TX System RISC
TX79 Core Architecture
(Symmetric 2-way superscalar 64-bit CPU) Rev. 2.0
The information contained herein is subject to change without notice.

The information contained herein is presented only as a guide for the applications of our products. No responsibility is assumed by TOSHIBA for any infringements of patents or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of TOSHIBA or others.

TOSHIBA is continually working to improve the quality and reliability of its products. Nevertheless, semiconductor devices in general can malfunction or fail due to their inherent electrical sensitivity and vulnerability to physical stress. It is the responsibility of the buyer, when utilizing TOSHIBA products, to comply with the standards of safety in making a safe design for the entire system, and to avoid situations in which a malfunction or failure of such TOSHIBA products could cause loss of human life, bodily injury or damage to property.

In developing your designs, please ensure that TOSHIBA products are used within specified operating ranges as set forth in the most recent TOSHIBA products specifications.

Also, please keep in mind the precautions and conditions set forth in the “Handling Guide for Semiconductor Devices,” or “TOSHIBA Semiconductor Reliability Handbook” etc.

The Toshiba products listed in this document are intended for usage in general electronics applications (computer, personal equipment, office equipment, measuring equipment, industrial robotics, domestic appliances, etc.). These Toshiba products are neither intended nor warranted for usage in equipment that requires extraordinarily high quality and/or reliability or a malfunction or failure of which may cause loss of human life or bodily injury (“Unintended Usage”). Unintended Usage include atomic energy control instruments, airplane or spaceship instruments, transportation instruments, traffic signal instruments, combustion control instruments, medical instruments, all types of safety devices, etc.. Unintended Usage of Toshiba products listed in this document shall be made at the customer’s own risk.

The products described in this document may include products subject to the foreign exchange and foreign trade laws.
Preface

Thank you for choosing Toshiba semiconductor products. This is the year 2000 edition of the user’s manual for the architecture of the TX79 RISC microprocessor core, a member of the TX System RISC Family of Toshiba microprocessors.

This user’s manual is designed to be easily understood by engineers who are designing a Toshiba microprocessor into their products for the first time. No special knowledge of this architecture is assumed – the contents includes basic information about the architecture of the TX79 microprocessor core as well as more advanced, in-depth description.

Toshiba are continually updating technical publications. Any comments and suggestions regarding any Toshiba document are most welcome and will be taken into account when subsequent editions are prepared. To receive updates to the information in this manual, or for additional information about this architecture, please contact your nearest Toshiba office or authorized Toshiba dealer.

April 2001
CONTENTS

Handling Precautions
C790 User’s Manual

1. Introduction ................................................................................................................. 1-1
   1.1 Features.................................................................................................................... 1-2
   1.2 Related Documents................................................................................................. 1-3
   1.3 Revision History....................................................................................................... 1-4
   1.4 Conventions Used in This Manual ........................................................................... 1-5
   1.5 Restrictions for Use of the C790 CPU Core............................................................. 1-6

2. Architecture Overview.................................................................................................... 2-1
   2.1 Block Diagram and Functional Block Descriptions .................................................. 2-2
      2.1.1 PC Unit ............................................................................................................. 2-3
      2.1.2 MMU ............................................................................................................... 2-3
      2.1.3 Caches............................................................................................................. 2-3
      2.1.4 Issue Logic and Staging Registers....................................................................... 2-3
      2.1.5 GPR (General Purpose Registers) and FPR (Floating-Point Registers)............... 2-3
      2.1.6 The Five Execution Pipes.................................................................................. 2-3
         2.1.6.1 I0 and I1 Pipes ............................................................................................. 2-3
         2.1.6.2 LS - Load/Store Pipe..................................................................................... 2-3
         2.1.6.3 BR - Branch Pipe ......................................................................................... 2-3
         2.1.6.4 C1 - COP1/FPU Pipe.................................................................................. 2-3
      2.1.7 Operand/Bypass logic ....................................................................................... 2-4
      2.1.8 Response Buffer and Writeback Buffer............................................................. 2-4
      2.1.9 UCAB .............................................................................................................. 2-4
      2.1.10 Result and Move Buses .................................................................................. 2-4
      2.1.11 Bus Interface Unit and BIU Bus......................................................................... 2-4
   2.2 Superscalar Pipeline Operation .................................................................................. 2-5
      2.2.1 Integer Instruction Pipeline Stages ..................................................................... 2-5
      2.2.2 C1 (COP1/FPU) Instruction Pipeline Stages ....................................................... 2-8
      2.2.3 Classification and Routing of Instructions According to Execution Pipelines........ 2-10
      2.2.4 Instruction Issue Combinations......................................................................... 2-12
   2.3 Registers................................................................................................................ 2-14
      2.3.1 CPU Registers................................................................................................. 2-14
      2.3.2 FPU Registers ................................................................................................. 2-14
      2.3.3 COP0 Registers............................................................................................... 2-15
2.4 Memory Management ................................................................. 2-16
2.5 Cache Memory ........................................................................... 2-17
2.6 Bus Interface .............................................................................. 2-18
2.7 Floating Point Unit ..................................................................... 2-18
2.8 Performance Counter ................................................................. 2-19
2.9 Debug and Tracing Functions ....................................................... 2-19

3. Instruction Set Overview and Summary ........................................... 3-1
  3.1 Introduction .................................................................................. 3-2
  3.2 CPU Instruction Set Formats ......................................................... 3-3
  3.3 Instruction Set Summary ................................................................. 3-4
    3.3.1 Load/Store Instructions ............................................................. 3-4
      3.3.1.1 Normal Loads and Stores ...................................................... 3-4
      3.3.1.2 Multimedia Loads and Stores .............................................. 3-5
      3.3.1.3 Coprocessor Loads and Stores .............................................. 3-5
      3.3.1.4 Data Formats and Addressing ............................................. 3-5
      3.3.1.5 Defining Access Types ....................................................... 3-9
      3.3.1.6 Scheduling a Load Delay Slot ............................................ 3-13
    3.3.2 Computational Instructions ....................................................... 3-14
      3.3.2.1 ALU Immediate Instructions .............................................. 3-14
      3.3.2.2 Three Operand Register-Type Instructions .......................... 3-15
      3.3.2.3 Shift Instructions ............................................................. 3-15
      3.3.2.4 Multiply and Divide Instructions ........................................ 3-15
      3.3.2.5 64-Bit Operations ............................................................ 3-15
    3.3.3 Jump and Branch Instructions .................................................. 3-16
      3.3.3.1 Jump Instructions ............................................................. 3-16
      3.3.3.2 Branch Instructions .......................................................... 3-17
    3.3.4 Miscellaneous Instructions ...................................................... 3-18
      3.3.4.1 Exception Instructions ...................................................... 3-18
      3.3.4.2 Serialization Instructions .................................................. 3-18
      3.3.4.3 MIPS IV Instructions ....................................................... 3-19
    3.3.5 System Control Coprocessor (COP0) Instructions .................. 3-20
    3.3.6 Coprocessor 1 (COP1) ............................................................. 3-21
      3.3.6.1 Coprocessor 1 (COP1) Instructions .................................. 3-21
    3.3.7 C790-Specific Instructions ....................................................... 3-22
      3.3.7.1 Integer Multiply / Divide Instructions ................................. 3-22
      3.3.7.2 Multimedia Instructions .................................................... 3-23
    3.4 User Instruction Latency and Repeat Rate ................................. 3-25

4. CPU and COP0 Registers ............................................................. 4-1
  4.1 CPU Registers .............................................................................. 4-2
4.1.1 General Purpose Registers ...............................................................4-4
4.1.2 HI and LO Registers ...............................................................4-4
4.1.3 Shift Amount (SA) Register ...............................................................4-4
4.1.4 Program Counter (PC) ...............................................................4-4
4.2 System Control Coprocessor (COP0) Registers ................................4-5
4.2.1 Index Register (0) ...............................................................4-6
4.2.2 Random Register (1) ...............................................................4-7
4.2.3 EntryLo0 Register (2), and EntryLo1 Register (3) ...............................................................4-8
4.2.4 Context Register (4) ...............................................................4-9
4.2.5 PageMask Register (5) ...............................................................4-10
4.2.6 Wired Register (6) ...............................................................4-11
4.2.7 BadVAddr Register (8) ...............................................................4-12
4.2.8 Count Register (9) ...............................................................4-13
4.2.9 EntryHi Register (10) ...............................................................4-14
4.2.10 Compare Register (11) ...............................................................4-15
4.2.11 Status Register (12) ...............................................................4-16
4.2.11.1 Status Register Format ...............................................................4-17
4.2.11.2 Status Register Modes and Access States ...............................................................4-18
4.2.12 Cause Register (13) ...............................................................4-19
4.2.13 EPC Register (14) ...............................................................4-21
4.2.14 PRId Register (15) ...............................................................4-22
4.2.15 Config Register (16) ...............................................................4-23
4.2.16 BadPAddr Register (23) ...............................................................4-25
4.2.17 Debug Registers (24) ...............................................................4-26
4.2.18 Performance Counter Registers (25) ...............................................................4-28
4.2.19 TagLo (28) and TagHi (29) Registers ...............................................................4-31
4.2.20 ErrorEPC (30) ...............................................................4-33

5. Exception Processing and Reset ...............................................................5-1
5.1 The Exception Handling Process ...............................................................5-2
5.1.1 Level 1 Exceptions ...............................................................5-2
5.1.2 Level 2 Exceptions ...............................................................5-5
5.2 Exception Vector Locations ...............................................................5-7
5.3 Cause Register Setting ...............................................................5-8
5.4 Masking an exception ...............................................................5-9
5.5 Detailed Description ...............................................................5-10
5.5.1 Exception Priority ...............................................................5-10
5.5.2 Reset Exception ...............................................................5-11
5.5.3 Non-Maskable Interrupt (NMI) Exception ...............................................................5-12
5.5.4 Performance Counter Exception ...............................................................5-13
5.5.5 Debug Exception ................................................................. 5-14
5.5.6 Address Error Exception ..................................................... 5-15
5.5.7 TLB Refill Exception .......................................................... 5-16
5.5.8 TLB Invalid Exception ........................................................ 5-17
5.5.9 TLB Modified Exception .................................................... 5-18
5.5.10 Bus Error Exception ......................................................... 5-19
5.5.11 System Call Exception ....................................................... 5-20
5.5.12 BREAK Instruction Exception ............................................ 5-21
5.5.13 Reserved Instruction Exception ......................................... 5-22
5.5.14 Coprocessor Unusable Exception ...................................... 5-23
5.5.15 Interrupt Exception .......................................................... 5-24
5.5.16 SIO Exception ................................................................. 5-25
5.5.17 Integer Overflow Exception .............................................. 5-26
5.5.18 Trap Exception ............................................................... 5-27
5.5.19 Floating-Point Exception .................................................. 5-28

6. Memory Management ............................................................. 6-1
6.1 Translation Look-aside Buffer (TLB) ........................................ 6-2
  6.1.1 Translation Status .............................................................. 6-2
  6.1.2 Multiple Matches ............................................................. 6-2
6.2 Address Spaces ................................................................. 6-3
  6.2.1 Virtual Address Space ....................................................... 6-3
  6.2.2 Physical Address Space .................................................... 6-4
  6.2.3 Virtual-to-Physical Address Translation .......................... 6-4
  6.2.4 32-bit Address Translation Mode ....................................... 6-5
  6.2.5 Operating Modes ............................................................. 6-6
  6.2.6 User Mode Operations .................................................... 6-8
  6.2.7 Supervisor Mode Operations ............................................ 6-10
  6.2.8 Kernel Mode Operations ................................................ 6-11
6.3 System Control Coprocessor ................................................ 6-14
  6.3.1 Format of a TLB Entry ...................................................... 6-15
6.4 Virtual-to-Physical Address Translation Process .................. 6-18
6.5 TLB Instructions ............................................................... 6-20

7. Caches 7-1
7.1 Cache Features ................................................................. 7-2
7.2 Organization of the Caches ................................................... 7-3
  7.2.1 Data Cache ................................................................. 7-3
  7.2.2 Instruction Cache .......................................................... 7-4
  7.2.3 Tag Structure ............................................................... 7-5
7.2.3.1 Data Cache Tag Structure ................................................................. 7-6
7.2.3.2 Instruction Cache Tag Structure ...................................................... 7-6
7.2.4 State of Cache Tags After Reset ......................................................... 7-7
7.3 Cache Operations .................................................................................... 7-8
  7.3.1 Line Replacement Algorithm ............................................................. 7-8
  7.3.2 Non-blocking Loads and Hit Under Miss ........................................... 7-8
  7.3.3 Cache Miss and Hit Operations ......................................................... 7-9
  7.3.4 Data Cache Writeback Policy ............................................................. 7-10
  7.3.5 Data Cache State Transitions ............................................................. 7-11
  7.3.6 Instruction Cache State Transitions .................................................. 7-12
  7.3.7 Data Cache Lock Function ............................................................... 7-12
    7.3.7.1 Operations During Lock ............................................................... 7-13
  7.3.8 Relationship Between Cached and Uncached Operations ............... 7-13
7.4 Uncached Accelerated Buffer ............................................................... 7-14
  7.4.1 UCAB Configuration ........................................................................ 7-14
  7.4.2 Tag Structure .................................................................................... 7-14
  7.4.3 Non-blocking Loads and Hit under Miss .......................................... 7-14
7.5 Cache Control Registers ........................................................................ 7-15
7.6 CACHE Instruction .................................................................................. 7-16
8. CPU Bus .................................................................................................... 8-1
  8.1 Introduction .......................................................................................... 8-2
    8.1.1 Terminology ................................................................................... 8-3
    8.1.2 Signal Naming Convention ............................................................ 8-3
  8.2 CPU Bus Architecture ........................................................................... 8-4
    8.2.1 CPU Bus Connectivity for Address and Control Paths ................. 8-5
    8.2.2 CPU Bus Connectivity for Data Paths ........................................... 8-6
  8.3 CPU Bus Signal Descriptions ............................................................... 8-7
    8.3.1 Address Bus Signals ...................................................................... 8-7
  8.4 Overview of CPU Bus Operations ....................................................... 8-12
    8.4.1 CPU Bus Operations ...................................................................... 8-12
    8.4.2 Processor Requests ........................................................................ 8-12
      8.4.2.1 Read Requests .......................................................................... 8-12
      8.4.2.2 Write Requests ........................................................................ 8-13
    8.4.3 Bus Error Operations ..................................................................... 8-13
  8.5 CPU Bus Transaction Protocols and Timing ........................................ 8-14
    8.5.1 Arbitration Operations .................................................................... 8-14
      8.5.1.1 Cycle Stealing .......................................................................... 8-15
    8.5.2 CPU Single Operations ................................................................. 8-16
      8.5.2.1 CPU Single Reads .................................................................... 8-16
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.5.2.2</td>
<td>CPU Single Writes</td>
<td>8-17</td>
</tr>
<tr>
<td>8.5.2.3</td>
<td>CPU Single Read-Write-Read-Write Cycles</td>
<td>8-18</td>
</tr>
<tr>
<td>8.5.3</td>
<td>CPU Burst Operations</td>
<td>8-19</td>
</tr>
<tr>
<td>8.5.3.1</td>
<td>CPU Burst Reads</td>
<td>8-19</td>
</tr>
<tr>
<td>8.5.3.2</td>
<td>CPU Burst Writes</td>
<td>8-20</td>
</tr>
<tr>
<td>8.5.3.3</td>
<td>CPU Burst Read-Write Cycles</td>
<td>8-21</td>
</tr>
<tr>
<td>8.5.3.4</td>
<td>CPU Burst Write-Read Cycles</td>
<td>8-21</td>
</tr>
<tr>
<td>8.5.4</td>
<td>CPU Non-Pipeline Single Operations</td>
<td>8-22</td>
</tr>
<tr>
<td>8.5.4.1</td>
<td>CPU Non-Pipeline Single Reads</td>
<td>8-22</td>
</tr>
<tr>
<td>8.5.4.2</td>
<td>CPU Non-Pipeline Single Writes</td>
<td>8-23</td>
</tr>
<tr>
<td>8.5.5</td>
<td>CPU Non-Pipeline Burst Operations</td>
<td>8-23</td>
</tr>
<tr>
<td>8.5.5.1</td>
<td>CPU Non-Pipeline Burst Reads</td>
<td>8-23</td>
</tr>
<tr>
<td>8.5.5.2</td>
<td>CPU Non-Pipeline Burst Writes</td>
<td>8-24</td>
</tr>
<tr>
<td>8.5.6</td>
<td>Bus Error Operations</td>
<td>8-25</td>
</tr>
<tr>
<td>8.5.6.1</td>
<td>Bus Error Exceptions</td>
<td>8-25</td>
</tr>
<tr>
<td>8.5.6.2</td>
<td>CPU Bus Cycle Termination</td>
<td>8-26</td>
</tr>
<tr>
<td>8.5.6.3</td>
<td>Bus Error Timing with No Pending Operation</td>
<td>8-26</td>
</tr>
<tr>
<td>8.5.6.4</td>
<td>Bus Error Timing with One Pending Operation</td>
<td>8-26</td>
</tr>
<tr>
<td>8.5.6.5</td>
<td>Bus Error Timing with Two Pending Operations</td>
<td>8-28</td>
</tr>
<tr>
<td>9.</td>
<td>Performance Counter</td>
<td>9-1</td>
</tr>
<tr>
<td>9.1</td>
<td>Overview</td>
<td>9-2</td>
</tr>
<tr>
<td>9.2</td>
<td>Performance Counters and Performance Control Registers</td>
<td>9-2</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Accessing Counters and Registers</td>
<td>9-2</td>
</tr>
<tr>
<td>9.2.2</td>
<td>State of Performance Counter Control Registers Upon Reset</td>
<td>9-4</td>
</tr>
<tr>
<td>9.3</td>
<td>Counter Operation</td>
<td>9-5</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Counter Events</td>
<td>9-6</td>
</tr>
<tr>
<td>9.3.1.1</td>
<td>Event Descriptions</td>
<td>9-7</td>
</tr>
<tr>
<td>9.3.2</td>
<td>Handling Performance Counter Exceptions</td>
<td>9-10</td>
</tr>
<tr>
<td>9.3.3</td>
<td>Priority of Counter Exceptions</td>
<td>9-11</td>
</tr>
<tr>
<td>9.3.4</td>
<td>Initializing Counters</td>
<td>9-11</td>
</tr>
<tr>
<td>9.3.5</td>
<td>The Note to Read Counters</td>
<td>9-12</td>
</tr>
<tr>
<td>10.</td>
<td>Floating-Point Unit, CP1 (Option)</td>
<td>10-1</td>
</tr>
<tr>
<td>10.1</td>
<td>Overview</td>
<td>10-2</td>
</tr>
<tr>
<td>10.2</td>
<td>Floating Point Register</td>
<td>10-2</td>
</tr>
<tr>
<td>10.2.1</td>
<td>Floating-Point General Registers (FGRs)</td>
<td>10-2</td>
</tr>
<tr>
<td>10.2.2</td>
<td>Floating-Point Registers (FPRs)</td>
<td>10-4</td>
</tr>
<tr>
<td>10.2.3</td>
<td>Floating-Point Control Registers</td>
<td>10-4</td>
</tr>
<tr>
<td>10.2.4</td>
<td>Accessing the FP Control and Implementation/Revision Registers</td>
<td>10-9</td>
</tr>
<tr>
<td>10.3</td>
<td>Floating-Point Formats</td>
<td>10-10</td>
</tr>
</tbody>
</table>
10.4 Binary Fixed-Point Format .......................................................... 10-12
10.5 Floating-Point Instruction Set Summary ...................................... 10-13
  10.5.1 Load, Store and Move Instructions (Table 10-10) .................. 10-13
  10.5.2 Conversion Instructions (Table 10-11) ............................... 10-14
  10.5.3 Computational Instructions (Table 10-12) ......................... 10-14
  10.5.4 Compare and Branch Instructions (Table 10-13) ............... 10-15

11. Floating-Point Exception (Option) .................................................. 11-1
  11.1 Introduction ........................................................................... 11-2
  11.2 Exception Types ..................................................................... 11-2
  11.3 Exception Trap Processing ...................................................... 11-3
  11.4 Flags .................................................................................... 11-3
  11.5 FPU Exceptions ..................................................................... 11-5
  11.6 Saving and Restoring State .................................................... 11-9
  11.7 Trap Handlers for IEEE Standard 754 Exceptions ................... 11-9

12. PC Trace ....................................................................................... 12-1
  12.1 Real-Time PC Tracing ............................................................... 12-2
    12.1.1 Classification of Branch and Jump Instructions ............... 12-2
    12.1.2 PC Trace Signals .............................................................. 12-3
    12.1.3 Priority of Target Addresses .......................................... 12-7
    12.1.4 Examples of PC Tracing .................................................. 12-8
      12.1.4.1 Sequential Execution ............................................... 12-9
      12.1.4.2 Conditional Branch ............................................... 12-10
      12.1.4.3 Indirect Jump (Target in Phase A) ......................... 12-11
      12.1.4.4 Indirect Jump (Target in Phase B) ......................... 12-12
      12.1.4.5 Indirect Jump (During Target PC Output) ............... 12-13
      12.1.4.6 Exception (Target in Phase B) ............................. 12-14
      12.1.4.7 Exception (During Target PC Output) ................. 12-15
      12.1.4.8 Exception Generated by Branch or Jump Instruction .. 12-16
      12.1.4.9 Exception Generated by Branch Delay Slot Instruction .. 12-17
      12.1.4.10 Exception Generated by Target Instruction ........... 12-18
      12.1.4.11 Back to Back Exceptions (Case I) ....................... 12-19
      12.1.4.12 Back to Back Exceptions (Case II) ...................... 12-20

13. Hardware Breakpoint .................................................................... 13-1
  13.1 Hardware Breakpoint .............................................................. 13-2
    13.1.1 Hardware Breakpoint signal .......................................... 13-2
    13.2 Breakpoint Registers .......................................................... 13-3
      13.2.1 Breakpoint Control Register (BPC) ............................ 13-4
      13.2.2 Instruction Address Breakpoint Register (IAB) / Instruction Address Breakpoint Mask ...
B.2 Definitions for Pseudocode Functions Used in Operation Descriptions ........................................... B-2
B.3 Summary of C790-Specific Instructions .................................................................................................. B-3
  B.3.1 Multiply and Multiply-Add Instructions .......................................................................................... B-3
  B.3.2 Multimedia Instructions .................................................................................................................. B-3
B.4 Instruction Set Details .................................................................................................................. B-6
B.5 C790-Specific Instruction Encoding .................................................................................................. B-163

