[55]	374. SysAD[54]	375. SysAD[53]	376. SysAD[52]	377. SysAD[51]	378. SysAD[50]
379. SysAD[49]	380. SysAD[48]	381. SysADChk[7]	382. SysADChk[6]	383. SysADChk[5]	384. SysADChk[4]
385. SysADChk[3]	386. SysADChk[2]	387. SysADChk[1]	388. SysADChk[0]	389. SysAD[31]	390. SysAD[30]
391. SysAD[29]	392. SysAD[28]	393. SysAD[27]	394. SysAD[26]	395. SysAD[25]	396. SysAD[24]
397. SysAD[23]	398. SysAD[22]	399. SysAD[21]	400. SysAD[20]	401. SysAD[19]	402. SysAD[18]
403. SysAD[17]	404. SysAD[16]	405. SCData[103]	406. SCData[102]	407. SCData[101]	408. SCData[100]
409. SCData[99]	410. SCData[98]	411. SCData[97]	412. SCData[96]	413. SCDataChk[3]	414. SCBAddr[9]
415. SCBAddr[10]	416. SCBAddr[11]	417. SCBAddr[12]	418. SCBAddr[13]	419. SCBAddr[14]	420. SCBAddr[15]
421. SCBAddr[16]	422. SCBAddr[17]	423. SCBAddr[18]			

[‡] Will be eliminated after rev. 1.2.

11. Coprocessor 0

This chapter describes the Coprocessor 0 operation, concentrating on the CP0 register definitions and the R10000 processor implementation of CP0 instructions.

The Coprocessor 0 (CP0) registers control the processor state and report its status. These registers can be read using MFC0 instructions and written using MTC0 instructions. CP0 registers are listed in Table 11-1.

Table 11-1 Coprocessor 0 Registers

Register No.	Register Name	Description
0	Index	Programmable register to select TLB entry for reading or writing
1	Random	Pseudo-random counter for TLB replacement
2	EntryLo0	Low half of TLB entry for even VPN (Physical page number)
3	EntryLo1	Low half of TLB entry for odd VPN (Physical page number)
4	Context	Pointer to kernel virtual PTE table in 32-bit addressing mode
5	PageMask	Mask that sets the TLB page size
6	Wired	Number of wired TLB entries (lowest TLB entries not used for random replacement)
7	Undefined	Undefined
8	BadVAddr	Bad virtual address
9	Count	Timer count
10	EntryHi	High half of TLB entry (Virtual page number and ASID)
11	Compare	Timer compare
12	Status	Processor Status Register
13	Cause	Cause of the last exception taken
14	EPC	Exception Program Counter
15	PRId	Processor Revision Identifier
16	Config	Configuration Register (secondary cache size, etc.)
17	LLAddr	Load Linked memory address
18	WatchLo	Memory reference trap address (low bits Adr[39:32])
19	WatchHi	Memory reference trap address (high bits Adr[31:3])
20	XContext	Pointer to kernel virtual PTE table in 64-bit addressing mode
21	FrameMask	Mask the physical addresses of entries which are written into the TLB
22	Diagnostic	Branch diagnostic register
23	Undefined	Undefined
24	Undefined	Undefined
25	Performance Counter	Performance count and control
26	ECC	Secondary cache ECC and primary cache parity
27	CacheErr	Cache Error and Status register
28	TagLo	Cache Tag register - low bits
29	TagHi	Cache Tag register - high bits
30	ErrorEPC	Error Exception Program Counter

Coprocessor 0 instructions are enabled if the processor is in Kernel mode, or if bit 28 (*CU0*) is set in the *Status* register. Otherwise, executing one of these instructions generates a Coprocessor 0 Unusable exception.

11.1 Index Register (0)

The *Index* register is a 32-bit, read/write register containing six bits to index an entry in the TLB. The high-order bit of the register shows the success or failure of a TLB Probe (TLBP) instruction.

The *Index* register also specifies the TLB entry affected by TLB Read (TLBR) or TLB Write Index (TLBWI) instructions.

Figure 11-1 shows the format of the *Index* register; Table 11-2 describes the *Index* register fields.

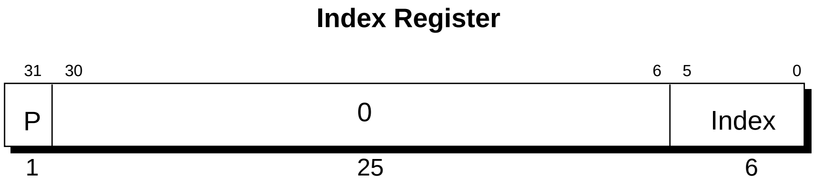


Figure 11-1 Index Register

Table 11-2 Index Register Field Descriptions

Field	Description
P	Probe failure. Set to 1 when the previous TLBProbe (TLBP) instruction was unsuccessful.
Index	Index to the TLB entry affected by the TLBRead and TLBWrite instructions
0	Reserved. Must be written as zeroes, and returns zeroes when read.

11.2 Random Register (1)

The *Random* register is a read-only register of which six bits index an entry in the TLB. This register decrements when any instruction graduates at that particular cycle, and its values range between an upper and a lower bound, as follows:

- The lower bound is set by the number of TLB entries reserved for exclusive use by the operating system (the contents of the *Wired* register).
- The upper bound is set by the total number of TLB entries minus 1 (64 – 1 maximum).

The *Random* register specifies the entry in the TLB that is affected by the TLB Write Random instruction. The register does not need to be read for this purpose; however, the register is readable to verify proper operation of the processor.

To simplify testing, the *Random* register is set to the value of the upper bound upon system reset. This register is also set to the upper bound when the *Wired* register is written.

Figure 11-2 shows the format of the *Random* register; Table 11-3 describes the *Random* register fields.

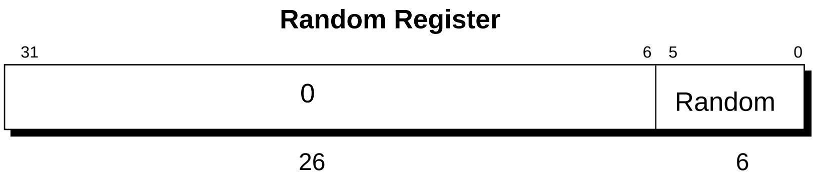


Figure 11-2 Random Register

Table 11-3 Random Register Field Descriptions

Field	Description
Random	TLB Random index
0	Reserved. Must be written as zeroes, and returns zeroes when read.

11.3 EntryLo0 (2) and EntryLo1 (3) Registers

The *EntryLo* register consists of two registers with identical formats:

- *EntryLo0* is used for even virtual pages.
- *EntryLo1* is used for odd virtual pages.

The *EntryLo0* and *EntryLo1* registers are read/write registers. They hold the physical page frame number (PFN) of the TLB entry for even and odd pages, respectively, when performing TLB read and write operations. Figure 11-3 shows the format of these registers.

EntryLo0 and EntryLo1 Registers

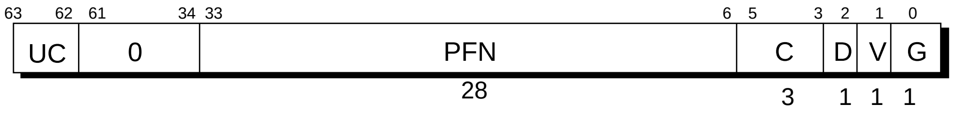


Figure 11-3 Fields of the EntryLo0 and EntryLo1 Registers

Table 11-4 Description of EntryLo Registers' Fields

Field	Description
UC	Uncached attribute
PFN	Page frame number; the upper bits of the physical address.
C	Specifies the TLB page coherency attribute.
★ D	Dirty. If this bit is set, the page is marked as dirty and, therefore, writable. This bit is actually a write-protect bit that software can use to prevent alteration of data.
V	Valid. If this bit is set, it indicates that the TLB entry is valid; otherwise, a TLBL or TLBS invalid exception occurs.
G	Global. If this bit is set in both Lo0 and Lo1, then the processor ignores the ASID during TLB lookup.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

The *PFN* fields of the *EntryLo0* and *EntryLo1* registers span bits 33:6 of the 40-bit physical address.

Two additional bits for the mapped space's *uncached attribute* can be loaded into bits 63:62 of the *EntryLo* register, which are then written into the TLB with a TLB Write. During the address cycle of processor double/single/partial-word read and write requests, and during the address cycle of processor *uncached accelerated* block write requests, the processor drives the uncached attribute on **SysAD[59:58]**. The same *EntryLo* registers are used for the 64-bit and 32-bit addressing modes. In both modes the registers are 64 bits wide, however when the MIPS III ISA is not enabled (32-bit User and Supervisor modes) only the lower 32 bits of the *EntryLo* registers are accessible.

MIPS III is disabled when the processor is in 32-bit Supervisor or User mode. Loading of the integer registers is limited to bits *31:0*, sign-extended through bits *63:32*. *EntryLo[33:31]* or *PFN[39:38]* can only be set to all zeroes or all ones. In 32- and 64-bit modes, the *UC* and *PFN* bits of both *EntryLo* registers are written into the TLB. The *PFN* bits can be masked by setting bits in the *FrameMask* register (described in this chapter) but the *UC* bits cannot be masked or initialized in 32-bit User or Supervisor modes. In 32-bit Kernel mode, MIPS III is enabled and 64-bit operations are always available to program the *UC* bits.

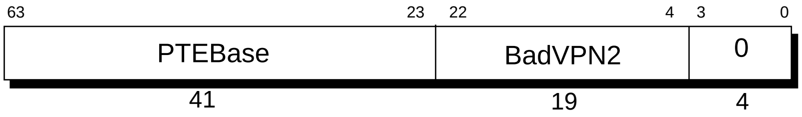
There is only one *G* bit per TLB entry, and it is written with *EntryLo0[0]* and *EntryLo1[0]* on a TLB write.

11.4 Context Register (4)

The *Context* register is a read/write register containing the pointer to an entry in the page table entry (PTE) array; this array is an operating system data structure that stores virtual-to-physical address translations.

When there is a TLB miss, the CPU loads the TLB with the missing translation from the PTE array. Normally, the operating system uses the *Context* register to address the current page map which resides in the kernel-mapped segment, *kseg3*. The *Context* register duplicates some of the information provided in the *BadVAddr* register, but the information is arranged in a form that is more useful for a software TLB exception handler.

Figure 11-4 shows the format of the *Context* register; Table 11-5 describes the *Context* register fields.



Context Register

Figure 11-4 Context Register Format

Table 11-5 Context Register Fields

Field	Description
BadVPN2	This field is written by hardware on a miss. It contains the virtual page number (VPN) of the most recent virtual address that did not have a valid translation.
0	Reserved. Must be written as zeroes, and returns zeroes when read.
PTEBase	This field is a read/write field for use by the operating system. It is normally written with a value that allows the operating system to use the *Context* register as a pointer into the current PTE array in memory.

The 19-bit *BadVPN2* field contains bits 31:13 of the virtual address that caused the TLB miss; bit 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4-Kbyte page size, this format can directly address the pair-table of 8-byte PTEs. For other page and PTE sizes, shifting and masking this value produces the appropriate address.

11.5 PageMask Register (5)

The *PageMask* register is a read/write register used for reading from or writing to the TLB; it holds a comparison mask that sets the variable page size for each TLB entry, as shown in Table 11-6. Format of the register is shown in Figure 11-5.

TLB read and write operations use this register as either a source or a destination; when virtual addresses are presented for translation into physical address, the corresponding bits in the TLB identify which virtual address bits among bits 24:13 are used in the comparison. When the *Mask* field is not one of the values shown in Table 11-6, the operation of the TLB is undefined. The 0 field is reserved; it must be written as zeroes, and returns zeroes when read.

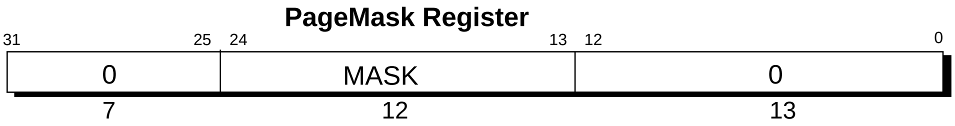


Figure 11-5 PageMask Register

Table 11-6 Mask Field Values for Page Sizes

Page Size (Mask)	**Bit**											
	24	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**
4 Kbytes	0	0	0	0	0	0	0	0	0	0	0	0
16 Kbytes	0	0	0	0	0	0	0	0	0	0	1	1
64 Kbytes	0	0	0	0	0	0	0	0	1	1	1	1
256 Kbytes	0	0	0	0	0	0	1	1	1	1	1	1
1 Mbyte	0	0	0	0	1	1	1	1	1	1	1	1
4 Mbytes	0	0	1	1	1	1	1	1	1	1	1	1
16 Mbytes	1	1	1	1	1	1	1	1	1	1	1	1

11.6 Wired Register (6)

The *Wired* register is a read/write register that specifies the boundary between the *wired* and *random* entries of the TLB as shown in Figure 11-6. Wired entries are fixed, nonreplaceable entries, which cannot be overwritten by a TLB write operation. Random entries can be overwritten.

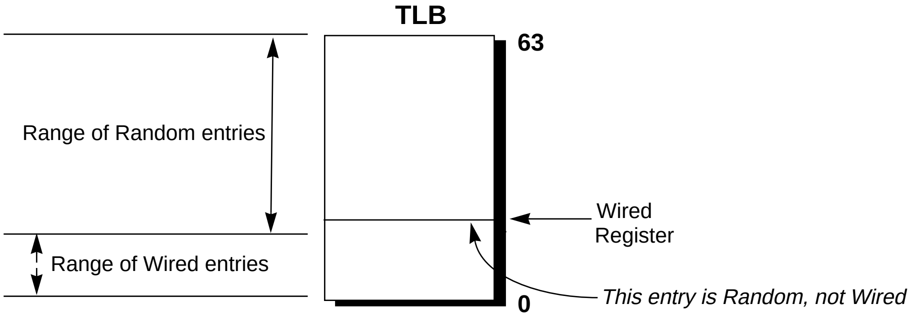

TLB
63
0
Range of Random entries
Range of Wired entries
Wired Register
This entry is Random, not Wired

Figure 11-6 Wired Register Boundary

The *Wired* register is set to 0 upon system reset. Writing this register also sets the *Random* register to the value of its upper bound (see *Random* register, above). Figure 11-7 shows the format of the *Wired* register; Table 11-7 describes the register fields.

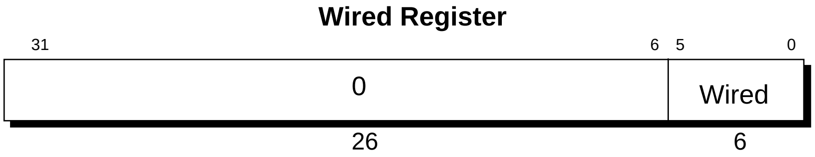

Wired Register
31
6
5
0
0
Wired
26
6

Figure 11-7 Wired Register

Table 11-7 Wired Register Field Descriptions

Field	Description
Wired	TLB Wired boundary
0	Reserved. Must be written as zeroes, and returns zeroes when read.

11.7 BadVAddr Register (8)

The Bad Virtual Address register (*BadVAddr*) is a read-only register that displays the most recent virtual address that caused either a TLB or Address Error exception. The *BadVAddr* register remains unchanged during Soft Reset, NMI, or Cache Error exceptions. Otherwise, the architecture leaves this register undefined.

Figure 11-8 shows the format of the *BadVAddr* register.

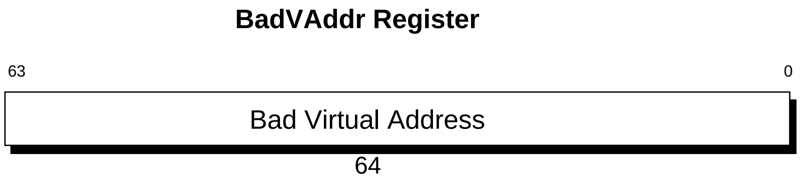

BadVAddr Register
63
0
Bad Virtual Address
64

Figure 11-8 BadVAddr Register Format

11.8 Count and Compare Registers (9 and 11)

The *Count* and *Compare* registers are 32-bit read/write registers whose formats are shown in Figure 11-9.

The *Count* register acts as a real-time timer. Like the R4400 implementation, the R10000 *Count* register is incremented every *other* **PClk** cycle. However, unlike the R4400, the R10000 processor has no Timer Interrupt Enable boot-mode bit, so the only way to disable the timer interrupt is to negate the interrupt mask bit, *IM[7]*, in the *Status* register. This means the timer interrupt cannot be disabled without also disabling the *Performance Counter* interrupt, since they share *IM[7]*.

The *Compare* register can be programmed to generate an interrupt at a particular time, and is continually compared to the *Count* register. Whenever their values equal, the interrupt bit *IP[7]* in the *Cause* register is set. This interrupt bit is reset whenever the *Compare* register is written.

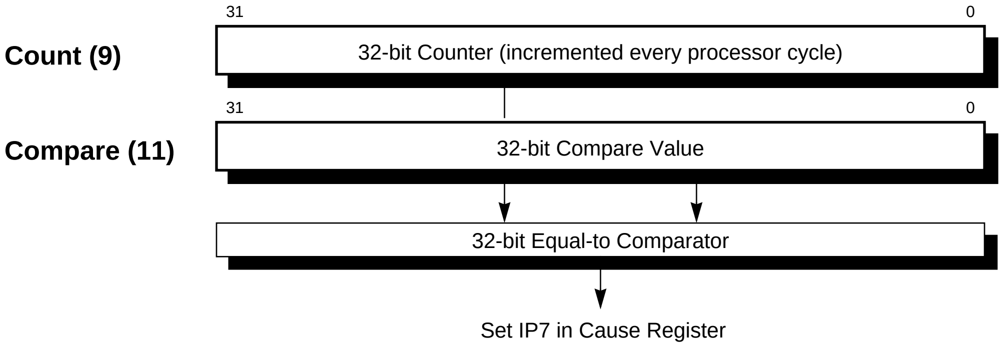

Count (9)
31
0
32-bit Counter (incremented every processor cycle)
Compare (11)
31
0
32-bit Compare Value
32-bit Equal-to Comparator
Set IP7 in Cause Register

Figure 11-9 Count and Compare Registers

11.9 EntryHi Register (10)

The *EntryHi* register holds the high-order bits of a TLB entry for TLB read and write operations.

The *EntryHi* register is accessed by the TLB Probe, TLB Write Random, TLB Write Indexed, and TLB Read Indexed instructions.

Figure 11-10 shows the format of this register and Table 11-8 describes the register's fields.

EntryHi Register

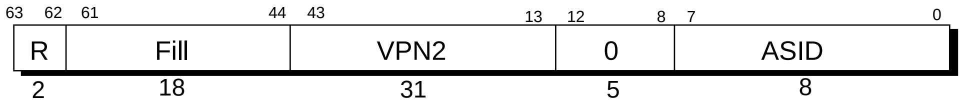


Figure 11-10 EntryHi Register

Table 11-8 EntryHi Register Fields

Field	Description
VPN2	Virtual page number divided by two (maps to two pages); upper bits of the virtual address
ASID	Address space ID field. An 8-bit field that lets multiple processes share the TLB; each process has a distinct mapping of otherwise identical virtual page numbers.
R	Region. (00 → user, 01 → supervisor, 11 → kernel) used to match $\text{vAddr}_{63...62}$
Fill	Reserved. 0 on read; ignored on write.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

In 64-bit addressing mode, the *VPN2* field contains bits 43:13 of the 44-bit virtual address.

In 32-bit addressing mode only the lower 32 bits of the *EntryHi* register are used, so the format remains the same as in the R4400 processor's 32-bit addressing mode. The *FILL* field is ignored on write and read as zeroes, as it was in the R4400 implementation.

When either a TLB refill, TLB invalid, or TLB modified exception occurs, the *EntryHi* register is loaded with the virtual page number (VPN2) and the ASID of the virtual address that did not have a matching TLB entry.

11.10 Status Register (12)

The *Status* register (SR) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. The following list describes the more important *Status* register fields; Figure 11-11 shows the format of the entire register, and Table 11-10 describes the *Status* register fields.

Some of the important fields include:

- The 4-bit *Coprocessor Usability* (*CU*) field controls the usability of 4 possible coprocessors. Regardless of the *CU0* bit setting, CP0 is always usable in Kernel mode. The *XX* bit enables the MIPS IV ISA in User mode.
- By default, the R10000 processor implements the same user instruction set as the R4400 processor. To enable execution of the MIPS IV instructions in User mode, the *MIPS IV User Mode* bit, *(XX*) of the CP0 *Status* register must be set.

  The MIPS IV instruction extension uses COP1X as the opcode; this designation was COP3 in the R4400 processor. For this reason the *CU3* bit is omitted in the R10000 processor, and is used as the *XX* bit. In *Kernel* and *Supervisor* modes, the state of the *XX* bit is ignored, and MIPS IV instructions are always available.

  Mode bit settings are shown in Table 11-9; dashes in the table represent *don't cares*.

Table 11-9 ISA and Status Register Settings for User, Supervisor and Kernel Mode Operations

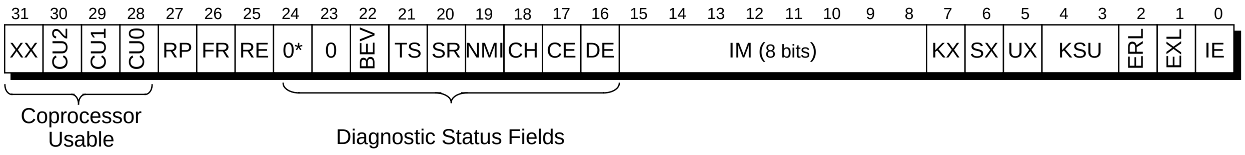
Mode	UX	SX	KX	XX	MIPS II	MIPS III	MIPS IV
User	0	-	-	0	Yes	No	No
	0	-	-	1	Yes	No	Yes
	1	-	-	0	Yes	Yes	No
	1	-	-	1	Yes	Yes	Yes
Supervisor	-	0	-	-	Yes	No	Yes
	-	1	-	-	Yes	Yes	Yes
Kernel	-	-	-	-	Yes	Yes	Yes

NOTE: Operation with the MIPS IV ISA does not assume or require that the MIPS III instruction set or 64-bit addressing be enabled — *KX*, *SX* and *UX* may all be set to zero.

- The *Reduced Power (RP*) bit is reserved and should be zero. The R10000 processor does not define a reduced power mode.
- The *Reverse-Endian (RE)* bit, bit 25, reverses the endianness of the machine. The processor can be configured as either little-endian or big-endian at system reset; reverse-endian selection is available in Kernel and Supervisor modes, and in the User mode when the *RE* bit is 0. Setting the *RE* bit to 1 inverts the User mode endianness.
- The 9-bit *Diagnostic Status* (*DS*) field is used for self-testing, and checks the cache and virtual memory system. This field is described in Table 11-11 and Figure 11-12.
- The 8-bit *Interrupt Mask* (*IM*) field controls the enabling of eight interrupt conditions. Interrupts must be enabled before they can be asserted, and the corresponding bits are set in both the *Interrupt Mask* field of the *Status* register and the *Interrupt Pending* field of the *Cause* register.
- The processor mode is undefined if the *KSU* field is set to 3 (11_2). The R10000 processor implements this as User mode.

Status Register

★


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
XX
CU2
CU1
CU0
RP
FR
RE
0*
0
BEV
TS
SR
NMI
CH
CE
DE
IM (8 bits)
KX
SX
UX
KSU
ERL
EXL
IE
Coprocessor Usable
Diagnostic Status Fields

* For R10000. This bit is used as DSD bit in the R12000.

Figure 11-11 Status Register

Status Register Fields

Table 11-10 describes the *Status* register fields.

Table 11-10 Status Register Fields

Field	Description
XX	Enables execution of MIPS IV instructions in User mode. 1 → MIPS IV instructions usable 0 → MIPS IV instructions unusable
CU	Controls the usability of each of the four coprocessor unit numbers. CP0 is always usable when in Kernel mode, regardless of the setting of the CU_0 bit. 1 → usable 0 → unusable
RP	In the R4400 processor, this bit enables reduced-power operation by reducing the internal clock frequency. In the R10000 processor, this bit should be set to zero.
FR	Enables additional floating-point registers 0 → 16 registers 1 → 32 registers
RE	*Reverse-Endian* bit, valid in User mode.
DS	*Diagnostic Status* field (see Figure 11-12).
IM	*Interrupt Mask*: controls the enabling of each of the external, internal, and software interrupts. An interrupt is taken if interrupts are enabled, and the corresponding bits are set in both the *Interrupt Mask* field of the *Status* register and the *Interrupt Pending* field of the *Cause* register. 0 → disabled 1 → enabled
KX	Enables 64-bit addressing in Kernel mode. The extended-addressing TLB refill exception is used for TLB misses on kernel addresses. 0 → 32-bit 1 → 64-bit
SX	Enables 64-bit addressing and operations in Supervisor mode. The extended-addressing TLB refill exception is used for TLB misses on supervisor addresses. 0 → 32-bit 1 → 64-bit
UX	Enables 64-bit addressing and operations in User mode. The extended-addressing TLB refill exception is used for TLB misses on user addresses. 0 → 32-bit 1 → 64-bit

Table 11-10 (cont.) Status Register Fields

Field	Description
KSU	Mode bits 11_2 → Undefined (implemented as User mode) 10_2 → User 01_2 → Supervisor 00_2 → Kernel
ERL	Error Level; set by the processor when Reset, Soft Reset, NMI, or Cache Error exception are taken. 0 → normal 1 → error
EXL	Exception Level; set by the processor when any exception other than Reset, Soft Reset, NMI, or Cache Error exception are taken. 0 → normal 1 → exception
IE	Interrupt Enable 0 → disable all interrupts 1 → enables all interrupts

Diagnostic Status Field

The 9-bit *Diagnostic Status* (*DS*) field is used for self-testing, and checks the cache and virtual memory system. This field is described in Table 11-11 and shown Figure 11-12.

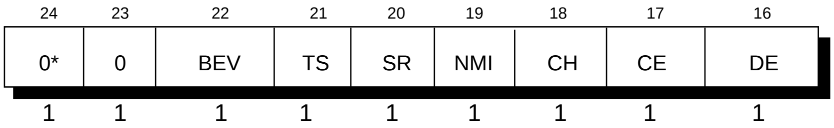
Some of the important *DS* fields include:

- In the R4400, the *TS* bit of the diagnostic field indicates a TLB *shutdown* has occurred due to matching of multiple virtual page entries during address translation. In the R10000 processor, the *TS* bit indicates a TLB write has introduced an entry that would allow matching of more than one virtual page entry during translation. In this case, the TLB entries that allow the multiple matches, even in the *Wired* area, are invalidated before the new TLB entry is written. This prevents multiple matches during address translation.

  The *TS* bit is updated for each TLB write. It can also be read and written by software (in the R4400, the *TS* bit is read-only); to clear the *TS* bit one needs to write a 0 into it. As in the R4400, Reset/Soft Reset/NMI exceptions also clear the *TS* bit.

- The *NMI* bit is new to the R10000 processor; it distinguishes between Soft Reset and NMI exceptions. Both exceptions set the *SR* bit to 1; the NMI exception sets the *NMI* bit to 1, whereas the Soft Reset exception sets it to 0.
- The *CE* bit is reserved in the R10000 processor and should be a 0.

★



* For R10000. This bit is used as DSD bit in the R12000.

Figure 11-12 Diagnostic Status Field

Table 11-11 Status Register Diagnostic Status Bits

★

Bit	Description
DSD	Specifies DSD mode (R12000 only). If this bit is set, the R12000 will not set the Dirty bit for a secondary cache block until the store instruction is the oldest in the Active List and is about to be executed. 0 → normal 1 → delay speculative dirty (fix for speculative store)
BEV	Controls the location of TLB refill and general exception vectors. 0 → normal 1 → bootstrap
TS	This bit is set when a TLB write presents an entry that matches any other virtual page entry in the TLB. Should this occur, any TLB entries that allow multiple matches, even in the *Wired* area, are invalidated before this new entry can be written into the TLB. This prevents multiple matches during address translation. 0 → normal 1 → TLB shutdown has occurred.
SR	1 → Indicates a Soft Reset or NMI exception.
NMI	1 → Indicates a nonmaskable interrupt has occurred. Used to distinguish between a Soft Reset and a nonmaskable interrupt in a Soft Reset exception.
CH	Hit (tag match and valid state) or miss indication for last CACHE Hit Invalidate, Hit Write Back Invalidate for a secondary cache. 0 → miss 1 → hit
CE	Reserved in the R10000, and should be set to 0.
DE	Specifies that cache parity or ECC errors cannot cause exceptions. 0 → parity/ECC remain enabled 1 → disables parity/ECC
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Coprocessor Accessibility

Three *Status* register *CU* bits control coprocessor accessibility: *CU0*, *CU1*, and *CU2* enable coprocessors 0, 1, and 2, respectively. If a coprocessor is unusable, any instruction that accesses it generates an exception.

The following describes the coprocessor implementations and operations on the R10000:

- Coprocessor 0 is always enabled in kernel mode, regardless of the *CU0* bit.
- Coprocessor 1 is the floating-point coprocessor. If *CU1* is 0 (disabled), all floating-point instructions generate a Coprocessor Unusable exception. In MIPS IV, the COP3 instruction is replaced with a second floating-point instruction, COP1X. In addition, new functions are added to COP1 (see **V$_R$5000**, **V$_R$10000 INSTRUCTION User's Manual**). The floating-point branch conditional and compare instructions are expanded to use the eight Floating-Point *Status* register condition bits, instead of the original single bit. If any of these extra bits are referenced (cc > 0) when not using the MIPS IV ISA, an Unimplemented Instruction exception is taken. The integer conditional move (MOVC) instruction tests a floating-point condition bit; it causes a coprocessor unusable exception if coprocessor 1 is disabled.
- Coprocessor 2 is defined, but does not exist in the R10000; its instructions (COP2, LWC2, LDC2, SWC2, SDC2) always cause an exception, but the exception code depends upon whether the coprocessor, as indicated by *CU2*, is enabled.
- Coprocessor 3 has been removed from the MIPS III ISA, and is no longer defined. If MIPS IV is disabled, the coprocessor 3 instruction (COP3) always causes a Reserved Instruction exception.

11.11 Cause Register (13)

The 32-bit read/write *Cause* register describes the cause of the most recent exception.

Figure 11-13 shows the fields of this register; Table 11-12 describes the *Cause* register fields. A 5-bit exception code (*ExcCode*) indicates one of the causes, as listed in Table 11-13.

All bits in the *Cause* register, with the exception of the *IP[1:0]* bits, are read-only; *IP[1:0]* are used for software interrupts.

Table 11-12 Cause Register Fields

Field	Description
BD	Indicates whether the last exception taken occurred in a branch delay slot. 1 → delay slot 0 → normal
CE	Coprocessor unit number referenced when a Coprocessor Unusable exception is taken. This bit is undefined for any other exception.
IP	Indicates an interrupt is pending. This bit remains unchanged for NMI, Soft Reset, and Cache Error exceptions. 1 → interrupt pending 0 → no interrupt
ExcCode	Exception code field (see Table 11-13)
0	Reserved. Must be written as zeroes, and returns zeroes when read.

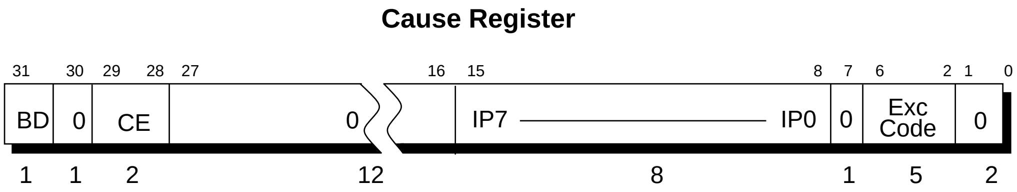


Figure 11-13 Cause Register Format

Table 11-13 Cause Register ExcCode Field

Exception Code Value	**Mnemonic**	**Description**
0	Int	Interrupt
1	Mod	TLB modification exception
2	TLBL	TLB exception (load or instruction fetch)
3	TLBS	TLB exception (store)
4	AdEL	Address error exception (load or instruction fetch)
5	AdES	Address error exception (store)
6	IBE	Bus error exception (instruction fetch)
7	DBE	Bus error exception (data reference: load or store)
8	Sys	Syscall exception
9	Bp	Breakpoint exception
10	RI	Reserved instruction exception
11	CpU	Coprocessor Unusable exception
12	Ov	Arithmetic Overflow exception
13	Tr	Trap exception
14	–	Reserved
15	FPE	Floating-Point exception
16–22	–	Reserved
23	WATCH	Reference to *WatchHi*/*WatchLo* address
24–30	–	Reserved
31	–	Reserved

11.12 Exception Program Counter (14)

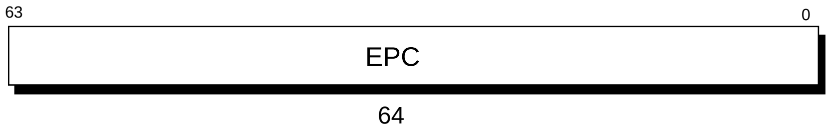
The Exception Program Counter (*EPC*)[†] is a read/write register that contains the address at which processing resumes after an exception has been serviced.

For synchronous exceptions, the *EPC* register contains either:

- the virtual address of the instruction that was the direct cause of the exception, or
- the virtual address of the immediately preceding branch or jump instruction (when the instruction is in a branch delay slot, and the *Branch Delay* bit in the *Cause* register is set).

The processor does not write to the *EPC* register when the *EXL* bit in the *Status* register is set to a 1.

Figure 11-14 shows the format of the *EPC* register.



EPC Register

Figure 11-14 EPC Register Format

[†] The *ErrorEPC* register provides a similar capability, described later in this chapter.

11.13 Processor Revision Identifier (PRId) Register (15)

The 32-bit, read-only *Processor Revision Identifier* (*PRId*) register contains information identifying the implementation and revision level of the CPU and CP0. Figure 11-15 shows the format of the *PRId* register; Table 11-14 describes the *PRId* register fields.

PRId Register

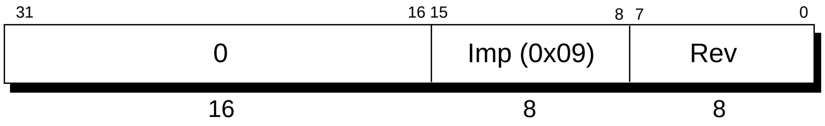


Figure 11-15 Processor Revision Identifier Register Format

Table 11-14 PRId Register Fields

Field	Description
Imp	Implementation number
Rev	Revision number
0	Reserved. Must be written as zeroes, and returns zeroes when read.

The low-order byte (bits 7:0) of the *PRId* register is interpreted as a revision number, and the high-order byte (bits 15:8) is interpreted as an implementation number. The implementation number of the R10000 processor is 0x09. The content of the high-order halfword (bits 31:16) of the register are reserved.

The revision number is stored as a value in the form *y.x*, where *y* is a major revision number in bits 7:4 and *x* is a minor revision number in bits 3:0.

The revision number can distinguish some chip revisions, however there is no guarantee that changes to the chip will necessarily be reflected in the *PRId* register, or that changes to the revision number necessarily reflect real chip changes. For this reason, software should not rely on the revision number in the *PRId* register to characterize the chip.

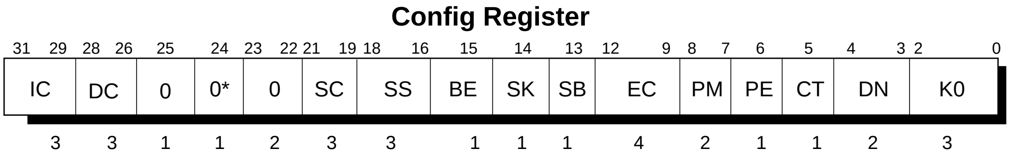
11.14 Config Register (16)

The R10000 processor's *Config* register has a different format from that of the R4400, since the R10000 processor has different mode bits and configurations, however some fields are still compatible: *K0*, *DC*, *IC*, and *BE*. The value of bits 24:0 are taken directly from the Mode bit settings during a reset sequence; refer to Table 8-1 for these bit definitions. Table 11-15 shows the R10000 *Config* register fields, along with values which are hardwired into the register at boot time; Figure 11-16 shows the *Config* register format.

Table 11-15 Config Register Field Definitions

Field	Bits	Name		Hardwired Values
		R10000	**R12000**	
K0	2:0	Coherency algorithm 000_2 → reserved 001_2 → reserved 010_2 → uncached 011_2 → cacheable noncoherent 100_2 → cacheable coherent exclusive 101_2 → cacheable coherent exclusive on write 110_2 → reserved 111_2 → uncached accelerated		
DN	4:3	Device number		
CT	5	CohPrcReqTar		
PE	6	PrcElmReq		
PM	8:7	PrcReqMax		
EC	12:9	SysClkDiv		
SB	13	SCBlkSize		
SK	14	SCCorEn		
BE	15	MemEnd		
SS	18:16	SCSize		
SC	21:19	SCClkDiv		
★	25:22	Reserved	**Field** | **Bit** | **Name** DSD | 24 | Delay Speculative Dirty	0
DC	28:26	Primary data cache size (hardwired to 011_2)		32 Kbytes
IC	31:29	Primary instruction cache size (hardwired to 011_2)		32 Kbytes

★



* For R10000. This bit is used as DSD bit in the R12000.

Figure 11-16 Config Register Format

11.15 Load Linked Address (LLAddr) Register (17)

Physical addresses for Load Link instructions are no longer written into this register. *LLAddr* is implemented as a read/write scratch register used for NT compatibility.

Figure 11-17 shows the format of the *LLAddr* register.