C. COP0 System Control Coprocessor Instruction Set Details............................................................... C-1
  C.1.1 Notes on the CACHE Instruction Sub-operations ......................................................................... C-7
    Cache Virtual Address .............................................................................................................................. C-7
    Cache Physical Address .......................................................................................................................... C-7
    BTAC Virtual Address ........................................................................................................................... C-7
    BTAC Index Bits ...................................................................................................................................... C-7
    COP0 Not Usable .................................................................................................................................... C-7
    TLB Exceptions on Cache Operations .................................................................................................. C-8
    Hit Sub-operation Accesses ................................................................................................................ C-8
    Breakpoint Exception .......................................................................................................................... C-8
    Address Error Exception ..................................................................................................................... C-8
  C.1.2 Sub-Operation Descriptions ........................................................................................................ C-9
  C.1.3 Updates of Data Tag Status Bits .................................................................................................. C-13
C.2 COP0 Instruction Encoding ................................................................................................................ C-41

D. COP1 (FPU) Instruction Set Details .................................................................................................. D-1
  D.1 Conventions Used in This Chapter .................................................................................................. D-2
    D.1.1 Instruction Description Notation and Functions ........................................................................ D-2
    D.1.2 Pseudocode Language Statement Execution ........................................................................... D-2
    D.1.3 Pseudocode Symbols ................................................................................................................ D-2
  D.2 Definitions for Pseudocode Functions Used in Operation Descriptions ........................................ D-2
  D.3 Instruction Descriptions ................................................................................................................ D-3
  D.4 COP1 Instruction Encoding ............................................................................................................. D-40
FIGURES

Figure 2-1. C790 Block Diagram .....................................................................................................2-2
Figure 2-2. C790 Integer Instruction Pipeline ..............................................................................2-5
Figure 2-3. FPU Pipeline ..............................................................................................................2-8
Figure 2-4. Instruction Routing in Logical Pipes and Physical Pipes ............................................2-10
Figure 3-1. CPU Instruction Formats .............................................................................................3-3
Figure 3-2. Big-Endian Byte Ordering ..........................................................................................3-6
Figure 3-3. Little-Endian Byte Ordering .......................................................................................3-6
Figure 3-4. Little-Endian Data in a Doubleword ..........................................................................3-7
Figure 3-5. Big-Endian Data in a Doubleword ..............................................................................3-7
Figure 3-6. Big-Endian Misaligned Word Addressing .................................................................3-8
Figure 3-7. Little-Endian Misaligned Word Addressing ...............................................................3-8
Figure 4-1. CPU Registers ............................................................................................................4-3
Figure 4-2. Index Register ..........................................................................................................4-6
Figure 4-3. Random Register ......................................................................................................4-7
Figure 4-4. EntryLo0 and EntryLo1 Registers .............................................................................4-8
Figure 4-5. Context Register Format ..........................................................................................4-9
Figure 4-6. PageMask Register ..................................................................................................4-10
Figure 4-7. Wired Register ........................................................................................................4-11
Figure 4-8. Wired Register Boundary .........................................................................................4-11
Figure 4-9. BadVAddr Register .................................................................................................4-12
Figure 4-10. Count Register .......................................................................................................4-13
Figure 4-11. EntryHi Register ......................................................................................................4-14
Figure 4-12. Compare Register ................................................................................................4-15
Figure 4-13. Status Register ......................................................................................................4-16
Figure 4-14. Cause Register ......................................................................................................4-19
Figure 4-15. EPC Register ..........................................................................................................4-21
Figure 4-16. PRId Register ..........................................................................................................4-22
Figure 4-17. Config Register Format ..........................................................................................4-23
Figure 4-18. BadPAddr Register Format ....................................................................................4-25
Figure 4-19. Performance Counter Registers ..............................................................................4-28
Figure 4-20. TagLo and TagHi Registers .....................................................................................4-31
Figure 4-21. ErrorEPC Register ..................................................................................................4-33
Figure 5-1. Level 1 Exception processing flowchart ....................................................................5-4
Figure 5-2. Level 2 Exception processing flowchart ....................................................................5-6
Figure 6-1. Overview of a Virtual-to-Physical Address Translation .............................................6-3
Figure 6-2. 32-bit Mode Virtual Address Translation ...................................................................6-5
Figure 6-3. State Transition among Operating Modes .................................................................6-6
Figure 6-4. User Mode Virtual Address Space ..........................................................................6-8
Figure 6-5. Supervisor Mode Virtual Address Space ...............................................................6-10
Figure 6-6. Kernel Mode Address Space ....................................................................................6-11
Figure 6-7. COP0 Registers and the TLB ....................................................................................6-14
Figure 6-8. Format of a TLB Entry ..............................................................................................6-15
Figure 6-9. TLB Address Translation ..........................................................................................6-19
Figure 7-1. Organization of Data Cache .....................................................................................7-3
Figure 7-2. Organization of Instruction Cache .............................................................................7-4
Figure 7-3. Read Missed Processed in Sequential Order ............................................................7-10
Figure 7-4. Data Cache Transition Diagram, Writeback Protocol .............................................7-11
Figure 7-5. Instruction Cache Transition Diagram ......................................................................7-12
Figure 8-1. CPU Bus Architecture .............................................................................................8-4
Figure 8-2. CPU Bus Address and Control Path Connections in System ................................7-5
Figure 8-3. CPU Bus Data Path Connections in System .............................................................8-6
Figure 8-4. Connection of Arbitration Signals .........................................................................8-14
Figure 8-5. Arbitration Protocol .................................................................................................8-15
Figure 8-6. Cycle Stealing Protocol ...........................................................................................8-15
Figure 8-7. CPU Single Reads .....................................................................................................8-16
Figure 8-8. CPU Single Writes .....................................................................................................8-17
Figure 8-9. CPU Single Read-Write-Read-Write Cycles ...............................................................8-18
Figure 8-10. CPU Burst Reads ....................................................................................................8-19
Figure 8-11. CPU Burst Writes ....................................................................................................8-20
Figure 8-12. CPU Burst Read-Write Cycles ...............................................................................8-21
Figure 8-13. CPU Burst Write-Read Cycles ...............................................................................8-21
Figure 8-14. CPU Non-Pipeline Single Reads .........................................................................8-22
Figure 8-15. CPU Non-Pipeline Single Writes ..........................................................................8-23
Figure 8-16. CPU Non-Pipeline Burst Reads ..............................................................................8-23
Figure 8-17. CPU Non-Pipeline Burst Writes ..............................................................................8-24
Figure 8-18. One Operation with BUSERR* as the Last SYSDACK* .........................................8-27
Figure 8-19. One Operation with BUSERR* as SYSAACK* ....................................................8-27
Figure 8-20. One Operation with BUSERR* as SYSAACK* and the Last SYSDACK* ..........8-28
Figure 8-21. Two Operations with Bus Error as the Last SYSDACK* .....................................8-29
Figure 9-1. Format of the Performance Counter Control Register PCCR ....................................9-2
Figure 9-2. Format of Performance Counter Registers PCR0 and PCR1 ..................................9-2
Figure 9-3. CAUSE Register Fields ...........................................................................................9-10
Figure 10-1. FP Registers ..........................................................................................................10-3
Figure 10-2. Implementation/Revision Register ........................................................................10-5
Figure 10-3. FP Control/Status Register Bit Assignments .........................................................10-6
Figure 10-4. Control/Status Register Cause, Flag, and Enable Fields ....................................10-7
TABLES

Table 1-1. Restriction List ................................................................. 1-6
Table 2-1. Categories of Instructions and How They Are Routed ........................................ 2-11
Table 2-2. Concurrently Issued Instruction Categories ......................................................... 2-13
Table 2-3. Coprocessor 0 Registers .................................................................................. 2-15
Table 3-1. Load / Store Instructions ................................................................................... 3-4
Table 3-2. Multimedia Load / Store Instructions .................................................................. 3-5
Table 3-3. Coprocessor Load / Store Instructions ................................................................. 3-5
Table 3-4. Defining Access Types (Big-Endian) .................................................................... 3-10
Table 3-5. Defining Access Types (Little-Endian) ................................................................. 3-12
Table 3-6. ALU Immediate Instructions .............................................................................. 3-14
Table 3-7. Three Operand Register-Type Instructions ......................................................... 3-15
Table 3-8. Shift Instructions ............................................................................................... 3-15
Table 3-9. Multiply and Divide Instructions ......................................................................... 3-15
Table 3-10. Jump Instructions Jumping Within a 256 MByte Region .................................... 3-16
Table 3-11. Jump Instructions to Absolute Address ............................................................ 3-16
Table 3-12. PC-Relative Conditional Branch Instructions Comparing 2 Registers .............. 3-17
Table 3-13. PC-Relative Conditional Branch Instructions Comparing Against Zero ............. 3-17
Table 3-14. Exception Instructions ...................................................................................... 3-18
Table 3-15. Serialization Instructions .................................................................................. 3-18
Table 3-16. MIPS IV Instructions ........................................................................................ 3-19
Table 3-17. System Control Coprocessor Instructions ......................................................... 3-20
Table 3-18. Coprocessor 1 Instructions ............................................................................... 3-21
Table 3-19. C790-Specific Multiply and Divide Instructions .............................................. 3-22
Table 3-20. Multimedia Instructions .................................................................................... 3-23
Table 3-21. Latencies and Repeat Rates for User Instruction ............................................... 3-25
Table 4-1. Coprocessor 0 Registers .................................................................................... 4-5
Table 4-2. Index Register Field Description ......................................................................... 4-6
Table 4-3. Random Register Fields ..................................................................................... 4-7
Table 4-4. EntryLo0 and EntryLo1 Register Fields ................................................................ 4-8
Table 4-5. Context Register Fields ....................................................................................... 4-9
Table 4-6. PageMask Register Field .................................................................................... 4-10
Table 4-7. Wired Register Field Descriptions ...................................................................... 4-11
Table 4-8. BadVAddr Register Field .................................................................................... 4-12
Table 4-9. Count Register Field .......................................................................................... 4-13
Table 4-10. EntryHi Register Fields ..................................................................................... 4-14
Table 4-11. Compare Register Field .................................................................................... 4-15
Table 4-12. Status Register Fields.................................................................................................4-17
Table 4-13. Cause Register Fields.................................................................................................4-19
Table 4-14. EPC Register Field ......................................................................................................4-21
Table 4-15. PRId Register Fields ..................................................................................................4-22
Table 4-16. Config Register Fields .................................................................................................4-23
Table 4-17. BadPAddr Register Fields ...........................................................................................4-25
Table 4-18. Performance Counter Control Register Fields ...........................................................4-29
Table 4-19. Performance Counter Register 0 Fields .....................................................................4-30
Table 4-20. Performance Counter Register 1 Fields .....................................................................4-30
Table 4-21. TagLo Register Fields .................................................................................................4-32
Table 4-22. TagHi Register Fields .................................................................................................4-32
Table 4-23. ErrorEPC Register Field .............................................................................................4-33
Table 5-1. Exception Levels.............................................................................................................5-2
Table 5-2. Exception Vectors for Level 1 exceptions.......................................................................5-7
Table 5-3. Exception Vectors for Level 2 exceptions.......................................................................5-7
Table 5-4. Cause.ExcCode Field .....................................................................................................5-8
Table 5-5. Cause.EXC2 Field ..........................................................................................................5-8
Table 5-6. Masking exceptions .......................................................................................................5-9
Table 5-7. Exception Priority Order ...............................................................................................5-10
Table 6-1 Processor Modes .............................................................................................................6-6
Table 6-2. Address Space................................................................................................................6-7
Table 6-3. User Mode Segments .....................................................................................................6-9
Table 6-4. Supervisor Mode Segments ...........................................................................................6-10
Table 6-5. Kernel Mode Segments ................................................................................................6-12
Table 6-6 TLB Page Coherency (C) Bit Values .............................................................................6-17
Table 6-7. TLB Instructions ............................................................................................................6-20
Table 7-1. Cache Configuration .......................................................................................................7-2
Table 7-2. Cache Size and Access Bits ...........................................................................................7-5
Table 7-3. Data Cache Line States ..................................................................................................7-6
Table 7-4. LRF Line Replacement Algorithm ................................................................................7-8
Table 7-5. Quadword Retrieved Address PA[5:4]..........................................................................7-10
Table 7-6. UCAB Configuration ....................................................................................................7-14
Table 7-7. UCAB Size and Access Bits ..........................................................................................7-14
Table 8-1. System Signal Naming Convention .............................................................................8-3
Table 8-2. Bus Transaction Types ..................................................................................................8-8
Table 8-3. CPU Transfer Size ........................................................................................................8-9
Table 8-4. Bus Error Exceptions .....................................................................................................8-25
Table 8-5. Operation Termination Sequence ................................................................................8-26
Table 9-1. PCCR Register Bits ......................................................................................................8-26
Table 9-2. Writing Performance Counters and Registers using MTC0 ...........................................9-3
Table 9-3. Reading Performance Counters and Registers using MFC0 ........................................ 9-3
Table 9-4. Mnemonics to Access the Performance Counters and Registers ................................... 9-3
Table 9-5. Counter Events ............................................................................................................... 9-6
Table 9-6. Definition of Data Cache Miss ....................................................................................... 9-7
Table 10-1. Floating-Point Control Register Assignments ............................................................. 10-4
Table 10-2. FCR0 Fields ................................................................................................................ 10-5
Table 10-3. Control/Status Register Fields .................................................................................... 10-6
Table 10-4. Flush Values of Denormalized Results ....................................................................... 10-7
Table 10-5. Rounding Mode Bit Decoding ..................................................................................... 10-9
Table 10-6. Equations for Calculating Values in Single and Double-Precision Floating-Point Format ........................................................................................................... 10-11
Table 10-7. Floating-Point Format Parameter Values ....................................................................... 10-11
Table 10-8. Minimum and Maximum Floating-Point Values ............................................................. 10-11
Table 10-9. Binary Fixed-Point Format Fields ............................................................................... 10-12
Table 10-10. FPU Instruction Set (Optional): Load, Move and Store Instruction ........................ 10-13
Table 10-11. FPU Instruction Set(Optional): Conversion Instruction ........................................ 10-14
Table 10-12. FPU Instruction Set(Optional): Computational Instruction ..................................... 10-14
Table 10-13. FPU Instruction Set(Optional): Compare and Branch Instruction .......................... 10-15
Table 11-1. Default FPU Exception Actions .................................................................................... 11-3
Table 11-2. FPU Exception-Causing Conditions ........................................................................ 11-4
Table 11-3. Values of Overflow Results ....................................................................................... 11-7
Table 12-1. Classification of Branch and Jump Instruction .............................................................. 12-2
Table 12-2. Exception Vector Address Codes .................................................................................. 12-6
Table 13-1. Set a new value into breakpoint registers ................................................................ 13-4
Table 13-2. Get the value from breakpoint registers ...................................................................... 13-4
Table 13-3. BPC Register Fields .................................................................................................... 13-5
Table A-1. Symbols in Instruction Operation Statements ............................................................... A-3
Table A-2. Coprocessor General Register Access Functions .......................................................... A-5
Table A-3. Load and Store Functions ............................................................................................... A-6
Table A-4. AccessLength Specifications for Loads / Stores ............................................................ A-7
Table A-5. Miscellaneous Functions ............................................................................................... A-8
Table B-1. Quotient and Remainder Signs ..................................................................................... B-8
Table C-1. CACHE Instruction Op Field Encoding ........................................................................ C-6
Table C-2. Data Tag Status Bit Modifications ................................................................................ C-13
Table D-1. FPU Comparisons Without Special Operand Exceptions ....................................... D-9
Table D-2 FPU Comparisons With Special Operand Exceptions for QNaNs ............................... D-10
Handling Precautions
1. **Using Toshiba Semiconductors Safely**

   TOSHIBA is continually working to improve the quality and the reliability of its products.

   Nevertheless, semiconductor devices in general can malfunction or fail due to their inherent electrical sensitivity and vulnerability to physical stress. It is the responsibility of the buyer, when utilizing TOSHIBA products, to observe standards of safety, and to avoid situations in which a malfunction or failure of a TOSHIBA product could cause loss of human life, bodily injury or damage to property.

   In developing your designs, please ensure that TOSHIBA products are used within specified operating ranges as set forth in the most recent products specifications. Also, please keep in mind the precautions and conditions set forth in the TOSHIBA Semiconductor Reliability Handbook.
2. Safety Precautions

This section lists important precautions which users of semiconductor devices (and anyone else) should observe in order to avoid injury and damage to property, and to ensure safe and correct use of devices.

Please be sure that you understand the meanings of the labels and the graphic symbol described below before you move on to the detailed descriptions of the precautions.

[Explanation of labels]

<table>
<thead>
<tr>
<th>Graphic symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="DANGER" /></td>
<td>Indicates an imminently hazardous situation which will result in death or serious injury if you do not follow instructions.</td>
</tr>
<tr>
<td><img src="image" alt="WARNING" /></td>
<td>Indicates a potentially hazardous situation which could result in death or serious injury if you do not follow instructions.</td>
</tr>
<tr>
<td><img src="image" alt="CAUTION" /></td>
<td>Indicates a potentially hazardous situation which if not avoided, may result in minor injury or moderate injury.</td>
</tr>
</tbody>
</table>

[Explanation of graphic symbol]

<table>
<thead>
<tr>
<th>Graphic symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="triangle" /></td>
<td>Indicates that caution is required (laser beam is dangerous to eyes).</td>
</tr>
</tbody>
</table>
### 2.1 General Precautions regarding Semiconductor Devices

<table>
<thead>
<tr>
<th>CAUTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do not use devices under conditions exceeding their absolute maximum ratings (e.g. current, voltage, power dissipation or temperature). This may cause the device to break down, degrade its performance, or cause it to catch fire or explode resulting in injury.</td>
</tr>
<tr>
<td>Do not insert devices in the wrong orientation. Make sure that the positive and negative terminals of power supplies are connected correctly. Otherwise the rated maximum current or power dissipation may be exceeded and the device may break down or undergo performance degradation, causing it to catch fire or explode and resulting in injury.</td>
</tr>
<tr>
<td>When power to a device is on, do not touch the device’s heat sink. Heat sinks become hot, so you may burn your hand.</td>
</tr>
<tr>
<td>Do not touch the tips of device leads. Because some types of device have leads with pointed tips, you may prick your finger.</td>
</tr>
<tr>
<td>When conducting any kind of evaluation, inspection or testing, be sure to connect the testing equipment’s electrodes or probes to the pins of the device under test before powering it on. Otherwise, you may receive an electric shock causing injury.</td>
</tr>
<tr>
<td>Before grounding an item of measuring equipment or a soldering iron, check that there is no electrical leakage from it. Electrical leakage may cause the device which you are testing or soldering to break down, or could give you an electric shock.</td>
</tr>
<tr>
<td>Always wear protective glasses when cutting the leads of a device with clippers or a similar tool. If you do not, small bits of metal flying off the cut ends may damage your eyes.</td>
</tr>
</tbody>
</table>
2.2 Precautions Specific to Each Product Group

2.2.1 Optical semiconductor devices

<table>
<thead>
<tr>
<th>DANGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>When a visible semiconductor laser is operating, do not look directly into the laser beam or look through the optical system. This is highly likely to impair vision, and in the worst case may cause blindness. If it is necessary to examine the laser apparatus, for example to inspect its optical characteristics, always wear the appropriate type of laser protective glasses as stipulated by IEC standard IEC825-1.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WARNING</th>
</tr>
</thead>
</table>
| Ensure that the current flowing in an LED device does not exceed the device’s maximum rated current. This is particularly important for resin-packaged LED devices, as excessive current may cause the package resin to blow up, scattering resin fragments and causing injury.  

When testing the dielectric strength of a photocoupler, use testing equipment which can shut off the supply voltage to the photocoupler. If you detect a leakage current of more than 100 µA, use the testing equipment to shut off the photocoupler’s supply voltage; otherwise a large short-circuit current will flow continuously, and the device may break down or burst into flames, resulting in fire or injury.  

When incorporating a visible semiconductor laser into a design, use the device’s internal photodetector or a separate photodetector to stabilize the laser’s radiant power so as to ensure that laser beams exceeding the laser’s rated radiant power cannot be emitted. If this stabilizing mechanism does not work and the rated radiant power is exceeded, the device may break down or the excessively powerful laser beams may cause injury. |

2.2.2 Power devices

<table>
<thead>
<tr>
<th>DANGER</th>
</tr>
</thead>
</table>
| Never touch a power device while it is powered on. Also, after turning off a power device, do not touch it until it has thoroughly discharged all remaining electrical charge. Touching a power device while it is powered on or still charged could cause a severe electric shock, resulting in death or serious injury.  

When conducting any kind of evaluation, inspection or testing, be sure to connect the testing equipment’s electrodes or probes to the device under test before powering it on. When you have finished, discharge any electrical charge remaining in the device. Connecting the electrodes or probes of testing equipment to a device while it is powered on may result in electric shock, causing injury. |
Do not use devices under conditions which exceed their absolute maximum ratings (current, voltage, power dissipation, temperature etc.). This may cause the device to break down, causing a large short-circuit current to flow, which may in turn cause it to catch fire or explode, resulting in fire or injury.

Use a unit which can detect short-circuit currents and which will shut off the power supply if a short-circuit occurs. If the power supply is not shut off, a large short-circuit current will flow continuously, which may in turn cause the device to catch fire or explode, resulting in fire or injury.

When designing a case for enclosing your system, consider how best to protect the user from shrapnel in the event of the device catching fire or exploding. Flying shrapnel can cause injury.

When conducting any kind of evaluation, inspection or testing, always use protective safety tools such as a cover for the device. Otherwise you may sustain injury caused by the device catching fire or exploding.