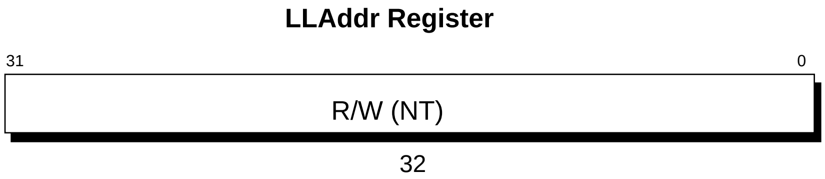


Figure 11-17 LLAddr Register Format

11.16 WatchLo (18) and WatchHi (19) Registers

WatchHi and *WatchLo* are 32-bit read/write registers which contain a physical address of a doubleword location in main memory. If enabled, any attempt to read or write this location causes a Watch exception. This feature is used for debugging.

Bits 7:0 of the *WatchHi* register contain bits 39:32 of the trap physical address, shown in Figure 11-18. The WatchLo register contains physical address bits 31:3. The remaining bits of the register are ignored on write and read as zero.

Table 11-16 describes the *WatchLo* and *WatchHi* register fields.

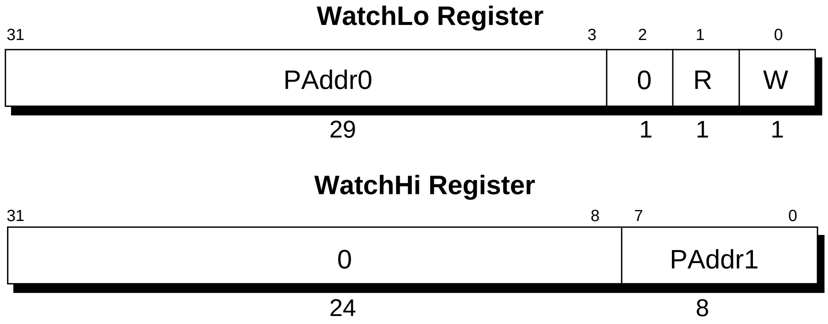


Figure 11-18 WatchLo and WatchHi Register Formats

Table 11-16 WatchHi and WatchLo Register Fields

Field	Description
PAddr1	Bits 39:32 of the physical address
PAddr0	Bits 31:3 of the physical address
R	Trap on load references if set to 1
W	Trap on store references if set to 1
0	Ignored on write and read as zero.

11.17 XContext Register (20)

The read/write *XContext* register contains a pointer to an entry in the page table entry (PTE) array, an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the operating system software loads the TLB with the missing translation from the PTE array. The *XContext* register no longer shares the information provided in the *BadVAddr* register, as it did in the R4400.

The *XContext* register is for use with the XTLB refill handler, which loads TLB entries for references to a 64-bit address space, and is included solely for operating system use. The operating system sets the PTE base field in the register, as needed. Normally, the operating system uses the *Context* register to address the current page map, which resides in the kernel-mapped segment *kseg3*.

Figure 11-19 shows the format of the *XContext* register; Table 11-17 describes the *XContext* register fields.

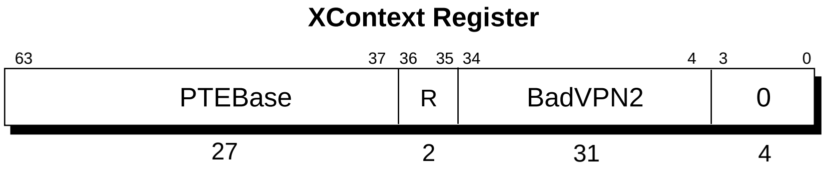


Figure 11-19   XContext Register Format

The 31-bit *BadVPN2* field holds bits 43:13 of the virtual address that caused the TLB miss; bit 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4-Kbyte page size, this format may be used directly to address the pair-table of 8-byte PTEs. For other page and PTE sizes, shifting and masking this value produces the appropriate address.

Table 11-17   XContext Register Fields

Field	Description
BadVPN2	The *Bad Virtual Page Number*/2 field is written by hardware on a miss. It contains the VPN of the most recent invalidly translated virtual address.
R	The *Region* field contains bits 63:62 of the virtual address. 00_2 = user 01_2 = supervisor 11_2 = kernel.
0	Reserved. Must be written as zeroes, and returns zeroes when read.
PTEBase	The *Page Table Entry Base* read/write field is normally written with a value that allows the operating system to use the *Context* register as a pointer into the current PTE array in memory.

11.18 FrameMask Register (21)

The *FrameMask* register is new with the R10000 processor. It masks bits of the *EntryLo0* and *EntryLo1* registers so that these masked bits are not passed to the TLB while doing a TLB write (either TLBWI or TLBWR).

A zero in the *FrameMask* register allows its corresponding bit in the *EntryLo*[*1,0]* registers to pass to the TLB; a one in the *FrameMask* register masks off its corresponding bit in the *EntryLo* registers and passes a zero to the TLB. Bits 15:0 of the *FrameMask* register control bits 33:18 of the *EntryLo* registers.

The remaining bits of this register are ignored on write and read as zeroes. The content of this register is set to zero after a processor reset or a power-up event.

Figure 11-20 shows the *FrameMask* register format.

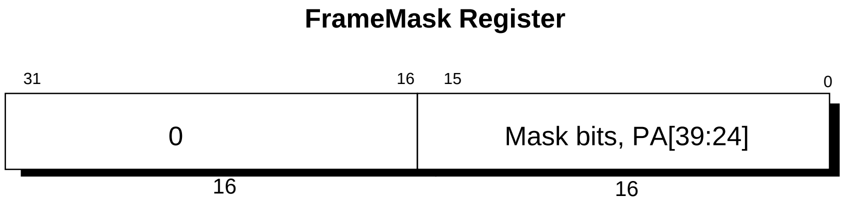


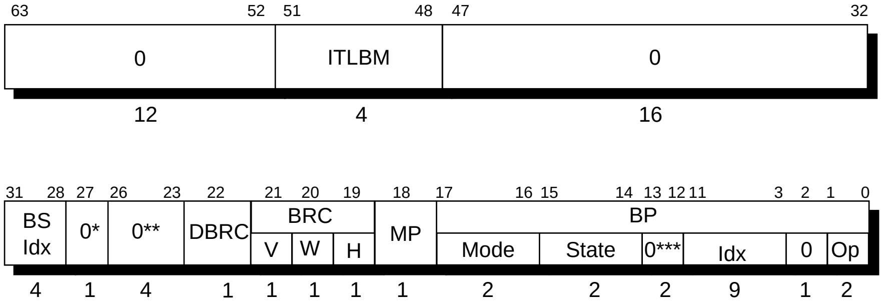
Figure 11-20 FrameMask Register Format

11.19 Diagnostic Register (22)

CP0 register 22, the *Diagnostic* register, is a new 64-bit register for processor-specific diagnostic functions. (Since this register is designed for local use, the diagnostic functions are subject to change without notice.) Currently, this register helps test the ITLB, branch caches, and the branch prediction scheme. In addition, it provides choices for branch prediction algorithms, to help diagnostic program writing.

Figure 11-21 shows the format of the *Diagnostic* register.

★


63
52 51
48 47
32
0
ITLBM
0
12
4
16
31 28 27 26 23 22 21 20 19 18 17 16 15 14 13 12 11 3 2 1 0
BS Idx
0*
0**
DBRC
BRC
V
W
H
MP
BP
Mode
State
0***
Idx
0
Op
4 1 4 1 1 1 1 1 2 2 2 9 1 2

* For R10000. This bit is used as “BTAC disable” bit in the R12000.

** For R10000. This field is used as “ghistory enable” field in the R12000.

*** For R10000. These two bits are used as the high-order bits of the *Idx* field in the R12000.

Figure 11-21 Diagnostic Register Format

Table 11-18 describes the *Diagnostic* register fields.

Table 11-18 Diagnostic Register Fields

Field	**Description**	
	R10000	**R12000**
ITLBM	This field is a 4-bit read-only counter. This field is incremented by one for each ITLB miss, and any overflow is ignored. Its value is undefined during reset, and its value is meaningless when used in an unmapped space.	
BSIdx	This field defines the entry in the branch stack to be used for the latest conditional branch decoded. Its value is meaningless if the latest branch was an unconditional branch.	
DBRC	This field disables the use of the branch return cache (BRC).	
BRCV	This field indicates whether or not the branch return cache (BRC) is valid. BRC has only one entry (four instructions).	
BRCW	This field indicates whether or not the latest branch (JAL, JALR rx, BGEZAL, BGEZALL, BLTZAL, or BLTZALL) caused a write into BRC. It is not affected by any other type of branch.	
BRCH	This field indicates whether or not the latest branch (JR r31 or JALR rx,r31) has a BRC hit. It is not affected by any other type of branch.	
MP	This field indicates whether or not the latest conditional branch verified was mispredicted.	
BPMode	This is a read-write field for branch prediction algorithm control. 00_2 → 2-bit counter scheme 01_2 → All conditional branches are predicted not taken 10_2 → All conditional branches are predicted taken 11_2 → Forward conditional branches are predicted not taken and backward conditional branches are predicted taken. The default mode is 00 on processor reset.	
BPState	This field contains the new 2-bit state for a conditional branch after it is verified. It is also used to hold the 2-bit state to read/write when a branch prediction table read/write operation is executed.	
★ BPIdx	Contains the index to the Branch Prediction Table (BPT) for BPT read/write/initialization operations. This field should contain VA(11:3) of the branch for BPT read/write/initialization operations. The upper six bits of this field contain the line address for BPT line initialization operation; the lower three bits of this field are ignored.	Contains the index to the Branch Prediction Table (BPT) for BPT read/write/initialization operations. This field should contain VA(13:3) of the branch for BPT read/write/initialization operations. The upper eight bits of this field contain the line address for BPT line initialization operation; the lower three bits of this field are ignored.
BPOp	Indicates the following BPT operations: 00_2 → BPT read 01_2 → BPT write 10_2 → Initializes BPT line to all zeroes (strongly not taken) 11_2 → Initializes BPT line to all ones (strongly taken)	
0	Reserved. Must be written as zeroes, and returns zeroes when read.	

In R12000 two fields are added to the *Diagnostic* Register - CP0 Register 22. One field is “ghistory enable”, bits 26:23. The other is “BTAC disable”, bit 27.

The definitions are:

Ghistory enable:

If bit 26 is set, branch prediction uses all eight bits of the global history register. If bit 26 is not set, then bits 25:23 specify a count of the number of bits of global history to be used. Thus if bits 26:23 are all zero, global history is disabled.

The global history contains a record of the taken/not-taken status of recently executed branches, and when used is XOR’ed with the PC of a branch being predicted to produce a hashed value for indexing the BPT. Some programs with small “working set of conditional branches” benefit significantly from the use of such hashing, some see slight performance degradation.

BTAC disable:

If bit 27 is set, the use of the Branch Target Address Cache (BTAC) is disabled. The BTAC is used to reduce the instruction fetch penalty of taken branches by providing the target address of fixed-address branch and jump instructions.

There are two ways to read the branch prediction state from the *Branch Prediction Table* (BPT):

- Place an *mfc0 rx*, *C0_Diag* (a Move From *Diagnostic* register to *GPR rx*) in the delay slot of the conditional branch. This read of the *Diagnostic* register returns the next predicted state from the branch stacks before the *BPT* is updated.
- Move the *Index* and the *BPT read* operation into the *Idx* and *BPOp* field of the *Diagnostic* register. This *mtc0* into *CP0_Diag* graduates as soon as the write is completed; however, there could be a significant delay in transferring the data from *BPT* to *CP0_Diag*. This delay occurs because *C0_Diag* has a lower priority to access the BPT as compared to the accesses by IFETCH and other processes. Thus, the prediction state read from the *C0_Diag* may not reflect the content of the BPT. Use the code sequence shown below to get the correct prediction state from the BPT:

```
        li      rx              # rx has index and BPT read for
                                # Idx and BPOp, respectively.
        mtc0    rx, C0_Diag     # Set the Diagnostic register for reading the BPT
        la      ry, label       # ry !=r31; la could be replaced by a dla for 64-bits
        jr      ry              # This gives priority for C0_Diag to access BPT
label:  mfc0    rz, C0_Diag     # rz holds the state from BPT entry pointed by Idx
```

11.20 Performance Counter Registers (25)

R10000 Implementation

The R10000 processor defines two performance counters and two associated control registers, which are mapped into CP0 register 25. An encoding in the MTC0/MFC0 instructions on register 25 indicates which counter or control register is used.

Each counter is a 32-bit read/write register and is incremented by one each time the countable event, specified in its associated control register, occurs. Each counter can independently count one type of event at a time.

The counter asserts an interrupt, *IP[7]*, when its most significant bit (bit 31) becomes one (the counter overflows) and the associated performance control register enables the interrupt.

The counting continues after counter overflow whether or not an interrupt is signalled.

The format of the control registers are shown in Figure 11-22.

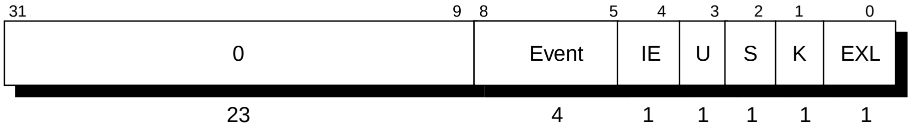


Figure 11-22 Control Register Format (R10000)

The fields of the *Control* register are:

- The *Event* field specifies the event to be counted, listed in Table 11-19.

Table 11-19 Counter Events (R10000)

Event	Counter 0	Counter 1
0	Cycles	Cycles
1	Instructions issued	Instructions graduated
2	Load/prefetch/sync/CacheOp issued	Load/prefetch/sync/CacheOp graduated
3	Stores (including store-conditional) issued	Stores (including store-conditional) graduated
4	Store conditional issued	Store conditional graduated
5	Failed store conditional	Floating-point instructions graduated
6	Branches resolved	Quadwords written back from primary data cache
7	Quadwords written back from secondary cache	TLB refill exceptions
8	Correctable ECC errors on secondary cache data	Branches mispredicted
9	Instruction cache misses	Secondary cache load/store and cache-ops operations
10	Secondary cache misses (instruction)	Secondary cache misses (data)
11	Secondary cache way mispredicted (instruction)	Secondary cache way mispredicted (data)
12	External intervention requests	External intervention request is determined to have hit in secondary cache
13	External invalidate requests	External invalidate request is determined to have hit in secondary cache
14	Functional unit completion cycles	Stores or prefetches with store hint to *CleanExclusive* secondary cache blocks
15	Instructions graduated	Stores or prefetches with store hint to *Shared* secondary cache blocks

NOTE: Note that the updated material reflects the functionality of silicon revision 3.0 and later. The status of earlier silicon revisions are documented as silicon errata available on **www.sgi.com**.

- The *IE* bit enables the assertion of *IP[7]* when the associated counter overflows.
- The *U*, *S*, *K*, and *EXL* bits indicate the processor modes in which the event is counted: U is user mode; S is supervisor mode; K is kernel mode when *EXL* and *ERL* both are set to 0; the system is in kernel mode and handling an exception when *EXL* is set to 1, as shown in Table 11-23.
- *0*: Reserved. Must be written as zeroes, and returns zeroes when read.

These modes can be set individually; for example, one could set all four bits to count a certain event in all processor modes except during a cache error exception.

The performance counters and associated control registers are written by using an MTC0 instruction, as shown in Table 11-20.

Table 11-20 Writing Performance Registers Using MTC0 (R10000)

Opcode[15:11]	Opcode[1:0]	Operation
11001	00	Move to Performance Control 0
11001	01	Move to Performance Counter 0
11001	10	Move to Performance Control 1
11001	11	Move to Performance Counter 1

The performance counters and associated control registers are read by using a MFC0 instruction, as shown in Table 11-21.

Table 11-21 Reading Performance Registers Using MFC0 (R10000)

Opcode[15:11]	Opcode[1:0]	Operation
11001	00	Move from Performance Control 0
11001	01	Move from Performance Counter 0
11001	10	Move from Performance Control 1
11001	11	Move from Performance Counter 1

The format of the performance control registers are shown in Table 11-22.

Table 11-22 Performance Control Register Format (R10000)

Bits	Definition
8:5	Event select
4	IP[7] interrupt enable
3:0	Count enable (U/S/K/EXL)

The count enable field specifies whether counting is to be enabled during User, Supervisor, Kernel, and/or Exception level mode. Any combination of count enable bits may be asserted.

All unused bits in the performance control registers are reserved.

All counting is disabled when the *ERL* bit of the CP0 *Status* register is asserted.

Table 11-23 defines the operation of the count enable bits of the performance control registers.

Table 11-23 Count Enable Bit Definition (R10000)

Count Enable Bit	Count Qualifier (CP0 *Status* Register Fields)
U	KSU = 2 (User mode), EXL = 0, ERL = 0
S	KSU = 1 (Supervisor mode), EXL = 0, ERL = 0
K	KSU = 0 (Kernel mode), EXL = 0, ERL = 0
EXL	EXL = 1, ERL = 0

The following rules apply:

- The performance counter registers may be preloaded with an MTC0 instruction, and counting is enabled by asserting one or more of the count enable bits in the performance control registers.
- The interrupt enable bit must be asserted to cause *IP[7]*.
- To determine the cause of the interrupt, the interrupt handler routine must query the following:
  - the performance counter register
  - the interrupt enable bit of the associated performance control register of both counters
- If neither of the counters caused the interrupt, *IP[7]* must be the result of the CP0 *Count* register matching the CP0 *Compare* register.

Details of Counting Events

In describing the rules that are applied for the counting of each events listed in Table 11-19, following terminology is used:

Done is defined as the point at which the instruction is successfully executed by the functional unit but is not yet graduated.

Graduated is defined as the point in time when the instruction is successfully executed (done), and it is the oldest instruction.

Secondary Cache Transaction Processing (SCTP) logic is on-chip logic in which up to four internally-generated and one-externally generated secondary cache transactions are queued to be processed.

The following rules apply for the counting of each event listed in Table 11-16:

Event 0 for Counter 0 and Counter 1: Cycles

The counter is incremented on each **PClk** cycle.

Event 1 for Counter 0: Instructions Issued

The counter is incremented on each cycle by the sum of the three following events:

- Integer operations marked as *done* on the cycle. 0, 1 or 2 such operations can be marked on each cycle. Since these operations (all except for MUL and DIV) are marked done on the cycle following their being issued to a functional unit, this number is nearly identical to the number issued. The only difference is that re-issues are not counted.
- Floating point operations marked *done* in the active list. Possible values are 0, 1 or 2. Since these operations take more than one cycle to complete, it is possible for an instruction to be issued and then aborted before it is counted, due to a branch-misprediction or exception rollback.
- Load/store instructions first issued to the address calculation unit on the previous cycle. Possible values are 0 or 1. Prefetch instructions are counted as issued. Load/store instructions are counted as being issued only once, even though they may have been issued more than one time.† Any instruction which does not go to the load/store unit, integer functional unit, or FP functional is counted. Some of those not counted are: nops, bc1{f,t,fl,tl}, break, syscall, j, jal, jr, jalr, cp0 instructions.

Event 1 for Counter 1: Instruction Graduation.

The counter is incremented by the number of instructions that were graduated on the previous cycle. When an integer multiply or divide instruction graduates, it is counted as two instructions.

Event 2 for Counter 0: Load/Prefetch/Sync/CacheOp Issue.

Each of these instructions are counted as they are issued. A load instruction is only counted once, even though it may have been issued more than one time.†

Event 2 for Counter 1: Load/Prefetch/Sync/CacheOp Graduation.

Each of these instructions are counted as they are graduated. Up to four loads can graduate in one cycle.

---

† This could be a result of *DCache Tag* being busy or four Instruction or Data cache misses already present and waiting to be processed in the Secondary Cache Transaction Processing (SCTP) logic.

Event 3 for Counter 0: Stores (Including Store-Conditional) Issued.

The counter is incremented on the cycle after a store instruction is issued to the address-calculation unit. Note that a store can only be counted as having been issued once, even though it may actually be issued more than once due to DCache Tag being busy or there already being four load/store cache misses waiting in the SCTP logic.

Event 3 For Counter 1: Store (Including Store-Conditional) Graduation.

Each graduating store (including SC) increments the counter. At most one store can graduate per cycle.

Event 4 for Counter 0: Store-Conditional Issued.

This counter is incremented on the cycle after a store conditional instruction is issued to the address-calculation unit. Note that an SC can only be counted as having been issued once, even though it may actually be issued more than once due to DCache Tag being busy or there already being four load/store cache misses waiting in the SCTP logic.

Event 4 for Counter 1: Store-Conditional Graduation.

At most, one store-conditional can graduate per cycle. This counter is incremented on the cycle following the graduation of a store-conditional instruction.

Event 5 for Counter 0: Failed Store Conditional.

This counter is incremented when a store-conditional instruction fails.

Event 5 for Counter 1: Floating-Point Instruction Graduation.

This counter is incremented by the number of FP instructions which graduated on the previous cycle. Any instruction that sets the FP *Status* register bits (*EVZOUI*) is counted as a graduated floating point instruction. There can be 0 to 4 such instructions each cycle.

Event 6 for Counter 0: Conditional Branch Resolved

This counter is incremented when a conditional branch is determined to have been “resolved.”[†] Note that when multiple floating-point conditional branches are resolved in a single cycle, this counter is still only incremented by one. Although this is a rare event, in this case the count would be incorrect.

[†] In other words, this count is the sum of the conditional branches that are known to be both correctly predicted and mispredicted.

Event 6 for Counter 1: Quadwords Written Back From Primary Data Cache

This counter is incremented once each cycle that a quadword of data is written from primary data cache to secondary cache.

Event 7 for Counter 0: Quadwords Written Back From Secondary Cache

This counter is incremented once each cycle that a quadword of data is written back from the secondary cache to the outgoing buffer located in the on-chip system-interface unit. (Note that data from the outgoing buffer could be invalidated by an external request and not sent out of the processor.)

Event 7 for Counter 1: TLB Refill Exception (Due To TLB Miss)

This counter is incremented on the cycle after the TLB miss handler is invoked. All TLB misses are counted, whether they occur in the native code or within the TLB handler.

Event 8 for Counter 0: Correctable ECC Errors On Secondary Cache Data.

This counter is incremented on the cycle after the correction of a single-bit error on a quadword read from the secondary cache data array.

Event 8 for Counter 1: Branch Misprediction.

This counter is incremented on the cycle after a branch is restored because of misprediction. Note that the misprediction is determined on the same cycle that the conditional branch is resolved. The misprediction rate is the ratio of *branch mispredicted* count to *conditional branch resolve* count.

Event 9 for Counter 0: Primary Instruction Cache Misses.

This counter is incremented one cycle after an instruction refill request is sent to the SCTP logic.

Event 9 for Counter 1: Secondary Cache Load/Store and Cache-ops Operations

This counter is incremented one cycle after a request is entered into the SCTP logic, provided the request was initially targeted at the primary data cache. Such requests fall into three categories:

- primary data cache misses
- requests to change the state of primary and secondary and primary data cache lines from *Clean* to *Dirty*, due to stores hitting a clean line in the primary data cache
- requests initiated by Cache-op instructions

Event 10 for Counter 0: Secondary Cache Misses (Instruction)

This counter is incremented the cycle after the last quadword of a primary instruction cache line is written from the main memory, while the secondary cache refill continues.

Event 10 for Counter 1: Secondary Cache Misses (Data)

This counter is incremented the cycle after the second quadword of a data cache line is written from the main memory, while the secondary cache refill continues.

Event 11 for Counter 0: Secondary Cache Way Misprediction (Instruction)

This counter is incremented when the secondary cache controller begins to retry an access to the secondary cache after it hit in the non-predicted way, provided the secondary cache access was initiated by the primary instruction cache.

Event 11 for Counter 1: Secondary Cache Way Misprediction (Data)

This counter is incremented when the secondary cache controller begins to retry an access to the secondary cache because it hit in the non-predicted way, provided the secondary cache access was initiated by the primary data cache.

Event 12 for Counter 0: External Intervention Requests

This counter is incremented on the cycle after an external intervention request enters the SCTP logic.

Event 12 for Counter 1: External Intervention Requests Hits In Secondary Cache

This counter is incremented on the cycle after an external intervention request is determined to have hit in the secondary cache.

Event 13 for Counter 0: External Invalidate Requests

This counter is incremented on the cycle after an external invalidate request enters the SCTP logic.

Event 13 for Counter 1: External Invalidate Requests Hits In Secondary Cache

This counter is incremented on the cycle after an external invalidate request is determined to have hit in the secondary cache.

Event 14 for Counter 0: Functional Unit Completion Cycles

This counter is incremented once on the cycle after at least one of the functional units — ALU1, ALU2, FPU1, or FPU2 — marks an instruction as done.

Event 14 for Counter 1: Stores, or Prefetches with Store Hint to Clean Exclusive Secondary Cache Blocks.

This counter is incremented on the cycle after a request to change the *Clean Exclusive* state of the targeted secondary cache line to *Dirty Exclusive* is sent to the SCTP logic.

Event 15 for Counter 0: Instruction Graduation.

This counter is incremented by the number of instructions that were graduated on the previous cycle. When an integer multiply or divide instruction graduates, it is counted as two graduated instructions.

Event 15 for Counter 1: Stores or Prefetches with Store Hint to Shared Secondary Cache Blocks.

This counter is incremented on the cycle after a request to change the *Shared* state of the targeted secondary cache line to *Dirty Exclusive* is sent to the SCTP logic.

★ R12000 Implementation

The R12000 processor defines four performance counters and four associated control registers, which are mapped into CP0 register 25. An encoding in the MTC0/MFC0 instructions on register 25 indicates which counter or control register is used.

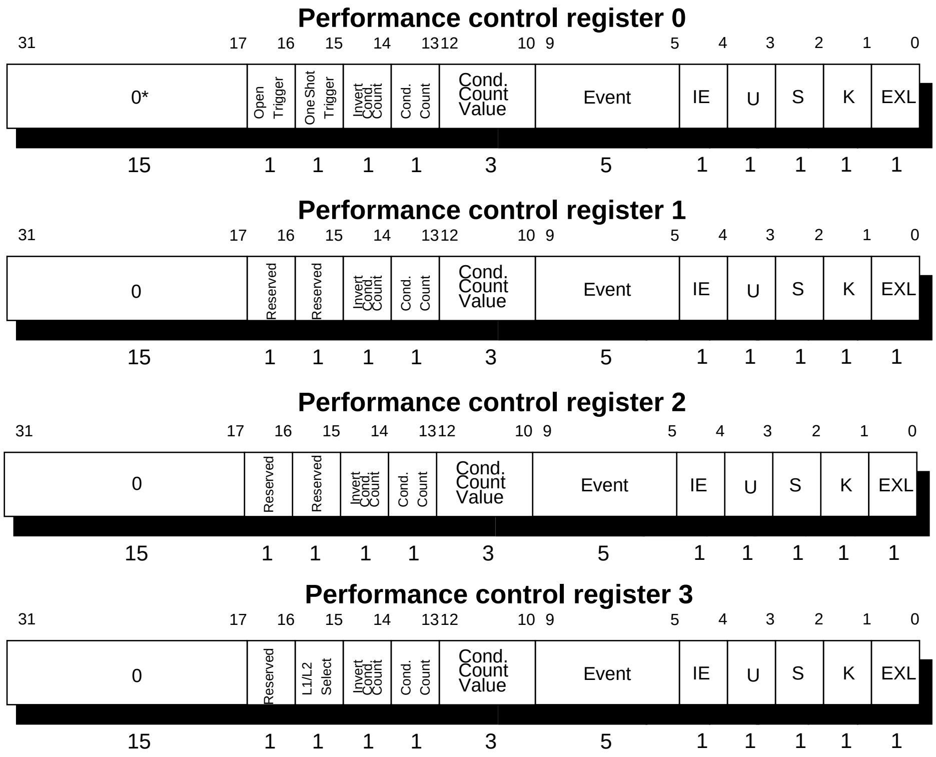
Each counter is a 32-bit read/write register and is incremented by one each time the countable event, specified in its associated control register, occurs. Each counter can independently count one type of event at a time.

The counter asserts an interrupt, *IP[7]*, when its most significant bit (bit 31) becomes one (the counter overflows) and the associated performance control register enables the interrupt.

The counting continues after counter overflow whether or not an interrupt is signalled.

Due to cycle time constraints, events counts are updated 2 cycles later in R12000, compared to similar events in R10000. Also when setting a count mode by writing a performance monitor control register, it is necessary to insert a ‘delay’ instruction between the ‘mtc0 r25’ which does the write, and any initialization of the count register itself.

The format of the control registers are shown in Figure 11-23.


Performance control register 0
31 17 16 15 14 13 12 10 9 5 4 3 2 1 0
0*
Open Trigger
One Shot Trigger
Invert Cond. Count
Cond. Count
Cond. Count Value
Event
IE
U
S
K
EXL
15 1 1 1 1 3 5 1 1 1 1 1
Performance control register 1
31 17 16 15 14 13 12 10 9 5 4 3 2 1 0
0
Reserved
Reserved
Invert Cond. Count
Cond. Count
Cond. Count Value
Event
IE
U
S
K
EXL
15 1 1 1 1 3 5 1 1 1 1 1
Performance control register 2
31 17 16 15 14 13 12 10 9 5 4 3 2 1 0
0
Reserved
Reserved
Invert Cond. Count
Cond. Count
Cond. Count Value
Event
IE
U
S
K
EXL
15 1 1 1 1 3 5 1 1 1 1 1
Performance control register 3
31 17 16 15 14 13 12 10 9 5 4 3 2 1 0
0
Reserved
L1/L2 Select
Invert Cond. Count
Cond. Count
Cond. Count Value
Event
IE
U
S
K
EXL
15 1 1 1 1 3 5 1 1 1 1 1

* For R12000 and R12000L. Bits 31:23 are used as syndrome bits in the R12000A.

Figure 11-23 Control Register Format (R12000)

The fields of the *Control* register are:

- The *Event* field specifies the event to be counted, listed in Table 11-24.

Table 11-24 Counter Events (R12000)

Event	Description
0	Cycles
1	Decoded instructions
2	Decoded loads
3	Decoded stores
4	Miss Handling Table Occupancy
5	Failed store conditional
6	Resolved conditional branches
7	Quadwords written back from secondary cache
8	Correctable ECC errors on secondary cache data
9	Instruction cache misses
10	Secondary cache misses (instruction)
11	Secondary cache way mispredicted (instruction)
12	External intervention requests
13	External invalidate requests
14	Not Used
15	Instructions graduated
16	Executed prefetch instructions
17	Primary data cache misses by prefetch instructions
18	Graduated loads
19	Graduated stores
20	Graduated store conditionals
21	Graduated floating point instructions
22	Quadwords written back from primary data cache
23	TLB misses
24	Mispredicted branches
25	Primary data cache misses
26	Secondary cache misses (data)
27	Misprediction from scache way prediction table (data)
28	State of external intervention hit in secondary cache
29	State of external invalidation hits in secondary cache
30	Store/prefetch exclusive to clean block in secondary cache
31	Store/prefetch exclusive to shared block in secondary cache

- The *IE* bit enables the assertion of *IP[7]* when the associated counter overflows.
- The *U*, *S*, *K*, and *EXL* bits indicate the processor modes in which the event is counted: U is user mode; S is supervisor mode; K is kernel mode when *EXL* and *ERL* both are set to 0; the system is in kernel mode and handling an exception when *EXL* is set to 1, as shown in Table 11-28.
- 0: Reserved. Must be written as zeroes, and returns zeroes when read.

These modes can be set individually; for example, one could set all four bits to count a certain event in all processor modes except during a cache error exception.

The performance counters and associated control registers are written by using an MTC0 instruction, as shown in Table 11-25.

Table 11-25 Writing Performance Registers Using MTC0 (R12000)

Opcode[15:11]	Opcode[2:0]	Operation
11001	000	Move to Performance Control 0
11001	001	Move to Performance Counter 0
11001	010	Move to Performance Control 1
11001	011	Move to Performance Counter 1
11001	100	Move to Performance Control 2
11001	101	Move to Performance Counter 2
11001	110	Move to Performance Control 3
11001	111	Move to Performance Counter 3

The performance counters and associated control registers are read by using a MFC0 instruction, as shown in Table 11-26.

Table 11-26 Reading Performance Registers Using MFC0 (R12000)

Opcode[15:11]	Opcode[2:0]	Operation
11001	000	Move from Performance Control 0
11001	001	Move from Performance Counter 0
11001	010	Move from Performance Control 1
11001	011	Move from Performance Counter 1
11001	100	Move from Performance Control 2
11001	101	Move from Performance Counter 2
11001	110	Move from Performance Control 3
11001	111	Move from Performance Counter 3

The format of the performance control registers are shown in Table 11-27.

Table 11-27 Performance Control Register Format (R12000)

Bits	Definition
16 (Performance control register 0)	Open Trigger
15 (Performance control register 0)	One Shot Trigger
15 (Performance control register 3)	L1/L2 select
14	Invert conditional count
13	Conditional count
12:10	Conditional count value
9:5	Event select
4	IP[7] Interrupt enable
3:0	Count enable (U/S/K/EXL)

The count enable field specifies whether counting is to be enabled during User, Supervisor, Kernel, and/or Exception level mode. Any combination of count enable bits may be asserted.

All unused bits in the performance control registers are reserved.

All counting is disabled when the *ERL* bit of the CP0 *Status* register is asserted.

Table 11-28 defines the operation of the count enable bits of the performance control registers.

Table 11-28 Count Enable Bit Definition (R12000)

Count Enable Bit	Count Qualifier (CP0 Status Register Fields)
U	KSU = 2 (User mode), EXL = 0, ERL = 0
S	KSU = 1 (Supervisor mode), EXL = 0, ERL = 0
K	KSU = 0 (Kernel mode), EXL = 0, ERL = 0
EXL	EXL = 1, ERL = 0

The following rules apply:

- The performance counter registers may be preloaded with an MTC0 instruction, and counting is enabled by asserting one or more of the count enable bits in the performance control registers.
- The interrupt enable bit must be asserted to cause *IP[7]*.
- To determine the cause of the interrupt, the interrupt handler routine must query the following:
  - the performance counter register
  - the interrupt enable bit of the associated performance control register of both counters
- If neither of the counters caused the interrupt, *IP[7]* must be the result of the CP0 *Count* register matching the CP0 *Compare* register.

Counters may each count any of 32 event types

All four counters are able to count any of 32 performance events. Access to these events is provided by extending the ‘event select’ field in the Performance Control Register. In R12000, bits[9:5] of the Performance Control Register specify the event to be counted.

Conditional counting

All four counters can be set to count an event only when the value of that event is equal to, or not equal to, a specific value. This is called ‘conditional’ or ‘inverted conditional’ counting.

- Conditional counting is enabled for a counter by setting bit[13] of the corresponding performance control register.
- Inverted conditional counting is enabled by setting bit[14] of the performance control register.
- If both bits[13] and [14] are set, inverted conditional counting is enabled.
- The value to be compared against is set in bit[12:10] in the performance control register.

If bit[13] is set, and bit[14] is not set, on every cycle the value of the selected event is compared to the value of bits[12:10]. If the two values are equal, then the counter is incremented by 1.

If bit[14] is set, regardless of the state of bit[13], the counter is incremented by 1 if the two values are not equal.

Special case for intervention/invalidate hit events

A special case is used for events 28 and 29, external intervention and invalidate hits. The default of non-conditional counting does not make sense for those events because they encode cache-line state information and so should not be summed as usual. For those two events, the sense of bit [14] is reversed. When a performance control register specifies event 28 or 29, then if bit [14] in the control register is ‘0’, inverted conditional counting is enabled. Similarly, when monitoring events 28 or 29, if bit[14] is ‘1’ then conditional counting is enabled by bit [13], as usual. Thus, for these two event types, the normal ‘default’ of setting control register bits [14:10] to ‘00000’ enables inverted conditional counting with a target value of zero, In this case corresponding count register is incremented by one whenever a non-zero state is seen on the event lines. Such a counter presents a count of ‘generic’ intervention or invalidation hits, since any hit will set the event to a non-zero value, and any miss will leave the value at zero. Consider an example. If a user is interested in obtaining a count of the number of intervention hits to dirty-exclusive cache-lines, then the control register should be set to a value of:

Table 11-29 Performance Control Register 1 Value

Bits	Definition	Value
16	Reserved	0
15	Reserved	0
14	Invert conditional count	1
13	Conditional count	1
12:10	Conditional count value	011
9:5	Event select	11100
4	IP[7] Interrupt enable	0
3:0	Count enable (U/S/K/EXL)	1000

Given this setting, the counter will test the event value on each cycle, and increment on those cycles where the value is equal to ‘011’. Since an invalidate hit to a dirty-exclusive line will set the event to ‘011’, the counter will contain a count of the number of such hits. The default setting of 0’s for all bits [14:10] means that the counter will increment by one on each cycle that a line in any state (except invalid) is hit by an external intervention.

Triggered counting

The operation of monitor register 0 can be selectively tied to that of registers 1 and 2. When bits [15] or [16] in control register 0 are set, then the performance event selected by control register 1is used as a ‘window count’ and the event selected by control register 2 is used as a ‘trigger’. This feature allows a user to set up counter register 0 to correlate the occurrence of different types of events. Because of the generality of the control mechanism, several ways to use this mode result.

Two bits are used to specify this mode because there are two variants that are supported. The operation of the system when bit [15] is set is described in detail, and then the differences for what happens when bit [16] is set are given. When either bit [15] or bit [16] of performance control register 0 is set, the performance counters are said to be in ‘triggering mode’. When in triggering mode, then the event selected by control register 2 is used to enable counting of events in register 0 and 1. When an event monitored by register 2 occurs, bits [15:0] of counter register 1 are reloaded with the last values written into those bits of that register by the execution of an MTC0 instruction. Bit [16] of counter register 1 is set to 1, and bits [31:17] are incremented as if they were a 15-bit unsigned integer. Also, when a'register 2' event occurs, counters 0 and 1 begin counting their respective events. Counting continues until the carry out of the low-order 16 bits of counter 1 causes bit [16] in counter 1 becomes a ‘0’, at which point counting in registers 0 and 1 ceases. Counting restarts in registers 0 and 1 when a new event is seen on register 2.

> **NOTE:** for the purpose of triggering the count mode, the triggering event monitored on register 2 must simply be non-zero. For event types that allow values other than zero or one, counter 2 will continue to count normally, but the window will open only once, and for the same number of counter 1'window' events, regardless of the value on counter 2 which was the trigger. If conditional counting is enabled by setting bits [13] (or [14]) in control register 2, then the triggering event must be equal to (or not equal to) the values set in bits [12:10] of control register 2. In order to use this mode, the user should first load monitor register 1 with a value of (2^{16} - {windowsize}), so that when {windowsize} events have been counted by count register 1, bit [16] of count register

1 will be set and counting will stop. Then the user should load monitor registers 0 and 2 with a value of 0. The three control registers should be set to count the events of interest, and bit [15] should be set in control register 0 at the time that register is written. When controlled by bit [15], trigger mode works in a ‘one shot’ manner. That is, once counting is enabled by a trigger event, any further trigger events are ignored until the window closes because bit 16 of count register 1 has become true. Only after that occurs will a new trigger event cause the window to'reopen' and counting to begin again. If bit [16] is used to arm the trigger mode rather than bit [15], then a slightly different scheme results. In this case, whenever a trigger event occurs, the lower half of count register 1 has its value reset to the initial value, so that the window ‘remains open’, in a sense.

Note that when the performance monitor is used in triggering mode, the sum of trigger events is available in counter 2, and the number of ‘window closings’ is available in the upper half of counter 1. These two values may be different if the event which is used a trigger can potentially take on values > 1 during a single cycle, or if multiple triggers can occur during a window interval.

Data-cache miss-address recording

Access to the miss-address information is provided by using opcode bit [3] in the MTC0 instruction to specify the miss-address register when accessing CP0 register 25. If bit [3] is set in the opcode of an MTC0 instruction which accesses register 25, then the values of opcode bits [2:0] are ignored. When an MTC0-r25 instruction is executed with bit [3] of the opcode set, the address of the most recently-refilled cache miss is transferred from a holding register into performance monitor register 3. From there it can be read via a DMFC0-r25 instruction with bits [0:1] set, so that it refers to performance counter register 3. This “arm then read” sequence is necessary due to implementation constraints. The use of a “double move-from C0” instruction is necessary because 35 bits are retrieved from the address holding register. The miss address holding register itself (unlike the other performance monitor registers) is not writable, and there is no control register associated with this register, either. There is no way to take an interrupt when the register is written or to test the value or than by reading it. This address value in the holding register is updated asynchronously to other operations of the processor, and so it does not necessarily represent the last data miss that was generated. The mechanism that is used closely approximates this. The actual address held in the register is the address of the last line of the primary data cache to be refilled. Bit [15] in performance control register 3 determines whether the address recorded corresponds to any primary cache miss or only those cache accesses which also missed in the Scache. If bit [15] of the control register is set then only those refills corresponding to primary cache misses which also miss in the secondary cache will have their addresses recorded.

Table 11-30 Format of the “Arm Cache-miss Register” Instruction.

Opcode[15:11]	Opcode[3]	Opcode[2:0]	Operation
11001	1	xxx	Move to DCache miss-address control

The address information is 35 bits in size, and corresponds to the address of the cache-line that was refilled. Thus it is a physical address and has a granularity of 32 bytes. The address is output in a split format. The least significant bit returned by the cache-miss register is always zero. Bits [31:1] represent bits [35:5] of the physical address. Bits [51:48] represent bits [39:36] of the physical address.

Syndrome bits

In R12000A, the syndrome bits that are generated from the data coming into the processor from the SCache are captured in a 9-bit register whenever there is a single or multiple bit error. Therefore this register will always contain the syndrome bits generated for the most recent error encountered. The register is uninitialized on power up and is not writable by any other means. Architecturally, the 9-bit register appears as bits 31:23 of the CP0 Performance Counter (Cop 25) Control register 0. These bits were previously unused. These 9 bits are read only bits. A write to this control register will not affect these bits.

The syndrome bits are generated for Secondary to Primary refills and Secondary to Main memory writebacks, but not for CacheOp reads from Secondary cache.

The following are the equations used by the R12000A to generate the 9 syndrome bits:

```
DK0SyndrmP[8] := Parity(
    SCData[127:120] || SCData[113:112] || SCData[111:104] ||
    SCData[103:101] || SCData[97: 96] || SCData[90: 89] ||
    SCData[87: 80] || SCData[74: 72] || SCData[69] ||
    SCData[64] || SCData[62: 61] || SCData[57] ||
    SCData[55: 54] || SCData[52] || SCData[47] ||
    SCData[45: 44] || SCData[39: 38] || SCData[36] ||
    SCData[21] || SCData[13] || SCData[7] ||
    SCDataChk[8]
          );
```

```
DK0SyndrmP[7] := Parity(
    SCData[127:120] || SCData[119:112] || SCData[105:104] ||
    SCData[99: 96] || SCData[95: 88] || SCData[82: 80] ||
    SCData[73: 72] || SCData[71] || SCData[ 69: 68] ||
    SCData[65] || SCData[60] || SCData[51] ||
    SCData[46] || SCData[43] || SCData[39: 38] ||
    SCData[35] || SCData[25] || SCData[23] ||
    SCData[20] || SCData[15] || SCData[12] ||
    SCData[5] ||
    SCDataChk[7]
          );
```

```
DK0SyndrmP[6] := Parity(
        SCData[127] || SCData[121:120] || SCData[119:112] ||
        SCData[111:104] || SCData[97: 96] || SCData[90] ||
        SCData[88] || SCData[81: 80] || SCData[79: 72] ||
        SCData[70] || SCData[68: 66] || SCData[63] ||
        SCData[55] || SCData[50] || SCData[42] ||
        SCData[39: 38] || SCData[34] || SCData[29: 27] ||
        SCData[22] || SCData[19] || SCData[14] ||
        SCData[11] || SCData[6] || SCData[4] ||
        SCDataChk[6]
                );


DK0SyndrmP[5] := Parity(
        SCData[126] || SCData[121] || SCData[119] ||
        SCData[111] || SCData[100] || SCData[97] ||
        SCData[95: 90] || SCData[87: 83] || SCData[81: 80] ||
        SCData[79: 74] || SCData[69: 64] || SCData[63: 58] ||
        SCData[56] || SCData[54: 53] || SCData[49] ||
        SCData[41] || SCData[37] || SCData[33] ||
        SCData[31] || SCData[26] || SCData[24] ||
        SCData[18] || SCData[15: 14] || SCData[10] ||
        SCData[3] ||
        SCDataChk[5]
               );


DK0SyndrmP[4] := Parity(
        SCData[125] || SCData[120] || SCData[118] ||
        SCData[110] || SCData[105:104] || SCData[103: 96] ||
        SCData[95] || SCData[87] || SCData[79] ||
        SCData[74] || SCData[71: 64] || SCData[63: 56] ||
        SCData[53] || SCData[48] || SCData[40] ||
        SCData[32] || SCData[31: 24] || SCData[23: 22] ||
        SCData[17] || SCData[9] || SCData[7] ||
        SCData[2] ||
        SCDataChk[4]
               );
```

```
DK0SyndrmP[3] := Parity(
    SCData[124] || SCData[117] || SCData[113:112] ||
    SCData[109] || SCData[103] || SCData[96] ||
    SCData[94] || SCData[90] || SCData[86] ||
    SCData[78] || SCData[74: 73] || SCData[71] ||
    SCData[69: 64] || SCData[63: 58] || SCData[53: 48] ||
    SCData[47: 46] || SCData[44: 40] || SCData[37: 32] ||
    SCData[30] || SCData[27: 26] || SCData[16] ||
    SCData[8] || SCData[6] || SCData[1] ||
    SCDataChk[3]
        );


DK0SyndrmP[2] := Parity(
    SCData[123] || SCData[121] || SCData[116] ||
    SCData[113] || SCData[108] || SCData[105] ||
    SCData[100: 99] || SCData[93] || SCData[89: 88] ||
    SCData[85] || SCData[77] || SCData[72] ||
    SCData[63] || SCData[61: 59] || SCData[57] ||
    SCData[55: 48] || SCData[47: 46] || SCData[39] ||
    SCData[37] || SCData[31: 30] || SCData[28] ||
    SCData[23: 16] || SCData[15: 8] || SCData[7: 6] ||
    SCData[0] ||
    SCDataChk[2]
        );


DK0SyndrmP[1] := Parity(
    SCData[122] || SCData[115] || SCData[112] ||
    SCData[107] || SCData[104] || SCData[102] ||
    SCData[98] || SCData[92] || SCData[89: 88] ||
    SCData[84] || SCData[81] || SCData[76] ||
    SCData[67] || SCData[62] || SCData[59: 58] ||
    SCData[56] || SCData[55: 54] || SCData[47: 45] ||
    SCData[39: 32] || SCData[31: 29] || SCData[23: 22] ||
    SCData[15: 8] || SCData[7: 0] ||
    SCDataChk[1]
        );
```

```
DK0SyndrmP[0] := Parity(
    SCData[120] || SCData[114] || SCData[106] ||
    SCData[101] || SCData[91] || SCData[89: 88] ||
    SCData[83: 82] || SCData[80] || SCData[75] ||
    SCData[73: 72] || SCData[70] || SCData[66: 64] ||
    SCData[58] || SCData[55: 53] || SCData[47: 40] ||
    SCData[38: 37] || SCData[31: 30] || SCData[25: 24] ||
    SCData[23: 16] || SCData[15: 14] || SCData[7: 0] ||
    SCDataChk[0]
        );
```

Details of Counting Events

In describing the rules that are applied for the counting of each events listed in Table 11-24, following terminology is used:

Done is defined as the point at which the instruction is successfully executed by the functional unit but is not yet graduated.

Graduated is defined as the point in time when the instruction is successfully executed (done), and it is the oldest instruction.

Secondary Cache Transaction Processing (SCTP) logic is on-chip logic in which up to four internally-generated and one-externally generated secondary cache transactions are queued to be processed.

The following rules apply for the counting of each event listed in Table 11-24:

Event 0: Cycles

The counter is incremented on each PClk cycle.

Event 1: Decoded instructions

The counter is incremented by the total number of instructions decoded on the previous cycle. Since decoded instructions may later be killed (for a variety of reasons) this count reflects the overhead due to incorrectly speculated branches and exception processing.

Event 2: Decoded loads

This counter is incremented when a load instruction was decoded on the previous cycle. Prefetch, cache-op and sync instructions are not included in the count of decoded loads.

Event 3: Decoded stores

The counter is incremented if a store instruction was decoded on the previous cycles. Store-conditionals are included in this count.

Event 4: Miss Handling Table Occupancy

This counter is incremented on each cycle by the number of currently valid entries in the Miss Handling Table (MHT). The MHT has five entries. Four entries are used for internally generated accesses; the fifth entry is reserved for externally-generated events. All five entries are included in this count. See event 8 for a related definition.

Event 5: Failed Store Conditional.

This counter is incremented when a store-conditional instruction fails. A failed store-conditional instruction will, in the normal course of events, graduate; so this event represents a subset of the store-conditional instructions counted on event 20 (graduated store-conditionals).

Event 6: Resolved Conditional Branch

This counter is incremented each time a branch is determined to have been mispredicted, and each time a branch is determined to have been correctly predicted. This determination of a branch-prediction’s accuracy is know as the branch being ‘resolved’. This counter correctly reflects the case of multiple FP-conditional branches being resolved in a single cycle.

Event 7: Quadwords Written Back From Secondary Cache

This counter is incremented on each cycle that the data for a quadword is written back from the secondary cache to the outgoing buffer located in the on-chip system-interface unit. (Note that data from the outgoing buffer could be invalidated by an external request and not sent out of the processor.)

Event 8: Correctable ECC Errors On Secondary Cache Data.

This counter is incremented on the cycle following the correction of a single-bit error in a quadword read from the secondary cache data array.

Event 9: Primary Instruction Cache Misses.

This counter is incremented one cycle after an instruction-fetch request is entered into the Miss Handling Table.

Event 10: Secondary Cache Misses (Instruction)

This counter is incremented the cycle after a refill request is sent to the system-interface module of the CPU. This is normally just after the secondary cache tags are checked and a miss is detected, but can be delayed if the system interface module is busy with another request.

Event 11: Secondary Cache Way Misprediction (Instruction)

This counter is incremented when the secondary cache controller begins to retry an access because it hit in the not-predicted way, provided the access that initiated the access was an instruction fetch.

Event 12: External Intervention Requests

This counter is incremented on the cycle after an external intervention request is entered into the Miss Handling Table, provided that the intervention is not an invalidate type.

Event 13: External Invalidate Requests

This counter is incremented on the cycle after an external invalidate request is entered into the Miss Handling Table, provided that the intervention is an invalidate type.

Event 14: Not Used

This counter is not counting any event.

Event 15: Instruction Graduation.

This counter is incremented by the number of instructions that were graduated on the previous cycle. When an integer multiply or divide instruction graduates, it is counted as two graduated instructions.

Event 16: Executed prefetch instructions

This counter is incremented on the cycle after a prefetch instruction does its tag-check, regardless of whether a primary data cache line refill is initiated.

Event 17: Primary data cache misses by prefetch instructions

This counter is incremented on the cycle after a prefetch instruction does its tag-check and a refill of the corresponding primary data cache line is initiated.

Event 18: Graduated loads

This counter is incremented by the number of loads which graduated on the previous cycle. Prefetch instructions are included in this count. Up to four loads can graduate in one cycle.

Event 19: Graduated stores

This counter is incremented on the cycle after a store graduates. At most one store can graduate per cycle. Store-conditional’s are included in this count.

Event 20: Graduated store conditionals

This counter is incremented on the cycle following the graduation of a store-conditional instruction. Both failed and successful store-conditional instructions are included in this count; so successful store-conditionals can be determined as the difference between this event and event 5 (failed store-conditionals).

Event 21: Graduated floating point instructions

This counter is incremented by the number of FP instructions which graduated on the previous cycle. There can be 0 to 4 such instructions.

Event 22: Quadwords written back from primary data cache

This counter is incremented on each cycle that a quadword of data is valid and being written from primary data cache to secondary cache.

Event 23: TLB misses

This counter is incremented on the cycle after the TLB miss handler is invoked.

Event 24: Mispredicted branches

This counter is incremented on the cycle after a branch is “restored” because it was mispredicted.

Event 25: Primary data cache misses

This counter is incremented one cycle after a request is entered into the SCTP logic, provided that the request was initially targeted at the primary data cache. Such requests fall into three categories:

- Primary data cache misses.
- Requests to change the state of secondary and primary data cache lines from clean to dirty (“update” requests) due to stores that hit a clean line in the primary data cache.
- Requests initiated by cache-op instructions.

Event 26: Secondary cache misses (data)

This counter is incremented the cycle after a refill request is sent to the system-interface module of the CPU. This is normally just after the secondary cache tags are checked and a miss is detected, but can be delayed if the system interface module is busy with another request.

Event 27: Misprediction from secondary cache way prediction table (data)

This counter is incremented when the secondary cache control begins to retry an access because it hit in the not-predicted way, provided the access that initiated the access was not an instruction fetch.

Event 28: Store of external intervention hit in secondary cache

This event is set on the cycle after an external intervention is determined to have hit in the secondary cache. The value of the event is equal to the state of the secondary cache line which was hit.

Table 11-31 State of external intervention hit in secondary cache

Event value	State of secondary cache
00	Invalid, no hit seen
01	Clean, Shared
10	Clean, Exclusive
11	Dirty, Exclusive

Setting a performance control register to select this event has a special effect on the conditional-counting behavior. If event 28 or 29 is selected, the sense of the “Negated conditional counting” bit is inverted. See the description of conditional counting for details.

Event 29: State of external invalidation hits in secondary cache

This event is set on the cycle after an external invalidate requests determined to have hit in the secondary cache. It's value is equivalent to that described for event 28.

Event 30: Store/prefetch exclusive to clean block in secondary cache

This counter is incremented on the cycle after an update request is issued for a clean line in the secondary cache.

Event 31: Store/prefetch exclusive to shared block in secondary cache

This counter is incremented on the cycle after an update request is issued for a shared line in the secondary cache.

11.21 ECC Register (26)

The R10000 processor implements a 10-bit read/write *ECC* register which is used to read and write the secondary cache data ECC or the primary cache data parity bits. (Tag ECC and parity are loaded to and stored from the *TagLo* register.) Unlike the R4400, the only CacheOps that use *ECC* register are *Index Load Data* and *Index Store Data*.

In the R4400, both the primary instruction and data caches are parity byte-protected.

In the R10000 processor, the following protection schemes are used:

- The primary instruction cache is word-protected (where one word contains 36 bits), and one parity bit is used for each instruction word (*IP* in Figure 11-24).
- The primary data cache is byte-protected, with four bits used for each 32-bit data word (*DP* in Figure 11-24).
- Each quadword of the secondary cache data uses nine bits of ECC and one bit of parity (*SP* and *ECC* in Figure 11-24).

The primary instruction CacheOps load or store one instruction word at a time; therefore, one bit is used in the *ECC* register. The primary data CacheOps load or store four bytes at a time; therefore, four bits are used in the *ECC* register. The secondary CacheOps use **ECC[9]** as the parity bit and **ECC[8:0]** as the 9-bit ECC. For the *Index Store Data* CacheOps, the unused bits are ignored. For *Index Load Data* CacheOps, the unused a bits are with zeroes.

Figure 11-24 shows the format of the *ECC* register; Table 11-32 describes the register fields.

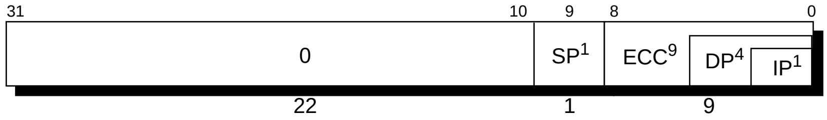


Figure 11-24 ECC Register Format

Table 11-32 ECC Register Fields

Field	Description
SP	A 1-bit field specifying the parity bit read from or written to a secondary cache.
ECC	An 9-bit field specifying the ECC bits read from or written to a secondary cache.
DP	An 4-bit field specifying the parity bits read from or written to a primary data cache.
IP	An 1-bit field specifying the parity bit read from or written to a primary instruction cache.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

11.22 CacheErr Register (27)

The *CacheErr* register is a 32-bit read-only register that handles ECC errors in the secondary cache or system interface, and parity errors in the primary caches.

R10000 processor correction policy is as follows:

- Parity errors cannot be corrected.
- Single-bit ECC errors can be corrected by hardware without taking a Cache Error exception.
- Double-bit ECC errors can be detected but not corrected by hardware.
- All uncorrectable errors take Cache Error exceptions unless the *DE* bit of the *Status* register is set.
- As in the R4400, cache errors are imprecise.

The *CacheErr* register provides cache index and status bits which indicate the source and nature of the error; it is loaded when a Cache Error exception is taken.

CacheErr Register Format for Primary Instruction Cache Errors

Figure 11-25 shows the format of the *CacheErr* register when a primary instruction cache error occurs.

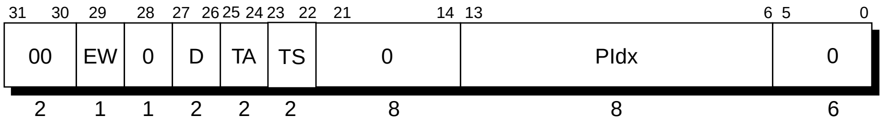


Figure 11-25 CacheErr Register Format for Primary Instruction Cache Errors

EW: set when *CacheErr* register is already holding the values of a previous error

D: data array error (way1‖ way0)

TA: tag address array error (way1‖ way0)

TS: tag state array error (way1‖ way0)

PIdx: primary cache virtual block index, **VA[13:6]**

0: Reserved. Must be written as zeroes, and returns zeroes when read.

CacheErr Register Format for Primary Data Cache Errors

Figure 11-26 shows the format of the *CacheErr* register when a primary data cache error occurs.

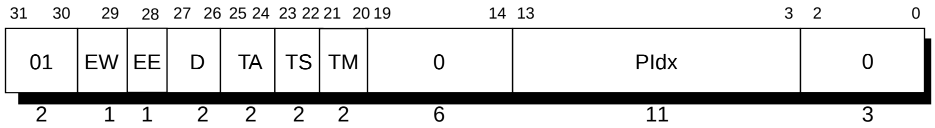

31
30
29
28
27
26
25
24
23
22
21
20
19
14
13
3
2
0
01
EW
EE
D
TA
TS
TM
0
PIdx
0
2
1
1
2
2
2
2
6
11
3

Figure 11-26 CacheErr Register Format for Primary Data Cache Errors

EW: set when *CacheErr* register is already holding the values of a previous error

EE: tag error on an inconsistent block

D: data array error (way1 || way0)

TA: tag address array error (way1 || way0)

TS: tag state array error (way1 || way0)

TM: tag mod array error (way1 || way0)

PIdx: primary cache virtual double word index, **VA[13:6]**

0: Reserved. Must be written as zeroes, and returns zeroes when read.

CacheErr Register Format for Secondary Cache Errors

Figure 11-27 shows the format of the *CacheErr* register when a secondary cache error occurs.

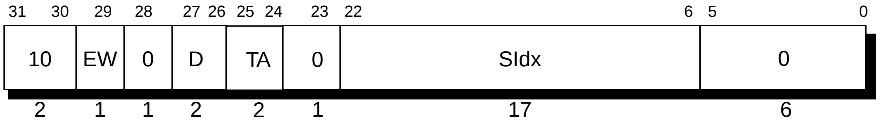


Figure 11-27 CacheErr Register Format for Secondary Cache Errors

EW: set when *CacheErr* register is already holding the values of a previous error

D: uncorrectable data array error (way1 || way0)

TA: uncorrectable tag array error (way1 || way0)

SIdx: secondary cache physical block index (**PA[22:6]** for 16-word block size or **PA[22:7]** for 32-word block size)

0: Reserved. Must be written as zeroes, and returns zeroes when read.

CacheErr Register Format for System Interface Errors

Figure 11-28 shows the format of the *CacheErr* register when a System interface error occurs.

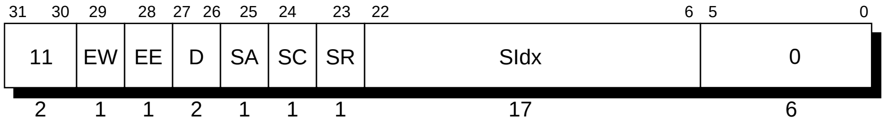


Figure 11-28 CacheErr Register Format for System Interface Errors

EW: set when *CacheErr* register is already holding the values of a previous error

EE: data error on a *CleanExclusive* or *DirtyExclusive*

D: uncorrectable system block data response error (way1 || way0)

SA: uncorrectable system address bus error

*S*C: uncorrectable system command bus error

*S*R: uncorrectable system response bus error

SIdx: secondary cache physical block index

0: Reserved. Must be written as zeroes, and returns zeroes when read.

11.23 TagLo (28) and TagHi (29) Registers

The *TagHi* and *TagLo* registers are 32-bit read/write registers used to hold the following:†

- the primary cache tag and parity
- the secondary cache tag and ECC
- the data in primary or secondary caches for certain CacheOps

TagHi/Lo formats in the R10000 processor differ from those in the R4400 due to changes in CacheOps and cache architecture. R10000 formats depend on the type of CacheOp executed and the cache to which it is applied. The reserved fields are read as zeroes after executing an *Index Load Tag* or an *Index Load Data* CacheOp and ignored when executing an *Index Store Tag* or an *Index Store Data* CacheOp.

To ensure NT kernel compatibility, the *TagLo* register is implemented as a 32-bit read/write register. The value written by an MTC0 instruction can be retrieved by a MFC0 instruction, unless an intervening CACHE instruction has modified the content.

This section gives the TagLo and TagHi register formats for the following CacheOp and cache combinations:

- CacheOp is Index Load/Store Tag
    - primary instruction cache operation
    - primary data cache operation
    - secondary cache operation
- CacheOp is Index Load/Store Data
    - primary instruction cache operation
    - primary data cache operation
    - secondary cache operation

CacheOp is Index Load/Store Tag

This section describes the three states of the *TagLo* and *TagHi* registers, when the CacheOp is an *Index Load/Store Tag* for the following operations:

- primary instruction cache operation
- primary data cache operation
- secondary cache operation

---

† To ensure NT kernel compatibility, the TagLo register is implemented as a 32-bit read/write register. The value written by a MTC0 instruction can be retrieved by a MFC0 instruction, unless intervening CACHE instructions modify the content.

Primary Instruction Cache Operation

If the CacheOp is an *Index Load/Store Tag* for a primary instruction cache operation, the fields of the *TagHi* and *TagLo* registers are defined as follows:

PTag0: contains physical address bits [35:12] stored in the cache tag

PState: contains the primary instruction cache state for the line, as follows:

1 = *Valid*

0 = *Invalid*

LRU: indicates which way is the least recently used of the set.

SP: state even parity bit for the *PState* field

TP: tag even parity bit.

PTag1: contains physical address bits [39:36] stored in the cache tag

0: Reserved. Must be written as zeroes, and returns zeroes when read.

Figure 11-29 shows the fields of the *TagHi* and *TagLo* registers.

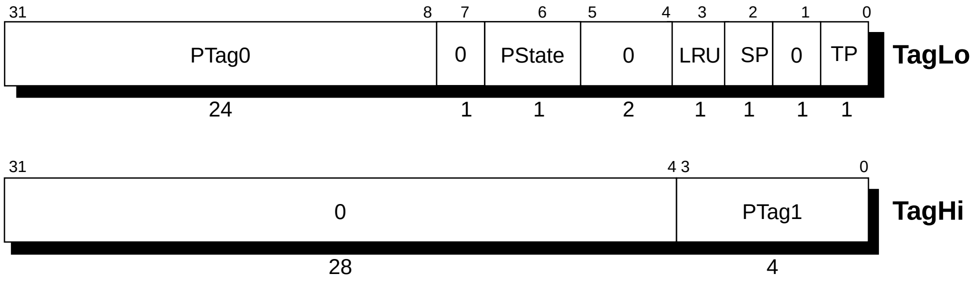


Figure 11-29 TagHi/Lo Register Fields in Primary Instruction Cache When CacheOp is Index Load/Store Tag

Primary Data Cache Operation

If the CacheOp is an *Index Load/Store Tag* for primary data cache operations, the fields of the *TagHi* and *TagLo* registers are defined as follows:

State Modifier: holds the status of the line, as follows:

001_2 = neither refilled or written

010_2 = this line may have been written and inconsistent from the secondary cache (*W* bit)

100_2 = this line is being refilled (*Refill* bit).

PTag1: contains physical address bits [39:36] stored in the cache tag

PTag0: contains physical address bits [35:12] stored in the cache tag

PState: together with the *Refill* bit of the *State Modifier* in the *TagHi* register, *PState* determines the state of the cache block in the primary data cache, as shown in Table 11-33.

Table 11-33 PState Field Definition in TagHi/Lo Registers, For Primary Data Cache Operation When CacheOp is Index Load/Store Tag

PState	Refill=0	Refill=1
$\mathbf{00}_2$	Invalid	Refill *clean* (block is being refilled)
$\mathbf{01}_2$	Shared	Upgrade Share (converting *shared* to *dirty*)
$\mathbf{10}_2$	Clean Exclusive	Upgrade Clean (converting *clean* to *dirty*).
$\mathbf{11}_2$	Dirty Exclusive	Refill *dirty* (block is being refilled for a store)

LRU: indicates which way is the least recently used of the set.

SP: state even parity bit for the *PState* field and the *Way* bit

Way: indicates which secondary cache set contains the primary cache line for this tag

TP: tag even parity bit.

0: Reserved. Must be written as zeroes, and returns zeroes when read.

Figure 11-30 shows the fields of the *TagHi* and *TagLo* registers.

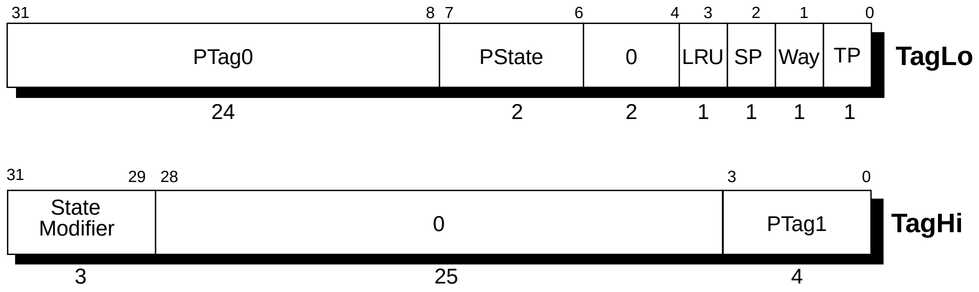


Figure 11-30 TagHi/Lo Register Fields in Primary Data Cache When CacheOp is Index Load/Store Tag

Secondary Cache Operation

If the CacheOp is an *Index Load/Store Tag* for secondary cache operations, the fields of the *TagHi* and *TagLo* registers are defined as follows:

STag0: contains physical address bits [35:18] stored in the cache tag

SState: contains the secondary cache state of the line, *as follows:*

00_2 = *Invalid*

01_2 = *Shared*

10_2 = *Clean Exclusive*

11_2 = *Dirty Exclusive*

VIndex (virtual index): contains only two bits of significance since the32 Kbyte 2-way set associative primary caches are addressed using only two untranslated address bits (**VA[13:12]**) plus the offset within the virtual page.

ECC: contains the ECC for the *STag*, *SState* and *VIndex* fields.

MRU: indicates which way was the most recently used in the set.

STag1: contains the physical address bits [39:36] stored in the cache tag.

0: Reserved.   Must be written as zeroes, and returns zeroes when read.

Figure 11-31 shows the fields of the *TagHi* and *TagLo* registers.

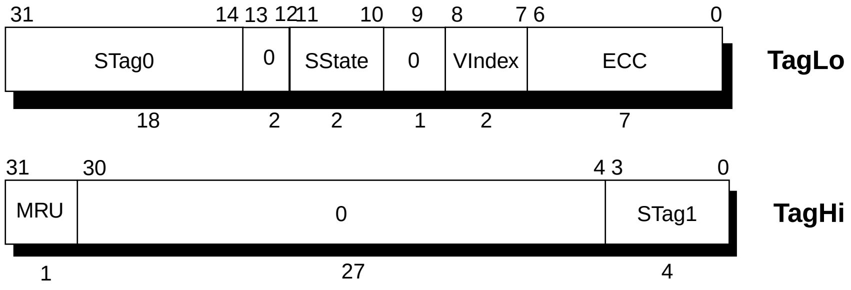


Figure 11-31   TagHi/Lo Register Fields in Secondary Cache When CacheOp is Index Load/Store Tag

CacheOp is Index Load/Store Data

This section describes the following three states of the *TagLo* and *TagHi* registers, when the CacheOp is an *Index Load/Store Data*:

- primary instruction cache operation
- primary data cache operation
- secondary cache operation

Primary Instruction Cache Operation

If the CacheOp is an *Index Load/Store Data* for the primary instruction cache, the *TagHi* register stores the most significant four bits of a 36-bit instruction, as shown in Figure 11-32; the rest of the instruction is stored in the *TagLo* register.

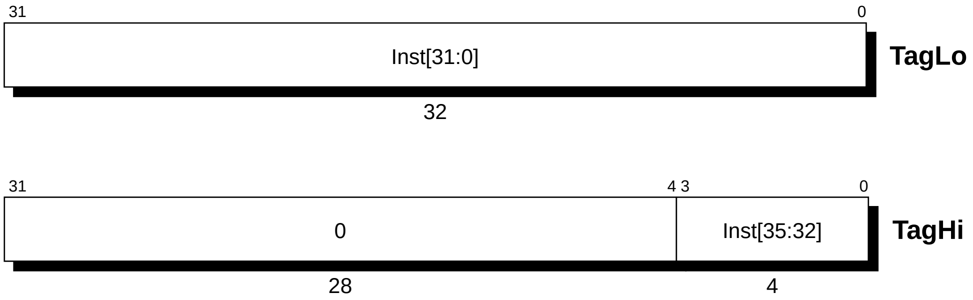


Figure 11-32 TagHi/Lo Register Fields in Primary Instruction Cache When CacheOp is Index Load/Store Data

0: Reserved. Must be written as zeroes, and returns zeroes when read.

Primary Data Cache Operation

If the CacheOp is *Index Load/Store Data* for primary data cache, the *TagHi* register is not used. The *TagLo* registers contains a 32-bit data word for the cache operation, as shown in Figure 11-33.

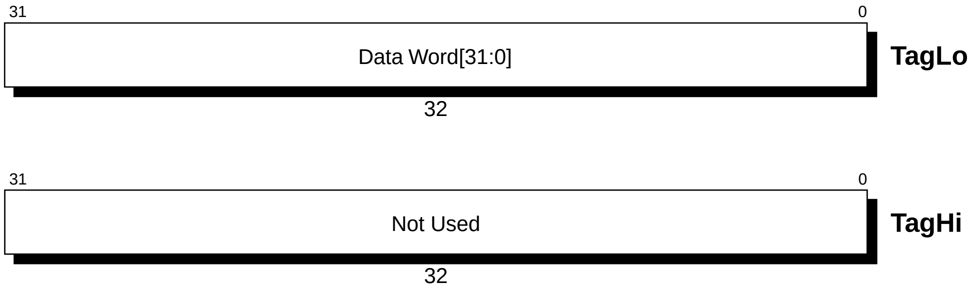

31
0
Data Word[31:0]
TagLo
32
31
0
Not Used
TagHi
32

Figure 11-33 TagHi/Lo Register Fields in Primary Data Cache When CacheOp is Index Load/Store Data

Secondary Cache Operation

If the CacheOp is *Index Load/Store Data* for the secondary cache, a doubleword of data is required for the CacheOp. The *TagHi* register stores the upper 32 bits of the doubleword and the *TagLo* register stores the lower 32 bits, as shown below in Figure 11-34.

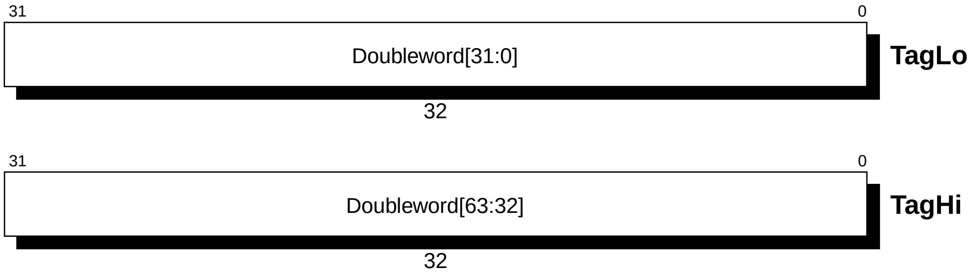

31
0
Doubleword[31:0]
TagLo
32
31
0
Doubleword[63:32]
TagHi
32

Figure 11-34 TagHi/Lo Register Fields in Secondary Cache When CacheOp is Index Load/Store Data

11.24 ErrorEPC Register (30)

The *ErrorEPC* register is similar to the *EPC* register, except that *ErrorEPC* is used on ECC and parity error exceptions. It is also used to store the program counter (PC) on Reset, Soft Reset, and nonmaskable interrupt (NMI) exceptions.

The read/write *ErrorEPC* register contains the virtual address at which instruction processing can resume after servicing an error. Figure 11-35 shows the format of the *ErrorEPC* register.

ErrorEPC Register

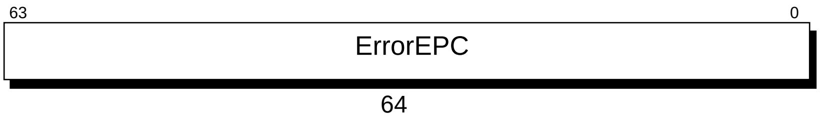


Figure 11-35 ErrorEPC Register Format

12. Floating-Point Unit

This section describes the operation of the FPU, including the register definitions.

The Floating-Point unit consists of the following functional units:

- add unit
- multiply unit
- divide unit
- square-root unit

The **add unit** performs floating-point add and subtract, compare, and conversion operations. Except for Convert Integer To Single-Precision (float), all operations have a 2-cycle latency and a 1-cycle repeat rate.