Make sure that all metal casings in your design are grounded to earth. Even in modules where a device’s electrodes and metal casing are insulated, capacitance in the module may cause the electrostatic potential in the casing to rise. Dielectric breakdown may cause a high voltage to be applied to the casing, causing electric shock and injury to anyone touching it.

When designing the heat radiation and safety features of a system incorporating high-speed rectifiers, remember to take the device’s forward and reverse losses into account. The leakage current in these devices is greater than that in ordinary rectifiers; as a result, if a high-speed rectifier is used in an extreme environment (e.g. at high temperature or high voltage), its reverse loss may increase, causing thermal runaway to occur. This may in turn cause the device to explode and scatter shrapnel, resulting in injury to the user.

A design should ensure that, except when the main circuit of the device is active, reverse bias is applied to the device gate while electricity is conducted to control circuits, so that the main circuit will become inactive. Malfunction of the device may cause serious accidents or injuries.

When conducting any kind of evaluation, inspection or testing, either wear protective gloves or wait until the device has cooled properly before handling it. Devices become hot when they are operated. Even after the power has been turned off, the device will retain residual heat which may cause a burn to anyone touching it.

2.2.3 Bipolar ICs (for use in automobiles)

If your design includes an inductive load such as a motor coil, incorporate diodes or similar devices into the design to prevent negative current from flowing in. The load current generated by powering the device on and off may cause it to function erratically or to break down, which could in turn cause injury.

Ensure that the power supply to any device which incorporates protective functions is stable. If the power supply is unstable, the device may operate erratically, preventing the protective functions from working correctly. If protective functions fail, the device may break down causing injury to the user.
3. General Safety Precautions and Usage Considerations

This section is designed to help you gain a better understanding of semiconductor devices, so as to ensure the safety, quality and reliability of the devices which you incorporate into your designs.

3.1 From Incoming to Shipping

3.1.1 Electrostatic discharge (ESD)

When handling individual devices (which are not yet mounted on a printed circuit board), be sure that the environment is protected against electrostatic electricity. Operators should wear anti-static clothing, and containers and other objects which come into direct contact with devices should be made of anti-static materials and should be grounded to earth via an 0.5- to 1.0-MΩ protective resistor.

Please follow the precautions described below; this is particularly important for devices which are marked “Be careful of static.”:

(1) Work environment

- When humidity in the working environment decreases, the human body and other insulators can easily become charged with static electricity due to friction. Maintain the recommended humidity of 40% to 60% in the work environment, while also taking into account the fact that moisture-proof-packed products may absorb moisture after unpacking.

- Be sure that all equipment, jigs and tools in the working area are grounded to earth.

- Place a conductive mat over the floor of the work area, or take other appropriate measures, so that the floor surface is protected against static electricity and is grounded to earth. The surface resistivity should be $10^4$ to $10^8$ Ω/sq and the resistance between surface and ground, $7.5 \times 10^5$ to $10^8$ Ω.

- Cover the workbench surface also with a conductive mat (with a surface resistivity of $10^4$ to $10^8$ Ω/sq, for a resistance between surface and ground of $7.5 \times 10^5$ to $10^8$ Ω). The purpose of this is to disperse static electricity on the surface (through resistive components) and ground it to earth. Workbench surfaces must not be constructed of low-resistance metallic materials that allow rapid static discharge when a charged device touches them directly.

- Pay attention to the following points when using automatic equipment in your workplace:

  (a) When picking up ICs with a vacuum unit, use a conductive rubber fitting on the end of the pick-up wand to protect against electrostatic charge.

  (b) Minimize friction on IC package surfaces. If some rubbing is unavoidable due to the device’s mechanical structure, minimize the friction plane or use material with a small friction coefficient and low electrical resistance. Also, consider the use of an ionizer.

  (c) In sections which come into contact with device lead terminals, use a material which dissipates static electricity.

  (d) Ensure that no statically charged bodies (such as work clothes or the human body) touch the devices.
(e) Make sure that sections of the tape carrier which come into contact with installation devices or other electrical machinery are made of a low-resistance material.

(f) Make sure that jigs and tools used in the assembly process do not touch devices.

(g) In processes in which packages may retain an electrostatic charge, use an ionizer to neutralize the ions.

- Make sure that CRT displays in the working area are protected against static charge, for example by a VDT filter. As much as possible, avoid turning displays on and off. Doing so can cause electrostatic induction in devices.

- Keep track of charged potential in the working area by taking periodic measurements.

- Ensure that work chairs are protected by an anti-static textile cover and are grounded to the floor surface by a grounding chain. (Suggested resistance between the seat surface and grounding chain is $7.5 \times 10^5$ to $10^{12} \Omega$.)

- Install anti-static mats on storage shelf surfaces. (Suggested surface resistivity is $10^4$ to $10^8 \Omega$ sq; suggested resistance between surface and ground is $7.5 \times 10^5$ to $10^8 \Omega$.)

- For transport and temporary storage of devices, use containers (boxes, jigs or bags) that are made of anti-static materials or materials which dissipate electrostatic charge.

- Make sure that cart surfaces which come into contact with device packaging are made of materials which will conduct static electricity, and verify that they are grounded to the floor surface via a grounding chain.

- In any location where the level of static electricity is to be closely controlled, the ground resistance level should be Class 3 or above. Use different ground wires for all items of equipment which may come into physical contact with devices.

(2) Operating environment

- Operators must wear anti-static clothing and conductive shoes (or a leg or heel strap).

- Operators must wear a wrist strap grounded to earth via a resistor of about 1 M\(\Omega\).

- Soldering irons must be grounded from iron tip to earth, and must be used only at low voltages (6 V to 24 V).

- If the tweezers you use are likely to touch the device terminals, use anti-static tweezers and in particular avoid metallic tweezers. If a charged device touches a low-resistance tool, rapid discharge can occur. When using vacuum tweezers, attach a conductive chucking pat to the tip, and connect it to a dedicated ground used especially for anti-static purposes (suggested resistance value: $10^4$ to $10^8 \Omega$).

- Do not place devices or their containers near sources of strong electrical fields (such as above a CRT).
• When storing printed circuit boards which have devices mounted on them, use a board container or bag that is protected against static charge. To avoid the occurrence of static charge or discharge due to friction, keep the boards separate from one another and do not stack them directly on top of one another.

• Ensure, if possible, that any articles (such as clipboards) which are brought to any location where the level of static electricity must be closely controlled are constructed of anti-static materials.

• In cases where the human body comes into direct contact with a device, be sure to wear anti-static finger covers or gloves (suggested resistance value: \(10^8\) \(\Omega\) or less).

• Equipment safety covers installed near devices should have resistance ratings of \(10^9\) \(\Omega\) or less.

• If a wrist strap cannot be used for some reason, and there is a possibility of imparting friction to devices, use an ionizer.

• The transport film used in TCP products is manufactured from materials in which static charges tend to build up. When using these products, install an ionizer to prevent the film from being charged with static electricity. Also, ensure that no static electricity will be applied to the product’s copper foils by taking measures to prevent static occurring in the peripheral equipment.

3.1.2 Vibration, impact and stress

Handle devices and packaging materials with care. To avoid damage to devices, do not toss or drop packages. Ensure that devices are not subjected to mechanical vibration or shock during transportation. Ceramic package devices and devices in canister-type packages which have empty space inside them are subject to damage from vibration and shock because the bonding wires are secured only at their ends.

Plastic molded devices, on the other hand, have a relatively high level of resistance to vibration and mechanical shock because their bonding wires are enveloped and fixed in resin. However, when any device or package type is installed in target equipment, it is to some extent susceptible to wiring disconnections and other damage from vibration, shock and stressed solder junctions. Therefore when devices are incorporated into the design of equipment which will be subject to vibration, the structural design of the equipment must be thought out carefully.

If a device is subjected to especially strong vibration, mechanical shock or stress, the package or the chip itself may crack. In products such as CCDs which incorporate window glass, this could cause surface flaws in the glass or cause the connection between the glass and the ceramic to separate.

Furthermore, it is known that stress applied to a semiconductor device through the package changes the resistance characteristics of the chip because of piezoelectric effects. In analog circuit design attention must be paid to the problem of package stress as well as to the dangers of vibration and shock as described above.
3.2 Storage

3.2.1 General storage

- Avoid storage locations where devices will be exposed to moisture or direct sunlight.

- Follow the instructions printed on the device cartons regarding transportation and storage.

- The storage area temperature should be kept within a temperature range of 5°C to 35°C, and relative humidity should be maintained at between 45% and 75%.

- Do not store devices in the presence of harmful (especially corrosive) gases, or in dusty conditions.

- Use storage areas where there is minimal temperature fluctuation. Rapid temperature changes can cause moisture to form on stored devices, resulting in lead oxidation or corrosion. As a result, the solderability of the leads will be degraded.

- When repacking devices, use anti-static containers.

- Do not allow external forces or loads to be applied to devices while they are in storage.

- If devices have been stored for more than two years, their electrical characteristics should be tested and their leads should be tested for ease of soldering before they are used.

3.2.2 Moisture-proof packing

Moisture-proof packing should be handled with care. The handling procedure specified for each packing type should be followed scrupulously. If the proper procedures are not followed, the quality and reliability of devices may be degraded. This section describes general precautions for handling moisture-proof packing. Since the details may differ from device to device, refer also to the relevant individual datasheets or databook.

(1) General precautions

Follow the instructions printed on the device cartons regarding transportation and storage.

- Do not drop or toss device packing. The laminated aluminum material in it can be rendered ineffective by rough handling.

- The storage area temperature should be kept within a temperature range of 5°C to 30°C, and relative humidity should be maintained at 90% (max). Use devices within 12 months of the date marked on the package seal.
• If the 12-month storage period has expired, or if the 30% humidity indicator shown in Figure 1 is pink when the packing is opened, it may be advisable, depending on the device and packing type, to back the devices at high temperature to remove any moisture. Please refer to the table below. After the pack has been opened, use the devices in a 5°C to 30°C, 60% RH environment and within the effective usage period listed on the moisture-proof package. If the effective usage period has expired, or if the packing has been stored in a high-humidity environment, back the devices at high temperature.

<table>
<thead>
<tr>
<th>Packing</th>
<th>Moisture removal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tray</td>
<td>If the packing bears the “Heatproof” marking or indicates the maximum temperature which it can withstand, bake at 125°C for 20 hours. (Some devices require a different procedure.)</td>
</tr>
<tr>
<td>Tube</td>
<td>Transfer devices to trays bearing the “Heatproof” marking or indicating the temperature which they can withstand, or to aluminum tubes before baking at 125°C for 20 hours.</td>
</tr>
<tr>
<td>Tape</td>
<td>Device packed on tape cannot be baked and must be used within the effective usage period after unpacking, as specified on the packing.</td>
</tr>
</tbody>
</table>

• When baking devices, protect the devices from static electricity.

• Moisture indicators can detect the approximate humidity level at a standard temperature of 25°C. 6-point indicators and 3-point indicators are currently in use, but eventually all indicators will be 3-point indicators.

![Humidity Indicator Diagram](image)

**Figure 1** Humidity indicator
3.3 Design

Care must be exercised in the design of electronic equipment to achieve the desired reliability. It is important not only to adhere to specifications concerning absolute maximum ratings and recommended operating conditions, it is also important to consider the overall environment in which equipment will be used, including factors such as the ambient temperature, transient noise and voltage and current surges, as well as mounting conditions which affect device reliability. This section describes some general precautions which you should observe when designing circuits and when mounting devices on printed circuit boards.

For more detailed information about each product family, refer to the relevant individual technical datasheets available from Toshiba.

3.3.1 Absolute maximum ratings

**CAUTION**

Do not use devices under conditions in which their absolute maximum ratings (e.g. current, voltage, power dissipation or temperature) will be exceeded. A device may break down or its performance may be degraded, causing it to catch fire or explode resulting in injury to the user.

The absolute maximum ratings are rated values which must not be exceeded during operation, even for an instant. Although absolute maximum ratings differ from product to product, they essentially concern the voltage and current at each pin, the allowable power dissipation, and the junction and storage temperatures.

If the voltage or current on any pin exceeds the absolute maximum rating, the device’s internal circuitry can become degraded. In the worst case, heat generated in internal circuitry can fuse wiring or cause the semiconductor chip to break down.

If storage or operating temperatures exceed rated values, the package seal can deteriorate or the wires can become disconnected due to the differences between the thermal expansion coefficients of the materials from which the device is constructed.

3.3.2 Recommended operating conditions

The recommended operating conditions for each device are those necessary to guarantee that the device will operate as specified in the datasheet. If greater reliability is required, derate the device’s absolute maximum ratings for voltage, current, power and temperature before using it.

3.3.3 Derating

When incorporating a device into your design, reduce its rated absolute maximum voltage, current, power dissipation and operating temperature in order to ensure high reliability. Since derating differs from application to application, refer to the technical datasheets available for the various devices used in your design.

3.3.4 Unused pins

If unused pins are left open, some devices can exhibit input instability problems, resulting in malfunctions such as abrupt increase in current flow. Similarly, if the unused output pins on a device are connected to the power supply pin, the ground pin or to other output pins, the IC may malfunction or break down.
Since the details regarding the handling of unused pins differ from device to device and from pin to pin, please follow the instructions given in the relevant individual datasheets or databook.

CMOS logic IC inputs, for example, have extremely high impedance. If an input pin is left open, it can easily pick up extraneous noise and become unstable. In this case, if the input voltage level reaches an intermediate level, it is possible that both the P-channel and N-channel transistors will be turned on, allowing unwanted supply current to flow. Therefore, ensure that the unused input pins of a device are connected to the power supply (Vcc) pin or ground (GND) pin of the same device. For details of what to do with the pins of heat sinks, refer to the relevant technical datasheet and databook.

3.3.5 Latch-up

Latch-up is an abnormal condition inherent in CMOS devices, in which Vcc gets shorted to ground. This happens when a parasitic PN-PN junction (thyristor structure) internal to the CMOS chip is turned on, causing a large current of the order of several hundred mA or more to flow between Vcc and GND, eventually causing the device to break down.

Latch-up occurs when the input or output voltage exceeds the rated value, causing a large current to flow in the internal chip, or when the voltage on the Vcc (Vdd) pin exceeds its rated value, forcing the internal chip into a breakdown condition. Once the chip falls into the latch-up state, even though the excess voltage may have been applied only for an instant, the large current continues to flow between Vcc (Vdd) and GND (Vss). This causes the device to heat up and, in extreme cases, to emit gas fumes as well. To avoid this problem, observe the following precautions:

1. Do not allow voltage levels on the input and output pins either to rise above Vcc (Vdd) or to fall below GND (Vss). Also, follow any prescribed power-on sequence, so that power is applied gradually or in steps rather than abruptly.

2. Do not allow any abnormal noise signals to be applied to the device.

3. Set the voltage levels of unused input pins to Vcc (Vdd) or GND (Vss).

4. Do not connect output pins to one another.

3.3.6 Input/Output protection

Wired-AND configurations, in which outputs are connected together, cannot be used, since this short-circuits the outputs. Outputs should, of course, never be connected to Vcc (Vdd) or GND (Vss).

Furthermore, ICs with tri-state outputs can undergo performance degradation if a shorted output current is allowed to flow for an extended period of time. Therefore, when designing circuits, make sure that tri-state outputs will not be enabled simultaneously.

3.3.7 Load capacitance

Some devices display increased delay times if the load capacitance is large. Also, large charging and discharging currents will flow in the device, causing noise. Furthermore, since outputs are shorted for a relatively long time, wiring can become fused.

Consult the technical information for the device being used to determine the recommended load capacitance.
3.3.8 Thermal design

The failure rate of semiconductor devices is greatly increased as operating temperatures increase. As shown in Figure 2, the internal thermal stress on a device is the sum of the ambient temperature and the temperature rise due to power dissipation in the device. Therefore, to achieve optimum reliability, observe the following precautions concerning thermal design:

1. Keep the ambient temperature \( T_a \) as low as possible.

2. If the device's dynamic power dissipation is relatively large, select the most appropriate circuit board material, and consider the use of heat sinks or forced air cooling. Such measures will help lower the thermal resistance of the package.

3. Derate the device's absolute maximum ratings to minimize thermal stress from power dissipation.

\[
\begin{align*}
\theta_{ja} &= \theta_{jc} + \theta_{ca} \\
\theta_{ja} &= \frac{T_j - T_a}{P} \\
\theta_{jc} &= \frac{T_j - T_c}{P} \\
\theta_{ca} &= \frac{T_c - T_a}{P}
\end{align*}
\]

in which \( \theta_{ja} \) = thermal resistance between junction and surrounding air \((^\circ C/W)\)

\( \theta_{jc} \) = thermal resistance between junction and package surface, or internal thermal resistance \((^\circ C/W)\)

\( \theta_{ca} \) = thermal resistance between package surface and surrounding air, or external thermal resistance \((^\circ C/W)\)

\( T_j \) = junction temperature or chip temperature \(^\circ C\)

\( T_c \) = package surface temperature or case temperature \(^\circ C\)

\( T_a \) = ambient temperature \(^\circ C\)

\( P \) = power dissipation (W)

Figure 2  Thermal resistance of package

3.3.9 Interfacing

When connecting inputs and outputs between devices, make sure input voltage \((V_{IL}/V_{IH})\) and output voltage \((V_{OL}/V_{OH})\) levels are matched. Otherwise, the devices may malfunction. When connecting devices operating at different supply voltages, such as in a dual-power-supply system, be aware that erroneous power-on and power-off sequences can result in device breakdown. For details of how to interface particular devices, consult the relevant technical datasheets and databooks. If you have any questions or doubts about interfacing, contact your nearest Toshiba office or distributor.
3.3.10 Decoupling

Spike currents generated during switching can cause Vcc (Vdd) and GND (Vss) voltage levels to fluctuate, causing ringing in the output waveform or a delay in response speed. (The power supply and GND wiring impedance is normally 50 Ω to 100 Ω.) For this reason, the impedance of power supply lines with respect to high frequencies must be kept low. This can be accomplished by using thick and short wiring for the Vcc (Vdd) and GND (Vss) lines and by installing decoupling capacitors (of approximately 0.01 µF to 1 µF capacitance) as high-frequency filters between Vcc (Vdd) and GND (Vss) at strategic locations on the printed circuit board.

For low-frequency filtering, it is a good idea to install a 10- to 100-µF capacitor on the printed circuit board (one capacitor will suffice). If the capacitance is excessively large, however, (e.g., several thousand µF) latch-up can be a problem. Be sure to choose an appropriate capacitance value.

An important point about wiring is that, in the case of high-speed logic ICs, noise is caused mainly by reflection and crosstalk, or by the power supply impedance. Reflections cause increased signal delay, ringing, overshoot and undershoot, thereby reducing the device's safety margins with respect to noise. To prevent reflections, reduce the wiring length by increasing the device mounting density so as to lower the inductance (L) and capacitance (C) in the wiring. Extreme care must be taken, however, when taking this corrective measure, since it tends to cause crosstalk between the wires. In practice, there must be a trade-off between these two factors.

3.3.11 External noise

Printed circuit boards with long I/O or signal pattern lines are vulnerable to induced noise or surges from outside sources. Consequently, malfunctions or breakdowns can result from overcurrent or overvoltage, depending on the types of device used. To protect against noise, lower the impedance of the pattern line or insert a noise-canceling circuit. Protective measures must also be taken against surges.

For details of the appropriate protective measures for a particular device, consult the relevant databook.

3.3.12 Electromagnetic interference

Widespread use of electrical and electronic equipment in recent years has brought with it radio and TV reception problems due to electromagnetic interference. To use the radio spectrum effectively and to maintain radio communications quality, each country has formulated regulations limiting the amount of electromagnetic interference which can be generated by individual products.

Electromagnetic interference includes conduction noise propagated through power supply and telephone lines, and noise from direct electromagnetic waves radiated by equipment. Different measurement methods and corrective measures are used to assess and counteract each specific type of noise.

Difficulties in controlling electromagnetic interference derive from the fact that there is no method available which allows designers to calculate, at the design stage, the strength of the electromagnetic waves which will emanate from each component in a piece of equipment. For this reason, it is only after the prototype equipment has been completed that the designer can take measurements using a dedicated instrument to determine the strength of electromagnetic interference waves. Yet it is possible during system design to incorporate some measures for the prevention of electromagnetic interference, which can facilitate taking corrective measures once the design has been completed. These include installing shields and noise filters, and increasing
the thickness of the power supply wiring patterns on the printed circuit board. One effective method, for example, is to devise several shielding options during design, and then select the most suitable shielding method based on the results of measurements taken after the prototype has been completed.

3.3.13 Peripheral circuits

In most cases semiconductor devices are used with peripheral circuits and components. The input and output signal voltages and currents in these circuits must be chosen to match the semiconductor device's specifications. The following factors must be taken into account.

(1) Inappropriate voltages or currents applied to a device's input pins may cause it to operate erratically. Some devices contain pull-up or pull-down resistors. When designing your system, remember to take the effect of this on the voltage and current levels into account.

(2) The output pins on a device have a predetermined external circuit drive capability. If this drive capability is greater than that required, either incorporate a compensating circuit into your design or carefully select suitable components for use in external circuits.

3.3.14 Safety standards

Each country has safety standards which must be observed. These safety standards include requirements for quality assurance systems and design of device insulation. Such requirements must be fully taken into account to ensure that your design conforms to the applicable safety standards.

3.3.15 Other precautions

(1) When designing a system, be sure to incorporate fail-safe and other appropriate measures according to the intended purpose of your system. Also, be sure to debug your system under actual board-mounted conditions.

(2) If a plastic-package device is placed in a strong electric field, surface leakage may occur due to the charge-up phenomenon, resulting in device malfunction. In such cases take appropriate measures to prevent this problem, for example by protecting the package surface with a conductive shield.

(3) With some microcomputers and MOS memory devices, caution is required when powering on or resetting the device. To ensure that your design does not violate device specifications, consult the relevant databook for each constituent device.

(4) Ensure that no conductive material or object (such as a metal pin) can drop onto and short the leads of a device mounted on a printed circuit board.