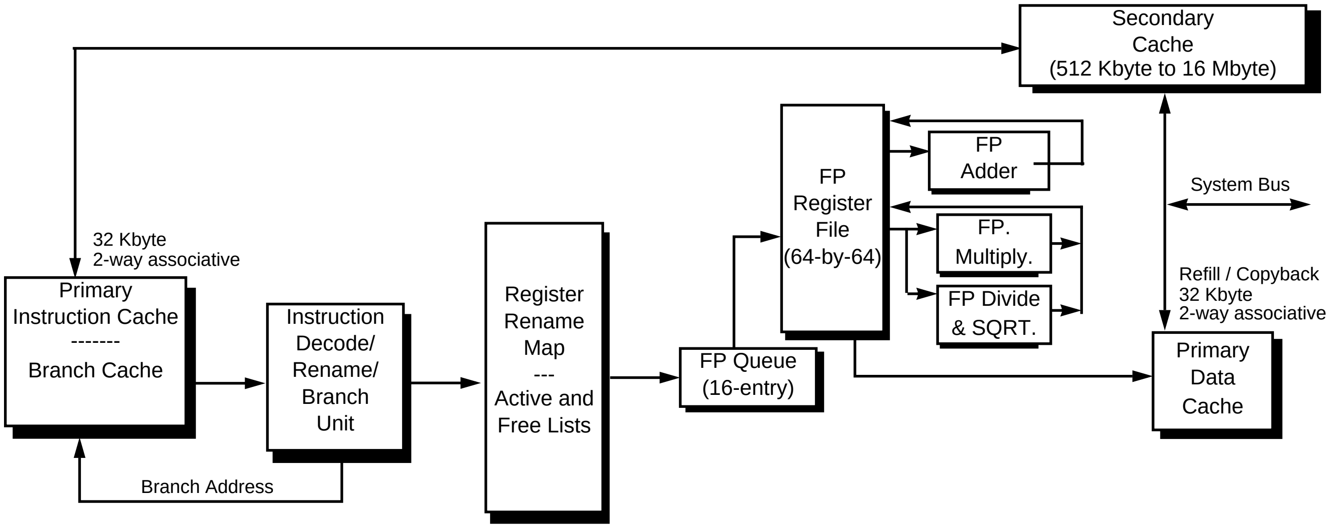
The **multiply unit** performs single-precision or double-precision multiplication with a 2-cycle latency and a 1-cycle repeat rate.

The **divide and square-root units** do single- or double-precision operations. They have long latencies and low repeat rates (20 to 40 cycles).

12.1 Floating-Point Unit Operations

The floating-point add, multiply, divide, and square-root units read their operands and store their results in the floating-point register file. Values are loaded to or stored from the register file by the load/store and move units.

A logic diagram of floating-point operations is shown in Figure 12-1, in which data and instructions are read from the secondary cache into the primary caches, and then into the processor. There they are decoded and appended to the floating-point queue, passed into the FP register file where each is dynamically issued to the appropriate functional unit. After execution in the functional unit, results are stored, through the register file, in the primary data cache.


Secondary
Cache
(512 Kbyte to 16 Mbyte)
FP
Register
File
(64-by-64)
FP
Adder
FP.
Multiply.
FP Divide
& SQRT.
System Bus
32 Kbyte
2-way associative
Primary
Instruction Cache

Branch Cache
Instruction
Decode/
Rename/
Branch
Unit
Register
Rename
Map

Active and
Free Lists
FP Queue
(16-entry)
Refill / Copyback
32 Kbyte
2-way associative
Primary
Data
Cache
Branch Address

Figure 12-1 Logical Diagram of FP Operations

The floating-point queue can issue one instruction to the adder unit and one instruction to the multiplier unit. The adder and multiplier each have two dedicated read ports and a dedicated write port in the floating-point register file.

Because of their low repeat rates, the divide and square-root units do not have their own issue port. Instead, they decode instructions issued to the multiplier unit, using its operand registers and bypass logic. They appropriate a second cycle later for storing their result.

When an instruction is issued, up to two operands are read from dedicated read ports in the floating-point register file. After the operation has been completed, the result can be written back into the register file using a dedicated write port. For the add and multiply units, this write occurs four cycles after its operands were read.

12.2 Floating-Point Unit Control

The control of floating-point execution is shared by the following units:

- The floating-point queue determines operand dependencies and dynamically issues instructions to the execution units. It also controls the destination registers and register bypass.
- The execution units control the arithmetic operations and generate status.
- The graduate unit saves the status until the instructions graduate, and then it updates the *Floating-Point Status* register.

Eliminate traps for Denorm/NaN FP inputs (R12000)

The R10000 currently takes Unimplemented Exception when an FPU gets a NaN or Denorm as an input. R12000 suppresses these traps whenever the FS bit is set in the FCSR (ref. **V$_{R}$5000**, **V$_{R}$10000 INSTRUCTION User's Manual**). R12000 simply passes through NaN's and Denorm's when the bit is set. This change in no way affects the handling of QNaNs and Denorms when they are produced, it only changes the way they are handled when they are received as input operands.

Case of Denorm when the FS bit is set to 1: A Denorm received as an input to the FP unit is flushed to zero before the FP unit begins to process the operand. The behavior of the unit (when FS is 1) will be exactly that seen when the input is zero. Specifically, if the zero input would itself cause a trap (due to divide by zero, for example) then the that zero-generated trap will be taken.

When a Denorm is seen at the input, the Inexact bit is set, except in the cases described below:

The Inexact bit will not be set, even if FS=1 and a Denorm is seen on input, if the other input to the FP operation is a value which pre-determines the FP result (e.g. QNaN). When the result is not affected by the presence or absence of the Denorm input, the result is EXACT. Hence the Inexact bit should not be set, even if Flush to Zero mode is ON.

Case of QNaNs when the FS bit is set to 1: A QNaN received as an input operand for an FP unit will cause the unit to produce the standard QNaN (which is not necessarily same as the input QNaN). Note that FP units will not propagate the QNaN to the output, but will always produce the same, standard, QNaN.

When the FS bit is set to zero, the behavior will be exactly as in R10000.

12.3 Floating-Point General Registers (FGRs)

The Floating-Point Unit is the hardware implementation of Coprocessor 1 in the MIPS IV Instruction Set Architecture. The MIPS IV ISA defines 32 logical floating-point general registers (FGRs), as shown in Figure 12-2. Each FGR is 64 bits wide and can hold either 32-bit single-precision or 64-bit double-precision values. The hardware actually contains 64 physical 64-bit registers in the Floating-Point Register File, from which the 32 logical registers are taken.

FP instructions use a 5-bit logical number to select an individual FGR. These logical numbers are mapped to physical registers by the rename unit (in pipeline stage 2), before the Floating-Point Unit executes them. Physical registers are selected using 6-bit addresses.

32- and 64-Bit Operations

The *FR* bit (26) in the *Status* register determines the number of logical floating-point registers available to the program, and it alters the operation of single-precision load/store instructions, as shown in Figure 12-2.

- *FR* is reset to 0 for compatibility with earlier MIPS I and MIPS II ISAs, and instructions use only the 16 physical even-numbered floating-point registers (32 logical registers). Each logical register is 32 bits wide.
- *FR* is set to 1 for normal MIPS III and MIPS IV operations, and all 32 of the 64-bit logical registers are available.

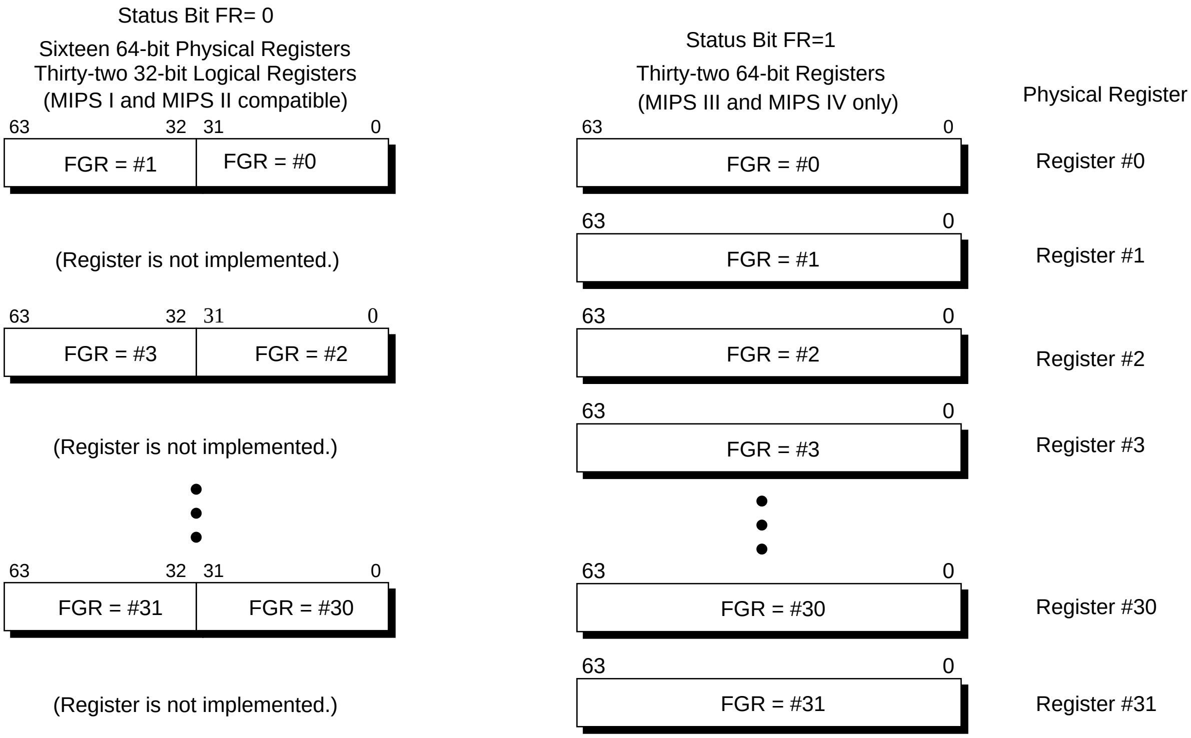

Status Bit FR= 0
Sixteen 64-bit Physical Registers
Thirty-two 32-bit Logical Registers
(MIPS I and MIPS II compatible)
Status Bit FR=1
Thirty-two 64-bit Registers
(MIPS III and MIPS IV only)
Physical Register
63
32
31
0
FGR = #1
FGR = #0
FGR = #0
Register #0
(Register is not implemented.)
FGR = #1
Register #1
FGR = #3
FGR = #2
FGR = #2
Register #2
(Register is not implemented.)
FGR = #3
Register #3
FGR = #31
FGR = #30
FGR = #30
Register #30
(Register is not implemented.)
FGR = #31
Register #31

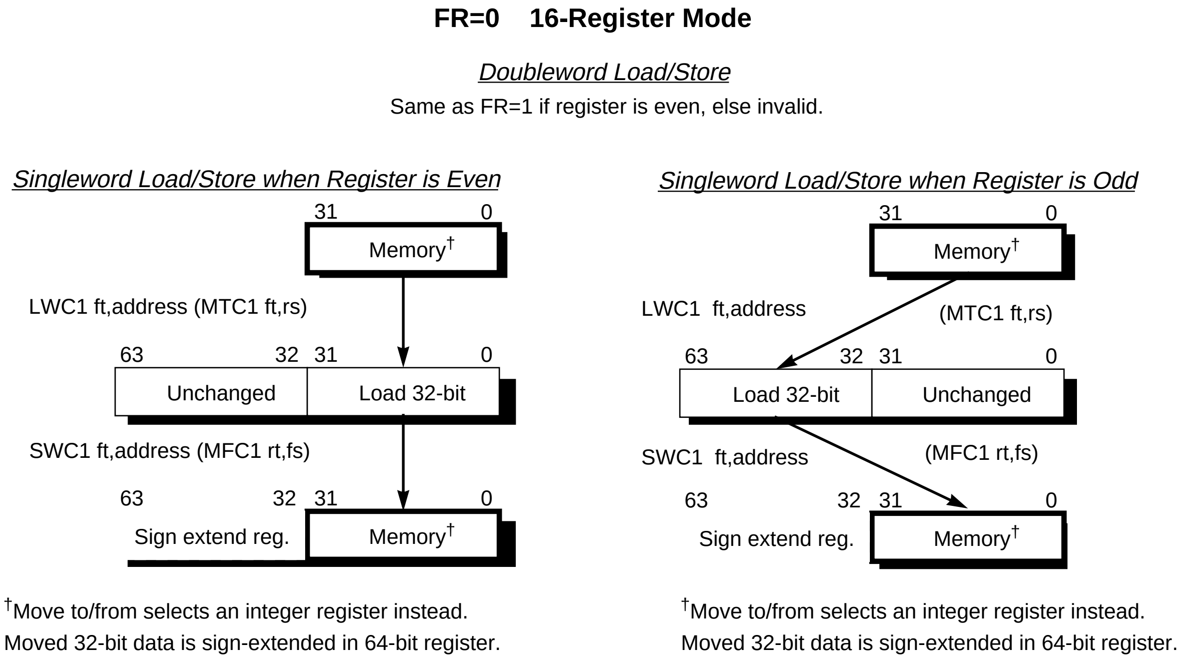
Figure 12-2 Floating-Point Registers

Load and Store Operations

When FR = 0, floating-point load and stores operate as follows:

- A doubleword load or store is handled the same as if the FR bit was set to 1, as long as the register selected is even (0, 2, 4, etc.).
- If the register selected is odd, the load/store is invalid.

These operations are shown in Figure 12-3. Singleword loads/stores to even and odd registers are also shown.


FR=0 16-Register Mode
Doubleword Load/Store
Same as FR=1 if register is even, else invalid.
Singleword Load/Store when Register is Even
31
0
Memory†
LWC1 ft,address (MTC1 ft,rs)
63
32
31
0
Unchanged
Load 32-bit
SWC1 ft,address (MFC1 rt,fs)
63
32
31
0
Sign extend reg.
Memory†
Singleword Load/Store when Register is Odd
31
0
Memory†
LWC1 ft,address
(MTC1 ft,rs)
63
32
31
0
Load 32-bit
Unchanged
SWC1 ft,address
(MFC1 rt,fs)
63
32
31
0
Sign extend reg.
Memory†

†Move to/from selects an integer register instead.
Moved 32-bit data is sign-extended in 64-bit register.

†Move to/from selects an integer register instead.
Moved 32-bit data is sign-extended in 64-bit register.

Figure 12-3 Loading and Storing Floating-Point Registers in 16-Register Mode

NOTE: Move (MOV) and conditional move (MOVC, MOVN, MOVZ are included in these arithmetic operations, although no arithmetic is actually performed.

When FR = 1, floating-point load and stores operate as follows:

- Single-precision operands are read from the low half of a register, leaving the upper half ignored. Single-precision results are written into the low half of the register. The high half of the result register is architecturally undefined; in the R10000 implementation, it is set to zero.
- Double-precision arithmetic operations use the entire 64-bit contents of each operand or result register.

Because of register renaming, every new result is written into a temporary register, and conditional move instructions select between a new operand and the previous old value. The high half of the destination register of a single-precision conditional move instruction is undefined (shown in Figure 12-5), even if no move occurs.

Singleword and doubleword loads and stores with the FPU in 32-register mode (FR=1) are shown in Figure 12-4.

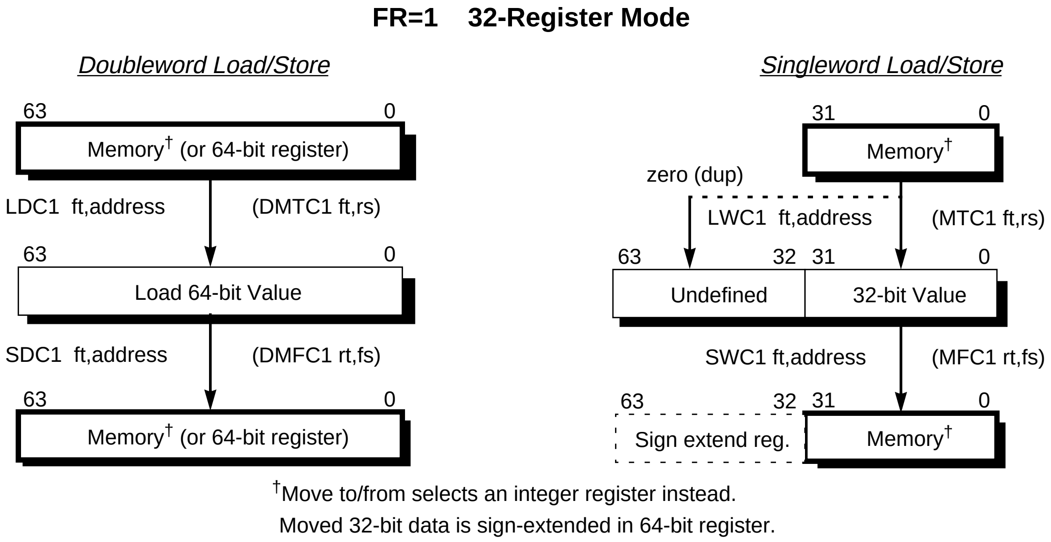

FR=1 32-Register Mode
Doubleword Load/Store
63
0
Memory† (or 64-bit register)
LDC1 ft,address
(DMTC1 ft,rs)
63
0
Load 64-bit Value
SDC1 ft,address
(DMFC1 rt,fs)
63
0
Memory† (or 64-bit register)
Singleword Load/Store
31
0
Memory†
zero (dup)
LWC1 ft,address
(MTC1 ft,rs)
63
32
31
0
Undefined
32-bit Value
SWC1 ft,address
(MFC1 rt,fs)
63
32
31
0
Sign extend reg.
Memory†
†Move to/from selects an integer register instead.
Moved 32-bit data is sign-extended in 64-bit register.

Figure 12-4 Loading and Storing Floating-Point Registers in 32-Register Mode

Doubleword load, store and move to/from instructions load or store an entire 64-bit floating-point register, as shown in Figure 12-5.

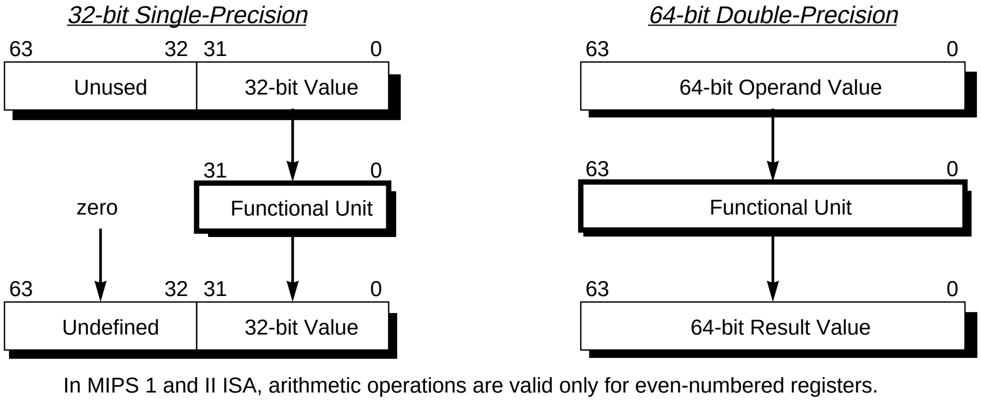

32-bit Single-Precision
63
32
31
0
Unused
32-bit Value
31
0
zero
Functional Unit
63
32
31
0
Undefined
32-bit Value
64-bit Double-Precision
63
0
64-bit Operand Value
63
0
Functional Unit
63
0
64-bit Result Value
In MIPS 1 and II ISA, arithmetic operations are valid only for even-numbered registers.

Figure 12-5 Operators on Floating-Point Registers

In MIPS I and MIPS II ISAs, all arithmetic instructions, whether single- or double-precision, are limited to using even register numbers. Load, store and move instructions transfer only a single word. Even and odd register numbers are used to access the low and high halves, respectively, of double-precision registers. When storing a floating-point register (SWC1 or MFC1), the processor reads the entire register but writes only the selected half to memory or to an integer register.

Because the register renaming scheme creates a new physical register for every destination, it is not sufficient just to enable writing half of the Floating-Point register file when loading (LWC1 or MTC1); the unchanged half must also be copied into the destination. This old value is read using the shared read port, it is then merged with the new word, and the merged doubleword value is written. (A write to the register file writes all 64 bits in parallel.)

When instructions are renamed in MIPS I or II, the low bit of any FGR field is forced to zero. Thus, each even/odd logical register number pair is treated as an even-numbered double-precision register. Odd numbered logical registers are not used in the mapping tables and dependency logic, but they remain mapped to their latest physical registers.

12.4 Floating-Point Control Registers

The MIPS IV ISA permits up to 32 control registers to be defined for each coprocessor, but the Floating-Point Unit uses only two:

- Control register 0, the FP *Implementation and Revision* register
- Control register 31, the *Floating-Point Status* register (*FSR*)

Floating-Point *Implementation and Revision* Register

The following fields are defined for control register 0 in Coprocessor 1, the FP *Implementation and Revision* register, as shown in Figure 12-6:

- The *Implementation* field holds an 8-bit number, 0x09, which identifies the R10000 implementation of the floating-point coprocessor.
- The *Revision* field is an 8-bit number that defines a particular revision of the floating-point coprocessor. Since it can be arbitrarily changed, it is not defined here.

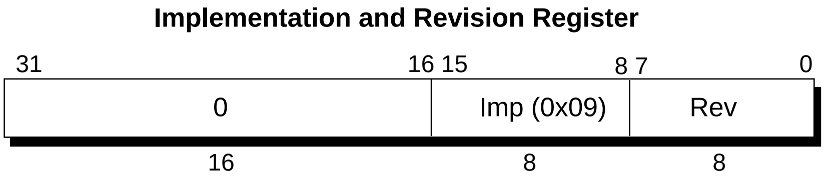


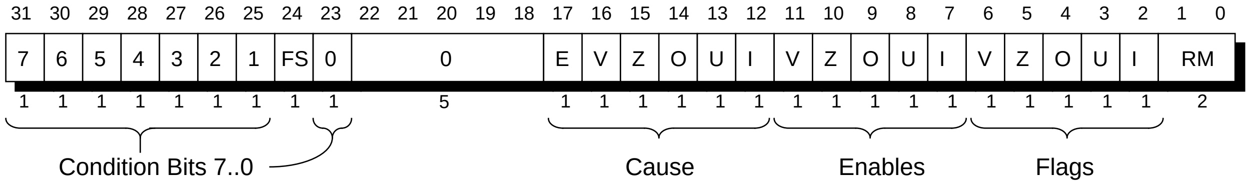
Figure 12-6 FP Implementation and Revision Register Format

Floating-Point Status Register (FSR)

Figure 12-7 shows the Floating-Point *Status* register (*FSR*), control register 31 in Coprocessor 1. It is implemented in the graduation unit rather than the Floating-Point Unit, because it is closely tied to the active list.

Bits 22:18 are unimplemented and must be set to zero. All other bits may be read or written using Control Move instructions from or to Coprocessor 1 (subfunctions CFC1 or CTC1). These move instructions are fully interlocked; they are delayed in the decode stage until all previous instructions have been graduated, and no subsequent instruction is decoded until they have been completed.

FP Status Register



Condition bits are True/False values set by floating-point compare instructions.
Flush (FS) bit: 0: A denormalized result causes an Unimplemented Operation exception.
1: A denormalized result is replaced with zero. No exception is flagged.
Cause bits indicate the status of each floating-point arithmetic instruction. (Not by load, store, or move.)
Enable bits enable an exception if the corresponding *Cause* bit is set.
Flag bits are set whenever the corresponding *Cause* bit is a 1. These bits are cumulative. Once a bit is set, it remains set until the FSR is written by a CTC1 instruction.
E Unimplemented operation. This exception is always enabled.
IEEE 754 Exception bits: The following bits may be individually enabled:
V Invalid operation.
Z Division by zero. (Divide unit only.)
O Overflow.
U Underflow.
I Inexact operation. (Result can not be stored precisely.)
Round Mode (RM): (IEEE specification)
0: *RN*, Round to nearest representable value. If two values are equally near, set the lowest bit to zero.
1: *RZ*, Round toward Zero. Round to the closest value whose magnitude is not greater than the result.
2: *RP*, Round to Plus Infinity. Round to the closest value whose magnitude is not less than the result.
3: *RM*, Round to Minus Infinity. Round to the closest value whose magnitude is not greater.

Figure 12-7 Floating-Point Status Register (FSR)

Bit Descriptions of the FSR

Description of the bits in the FSR are as follows:

Condition Bits [31:25,23]: The *Condition* bits indicate the result of floating-point compare instructions. The active list keeps track of these bits.

Cause Bits [17:12]: Each functional unit can detect exceptional cases in their function codes, operands, or results. These cases are indicated by setting one of six specific *Cause* bits. The *Cause* bits indicate the status of the floating-point arithmetic instruction which graduated most recently or caused an exception to be taken. The *FSR* is not modified by load, store, or move instructions. All cause bits, except *E*, have corresponding *Enable* and *Flag* bits in the FSR.

E	*Unimplemented operation*: the execution unit does not perform the specified operation. This exception is always enabled.
V	*Invalid operation*: this operation is not valid for the given operands.
Z	*Division by zero*: (divide unit only) the result of division by zero is not defined.
O	*Overflow*: the result is too large in magnitude to be correctly represented in the result format.
U	*Underflow*: the result is too small in magnitude to be correctly represented in the result format.
I	*Inexact Result*: the result cannot be represented exactly.

NOTE: The *FSR* is modified only for instructions issued by the floating-point queue. Move From (MFC or DMFC) instructions never set the *Cause* field; status bits from the functional unit (multiplier) must be ignored. Move or Move Conditional instructions can set the Unimplemented Operation exception only in the *Cause* field. Load and store instructions are issued by the address queue.)

The functional units generate the *Cause* bits and send them to the graduation unit when the operation is completed.

Enable Bits [11:7]: The five *Enable* bits individually enable (when set to a 1) or disable (when set to a 0) exceptions when the corresponding *Cause* bit is set.

Flag Bits [6:2]: One of the five *Flag* bits is set when a floating-point arithmetic instruction graduates, if the corresponding *Cause* bit is set. The *Flag* bits are sticky and remain set until the *FSR* is written. Thus, the *Flag* bits indicate the status of all floating-point instructions graduated since the *FSR* was last written. The *Flag* bits are not modified for any instructions which cause an exception to be taken.

Round Mode [1:0]: RM bits select one of the four IEEE rounding modes. Most floating-point results cannot be precisely represented by the 32-bit or 64-bit register formats, and must be truncated and rounded to a representable value. The modes selected by the *RM* bit values are:

> 0: *RN*, round to nearest representable value. If two values are equally near, set the lowest bit to zero.
>
> 1: *RZ*, round toward zero. Round to the closest value whose magnitude is not greater than the result.
>
> 2: *RP*, round to plus infinity. Round to the closest value whose magnitude is not less than the result.
>
> 3: *RM*, round to minus infinity. Round to the closest value whose magnitude is not greater.

The *Round* and *Enable* bits only change when the *FSR* is written by a CTC1 (Move To Coprocessor 1 Control Register) instruction. Each CTC1 instruction is executed sequentially, after all previous floating-point instructions have been completed, so these *FSR* bits do not change while any floating-point instruction is active. These bits are broadcast from the graduation unit to all the floating-point functional units.

When a *Cause* bit is set and its corresponding *Enable* bit is also set, an exception is taken on the instruction. The result of the instruction is not stored, and the *Flag* bits are not changed. If no exception is taken, the corresponding *Flag* bits are set.

The *Cause* and *Flag* bits may be read or written. If a CTC1 instruction sets both a *Cause* bit and its *Enable* bit, an exception is taken immediately. The *FSR* is written, but the exception is reported on the move instruction.

Loading the FSR

The *FSR* may be loaded from an integer register by a CTC1 instruction which selects control register 31. This instruction is executed serially; that is, it is delayed during decode until the entire pipeline has emptied, and it is completed before the next instruction is decoded. This instruction writes all *FSR* bits.

If any *Cause* bit and its corresponding *Enable* bit are both set, an exception is taken after FSR has been modified. The CTC1 instruction is aborted; it does not graduate, even though it has changed the processor state.

13. Memory Management

This section describes the R10000 processor memory management, including:

- processor modes and exceptions
- virtual address space
- virtual address translation

13.1 Processor Modes

The R10000 has three operating modes and two addressing modes. All are described in this section.

Processor Operating Modes

The three operating modes are listed in order of decreasing system privilege:

- **Kernel mode** (highest system privilege): can access and change any register. The innermost core of the operating system runs in kernel mode.
- **Supervisor mode**: has fewer privileges and is used for less critical sections of the operating system.
- **User mode** (lowest system privilege): prevents users from interfering with one another.

Selection between the three modes can be made by the operating system (when in Kernal mode) by writing into *Status* register's *KSU* field. The processor is forced into Kernel mode when the processor is handling an error (the *ERL* bit is set) or an exception (the *EXL* bit is set). Table 13-1 shows the selection of operating modes with respect to the *KSU*, *EXL* and *ERL* bits.

Table 13-1 also shows how different instruction sets and addressing modes are enabled by the *Status* register's *XX*, *UX*, *SX* and *KX* bits. A dash ("-") in this table indicates a "don't care." For detailed information on the address spaces available in each mode, refer to section titled, "Virtual Address Space," in this chapter.

The R10000 processor was designed for use with the MIPS IV ISA; however, for compatibility with earlier machines, the useable ISAs can be limited to either MIPS III or MIPSI/II.

Table 13-1 Processor Modes

XX 31	KX 7	SX 6	UX 5	KSU 4:3	ERL 2	EXL 1	Description	ISA‡ III	ISA‡ IV	Addressing Mode 32-Bit/64-Bit
0	-*	-	0	10	0	0	User mode.	No	No	32
1	-	-	0	10	0	0		No	Yes	32
0	-	-	1	10	0	0		Yes	No	64
1	-	-	1	10	0	0		Yes	Yes	64
-	-	0	-	01	0	0	Supervisor mode.	No	Yes	32
-	-	1	-	01	0	0		Yes	Yes	64
-	0	-	-	00	0	0	Kernel mode.	Yes	Yes	32
-	1	-	-	00	0	0		Yes	Yes	64
-	0	-	-	-	0	1	Exception Level	Yes	Yes	32
-	1	-	-	-	0	1		Yes	Yes	64
-	0	-	-	-	1	X	Error Level.	Yes	Yes	32
-	1	-	-	-	1	X		Yes	Yes	64

‡ No means the ISA is disabled; Yes means the ISA is enabled.

* Dashes (-) are "don't care."

Addressing Modes

The processor's *addressing mode* determines whether it generates 32-bit or 64-bit memory addresses.

Refer to Table 13-1 for the following addressing mode encodings:

- In Kernel mode the *KX* bit allows 64-bit addressing; all instructions are always valid.
- In Supervisor mode, the *SX* bit allows 64-bit addressing and the MIPS III instructions. MIPS IV ISA is enabled all the time in Supervisor mode.
- In User mode, the *UX* bit allows 64-bit addressing and the MIPS III instructions; the *XX* bit allows the new MIPS IV instructions.

13.2 Virtual Address Space

The processor uses either 32-bit or 64-bit address spaces, depending on the operating and addressing modes set by the *Status* register. Table 13-1 lists the decoding of these modes.

The processor uses the following addresses:

- virtual address **VA[43:0]**
- region bits **VA[63:59]**

If a region is **mapped**, virtual addresses are translated in the TLB. Bits **VA[58:44]** are not translated in the TLB and are sign extensions of bit **VA[43]**.

In both 32-bit and 64-bit address mode, the memory address space is divided into many regions, as shown in Figure 13-3. Each region has specific characteristics and uses. The user can access only the *useg* region in 32-bit mode, or *xuseg* in 64-bit mode, as shown in Figure 13-1. The supervisor can access user regions as well as *sseg* (in 32-bit mode) or *xsseg* and *csseg* (in 64-bit mode), shown in Figure 13-2. The kernel can access all regions except those restricted because bits **VA[58:44]** are not implemented in the TLB, as shown in Figure 13-3.

The R10000 processor follows the R4400 implementation for *data* references only, ensuring compatibility with the NT kernel. If any of the upper 33 bits are nonzero for an instruction fetch, an Address Error is generated. Refer to Table 13-2 for delineation of the address spaces.

User Mode Operations

In User mode, a single, uniform virtual address space—labelled User segment—is available; its size is:

- 2 Gbytes (2^{31} bytes) in 32-bit mode (*useg*)
- 16 Tbytes (2^{44} bytes) in 64-bit mode (*xuseg*)

Figure 13-1 shows User mode virtual address space.

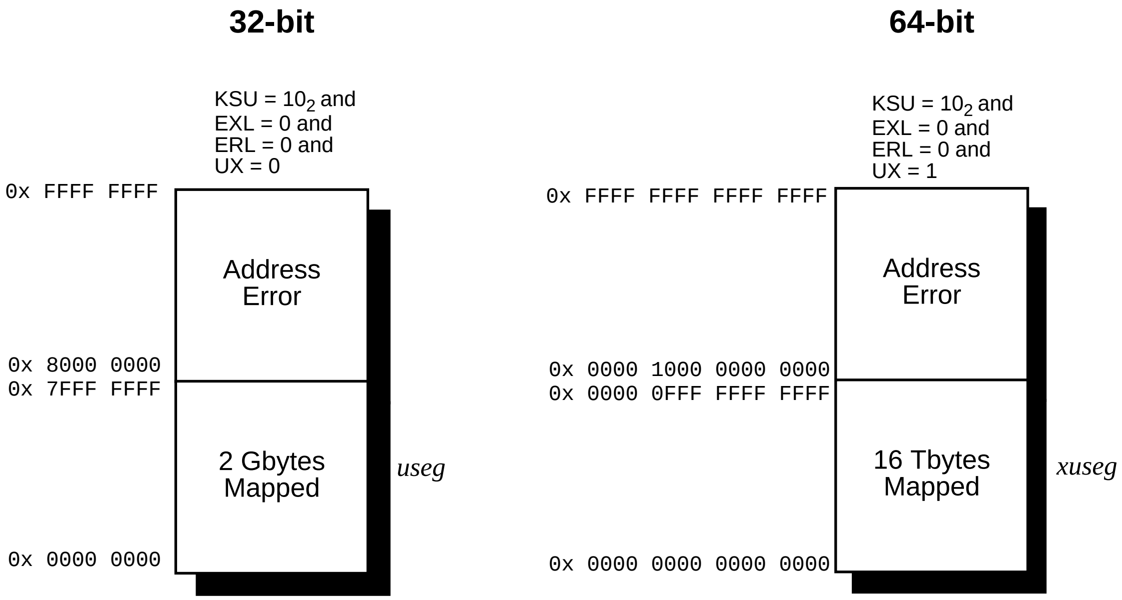

32-bit
KSU = 10₂ and
EXL = 0 and
ERL = 0 and
UX = 0
0x FFFF FFFF
Address Error
0x 8000 0000
0x 7FFF FFFF
2 Gbytes Mapped
useg
0x 0000 0000
64-bit
KSU = 10₂ and
EXL = 0 and
ERL = 0 and
UX = 1
0x FFFF FFFF FFFF FFFF
Address Error
0x 0000 1000 0000 0000
0x 0000 0FFF FFFF FFFF
16 Tbytes Mapped
xuseg
0x 0000 0000 0000 0000

Figure 13-1 User Mode Virtual Address Space

The User segment starts at address 0 and the current active user process resides in either *useg* (in 32-bit mode) or *xuseg* (in 64-bit mode). The TLB identically maps all references to *useg*/*xuseg* from all modes, and controls cache accessibility.

32-bit User Mode (*useg*)

In User mode, when *UX* = 0 in the *Status* register, User mode addressing is compatible with the 32-bit addressing model shown in Figure 13-1, and a 2-Gbyte user address space is available, labelled *useg*.

All valid User mode virtual addresses have their most-significant bit cleared to 0; any attempt to reference an address with the most-significant bit set while in User mode causes an Address Error exception.

The system maps all references to *useg* through the TLB, and bit settings within the TLB entry for the page determine the cacheability of a reference.

64-bit User Mode (*xuseg*)

In User mode, when *UX* =1 in the *Status* register, User mode addressing is extended to the 64-bit model shown in Figure 13-1. In 64-bit User mode, the processor provides a single, uniform virtual address space of 2^{44} bytes, labelled *xuseg*.

All valid User mode virtual addresses have bits 63:44 equal to 0; an attempt to reference an address with bits 63:44 not equal to 0 causes an Address Error exception.

Although the system may be in 32-bit mode, address logic still generates 64-bit values. In this case the high 32 bits must equal the sign bit (31), or an Address Error exception is taken.

Supervisor Mode Operations

Supervisor mode is designed for layered operating systems in which a true kernel runs in processor Kernel mode, and the rest of the operating system runs in Supervisor mode.

The processor operates in Supervisor mode when the *Status* register contains the Supervisor-mode bit-values shown in Table 13-1.

Figure 13-2 shows Supervisor mode address mapping.

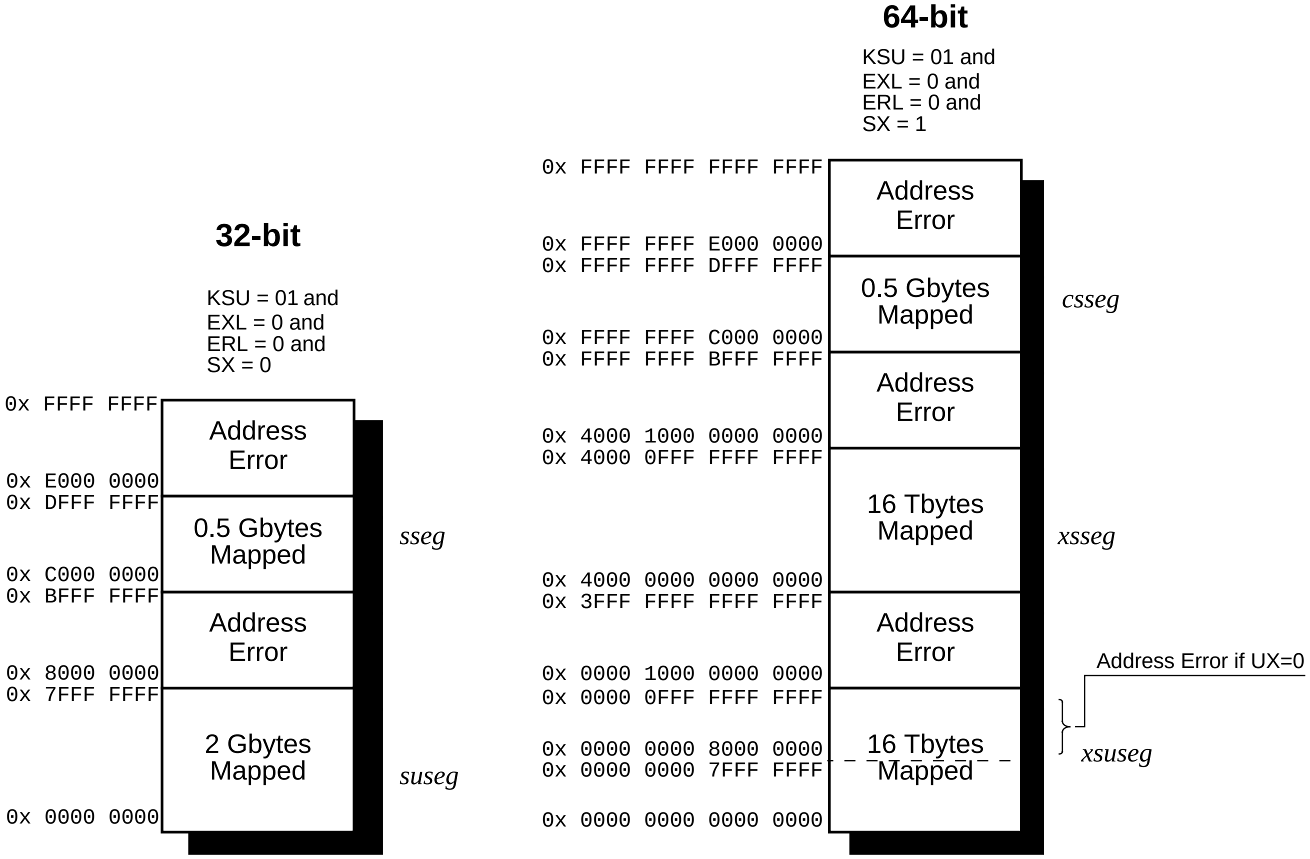

64-bit
KSU = 01 and
EXL = 0 and
ERL = 0 and
SX = 1
0x FFFF FFFF FFFF FFFF
Address Error
0x FFFF FFFF E000 0000
0x FFFF FFFF DFFF FFFF
0.5 Gbytes Mapped
csseg
0x FFFF FFFF C000 0000
0x FFFF FFFF BFFF FFFF
Address Error
0x 4000 1000 0000 0000
0x 4000 0FFF FFFF FFFF
16 Tbytes Mapped
xsseg
0x 4000 0000 0000 0000
0x 3FFF FFFF FFFF FFFF
Address Error
0x 0000 1000 0000 0000
0x 0000 0FFF FFFF FFFF
Address Error if UX=0
16 Tbytes Mapped
xsuseg
0x 0000 0000 8000 0000
0x 0000 0000 7FFF FFFF
0x 0000 0000 0000 0000
32-bit
KSU = 01 and
EXL = 0 and
ERL = 0 and
SX = 0
0x FFFF FFFF
Address Error
0x E000 0000
0x DFFF FFFF
0.5 Gbytes Mapped
sseg
0x C000 0000
0x BFFF FFFF
Address Error
0x 8000 0000
0x 7FFF FFFF
2 Gbytes Mapped
suseg
0x 0000 0000

Figure 13-2 Supervisor Mode Address Space

32-bit Supervisor Mode, User Space (*suseg*)

In Supervisor mode, when *SX* = 0 in the *Status* register and the most-significant bit of the 32-bit virtual address is set to 0, the *suseg* virtual address space is selected; it covers the full 2^{31} bytes (2 Gbytes) of the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space starts at virtual address 0x0000 0000 and runs through 0x7FFF FFFF.

32-bit Supervisor Mode, Supervisor Space (*sseg*)

In Supervisor mode, when *SX* = 0 in the *Status* register and the three most-significant bits of the 32-bit virtual address are 110_2, the *sseg* virtual address space is selected; it covers 2^{29}-bytes (512 Mbytes) of the current supervisor address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space begins at virtual address 0xC000 0000 and runs through 0xDFFF FFFF.

64-bit Supervisor Mode, User Space (*xsuseg*)

In Supervisor mode, when *SX* = 1 in the *Status* register and bits 63:62 of the virtual address are set to 00_2, selection of the *xsuseg* virtual address space is dependent upon the *UX* bit.

- if *UX* = 1, the entire space from 0x0000 0000 0000 0000 through 0000 0FFF FFFF FFFF (16 Tbytes) is selected.
- If *UX* = 0, the address space 0x0000 0000 0000 0000 through 0000 0000 7FFF FFFF (2 Gbytes) is selected. Addressing the space ranging from 0000 0000 8000 0000 through 0000 0FFF FFFF FFFF will cause an address error.

The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

64-bit Supervisor Mode, Current Supervisor Space (*xsseg*)

In Supervisor mode, when *SX* = 1 in the *Status* register and bits 63:62 of the virtual address are set to 01_2, the *xsseg* current supervisor virtual address space is selected. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space begins at virtual address 0x4000 0000 0000 0000 and runs through 0x4000 0FFF FFFF FFFF.

64-bit Supervisor Mode, Separate Supervisor Space (*csseg*)

In Supervisor mode, when *SX* = 1 in the *Status* register and bits 63:62 of the virtual address are set to 11_2, the *csseg* separate supervisor virtual address space is selected. Addressing of the *csseg* is compatible with addressing *sseg* in 32-bit mode. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space begins at virtual address 0xFFFF FFFF C000 0000 and runs through 0xFFFF FFFF DFFF FFFF.

Kernel Mode Operations

The processor operates in Kernel mode when the *Status* register contains the Kernel-mode bit-values shown in Table 13-1.

Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 13-3.

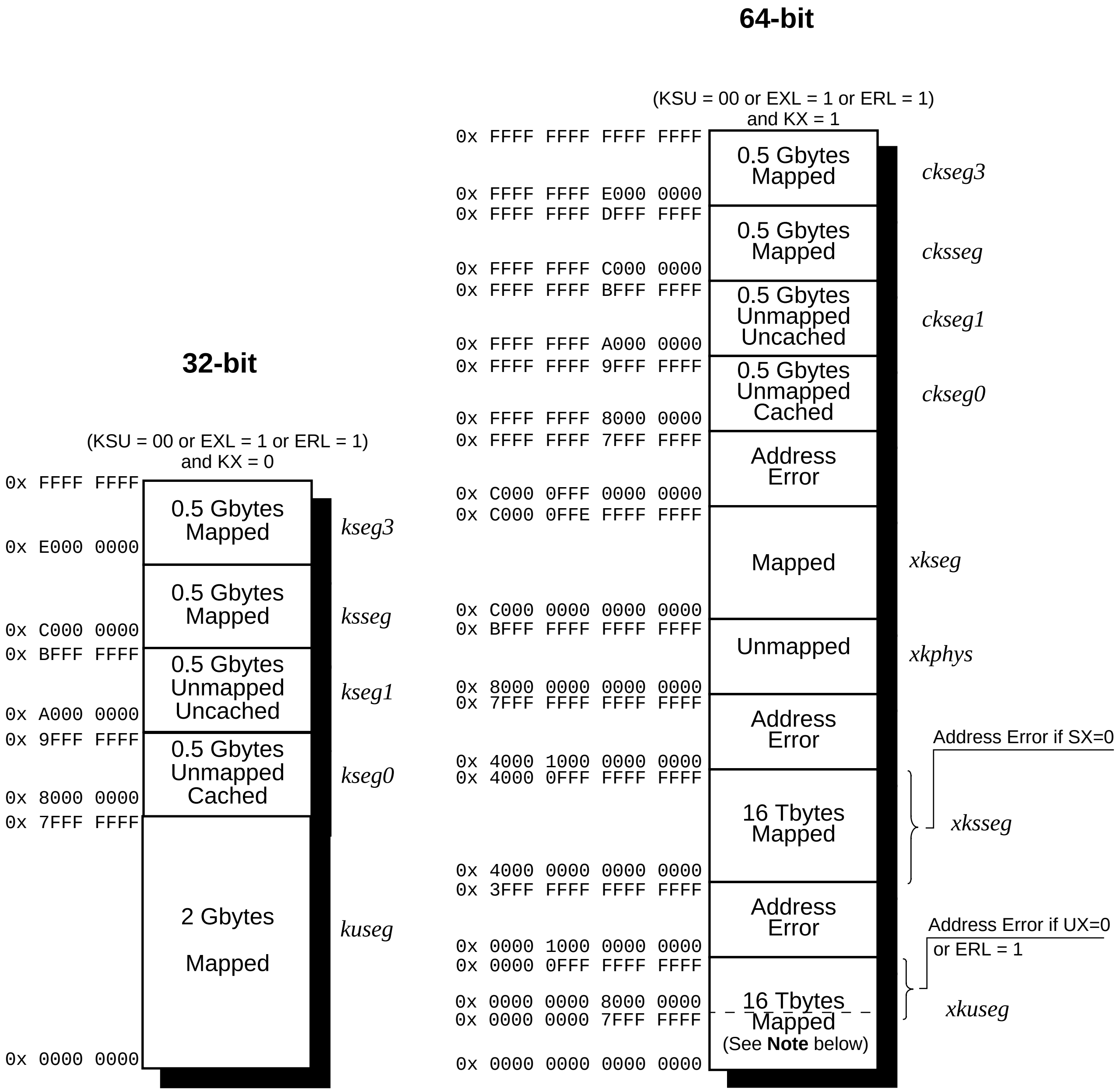

64-bit
(KSU = 00 or EXL = 1 or ERL = 1)
and KX = 1
0x FFFF FFFF FFFF FFFF
0.5 Gbytes Mapped
ckseg3
0x FFFF FFFF E000 0000
0x FFFF FFFF DFFF FFFF
0.5 Gbytes Mapped
cksseg
0x FFFF FFFF C000 0000
0x FFFF FFFF BFFF FFFF
0.5 Gbytes Unmapped Uncached
ckseg1
0x FFFF FFFF A000 0000
0x FFFF FFFF 9FFF FFFF
0.5 Gbytes Unmapped Cached
ckseg0
0x FFFF FFFF 8000 0000
0x FFFF FFFF 7FFF FFFF
Address Error
0x C000 0FFF 0000 0000
0x C000 0FFE FFFF FFFF
Mapped
xkseg
0x C000 0000 0000 0000
0x BFFF FFFF FFFF FFFF
Unmapped
xkphys
0x 8000 0000 0000 0000
0x 7FFF FFFF FFFF FFFF
Address Error
Address Error if SX=0
0x 4000 1000 0000 0000
0x 4000 0FFF FFFF FFFF
16 Tbytes Mapped
xksseg
0x 4000 0000 0000 0000
0x 3FFF FFFF FFFF FFFF
Address Error
Address Error if UX=0
or ERL = 1
0x 0000 1000 0000 0000
0x 0000 0FFF FFFF FFFF
0x 0000 0000 8000 0000
0x 0000 0000 7FFF FFFF
16 Tbytes Mapped
(See Note below)
xkuseg
0x 0000 0000 0000 0000
32-bit
(KSU = 00 or EXL = 1 or ERL = 1)
and KX = 0
0x FFFF FFFF
0.5 Gbytes Mapped
kseg3
0x E000 0000
0.5 Gbytes Mapped
ksseg
0x C000 0000
0x BFFF FFFF
0.5 Gbytes Unmapped Uncached
kseg1
0x A000 0000
0x 9FFF FFFF
0.5 Gbytes Unmapped Cached
kseg0
0x 8000 0000
0x 7FFF FFFF
2 Gbytes
Mapped
kuseg
0x 0000 0000

Figure 13-3 Kernel Mode Address Space

NOTE: If *ERL* = 1, the selected 2 Gbyte space becomes uncached and unmapped.

32-bit Kernel Mode, User Space (*kuseg*)

In Kernel mode, when *KX* = 0 in the *Status* register, and the most-significant bit of the virtual address, A31, is cleared, the 32-bit *kuseg* virtual address space is selected; it covers the full 2^{31} bytes (2 Gbytes) of the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

32-bit Kernel Mode, Kernel Space 0 (*kseg0*)

In Kernel mode, when *KX* = 0 in the *Status* register and the most-significant three bits of the virtual address are 100_2, 32-bit *kseg0* virtual address space is selected; it is the 2^{29}-byte (512-Mbyte) kernel physical space. References to *kseg0* are not mapped through the TLB; the physical address is selected by subtracting 0x8000 0000 from the virtual address. The *K0* field of the *Config* register determines cacheability and coherency.

32-bit Kernel Mode, Kernel Space 1 (*kseg1*)

In Kernel mode, when *KX* = 0 in the *Status* register and the most-significant three bits of the 32-bit virtual address are 101_2, 32-bit *kseg1* virtual address space is selected; it is the 2^{29}-byte (512-Mbyte) kernel physical space.

References to *kseg1* are not mapped through the TLB; the physical address is selected by subtracting 0xA000 0000 from the virtual address.

Caches are disabled for accesses to these addresses, and physical memory (or memory-mapped I/O device registers) are accessed directly.

32-bit Kernel Mode, Supervisor Space (*ksseg*)

In Kernel mode, when *KX* = 0 in the *Status* register and the most-significant three bits of the 32-bit virtual address are 110_2, the *ksseg* virtual address space is selected; it is the current 2^{29}-byte (512-Mbyte) supervisor virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

References to *ksseg* are mapped through the TLB.

32-bit Kernel Mode, Kernel Space 3 (*kseg3*)

In Kernel mode, when *KX* = 0 in the *Status* register and the most-significant three bits of the 32-bit virtual address are 111_2, the *kseg3* virtual address space is selected; it is the current 2^{29}-byte (512-Mbyte) kernel virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

References to *kseg3* are mapped through the TLB.

64-bit Kernel Mode, User Space (*xkuseg*)

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63:62 of the 64-bit virtual address are 00_2, selection of the *xkuseg* virtual address space is dependent upon the *UX* and *ERL* bits.

- if *UX* = 1 and *ERL* = 0, the entire space from 0x0000 0000 0000 0000 through 0000 0FFF FFFF FFFF (16 Tbytes) is selected.
- If *UX* = 0 or *ERL* = 1, the address space 0x0000 0000 0000 0000 through 0000 0000 7FFF FFFF (2 Gbytes) is selected. Addressing the space ranging from 0000 0000 8000 0000 through 0000 0FFF FFFF FFFF will cause an address error. Moreover, if *ERL*=1, the selected 2-Gbyte address space becomes unmapped and uncached.

The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

64-bit Kernel Mode, Current Supervisor Space (*xksseg*)

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63:62 of the 64-bit virtual address are 01_2, selection of the *xksseg* virtual address space is dependent upon the *SX* bit.

- if *SX* = 1, the entire space from 0x4000 0000 0000 0000 through 4000 0FFF FFFF FFFF (16 Tbytes) is selected.
- If *SX* = 0, access to any address in the space ranging from 0x4000 0000 0000 0000 through 4000 0FFF FFFF FFFF causes an address error.

The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

64-bit Kernel Mode, Physical Spaces (*xkphys*)

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63:62 of the 64-bit virtual address are 10_2, the *xkphys* virtual address space is selected; it is a set of eight kernel physical spaces. Each kernel physical space contains either one or four 2^{40}-byte physical pages.

References to this space are not mapped; the physical address selected is taken directly from bits 39:0 of the virtual address. Bits 61:59 of the virtual address specify the *cache algorithm*, described in Chapter 4, the section titled “Cache Algorithms.” If the cache algorithm is either uncached or uncached accelerated (values of 2 or 7) the space contains four physical pages; access to addresses whose bits 56:40 are not equal to 0 cause an Address Error exception. Address bits 58:57 carry the *uncached attribute* (described in Chapter 6, the section titled “Support for Uncached Attribute”), and are not checked for address errors.

If the cache algorithm is neither uncached nor uncached accelerated, the space contains a single physical page, as on the R4400 processor. In this case, access to addresses whose bits 58:40 are not equal to a zero cause an Address Error exception, as shown in Figure 13-4.

0X BFFFFFFF FFFFFFFF 0X BE000100 00000000	Address Error
0X BE0000FF FFFFFFFF 0X BE000000 00000000	Uncached Accelerated
0X BDFFFFFF FFFFFFFF 0X BC000100 00000000	Address Error
0X BC0000FF FFFFFFFF 0X BC000000 00000000	Uncached Accelerated
0X BBFFFFFF FFFFFFFF 0X BA000100 00000000	Address Error
0X BA0000FF FFFFFFFF 0X BA000000 00000000	Uncached Accelerated
0X B9FFFFFF FFFFFFFF 0X B8000100 00000000	Address Error
0X B80000FF FFFFFFFF 0X B8000000 00000000	Uncached Accelerated
0X B7FFFFFF FFFFFFFF 0X B0000100 00000000	Address Error
0X B00000FF FFFFFFFF 0X B0000000 00000000	Reserved‡
0X AFFFFFFF FFFFFFFF 0X A8000100 00000000	Address Error
0X A80000FF FFFFFFFF 0X A8000000 00000000	Cacheable Exclusive Write
0X A7FFFFFF FFFFFFFF 0X A0000100 00000000	Address Error
0X A00000FF FFFFFFFF 0X A0000000 00000000	Cacheable Exclusive

0X 9FFFFFFF FFFFFFFF 0X 98000100 00000000	Address Error
0X 980000FF FFFFFFFF 0X 98000000 00000000	Cacheable Noncoherent
0X 97FFFFFF FFFFFFFF 0X 96000100 00000000	Address Error
0X 960000FF FFFFFFFF 0X 96000000 00000000	Uncached
0X 95FFFFFF FFFFFFFF 0X 94000100 00000000	Address Error
0X 940000FF FFFFFFFF 0X 94000000 00000000	Uncached
0X 93FFFFFF FFFFFFFF 0X 92000100 00000000	Address Error
0X 920000FF FFFFFFFF 0X 92000000 00000000	Uncached
0X 91FFFFFF FFFFFFFF 0X 90000100 00000000	Address Error
0X 900000FF FFFFFFFF 0X 90000000 00000000	Uncached
0X 8FFFFFFF FFFFFFFF 0X 88000100 00000000	Address Error
0X 880000FF FFFFFFFF 0X 88000000 00000000	Reserved‡
0X 87FFFFFF FFFFFFFF 0X 80000100 00000000	Address Error
0X 800000FF FFFFFFFF 0X 80000000 00000000	Reserved‡

‡ Accessing a reserved space results in undefined behavior.

Figure 13-4 xkphys Virtual Address Space

64-bit Kernel Mode, Kernel Space (*xkseg*)

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63:62 of the 64-bit virtual address are 11_2, the address space selected is one of the following:

- kernel virtual space, *xkseg*, the current kernel virtual space*;* the virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address
- one of the four 32-bit kernel mode compatibility spaces (described below).

64-bit Kernel Mode, Compatibility Spaces (*ckseg1:0, cksseg, ckseg3*)

In Kernel mode, when *KX* = 1 in the *Status* register, bits 63:62 of the 64-bit virtual address are 11_2, and bits 61:31 of the virtual address equal –1, the lower two bytes of address, as shown in Figure 13-3, select one of the following 512-Mbyte compatibility spaces.

- *ckseg0*. This 64-bit virtual address space is an unmapped region, compatible with the 32-bit address model *kseg0*. The *K0* field of the *Config* register controls cacheability and coherency.
- *ckseg1*. This 64-bit virtual address space is an unmapped and uncached region, compatible with the 32-bit address model *kseg1*.
- *cksseg*. This 64-bit virtual address space is the current supervisor virtual space, compatible with the 32-bit address model *ksse*g.
- *ckseg3*. This 64-bit virtual address space is kernel virtual space, compatible with the 32-bit address model *kseg3*.

Address Space Access Privilege Differences Between the R4400 and R10000

In the R4400, the 64-bit Supervisor mode can access the entire *xsuseg* space, and the 64-bit Kernel mode can access the entire *xksseg* and *xkuseg* spaces. Access privileges in the R10000 are also dependent on the *UX* and *SX* bits:

- Access to the 64-bit user space in 64-bit Supervisor or Kernel mode (*xsuseg* or *xkuseg*) is controlled by the *UX* bit. If *UX*=0, the 64-bit Supervisor and Kernel modes can only access the 32-bit user space (*suseg* or *kuseg*).
- Access to the 64-bit supervisor space in Kernel mode (*xksseg*) is controlled by the *SX* bit. If *SX*=0, the 64-bit Kernel mode can only access the 32-bit supervisor space (*ksseg*).

An Address Error exception is taken on an illegal access.

The R10000 processor implements the same access privileges for 32-bit processor modes as in the R4400. The Table 13-2 summarizes the access privileges for all processor modes in the R10000 processor.

Table 13-2 Access Privileges for User, Supervisor and Kernel Mode Operations

64-bit Virtual Address	**32-bit Mode**			**64-bit Mode**			
	User‡	**Supervisor**	**Kernel**	**User**	**Supervisor**	**Kernel & ERL=0**	**Kernel & ERL=1**

`FFFFFFFF E0000000`
`TO`
`FFFFFFFF FFFFFFFF`
	AddrErr	*AddrErr*	OK	*AddrErr*	*AddrErr*	OK	OK

`FFFFFFFF C0000000`
`TO`
`FFFFFFFF DFFFFFFF`
		OK			OK		

`FFFFFFFF A0000000`
`TO`
`FFFFFFFF BFFFFFFF`
		AddrErr			*AddrErr*		

`FFFFFFFF 80000000`
`TO`
`FFFFFFFF 9FFFFFFF`							

`C0000FFF 00000000`
`TO`
`FFFFFFFF 7FFFFFFF`
			AddrErr			*AddrErr*	*AddrErr*

`C0000000 00000000`
`TO`
`C0000FFE FFFFFFFF`
						OK	OK

`80000000 00000000`
`TO`
`BFFFFFFF FFFFFFFF`
						OK	OK

`40001000 00000000`
`TO`
`7FFFFFFF FFFFFFFF`
						AddrErr	*AddrErr*

`40000000 00000000`
`TO`
`40000FFF FFFFFFFF`
					OK	*AddrErr if SX=0*	*AddrErr if SX=0*

`00001000 00000000`
`TO`
`3FFFFFFF FFFFFFFF`
					AddrErr	*AddrErr*	*AddrErr*

`00000000 80000000`
`TO`
`00000FFF FFFFFFFF`
				OK	*AddrErr if UX=0*	*AddrErr if UX=0*	*AddrErr*

`00000000 00000000`
`TO`
`00000000 7FFFFFFF`
	OK	OK	OK		OK	OK	OK

‡ For data references, the upper 32 bits of the virtual addresses are cleared before checking access privilege and TLB translation.

13.3 Virtual Address Translation

Programs can operate using either **physical** or **virtual** memory addresses:

- physical addresses correspond to hardware locations in main memory
- virtual addresses are logical values only, and do not correspond to fixed hardware locations

Virtual addresses must first be **translated** (finding the physical address at which the virtual address points) before main memory can be accessed. This translation is essential for multitasking computer systems, because it allows the operating system to load programs anywhere in main memory independent of the logical addresses used by the programs.

This translation also implements a memory protection scheme, which limits the amount of memory each program may access. The scheme prevents programs from interfering with the memory used by other programs or the operating system.

Virtual Pages

Translated virtual addresses retrieve data in blocks, which are called **pages**. In the R10000 processor, the size of each page may be selected from a range that runs from 4 Kbytes to 16 Mbytes inclusive, in powers of 4 (that is, 4 Kbytes, 16 Kbytes, 64 Kbytes, etc.).

The virtual address bits which select a page (and thus are translated) are called the *page address*. The lower bits which select a byte within the selected page are called the *offset* and are not translated. The number of offset bits varies from 12 to 24 bits, depending on the page size.

Virtual Page Size Encodings

Page size is defined in each TLB entry's *PageMask* field. This field is loaded or read using the *PageMask* register, as described in Chapter 11, the section titled "PageMask Register (5)."

Each entry translates a pair of physical pages. The low bit of the virtual address page is not compared, because it is used to select between these two physical pages.

Using the TLB

Translations are maintained by the operating system, using page tables in memory. A subset of these translations are loaded into a hardware buffer called the **translation-lookaside buffer** or TLB. The contents of this buffer are maintained by the operating system; if an instruction needs a translation which is not already in the buffer, an exception is taken so the operating system can compute and load the needed translation. If all the necessary translations are present, the program is executed without any delays.

The TLB contains 64 entries, each of which maps a pair of virtual pages. Formats of TLB entries are shown in Figure 13-5.

Cache Algorithm Field

The *Cache Algorithm* fields of the TLB, *EntryLo0*, *EntryLo1*, and *Config* registers indicate how data is cached. Cache algorithms are described in Chapter 4, the section titled “Cache Algorithms.”

Format of a TLB Entry

Figure 13-5 shows the TLB entry formats for both 32- and 64-bit modes. Each field of an entry has a corresponding field in the *EntryHi*, *EntryLo0*, *EntryLo1*, or *PageMask* registers, as shown in Chapter 11; for example the *PFN* and uncached attribute (*UC*) fields of the TLB entry are also held in the *EntryLo* registers.

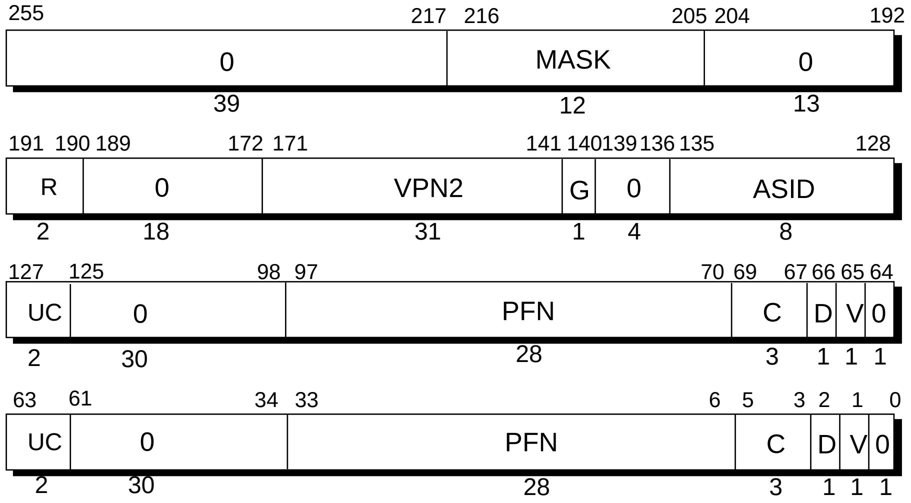

255 217 216 205 204 192
0 MASK 0
39 12 13
191 190 189 172 171 141 140 139 136 135 128
R 0 VPN2 G 0 ASID
2 18 31 1 4 8
127 125 98 97 70 69 67 66 65 64
UC 0 PFN C D V 0
2 30 28 3 1 1 1
63 61 34 33 6 5 3 2 1 0
UC 0 PFN C D V 0
2 30 28 3 1 1 1

Figure 13-5 Format of a TLB Entry

Address Translation

Because a 64-bit address is unnecessarily large, only the low 44 address bits are translated. The high two virtual address bits (bits 63:62) select between user, supervisor, and kernel address spaces. The intermediate address bits (61:44) must either be all zeros or all ones, depending on the address region. The TLB does not include virtual address bits 61:59, because these are decoded only in the *xkphys* region, which is unmapped.

For data cache accesses, the **joint TLB** (JTLB) translates addresses from the address calculate unit. For instruction accesses, the JTLB translates the PC address if it misses in the instruction TLB (ITLB). That entry is copied into the ITLB for subsequent accesses. The ITLB is transparent to system software.

Address Space Identification (*ASID*)

Each independent task, or *process*, has a separate address space, assigned a unique 8-bit Address Space Identifier (**ASID**). This identifier is stored with each TLB entry to distinguish between entries loaded for different processes. The ASID allows the processor to move from one process to another (called a **context switch**) without having to invalidate TLB entries.

The processor's current ASID is stored in the low 8 bits of the *EntryHi* register. These bits are also used to load the *ASID* field of an entry during TLB refill.

The *ASID* field of each TLB entry is compared to the *EntryHi* register; if the ASIDs are equal or if the entry is global (see below), this TLB entry may be used to translate virtual addresses. The ASID comparison is performed only when a new value is loaded into the *EntryHi* register; the one-bit result of the match is stored in a static Enable latch. (This bit is set whenever a new entry is loaded.)

Global Processes (*G*)

A translation may be defined as *global* so that it can be shared by all processes. This *G* bit is set in the TLB entry and enables the entry independent of its ASID value.

Avoiding TLB Conflict

Setting the *TS* bit in the *Status* register indicates an entry being presented to the TLB matches more than one virtual page entry in the TLB. Any TLB entries that allow multiple matches, even in the *Wired* area, are invalidated before the new entry can be written into the TLB. This prevents multiple matches during address translation.

14. CPU Exceptions

This chapter describes the processor exceptions—a general view of the cause and return of an exception, exception vector locations, and the types of exceptions that are supported, including the cause, processing, and servicing of each exception.

14.1 Causing and Returning from an Exception

When the processor takes an exception, the *EXL* bit in the *Status* register is set to 1, which means the system is in Kernel mode. After saving the appropriate state, the exception handler typically changes the *KSU* bits in the *Status* register to Kernel mode and resets the *EXL* bit back to 0. When restoring the state and restarting, the handler restores the previous value of the *KSU* field and sets the *EXL* bit back to 1.

Returning from an exception also resets the *EXL* bit to 0 (see the ERET instruction in **V$_R$5000**, **V$_R$10000 INSTRUCTION User's Manual**).

14.2 Exception Vector Locations

The Cold Reset, Soft Reset, and NMI exceptions are always vectored to the dedicated Cold Reset exception vector at an uncached and unmapped address. Addresses for all other exceptions are a combination of a *vector offset* and a *base address*.

The boot-time vectors (when *BEV* = 1 in the *Status* register) are at uncached and unmapped addresses. During normal operation (when *BEV* = 0) the regular exceptions have vectors in cached address spaces; Cache Error is always at an uncached address so that cache error handling can bypass a suspect cache.

The exception vector assignments for the R10000 processor shown in Table 14-1; the addresses are the same as for the R4400.

Table 14-1 Exception Vector Addresses

BEV	Exception Type	Exception Vector Address	
		32-bit	64-bit
	Cold Reset/Soft Reset/ NMI	`0xBFC00000`	`0xFFFFFFFF BFC00000`
BEV=0	TLB Refill (EXL=0)	`0x80000000`	`0xFFFFFFFF 80000000`
	XTLB Refill (EXL=0)	`0x80000080`	`0xFFFFFFFF 80000080`
	Cache Error	`0xA0000100`	`0xFFFFFFFF A0000100`
	Others	`0x80000180`	`0xFFFFFFFF 80000180`
BEV=1	TLB Refill (EXL=0)	`0xBFC00200`	`0xFFFFFFFF BFC00200`
	XTLB Refill (EXL=0)	`0xBFC00280`	`0xFFFFFFFF BFC00280`
	Cache Error	`0xBFC00300`	`0xFFFFFFFF BFC00300`
	Others	`0xBFC00380`	`0xFFFFFFFF BFC00380`

14.3 TLB Refill Vector Selection

In all present implementations of the MIPS III ISA, there are two TLB refill exception vectors:

- one for references to 32-bit address space (TLB Refill)
- one for references to 64-bit address space (XTLB Refill)

Table 14-2 lists the exception vector addresses.

The TLB refill vector selection is based on the address space of the address (*user*, *supervisor*, or *kernel*) that caused the TLB miss, and the value of the corresponding extended addressing bit in the *Status* register (*UX*, *SX*, or *KX*). The current operating mode of the processor is not important except that it plays a part in specifying in which address space an address resides. The *Context* and *XContext* registers are entirely separate page-table-pointer registers that point to and refill from two separate page tables, however these two registers share *BadVPN2* fields (see Chapter 11 for more information). For all TLB exceptions (Refill, Invalid, TLBL or TLBS), the *BadVPN2* fields of both registers are loaded as they were in the R4400.

In contrast to the R10000, the R4400 processor selects the vector based on the current operating mode of the processor (*user*, *supervisor*, or *kernel*) and the value of the corresponding extended addressing bit in the *Status* register (*UX*, *SX* or *KX*). In addition, the *Context* and *XContext* registers are not implemented as entirely separate registers; the *PTEbase* fields are shared. A miss to a particular address goes through either TLB Refill or XTLB Refill, depending on the source of the reference. There can be only be a single page table unless the refill handlers execute address-deciphering and page table selection in software.

> **NOTE:** Refills for the 0.5 Gbyte supervisor mapped region, *sseg/ksseg*, are controlled by the value of *KX* rather than *SX*. This simplifies control of the processor when supervisor mode is not being used.

Table 14-2 lists the TLB refill vector locations, based on the address that caused the TLB miss and its corresponding mode bit.

Table 14-2 TLB Refill Vectors

Space	**Address Range**	**Regions**	Exception Vector

Kernel	`0xFFFF FFFF E000 0000`
	to
	`0xFFFF FFFF FFFF FFFF`
		kseg3	Refill (KX=0)
			or
			XRefill (KX=1)

Supervisor	`0xFFFF FFFF C000 0000`
	to
	`0xFFFF FFFF DFFF FFFF`
		sseg, *ksseg*	Refill (KX=0)
			or
			XRefill (KX=1)

Kernel	`0xC000 0000 0000 0000`
	to
	`0xC000 0FFE FFFF FFFF`
		xkseg	XRefill(KX=1)

Supervisor	`0x4000 0000 0000 0000`
	to
	`0x4000 0FFF FFFF FFFF`
		xsseg, *xksseg*	XRefill (SX=1)

User	`0x0000 0000 8000 0000`
	to
	`0x0000 0FFF FFFF FFFF`
		xsuseg, *xuseg*, *xkuseg*	XRefill (UX=1)

User	`0x0000 0000 0000 0000`
	to
	`0x0000 0000 7FFF FFFF`
		useg, *xuseg*, *suseg*, *xsuseg*, *kuseg*, *xkuseg*	Refill (UX=0)
			or
			XRefill (UX=1)

Priority of Exceptions

The remainder of this chapter describes exceptions in the order of their priority shown in Table 14-3 (with certain of the exceptions, such as the TLB exceptions and Instruction/Data exceptions, grouped together for convenience). While more than one exception can occur for a single instruction, only the exception with the highest priority is reported. Some exceptions are not caused by the instruction executed at the time, and some exceptions may be deferred. See the individual description of each exception in this chapter for more detail.

Table 14-3 Exception Priority Order

Cold Reset *(highest priority)*
Soft Reset
Nonmaskable Interrupt (NMI)‡
Cache error — Instruction cache‡
Cache error — Data cache‡
Cache error — Secondary cache‡
Cache error — System interface‡
Address error — Instruction fetch
TLB refill — Instruction fetch
TLB invalid — Instruction fetch
Bus error — Instruction fetch
Integer overflow, Trap, System Call, Breakpoint, Reserved Instruction, Coprocessor Unusable, or Floating-Point Exception
Address error — Data access
TLB refill — Data access
TLB invalid — Data access
TLB modified — Data write
Watch‡
Bus error — Data access
Interrupt *(lowest priority)*‡

‡ These exceptions are interrupt types, and may be imprecise. Priority may not be followed when considering a specific instruction.

Generally speaking, the exceptions described in the following sections are handled (“processed”) by hardware; these exceptions are then serviced by software.

Cold Reset Exception

Cause

The Cold Reset exception is taken for a power-on or "cold" reset; it occurs when the **SysGnt*** signal is asserted while the **SysReset*** signal is also asserted.[†] This exception is not maskable.

Processing

The CPU provides a special interrupt vector for this exception:

- location 0xBFC0 0000 in 32-bit mode
- location 0xFFFF FFFF BFC0 0000 in 64-bit mode

The Cold Reset vector resides in unmapped and uncached CPU address space, so the hardware need not initialize the TLB or the cache to process this exception. It also means the processor can fetch and execute instructions while the caches and virtual memory are in an undefined state.

The contents of all registers in the CPU are undefined when this exception occurs, except for the following register fields:

- In the *Status* register, *SR* and *TS* are cleared to 0, and *ERL* and *BEV* are set to 1. All other bits are undefined.
- *Config* register is initialized with the boot mode bits read from the serial input.
- The *Random* register is initialized to the value of its upper bound.
- The *Wired* register is initialized to 0.
- The *EW* bit in the *CacheErr* register is cleared.
- The *ErrorEPC* register gets the PC.
- The *FrameMask* register is set to 0.
- Branch prediction bits are set to 0.
- *Performance Counter* register *Event* field is set to 0.
- All pending cache errors, delayed watch exceptions, and external interrupts are cleared.

Servicing

The Cold Reset exception is serviced by:

- initializing all processor registers, coprocessor registers, caches, and the memory system
- performing diagnostic tests
- bootstrapping the operating system

[†] If **SysGnt*** remains deasserted (high) while **SysReset*** is asserted, the processor interprets this as a Soft Reset exception.

Soft† Reset Exception

Cause

The Soft Reset exception occurs in response to a Soft Reset (See Chapter 8, the section titled “Soft Reset Sequence”).

A Soft Reset exception is not maskable.

The processor differentiates between a Cold Reset and a Soft Reset as follows:

- A Cold Reset occurs when the **SysGnt*** signal is asserted while the **SysReset*** signal is also asserted.
- A Soft Reset occurs if the **SysGnt*** signal remains negated when a **SysReset*** signal is asserted.

In R4400 processor, there is no way for software to differentiate between a Soft Reset exception and an NMI exception. In the R10000 processor, a bit labelled *NMI* has been added to the *Status* register to distinguish between these two exceptions. Both Soft Reset and NMI exceptions set the *SR* bit and use the same exception vector. During an NMI exception, the *NMI* bit is set to 1; during a Soft Reset, the *NMI* bit is set to 0.