3.4 Inspection, Testing and Evaluation

3.4.1 Grounding

⚠️ CAUTION

Ground all measuring instruments, jigs, tools and soldering irons to earth. Electrical leakage may cause a device to break down or may result in electric shock.
3.4.2 Inspection Sequence

\textbf{CAUTION}

1. Do not insert devices in the wrong orientation. Make sure that the positive and negative electrodes of the power supply are correctly connected. Otherwise, the rated maximum current or maximum power dissipation may be exceeded and the device may break down or undergo performance degradation, causing it to catch fire or explode, resulting in injury to the user.

2. When conducting any kind of evaluation, inspection or testing using AC power with a peak voltage of 42.4 V or DC power exceeding 60 V, be sure to connect the electrodes or probes of the testing equipment to the device under test before powering it on. Connecting the electrodes or probes of testing equipment to a device while it is powered on may result in electric shock, causing injury.

(1) Apply voltage to the test jig only after inserting the device securely into it. When applying or removing power, observe the relevant precautions, if any.

(2) Make sure that the voltage applied to the device is off before removing the device from the test jig. Otherwise, the device may undergo performance degradation or be destroyed.

(3) Make sure that no surge voltages from the measuring equipment are applied to the device.

(4) The chips housed in tape carrier packages (TCPs) are bare chips and are therefore exposed. During inspection take care not to crack the chip or cause any flaws in it. Electrical contact may also cause a chip to become faulty. Therefore make sure that nothing comes into electrical contact with the chip.

3.5 Mounting

There are essentially two main types of semiconductor device package: lead insertion and surface mount. During mounting on printed circuit boards, devices can become contaminated by flux or damaged by thermal stress from the soldering process. With surface-mount devices in particular, the most significant problem is thermal stress from solder reflow, when the entire package is subjected to heat. This section describes a recommended temperature profile for each mounting method, as well as general precautions which you should take when mounting devices on printed circuit boards. Note, however, that even for devices with the same package type, the appropriate mounting method varies according to the size of the chip and the size and shape of the lead frame. Therefore, please consult the relevant technical datasheet and databook.

3.5.1 Lead forming

\textbf{CAUTION}

1. Always wear protective glasses when cutting the leads of a device with clippers or a similar tool. If you do not, small bits of metal flying off the cut ends may damage your eyes.

2. Do not touch the tips of device leads. Because some types of device have leads with pointed tips, you may prick your finger.

Semiconductor devices must undergo a process in which the leads are cut and formed before the devices can be mounted on a printed circuit board. If undue stress is applied to the interior of a device during this process, mechanical breakdown or performance degradation can result. This is attributable primarily to differences between the stress on the device’s external leads and the stress on the internal leads. If the relative difference is great enough, the device’s internal leads, adhesive properties or sealant can be damaged. Observe these precautions during the lead-forming process (this does not apply to surface-mount devices):
3. General Safety Precautions and Usage Considerations

(1) Lead insertion hole intervals on the printed circuit board should match the lead pitch of the device precisely.

(2) If lead insertion hole intervals on the printed circuit board do not precisely match the lead pitch of the device, do not attempt to forcibly insert devices by pressing on them or by pulling on their leads.

(3) For the minimum clearance specification between a device and a printed circuit board, refer to the relevant device's datasheet and databook. If necessary, achieve the required clearance by forming the device's leads appropriately. Do not use the spacers which are used to raise devices above the surface of the printed circuit board during soldering to achieve clearance. These spacers normally continue to expand due to heat, even after the solder has begun to solidify; this applies severe stress to the device.

(4) Observe the following precautions when forming the leads of a device prior to mounting.

- Use a tool or jig to secure the lead at its base (where the lead meets the device package) while bending so as to avoid mechanical stress to the device. Also avoid bending or stretching device leads repeatedly.

- Be careful not to damage the lead during lead forming.

- Follow any other precautions described in the individual datasheets and databooks for each device and package type.

3.5.2 Socket mounting

(1) When socket mounting devices on a printed circuit board, use sockets which match the inserted device's package.

(2) Use sockets whose contacts have the appropriate contact pressure. If the contact pressure is insufficient, the socket may not make a perfect contact when the device is repeatedly inserted and removed; if the pressure is excessively high, the device leads may be bent or damaged when they are inserted into or removed from the socket.

(3) When soldering sockets to the printed circuit board, use sockets whose construction prevents flux from penetrating into the contacts or which allows flux to be completely cleaned off.

(4) Make sure the coating agent applied to the printed circuit board for moisture-proofing purposes does not stick to the socket contacts.

(5) If the device leads are severely bent by a socket as it is inserted or removed and you wish to repair the leads so as to continue using the device, make sure that this lead correction is only performed once. Do not use devices whose leads have been corrected more than once.

(6) If the printed circuit board with the devices mounted on it will be subjected to vibration from external sources, use sockets which have a strong contact pressure so as to prevent the sockets and devices from vibrating relative to one another.

3.5.3 Soldering temperature profile

The soldering temperature and heating time vary from device to device. Therefore, when specifying the mounting conditions, refer to the individual datasheets and databooks for the devices used.
(1) Using a soldering iron

Complete soldering within ten seconds for lead temperatures of up to 260°C, or within three seconds for lead temperatures of up to 350°C.

(2) Using medium infrared ray reflow

- Heating top and bottom with long or medium infrared rays is recommended (see Figure 3).

![Figure 3 Heating top and bottom with long or medium infrared rays](image)

- Complete the infrared ray reflow process within 30 seconds at a package surface temperature of between 210°C and 240°C.
- Refer to Figure 4 for an example of a good temperature profile for infrared or hot air reflow.

![Figure 4 Sample temperature profile for infrared or hot air reflow](image)

(3) Using hot air reflow

- Complete hot air reflow within 30 seconds at a package surface temperature of between 210°C and 240°C.
- For an example of a recommended temperature profile, refer to Figure 4 above.

(4) Using solder flow

- Apply preheating for 60 to 120 seconds at a temperature of 150°C.
- For lead insertion-type packages, complete solder flow within 10 seconds with the temperature at the stopper (or, if there is no stopper, at a location more than 1.5 mm from the body) which does not exceed 260°C.
• For surface-mount packages, complete soldering within 5 seconds at a temperature of 250°C or less in order to prevent thermal stress in the device.

• Figure 5 shows an example of a recommended temperature profile for surface-mount packages using solder flow.

![Figure 5](sample temperature profile for solder flow)

### 3.5.4 Flux cleaning and ultrasonic cleaning

1. When cleaning circuit boards to remove flux, make sure that no residual reactive ions such as Na or Cl remain. Note that organic solvents react with water to generate hydrogen chloride and other corrosive gases which can degrade device performance.

2. Washing devices with water will not cause any problems. However, make sure that no reactive ions such as sodium and chlorine are left as a residue. Also, be sure to dry devices sufficiently after washing.

3. Do not rub device markings with a brush or with your hand during cleaning or while the devices are still wet from the cleaning agent. Doing so can rub off the markings.

4. The dip cleaning, shower cleaning and steam cleaning processes all involve the chemical action of a solvent. Use only recommended solvents for these cleaning methods. When immersing devices in a solvent or steam bath, make sure that the temperature of the liquid is 50°C or below, and that the circuit board is removed from the bath within one minute.

5. Ultrasonic cleaning should not be used with hermetically-sealed ceramic packages such as a leadless chip carrier (LCC), pin grid array (PGA) or charge-coupled device (CCD), because the bonding wires can become disconnected due to resonance during the cleaning process. Even if a device package allows ultrasonic cleaning, limit the duration of ultrasonic cleaning to as short a time as possible, since long hours of ultrasonic cleaning degrade the adhesion between the mold resin and the frame material. The following ultrasonic cleaning conditions are recommended:

   - Frequency: 27 kHz ~ 29 kHz
   - Ultrasonic output power: 300 W or less (0.25 W/cm² or less)
   - Cleaning time: 30 seconds or less

Suspend the circuit board in the solvent bath during ultrasonic cleaning in such a way that the ultrasonic vibrator does not come into direct contact with the circuit board or the device.
3.5.5 No cleaning

If analog devices or high-speed devices are used without being cleaned, flux residues may cause minute amounts of leakage between pins. Similarly, dew condensation, which occurs in environments containing residual chlorine when power to the device is on, may cause between-lead leakage or migration. Therefore, Toshiba recommends that these devices be cleaned. However, if the flux used contains only a small amount of halogen (0.05W% or less), the devices may be used without cleaning without any problems.

3.5.6 Mounting tape carrier packages (TCPs)

1. When tape carrier packages (TCPs) are mounted, measures must be taken to prevent electrostatic breakdown of the devices.

2. If devices are being picked up from tape, or outer lead bonding (OLB) mounting is being carried out, consult the manufacturer of the insertion machine which is being used, in order to establish the optimum mounting conditions in advance and to avoid any possible hazards.

3. The base film, which is made of polyimide, is hard and thin. Be careful not to cut or scratch your hands or any objects while handling the tape.

4. When punching tape, try not to scatter broken pieces of tape too much.

5. Treat the extra film, reels and spacers left after punching as industrial waste, taking care not to destroy or pollute the environment.

6. Chips housed in tape carrier packages (TCPs) are bare chips and therefore have their reverse side exposed. To ensure that the chip will not be cracked during mounting, ensure that no mechanical shock is applied to the reverse side of the chip. Electrical contact may also cause a chip to fail. Therefore, when mounting devices, make sure that nothing comes into electrical contact with the reverse side of the chip.
   If your design requires connecting the reverse side of the chip to the circuit board, please consult Toshiba or a Toshiba distributor beforehand.

3.5.7 Mounting chips

Devices delivered in chip form tend to degrade or break under external forces much more easily than plastic-packaged devices. Therefore, caution is required when handling this type of device.

1. Mount devices in a properly prepared environment so that chip surfaces will not be exposed to polluted ambient air or other polluted substances.

2. When handling chips, be careful not to expose them to static electricity. In particular, measures must be taken to prevent static damage during the mounting of chips. With this in mind, Toshiba recommend mounting all peripheral parts first and then mounting chips last (after all other components have been mounted).

3. Make sure that PCBs (or any other kind of circuit board) on which chips are being mounted do not have any chemical residues on them (such as the chemicals which were used for etching the PCBs).

4. When mounting chips on a board, use the method of assembly that is most suitable for maintaining the appropriate electrical, thermal and mechanical properties of the semiconductor devices used.

* For details of devices in chip form, refer to the relevant device’s individual datasheets.
3.5.8 Circuit board coating

When devices are to be used in equipment requiring a high degree of reliability or in extreme environments (where moisture, corrosive gas or dust is present), circuit boards may be coated for protection. However, before doing so, you must carefully consider the possible stress and contamination effects that may result and then choose the coating resin which results in the minimum level of stress to the device.

3.5.9 Heat sinks

(1) When attaching a heat sink to a device, be careful not to apply excessive force to the device in the process.

(2) When attaching a device to a heat sink by fixing it at two or more locations, evenly tighten all the screws in stages (i.e. do not fully tighten one screw while the rest are still only loosely tightened). Finally, fully tighten all the screws up to the specified torque.

(3) Drill holes for screws in the heat sink exactly as specified. Smooth the surface by removing burrs and protrusions or indentations which might interfere with the installation of any part of the device.

(4) A coating of silicone compound can be applied between the heat sink and the device to improve heat conductivity. Be sure to apply the coating thinly and evenly; do not use too much. Also, be sure to use a non-volatile compound, as volatile compounds can crack after a time, causing the heat radiation properties of the heat sink to deteriorate.

(5) If the device is housed in a plastic package, use caution when selecting the type of silicone compound to be applied between the heat sink and the device. With some types, the base oil separates and penetrates the plastic package, significantly reducing the useful life of the device. Two recommended silicone compounds in which base oil separation is not a problem are YG6260 from Toshiba Silicone.

(6) Heat-sink-equipped devices can become very hot during operation. Do not touch them, or you may sustain a burn.

3.5.10 Tightening torque

(1) Make sure the screws are tightened with fastening torques not exceeding the torque values stipulated in individual datasheets and databooks for the devices used.

(2) Do not allow a power screwdriver (electrical or air-driven) to touch devices.

3.5.11 Repeated device mounting and usage

Do not remount or re-use devices which fall into the categories listed below; these devices may cause significant problems relating to performance and reliability.

(1) Devices which have been removed from the board after soldering

(2) Devices which have been inserted in the wrong orientation or which have had reverse current applied

(3) Devices which have undergone lead forming more than once
3.6 Protecting Devices in the Field

3.6.1 Temperature

Semiconductor devices are generally more sensitive to temperature than are other electronic components. The various electrical characteristics of a semiconductor device are dependent on the ambient temperature at which the device is used. It is therefore necessary to understand the temperature characteristics of a device and to incorporate device derating into circuit design. Note also that if a device is used above its maximum temperature rating, device deterioration is more rapid and it will reach the end of its usable life sooner than expected.

3.6.2 Humidity

Resin-molded devices are sometimes improperly sealed. When these devices are used for an extended period of time in a high-humidity environment, moisture can penetrate into the device and cause chip degradation or malfunction. Furthermore, when devices are mounted on a regular printed circuit board, the impedance between wiring components can decrease under high-humidity conditions. In systems which require a high signal-source impedance, circuit board leakage or leakage between device lead pins can cause malfunctions. The application of a moisture-proof treatment to the device surface should be considered in this case. On the other hand, operation under low-humidity conditions can damage a device due to the occurrence of electrostatic discharge. Unless damp-proofing measures have been specifically taken, use devices only in environments with appropriate ambient moisture levels (i.e. within a relative humidity range of 40% to 60%).

3.6.3 Corrosive gases

Corrosive gases can cause chemical reactions in devices, degrading device characteristics. For example, sulphur-bearing corrosive gases emanating from rubber placed near a device (accompanied by condensation under high-humidity conditions) can corrode a device's leads. The resulting chemical reaction between leads forms foreign particles which can cause electrical leakage.

3.6.4 Radioactive and cosmic rays

Most industrial and consumer semiconductor devices are not designed with protection against radioactive and cosmic rays. Devices used in aerospace equipment or in radioactive environments must therefore be shielded.

3.6.5 Strong electrical and magnetic fields

Devices exposed to strong magnetic fields can undergo a polarization phenomenon in their plastic material, or within the chip, which gives rise to abnormal symptoms such as impedance changes or increased leakage current. Failures have been reported in LSIs mounted near malfunctioning deflection yokes in TV sets. In such cases the device's installation location must be changed or the device must be shielded against the electrical or magnetic field. Shielding against magnetism is especially necessary for devices used in an alternating magnetic field because of the electromotive forces generated in this type of environment.
3.6.6 Interference from light (ultraviolet rays, sunlight, fluorescent lamps and incandescent lamps)

Light striking a semiconductor device generates electromotive force due to photoelectric effects. In some cases the device can malfunction. This is especially true for devices in which the internal chip is exposed. When designing circuits, make sure that devices are protected against incident light from external sources. This problem is not limited to optical semiconductors and EPROMs. All types of device can be affected by light.

3.6.7 Dust and oil

Just like corrosive gases, dust and oil can cause chemical reactions in devices, which will adversely affect a device’s electrical characteristics. To avoid this problem, do not use devices in dusty or oily environments. This is especially important for optical devices because dust and oil can affect a device’s optical characteristics as well as its physical integrity and the electrical performance factors mentioned above.

3.6.8 Fire

Semiconductor devices are combustible; they can emit smoke and catch fire if heated sufficiently. When this happens, some devices may generate poisonous gases. Devices should therefore never be used in close proximity to an open flame or a heat-generating body, or near flammable or combustible materials.

3.7 Disposal of devices and packing materials

When discarding unused devices and packing materials, follow all procedures specified by local regulations in order to protect the environment against contamination.
4. Precautions and Usage Considerations

This section describes matters specific to each product group which need to be taken into consideration when using devices. If the same item is described in Sections 3 and 4, the description in Section 4 takes precedence.

4.1 Microcontrollers

4.1.1 Design

(1) Using resonators which are not specifically recommended for use

Resonators recommended for use with Toshiba products in microcontroller oscillator applications are listed in Toshiba databooks along with information about oscillation conditions. If you use a resonator not included in this list, please consult Toshiba or the resonator manufacturer concerning the suitability of the device for your application.

(2) Undefined functions

In some microcontrollers certain instruction code values do not constitute valid processor instructions. Also, it is possible that the values of bits in registers will become undefined. Take care in your applications not to use invalid instructions or to let register bit values become undefined.
1. Introduction

This user’s manual describes the C790 superscalar microprocessor for the system designer, paying special attention to the software interface and the bus interface.

The C790 is a superscalar integrated implementation of the subset of the 64-bit MIPS IV Instruction Set Architecture. It also implements a large extension to this instruction set specially tailored for multimedia applications. It contains a CPU, a floating point execution unit (Coprocessor 1), primary instruction and data caches.

Two instructions can be decoded each cycle. These instructions are issued in-order and are always completed in-order. Data cache misses are non-blocking. A single outstanding cache miss does not stall the pipeline, so that load misses or uncached loads are retired out-of-order. Multiply, Multiply-Accumulate, Divide, Prefetch, and Coprocessor 1 instructions are also retired out-of-order.

\footnote{However, some instructions are retired out-of-order.}
1.1 Features

The C790 core has the following features:

- 2-way superscalar pipeline
- 128-bit (two 64-bit) data path and 128-bit system bus
- Instruction set architecture
  - 64-bit MIPS III instruction set implementation (except LL, SC, LLD and SCD)
  - Selected MIPS IV instruction set implementation (Prefetch and Move conditional instructions)
  - Three-operand Multiply and Multiply-Accumulate instructions
  - 128-bit (Quadword) load/store instructions
  - 128-bit multimedia instructions which configure the 128-bit data path as two 64-bit, four 32-bit, eight 16-bit or sixteen 8-bit paths
- Configurable Endianness
- Branch prediction with Branch History Table (BHT) and Branch Target Address Cache (BTAC)
- Large on-chip caches
  - Instruction cache: 32KB, 2-way set associative
  - Data cache: 32KB, 2-way set-associative (with write-back protocol)
  - Non-blocking load, hit under miss and early restart on first quadword
  - Data cache line locking
  - Prefetch functions
  - 64 Byte cache line
- Fast integer Multiply and Multiply-Accumulate operations
- Memory management unit
  - 48-entry (96 pages) fully associative translation look-aside buffer (TLB)
  - 32-bit physical address space and 32-bit virtual address space
- IEEE 754-1985 compatible FPU (MIPS III ISA supported)
- Performance counters supported
- Debug support
  - Multi-stepping of instruction execution
  - Hardware breakpoint on instruction addresses
  - Hardware breakpoint on data address and data value
  - PC tracing capability
- 128-bit demultiplexed data bus and 32-bit address bus
  - Pipelined addresses
  - Bus error supported
  - Multiple masters supported
1.2 Related Documents

The following documents should be referenced:

[3] MIPS IV Instruction Set (Revision 3.2)
1.3 Revision History

Rev. 1.0: June 24th, 1999

Rev. 1.1: December 25th, 1999

Add IEEE 754 compatible FPU feature (both single- and double-precision)

Rev. 1.2: March, 2000

Publish

Rev. 2.0: April, 2001

Fixed a lot of typo
1.4 Conventions Used in This Manual

The names of registers, fields, and instructions are italicized as in this example:

The Status register (SR) is a read/write register that contains the operating mode, interrupt enabling, and diagnostic states of the processor.

When a name is first introduced, it is shown in bold type.

Ranges are denoted by a colon as in the following example:

The 4-bit Coprocessor Usability (CU[3:0]) field controls the usability of four possible coprocessors.

Conventions used in instruction descriptions are defined at the beginning of Appendices A, B, C, and D.
1.5 Restrictions for Use of the C790 CPU Core

1. Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>4/2/2001</td>
<td>FLX01-FLX06; Restrictions for User's Manual Rev.2.0</td>
</tr>
</tbody>
</table>

Items 1 through 6 in the description below are the restrictions that must be obeyed when using the C790 CPU core (User's Manual Rev.2.0).

Table 1-1. Restriction List

<table>
<thead>
<tr>
<th>ID</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLX01</td>
<td>TLB exceptions masks bus errors.</td>
</tr>
<tr>
<td>FLX02</td>
<td>Bus errors are masked when Status.ERL==1 or Status.EXL = 1.</td>
</tr>
<tr>
<td>FLX03</td>
<td>AdEL occurs in index-type ICACHE or BTAC CACHE instructions.</td>
</tr>
<tr>
<td>FLX04</td>
<td>kuseg becomes an uncached area when an error exception (Status.ERL = 1) occurs.</td>
</tr>
<tr>
<td>FLX05</td>
<td>First two instructions in an exception handler are executed as NOP when a bus error occurs.</td>
</tr>
<tr>
<td>FLX06</td>
<td>Unexpected instruction-fetch bus-errors occur when executing a Crashme program.</td>
</tr>
</tbody>
</table>
2. Description

2.1 TLB exceptions mask bus errors (FLX01)

2.1.1 Phenomenon

There are cases in which TLB exceptions occurring immediately after a bus error mask the bus error and the bus error can not be detected.

2.1.2 Corrective measures

This is caused by bus error exceptions having a lower priority than TLB exceptions in instruction fetch and data access (refer to “5.5.1 Exception Priority”). Check the followings when programming a TLB exception handler.

1) Using the TLB exception handler, check for occurrence of any bus error exceptions before a page refill.

2) Using the TLB exception handler, check for occurrence of any bus error exceptions if a page that should be refilled is incorrect.

3) Using the TLB exception handler, execute at Status.EXL==0 and Status.ERL==0 after the TLB exception handler stores to EPC, Cause, and Status registers.

Pending bus errors can be confirmed by referring to Status.BEM.
2.2 Bus errors are masked when Status.ERL==1 or Status.EXL = 1 (FLX02)

2.2.1 Phenomenon
Even if a bus error occurs during instruction fetch in an exception handler (Status.EXL==1 or Status.ERL==1), the CPU does not accept the exception and executes instruction code with indeterminate values read from the bus.

2.2.2 Corrective measures
This is caused by bus error exceptions being masked by Status.EXL==1 or Status.ERL==1. Do not cause exceptions due to instruction fetch in Status.EXL==1 or Status.ERL==1. Generating exceptions in an exception handler is dangerous. For example:

1) The JR instruction may potentially cause an address error or a bus error. Do not use JR instruction in Status.EXL==1 or Status.ERL==1.