Processing

When a Soft Reset exception occurs, the *SR* bit of the *Status* register is set, distinguishing this exception from a Cold Reset exception.

When a Soft Reset is detected, the processor initializes minimum processor state. This allows the processor to fetch and execute the instructions of the exception handler, which in turn dumps the current architectural state to external logic. Hardware state that loses architectural state is not initialized unless it is necessary to execute instructions from unmapped uncached space that reads the registers, TLB, and cache contents.

The Soft Reset can begin on an arbitrary cycle boundary and can abort multicycle operations in progress, so it may alter machine state. Hence, caches, memory, or other processor states can be inconsistent: data cache blocks may stay at the refill state and any cached loads/stores to these blocks will hang the processor. Therefore, CacheOps should be used to dump the cache contents.

After the processor state is read out, the processor should be reset with a Cold Reset sequence.

---

† Soft Reset is also known colloquially as *Warm* Reset.

A Soft Reset exception preserves the contents of all registers, except for:

- *ErrorEPC* register, which contains the PC
- *ERL* bit of the *Status* register, which is set to 1
- *SR* bit of the *Status* register, which is set to 1 on Soft Reset or an NMI; 0 for a Cold Reset
- *BEV* bit of the *Status* register, which is set to 1
- *TS* bit of the *Status* register, which is set to 0
- PC is set to the reset vector 0xFFFF FFFF BFC0 0000
- clears any pending Cache Error exceptions

Servicing

A Soft Reset exception is intended to quickly reinitialize a previously operating processor after a fatal error.

It is not normally possible to continue program execution after returning from this exception, since a **SysReset*** signal can be accepted anytime.

NMI Exception

Cause

The NMI exception is caused by assertion of the **SysNMI*** signal.

An NMI exception is not maskable.

In R4400 processor, there is no way for software to differentiate between a Soft Reset exception and an NMI exception. In the R10000 processor, a bit labelled *NMI* has been added to the *Status* register to distinguish between these two exceptions. Both Soft Reset and NMI exceptions set the *SR* bit and use the same exception vector. During an NMI exception, the *NMI* bit is set to 1; during a Soft Reset, the *NMI* bit is set to 0.

Processing

When an NMI exception occurs, the *SR* bit of the *Status* register is set, distinguishing this exception from a Cold Reset exception.

An exception caused by an NMI is taken at the instruction boundary. It does not abort any state machines, preserving the state of the processor for diagnosis. The *Cause* register remains unchanged and the system jumps to the NMI exception handler (see Table 14-1).

An NMI exception preserves the contents of all registers, except for:

- *ErrorEPC* register, which contains the PC
- *ERL* bit of the *Status* register, which is set to 1
- *SR* bit of the *Status* register, which is set to 1 on Soft Reset or an NMI; 0 for a Cold Reset
- *BEV* bit of the *Status* register, which is set to 1
- *TS* bit of the *Status* register, which is set to 0
- PC is set to the reset vector 0xFFFF FFFF BFC0 0000
- clears any pending Cache Error exceptions

Servicing

The NMI can be used for purposes other than resetting the processor while preserving cache and memory contents. For example, the system might use an NMI to cause an immediate, controlled shutdown when it detects an impending power failure.

It is not normally possible to continue program execution after returning from this exception, since an NMI can occur during another error exception.

Address Error Exception

Cause

The Address Error exception occurs when an attempt is made to execute one of the following:

- reference to an illegal address space
- reference the supervisor address space from User mode
- reference the kernel address space from User or Supervisor mode
- load or store a doubleword that is not aligned on a doubleword boundary
- load, fetch, or store a word that is not aligned on a word boundary
- load or store a halfword that is not aligned on a halfword boundary

This exception is not maskable.

Processing

The common exception vector is used for this exception. The *AdEL* or *AdES* code in the *Cause* register is set, indicating whether the instruction caused the exception with an instruction reference, load operation, or store operation shown by the *EPC* register and *BD* bit in the *Cause* register.

When this exception occurs, the *BadVAddr* register retains the virtual address that was not properly aligned or that referenced protected address space. The contents of the *VPN* field of the *Context*, *XContext*, and *EntryHi* registers are undefined, as are the contents of the *EntryLo* register.

The *EPC* register contains the address of the instruction that caused the exception, unless this instruction is in a branch delay slot. If it is in a branch delay slot, the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set as indication.

Servicing

The process executing at the time is handed a UNIX^{TM} SIGSEGV (segmentation violation) signal. This error is usually fatal to the process incurring the exception.

TLB Exceptions

Three types of TLB exceptions can occur:

- TLB Refill occurs when there is no TLB entry that matches an attempted reference to a mapped address space.
- TLB Invalid occurs when a virtual address reference matches a TLB entry that is marked invalid.
- TLB Modified occurs when a store operation virtual address reference to memory matches a TLB entry which is marked valid but is not dirty (the entry is not writable).

The following three sections describe these TLB exceptions.

NOTE: TLB Refill vector selection is also described earlier in this chapter, in the section titled, TLB Refill Vector Selection.

TLB Refill Exception

Cause

The TLB refill exception occurs when there is no TLB entry to match a reference to a mapped address space. This exception is not maskable.

Processing

There are two special exception vectors for this exception; one for references to 32-bit address spaces, and one for references to 64-bit address spaces. The *UX*, *SX*, and *KX* bits of the *Status* register determine whether the user, supervisor or kernel address spaces referenced are 32-bit or 64-bit spaces; the TLB refill vector is selected based upon the address space of the address causing the TLB miss (user, supervisor, or kernel mode address space), together with the value of the corresponding extended addressing bit in the *Status* register (*UX*, *SX*, or *KX*). The current operating mode of the processor is not important except that it plays a part in specifying in which space an address resides. An address is in *user* space if it is in *useg*, *suseg*, *kuseg*, *xuseg*, *xsuseg*, or *xkuseg* (see the description of virtual address spaces in Chapter 13). An address is in *supervisor* space if it is in *sseg*, *ksseg*, *xsseg* or *xksseg*, and an address is in *kernel* space if it is in either *kseg3* or *xkseg*. *Kseg0*, *kseg1*, and kernel physical spaces (*xkphys*) are kernel spaces but are not mapped.

All references use these vectors when the *EXL* bit is set to 0 in the *Status* register. This exception sets the *TLBL* or *TLBS* code in the *ExcCode* field of the *Cause* register. This code indicates whether the instruction, as shown by the *EPC* register and the *BD* bit in the *Cause* register, caused the miss by an instruction reference, load operation, or store operation.

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers hold the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The *Random* register normally contains a valid location in which to place the replacement TLB entry. The contents of the *EntryLo* register are undefined. The *EPC* register contains the address of the instruction that caused the exception, unless this instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

To service this exception, the contents of the *Context* or *XContext* register are used as a virtual address to fetch memory locations containing the physical page frame and access control bits for a pair of TLB entries. The two entries are placed into the E*ntryLo0/ EntryLo1* register; the *EntryHi* and *EntryLo* registers are written into the TLB.

It is possible that the virtual address used to obtain the physical address and access control information is on a page that is not resident in the TLB. This condition is processed by allowing a TLB refill exception in the TLB refill handler. This second exception goes to the common exception vector because the *EXL* bit of the *Status* register is set.

TLB Invalid Exception

Cause

The TLB invalid exception occurs when a virtual address reference matches a TLB entry that is marked invalid (TLB valid bit cleared). This exception is not maskable.

Processing

The common exception vector is used for this exception. The *TLBL* or *TLBS* code in the *ExcCode* field of the *Cause* register is set. This indicates whether the instruction, as shown by the *EPC* register and *BD* bit in the *Cause* register, caused the miss by an instruction reference, load operation, or store operation.

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers contain the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The *Random* register normally contains a valid location in which to put the replacement TLB entry. The contents of the *EntryLo* registers are undefined.

The *EPC* register contains the address of the instruction that caused the exception unless this instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

A TLB entry is typically marked invalid when one of the following is true:

- a virtual address does not exist
- the virtual address exists, but is not in main memory (a page fault)
- a trap is desired on any reference to the page (for example, to maintain a reference bit)

After servicing the cause of a TLB Invalid exception, the TLB entry is located with TLBP (TLB Probe), and replaced by an entry with that entry's *Valid* bit set.

TLB Modified Exception

Cause

The TLB modified exception occurs when a store operation virtual address reference to memory matches a TLB entry that is marked valid but is not dirty and therefore is not writable. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *Mod* code in the *Cause* register is set.

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers contain the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The contents of the *EntryLo* register are undefined.

The *EPC* register contains the address of the instruction that caused the exception unless that instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

The kernel uses the failed virtual address or virtual page number to identify the corresponding access control information. The page identified may or may not permit write accesses; if writes are not permitted, a write protection violation occurs.

If write accesses are permitted, the page frame is marked dirty/writable by the kernel in its own data structures. The TLBP instruction places the index of the TLB entry that must be altered into the *Index* register. The *EntryLo* register is loaded with a word containing the physical page frame and access control bits (with the *D* bit set), and the *EntryHi* and *EntryLo* registers are written into the TLB.

Cache Error Exception

The Cache Error exception is described in Chapter 9, the section titled “Cache Error Exception”.

Virtual Coherency Exception

The Virtual Coherency exception is not implemented in the R10000 processor, since the virtual coherency condition is handled in hardware. When the hardware detects the Virtual Coherency exception, it invalidates the lines in all other segments of the primary cache that could cause aliasing. This takes six cycles more than that needed to refill the primary cache line (the refill would have occurred even if there was no Virtual Coherency exception detected).

In the R4400 processor, a Virtual Coherency exception occurs when a primary cache miss hits in the secondary cache but **VA[14:12]** are not the same as the *PIdx* field of the secondary cache tag, and the cache algorithm specifies that the page is cached. When such a situation is detected in the R10000 processor, the primary cache lines at the old virtual index are invalidated and the *PIdx* field of the secondary cache is written with the new virtual index.

Bus Error Exception

Cause

A Bus Error exception occurs when a processor block read, upgrade, or double/single/partial-word read request receives an external ERR completion response, or a processor double/single/partial-word read request receives an external ACK completion response where the associated external double/single/partial-word data response contains an uncorrectable error. This exception is not maskable.

Processing

The common interrupt vector is used for a Bus Error exception. The *IBE* or *DBE* code in the *ExcCode* field of the *Cause* register is set, signifying whether the instruction (as indicated by the *EPC* register and *BD* bit in the *Cause* register) caused the exception by an instruction reference, load operation, or store operation.

The *EPC* register contains the address of the instruction that caused the exception, unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

The physical address at which the fault occurred can be computed from information available in the CP0 registers.

- If the *IBE* code in the *Cause* register is set (indicating an instruction fetch reference), the instruction that caused the exception is located at the virtual address contained in the *EPC* register (or 4+ the contents of the *EPC* register if the *BD* bit of the *Cause* register is set).
- If the *DBE* code is set (indicating a load or store reference), the instruction that caused the exception is located at the virtual address contained in the *EPC* register (or 4+ the contents of the *EPC* register if the *BD* bit of the *Cause* register is set).

The virtual address of the load and store reference can then be obtained by interpreting the instruction. The physical address can be obtained by using the TLBP instruction and reading the *EntryLo* registers to compute the physical page number. The process executing at the time of this exception is handed a UNIX SIGBUS (bus error) signal, which is usually fatal.

Integer Overflow Exception

Cause

An Integer Overflow exception occurs when an ADD, ADDI, SUB, DADD, DADDI or DSUB instruction results in a 2’s complement overflow. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *OV* code in the *Cause* register is set.

The *EPC* register contains the address of the instruction that caused the exception unless the instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

The process executing at the time of the exception is handed a UNIX SIGFPE/ FPE_INTOVF_TRAP (floating-point exception/integer overflow) signal. This error is usually fatal to the current process.

Trap Exception

Cause

The Trap exception occurs when a TGE, TGEU, TLT, TLTU, TEQ, TNE, TGEI, TGEUI, TLTI, TLTUI, TEQI, or TNEI instruction results in a TRUE condition. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *Tr* code in the *Cause* register is set.

The *EPC* register contains the address of the instruction causing the exception unless the instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

The process executing at the time of a Trap exception is handed a UNIX SIGFPE/FPE_INTOVF_TRAP (floating-point exception/integer overflow) signal. This error is usually fatal.

System Call Exception

Cause

A System Call exception occurs during an attempt to execute the SYSCALL instruction. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *Sys* code in the *Cause* register is set.

The *EPC* register contains the address of the SYSCALL instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

If the SYSCALL instruction is in a branch delay slot, the *BD* bit of the *Status* register is set; otherwise this bit is cleared.

Servicing

When the System Call exception occurs, control is transferred to the applicable system routine. Additional distinctions can be made by analyzing the *Code* field of the SYSCALL instruction (bits 25:6), and loading the contents of the instruction whose address the EPC register contains.

To resume execution, the *EPC* register must be altered so that the SYSCALL instruction does not re-execute; this is accomplished by adding a value of 4 to the *EPC* register (*EPC* register + 4) before returning.

If a SYSCALL instruction is in a branch delay slot, a more complicated algorithm, beyond the scope of this description, may be required.

Breakpoint Exception

Cause

A Breakpoint exception occurs when an attempt is made to execute the BREAK instruction. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *BP* code in the *Cause* register is set.

The *EPC* register contains the address of the BREAK instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

If the BREAK instruction is in a branch delay slot, the *BD* bit of the *Status* register is set, otherwise the bit is cleared.

Servicing

When the Breakpoint exception occurs, control is transferred to the applicable system routine. Additional distinctions can be made by analyzing the *Code* field of the BREAK instruction (bits 25:6), and loading the contents of the instruction whose address the EPC register contains. A value of 4 must be added to the contents of the EPC register (EPC register + 4) to locate the instruction if it resides in a branch delay slot.

To resume execution, the EPC register must be altered so that the BREAK instruction does not re-execute; this is accomplished by adding a value of 4 to the EPC register (EPC register + 4) before returning.

If a BREAK instruction is in a branch delay slot, interpretation of the branch instruction is required to resume execution.

Reserved Instruction Exception

Cause

The Reserved Instruction exception occurs when one of the following conditions occurs:

- an attempt is made to execute an instruction with an undefined major opcode (bits 31:26)
- an attempt is made to execute a SPECIAL instruction with an undefined minor opcode (bits 5:0)
- an attempt is made to execute a REGIMM instruction with an undefined minor opcode (bits 20:16)
- an attempt is made to execute 64-bit operations in 32-bit mode when in User or Supervisor modes
- an attempt is made to execute a COP1X when the MIPS IV ISA is not enabled

64-bit operations are always valid in Kernel mode regardless of the value of the *KX* bit in the *Status* register.

This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *RI* code in the *Cause* register is set.

The *EPC* register contains the address of the reserved instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

Servicing

No instructions in the MIPS ISA are currently interpreted. The process executing at the time of this exception is handed a UNIX SIGILL/ILL_RESOP_FAULT (illegal instruction/reserved operand fault) signal. This error is usually fatal.

Coprocessor Unusable Exception

Cause

The Coprocessor Unusable exception occurs when an attempt is made to execute a coprocessor instruction for either:

- a corresponding coprocessor unit (CP1 or CP2) that has not been marked usable, or
- CP0 instructions, when the unit has not been marked usable and the process executes in either User or Supervisor mode.

This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *CpU* code in the *Cause* register is set. The contents of the *Coprocessor Usage Error* field of the coprocessor *Control* register indicate which of the four coprocessors was referenced. The *EPC* register contains the address of the unusable coprocessor instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

Servicing

The coprocessor unit to which an attempted reference was made is identified by the Coprocessor Usage Error field, which results in one of the following situations:

- If the process is entitled access to the coprocessor, the coprocessor is marked usable and the corresponding user state is restored to the coprocessor.
- If the process is entitled access to the coprocessor, but the coprocessor does not exist or has failed, interpretation of the coprocessor instruction is possible.
- If the *BD* bit is set in the *Cause* register, the branch instruction must be interpreted; then the coprocessor instruction can be emulated and execution resumed with the *EPC* register advanced past the coprocessor instruction.
- If the process is not entitled access to the coprocessor, the process executing at the time is handed a UNIX SIGILL/ILL_PRIVIN_FAULT (illegal instruction/privileged instruction fault) signal. This error is usually fatal.

Floating-Point Exception

Cause

The Floating-Point exception is used by the floating-point coprocessor. This exception is not maskable.

Processing

The common exception vector is used for this exception, and the *FPE* code in the *Cause* register is set.

The contents of the *Floating-Point Control/Status* register indicate the cause of this exception.

Servicing

This exception is cleared by clearing the appropriate bit in the *Floating-Point Control/Status* register.

Watch Exception

Cause

A Watch exception occurs when a load or store instruction references the physical address specified in the *WatchLo*/*WatchHi* System Control Coprocessor (CP0) registers. The *WatchLo* register specifies whether a load or store initiated this exception.

A Watch exception violates the rules of a precise exception in the following way: If the load or store reference which triggered the Watch exception has a cacheable address and misses in the data cache, the line will then be read from memory into the secondary cache if necessary, and refilled from the secondary cache into the data cache. In all other cases, cache state is not affected by an instruction which takes a Watch exception.

The CACHE instruction never causes a Watch exception.

The Watch exception is postponed if either the *EXL* or *ERL* bit is set in the *Status* register. If either bit is set, the instruction referencing the *WatchLo*/*WatchHi* address is executed and the exception is delayed until the delay condition is cleared; that is, until *ERL* and *EXL* both are cleared (set to 0). The *EPC* contains the address of the next unexecuted instruction.

A delayed Watch exception is cleared by system reset or by writing a value to the *WatchLo* register.[†]

Watch is maskable by setting the *EXL* or *ERL* bits in the *Status* register.

Processing

The common exception vector is used for this exception, and the *Watch* code in the *Cause* register is set.

Servicing

The Watch exception is a debugging aid; typically the exception handler transfers control to a debugger, allowing the user to examine the situation.

To continue program execution, the Watch exception must be disabled to execute the faulting instruction. The Watch exception must then be reenabled. The faulting instruction can be executed either by interpretation or by setting breakpoints.

[†] An MTC0 to the *WatchLo* register clears a delayed Watch exception.

Interrupt Exception

Cause

The Interrupt exception occurs when one of the eight interrupt conditions is asserted. The significance of these interrupts is dependent upon the specific system implementation.

Each of the eight interrupts can be masked by clearing the corresponding bit in the *Interrupt-Mask (IM)* field of the *Status* register, and all of the eight interrupts can be masked at once by clearing the *IE* bit of the *Status* register.

Processing

The common exception vector is used for this exception, and the *Int* code in the *Cause* register is set.

The *IP* field of the *Cause* register indicates current interrupt requests. It is possible that more than one of the bits can be simultaneously set (or even *no* bits may be set) if the interrupt is asserted and then deasserted before this register is read.

On Cold Reset, an R4400 processor can be configured with *IP[7]* either as a sixth external interrupt, or as an internal interrupt set when the *Count* register equals the *Compare* register. There is no such option on the R10000 processor; *IP[7]* is always an internal interrupt that is set when one of the following occurs:

- the *Count* register is equal to the *Compare* register
- either one of the two performance counters overflows

Software needs to poll each source to determine the cause of the interrupt (which could come from more than one source at a time). For instance, writing a value to the *Compare* register clears the timer interrupt but it may not clear *IP[7]* if one of the performance counters is simultaneously overflowing. Performance counter interrupts can be disabled individually without affecting the timer interrupt, but there is no way to disable the timer interrupt without disabling the performance counter interrupt.

Servicing

If the interrupt is caused by one of the two software-generated exceptions (described in Chapter 6, the section titled “Software Interrupts”), the interrupt condition is cleared by setting the corresponding *Cause* register bit, *IP[1:0],* to 0. Software interrupts are imprecise. Once the software interrupt is enabled, program execution may continue for several instructions before the exception is taken. Timer interrupts are cleared by writing to the *Compare* register. The Performance Counter interrupt is cleared by writing a 0 to bit 31, the overflow bit, of the counter.

Cold Reset and Soft Reset exceptions clear all the outstanding external interrupt requests**,** *IP[2]* to *IP[6]*.

If the interrupt is hardware-generated, the interrupt condition is cleared by correcting the condition causing the interrupt pin to be asserted.

14.4 MIPSIV Instructions

The system must either be in Kernel or Supervisor mode, or have set the *XX* bit of the *Status* register to a 1 in order to use the MIPS IV instruction set. In User mode, if *XX* is a 0 and an attempt is made to execute MIPS IV instructions, an exception will be taken. The type of exception that will be taken depends upon the type of instruction whose execution was attempted; a list is given in Table 14-4. Note that operating with MIPS IV instructions does not require that MIPS III instruction set or 64-bit addressing is enabled.

MIPS IV instructions that use or modify the floating-point registers (CP1 state) are also affected by the *CU1* bit of the CP0 Status register. If *CU1* is not set, a Coprocessor Unusable exception may be signaled.

The Reserved Instruction (RI), Coprocessor Unusable (CU), and Unimplemented Operation (UO) exceptions for MIPS IV instructions are listed in the Table 14-4 below.

Table 14-4 MIPS IV Instruction Exceptions

Exceptions	Instructions	CU1	MIPS4
RI	CPU (undefined)	-	-
RI	MOVN,Z	-	0
RI	MOVT,F	-	0
CU		0	1
RI	PREF	-	0
CU	COP1 (all instructions)	0	-
UO	(undefined)	1	-
RI	BC (cc>0)	1	0
UO	C (cc>0)	1	0
UO	MOVN,Z,T,F	1	0
UO	RECIP, RSQRT	1	0
RI	COP1X (all instructions)	-	0
CU	(all instructions)	0	1
RI	(undefined)	1	1

14.5 COP0 Instructions

Execution of an RFE instruction causes a Reserved Instruction exception in the R10000 processor.

The execution of undefined COP0 functions is undefined in the R10000 processor.

14.6 COP1 Instructions

The R10000 and R4400 processors do not generate the same exceptions for undefined COP1 instructions. In the R4400 processor, undefined opcodes or formats in the *sub* field take an Unimplemented Operation exceptions. In the R10000 processor, undefined opcodes (bits 25:24 are 0 or 1) take Reserved Instruction exceptions and undefined formats (bits 25:24 are 2 or 3) take Unimplemented Operation exceptions.

In MIPS II on an R4400 processor, the execution of DMTC1, DMFC1, and L format take Unimplemented Operation exceptions. In MIPS II on the R10000 processor, the execution of DMTC1 and DMFC1 take Reserved Instruction exceptions

The attempted execution of the L format takes an Unimplemented Operation exception when the MIPS III mode is not enabled.

A CTC1 instruction that sets both *Cause* and *Enable* bits also forces an immediate floating-point exception; the *EPC* register points to the offending CTC1 instruction.

14.7 COP2 Instructions

If the *CU2* bit of the CP0 *Status* register is not set during an attempted execution of such Coprocessor 2 instructions as COP2, LWC2, SWC2, LDC2, and SDC2, the system takes a Coprocessor Unusable exception.

In the R4400 processor, if the *CU2* bit is set, COP2 instructions are handled as NOPs; the operations of Coprocessor 2 load/store instructions are undefined. In the R10000 processor, an execution of a Coprocessor 2 instruction takes a Reserved Instruction exception when *CU2* bit is set.

15. Cache Test Mode

The R10000 processor provides a cache test mode that may be used during manufacturing test and system debug to access the following internal RAM arrays:

- data cache data array
- data cache tag array
- instruction cache data array
- instruction cache tag array
- secondary cache way predication table

15.1 Interface Signals

Cache test mode is accessed by using a subset of the system interface signals. By not requiring the use of any secondary cache interface signals, the internal RAM arrays may be accessed for single-chip LGA as well as R10000/secondary cache module configurations.

The following system interface signals are used during cache test mode:

- **SysAD(57:0)**
- **SysVal***

Any input signals not listed above are ignored by the processor when it is operating in cache test mode, and any output signals not listed above are undefined during cache test mode.

15.2 System Interface Clock Divisor

Cache test mode is supported for all system interface clock speeds. However, since cache test mode repeat rates and latencies are expressed in terms of **PClk** cycles, the external agent must take care when operating at any system interface clock divisor other than Divide-by-1.

15.3 Entering Cache Test Mode

In order for the processor to enter cache test mode, the external agent must begin a Power-on or Cold Reset sequence.

Rather than negating **SysReset*** at the end of the reset sequence, the external agent loads the mode bits into the processor by driving the mode bits (with the **CTM** signal asserted) on **SysAD(63:0)**, waits at least two **SysClk** cycles, and then asserts **SysGnt*** for at least one **SysClk** cycle.

After waiting at least another 100 ms, the external agent may issue the first cache test mode command.

Figure 15-1 shows the cache test mode entry sequence.

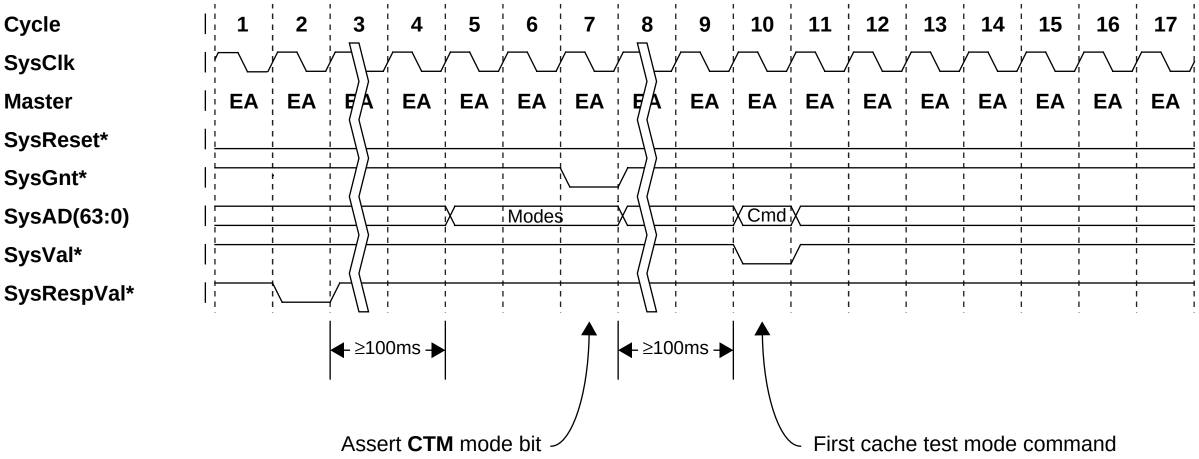

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
EA
SysReset*
SysGnt*
SysAD(63:0)
Modes
Cmd
SysVal*
SysRespVal*
≥100ms
≥100ms
Assert CTM mode bit
First cache test mode command

Figure 15-1 Cache Test Mode Entry Sequence

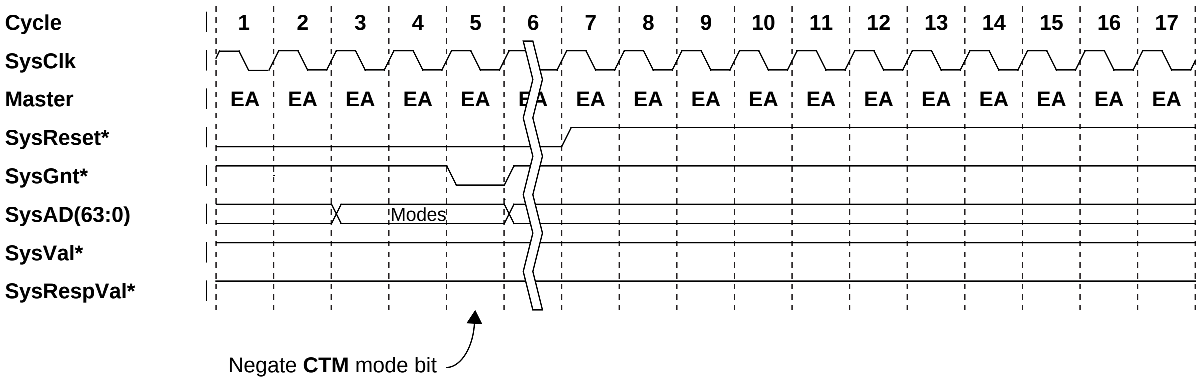
15.4 Exit Sequence

To leave cache test mode, the external agent does the following:

- loads the mode bits into the processor by driving the mode bits (with the **CTM** mode bit negated) on **SysAD(63:0)**
- waits at least two **SysClk** cycles
- asserts **SysGnt*** for at least one **SysClk** cycle

After at least one **SysClk** cycle, the external agent may negate **SysReset*** to end the reset sequence.

Figure 15-2 shows the cache test mode exit sequence.


Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
EA
SysReset*
SysGnt*
SysAD(63:0)
Modes
SysVal*
SysRespVal*

Negate **CTM** mode bit

Figure 15-2 Cache Test Mode Exit Sequence

15.5 SysAD(63:0) Encoding

Encoding of the **SysAD(63:0)** bus during cache test mode is shown in Table 15-1. “Unused” fields are read as “undefined,” and must be written as zeroes.

Table 15-1 Cache Test Mode **SysAD(63:0)** *Encoding*

SysAD Bit	**Data Cache Data Array**	**Data Cache Tag Array**	**Instruction Cache Data Array**	**Instruction Cache Tag Array**	**Secondary Cache Way Predication Array**
0	Data	Tag parity	Data	Tag parity	MRU
1		SCWay		Unused	Unused
2		State parity		State parity	
3		LRU		LRU	
4		Unused		Unused	
5					
6		State		State	
7				Unused	
31:8		Tag		Tag	
35:32	Data parity				
36	Unused	StateMod	Data parity	Unused	
38:37			Unused		
39		Unused			
42:40	0	1	2	3	4
	Array select				
43	Write/Read select				
44	Auto-increment select				
45	Way				
57:46	Address				
63:58	Unused				

15.6 Cache Test Mode Protocol

This section describes the cache test mode protocol in detail, including:

- normal write protocol
- auto-increment protocol
- normal read protocol
- auto-increment read protocol

Normal Write Protocol

A cache test mode **normal write** operation writes a selected RAM array. The write address, way, array, and data are specified in the write command.

The external agent issues a normal write command by:

- driving the address on **SysAD(57:46)**
- driving the way on **SysAD(45)**
- negating the auto-increment select on **SysAD(44)**
- asserting the Write/Read select on **SysAD(43)**
- driving the array select on **SysAD(42:40)**
- driving the write data on **SysAD(39:0)**
- asserting **SysVal*** for one **SysClk** cycle

Normal writes have a repeat rate of 8 **PClk** cycles.

Figure 15-3 depicts two cache test mode normal writes.

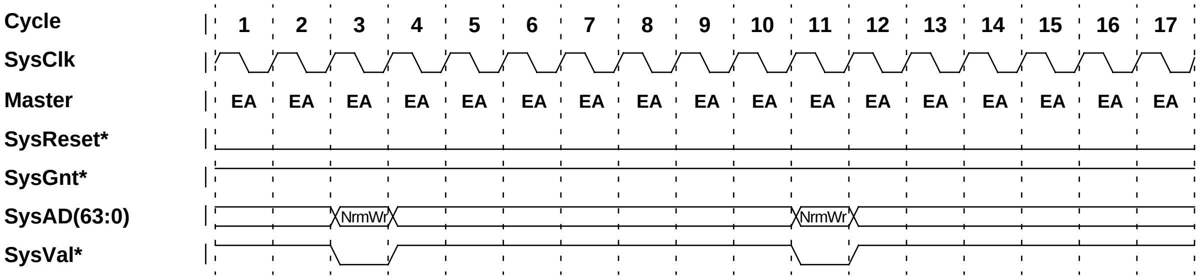

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
EA
SysReset*
SysGnt*
SysAD(63:0)
NrmWr
NrmWr
SysVal*

Figure 15-3 Cache Test Mode Normal Write Protocol

Auto-Increment Write Protocol

A cache test mode **auto-increment write** operation writes a selected RAM array. The write address is obtained by incrementing the previous write address, and the write way is obtained from the previous write way.

If an overflow occurs when incrementing the previous write address, the address wraps to 0, and the way is toggled.

The write data is identical to the previous write data.

For proper results, an auto-increment write must always be proceeded by a normal or auto-increment write.

The external agent issues an auto-increment write command by:

- asserting the auto-increment select on **SysAD(44)**
- asserting the Write/Read select on **SysAD(43)**
- driving the array select on **SysAD(42:40)**
- asserting **SysVal*** for one **SysClk** cycle

Auto-increment writes have a repeat rate of one **PClk** cycle.

Figure 15-4 depicts three cache test mode auto-increment writes.

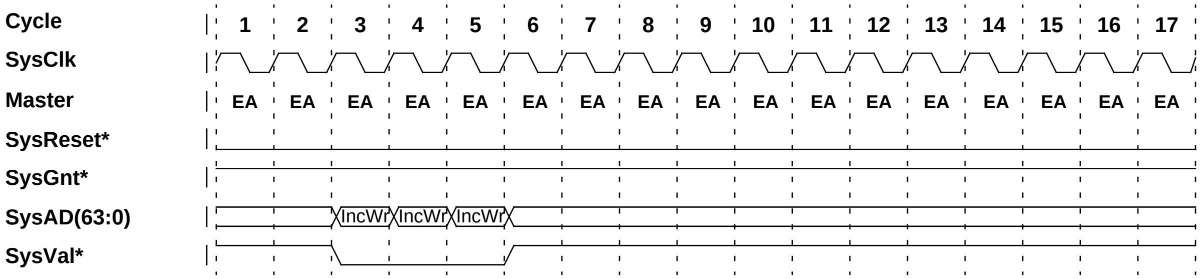

Cycle
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
SysClk
Master
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
EA
SysReset*
SysGnt*
SysAD(63:0)
IncWr
IncWr
IncWr
SysVal*

Figure 15-4 Cache Test Mode Auto-Increment Write Protocol

Normal Read Protocol

A cache test mode **normal read** operation reads a selected RAM array. The read address, way, and array are specified by the read command.

The external agent issues a normal read command by:

- driving the address on **SysAD(57:46)**
- driving the way on **SysAD(45)**
- negating the auto-increment select on **SysAD(44)**
- negating the Write/Read select on **SysAD(43)**
- driving the array select on **SysAD(42:40)**
- asserting **SysVal*** for one **SysClk** cycle.

After a read latency of 15 **PClk** cycles, the processor provides the read response by:

- entering *Master* state
- driving the read data on **SysAD(39:0)**
- asserting **SysVal*** for one **SysClk** cycle.

In the following **SysClk** cycle, the processor reverts to *Slave* state.

Normal reads have a repeat rate of 17 **PClk** cycles.

Figure 15-5 depicts two cache test mode normal reads.

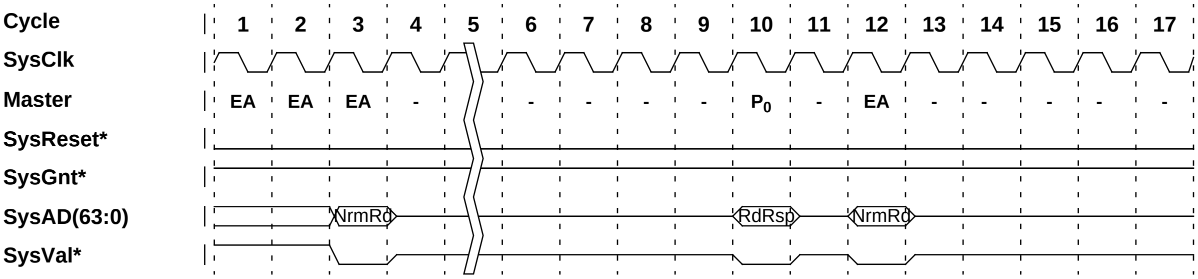

Cycle
SysClk
Master
SysReset*
SysGnt*
SysAD(63:0)
SysVal*
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
EA
EA
EA
-
-
-
-
-
P0
-
EA
-
-
-
-
-
NrmRd
RdRsp
NrmRd

Figure 15-5 Cache Test Mode Normal Read Protocol

Auto-Increment Read Protocol

A cache test mode **auto-increment read** operation reads a selected RAM array. The read address is obtained by incrementing the previous access address, and the read way is obtained from the previous access way.

If an overflow occurs when incrementing the previous access address, the address wraps to 0, and the way is toggled.

The external agent issues an auto-increment read command by:

- asserting the auto-increment select on **SysAD(44)**
- negating the Write/Read select on **SysAD(43)**
- driving the array select on **SysAD(42:40)**
- asserting **SysVal*** for one **SysClk** cycle.

After a read latency of 15 **PClk** cycles, the processor provides the read response by:

- entering *Master* state
- driving the read data on **SysAD(39:0)**
- asserting **SysVal*** for one **SysClk** cycle.

In the following **SysClk** cycle, the processor reverts to *Slave* state.

Auto-increment reads have a repeat rate of 17 **PClk** cycles.

Figure 15-6 depicts two cache test mode auto-increment reads.

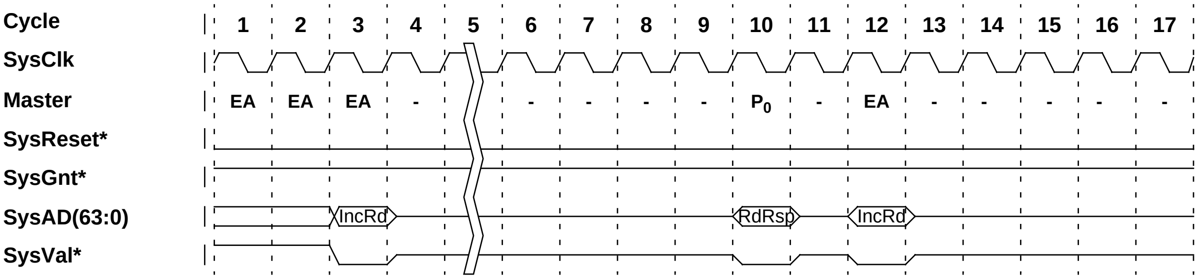


Figure 15-6 Cache Test Mode Auto-Increment Read Protocol

Appendix A Glossary

The following terms are defined in this Glossary:

- superscalar processor
- pipeline
- pipeline latency
- pipeline repeat rate
- out-of-order execution
- dynamic scheduling
- instruction fetch, decode, issue, execution, completion, and graduation
- active list
- free list and busy registers
- register renaming and unnaming
- nonblocking loads and stores
- speculative branching
- logical and physical registers
- register files
- ANDES architecture

A.1 Superscalar Processor

A superscalar processor is one that can fetch, execute and complete more than one instruction in parallel. By implication, a superscalar processor has more than one pipeline (see below).