2) A mapped region may potentially cause a TLB exception. Be sure to execute using an unmapped region like that below:
   0x8000_0000 - 0xFFFF_FFFF: kseg0
   0xA000_0000 - 0xBFFF_FFFF: kseg1
2.3 AdEL occurs in index-type ICACHE or BTAC CACHE instructions (FLX03)

2.3.1 Phenomenon

When executing index-type CACHE instructions below in either the User mode or Supervisor mode, operation occasionally becomes undefined and generates AdEL (Address Error exception; load and inst fetch).

There are five index-type ICACHE sub operations as listed below.

- 00111 CACHE IXIN I$ index invalidate
- 00000 CACHE IXLTG I$ index load tag
- 00100 CACHE IXSTG I$ index store tag
- 00001 CACHE IXLDT I$ index load data
- 00101 CACHE IXSDT I$ index store data

There are four BTAC CACHE sub operations as listed below.

- 00010 CACHE BXLBT index load BTAC
- 00110 CACHE BXSBT index store BTAC
- 01100 CACHE BFH BTAC flush
- 01010 CACHE BHINBT hit invalidate BTAC

However, there is no problem when Status.KSU==Kernel. Please note that Status.KSU==Kernel includes the kernel mode at Status.EXL==1 or Status.ERL==1 as well. There is also no problem when Status.CU[0]==0, and Status.KSU==User mode or Supervisor mode.

2.3.2 Corrective measures

In Status.CU[0]==1 and Status.KSU==Supervisor or User, execute under VA[31]==0 when executing either index-type ICACHE or BTAC CACHE instructions. VA here represents base reg + offset.
2.4 kuseg becomes an uncached area when an error exception (Status.ERL = 1) occurs (FLX04)

2.4.1 Phenomenon

There are cases in which kuseg (0x0000_0000 – 0x7FFF_FFFF) becomes uncached in an error exception handler (Status.ERL == 1) and data consistency with cached area (kseg, ksseg, kseg0) is lost.

2.4.2 Corrective measures

In an error exception handler (Status.ERL == 1), when accessing kuseg (0x0000_0000 – 0x7FFF_FFFF), access it after guarding using SYNC.L as follows:

```
SYNC.L
SW ku seg
```
2.5 First two instructions in an exception handler are executed as NOP when a bus error occurs (FLX05)

2.5.1 Phenomenon

There are cases in which the first two instructions in an exception handler are executed as NOP instructions, when certain exception occurs and then a bus error occurs immediately before jumping to the exception handler.

2.5.2 Corrective measures

Place NOP in the first two instruction locations in all exception handlers.
2.6 Unexpected instruction-fetch bus-errors occur when executing a Crashme program (FLX06)

2.6.1 Phenomenon

In Kernel mode or Supervisor mode, unexpected Instruction-fetch bus errors occur when attempting to execute a program called "Crashme" of Linux, since prohibited instruction-sequences that do not obey the following programming restrictions are executed.

In User mode, such a phenomenon doesn’t occur.

2.6.2 Corrective measures

In Kernel mode or Supervisor mode, obey the following programming restrictions:

1) Any CACHE instruction must not be placed in a branch delay slot.

2) SYNC.P must be located immediately before or immediately after any CACHE instruction.
Chapter 2  Architecture Overview

2. Architecture Overview

This chapter includes an overview of the C790 architecture. It discusses the following items:

- Block diagram and main modules
- Superscalar pipeline operation
- Instruction set
- Registers
- Memory Management
- Cache Memory
- Bus interface
- Floating Point Unit
- Performance Monitors
- Debug Support
2.1 Block Diagram and Functional Block Descriptions

This section presents a block diagram of the main modules of the C790 and summarizes the modules.

**Figure 2-1. C790 Block Diagram**
2.1.1 PC Unit

The 32-bit Program Counter (PC) holds the address of the instruction which is being executed. It also contains a 64-entry Branch Target Address Cache (BTAC) which stores branch target addresses used during branch prediction.

2.1.2 MMU

The Memory Management Unit supports the address translation functions of the CPU. It supplies the DTLB (Data Translation Lookaside Buffer) and ITLB (Instruction Translation Lookaside Buffer) with data via the TLB Refill Bus. Usage of these buffers is described in chapter 6.

2.1.3 Caches

Operation of the Instruction Cache and the Data Cache is described in Chapter 7. For each branch instruction, present in the instruction cache, two bits of branch history are stored in the Branch History Table (BHT).

2.1.4 Issue Logic and Staging Registers

The issue logic decides how to route instructions to appropriate pipes. It issues up to 2 instructions every cycle. Routing is described and discussed later in section 2.2.

2.1.5 GPR (General Purpose Registers) and FPR (Floating-Point Registers)

The General-Purpose Registers and the Floating-Point Registers are discussed in Section 2.3.

2.1.6 The Five Execution Pipes

2.1.6.1 I0 and I1 Pipes

There are two integer ALU pipelines (I0 and I1), each of which contains a complete 64-bit ALU, Shifter and Multiply-Accumulate unit. The I0 pipeline contains the SA register used for funnel shift operations. The two 64-bit ALU pipelines can be configured dynamically (on an instruction-by-instruction basis) into a single 128-bit execution pipeline to execute 128-bit Multimedia ALU, Shift and Multiply-Accumulate instructions. Furthermore, the two ALU pipelines share a single 128-bit multimedia aligner.

2.1.6.2 LS - Load/Store Pipe

The Load/Store (LS) pipe contains logic to support a single 128-bit Load and Store instruction.

2.1.6.3 BR - Branch Pipe

The Branch (BR) pipe contains logic to implement a single Branch instruction including Branch comparators.

2.1.6.4 C1 - COP1/FPU Pipe

The C1 pipe contains logic to support a single/double Floating Point coprocessor unit (COP1).
2.1.7 **Operand/Bypass logic**

This module takes data from the GPRs and from the Result and Move Buses, and routes the data to the pipelines.

2.1.8 **Response Buffer and Writeback Buffer**

The Writeback Buffer (WBB) is an 8 entry by 16 byte (one quadword) FIFO queuing up stores prior to accessing the CPU bus. It increases C790 performance by decoupling the processor from the latencies of the CPU bus. It is also used during the gathering operation of uncached accelerated stores; sequential stores less than a quadword in length are gathered in the WBB, thereby reducing bus bandwidth usage.

2.1.9 **UCAB**

The Uncached Accelerated Buffer (UCAB) is a 1 entry by 8 quadword buffer. It caches 128 sequential bytes of data during an uncached accelerated load miss. Subsequent loads from the uncached accelerated address space get their data from this buffer if the address hits in the UCAB, thereby eliminating bus latencies and providing higher performance.

2.1.10 **Result and Move Buses**

The Result and Move Buses convey data between execution units, the data cache, and the Operand/Bypass Logic unit.

2.1.11 **Bus Interface Unit and BIU Bus**

The BIU connects the core to the rest of the system. It interfaces the core’s internal bus signals to the CPU Bus.
2.2 Superscalar Pipeline Operation

The C790 has a six-stage superscalar pipeline. It can fetch, decode and execute a maximum of two instructions in parallel each cycle.

This section discusses in more detail the six execution pipelines listed in Section 2.1. It also discusses how instructions are routed among pipes.

2.2.1 Integer Instruction Pipeline Stages

The C790 contains four integer pipelines: the I0 and the I1 pipes, and the Load/Store and Branch pipes. Each pipe consists of the following six stages with each stage having 2 phases:

- I: Instruction Address Select
- Q: Instruction Queue
- R: Register Fetch
- A: Execution
- D: Data Fetch
- W: Write-back

Figure 2-2 shows the six stages of an integer instruction pipeline

![Diagram of the C790 Integer Instruction Pipeline]

Figure 2-2. C790 Integer Instruction Pipeline
I: Instruction Address Select
During the I stage, the following occurs:
- The sequential address is calculated
- The branch address is calculated
- The instruction address is selected from the following sources
  - Sequential address
  - Actual Branch/Jump address
  - Predicted Branch Target address from the BTAC
  - Exception vector address
  - EPC and Error PC

Q: Instruction Queue
During the Q stage, the following occurs:
- The instruction translation look-aside buffer (ITLB) does the virtual-to-physical address translation
- The instruction cache (data, Tag, steering bits & BHT) fetch begins
- TLB read for instruction fetch starts
- The instruction cache fetch is completed
- TLB read for instruction fetch completes
- The instruction cache Tag hit check is determined and the way selection is done
- The appropriate instructions are selected by the steering bits

R: Register Fetch
During the R stage the following occurs:
- Instructions are bussed to the appropriate execution units
- Register file is read
- Execution unit structural hazards are determined
- Instructions are decoded, data dependencies are determined and the appropriate instructions are issued

A: Execution
During the A stage, the following occurs:
- Results from the D or W stages are bypassed
- The execution units start and complete the integer arithmetic, logical, shift and multimedia instructions
- The iterative steps of the Multiply, Multiply-Accumulate, or Divide instructions are executed
- The virtual address for load and store instructions is calculated
- The branch condition is determined
- The DTLB is read
- The Data Cache and UCAB read starts
D: Data Fetch
During the D stage, the following occurs:
- The TLB read for a data access
- The Data Cache and UCAB read is completed
- The Data Cache Tag checking is completed
- Load or register data is obtained from COP1 (FPU)
- COP0 registers are read
- Data alignment and way selection is done for the data from the Data Cache
- Data sign extension is done
- Complete updating BHT bits and the BTAC
- All the exceptions are detected

W: Write Back
During the W stage, the following occurs:
- For store operations data is written to the Data Cache
- Data for coprocessor data transfer instructions is transferred to COP1 (FPU)
- For register-to-register and load instructions, the result is written to the register file
- COP0, COP1 (FPU) registers are written for coprocessor data transfer instructions
2.2.2 C1 (COP1/FPU) Instruction Pipeline Stages

The C790’s C1 (COP1/FPU) pipeline consists of the following eight stages:

- I: Instruction Address Select
- Q: Instruction Queue
- R: Register Fetch
- T: COP1 Register Fetch
- X: FP Execution 1st Stage
- Y: FP Execution 2nd Stage
- Z: FP Execution 3rd Stage
- S: Register File Write Stage

The eight stages of the pipeline for COP1/FPU are shown in Figure 2-3 with some pipeline stages identified with two letters. COP1 instructions execute simultaneously in the main integer pipeline I0 and the coprocessor 1 pipeline. The first letter identifies the main integer pipeline stage and the second letter identifies the coprocessor pipeline stage.

The I, Q, and R stages were previously described in Section 2.2.1. The following describes stages specific to the COP1 pipeline:

**T: COP1 Register Fetch**

During the T stage, the following occurs:

- Register file read for operands
- Bypass muxes from the S Stage/W Stage for S/T overlap.
X: FP Execution 1st Stage

This stage is the first step for floating point operations.

During the X stage, the following occurs:
- Detect Exceptions for input data.
- Detect Exception possibilities for result.
- The Booth function/Wallace multiplication is performed for multiply, the de-normalization is performed for add/subtract.

Y: FP Execution 2nd Stage

This stage is the second step for floating point operations. The following occurs:
- Test overflow/underflow on exponent is done
- Normalization for multiplication is done.
- Add/subtract the significand for add/subtract operations.
- Count leading zeros, to determine the shift amount for the normalization

Z: FP Execution 3rd Stage

This stage is the third step for floating point operations. The following occurs:
- Overflow/underflow detection
- Exponent readjustment
- Shift the significand for normalization
- Round the result
- Detect inexact exception

S: Register File Write Stage

During the S stage, the following occurs:
- FPR registers are written.
- FCSR31 is updated.
- Bypass values are passed to the T stage.
2.2.3 Classification and Routing of Instructions According to Execution Pipelines

This section discusses how the five execution pipelines are used in conjunction with instruction routing. Figure 2-4 identifies the specific execution pipelines into which instructions of a particular class are routed, and shows which physical execution units handle instructions from a particular logical pipe. Instruction categories are identified in italics, and are shown within the physical pipes where they are executed. ALU instructions can be executed in either integer pipe I0 or I1. COP1 Operate, and COP1 Move instructions execute in two pipes as shown, as does the Wide Operate.

![Instruction Routing Diagram]

Figure 2-4. Instruction Routing in Logical Pipes and Physical Pipes
Table 2-1 shows the categories of instructions and the execution pipelines that can execute those instructions. The instructions in a single category have the same issuing policy. Instructions which require more than a single execution pipeline are identified in the pipeline column with the (✔ &) symbol. For example, COP1 Move requires both the LS and the C1 execution pipelines. On the other hand, the ALU instructions can be executed in either the I0 or the I1 execution pipelines.

<table>
<thead>
<tr>
<th>Categories</th>
<th>Execution Pipeline</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/Store</td>
<td>✔ I0</td>
<td>Load, Store, Wide Load, Wide Store, Prefetch, CACHE</td>
</tr>
<tr>
<td></td>
<td>✔ I1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>✔ LS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>✔ BR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>✔ C1</td>
<td></td>
</tr>
<tr>
<td>Sync</td>
<td>✔</td>
<td>Synchronization</td>
</tr>
<tr>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>SA Operate</td>
<td>✔</td>
<td>Move to/from to SA register</td>
</tr>
<tr>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>COP0</td>
<td>✔</td>
<td>COP0 Coprocessor move, COP0 Coprocessor operations</td>
</tr>
<tr>
<td>COP1 Move</td>
<td>✔ &amp;</td>
<td>COP1 Coprocessor move, COP1 Coprocessor Load/Store</td>
</tr>
<tr>
<td>COP1 Operate</td>
<td>✔ &amp;</td>
<td>COP1 Operate Instructions</td>
</tr>
<tr>
<td>ALU</td>
<td>✔ ✔</td>
<td>Arithmetic, Shift, Logical, Trap, SYSCALL, BREAK</td>
</tr>
<tr>
<td>MAC0</td>
<td>✔</td>
<td>Multiply and Multiply -Accumulate for HI/LO register, MFHI/LO</td>
</tr>
<tr>
<td>MAC1</td>
<td>✔</td>
<td>Multiply and Multiply -Accumulate for HI1/LO1 register, MFHI1/LO1, MTHI1/LO1</td>
</tr>
<tr>
<td>Branch</td>
<td>✔</td>
<td>Branch, Jump, Jump/Link, All Coprocessor Branches</td>
</tr>
<tr>
<td>Wide Operate</td>
<td>✔ ✔</td>
<td>Wide ALU, Wide shift, Wide MAC, Funnel shift, Wide HI/LO Moves</td>
</tr>
</tbody>
</table>

1 COP1 Move instructions execute concurrently in the LS and the C1 pipes.
2 COP1 Operate instructions execute concurrently in the I0 and the C1 pipes.
3 ALU instructions can be executed in either the I0 or the I1 pipes.
4 Wide Operate instructions execute concurrently in the I0 and the I1 pipes.
2.2.4 Instruction Issue Combinations

The C790 always fetches two instructions. A pair of staging registers acts as a ‘bellows’ between the Q and the R stage. If an instruction can’t be issued in a particular cycle, it is saved in the staging registers. In the next cycle the C790 again fetches two instructions and tries to issue two (the one left over in the staging register from the previous cycle and the next sequential one from the pair that is fetched). So the C790 always tries to issue two instructions each cycle whenever it can.

The two instructions that get issued go to the R-stage of the pipeline and get associated with one of two logical pipes: Pipe0 and Pipe1. The instructions are then routed to an appropriate physical pipe for processing.

Instruction categories that can get issued to logical Pipe0 are:

1. ALU
2. Branch
3. Wide Operate
4. SA Operate
5. MAC0
6. COP1 Operate

An alternate way to view this is to recognize that logical Pipe0 is made up of the I0, C1 and BR execution pipelines. When issuing Wide Operate instructions logical Pipe0 also uses the I1 execution pipeline.

Instruction categories that can get issued to logical Pipe1 are:

1. ALU
2. Branch
3. SYNC
4. ERET
5. Load/Store
6. COP1 Move
7. COP0
8. MAC1

An alternate way to view this is to recognize that logical Pipe1 is made up of the I1, LS, C1 and BR execution pipelines.

All instruction categories are statically bound to a single logical pipe, that is, they can only be issued to a particular logical pipe. However the ALU and Branch instruction categories can get issued to either of the two logical pipes. Thus the binding of these two instruction categories to a particular logical pipe is done at instruction issue time.

There are some special cases of instruction sequences that are not allowed in the MIPS ISA. An instruction from the Branch category is not allowed to have another instruction from either the Branch or ERET category in its branch delay slot. So the following pairs of instructions are illegal and effectively never issued together:

1. Branch - Branch
2. Branch - ERET
The following sequences of instructions are also not allowed in the C790. Branch-Likely instructions are a subset of the Branch category (limited to the branch likely instructions).

1. Branch - SYNC.P
2. Branch - SYNC.L
3. Branch - CACHE *1
4. Branch-Likely - MTSA
5. Branch-Likely - MTSAB
6. Branch-Likely - MTSAH
7. Branch-Likely - TLBR *2
8. Branch-Likely - TLBWI *2
9. Branch-Likely - TLBWR *2

*1 CACHE instruction must be guarded by Sync instructions.

<table>
<thead>
<tr>
<th>Sync.P</th>
<th>Sync.L</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHE I$</td>
<td>CACHE D$</td>
</tr>
</tbody>
</table>

*2 TLBR, TLBWI, TLBWR instructions must be followed by Sync.P

The following table shows the instruction categories which can be issued concurrently to the two logical pipes. All combinations are legal except the ones marked with an “X”. The combinations marked with a “Y” can be issued concurrently, i.e., enter the R stage together but then the younger instruction stalls in the A stage for a single cycle in order to avoid a resource hazard.

Table 2-2. Concurrently Issued Instruction Categories

<table>
<thead>
<tr>
<th>LOGICAL PIPE0</th>
<th>LOGICAL PIPE1</th>
<th>LOGICAL PIPE1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/Store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYNC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LZC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COP1 Move</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAC1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COP0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SA Oper.</th>
<th>COP1 Oper.</th>
<th>ALU Oper.</th>
<th>MAC0 Oper.</th>
<th>Branch Oper.</th>
<th>Wide Oper.</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

X: illegal combination
Y: Can be issued concurrently but it will stall due to structure hazard.
2.3 Registers

The C790 extends the normal MIPS compatible register set by extending the general purpose registers (GPRs) from 64-bits to 128-bits, adding an additional pair of HI/LO registers for the I1 pipe and adding the SA register for the funnel shift instruction.

2.3.1 CPU Registers

The C790 has 128-bit wide GPRs. The upper 64 bits of the GPRs are only used by the C790-specific “Quad Load/Store”, and “Multimedia (Parallel)” instructions.

The HI1 and LO1, which are the upper 64 bits of each of the 128-bit HI and LO registers, are also used by new multiply and divide instructions, such as MULT1, MULTU1, DIV1, DIVU1, MADD1, MADDU1, MFHI1, MFLO1, MTHI1, and MTLO1, which are non-parallel I1 pipeline-specific instructions.

The SA register contains the shift amount used by the 256 bit funnel shift instruction.

2.3.2 FPU Registers

The floating point unit (COP1) has 64-bit wide floating point registers. It also contains 2 floating point control registers.
### 2.3.3 COP0 Registers

Table 2-3 identifies the COP0 registers of the C790.

<table>
<thead>
<tr>
<th>Register No.</th>
<th>Register Name</th>
<th>Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Index</td>
<td>Programmable register to select TLB entry for reading or writing</td>
<td>MMU</td>
</tr>
<tr>
<td>1</td>
<td>Random</td>
<td>Pseudo-random counter for TLB replacement</td>
<td>MMU</td>
</tr>
<tr>
<td>2</td>
<td>EntryLo0</td>
<td>Low half of TLB entry for even PFN (Physical page number)</td>
<td>MMU</td>
</tr>
<tr>
<td>3</td>
<td>EntryLo1</td>
<td>Low half of TLB entry for odd PFN (Physical page number)</td>
<td>MMU</td>
</tr>
<tr>
<td>4</td>
<td>Context</td>
<td>Pointer to kernel virtual PTE table</td>
<td>Exception</td>
</tr>
<tr>
<td>5</td>
<td>PageMask</td>
<td>Mask that sets the TLB page size</td>
<td>MMU</td>
</tr>
<tr>
<td>6</td>
<td>Wired</td>
<td>Number of wired TLB entries</td>
<td>MMU</td>
</tr>
<tr>
<td>7</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>8</td>
<td>BadVAddr</td>
<td>Bad virtual address</td>
<td>Exception</td>
</tr>
<tr>
<td>9</td>
<td>Count</td>
<td>Timer compare</td>
<td>Exception</td>
</tr>
<tr>
<td>10</td>
<td>EntryHi</td>
<td>High half of TLB entry (Virtual page number and ASID)</td>
<td>MMU</td>
</tr>
<tr>
<td>11</td>
<td>Compare</td>
<td>Timer compare</td>
<td>Exception</td>
</tr>
<tr>
<td>12</td>
<td>Status</td>
<td>Processor Status Register</td>
<td>Exception</td>
</tr>
<tr>
<td>13</td>
<td>Cause</td>
<td>Cause of the last exception taken</td>
<td>Exception</td>
</tr>
<tr>
<td>14</td>
<td>EPC</td>
<td>Exception Program Counter</td>
<td>Exception</td>
</tr>
<tr>
<td>15</td>
<td>PRId</td>
<td>Processor Revision Identifier</td>
<td>MMU</td>
</tr>
<tr>
<td>16</td>
<td>Config</td>
<td>Configuration Register</td>
<td>MMU</td>
</tr>
<tr>
<td>17</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>18</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>19</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>20</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>21</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>22</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>23</td>
<td>BadPAddr</td>
<td>Bad Physical Address</td>
<td>Exception</td>
</tr>
<tr>
<td>24</td>
<td>Debug</td>
<td>This is used for Debug function</td>
<td>Debug</td>
</tr>
<tr>
<td>25</td>
<td>Perf</td>
<td>Performance Counter and Control Register</td>
<td>Exception</td>
</tr>
<tr>
<td>26</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>27</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>28</td>
<td>TagLo</td>
<td>Cache Tag register (low bits)</td>
<td>MMU</td>
</tr>
<tr>
<td>29</td>
<td>TagHi</td>
<td>Cache Tag register (high bits)</td>
<td>MMU</td>
</tr>
<tr>
<td>30</td>
<td>ErrorPC</td>
<td>Error Exception Program Counter</td>
<td>Exception</td>
</tr>
<tr>
<td>31</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
2.4 Memory Management

The C790 processor provides a memory management unit (MMU) which uses an on-chip translation look-aside buffer (TLB) to translate virtual addresses into physical addresses.