A.2 Pipeline

In the processor **pipeline**, the execution of each instruction is divided into a sequence of simpler suboperations. Each suboperation is performed by a separate hardware section called a **stage**, and each stage passes its result to a succeeding stage.

Normally, each instruction only remains in each stage for a single cycle, and each stage begins executing a new instruction as previous instructions are being completed in later stages. Thus, a new instruction can often begin during every cycle.

Pipelines greatly improve the rate at which instructions can be executed, as long as there are no dependencies. The efficient use of a pipeline requires that several instructions be executed in parallel, however the result of any instruction is not available for several cycles after that instruction has entered the pipeline. Thus, new instructions must not depend on the results of instructions which are still in the pipeline.

A.3 Pipeline Latency

The **latency** of an execution pipeline is the number of cycles between the time an instruction is issued and the time a dependent instruction (which uses its result as an operand) can be issued.

In the R10000 processor, most integer instructions have a single-cycle latency, load instructions have a 2-cycle latency for cache hits, and floating-point addition and multiplication have a 2-cycle latency. Integer multiply, floating-point square-root, and all divide instructions are computed iteratively and have longer latencies.

A.4 Pipeline Repeat Rate

The **repeat rate** of the pipeline is the number of cycles that occur between the issuance of one instruction and the issuance of the next instruction to the same execution unit. In the R10000 processor, the main five pipelines all have repeat rates of one cycle, but the iterative units have longer repeat delays.

A.5 Out-of-Order Execution

The “program order” of instructions is the sequence in which they are fetched and decoded. In the R10000 processor, instructions may be issued, executed, and completed **out of program order**. They are always graduated in program order.

A.6 Dynamic Scheduling

The R10000 processor can issue instructions to functional units out of program order; this capability is known as **dynamic scheduling** or **dynamic issuing**.

The R10000 processor can dynamically issue an instruction as soon as all its operands are available and the required execution unit is not busy. Thus, an instruction is not delayed by a stalled previous instruction unless it needs the results of that previous instruction.

A.7 Instruction Fetch, Decode, Issue, Execution, Completion, and Graduation

In general, instructions are *fetched*, *decoded*, and *graduated* in their original program order, but may be *issued*, *executed*, and *completed* out of program order, as shown in Figure A-1.

- **Instruction fetching** is the process of reading instructions from the instruction cache.
- **Instruction decode** includes register renaming and initial dependency checks. For branch instructions, the branch path is predicted and the target address is computed.
- An instruction is **issued** when it is handed over to a functional unit for *execution*.
- An instruction is **complete** when its result has been computed and stored in a temporary physical register.
- An instruction **graduates** when this temporary result is committed as the new state of the processor. An instruction can graduate only after it and all previous instructions have been successfully completed.

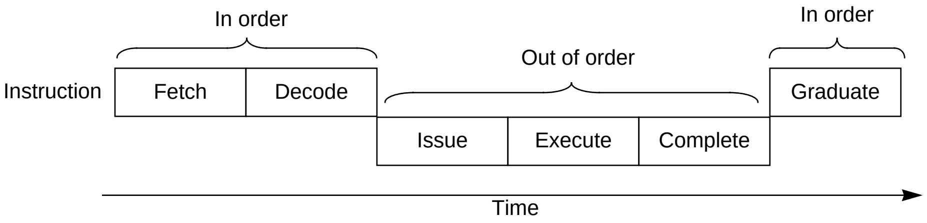


Figure A-1 Dynamic Scheduling

A.8 Active List

The R10000 processor’s **active list** is a program-order list of decoded instructions. For each instruction, the active list indicates the physical register which contained the *previous* value of the destination register (if any). If this instruction graduates, that previous value is discarded and the physical register is returned to the free list. The active list records status, such as those instructions that have completed, or those instructions that have detected exceptions. Instructions are appended to the bottom of the list as they are decoded and instructions are removed from the top as they graduate.

<R12000>

Active List entries are increased to 48:

The active list has been enlarged so that it now contains 48 entries.

Active list accepts conservatively

The read pointer for the active list is now evaluated on four-instruction blocks at a time. This has two effects:

a) There may be up to 11 empty slots in the active list and yet it will report to the decode unit that it cannot accept any new instructions. However this level of blockage only lasts for a single cycle. At most three empty slots will remain empty for more than one cycle. The time at which instructions are removed from the active list has also been changed. Integer and load/store instructions now remain in the list for one cycle after they graduate. This will be compensated for by the increased size of the active list.

b) The graduation of some instructions will be delayed, as the read pointer will not advance past the end of a four-instruction block during a cycle. Thus less than the maximum number of instructions might be graduated because the read pointer can get to them that cycle.

A.9 Free List and Busy Registers

A **busy-bit table** indicates whether or not a result has been written into each of the physical registers. Each register is initially defined to be busy when it is moved from the **free list** to the active list; the register becomes available ("not busy") when its instruction completes and its result is stored in the register file.

The busy-bit table is read for each operand while an instruction is decoded, and these bits are written into the queue with the instruction. If an operand is busy, the instruction must wait in the queue until the operand is "not busy." The queues determine when an operand is ready by comparing the register number of the result coming out of each execution unit with the register number of each operand of the instructions waiting in the queue.

With a few exceptions, the integer and address queues have integer operand registers, and the floating-point queue has floating-point operand registers.

A.10 Register Renaming

As it executes instructions, the processor generates a myriad of *temporary* register results. These temporary values are stored in register files together with *permanent* values. The temporary values become new permanent values when their corresponding instructions graduate.

Register renaming is used to resolve data dependencies during the dynamic execution of instructions.

To ensure each instruction is given correct operand values, the logical register numbers (**names**) used in the instruction are mapped to physical registers. Each time a new value is put in a logical register, it is assigned to a new physical register. Thus, each physical register has only a single value. Dependencies are determined using these physical register numbers.

An example of register renaming is shown below. The following Doubleword Shift Left Logical instruction,

opcode	rs	rt	dest	sa	function
spec	-	r2	r3	2	DSLL

```
DSLL              r3,r2,2
```

has one register operand (*r2*) plus a 5-bit shift count of value two stored in the *sa* field; the value in *r2* is shifted left by two and this value is stored in *r3*.

The physical execution of the instruction above, with register renaming, is given below:

```
Physical execution          Rename operation
p3←p2 shift left 2          r3 = p3
```

When the DSLL instruction is executed, the *l*ogical destination register *r3* is assigned a new physical register, p3, from the free list.

Register renaming also allows exceptions to be handled in a precise manner. *Out-of-order execution* means that an instruction can change its result register even before all prior instructions have been completed. However, if any of the prior instructions cause an exception, the original register value must be restored. Since each new register value is loaded into a new physical register (physical register values are not overwritten until the physical register is placed in the free list), previous values remain unchanged in the original physical registers and these previous values can be restored.†

An instruction can be aborted up until the time it graduates, and all register and memory values can be restored to a precise state following any exception. This state is restored by *unnaming* the temporary physical registers assigned to subsequent instructions.

Registers are **unnamed** by writing the old destination register into the mapping table and returning the new destination register to the free list. Unnaming is done in reverse program order, in case a logical register was used more than once. After renaming, the register files contain only the permanent values which were created by instructions prior to the exception.

Once an instruction has graduated, all previous values are lost.

A.11 Nonblocking Loads and Stores

Loads and stores are **nonblocking**; that is, cache misses do not stall the processor. All other parts of the processor may continue to work on non-dependent instructions while as many as four cache misses are being processed.

---

† This same technique is used to reverse mispredicted speculative branches.

A.12 Speculative Branching

Normally, about one of every six instructions is a branch. Since four instructions are fetched each cycle, the R10000 processor encounters, on average, a branch instruction every other cycle, as shown in Figure A-2.

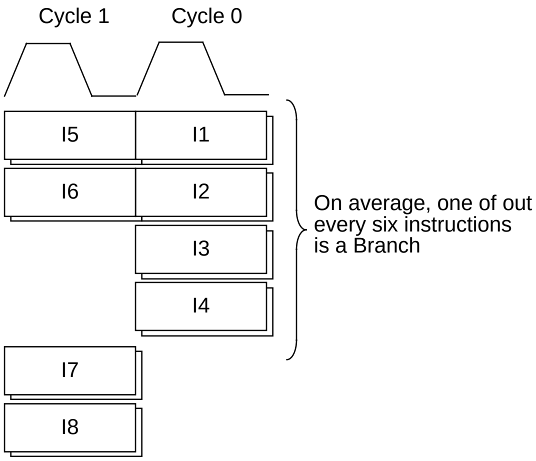


Figure A-2 Speculative Branching

When a branch instruction was encountered in previous processors, the instruction fetch and instruction issue halted until it was determined whether or not to take the branch. For instance, a branch delay slot was designed into the MIPS architecture to handle the intrinsic delay of a branch and to keep the pipeline filled.

Since the processor fetches up to four instructions each clock cycle, there is not enough time to resolve branches without stalling the fetch/decode circuitry. The processor therefore **predicts** the outcome of every branch and **speculatively executes** the branch based on this branch prediction.

The branch prediction circuit consists of a 512-entry RAM, using a 2-bit prediction scheme: two bits are assigned to a branch instruction, and indicate whether or not the branch was taken the last time it occurred. The four possible prediction states are: strongly taken, weakly taken, weakly not taken, strongly not taken. If the branch was taken the last two times, there is a good probability it will be taken this time too — or the inverse.†

The R10000 processor can speculate up to four branches deep. Shadow copies of the mapping tables are kept every time a prediction is made, allowing the R10000 processor to recover from a mispredicted branch in a single cycle.

---

† Simulations have shown the R10000 branch prediction algorithm to be over 90% accurate.

<R12000>

Use of global history in branch-prediction:

The history register is 8 bits wide, and implements the ‘gshare’ predictor (reference to paper that defines will be provided later). The history register is updated speculatively, with a one cycle delay after a prediction before the results are available for use in forming another prediction index. As mentioned earlier, some programs with small “working set of conditional branches” benefit significantly from the use of such hashing; however, a slightly variable number of previously-executed branches may be omitted from the predictions made for any given branch. This will reduce prediction accuracy somewhat. Global history register is enabled via bits 26:23 of the Diag Register (CP0 register 22). If bit 26 is set, branch prediction uses all eight bits of the global history register. If bit 26 is not set, then bits 25:23 specify a count of the number of bits of global history register to be used.

Increase in branch prediction table size:

The table size is increased to 2048 2-bit entries.

A.13 Logical and Physical Registers

Register renaming (described above) distinguishes between **logical registers**, which are referenced within instruction fields, and **physical registers**, which are actually located in the hardware register file. The programmer is only aware of logical registers; the implementation of physical registers is entirely transparent.

Logical register numbers are dynamically mapped onto physical register numbers. This mapping uses *mapping tables* which are updated after each instruction is decoded; each new result is written into a new physical register. This value is temporary and the previous contents of each logical register can be restored if its instruction must be aborted following an exception or a mispredicted branch.

Register renaming simplifies dependency checks. Logical register numbers can be ambiguous when instructions are executed out of order, since a succession of different values may be assigned to the same register. But physical register numbers uniquely identify each result, making dependency checking unambiguous.

The queues and execution units use physical register numbers. Integer and floating-point registers are implemented with separate renaming hardware and multi-port register files.

A.14 Register Files

The R10000 processor has two 64-bit-wide register files to store integer and floating-point values. Each file contains 64 registers. The integer register file has seven read and three write ports; the floating-point register file has five read and three write ports.

The integer and floating-point pipelines each use two dedicated operand ports and one dedicated result port in the appropriate register file. The Load/Store unit uses two dedicated integer operand ports for address calculation. It must also load or store either integer or floating-point values, sharing a result port and a read port in both register files.

These shared ports are also used to move data between the integer and floating-point register files, to store branch and link return addresses, and to read the target address for branch register instructions.

A.15 ANDES Architecture

The R10000 processor uses the MIPS ANDES architecture, or *Architecture with Non-sequential Dynamic Execution Scheduling*.

Appendix B Differences between R10000 and R12000

The following items are described in this Appendix:

- Mode bits changed in R12000
- DSD (Delay Speculative Dirty)
- Changes in the Branch Diag Register
- Eliminate traps for Denorm/NaN FP inputs
- Increase in pre-decode buffering
- Increased penalty for indirect branches
- Addition of a Branch Target Address Cache
- Use of global history in branch-prediction
- Increase in branch prediction table size
- Address calculation for load/store instructions uses integer queue
- Load/store dependency is speculatively ignored
- DCache set locking relaxed
- SC refill blocking reduced
- Increased the Way Prediction Table (MRU table) to 16K single-bit entries
- Additional cycles for System Interface transactions
- FP and Integer-Queue Issue Policy
- Active List entires are increased to 48
- Cache Error inhibits graduation
- Changed Spare (1, 3) pins to NC (No Connection)
- CacheOp Index Write Back Invalidate (D) also clears Primary Tag
- Summary of the differences

B.1 Mode bits changed in R12000

★ *Table B-1   Mode Bits 12:9 (SysClkDiv)*

Code	Divisor	SysClk (for PClk = 300 MHz)
`0000`	-	Reserved
`0001`	-	Reserved
`0010`	-	Reserved
`0011`	2	150 MHz
`0100`	2.5	120 MHz
`0101`	3	100 MHz
`0110`	3.5	85.70 MHz
`0111`	4	75.00 MHz
`1000`	4.5	66.00 MHz
`1001`	5	60.00 MHz
`1010`	5.5	54.55 MHz
`1011`	6	50.00 MHz
`1100`	-	Reserved[†]
`1101`	-	Reserved
`1110`	-	Reserved
`1111`	-	Reserved

[†] For R12000 and R12000L. This code can be set in the R12000A (See C.1).

★ *Table B-2   Mode Bits 21:19 (SCClkDiv)*

Code	Divisor	SCClk (for PClk = 300 MHz)
`000`	-	Reserved
`001`	-	Reserved
`010`	1.5	200MHz
`011`	2	150 MHz
`100`	2.5	120 MHz
`101`	3	100 MHz
`110`	-	Reserved
`111`	4	75 MHz (added for testing silicon)

Table B-3 Mode Bits 24:22

Code	Name	Comments
000	-	Reserved
001	-	Reserved
010	-	Reserved
011	-	Reserved
100	DSD	Delay Speculative Dirty - fix for speculative store (see B.2)
101	-	Reserved
110	-	Reserved
111	-	Reserved

B.2 DSD (Delay Speculative Dirty)

★ The Boot Mode bit 24 corresponds to the Config register[24] bit and this controls DSD during user mode. However, the DSD mode can also be enabled in the kernel mode by setting the Status register[24] bit. Config register[24] is read-only and can be set only at boot time.

If the DSD mode is set -

a) R12000 will not set the Dirty bit for a secondary cache block until the store instruction is the oldest in the Active List and is about to be executed. (An interrupt could cause a case where the dirty bit is set (store is no longer speculative), but the store does not immediately graduate. We believe this case should not cause any problem. This mode does prevent speculative stores from setting the dirty bit.)

b) This mode will have slightly lower performance due to the delay in the setting of the Dirty bit. This delay will occur just once per block refill from main memory, when it is necessary to set the dirty bit. Setting the bit requires about ten cycles; but usually the processor will continue to overlap execution of other instructions. Once a block becomes dirty in secondary cache, this mode has no performance effect.

c) In this mode, a miss in secondary cache, due to a store instruction which is not already the oldest in the pipeline, will cause a refill to the “clean exclusive” state. A hit to a shared line will immediately cause an upgrade to “clean exclusive”. Thus, bus operations (which are relatively slow) will still begin speculatively.

Independent of the DSD mode, R12000 will delay a “cached, non-coherent” load until it is the oldest instruction. This change is implemented because a speculative load accessing an unmapped “xkphys” address as “cached, non-coherent” might bring data into the secondary cache without the proper coherency checks.

R12000 is doing no changes to prevent it from speculatively refilling cache lines in shared or clean states except the “xkphys” case described above.

B.3 Changes in the Branch Diag Register

In R12000 two fields are added to the "*Diag Register*" - *CP0 Register 22*. One field is "*ghistory enable*", bits 26:23. The other is "*BTAC disable*", bit 27.

The definitions are:

- **Ghistory enable:**
  - If bit 26 is set, branch prediction uses all eight bits of the global history register.
  - If bit 26 is not set, then bits 25:23 specify a count of the number of bits of global history to be used.

Thus if bits 26:23 are all zero, global history is disabled.

The global history contains a record of the taken/not-taken status of recently executed branches, and when used is XOR'ed with the PC of a branch being predicted to produce a hashed value for indexing the BPT. Some programs with small "working set of conditional branches" benefit significantly from the use of such hashing, some see slight performance degradation.

- **BTAC disable:**

If bit 27 is set, the use of the Branch Target Address Cache (BTAC) is disabled. The BTAC is used to reduce the instruction fetch penalty of taken branches by providing the target address of fixed-address branch and jump instructions.

B.4 Eliminate traps for Denorm/NaN FP inputs

The R10000 currently takes Unimplemented Exception when an FPU gets a NaN or Denorm as an input. R12000 suppresses these traps whenever the FS bit is set in the FCSR. R12000 simply passes through NaN's and Denorm's when the bit is set. This change in no way affects the handling of QNaNs and Denorms when they are produced, it only changes the way they are handled when they are received as input operands.

Case of Denorm when the FS bit is set to 1: A Denorm received as an input to the FP unit is flushed to zero before the FP unit begins to process the operand. The behavior of the unit (when FS is 1) will be exactly that seen when the input is zero. Specifically, if the zero input would itself cause a trap (due to divide by zero, for example) then the that zero-generated trap will be taken. When a Denorm is seen at the input, the *Inexact bit* is set, except in the cases described below:

> The *Inexact bit* will not be set, even if FS=1 and a Denorm is seen on input, if the other input to the FP operation is a value which pre-determines the FP result (e.g. QNaN). When the result is not affected by the presence or absence of the Denorm input, the result is EXACT. Hence the Inexact bit should not be set, even if *Flush to Zero* mode is ON.

Case of QNaNs when the *FS bit* is set to 1: A QNaN received as an input operand for an FP unit will cause the unit to produce the standard QNaN (which is not necessarily same as the input QNaN). Note that FP units will not propagate the QNaN to the output, but will always produce the same, standard, QNaN.

When the *FS bit* is set to zero, the behavior will be exactly as in R10000.

When Denorms or QNaNs are produced by an FP operation, the behavior will be exactly as in R10000, regardless of the *FS bit* setting.

Handling of signalling NaNs will be unaffected by this change. Only the handling of input quiet NaNs and Denorms will be affected.

Arithmetic instructions (like *add/sub/madd/cvt/div/sqrt/recip/rsqrt*) will follow the above behavior in all respects.

There are some instructions that deserve special mention:

- *Mov, conditional mov* will not be affected by this mode, i.e. no exceptions based on QNaNs and Denorms. Denorms and QNaNs will be moved without generating an exception, regardless of the FS state. This behavior is unchanged from that of R10000.
- When FS=1, the *Abs, Neg* and *Compare* instructions will flush Denorm inputs to zero just as the arithmetic operations do. This is different from the behavior of the R8000, R4400 and R10000. In all cases where flushing the Denorm to zero made a difference in the result, the and inexact trap will be taken or the *Inexact bit* will be set. Compatibility with R4400 and R10000 can be achieved by setting FS=0.

- The behavior of FP to INT conversion instructions will change in that when FS=1, an input Denorm will be flushed to zero and the Inexact bit will be set. With other inputs, FP to INT conversion will not be affected by the FS mode bit. Previously, R10000 took an *unimplemented exception* whenever a conversion from an FP value would result in a value that cannot be represented in the target format. This will continue to be the case, with the noted exception of Denorm inputs.
- FP to FP convert instructions will be affected in the same way as arithmetic operations. That is, cvt FP to FP will not take exceptions on qNaN or Denorm inputs, if and only if FS=1.

The above changes in R12000 will allow the compilers and applications can do more aggressive optimizations during loop unrolling like if-conversion, speculative load execution and speculative code motion by making use of this feature. The change is gated by the FS bit so that strict IEEE-compliance is possible, as before, by setting the *FS bit* to zero.

B.5 Increase in pre-decode buffering

Up to 12 instruction may be buffered before being decoded. This should normally be invisible to the end user, but can be important when debugging systems in uncached-mode, since fetch and decode are now further de-coupled.

B.6 Increased penalty for indirect branches

Indirect branches, which were already an expensive operation, have become even more so. Instruction fetch now stalls for a minimum of 5 cycles, rather than the 4 for the R10000. This additional cycle of delay is seen by both *jr* and *jalr* instructions.

B.7 Addition of a *Branch Target Address Cache*

This 32-entry two-way set-associative cache holds the target addresses of previously-taken branches. When a branch is executed a hit in the *BTAC* eliminates the one-cycle fetch bubble with the R10000 experiences for every taken branch. However, if a branch which hits in the *BTAC* is actually predicted not-taken, then a one cycle fetch bubble is introduced where none was present before. Performance simulations indicate that the *BTAC* is a net win, but because of its “mixed-blessing” nature, a mechanism has been provided to disable it via software. (See description of changes to diag register)

B.8 Use of global history in branch-prediction

The *history register* is 8 bits wide, and implements the ‘*gshare*’ predictor (reference to paper that defines will be provided later). The *history register* is updated speculatively, with a one cycle delay after a prediction before the results are available for use in forming another prediction index. As mentioned earlier, some programs with small “working set of conditional branches” benefit significantly from the use of such hashing; however, a slightly variable number of previously-executed branches may be omitted from the predictions made for any given branch. This will reduce prediction accuracy somewhat. *Global history register* is enabled via bits 26:23 of the *Diag Register (CP0 register 22)*. If bit 26 is set, branch prediction uses all eight bits of the *global history register.* If bit 26 is not set, then bits 25:23 specify a count of the number of bits of *global history register* to be used.

B.9 Increase in branch prediction table size

The table size is increased to 2048 2-bit entries.

B.10 Address calculation for load/store instructions uses integer queue

When load, store, cacheop, or prefetch instructions are decoded, they are sent to both the AQ and IQ units. The IQ treats the address-calculate unit as a third “ALU” and issues instructions to it. When an instruction completes address calculation, the results are forwarded to the AQ. Unlike previously, if an address instruction must be retried for any reason, address calculation is not redone. If an the address queue is full, but the integer queue has free entries at the time a load/store instruction is decoded, the load/store is sent only to the integer queue. When the address queue has an available entry the calculated address is forwarded to that entry and the remainder of the load/store execution continues.

B.11 Load/store dependency is speculatively ignored

When a load follows a store in program-order, and the address of the load is known to the Address Queue (AQ) before the address of the store, then the AQ may speculatively issue the load to tag-check and data access. When the address of the store is determined, the AQ can undo the effects of the load through the use of the “soft-exception” mechanism. Since almost all loads which are actually dependent on previous stores use the same registers to form their addresses, normally either the two instructions are independent, or their addresses are resolved in program order, so the soft-exception should occur rarely.

B.12 DCache set locking relaxed

In R10000, when an AQ entry accesses a Dcache line, that line is locked into the cache until the entry graduates, so that the entry will not be removed from the cache until the access completes. If another entry which needs to access exactly the same line arrives in the AQ before the first completes, the two may share the lock. In this way, a line is locked in the cache until all access to it complete. In order to prevent a deadlock from arising, whenever a cache line is locked in this way, only the oldest AQ entry can obtain a lock on the other “way” of the same cache set, thus ensuring that forward progress can be made. This algorithm can cause problems, because often the oldest entry in the AQ is the one which already owns the lock on the first way - thus ensuring that no other entries can access the second way of the cache for that set index. For some algorithms, most notably FFT’s, this can cause severe performance degradation. R12000 allows an entry to obtain the lock on the second way of a set if it is the oldest entry which does not already own a lock. Thus, any entries which have already acquired a lock, including those locking the first way, will not prevent another, younger, entry from accessing that second way.

B.13 SC refill blocking reduced

In R10000, during the time that an SCache line is being refilled from system interface via the “incoming buffer (IB)”, no other accesses to the SCache are allowed. If the external interface sees an ACK to a line that is being refilled before the last words of the SCache line are received by R10000, this means that several cycles can elapse during which SCache access is blocked. By breaking the SCache refill transaction into 64-byte blocks, and allowing other requests to proceed during breaks between the blocks, this effect could be reduced. R12000 pulls in SCache lines with two “pause points.” This first occurs when R12000 receives the ACK for a request. If the first two quad-words are already valid in the Incoming Buffer at that time, then R12000 will proceed to refill the SCache with those two, and forward the results to the DCache or ICache at the same time as normal. The next two quad-words will be refilled as they return, thus continuing to block any other access to the SCache just as today. If however, when the initial ACK is received, the first two are not valid (i.e., either 0 or 1 quad-words are valid at that time) then R12000 will “pause” the SCache refill and wait for both of them to be brought in to the IB. Once the first half is filled in to the SCache, R12000 will again check the IB to see if an additional 3 quad-words are valid (thus 7 out of the 8 quad-words in the SCache line should have arrived into the IB). Until that is the case, R12000 will again “pause” the SCache refill and allow other accesses to reach the SCache. These two pauses allow for other requests to slip in during an SCache refill. Using only two pauses both simplifies the logic and reduces bus turnarounds.

B.14 Increased the Way Prediction Table (MRU table) to 16K single-bit entries

The size of the table has been increased to 16K entries, so that 4MB caches with 128B lines or 2MB caches with 64B lines can be fully mapped.

B.15 Additional cycles for System Interface transactions

All transactions which go through the system interface unit (in particular, SCache refills and writebacks) have one additional CPU-clock of latency added to them.

B.16 FP and Integer-Queue Issue Policy

The integer and floating-point queues are altered so that they are now composed of two 8-entry banks. Instructions are issued into the two banks in an alternating fashion. Each bank independently nominates instructions for the functional units. For each FU, the banks nominate the oldest instruction they contain which is ready to execute. If both banks nominate an instruction for a given FU, a winner is chosen by a priority bit which alternates between the two banks on each cycle.

B.17 Active List entries are increased to 48

The active list has been enlarged so that it now contains 48 entries.

B.18 Cache Error inhibits graduation

When a cache error is detected, all instruction graduation is inhibited on the following cycle. Since cache errors are rare, and an exception will occur soon afterwards, this should have minimal impact on performance.

B.19 Changed Spare(1, 3) pins to NC (No Connection)

The **Spare(1, 3)** are used in R12000 for diagnostic purpose and thus for R12000 should not be connected to anything.

B.20 CacheOp *Index Write Back Invalidate(D)* also clears Primary Tag

As a result of the CacheOp *Index Write Back Invalidate(D)* instruction, the Primary Tag is also cleared (set to zero) in addition to setting the cache state bits to zeros or (invalid) as described in **V_R5000**, **V_R10000 INSTRUCTION User’s Manual**.

B.21 Summary of the differences

- **Higher operation frequency.**
- **Core operating voltage for R12000 2.6V.**
- **Max case temperature for R12000 70°C.**
- **Less Power consumption.**
- **Increased options for PClk to SysClk and PClk to SCClk ratios.**
    - Added boot-time mode bits to allow processor upgrade without change in system interface and secondary cache interface frequency.
- **Added a mode in which the side effects of “Speculative Load/ Stores” are avoided.**
    - Speculative load/stores could cause problems in a system with non-coherent I/O. In this mode prevents the behavior that causes the side-effects with some trade-off in performance. This mode is optional and can be selected during boot-time.
- **Added an optional Branch Target Address Cache to reduce instruction fetch penalty.**
    - Since there are trade-offs, this feature can be disabled.
- **Added an optional “Global History Table” to improve branch prediction.**
    - Since not all the program benefit from this feature; so the feature can be disabled.
- **Added an option to eliminate traps for Denorm/NAN FP inputs**
    - This allows the compilers and applications to do more aggressive optimization. The change is optional if IEEE compliance is needed.
- **Quadrupled the branch prediction table size.**
- **Doubled the MRU table for SCache way prediction to improve SCache hit rate.**

- **Improved performance monitoring system.**
- **Increased Active list to 48 entries to improve performance.**
- **Changed the Spare(1,3) pins to NC (No Connection).**
- **Other miscellaneous changes to improve performance and simplify logic.**

★ Appendix C Differences between R12000 and R12000A

The following items are described in this Appendix:

- Mode bits changed in R12000A
- Changes in the Performance Counter Registers
- Summary of the differences

C.1 Mode bits changed in R12000A

Table C-1   Mode Bits 12:9 (SysClkDiv)

Code	**Divisor**	**SysClk (for PClk = 400 MHz)**
`0000`	-	Reserved
`0001`	-	Reserved
`0010`	-	Reserved
`0011`	2	200 MHz
`0100`	2.5	160 MHz
`0101`	3	133.3 MHz
`0110`	3.5	114.3 MHz
`0111`	4	100 MHz
`1000`	4.5	88.89 MHz
`1001`	5	80 MHz
`1010`	5.5	72.73 MHz
`1011`	6	66.67 MHz
`1100`	7	57.14 MHz
`1101`	-	Reserved
`1110`	-	Reserved
`1111`	-	Reserved

Table C-2   Mode Bits 21:19 (SCClkDiv)

Code	**Divisor**	**SCClk (for PClk = 400 MHz)**
`000`	-	Reserved
`001`	-	Reserved
`010`	1.5	266.7 MHz
`011`	2	200 MHz
`100`	2.5	160 MHz
`101`	3	133.3 MHz
`110`	-	Reserved
`111`	4	100 MHz (added for testing silicon)

NOTE: The selectable divisors of **PClk** to **SCClk** in the R12000A are the same as those in the R12000. Table C-2 is only for indication of actual frequencies of **SCClk** when each divisor is selected.

Table C-3   Mode Bits 30:29 (HSTL Mode)

Code	Comments
00	HSTL 1 on the outputs of the System interface HSTL 1 on the outputs of the secondary cache interface
01	HSTL 1 on the outputs of the System interface HSTL 2 on the outputs of the secondary cache interface
10	HSTL 2 on the outputs of the System interface HSTL 1 on the outputs of the secondary cache interface
11	HSTL 2 on the outputs of the System interface HSTL 2 on the outputs of the secondary cache interface

C.2 Changes in the Performance Counter Registers

In the R12000A, the syndrome bits that are generated from the data coming into the processor from the SCache are captured in a 9-bit register whenever there is a single or multiple bit error. Therefore this register will always contain the syndrome bits generated for the most recent error encountered. The register is uninitialized on power up and is not writable by any other means. Architecturally, the 9-bit register appears as bits 31:23 of the CP0 Performance Counter (Cop 25) Control register 0. These bits were previously unused. These 9 bits are read only bits. A write to this control register will not affect these bits.

The syndrome bits are generated for Secondary to Primary refills and Secondary to Main memory writebacks, but not for CacheOp reads from Secondary cache.

For details, see **11.20 Performance Counter Registers (25)**.

C.3 Summary of the differences

- **Higher operation frequency.**
- **Core operating voltage for R12000A 1.9V.**
- **Increased an option for PClk to SysClk ratio.**
- **Added options for HSTL modes of output pins.**
- **Added an error indication mechanism for received secondary cache data.**
- **Changed the packaging to plastic BGA.**
- **Added JTRST signal for asynchronous initialization of the TAP controller in the JTAG interface.**

Appendix D Index

Numerics

16-word, cache refill
    read sequence ... 88
    write sequence ... 93
32-bit
    address space ... 287
    mode, TLB entry format ... 299
32-word, cache refill
    read sequence ... 88
    write sequence ... 93
4-word, cache refill
    read sequence ... 86
    write sequence ... 91
64-bit
    address space ... 287
    mode, TLB entry format ... 299
8-word, cache refill
    read sequence ... 87
    write sequence ... 92

A

access privileges, address space ... 296
ACK completion response ... 146
ACK, signal ... 106
active list, definition of ... 339
add unit, FPU ... 274
address
    encodings, mode ... 287
    Kernel mode ... 292
    mapping
        Kernel mode ... 292
        Supervisor mode ... 290
        User mode ... 288
    mode ... 287
    page ... 298
    queue ... 22, 29
        instruction graduation ... 29
        issue ports ... 29
        number of entries ... 29
        number of instructions written per cycle ... 29
        organized as FIFO ... 29
        sequencing ... 29
    space
        access privileges ... 296
        kernel ... 287
        supervisor ... 287
        user ... 287
        virtual ... 287
    Supervisor mode ... 290
    translation ... 300
    User mode ... 288
Address Error exception ... 310
Address Space Identifier, *see also* ASID ... 300
address/data bus signals ... 57
AdEL, indication ... 310
AdES, indication ... 310
algorithms
    cache, five types of ... 70, 74
aliasing, virtual ... 84
allocate request number requests, external ... 150
ALU (arithmetic logic unit)
    No. 1 ... 36
    No. 2 ... 36
ALU1 ... 25, 28
ALU2 ... 25, 28
ANDES, Architecture with Non-sequential Dynamic Execution Scheduling ... 20, 344
arbitration protocol, System interface ... 124

arbitration rules, System interface ... 125

arbitration signals ... 57

arbitration, cluster bus ... 98

Architecture with Non-sequential Dynamic Execution Scheduling, *see also* ANDES ... 344

arithmetic instructions, FPU ... 283

arithmetic logic unit, *see also* ALU ... 36

array ... 79

array, page table entry (PTE) ... 215

ASID (Address Space Identifier)
- context switch ... 300
- relationship to Global (G) bit in TLB entry ... 300
- stored in EntryHi register ... 300

ASID, field ... 219

auto-increment read, cache test mode ... 336

auto-increment write, cache test mode ... 334

B

Bad Virtual Address register (BadVAddr) ... 218

BadVAddr register ... 215, 233, 310

BadVPN2, field ... 215, 233

BD, (branch delay) bit ... 226, 228

BE, (memory endianness) bit ... 230

BEV, (boot exception vector) bit ... 188, 224, 302

block
- instruction cache ... 25
- primary data cache ... 25
- secondary cache ... 27
- size
  - primary data cache ... 64
  - primary instruction cache ... 62
  - secondary cache ... 67

block data transfers ... 110
- external block data responses ... 110
- processor block write requests ... 110
- processor coherency data responses ... 110

boundary scan register, JTAG ... 206

BPIdx, field ... 236

BPMode, field ... 236

BPOp, field ... 236

BPState, field ... 236

branch
- determining next address ... 35
- instruction, limits on execution ... 35
- prediction ... 32, 47, 342
- prediction rates, improving ... 39
- speculative ... 342
- unit ... 26, 35

BRCH, field ... 236

BRCV, field ... 236

BRCW, field ... 236

Breakpoint exception ... 320

BSIdx, field ... 236

BTAC disable, bit ... 237

buffer
- cached request ... 105
- cluster request ... 105
- incoming ... 105, 106
- outgoing ... 105, 107
- uncached ... 105, 108

bus
- SysAD ... 118
- SysCmd ... 111
- SysResp ... 121
- SysState ... 120

Bus Error exception ... 316

busy-bit table ... 340

bypass register, JTAG ... 205

C

C, (coherency attribute) bit ... 213

cache... 20
- algorithms ... 70
  - and processor requests ... 74
  - cacheable coherent exclusive on write, description of ... 71
  - cacheable coherent exclusive, description of ... 71
  - cacheable noncoherent, description of ... 71
  - fields, encoding of ... 70
  - for kseg0 address space ... 70
  - for mapped address space ... 70
  - for xkphys address space ... 70
  - uncached accelerated, description of ... 72
  - uncached, description of ... 71
  - where specified ... 70
- associativity ... 61
- block ownership ... 75
- misses ... 43
  - address recording ... 253
- nonblocking ... 41, 43
- ordering constraints ... 33
- pages ... 298
- primary ... 20
- primary data ... 25
  - block size ... 64

    - changing states ... 65
    - description of ... 64
    - diagram, state ... 66
    - error handling ... 192
    - index and tag ... 65
    - interleaving ... 48
    - refill ... 47
    - state diagram ... 66
    - states ... 65
    - subset of secondary cache ... 65
    - write back protocol ... 64
  - primary instruction ... 25
    - block size ... 62
    - description of ... 62
    - diagram, state ... 63
    - error handling ... 191
    - error protection ... 191
    - index and tag ... 62
    - refill ... 47
    - state diagram ... 63
    - states ... 62
  - rules, ownership of a cache block ... 75
  - secondary ... 20
    - associativity ... 27, 67
    - block size ... 67
    - block state ... 84
    - blocks ... 27
    - changing states ... 68
    - clock domain ... 173
    - data array ... 77
    - data array width ... 79
    - description of ... 67
    - diagram, state ... 68
    - ECC ... 27
    - error handling ... 193
    - index and tag ... 67
    - indexing ... 79
    - indexing the data array ... 79
    - indexing the tag array ... 80
    - interface frequencies ... 78
    - sizes ... 27
    - specifying block size ... 77
    - specifying cache size ... 77
    - state diagram ... 68
    - states ... 67
    - syndrome bits ... 254
    - tag ... 83
    - tag and data array ECC ... 77
    - tag array ... 77
    - way prediction ... 81
    - way prediction table ... 80
    - write back protocol ... 67
  - strong ordering
    - example of ... 34
  - structure, two-level ... 61
- Cache Error exception... 188, 315
  - precision ... 188
  - prioritization ... 188
- Cache Error handler ... 188
- CACHE instructions ... 189
  - support for I/O ... 168
- cache miss stalls ... 43
- cache test mode
  - entry ... 330
  - exit ... 331
- cacheable coherent exclusive on write, cache algorithm ... 70, 71
- cacheable coherent exclusive, cache algorithm ... 70, 71
- cacheable noncoherent, cache algorithm ... 70, 71
- cached request buffer ... 105
- CacheErr register ... 188, 189, 191, 192, 263
- cause bits, FPU ... 283
- Cause register ... 121, 122, 218, 226, 228
- Cause, field (FP) ... 283
- CE, bit ... 223, 224, 226
- CH, bit ... 224
- chip revisions, R10000 ... 229
- ckseg0 space ... 296
- ckseg1 space ... 296
- ckseg3 space ... 296
- cksseg space ... 296
- clock
  - domain
    - in secondary cache ... 173
    - internal processor clock domain ... 171
    - secondary cache clock domain ... 171
    - System interface clock domain ... 171
  - signal
    - PClk ... 172
    - SCClk ... 173
    - SysClk ... 171
    - SysClkRet ... 172
  - signals, overview of ... 57
- clock divisor, system interface ... 96, 329
- cluster bus ... 52, 98
  - operation ... 164
- cluster coordinator ... 97, 98

cluster request buffer ... 105

coherency conflicts ... 159

coherency protocol, directory-based ... 169

coherency request, external ... 154, 156

coherency schemes ... 52

coherency, System interface
    external intervention exclusive request ... 157
    external intervention shared request ... 157
    external invalidate request ... 157

CohPrcReqTar, mode bit ... 118, 165, 168, 180

cold reset ... 175
    sequence ... 178

Cold Reset exception ... 302

Compare register ... 122, 218

completing, an instruction ... 339

completion, definition of ... 341

condition bit dependencies ... 32

Condition, field (FP) ... 283

Config register ... 230

conflicts
    coherency ... 159
        internal ... 159
    TLB, avoiding ... 300

Context register ... 215, 233

context switch ... 300

control registers, FPU ... 281

controller, TAP ... 204

coordinator, cluster ... 97

COP1 instructions ... 327

COP2 instructions ... 327

Coprocessor 0, *see also* CP0 ... 209

Coprocessor 1 *see also* CP1, COP1 ... 225

Coprocessor 2 *see also* CP2, COP2 ... 225

Coprocessor 3 *see also* CP3, COP3 ... 225

Coprocessor Unusable exception ... 322

correctable error ... 185

Count register ... 122, 218

CP0 (coprocessor 0) ... 209
    instructions ... 327
    registers, list of ... 210

csseg space ... 291

CT, bit ... 230

CTM, mode bit ... 182, 330, 331

CU, (coprocessor usability) field ... 220, 222, 225

D

D, (dirty) bit ... 213

data cache
    see also cache, primary data ... 64

data dependencies ... 38

data path, secondary cache ... 27

data quality indication ... 108

DBRC, field ... 236

DC, (data cache size) field ... 230

DCOk, signal ... 54, 176

DE, bit ... 189, 224

debugging, and Watch registers ... 232

decoding, an instruction ... 339

dependencies
    condition bit ... 32
    exception ... 33
    instruction ... 31
    memory ... 32
    pipeline ... 31
    register ... 32, 343

DevNum, mode bits ... 180

Diagnostic register ... 235

directory-based coherency protocol ... 169

divide unit, FPU ... 274

division by zero, FP ... 283

divisor, clock, system interface ... 96, 329

DN, (device number) field ... 230

Done, bit ... 28

done, *see also* completion ... 341

DP, (primary data cache parity) field ... 262

DS, (diagnostic status) field ... 221, 222, 223

DSD, (delay speculative dirty) bit ... 224, 230

DSD, mode bits ... 181

duplicate, external ... 50

dynamic issue ... 31, 339

dynamic scheduling ... 339

E

EC, field ... 230

ECC (error correcting code)
    matrix for secondary cache data array ... 194
    matrix for secondary cache tag array ... 196
    matrix for System interface ... 199
    register ... 262
    secondary cache ... 27

ECC register ... 86, 91

ECC, field ... 262

efficiency, program, suggestions for increasing ... 39

Enable, field (FP) ... 283

EntryHi register ... 219, 299
    ASID field in ... 300

EntryLo registers, and FrameMask register ... 234

EntryLo0 register ... 213, 299

EntryLo1 register ... 213, 299

EPC register ... 228

ERL, (error level) bit ... 188, 223, 286

ERR completion response ... 146

ERR, signal ... 106

error
    correctable ... 185
    handling ... 184
        protocol ... 202
    levels, in the Status register ... 286
    protection ... 184
        schemes used in R10000 ... 190
    protection schemes, used in R10000
        ECC ... 190
        parity ... 190
        sparse encoding ... 190
    uncorrectable ... 186
        handling an ... 188
        limiting the propagation of ... 187
    units that detect and report uncorrectable errors ... 188

error correcting code *see also* ECC ... 190

Error Exception Program Counter (ErrorEPC) register ... 273

Event, field ... 239, 248

EW, bit in CacheErr register ... 189

ExcCode, field ... 226, 227

exception levels, in the Status register ... 286

exception processing, CPU
    exception types
        Address Error ... 310
        Breakpoint ... 320
        Bus Error ... 316
        Cache Error ... 188, 315
        Coprocessor Unusable ... 322
        Floating-Point ... 323
        Integer Overflow ... 317
        Interrupt ... 325
        NMI ... 309
        Reserved Instruction ... 321
        Soft Reset ... 307
        System Call ... 319
        TLB ... 311
        TLB Invalid ... 311, 313
        TLB Modified ... 311, 314
        TLB Refill ... 311, 312
        Trap ... 318
        Virtual Coherency ... 315
        Watch ... 324
    exception vector location
        Reset ... 302
        TLB Refill ... 302
    exception vector selection ... 303
    precise handling ... 33
    priority of ... 303, 305
    TLB refill vector locations ... 304

Exception Program Counter (EPC) register ... 228

executing, an instruction ... 339

execution order ... 31

execution pipelines ... 22

execution units, iterative ... 344

execution, speculative ... 38, 342

EXL, (exception level) bit ... 223, 228, 286, 302

external ACK completion response ... 106, 146

external agent ... 50, 51, 95
    also referred to as cluster coordinator ... 97
    connecting to ... 97

external allocate request number request protocol ... 150

external block data response ... 110, 144
    protocol ... 143

external coherency conflicts ... 160

external coherency request latency ... 162

external coherency requests, action taken ... 158

external completion response ... 147
    protocol ... 146

external double/single/partial-word data response protocol ... 145

external duplicate tags, support for ... 168

external interface ... 27
    memory accesses ... 48
    priority operations ... 48

external interrupt request ... 121
    protocol ... 152

external intervention exclusive request ... 157

external intervention request ... 149
    protocol ... 149

external intervention shared request ... 157

external invalidate requestt ... 157

- protocol ... 151

external NACK completion response ... 146

external request ... 96, 103
- protocol ... 148

external response ... 96, 103
- protocol ... 143

F

fetch pipeline ... 22, 35

fetching, an instruction ... 339

FGR (Floating-Point General register)
- 32-bit operations ... 277
- 5-bit select ... 277
- 64-bit operations ... 277
- load operations ... 278
- operations ... 277
- Status register FR bit ... 277
- store operations ... 278

Fill, field ... 219

flag
- uncorrectable error ... 106

Flag, field (FP) ... 283

floating-point
- adder ... 36
- adder pipeline ... 22
- divide ... 36, 275
- multiplier ... 36
- pipeline ... 23
- queue ... 22, 29
  - instructions written each cycle ... 29
  - number of allowable entries ... 29
  - ports ... 29
  - sequencing ... 29
- registers ... 277
- rounding mode ... 284
- square root ... 36

Floating-Point exception ... 323

Floating-Point Status register *see also* FSR ... 282

Floating-Point Unit, *see also* FPU ... 274

flow control ... 109
- external data response ... 109
- external request ... 109
- processor coherency data response ... 109
- processor eliminate request ... 109
- processor read request ... 109
- processor upgrade request ... 109
- processor write request ... 109

- signals ... 57

format, TLB entry ... 299

FPU ... 274
- Active List, control of FSR ... 282
- add unit ... 274
- arithmetic instructions ... 283
- cause bits, FSR ... 283
- changing rounding mode using a CTC1 ... 284
- compare ... 283
- condition bits ... 283
- control registers ... 281
- divide unit ... 274
- FGRs (general registers) ... 277
- FSR, (Status register in FPU) ... 282
- graduation, control of FSR ... 282
- latency ... 274
- logic diagram ... 275
- move to floating-point ... 280
- multiply unit ... 274
- operations ... 275
- queue
  - controlling units ... 276
  - move unit, FPU ... 275
- read ports ... 275
- register file ... 275
- repeat rate ... 274
- rounding modes ... 284
- serial dependency circuit ... 280
- square-root unit ... 274

FR, field ... 222

FrameMask register ... 214, 234

free list ... 340

freeing the request number, with completion response ... 146

FSR (Floating-Point Status register)
- cause bits ... 283
- condition bits ... 283
- division by zero ... 283
- enable bits ... 283
- flag bits ... 283
- inexact result ... 283
- invalid operation ... 283
- load exceptions ... 284
- loading the FSR ... 284
- overflow ... 283
- RM, round to minus infinity ... 284
- RN, round to nearest representable value ... 284
- RP, round to plus infinity ... 284
- RZ, round toward zero ... 284
- underflow ... 283

  unimplemented operation ... 283

functional unit ... 25
  branch ... 26
  floating-point adder ... 25
  floating-point multiplier ... 25
  instruction decode and rename ... 26
  integer ALU ... 25
  iterative ... 25
  Load/Store Unit ... 25

G

G, (Global) bit in TLB ... 214, 300

gathering data, in identical mode ... 108

gathering data, in sequential mode ... 108

ghistory enable, bit ... 237

global processes (G bit in TLB) ... 300

graduation
  definition of ... 341
  of an instruction ... 339

Grant parking ... 124

H

hardware emulation, support for ... 170

hardware interrupts ... 121

HSTL Mode, mode bits ... 183

I

I/O, support for ... 168

IC, (instruction cache size) field ... 230

IE, (interrupt enable) bit ... 223, 239, 249

IM, (interrupt mask) field ... 221

implementation number, R10000 processor ... 229

incoming buffer ... 105, 106

Index Load Tag instruction ... 89

Index register ... 211

Index Store Data CACHE instruction ... 91

Index Store Tag CACHE instruction ... 94

indexing, the secondary cache ... 79

inexact result (FP) ... 283

initialization ... 175

instruction
  CACHE, *see also* CACHE instructions ... 189
  completion ... 38, 339
  COP0 *see also* CP0 ... 327
  COP1 ... 327
  COP2 ... 327
  decoding ... 339
  dependencies ... 31
  DMFC1 ... 283
  execution ... 339
  fetching ... 339
  graduation ... 339
  issue ... 38, 339
    superscalar ... 38
  latencies ... 45
  MFC1 ... 280, 283
  prefetch ... 43
  queue ... 28, 35
  repeat rates ... 45
  serializing ... 41
  SWC1 ... 280
  SYNC ... 73, 164

instruction cache, block size *see also* cache, primary instruction ... 62

instruction register, JTAG ... 205

integer
  queue 28
    branch instructions ... 28
    divide instructions ... 28
    multiply instructions ... 28
    ports ... 28
    shift instructions ... 28

integer ALU pipeline ... 22

Integer Overflow exception ... 317

integer queue ... 22

interface, external ... 27

internal coherency conflicts ... 159

internal processor clock domain ... 172

Interrupt exception ... 325

interrupt mask, bit ... 218

Interrupt register ... 121

interrupt request, external ... 121

interrupts ... 121
  hardware ... 121
  nonmaskable ... 122
  software ... 122
  timer ... 122

invalid operation, FP ... 283

invalidate request, external ... 151

IP, (interrupt pending) bit ... 226, 262

ISA (Instruction Set Architecture)
  MIPS I ... 18

  MIPS II ... 18
  MIPS III ... 18
  MIPS IV ... 18, 277
issue, dynamic ... 339
issuing, an instruction ... 339
iterative execution units ... 344
ITLB (instruction TLB) ... 300
ITLBM, field ... 236

J

JTAG
  boundary scan register ... 206
  bypass register ... 205
  Capture-DR state ... 206
  instruction register ... 205
  interface ... 203
    instruction register ... 205
    JTCK signal ... 204
    JTDI signal ... 204
    JTDO signal ... 204
    JTMS signal ... 204
    JTRST signal ... 204
    Tap controller ... 204
    test access port ... 204
  Shift-DR state ... 205, 206
  signals ... 59
  Update-DR state ... 206
  Update-IR state ... 205
JTCK, signal ... 59, 60, 204
JTDI, signal ... 59, 60, 204, 205
JTDO, signal ... 59, 204, 205
JTLB (joint TLB) ... 300
JTMS, signal ... 59, 60, 204
JTRST, signal ... 59, 60, 204

K

K0, field ... 230
Kernel mode ... 286
  address mapping ... 292
  ckseg0 space ... 296
  ckseg1 space ... 296
  ckseg3 space ... 296
  cksseg space ... 296
  kseg0 space ... 293
  kseg1 space ... 293
  kseg3 space ... 293
  ksseg space ... 293
  kuseg space ... 293
  operations ... 292
  xkphys space ... 294
  xkseg space ... 296
  xksseg space ... 294
  xkuseg space ... 294
kseg0 space ... 293
Kseg0CA, mode bits ... 180
kseg1 space ... 293
kseg3 space ... 293
ksseg space ... 293
KSU, field ... 221, 223, 302
kuseg space ... 293
KX, bit ... 222, 286

L

latency ... 45
  accessing secondary cache ... 47
  definition of ... 338
  external coherency request ... 162
  FPU ... 274
least-recently used replacement algorithm (LRU) ... 25
list, free ... 340
LLAddr register ... 231
load operations, FPU registers ... 278
Load/Store Unit pipeline ... 22
loads
  nonblocking ... 341
logic diagram, FPU ... 275
logical register
  initialization (necessity for) ... 176
logical register, *see also* physical register ... 343
LRU (least-recently used) replacement algorithm ... 25

M

mapped, virtual address region ... 287
mapping table ... 343
Mask, field ... 216
master state ... 97
  and flow control ... 109
matches, multiple, in TLB ... 300
MemEnd, mode bits ... 181
memory dependencies ... 32
memory ordering ... 33
memory protection ... 298

MIPS III ISA, disabled and enabled ... 214
MIPS IV, instruction set *see also* ISA ... 326
miscellaneous system signals ... 58
mispredicted branch ... 47
mode
  addressing ... 287
  addressing, encodings ... 287
    Kernel mode ... 287
    Supervisor mode ... 287
    User mode ... 287
  operating ... 286
mode bits ... 180
  CohPrcReqTar ... 118, 165, 168, 180
  CTM ... 182, 330, 331
  DevNum ... 180
  DSD ... 181
  HSTL Mode ... 183
  Kseg0CA ... 180
  MemEnd ... 181
  ODrainSys ... 182
  PrcElmReq ... 139, 169, 180
  PrcReqMax ... 109, 129, 131, 137, 141, 180
  SCBlkSize ... 67, 77, 108, 181
  SCClkDiv ... 78, 172, 176, 181
  SCClkTap ... 173, 182
  SCCorEn ... 181, 194, 196
  SCSize ... 67, 77, 181
  SysClkDiv ... 96, 172, 176, 181
MP, field ... 236
MTC0, instruction ... 86
multiple matches, in TLB ... 300
multiplier pipeline ... 22
multiply unit, FPU ... 274
multiprocessor system ... 51
  arbitration ... 127
  cluster bus ... 51
  with external agent ... 51
multiprocessor system, using dedicated external agents ... 100
multiprocessor system, using the cluster bus ... 101

N

NACK completion response ... 146
NACK, signal ... 106
NMI *see also* nonmaskable interrupt ... 273
NMI, bit ... 223, 224
nonblocking cache ... 43
nonblocking, loads and stores ... 341
Nonmaskable Interrupt (NMI) exception ... 122, 302, 309
normal read, cache test mode ... 335
normal write, cache test mode ... 333
NT compatibility, LLAddr register ... 231
number, request ... 103

O

ODrainSys, mode bit ... 182
offset, in page address ... 298
operating mode
  Kernel ... 286, 292
  Supervisor ... 286, 290
  User ... 286, 288
operations, FPU ... 275
ordering, memory ... 33
ordering, strong ... 33
out of program order, execution ... 338
outgoing buffer ... 105, 107, 108
outstanding requests ... 103
overflow (FP) ... 283

P

PAddr0, field ... 232
PAddr1, field ... 232
page
  address ... 298
  offset ... 298
  size
    code ... 298
    defined ... 298
  virtual ... 298
page table entry (PTE) array ... 215
PageMask register ... 216, 298, 299
parity protection ... 190
PClk, signal ... 78, 96, 335, 336
PE, bit ... 230
performance
  branch prediction ... 47
  cache ... 47
  R10000 ... 44, 47
Performance Counter interrupt ... 218
Performance Counter register ... 238
permanent register ... 340
PFN
  bits ... 214

- fields, in EntryLo registers ... 214

phase-locked loop ... 174

physical memory addresses ... 298

physical page frame number ... 213

physical register, *see also* logical register ... 343

PIdx, primary cache index ... 84

pipeline ... 35
- definition of ... 338
- fetch ... 22, 35
- floating-point ... 23
- floating-point multiplier ... 22
- integer ALU ... 22
- latency ... 338
- Load/Store Unit ... 22
- out of order execution ... 338
- repeat rate ... 338
- sequence ... 338
- stage (definition) ... 338
- stage 1 ... 35, 36
- stage 2 ... 35
- stages 4-6 ... 36
- stalls ... 31

PLL ... 174

PLLDis, signal ... 59, 60

PM, field ... 230

power interface signals, *see also* individual signals ... 54

power-on reset ... 175
- sequence ... 176

PrcElmReq, mode bit ... 139, 169, 180

PrcReqMax, mode bits ... 109, 129, 131, 137, 141, 180

precise exceptions ... 33

prediction, branch ... 342

prediction, secondary cache, way ... 80

prefetch instruction ... 43

primary data cache, *see also* cache, primary data ... 25

primary instruction cache, *see also* cache, primary instruction ... 25

processor block read request protocol ... 129

processor block write request ... 110
- protocol ... 133

processor coherency data response ... 110
- protocol ... 155

processor coherency state response protocol ... 154

processor double/single/partial-word read request protocol ... 131

processor double/single/partial-word write request protocol ... 135

processor eliminate request protocol ... 139

processor request ... 96, 102
- flow control protocol ... 141
- protocol ... 128

processor response ... 96, 103
- protocols ... 153

Processor Revision Identifier (PRId) register ... 229

processor upgrade request ... 147
- protocol ... 137

program order ... 31
- dynamic execution ... 31
- instruction completion ... 339
- instruction decoding ... 339
- instruction execution ... 339
- instruction fetching ... 339
- instruction graduation ... 339
- instruction issue ... 339

protection
- ECC ... 190
- memory ... 298
- parity ... 190
- SECDED ... 190
- sparse encoding ... 190

protocol
- arbitration, System interface ... 124
- error handling ... 202
- write back ... 61
- write invalidate cache coherency ... 61

PTE (page table entry) ... 215

PTEBase, field ... 215, 233

Q

queue
- address ... 22
- instruction ... 35
- integer ... 22

R

R, (region) field ... 219, 233

R, bit ... 232

R10000 processor
- ANDES architecture ... 20
- caches ... 20
- execution pipelines ... 22
- overview ... 20
- pipeline stages ... 21
- superscalar pipeline ... 21

R4000 superpipeline ... 19

Random entries ... 217

- Random register ... 212
- RE, (reverse endian) bit ... 221
- read port, FPU ... 275
- read sequences ... 85
  - 16-word ... 88
  - 32-word ... 88
  - 4-word ... 86
  - 8-word ... 87
  - tag ... 89
- register
  - BadVAddr ... 215, 218, 233, 310
  - boundary scan, JTAG ... 206
  - bypass, JTAG ... 205
  - CacheErr ... 188, 189, 191, 192, 263
  - Cause ... 121, 122, 218, 226, 228
  - Compare ... 122, 218
  - Config ... 230
  - Context ... 215, 233
  - Count ... 122, 218
  - CP0 (description of) ... 209
  - dependency ... 32, 343
  - Diagnostic ... 235
  - ECC ... 86, 91, 262
  - EntryHi ... 219
  - EntryLo0 ... 213
  - EntryLo1 ... 213
  - EPC ... 228
  - Error Exception Program Counter (ErrorEPC) ... 273
  - Exception Program Counter (EPC) ... 228
  - file
    - FPU ... 275
    - ports ... 343
  - FrameMask ... 214, 234
  - Index ... 211
  - instruction, JTAG ... 205
  - LLAddr ... 231
  - logical, *see also* physical register ... 35, 343
  - PageMask ... 216, 298
  - Performance Counter ... 238
  - permanent ... 340
  - physical, *see also* logical register ... 35, 343
  - Processor Revision Identifier (PRId) ... 229
  - Random ... 212
  - renaming ... 32, 340
  - Status ... 188, 189
    - ERL bit ... 286
    - EXL bit ... 286
    - SX bit ... 296
    - TS bit ... 300
    - USL field ... 286
    - UX bit ... 296
  - TagHi ... 86, 91, 267
  - TagLo ... 86, 91, 267
  - temporary ... 340
  - unnamed ... 341
  - WatchHi ... 232
  - WatchLo ... 232
  - Wired ... 212, 217
  - write before reading (necessity for) ... 176
  - XContext ... 233
- renaming, register ... 340
- repeat rate ... 45
  - accessing secondary cache ... 47
  - definition of ... 338
  - FPU ... 274
- replacement algorithm, cache ... 25
- request cycle ... 96
- request number ... 103
  - freeing with completion response ... 146
- request, outstanding ... 103
- Reserved Instruction exception ... 321
- reset
  - cold ... 175, 178
  - power-on ... 175, 176
  - soft (warm) ... 175, 179
- response bus signals ... 58
- response cycle ... 96
- revision number, R10000 processor ... 229
- RM, field (FP) ... 284
- RN, field (FP) ... 284
- rounding modes, in FSR ... 284
- RP, (reduced power) bit ... 221
- RP, field (FP) ... 284
- rules, arbitration for System interface ... 125
- RZ, field (FP) ... 284

S

- SB, (secondary cache block size) bit ... 230
- SC(A,B)Addr, signals ... 55, 79, 80
- SC(A,B)DWay, signals ... 55, 79, 87, 92
- SC, bit ... 230
- SCADCS, signal ... 55
- SCADOE, signal ... 55
- SCADWr, signal ... 55
- SCBDCS, signal ... 55

SCBDOE, signal ... 55
SCBDWr, signal ... 55
SCBlkSize, mode bits ... 67, 77, 108, 181
SCClk frequency ... 134, 155
SCClk, signal ... 55, 78, 173
SCClkDiv, mode bits ... 78, 172, 176, 181
SCClkTap, mode bits ... 173, 182
SCCorEn, mode bits ... 181, 194, 196
SCData, signal ... 55
SCDataChk, bus ... 193, 196
SCDataChk, signal ... 55
scheduling, dynamic ... 339
SCSize, mode bits ... 67, 77, 181
SCTag, signals ... 56, 83
SCTagChk, bus ... 196
SCTagChk, signal ... 56
SCTagLSBAddr, signal ... 55, 80
SCTCS, signal ... 56
SCTOE, signal ... 56
SCTWay, signal ... 56, 80, 82, 87
SCTWr, signal ... 56
SECDED ... 190
secondary cache interface signals, *see also* individual signals ... 55
secondary cache, *see also* cache, secondary ... 67
SelDVCO, signal ... 59, 60
serial operations ... 41
serializing instruction ... 41
signals
  power interface, *see also* individual signals ... 54
  secondary cache interface, *see also* individual signals ... 55
  System interface, *see also* individual signals ... 57
  test interface, *see also* individual signals ... 59
size, page in memory ... 298
SK, bit ... 230
slave state ... 97
  and flow control ... 109
soft (warm) reset ... 175, 179
Soft Reset
  exception ... 307
Soft Reset exception ... 302
software interrupts ... 122
SP, bit ... 262
sparse encoding protection ... 190
special interrupt vector ... 306
speculative branching ... 342
speculative execution ... 32, 39, 342
square-root unit, FPU ... 274
SR, bit ... 224, 307, 309
SS, (secondary cache size) field ... 230
sseg space ... 291
SSRAM ... 76, 81
  address signals ... 55
  clock signals ... 55
  data signals ... 55
  tag signals ... 56
stage, definition of ... 338
stalls, improving performance ... 31
state
  master ... 97
  slave ... 97
state bus signals ... 58
Status register ... 188
  in FPU, *see also* FSR ... 277
store operations, FPU registers ... 278
stores
  and uncached buffer ... 72
  nonblocking ... 341
strong ordering ... 33
  example of ... 34
superpipeline, architecture ... 19
superpipeline, R4000 ... 19
superscalar
  pipeline ... 19
  processor
    definition of ... 19, 338
superscalar processor ... 31
Supervisor mode ... 286
  address mapping ... 290
  csseg space ... 291
  operations ... 290
  sseg space ... 291
  suseg space ... 290
  xsseg space ... 291
  xsuseg space ... 291
suseg space ... 290
switch, context ... 300
SX, bit ... 222, 286, 296
SYNC
  instruction ... 73, 164
  prevented from graduating ... 108

SysAD, bus signals ... 57, 111, 116, 118, 198, 199, 329, 331, 332, 333, 334, 335, 336

SysAD[20:16]
- interrupt register ... 121

SysAD[39:0]
- during address cycle ... 119

SysAD[56:40]
- during address cycle ... 119

SysAD[57]
- secondary cache block way indication ... 119

SysAD[59:58]
- uncached attribute ... 118

SysAD[63:0]
- address cycle encoding ... 118
- data cycle encoding ... 120

SysAD[63:60]
- address cycle ... 118
- interrupt ... 121

SysADChk, bus ... 199

SysADChk, signal ... 58, 180

SysClk cycle ... 109, 143, 164

SysClk, signal ... 57, 96, 120, 122, 124, 125, 129, 137, 141, 170, 171, 335, 336

SysClkDiv, mode bits ... 172, 176, 181

SysClkRet, signal ... 57, 172, 174

SysCmd, bus ... 57, 111, 187, 198, 199

SysCmd[0] ... 106
- ECC ... 116
- processor data cycles ... 116

SysCmd[10:8] ... 111
- data response ... 115
- external intervention and invalidate requests ... 114

SysCmd[11] ... 111
- map ... 117
- protocol ... 123

SysCmd[2:0]
- processor write requests ... 114

SysCmd[2:1]
- block data response ... 116
- processor requests ... 113

SysCmd[4:3]
- data cycles ... 116
- external special requests ... 115
- processor read requests ... 112
- processor upgrade requests ... 113

SysCmd[5], bit... 106
- data cycles ... 115

SysCmd[7:5]
- external requests ... 114
- processor requests ... 112

SysCmdPar, signal ... 57, 198

SysCorErr, signal ... 58, 185, 194, 196, 199

SysCyc, signal ... 58, 170

SysGblPerf, signal ... 58, 73, 164

SysGnt, signal ... 57, 124, 125, 126, 128, 130, 132, 134, 136, 138, 140, 143, 148, 149, 150, 151, 152, 155, 164, 176, 178, 179, 306, 307, 330, 331

SysNMI, signal ... 58, 122, 309

SysRdRdy, signal ... 57, 125, 129, 131, 137, 141
- and flow control ... 109

SysRel, signal ... 57, 124, 126, 128, 130, 132, 134, 136, 138, 140, 143, 148, 149, 150, 151, 152, 155, 164

SysReq, signal ... 57, 124, 125, 128, 130, 132, 134, 136, 138, 140, 155, 164, 178

SysReset, signal ... 58, 176, 178, 179, 204, 306, 307, 308, 330, 331

SysResp, bus ... 58, 111, 121, 201

SysResp[4:0]
- external completion response ... 146

SysResp[4:2]
- driving completion indication ... 121

SysRespPar, signal ... 58, 201

SysRespVal, signal ... 58, 146, 176, 178, 179, 201

SysState, bus ... 58, 111, 120, 187, 201

SysState[0]
- processor coherency data response ... 162

SysState[2:0]
- encoding ... 120

SysStatePar, signal ... 58, 201

SysStateVal, signal ... 58, 120

System Call exception ... 319

system configuration
- multiprocessor ... 51
- uniprocessor ... 50

System interface
- arbitration
  - in a cluster bus system ... 98, 127
  - in a uniprocessor system ... 126
  - protocol ... 124
  - rules ... 125
- block write request protocol ... 133
- buffers ... 105
- bus encoding
  - description of buses ... 111

    - SysAD ... 118
    - SysCmd ... 111
    - SysResp ... 121
    - SysState ... 120
  - cached request buffer ... 105
  - clock domain ... 172
  - cluster bus ... 98
  - cluster request buffer ... 105
  - coherency ... 157
  - coherency conflicts, action taken ... 159
  - connecting to an external agent ... 97
  - connections to various system configurations ... 99
  - directory-based coherency protocol ... 169
  - error handling
    - on buses ... 198
    - on SysAD bus ... 199
    - on SysCmd bus ... 198
    - on SysResp bus ... 201
    - on SysState bus ... 201
    - schemes ... 197
  - error protection
    - for buses ... 197
    - schemes ... 197
  - external agent ... 95
  - external allocate request number request protocol ... 150
  - external block data response protocol ... 143
  - external coherency requests, action taken ... 158
  - external completion response protocol ... 146
  - external data response flow control ... 109, 110
  - external double/single/partial-word data response protocol ... 145
  - external duplicate tags, support for ... 168
  - external interrupt request protocol ... 152
  - external intervention exclusive request ... 157
  - external intervention request protocol ... 149
  - external intervention shared request ... 157
  - external invalidate request ... 157
    - protocol ... 151
  - external request ... 96, 103
    - flow control ... 109
    - protocol ... 148
  - external response ... 96, 103
    - protocol ... 143
  - flow control ... 109
  - frequencies ... 96
  - grant parking ... 124
  - hardware emulation, support for ... 170
  - I/O ... 168
  - incoming buffer ... 106
  - internal coherency conflicts ... 159
  - interrupts ... 121
  - master state ... 97
  - multiprocessor connections
    - with cluster bus ... 101
    - with dedicated external agents ... 100
  - outgoing buffer ... 107
  - outstanding processor requests ... 103
  - outstanding requests on the System interface ... 103
  - port ... 20
  - processor block read request protocol ... 129
  - processor coherency data response protocol ... 155
  - processor coherency state response protocol ... 154
  - processor double/single/partial-word read request protocol ... 131
  - processor double/single/partial-word write request protocol ... 135
  - processor eliminate request protocol ... 139
  - processor request ... 96, 102
    - flow control protocol ... 141
    - protocol ... 128
  - processor response ... 96, 103
    - protocols ... 153
  - processor upgrade request protocol ... 137
  - register-to-register operation ... 96
  - request ... 102
    - cycle ... 96
    - number field ... 103
    - protocol ... 128
  - response ... 102
    - cycle ... 96
    - protocol ... 128
  - signals ... 57, 97
  - slave state ... 97
  - split transaction ... 103
  - support for I/O ... 168
  - uncached attribute ... 169
  - uncached buffer ... 108
  - uniprocessor connections ... 99
- System interface ... 27, 95
- SysUncErr, signal ... 58, 186, 187, 191, 192, 196
- SysVal, signal ... 58, 129, 131, 133, 135, 137, 139, 143, 145, 149, 150, 151, 152, 155, 198, 329, 333, 334, 335, 336
- SysWrRdy, signal ... 57, 134, 135, 139, 141, 155
  - and flow control ... 109

T

- table
  - busy-bit ... 340
  - mapping ... 343

tag bus, secondary cache, SCTag ... 83

tag read sequence ... 89

tag write sequence ... 94

TagHi register ... 86, 91, 267

TagLo register ... 86, 91, 267

tags, external, duplicate ... 168

TAP controller ... 204, 205

TCA, signal ... 59, 60

TCB, signal ... 59, 60

temporary register ... 340

test access port (TAP) ... 204

test interface signals, *see also* individual signals ... 59

test mode, cache ... 330, 331

test signals, miscellaneous ... 59

Timer interrupt ... 122
  disabling ... 218

TLB ... 299
  32-bit-mode entry format ... 299
  64-bit-mode entry format ... 299
  address
    translation, avoiding multiple matches ... 300
  ASID field ... 300
  avoiding conflict ... 300
  Cache Algorithm fields ... 299
  entry formats ... 299
  exceptions ... 311
  Global (G) bit ... 300
  ITLB ... 300
  misses ... 215
  multiple matches, avoiding ... 300
  number of entries ... 299
  page size code ... 298
  used with Context register ... 215

TLB (Translation Lookaside Buffer) ... 23
  JTLB ... 300

TLB Invalid exception ... 311, 313

TLB Modified exception ... 311, 314

TLB Probe (TLBP) instruction ... 211, 219

TLB Read (TLBR) instruction ... 211

TLB Read Indexed (TLBR) instruction ... 219

TLB Refill ... 303

TLB Refill exception ... 311, 312

TLB Write Indexed (TLBWI) instruction ... 211, 219

TLB Write Random instruction ... 212, 219

Translation Look-Aside Buffer, *see also* TLB ... 299

translation, virtual address ... 298, 300

Trap exception ... 318

trap physical address, and Watch registers ... 232

TriState, signal ... 206

TS, (TLB shutdown) bit ... 223, 224

TS, bit, in Status register ... 300

two-level cache structure ... 61

U

UC, (uncached attribute) bit ... 213

uncached
  accelerated
    blocks, completely gathered ... 72
    blocks, incompletely gathered ... 72
    stores ... 72
  attribute, support for ... 169
  buffer ... 105, 108
  cache algorithm ... 70, 71

uncached accelerated ... 214

uncached accelerated, cache algorithm ... 70, 72

uncached attribute ... 214

uncorrectable error ... 186
  detection, suppressed ... 189
  flag ... 106, 108

underflow (FP) ... 283

unimplemented operation (FP) ... 283

uniprocessor system ... 50, 99
  arbitration rules ... 126

unnaming, register ... 341

useg space ... 288, 289

User mode ... 286
  address mapping ... 288
  operations ... 288
  useg space ... 289
  xuseg space ... 289

UX, bit ... 222, 286, 296

V

V, (valid) bit ... 213

Vcc, signal ... 54

VccPa, signal ... 54

VccPd, signal ... 54

VccQSC, signal ... 54

VccQSys, signal ... 54

vector locations, TLB refill ... 304

vector, special interrupt ... 306

virtual address
    space ... 287
    translation ... 298

virtual aliasing ... 84

Virtual Coherency exception ... 315

virtual memory addresses ... 298

VPN2, field ... 219

VrefByp, signal ... 54

VrefSC, signal ... 54

VrefSys, signal ... 54

Vss, signal ... 54

VssPa, signal ... 54

VssPd, signal ... 54

W

W, bit ... 232

Watch exception ... 324

WatchHi register ... 232

WatchLo register ... 232

way prediction table, secondary cache ... 81

Wired entries ... 217

Wired register ... 212, 217

write back protocol ... 61
    primary data cache ... 64

write sequences ... 90
    16-word ... 93
    32-word ... 93
    4-word ... 91
    8-word ... 92
    tag ... 94

X

XContext register ... 233

xkphys
    decoding virtual address bits VA(61:59) ... 300
    space ... 294

xkseg space ... 296

xksseg space ... 294

xkuseg space ... 294

xsseg space ... 291

xsuseg space ... 291

XTLB Refill ... 303

XTLB refill handler, used with XContext register ... 233

xuseg space ... 288, 289

XX, (MIPS IV User mode) bit ... 220, 222, 286, 326

Phase-out/Discontinued

NEC

Facsimile Message

From:

____
Name

____
Company

____
Tel. FAX

____
Address

Although NEC has taken all possible steps to ensure that the documentation supplied to our customers is complete, bug free and up-to-date, we readily accept that errors may occur. Despite all the care and precautions we've taken, you may encounter problems in the documentation. Please complete this form whenever you'd like to report errors or suggest improvements to us.

Thank you for your kind support.

North America
NEC Electronics Inc.
Corporate Communications Dept.
Fax: +1-800-729-9288
+1-408-588-6130

Europe
NEC Electronics (Europe) GmbH
Technical Documentation Dept.
Fax: +49-211-6503-274

South America
NEC do Brasil S.A.
Fax: +55-11-6462-6829

Hong Kong, Philippines, Oceania
NEC Electronics Hong Kong Ltd.
Fax: +852-2886-9022/9044

Korea
NEC Electronics Hong Kong Ltd.
Seoul Branch
Fax: +82-2-528-4411

Taiwan
NEC Electronics Taiwan Ltd.
Fax: +886-2-2719-5951

Asian Nations except Philippines
NEC Electronics Singapore Pte. Ltd.
Fax: +65-250-3583

Japan
NEC Semiconductor Technical Hotline
Fax: +81- 44-435-9608

I would like to report the following error/make the following suggestion:

Document title: ______________________________

Document number: ______________________________ Page number: ______________

____

____

____

If possible, please fax the referenced page or drawing.

Document Rating	Excellent	Good	Acceptable	Poor
Clarity	☐	☐	☐	☐
Technical Accuracy	☐	☐	☐	☐
Organization	☐	☐	☐	☐

CS 01.2