The C790 supports the MIPS compatible 32-bit address and 64-bit data mode. Only 32-bit virtual and physical addresses have been implemented. There is no requirement for address sign extension. Address error exception checking will not be done on the “upper” 32-bits (which are ignored). The only condition that will generate the address error exception will be address alignment errors and segment protection errors. In Kernel mode, it is free from address error exception for program counter to wrap-around from kseg3 to kuseg.

Since there is only one addressing mode, all the four MIPS ISAs (I, II, III, IV) and the C790 specific ISA are available without any restrictions in all of the three processor modes (with the appropriate MIPS ISA coprocessor usable restrictions). As such the reserved instruction (RI) exception will occur only when the processor really tries to execute an undefined opcode.

Features

- MIPS III-compatible 32-bit MMU
- Operating Modes: User, Supervisor, and Kernel
- TLB: 48 entries of even/odd page pairs (96 pages)  
  Fully associative
- Page Size: 4 KB, 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, 16 MB
- ITLB: 2 entries
- DTLB: 4 entries
- Address Sizes: Virtual Address Size = 32 bit, 2 Gbyte per user Process  
  Physical Address Size = 32 bit, 4 Gbyte
2.5 Cache Memory

The C790 core contains both an instruction cache and a separate data cache.

Features

The following are the main features of the caches:

- Separate Instruction Cache and Data Cache
- Virtually indexed and physically tagged caches
- Write-back policy for the Data Cache
- Data Cache and Instruction Cache burst read sequential ordering
- Cache Size: Instruction Cache: 32 KB
  Data Cache: 32 KB
- Line Size: 64 Bytes
- Refill size: 64 Bytes
- Associativity: 2-way set-associative
- Write Policy: Write-back and write allocate
- Data order for block reads: Sequential ordering
- Data order for block writes: Sequential ordering
- Instruction cache miss restart: After all data received
- Data cache miss restart: Early restart on first quadword
- Cache parity: No
- Cache Locking: Data Cache Line Lock. Controlled by CACHE instruction
- Cache Snooping: No
- Non-blocking load: Yes
- Hit Under Miss: Yes (Multiple hits under one miss are supported)
- Data Cache Prefetch: Yes
2.6 Bus Interface

The C790 CPU core is connected to the rest of the system, and to external devices, through the group of on-chip C790 system bus signals called the CPU Bus.

Features

- Separate data and address buses (Demultiplexed operation)
- 128-bit data bus
- Clocked synchronous operations
- Peak transfer rate of 2.1 GB/sec (@133 MHz bus clock)
- 8/16/32/64/128-bit and burst accesses
- Multimaster capability
- Pipelined operations
- No turn-around or dead cycles between transfers

The CPU Bus does not provide:

- Cache coherency support
- Split transactions

2.7 Floating Point Unit

The floating point unit is IEEE754-1985 compatible as same as FPU in the TX49HF CPU core.

Main Features:

- Tightly coupled to the C790 Integer pipeline.
- Supports both double and single precision format as defined in IEEE-754 specification
- No hardware support for Denormalized number in the IEEE-754 specification. Software (exception handler) supports it.
- The FPU supports five IEEE exceptions and one MIPS defined exception.
- *ADD, SUB, MUL, DIV, ABS, MOV, NEG, SQRT, compare and convert are supported*
2.8 Performance Counter

The performance counter provides the means for gathering statistical information about the internal events of the CPU and the pipeline during program execution. The statistics gathered during program execution aid in tuning the performance of hardware and software systems based on the processor.

The performance counter consists of one control register and two counters. The control register controls the functions of the performance counter while the counters count the number of events specified by the control register.

**Features:**
- Two performance counter registers
- Over twenty different events within the processor can be counted
- Counting can be selectively enabled in User, Supervisor, Kernel, and Exception modes

2.9 Debug and Tracing Functions

The C790 supports real-time PC tracing. Pipeline status, target addresses of indirect jumps, and exception vectors are made available on special signals. The executed instruction sequence can be restored from signals and the source program.

**Features:**
- One Instruction Address Breakpoint register
- One Instruction Address Breakpoint Mask register
- One Data Address Breakpoint register
- One Data Address Breakpoint Mask register
- One Data Value Breakpoint register
- One Data Value Breakpoint Mask register
- Each breakpoint individually enabled
- Breakpoint function can be selectively enabled in User, Supervisor, Kernel, and Exception modes
- External Trigger signal can be generated when breakpoint occurs
- 11 signals used to provide real-time PC tracing function
3. Instruction Set Overview and Summary

This chapter provides an overview of the C790 instruction set. Refer to Appendices A - D for detailed descriptions of individual instructions.
3.1 Introduction

The C790 supports all MIPS III instructions with the exception of 64-bit multiply, 64-bit divide, Load Linked and Store Conditional instructions. It also supports a limited number of MIPS IV instructions and additional C790-specific instructions, such as Multiply/Add instructions and multimedia instructions.

The instruction set can be divided into the following groups:

- Load and Store
- Computational
- Jump and Branch
- Miscellaneous
- System Control Coprocessor (COP0)
- Coprocessor 1 (COP1)
- C790-specific
### 3.2 CPU Instruction Set Formats

There are three instruction formats: **immediate** (I-type), **jump** (J-type), and **register** (R-type), as shown in Figure 3-1. The use of a small number of instruction formats simplifies instruction decoding (thus producing higher frequency operations) and allows the compiler to synthesize more complicated (and less frequently used) operations and address modes from these three formats as needed.

<table>
<thead>
<tr>
<th>I-type (Immediate)</th>
<th>J-type (Jump)</th>
<th>R-type (Register)</th>
</tr>
</thead>
<tbody>
<tr>
<td>op 31</td>
<td>rs 26 25</td>
<td>rt 21 20</td>
</tr>
<tr>
<td>op 31</td>
<td>rs 26 25</td>
<td>rt 21 20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>6-bit operation code</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs</td>
<td>5-bit source register specifier</td>
</tr>
<tr>
<td>rt</td>
<td>5-bit target (source/destination) register or branch condition</td>
</tr>
<tr>
<td>immediate</td>
<td>16-bit immediate value, branch displacement or address displacement</td>
</tr>
<tr>
<td>target</td>
<td>26-bit jump target address</td>
</tr>
<tr>
<td>rd</td>
<td>5-bit destination register specifier</td>
</tr>
<tr>
<td>sa</td>
<td>5-bit shift amount</td>
</tr>
<tr>
<td>funct</td>
<td>6-bit function field</td>
</tr>
</tbody>
</table>

Figure 3-1. CPU Instruction Formats
3.3 Instruction Set Summary

The C790 supports MIPS III instructions\(^1\) as well as a limited number of MIPS IV instructions. A large number of C790-specific instructions, such as multiply/add instructions and multimedia instructions have also been implemented.

3.3.1 Load/Store Instructions

The instructions in this group transfer data of different sizes: bytes, halfwords, words, doublewords and quadwords. Signed and unsigned integers of different sizes are supported by loads that either sign-extended or zero-extended the data loaded into the register.

Load and store instructions are immediate (I-type) instructions that move data between memory and the general registers. The only addressing mode that load and store instructions directly support is base register plus 16-bit signed immediate offset.

3.3.1.1 Normal Loads and Stores

The C790 does not support Load Linked and Store Conditional instructions, LL, LLD, SC and SCD. For details of these instructions refer to Appendix A.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>Load Byte</td>
<td>MIPS I</td>
</tr>
<tr>
<td>LBU</td>
<td>Load Byte Unsigned</td>
<td>MIPS I</td>
</tr>
<tr>
<td>LD</td>
<td>Load Doubleword</td>
<td>MIPS III</td>
</tr>
<tr>
<td>LDL</td>
<td>Load Doubleword Left</td>
<td>MIPS III</td>
</tr>
<tr>
<td>LDR</td>
<td>Load Doubleword Right</td>
<td>MIPS III</td>
</tr>
<tr>
<td>LH</td>
<td>Load Halfword</td>
<td>MIPS I</td>
</tr>
<tr>
<td>LHU</td>
<td>Load Halfword Unsigned</td>
<td>MIPS I</td>
</tr>
<tr>
<td>LW</td>
<td>Load Word</td>
<td>MIPS I</td>
</tr>
<tr>
<td>LWL</td>
<td>Load Word Left</td>
<td>MIPS I</td>
</tr>
<tr>
<td>LWR</td>
<td>Load Word Right</td>
<td>MIPS I</td>
</tr>
<tr>
<td>LWU</td>
<td>Load Word Unsigned</td>
<td>MIPS III</td>
</tr>
<tr>
<td>SB</td>
<td>Store Byte</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SD</td>
<td>Store Doubleword</td>
<td>MIPS III</td>
</tr>
<tr>
<td>SDL</td>
<td>Store Doubleword Left</td>
<td>MIPS III</td>
</tr>
<tr>
<td>SDR</td>
<td>Store Doubleword Right</td>
<td>MIPS III</td>
</tr>
<tr>
<td>SH</td>
<td>Store Halfword</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SW</td>
<td>Store Word</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SWL</td>
<td>Store Word Left</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SWR</td>
<td>Store Word Right</td>
<td>MIPS I</td>
</tr>
</tbody>
</table>

\(^1\) Note: The C790 does not support the following MIPS III instructions:

- 64-bit multiply and divide instructions (DMULT, DMULTU, DDIV, DDIVU)
- Semaphore instructions (LL, LLD, SC, SCD)
3.3.1.2 Multimedia Loads and Stores

The C790 implements 128-bit (quadword) load and store instructions for multimedia purpose. For details of these instructions refer to Appendix B.

Table 3-2. Multimedia Load / Store Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>LQ</td>
<td>Load Quadword</td>
<td>C790</td>
</tr>
<tr>
<td>SQ</td>
<td>Store Quadword</td>
<td>C790</td>
</tr>
</tbody>
</table>

3.3.1.3 Coprocessor Loads and Stores

These loads and stores are coprocessor instructions. A particular coprocessor is enabled if corresponding CU bit is set in CP0 Status register. Otherwise executing one of these instructions generates a Coprocessor Unusable exception. For details of these instructions refer to Appendices C and D.

Table 3-3. Coprocessor Load / Store Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDC1</td>
<td>Load Doubleword to Floating Point</td>
<td>MIPS II</td>
</tr>
<tr>
<td>LWC1</td>
<td>Load Word to Floating Point</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SDC1</td>
<td>Store Doubleword from Floating Point</td>
<td>MIPS II</td>
</tr>
<tr>
<td>SWC1</td>
<td>Store Word from Floating Point</td>
<td>MIPS I</td>
</tr>
</tbody>
</table>

3.3.1.4 Data Formats and Addressing

The C790 processor uses five data formats:

- 128-bit quadword
- 64-bit doubleword
- 32-bit word
- 16-bit halfword
- 8-bit byte

Byte ordering within each of the larger data formats — halfword, word, doubleword — can be configured in either big-endian or little-endian order. Endianness refers to the location of byte 0 within the multi-byte data structure. Figure 3-2 and Figure 3-3 show the ordering of bytes within words and the ordering of words within multiple-word structures for the big-endian and little-endian conventions.

When the C790 processor is configured as a big-endian system, byte 0 is the most-significant (leftmost) byte, thereby providing compatibility with MC 68000® and IBM 370® conventions. Figure 3-2 shows this configuration.
Figure 3-2. Big-Endian Byte Ordering

When configured as a little-endian system, byte 0 is always the least-significant (rightmost) byte, which is compatible with iAPX® x86 and DEC VAX® conventions.

Figure 3-3. Little-Endian Byte Ordering

In this text, bit 0 is always the least-significant (rightmost) bit: thus, bit designations are always little-endian (although no instructions explicitly designate bit positions within words).
Figure 3-4 and Figure 3-5 show little-endian and big-endian byte ordering in doublewords.
The CPU uses byte addressing for halfword, word, doubleword, and quadword accesses with the following alignment constraints:

- Halfword accesses must be aligned on an even byte boundary (0, 2, 4...).
- Word accesses must be aligned on a byte boundary divisible by four (0, 4, 8...).
- Doubleword accesses must be aligned on a byte boundary divisible by eight (0, 8, 16...).
- Quadword accesses must be aligned on a byte boundary divisible by sixteen (0, 16, 32...).

The following special instructions load and store words that are not aligned on 4-byte (word), 8-byte (doubleword), boundaries:

LWL  LWR  SWL  SWR
LDL  LDR  SDL  SDR

These instructions are used in pairs to provide addressing of misaligned words. Addressing misaligned data incurs one additional instruction cycle over that required for addressing aligned data. This extra cycle is because of an extra instruction for the “pair” (e.g., LWL and LWR form a pair). Also note that the CPU moves the unaligned data at the same rate as a hardware mechanism.

Figure 3-6 and Figure 3-7 show the access of a misaligned word that has byte address 3.
3.3.1.5 Defining Access Types

Access type indicates the size of the C790 processor data item to be loaded or stored, set by the load or store instruction opcode.

Regardless of access type or byte ordering ( endianess ), the address given specifies the low-order byte in the addressed field. For a big-endian configuration, the low-order byte is the most-significant byte; for a little-endian configuration, the low-order byte is the least-significant byte.

The access type, together with the four low-order bits of the address, defines the bytes accessed within the addressed doubleword (shown in Table 3-4 and Table 3-5). Only the combinations shown in Table 3-4 and Table 3-5 are permissible; other combinations cause address error exceptions.
Table 3-4. Defining Access Types (Big-Endian)

<table>
<thead>
<tr>
<th>Access Type Mnemonic</th>
<th>Low-Order Address Bits</th>
<th>Bytes Accessed Big endian</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3 2 1 0</td>
<td>(127---------------95----------------63-----------------31-----------------0) Byte</td>
</tr>
<tr>
<td>Quadword</td>
<td>0 0 0 0</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</td>
</tr>
<tr>
<td>Doubleword</td>
<td>0 0 0 0</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0</td>
<td>8 9 10 11 12 13 14 15</td>
</tr>
<tr>
<td>Septibyte</td>
<td>0 0 0 0</td>
<td>0 1 2 3 4 5 6</td>
</tr>
<tr>
<td></td>
<td>0 0 0 1</td>
<td>1 2 3 4 5 6 7</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0</td>
<td>8 9 10 11 12 13 14 15</td>
</tr>
<tr>
<td></td>
<td>1 0 0 1</td>
<td>9 10 11 12 13 14 15</td>
</tr>
<tr>
<td>Sextibyte</td>
<td>0 0 0 0</td>
<td>0 1 2 3 4</td>
</tr>
<tr>
<td></td>
<td>0 0 1 0</td>
<td>2 3 4 5 6 7</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0</td>
<td>8 9 10 11 12 13</td>
</tr>
<tr>
<td></td>
<td>1 0 1 0</td>
<td>10 11 12 13 14 15</td>
</tr>
<tr>
<td>Quintibyte</td>
<td>0 0 0 0</td>
<td>0 1 2 3 4</td>
</tr>
<tr>
<td></td>
<td>0 0 1 1</td>
<td>3 4 5 6 7</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0</td>
<td>8 9 10 11 12</td>
</tr>
<tr>
<td></td>
<td>1 0 1 1</td>
<td>11 12 13 14 15</td>
</tr>
<tr>
<td>Word</td>
<td>0 0 0 0</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0</td>
<td>4 5 6 7</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0</td>
<td>8 9 10 11</td>
</tr>
<tr>
<td></td>
<td>1 1 0 0</td>
<td>12 13 14 15</td>
</tr>
<tr>
<td>Triplebyte</td>
<td>0 0 0 0</td>
<td>0 1 2</td>
</tr>
<tr>
<td></td>
<td>0 0 0 1</td>
<td>1 2 3</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0</td>
<td>4 5 6</td>
</tr>
<tr>
<td></td>
<td>0 1 0 1</td>
<td>5 6 7</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0</td>
<td>8 9 10</td>
</tr>
<tr>
<td></td>
<td>1 0 0 1</td>
<td>9 10 11</td>
</tr>
<tr>
<td></td>
<td>1 1 0 0</td>
<td>12 13 14</td>
</tr>
<tr>
<td></td>
<td>1 1 0 1</td>
<td>13 14 15</td>
</tr>
<tr>
<td>Halfword</td>
<td>0 0 0 0</td>
<td>0 1</td>
</tr>
<tr>
<td></td>
<td>0 0 1 0</td>
<td>2 3</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0</td>
<td>4 5</td>
</tr>
<tr>
<td></td>
<td>0 1 1 0</td>
<td>6 7</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0</td>
<td>8 9</td>
</tr>
<tr>
<td></td>
<td>1 0 1 0</td>
<td>10 11</td>
</tr>
<tr>
<td></td>
<td>1 1 0 0</td>
<td>12 13</td>
</tr>
<tr>
<td></td>
<td>1 1 1 0</td>
<td>14 15</td>
</tr>
<tr>
<td>Access Type Mnemonic</td>
<td>Low-Order Address Bits</td>
<td>Bytes Accessed</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Byte</td>
<td></td>
<td>(127----------95----------63----------31----------0)</td>
</tr>
<tr>
<td>Byte</td>
<td>0 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>Byte</td>
<td>0 0 0 1</td>
<td>1</td>
</tr>
<tr>
<td>Byte</td>
<td>0 0 1 0</td>
<td>2</td>
</tr>
<tr>
<td>Byte</td>
<td>0 0 1 1</td>
<td>3</td>
</tr>
<tr>
<td>Byte</td>
<td>0 1 0 0</td>
<td>4</td>
</tr>
<tr>
<td>Byte</td>
<td>0 1 0 1</td>
<td>5</td>
</tr>
<tr>
<td>Byte</td>
<td>0 1 1 0</td>
<td>6</td>
</tr>
<tr>
<td>Byte</td>
<td>0 1 1 1</td>
<td>7</td>
</tr>
<tr>
<td>Byte</td>
<td>1 0 0 0</td>
<td>8</td>
</tr>
<tr>
<td>Byte</td>
<td>1 0 0 1</td>
<td>9</td>
</tr>
<tr>
<td>Byte</td>
<td>1 0 1 0</td>
<td>10</td>
</tr>
<tr>
<td>Byte</td>
<td>1 0 1 1</td>
<td>11</td>
</tr>
<tr>
<td>Byte</td>
<td>1 1 0 0</td>
<td>12</td>
</tr>
<tr>
<td>Byte</td>
<td>1 1 0 1</td>
<td>13</td>
</tr>
<tr>
<td>Byte</td>
<td>1 1 1 0</td>
<td>14</td>
</tr>
<tr>
<td>Byte</td>
<td>1 1 1 1</td>
<td>15</td>
</tr>
</tbody>
</table>
### Table 3-5. Defining Access Types (Little-Endian)

<table>
<thead>
<tr>
<th>Access Type Mnemonic</th>
<th>Low-Order Address Bits</th>
<th>Bytes Accessed Little endian</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3  2  1  0</td>
<td>(127---------------95----------------63-----------------31-----------------0) Byte</td>
</tr>
<tr>
<td>Quadword</td>
<td>0  0  0  0</td>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>Doubleword</td>
<td>0  0  0  0</td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td></td>
<td>1  0  0  0</td>
<td>15 14 13 12 11 10 9 8</td>
</tr>
<tr>
<td>Septibyte</td>
<td>0  0  0  0</td>
<td>6 5 4 3 2 1 0</td>
</tr>
<tr>
<td></td>
<td>0  0  0  1</td>
<td>7 6 5 4 3 2 1</td>
</tr>
<tr>
<td></td>
<td>1  0  0  0</td>
<td>14 13 12 11 10 9 8</td>
</tr>
<tr>
<td></td>
<td>1  0  0  1</td>
<td>15 14 13 12 11 10 9</td>
</tr>
<tr>
<td>Sextibyte</td>
<td>0  0  0  0</td>
<td>5 4 3 2 1 0</td>
</tr>
<tr>
<td></td>
<td>0  0  1  0</td>
<td>7 6 5 4 3 2</td>
</tr>
<tr>
<td></td>
<td>1  0  0  0</td>
<td>13 12 11 10 9 8</td>
</tr>
<tr>
<td></td>
<td>1  0  1  0</td>
<td>15 14 13 12 11 10</td>
</tr>
<tr>
<td>Quintibyte</td>
<td>0  0  0  0</td>
<td>4 3 2 1 0</td>
</tr>
<tr>
<td></td>
<td>0  0  1  1</td>
<td>7 6 5 4 3</td>
</tr>
<tr>
<td></td>
<td>1  0  0  0</td>
<td>12 11 10 9 8</td>
</tr>
<tr>
<td></td>
<td>1  0  1  1</td>
<td>15 14 13 12 11</td>
</tr>
<tr>
<td>Word</td>
<td>0  0  0  0</td>
<td>3 2 1 0</td>
</tr>
<tr>
<td></td>
<td>0  0  1  0</td>
<td>7 6 5 4</td>
</tr>
<tr>
<td></td>
<td>1  0  0  0</td>
<td>11 10 9 8</td>
</tr>
<tr>
<td></td>
<td>1  1  0  0</td>
<td>15 14 13 12</td>
</tr>
<tr>
<td>Triplebyte</td>
<td>0  0  0  0</td>
<td>2 1 0</td>
</tr>
<tr>
<td></td>
<td>0  0  0  1</td>
<td>3 2 1</td>
</tr>
<tr>
<td></td>
<td>0  1  0  0</td>
<td>6 5 4</td>
</tr>
<tr>
<td></td>
<td>0  1  0  1</td>
<td>7 6 5</td>
</tr>
<tr>
<td></td>
<td>1  0  0  0</td>
<td>10 9 8</td>
</tr>
<tr>
<td></td>
<td>1  0  0  1</td>
<td>11 10 9</td>
</tr>
<tr>
<td></td>
<td>1  1  0  0</td>
<td>14 13 12</td>
</tr>
<tr>
<td></td>
<td>1  1  0  1</td>
<td>15 14 13</td>
</tr>
<tr>
<td>Halfword</td>
<td>0  0  0  0</td>
<td>1 0</td>
</tr>
<tr>
<td></td>
<td>0  0  1  0</td>
<td>3 2</td>
</tr>
<tr>
<td></td>
<td>0  1  0  0</td>
<td>5 4</td>
</tr>
<tr>
<td></td>
<td>0  1  1  0</td>
<td>7 6</td>
</tr>
<tr>
<td></td>
<td>1  0  0  0</td>
<td>9 8</td>
</tr>
<tr>
<td></td>
<td>1  0  1  0</td>
<td>11 10</td>
</tr>
<tr>
<td></td>
<td>1  1  0  0</td>
<td>13 12</td>
</tr>
<tr>
<td></td>
<td>1  1  1  0</td>
<td>15 14</td>
</tr>
</tbody>
</table>
3.3.1.6 Scheduling a Load Delay Slot

A load instruction that does not allow its result to be used by the instruction immediately following is called a *delayed load instruction*. The instruction slot immediately following this delayed load instruction is referred to as the *load delay slot*.

In the C790 processor, the instruction immediately following a load instruction can use the contents of the loaded register. In such cases, however, hardware interlocks insert additional clock cycles. Consequently, scheduling load delay slots can be desirable, both for performance and R-Series processor compatibility. However, the scheduling of load delay slots is not absolutely required.
3.3.2 Computational Instructions

The instructions in this group perform two’s complement arithmetic, logical operations, or shifts on integers represented in two’s complement notation.

Computational instructions can be either in register (R-type) format, in which both operands are registers, or in immediate (I-type) format, in which one operand is a 16-bit immediate.

Computational instructions perform the following operations on register values:

- Arithmetic
- Logical
- Shift
- Multiply
- Divide

These operations fit in the following four categories of computational instructions:

- ALU immediate instructions
- Three-Operand Register-Type instructions
- Shift instructions
- Multiply and Divide instructions

For detailed information of individual instructions, refer to Appendix A.

*Note: The C790 does not support 64-bit Multiply and Divide instructions, DMULT, DMULTU, DDIV, and DDIVU.

3.3.2.1 ALU Immediate Instructions

Table 3-6. ALU Immediate Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td>Add Immediate</td>
<td>MIPS I</td>
</tr>
<tr>
<td>ADDIU</td>
<td>Add Immediate Unsigned</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SLTI</td>
<td>Set on Less Than Immediate</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SLTIU</td>
<td>Set on Less Than Immediate Unsigned</td>
<td>MIPS I</td>
</tr>
<tr>
<td>ANDI</td>
<td>AND Immediate</td>
<td>MIPS I</td>
</tr>
<tr>
<td>ORI</td>
<td>OR Immediate</td>
<td>MIPS I</td>
</tr>
<tr>
<td>XORI</td>
<td>Exclusive OR Immediate</td>
<td>MIPS I</td>
</tr>
<tr>
<td>LUI</td>
<td>Load Upper Immediate</td>
<td>MIPS I</td>
</tr>
<tr>
<td>DADDI</td>
<td>Doubleword Add Immediate</td>
<td>MIPS III</td>
</tr>
<tr>
<td>DADDIU</td>
<td>Doubleword Add Immediate Unsigned</td>
<td>MIPS III</td>
</tr>
</tbody>
</table>
### 3.3.2.2 Three Operand Register-Type Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add</td>
<td>MIPS I</td>
</tr>
<tr>
<td>ADDU</td>
<td>Add Unsigned</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SUBU</td>
<td>Subtract Unsigned</td>
<td>MIPS I</td>
</tr>
<tr>
<td>DADD</td>
<td>Doubleword Add</td>
<td>MIPS III</td>
</tr>
<tr>
<td>DADDU</td>
<td>Doubleword Add Unsigned</td>
<td>MIPS III</td>
</tr>
<tr>
<td>DSUB</td>
<td>Doubleword Subtract</td>
<td>MIPS III</td>
</tr>
<tr>
<td>DSUBU</td>
<td>Doubleword Subtract Unsigned</td>
<td>MIPS III</td>
</tr>
<tr>
<td>SLT</td>
<td>Set Less Than</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SLTU</td>
<td>Set Less Than Unsigned</td>
<td>MIPS I</td>
</tr>
<tr>
<td>AND</td>
<td>AND</td>
<td>MIPS I</td>
</tr>
<tr>
<td>OR</td>
<td>OR</td>
<td>MIPS I</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive OR</td>
<td>MIPS I</td>
</tr>
<tr>
<td>NOR</td>
<td>NOR</td>
<td>MIPS I</td>
</tr>
</tbody>
</table>

### 3.3.2.3 Shift Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLL</td>
<td>Shift Left Logical</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SRL</td>
<td>Shift Right Logical</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SRA</td>
<td>Shift Right Arithmetic</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SLLV</td>
<td>Shift Left Logical Variable</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SRLV</td>
<td>Shift Right Logical Variable</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SRAV</td>
<td>Shift Right Arithmetic Variable</td>
<td>MIPS I</td>
</tr>
<tr>
<td>DSLL</td>
<td>Doubleword Shift Left Logical</td>
<td>MIPS III</td>
</tr>
<tr>
<td>DSRL</td>
<td>Doubleword Shift Right Logical</td>
<td>MIPS III</td>
</tr>
<tr>
<td>DSRA</td>
<td>Doubleword Shift Right Arithmetic</td>
<td>MIPS III</td>
</tr>
<tr>
<td>DSLL32</td>
<td>Doubleword Shift Left Logical + 32</td>
<td>MIPS III</td>
</tr>
<tr>
<td>DSRL32</td>
<td>Doubleword Shift Right Logical + 32</td>
<td>MIPS III</td>
</tr>
<tr>
<td>DSRA32</td>
<td>Doubleword Shift Right Arithmetic + 32</td>
<td>MIPS III</td>
</tr>
<tr>
<td>DSSLV</td>
<td>Doubleword Shift Left Logical Variable</td>
<td>MIPS III</td>
</tr>
<tr>
<td>DSRLV</td>
<td>Doubleword Shift Right Logical Variable</td>
<td>MIPS III</td>
</tr>
<tr>
<td>DSRAV</td>
<td>Doubleword Shift Right Arithmetic Variable</td>
<td>MIPS III</td>
</tr>
</tbody>
</table>

### 3.3.2.4 Multiply and Divide Instructions

These are the standard MIPS instructions for multiply, divide, and move to/from HI/LO registers executed on the I0 pipeline’s MAC unit. See also C790-specific Multiply and Divide instructions discussion.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULT</td>
<td>Multiply</td>
<td>MIPS I</td>
</tr>
<tr>
<td>MULTU</td>
<td>Multiply Unsigned</td>
<td>MIPS I</td>
</tr>
<tr>
<td>DIV</td>
<td>Divide</td>
<td>MIPS I</td>
</tr>
<tr>
<td>DIVU</td>
<td>Divide Unsigned</td>
<td>MIPS I</td>
</tr>
<tr>
<td>MFHI</td>
<td>Move From HI</td>
<td>MIPS I</td>
</tr>
<tr>
<td>MTHI</td>
<td>Move To HI</td>
<td>MIPS I</td>
</tr>
<tr>
<td>MFLO</td>
<td>Move From LO</td>
<td>MIPS I</td>
</tr>
<tr>
<td>MTLO</td>
<td>Move To LO</td>
<td>MIPS I</td>
</tr>
</tbody>
</table>

### 3.3.2.5 64-Bit Operations

The result of operations that use incorrect sign-extended 32-bit values for 64-bit operations is unpredictable.
3.3.3 Jump and Branch Instructions

The architecture defines PC-relative conditional branches, a PC-region unconditional jump, an absolute (register) unconditional jump, and a similar set of procedure calls that record a return link address in a general register. For convenience, these are all referred to here as branches.

All branches have an architectural delay of one instruction. When a branch is taken, the instruction immediately following the branch instruction, in the branch delay slot, is executed before the branch to the target instruction takes place. Conditional branches come in two versions that treat the instruction in the delay slot differently when the branch is not taken and execution falls through. The ‘branch’ instructions execute the instruction in the delay slot, but the ‘branch likely’ instructions do not. (They are said to ‘nullify’ it.)

By convention, if an exception or interrupt prevents the completion of an instruction occupying a branch delay slot, the instruction stream is continued by re-executing the branch instruction. To permit this, branches must be restartable; procedure calls may not use the register in which the return link is stored (usually register 31) to determine the branch target address.

For detailed information of individual instructions, refer to Appendix A. Branch on Coprocessor instructions are covered under coprocessor’s discussions.

3.3.3.1 Jump Instructions

Subroutine calls in high-level languages are usually implemented with Jump or Jump and Link instructions, both of which are J-type instructions. In J-type format, the 26-bit target address shifts 2 bits and combines with the high-order 4-bits of the current program counter to form an absolute address.

Returns, dispatches, and large cross-page jumps are usually implemented with the Jump Register or Jump and Link Register instructions. Both are R-type instructions that take the 32-bit byte address contained in one of the general purpose registers.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>Jump</td>
<td>MIPS I</td>
</tr>
<tr>
<td>JAL</td>
<td>Jump and Link</td>
<td>MIPS I</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>JR</td>
<td>Jump Register</td>
<td>MIPS I</td>
</tr>
<tr>
<td>JALR</td>
<td>Jump and Link Register</td>
<td>MIPS I</td>
</tr>
</tbody>
</table>
3.3.3.2 Branch Instructions

All branch instruction target addresses are computed by adding the address of the instruction in the branch delay slot to the 16-bit offset (shifts left 2 bits and is sign-extended to 32-bits). All branches occur with a delay of one instruction.

In case of a Branch Likely instruction, if a condition is not taken, the instruction in the delay slot is nullified.

Table 3-12. PC-Relative Conditional Branch Instructions Comparing 2 Registers

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td>Branch on Equal</td>
<td>MIPS I</td>
</tr>
<tr>
<td>BNE</td>
<td>Branch on Not Equal</td>
<td>MIPS I</td>
</tr>
<tr>
<td>BLEZ</td>
<td>Branch on Less Than or Equal to Zero</td>
<td>MIPS I</td>
</tr>
<tr>
<td>BGTZ</td>
<td>Branch on Greater Than Zero</td>
<td>MIPS I</td>
</tr>
<tr>
<td>BEQL</td>
<td>Branch on Equal Likely</td>
<td>MIPS II</td>
</tr>
<tr>
<td>BNEL</td>
<td>Branch on Not Equal Likely</td>
<td>MIPS II</td>
</tr>
<tr>
<td>BLEZL</td>
<td>Branch on Less Than or Equal to Zero Likely</td>
<td>MIPS II</td>
</tr>
<tr>
<td>BGTZL</td>
<td>Branch on Greater Than Zero Likely</td>
<td>MIPS II</td>
</tr>
</tbody>
</table>

Table 3-13. PC-Relative Conditional Branch Instructions Comparing Against Zero

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLTZ</td>
<td>Branch on Less Than Zero</td>
<td>MIPS I</td>
</tr>
<tr>
<td>BGEZ</td>
<td>Branch on Greater Than or Equal to Zero</td>
<td>MIPS I</td>
</tr>
<tr>
<td>BLTZAL</td>
<td>Branch on Less Than Zero and Link</td>
<td>MIPS I</td>
</tr>
<tr>
<td>BGEZAL</td>
<td>Branch on Greater Than or Equal to Zero and Link</td>
<td>MIPS I</td>
</tr>
<tr>
<td>BLTZL</td>
<td>Branch on Less Than Zero Likely</td>
<td>MIPS II</td>
</tr>
<tr>
<td>BGEZL</td>
<td>Branch on Greater Than or Equal to Zero Likely</td>
<td>MIPS II</td>
</tr>
<tr>
<td>BLTZALL</td>
<td>Branch on Less Than Zero and Link Likely</td>
<td>MIPS II</td>
</tr>
<tr>
<td>BGEZALL</td>
<td>Branch on Greater Than or Equal to Zero and Link Likely</td>
<td>MIPS II</td>
</tr>
</tbody>
</table>
3.3.4 Miscellaneous Instructions

3.3.4.1 Exception Instructions

Exception instructions have as their sole purpose causing an exception that will transfer control to a software exception handler in the kernel. System call and breakpoint instructions cause exceptions unconditionally. The trap instructions cause exceptions conditionally based upon the result of a comparison. For detail of these instructions, refer to the individual instruction as described in Appendix A.

Table 3-14. Exception Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>BREAK</td>
<td>Breakpoint</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SYSCALL</td>
<td>System Call</td>
<td>MIPS I</td>
</tr>
<tr>
<td>TGE</td>
<td>Trap if Greater or Equal</td>
<td>MIPS II</td>
</tr>
<tr>
<td>TGEU</td>
<td>Trap if Greater or Equal Unsigned</td>
<td>MIPS II</td>
</tr>
<tr>
<td>TLT</td>
<td>Trap if Less Than</td>
<td>MIPS II</td>
</tr>
<tr>
<td>TLTU</td>
<td>Trap if Less Than Unsigned</td>
<td>MIPS II</td>
</tr>
<tr>
<td>TEQ</td>
<td>Trap if Equal</td>
<td>MIPS II</td>
</tr>
<tr>
<td>TNE</td>
<td>Trap if Not Equal</td>
<td>MIPS II</td>
</tr>
<tr>
<td>TGEI</td>
<td>Trap if Greater or Equal Immediate</td>
<td>MIPS II</td>
</tr>
<tr>
<td>TGEIU</td>
<td>Trap if Greater or Equal Immediate Unsigned</td>
<td>MIPS II</td>
</tr>
<tr>
<td>TLTI</td>
<td>Trap if Less Than Immediate</td>
<td>MIPS II</td>
</tr>
<tr>
<td>TLTIU</td>
<td>Trap if Less Than Immediate Unsigned</td>
<td>MIPS II</td>
</tr>
<tr>
<td>TEQI</td>
<td>Trap if Equal Immediate</td>
<td>MIPS II</td>
</tr>
<tr>
<td>TNEI</td>
<td>Trap if Not Equal Immediate</td>
<td>MIPS II</td>
</tr>
</tbody>
</table>

3.3.4.2 Serialization Instructions

The order in which memory accesses from load and store instructions appear outside the C790 is not specified by the architecture. The SYNC (or SYNC.L) instruction creates a point in the executing instruction stream at which the relative order of some loads and store is known. Loads and stores executed before the SYNC (or SYNC.L) are retired before loads and stores after the SYNC (or SYNC.L) can start.

In order to guarantee the completion of certain instructions a SYNC.P instruction can be used. Instructions executed before a SYNC.P instruction are completed before instructions after the SYNC.P can start. For detail of this instruction refer to SYNC instruction as described in Appendix A.

Table 3-15. Serialization Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNC²</td>
<td>Synchronization</td>
<td>MIPS II</td>
</tr>
</tbody>
</table>

² This includes the SYNC, SYNC.L and SYNC.P instructions.
3.3.4.3 MIPS IV Instructions

The C790 supports a part of the MIPS IV instructions: Conditional Move instructions and Prefetch instruction.

Conditional move operations allow ‘IF’ statements to be represented without branches. ‘THEN’ and ‘ELSE’ clauses are computed unconditionally and the results are placed in a temporary register. Conditional move operations then transfer the temporary results to their true register.

The Prefetch instruction fetches data expected to be used in the near future and places it in the data cache.

For detail of these instructions, refer to the individual instruction as described in Appendix A.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVN</td>
<td>Move Conditional on Not Zero</td>
<td>MIPS IV</td>
</tr>
<tr>
<td>MOVZ</td>
<td>Move Conditional on Zero</td>
<td>MIPS IV</td>
</tr>
<tr>
<td>PREF</td>
<td>Prefetch</td>
<td>MIPS IV</td>
</tr>
</tbody>
</table>
### 3.3.5 System Control Coprocessor (COP0) Instructions

COP0 instructions perform operations specifically on the System Control Coprocessor registers to manipulate the memory management, exception handling, performance monitor, and debug facilities of the processor.

COP0 instructions are enabled if the processor is in Kernel mode, or if bit 28 (CU) is set in the Status register. Otherwise executing one of these instructions generates a Coprocessor Unusable Exception.

For details of COP0 instructions refer to Appendix C.

#### Table 3-17. System Control Coprocessor Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC0F</td>
<td>Branch on Coprocessor 0 False</td>
<td>MIPS I</td>
</tr>
<tr>
<td>BC0T</td>
<td>Branch on Coprocessor 0 True</td>
<td>MIPS I</td>
</tr>
<tr>
<td>BC0TL</td>
<td>Branch on Coprocessor 0 False Likely</td>
<td>MIPS II</td>
</tr>
<tr>
<td>BC0TTL</td>
<td>Branch on Coprocessor 0 True Likely</td>
<td>MIPS II</td>
</tr>
<tr>
<td>CACHE</td>
<td>Cache Operation</td>
<td>R4000</td>
</tr>
<tr>
<td>DI</td>
<td>Disable Interrupt</td>
<td>C790</td>
</tr>
<tr>
<td>EI</td>
<td>Enable Interrupt</td>
<td>C790</td>
</tr>
<tr>
<td>ERET</td>
<td>Exception Return</td>
<td>R4000</td>
</tr>
<tr>
<td>TLBR</td>
<td>Read Indexed TLB Entry</td>
<td>R4000</td>
</tr>
<tr>
<td>TLBWI</td>
<td>Write Index TLB Entry</td>
<td>R4000</td>
</tr>
<tr>
<td>TLBWR</td>
<td>Write Random TLB Entry</td>
<td>R4000</td>
</tr>
<tr>
<td>TLBP</td>
<td>Probe TLB for Matching Entry</td>
<td>R4000</td>
</tr>
<tr>
<td>MTC0</td>
<td>Move To System Control Coprocessor</td>
<td>R4000</td>
</tr>
<tr>
<td>MFC0</td>
<td>Move From System Control Coprocessor</td>
<td>R4000</td>
</tr>
<tr>
<td>MTPC</td>
<td>Move To Performance Counter</td>
<td>C790</td>
</tr>
<tr>
<td>MFPC</td>
<td>Move From Performance Counter</td>
<td>C790</td>
</tr>
<tr>
<td>MTFS</td>
<td>Move To Performance Event Specifier</td>
<td>C790</td>
</tr>
<tr>
<td>MFPS</td>
<td>Move From Performance Event Specifier</td>
<td>C790</td>
</tr>
<tr>
<td>MTBPC</td>
<td>Move To Breakpoint Control Register</td>
<td>C790</td>
</tr>
<tr>
<td>MFBC</td>
<td>Move From Breakpoint Control Register</td>
<td>C790</td>
</tr>
<tr>
<td>MTDAB</td>
<td>Move To Data Address Breakpoint Register</td>
<td>C790</td>
</tr>
<tr>
<td>MFDB</td>
<td>Move From Data Address Breakpoint Register</td>
<td>C790</td>
</tr>
<tr>
<td>MTDABM</td>
<td>Move To Data Address Breakpoint Mask Register</td>
<td>C790</td>
</tr>
<tr>
<td>MFDB</td>
<td>Move From Data Address Breakpoint Mask Register</td>
<td>C790</td>
</tr>
<tr>
<td>MTIAB</td>
<td>Move To Instruction Address Breakpoint Register</td>
<td>C790</td>
</tr>
<tr>
<td>MFIAB</td>
<td>Move From Instruction Address Breakpoint Register</td>
<td>C790</td>
</tr>
<tr>
<td>MTIABM</td>
<td>Move To Instruction Address Breakpoint Mask Register</td>
<td>C790</td>
</tr>
<tr>
<td>MFIABM</td>
<td>Move From Instruction Address Breakpoint Mask Register</td>
<td>C790</td>
</tr>
<tr>
<td>MTDVB</td>
<td>Move To Data Value Breakpoint Register</td>
<td>C790</td>
</tr>
<tr>
<td>MFDB</td>
<td>Move From Data Value Breakpoint Register</td>
<td>C790</td>
</tr>
<tr>
<td>MTDVBM</td>
<td>Move To Data Value Breakpoint Mask Register</td>
<td>C790</td>
</tr>
<tr>
<td>MFDB</td>
<td>Move From Data Value Breakpoint Mask Register</td>
<td>C790</td>
</tr>
</tbody>
</table>
3.3.6 Coprocessor 1 (COP1)

Coprocessor instructions perform operations in their respective coprocessors. Coprocessor loads and stores are I-type, and coprocessor computational instructions have coprocessor-dependent formats. Coprocessor load and store instructions are summarized in 3.3.1.3.

3.3.6.1 Coprocessor 1 (COP1) Instructions

COP1 instructions are enabled if bit 29 (CU) is set in the Status register. Otherwise executing one of these instructions generates a Coprocessor Unusable Exception. For details of COP1 instructions refer to Appendix D.

Table 3-18. Coprocessor 1 Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC1F</td>
<td>Branch on Floating Point False</td>
<td>MIPS I</td>
</tr>
<tr>
<td>BC1T</td>
<td>Branch on Floating Point True</td>
<td>MIPS I</td>
</tr>
<tr>
<td>LWC1</td>
<td>Load Word to Floating Point</td>
<td>MIPS I</td>
</tr>
<tr>
<td>LDC1</td>
<td>Load Doubleword to Floating Point</td>
<td>MIPS II</td>
</tr>
<tr>
<td>SWC1</td>
<td>Store Word from Floating Point</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SDC1</td>
<td>Store Doubleword from Floating Point</td>
<td>MIPS II</td>
</tr>
<tr>
<td>MFC1</td>
<td>Move Word from Floating Point</td>
<td>MIPS I</td>
</tr>
<tr>
<td>MTC1</td>
<td>Move Word to Floating Point</td>
<td>MIPS I</td>
</tr>
<tr>
<td>DMFC1</td>
<td>Move Doubleword from Floating Point</td>
<td>MIPS III</td>
</tr>
<tr>
<td>DMTC1</td>
<td>Move Doubleword to Floating Point</td>
<td>MIPS III</td>
</tr>
<tr>
<td>CFC1</td>
<td>Move Control Word from Floating Point</td>
<td>MIPS I</td>
</tr>
<tr>
<td>CTC1</td>
<td>Move Control Word to Floating Point</td>
<td>MIPS I</td>
</tr>
<tr>
<td>CVT.D.fmt</td>
<td>Floating Point Convert to Double Floating Point</td>
<td>MIPS I, III</td>
</tr>
<tr>
<td>CVT.L.fmt</td>
<td>Floating Point Convert to Long Fixed Point</td>
<td>MIPS III</td>
</tr>
<tr>
<td>CVT.S.fmt</td>
<td>Floating Point Convert to Single Floating Point</td>
<td>MIPS I, III</td>
</tr>
<tr>
<td>CVT.W.fmt</td>
<td>Floating Point Convert to Word Fixed Point</td>
<td>MIPS I</td>
</tr>
<tr>
<td>ADD.fmt</td>
<td>Floating Point Add</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SUB.fmt</td>
<td>Floating Point Subtract</td>
<td>MIPS I</td>
</tr>
<tr>
<td>MUL.fmt</td>
<td>Floating Point Multiply</td>
<td>MIPS I</td>
</tr>
<tr>
<td>DIV.fmt</td>
<td>Floating Point Divide</td>
<td>MIPS I</td>
</tr>
<tr>
<td>ABS.fmt</td>
<td>Floating Point Absolute</td>
<td>MIPS I</td>
</tr>
<tr>
<td>MOV.fmt</td>
<td>Floating Point Move</td>
<td>MIPS I</td>
</tr>
<tr>
<td>NEG.fmt</td>
<td>Floating Point Negate</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SQRT.fmt</td>
<td>Floating Point Square Root</td>
<td>MIPS II</td>
</tr>
<tr>
<td>C.cond.fmt</td>
<td>Floating Point Compare</td>
<td>MIPS I</td>
</tr>
<tr>
<td>CEIL.L.fmt</td>
<td>Floating Point Ceiling Convert to Long Fixed Point</td>
<td>MIPS III</td>
</tr>
<tr>
<td>CEIL.W.fmt</td>
<td>Floating Point Ceiling Convert to Word Fixed Point</td>
<td>MIPS II</td>
</tr>
<tr>
<td>FLOOR.L.fmt</td>
<td>Floating Point Floor Convert to Long Fixed Point</td>
<td>MIPS III</td>
</tr>
<tr>
<td>FLOOR.W.fmt</td>
<td>Floating Point Floor Convert to Word Fixed Point</td>
<td>MIPS II</td>
</tr>
<tr>
<td>ROUND.L.fmt</td>
<td>Floating Point Round to Long Fixed Point</td>
<td>MIPS III</td>
</tr>
<tr>
<td>ROUND.W.fmt</td>
<td>Floating Point Round to Word Fixed Point</td>
<td>MIPS II</td>
</tr>
<tr>
<td>TRUNC.L.fmt</td>
<td>Floating Point Truncate to Long Fixed Point</td>
<td>MIPS III</td>
</tr>
<tr>
<td>TRUNC.W.fmt</td>
<td>Floating Point Truncate to Word Fixed Point</td>
<td>MIPS II</td>
</tr>
</tbody>
</table>
3.3.7 C790-Specific Instructions

The C790 extends its instruction set from the original MIPS architecture. The following instructions are supported:

- Three-operand Multiply and Multiply/Add instructions
- Multiply instructions for Pipeline 1
- Multimedia instructions
- Enable interrupt and Disable interrupt instructions

For more information, refer to Appendices B and C.

3.3.7.1 Integer Multiply / Divide Instructions

The standard MIPS instructions for multiply, divide and move to / from HI / LO registers execute on the I0 pipeline's MAC unit. A complete set of new instructions has also been defined to execute on the I1 pipeline's MAC unit. All of these instructions are shown in the following table.

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Description</th>
<th>OpCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Three Operand Multiply and Multiply-add)</td>
<td>DIV1</td>
<td>Divide 1</td>
<td></td>
</tr>
<tr>
<td>MADD</td>
<td>Multiply/Add</td>
<td>DIVU1</td>
<td>Divide Unsigned 1</td>
</tr>
<tr>
<td>MADDRU</td>
<td>Multiply/Add Unsigned</td>
<td>MADDRD1</td>
<td>Multiply/Add 1</td>
</tr>
<tr>
<td>MULT</td>
<td>Multiply(3-operand)</td>
<td>MADDRU1</td>
<td>Multiply/Add Unsigned 1</td>
</tr>
<tr>
<td>MULTU</td>
<td>Multiply Unsigned(3-operand)</td>
<td>MFHI1</td>
<td>Move From HI 1</td>
</tr>
<tr>
<td>(Multiply Instructions for Pipeline 1)</td>
<td>MFLO1</td>
<td>Move From LO 1</td>
<td></td>
</tr>
<tr>
<td>MULT1</td>
<td>Multiply 1</td>
<td>MTHI1</td>
<td>Move To HI 1</td>
</tr>
<tr>
<td>MULTU1</td>
<td>Multiply Unsigned 1</td>
<td>MTLO1</td>
<td>Move To LO 1</td>
</tr>
</tbody>
</table>

The C790 supports three-operand multiply instructions that store the multiply result to a general purpose register in addition to the LO register. These instructions, as such, don’t have to use the MFLO instruction to move data from the LO register to a general purpose register.

- **MULT rd, rs, rt**  
  \[ HI \|\| LO = rs \times rt \text{ (signed)} \]
  
  \[ rd = \text{new LO contents} \]

- **MULTU rd, rs, rt**  
  \[ HI \|\| LO = rs \times rt \text{ (unsigned)} \]
  
  \[ rd = \text{new LO contents} \]

The C790 also supports new multiply-add instructions, MADD and MADDRU. These instructions execute multiply-accumulate operations using the HI and LO registers as accumulators.

- **MADD rd, rs, rt**  
  \[ HI \|\| LO += rs \times rt \text{ (signed)} \]
  
  \[ rd = \text{new LO contents} \]

- **MADDRU rd, rs, rt**  
  \[ HI \|\| LO += rs \times rt \text{ (unsigned)} \]
  
  \[ rd = \text{new LO contents} \]
3.3.7.2 Multimedia Instructions

The C790 defines a new set of instructions to support multimedia applications. These instructions are shown in Table 3-20. Most of these instructions do parallel operations on data by combining the execution units of the two pipelines (I0 and I1). They form a 128-bit path and then do parallel operations on either two 64-bit data items, four 32-bit data items, eight 16-bit data items, or sixteen 8-bit data items.

In order to support the 128-bit datapath, 128-bit load/store operations are also implemented.

Table 3-20. Multimedia Instructions

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(Arithmetic)</strong></td>
<td></td>
</tr>
<tr>
<td>PADDB</td>
<td>Parallel Add Byte</td>
</tr>
<tr>
<td>PSUBB</td>
<td>Parallel Subtract Byte</td>
</tr>
<tr>
<td>PADDH</td>
<td>Parallel Add Halfword</td>
</tr>
<tr>
<td>PSUBH</td>
<td>Parallel Subtract Halfword</td>
</tr>
<tr>
<td>PADDW</td>
<td>Parallel Add Word</td>
</tr>
<tr>
<td>PSUBW</td>
<td>Parallel Subtract Word</td>
</tr>
<tr>
<td>PADSBH</td>
<td>Parallel Add/Subtract Halfword</td>
</tr>
<tr>
<td>PADDDB</td>
<td>Parallel Add with Signed Saturation Byte</td>
</tr>
<tr>
<td>PSUBSB</td>
<td>Parallel Subtract with Signed Saturation Byte</td>
</tr>
<tr>
<td>PADDSH</td>
<td>Parallel Add with Signed Saturation Halfword</td>
</tr>
<tr>
<td>PSUBSH</td>
<td>Parallel Subtract with Signed Saturation Halfword</td>
</tr>
<tr>
<td>PADDSW</td>
<td>Parallel Add with Signed Saturation Word</td>
</tr>
<tr>
<td>PSUBSW</td>
<td>Parallel Subtract with Signed Saturation Word</td>
</tr>
<tr>
<td>PADDUB</td>
<td>Parallel Add with Unsigned Saturation Byte</td>
</tr>
<tr>
<td>PSUBUB</td>
<td>Parallel Subtract with Unsigned Saturation Byte</td>
</tr>
<tr>
<td>PADDUH</td>
<td>Parallel Add with Unsigned Saturation Halfword</td>
</tr>
<tr>
<td>PSUBUH</td>
<td>Parallel Subtract with Unsigned Saturation Halfword</td>
</tr>
<tr>
<td>PADDUW</td>
<td>Parallel Add with Unsigned Saturation Word</td>
</tr>
<tr>
<td>PSUBUW</td>
<td>Parallel Subtract with Unsigned Saturation Word</td>
</tr>
<tr>
<td><strong>(Min/Max)</strong></td>
<td></td>
</tr>
<tr>
<td>PMAXH</td>
<td>Parallel Maximum Halfword</td>
</tr>
<tr>
<td>PMINH</td>
<td>Parallel Minimum Halfword</td>
</tr>
<tr>
<td>PMAXW</td>
<td>Parallel Maximum Word</td>
</tr>
<tr>
<td>PMINW</td>
<td>Parallel Minimum Word</td>
</tr>
<tr>
<td><strong>(Absolute)</strong></td>
<td></td>
</tr>
<tr>
<td>PABSH</td>
<td>Parallel Absolute Halfword</td>
</tr>
<tr>
<td>PABSW</td>
<td>Parallel Absolute Word</td>
</tr>
<tr>
<td><strong>(Multiply and Divide)</strong></td>
<td></td>
</tr>
<tr>
<td>PMULTW</td>
<td>Parallel Multiply Word</td>
</tr>
<tr>
<td>PMULTUW</td>
<td>Parallel Multiply Unsigned Word</td>
</tr>
<tr>
<td>PDIVW</td>
<td>Parallel Divide Word</td>
</tr>
<tr>
<td>PDIVUW</td>
<td>Parallel Divide Unsigned Word</td>
</tr>
<tr>
<td>PMADDW</td>
<td>Parallel Multiply/Add Word</td>
</tr>
<tr>
<td>PMADDUW</td>
<td>Parallel Multiply/Add Unsigned Word</td>
</tr>
<tr>
<td>PMSUBW</td>
<td>Parallel Multiply/Subtract Word</td>
</tr>
<tr>
<td>PMFHII</td>
<td>Parallel Move From HI</td>
</tr>
<tr>
<td>PMFLO</td>
<td>Parallel Move From LO</td>
</tr>
<tr>
<td>PMTHI</td>
<td>Parallel Move To HI</td>
</tr>
<tr>
<td>PMLTLO</td>
<td>Parallel Move To LO</td>
</tr>
<tr>
<td>PMLTHL</td>
<td>Parallel Move From HI/LO</td>
</tr>
<tr>
<td>PADDW</td>
<td>Parallel Multiply/Add Halfword</td>
</tr>
<tr>
<td>PMSUBH</td>
<td>Parallel Multiply/Subtract Halfword</td>
</tr>
<tr>
<td>PMFHHL</td>
<td>Parallel Move From HI/LO</td>
</tr>
<tr>
<td>PHMADH</td>
<td>Parallel Horizontal Multiply/Add Halfword</td>
</tr>
<tr>
<td>PHMSBH</td>
<td>Parallel Horizontal Multiply/Subtract Halfword</td>
</tr>
<tr>
<td>PDIVBW</td>
<td>Parallel Divide Broadcast Word</td>
</tr>
<tr>
<td>OpCode</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>MFSA</td>
<td>Move from SA Register</td>
</tr>
<tr>
<td>MTSA</td>
<td>Move to SA Register</td>
</tr>
<tr>
<td>MTSAB</td>
<td>Move Byte Count to SA Register</td>
</tr>
<tr>
<td>MTSAH</td>
<td>Move Halfword Count to SA Register</td>
</tr>
<tr>
<td>(Shift)</td>
<td></td>
</tr>
<tr>
<td>PSLLH</td>
<td>Parallel Shift Left Logical Halfword</td>
</tr>
<tr>
<td>PSRLH</td>
<td>Parallel Shift Right Logical Halfword</td>
</tr>
<tr>
<td>PSRAH</td>
<td>Parallel Shift Right Arithmetic Halfword</td>
</tr>
<tr>
<td>PSLLW</td>
<td>Parallel Shift Left Logical Word</td>
</tr>
<tr>
<td>PSRLW</td>
<td>Parallel Shift Right Logical Word</td>
</tr>
<tr>
<td>PSRAW</td>
<td>Parallel Shift Right Arithmetic Word</td>
</tr>
<tr>
<td>PSLLVW</td>
<td>Parallel Shift Left Logical Variable Word</td>
</tr>
<tr>
<td>PSRLVW</td>
<td>Parallel Shift Right Logical Variable Word</td>
</tr>
<tr>
<td>PSRAWV</td>
<td>Parallel Shift Right Arithmetic Variable Word</td>
</tr>
<tr>
<td>(Logical)</td>
<td></td>
</tr>
<tr>
<td>PAND</td>
<td>Parallel AND</td>
</tr>
<tr>
<td>POR</td>
<td>Parallel OR</td>
</tr>
<tr>
<td>PXOR</td>
<td>Parallel XOR</td>
</tr>
<tr>
<td>PNOR</td>
<td>Parallel NOR</td>
</tr>
<tr>
<td>(Compare)</td>
<td></td>
</tr>
<tr>
<td>PCGTB</td>
<td>Parallel Compare for Greater Than Byte</td>
</tr>
<tr>
<td>PCEQB</td>
<td>Parallel Compare for Equal Byte</td>
</tr>
<tr>
<td>PCGTH</td>
<td>Parallel Compare for Greater Than Halfword</td>
</tr>
<tr>
<td>PCEQH</td>
<td>Parallel Compare for Equal Halfword</td>
</tr>
<tr>
<td>PCGTW</td>
<td>Parallel Compare for Greater Than Word</td>
</tr>
<tr>
<td>PCEQW</td>
<td>Parallel Compare for Equal Word</td>
</tr>
<tr>
<td>(Quadword Load Store)</td>
<td></td>
</tr>
<tr>
<td>LQ</td>
<td>Load Quadword</td>
</tr>
<tr>
<td>SQ</td>
<td>Store Quadword</td>
</tr>
<tr>
<td>(Pack/Extend)</td>
<td></td>
</tr>
<tr>
<td>PPACB</td>
<td>Parallel Pack To Byte</td>
</tr>
<tr>
<td>PPACH</td>
<td>Parallel Pack To Halfword</td>
</tr>
<tr>
<td>PINTEH</td>
<td>Parallel Interleave Even Halfword</td>
</tr>
<tr>
<td>PPACW</td>
<td>Parallel Pack To Word</td>
</tr>
<tr>
<td>PEXTUB</td>
<td>Parallel Extend Upper From Byte</td>
</tr>
<tr>
<td>PEXTLB</td>
<td>Parallel Extend Lower From Byte</td>
</tr>
<tr>
<td>PEXTUH</td>
<td>Parallel Extend Upper From Halfword</td>
</tr>
<tr>
<td>PEXTULH</td>
<td>Parallel Extend Lower From Halfword</td>
</tr>
<tr>
<td>PEXTUW</td>
<td>Parallel Extend Upper From Word</td>
</tr>
<tr>
<td>PEXTLW</td>
<td>Parallel Extend Lower From Word</td>
</tr>
<tr>
<td>PEXT5</td>
<td>Parallel Extend from 5 bits</td>
</tr>
<tr>
<td>PPAC5</td>
<td>Parallel Pack to 5 bits</td>
</tr>
<tr>
<td>(Others)</td>
<td></td>
</tr>
<tr>
<td>PCPYH</td>
<td>Parallel Copy Halfword</td>
</tr>
<tr>
<td>PCPYLD</td>
<td>Parallel Copy Lower Doubleword</td>
</tr>
<tr>
<td>PCPYUD</td>
<td>Parallel Copy Upper Doubleword</td>
</tr>
<tr>
<td>PREVH</td>
<td>Parallel Reverse Halfword</td>
</tr>
<tr>
<td>PINTH</td>
<td>Parallel Interleave Halfword</td>
</tr>
<tr>
<td>PEXEH</td>
<td>Parallel Exchange Even Halfword</td>
</tr>
<tr>
<td>PEXCH</td>
<td>Parallel Exchange Center Halfword</td>
</tr>
<tr>
<td>PEXEW</td>
<td>Parallel Exchange Even Word</td>
</tr>
<tr>
<td>PEXCW</td>
<td>Parallel Exchange Center Word</td>
</tr>
<tr>
<td>PROT3W</td>
<td>Parallel Rotate 3 word</td>
</tr>
<tr>
<td>QFSRV</td>
<td>Quadword Funnel Shift Right Variable</td>
</tr>
<tr>
<td>PLZCW</td>
<td>Parallel Leading Zero Count Word</td>
</tr>
</tbody>
</table>
### 3.4 User Instruction Latency and Repeat Rate

Table 3-21 shows the latencies and repeat rates for all user instructions executed in I0, I1, BR, LS and C1 execution pipelines. Kernel instructions are not included, nor are instructions not issued to these execution pipelines. See Figure 2-1 and Figure 2-4 for execution pipeline name.

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Execution</th>
<th>Latency</th>
<th>Repeat Rate</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Instructions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add/Sub/Logical/Set</td>
<td>I0/I1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MF/MT/HI/LO</td>
<td>I0/I1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Shift/LUI</td>
<td>I0/I1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Branch/Jump</td>
<td>BR</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Conditional Move</td>
<td>I0/I1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MULT/MULTU</td>
<td>I0</td>
<td>4</td>
<td>2</td>
<td>Latency relative to Lo/Hi/GPR</td>
</tr>
<tr>
<td>MULT1/MULTU1</td>
<td>I1</td>
<td>4</td>
<td>2</td>
<td>Latency relative to Lo1/Hi1/GPR</td>
</tr>
<tr>
<td>DIV/DIVU1</td>
<td>I0</td>
<td>37</td>
<td>37</td>
<td>Latency relative to Lo/Hi</td>
</tr>
<tr>
<td>DIV1/DIVU1</td>
<td>I1</td>
<td>37</td>
<td>37</td>
<td>Latency relative to Lo1/Hi1</td>
</tr>
<tr>
<td>MADD/MADDU</td>
<td>I0</td>
<td>4</td>
<td>2</td>
<td>Latency relative to Lo/Hi/GPR</td>
</tr>
<tr>
<td>MADD1/MADDU1</td>
<td>I1</td>
<td>4</td>
<td>2</td>
<td>Latency relative to Lo1/Hi1/GPR</td>
</tr>
<tr>
<td>Load</td>
<td>LS</td>
<td>1</td>
<td>1</td>
<td>Assuming cache hit</td>
</tr>
<tr>
<td>Store</td>
<td>LS</td>
<td></td>
<td>1</td>
<td>Assuming cache hit</td>
</tr>
<tr>
<td>Multimedia Multiply</td>
<td>I0+I1</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Multimedia Multiply/Add</td>
<td>I0+I1</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Multimedia Divide</td>
<td>I0+I1</td>
<td>37</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>Floating-Point Instructions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.S/SUB.S/C.cond.S</td>
<td>C1</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ADD.D/SUB.D/C.cond.D</td>
<td>C1</td>
<td>8</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ABS/NEG/MOV</td>
<td>C1</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CVT</td>
<td>C1</td>
<td>8</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MUL.S</td>
<td>C1</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MUL.D</td>
<td>C1</td>
<td>8</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>DIV.S</td>
<td>C1</td>
<td>21</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>DIV.D</td>
<td>C1</td>
<td>35</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>SQRT.S</td>
<td>C1</td>
<td>21</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>SQRT.D</td>
<td>C1</td>
<td>35</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>MFC1/MTC1</td>
<td>C1+LS</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>DMFC1/DMTC1</td>
<td>C1+LS</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CFC1/CTC1</td>
<td>C1+LS</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>LWC1/LDC1</td>
<td>C1+LS</td>
<td>2</td>
<td>1</td>
<td>Assuming cache hit</td>
</tr>
<tr>
<td>SWC1/SDC1</td>
<td>C1+LS</td>
<td>–</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
4. CPU and COP0 Registers

This chapter describes the CPU registers and the System Control Coprocessor (COP0) registers.

The CPU registers group consists of:
- General Purpose Registers (GPRs),
- Multiply and Divide registers (HI and LO registers) that hold the results of integer multiply and divide,
- The SA register which is used by the funnel shift instructions,
- The Program Counter (PC) register.

The COP0 registers control the processor state and report its status. These registers can be read using the MFC0 instruction and written using the MTC0 instruction.
4.1 CPU Registers

The central processing unit (CPU) provides the following registers:

- **32 128-bit General Purpose Registers (GPR)**
- **Four registers that hold the results of integer multiply and divide operations** (HI0, LO0, HI1, and LO1)
- **Shift Amount (SA) register**
- **Program Counter**

The C790 has 128-bit-wide General Purpose Registers (GPRs). The upper 64 bits of the GPRs are only used by the C790-specific “Quad Load/Store”, and “Multimedia (Parallel)” instructions.

HI0 and LO0 are the standard 64-bit HI and LO registers. HI1 and LO1, which are the upper 64 bits of the 128-bit HI and LO registers, are only used by the new multiply and divide instructions, such as MULT1, MULTU1, DIV1, DIVU1, MADD1, MADDU1, MFHI1, MFLO1, MTHI1, and MTLO1. All these instructions are equivalent to existing instructions which operate on HI0 and LO0 registers.

The Shift Amount (SA) register specifies the shift amount used by the funnel shift instruction. The shaded registers in Figure 4-1 are new architecturally-visible registers that are specific to the C790.
# General Purpose Registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>$0</td>
</tr>
<tr>
<td>$1</td>
<td>$1</td>
</tr>
<tr>
<td>$2</td>
<td>$2</td>
</tr>
<tr>
<td>$31</td>
<td>$31</td>
</tr>
</tbody>
</table>

## HI and LO Register

<table>
<thead>
<tr>
<th>HI</th>
<th>HI1</th>
<th>HI (HI0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LO</td>
<td>LO1</td>
<td>LO (LO0)</td>
</tr>
</tbody>
</table>

## SA Register

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>SA</td>
</tr>
</tbody>
</table>

## Program Counter

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PC</td>
</tr>
</tbody>
</table>

Figure 4-1. CPU Registers
4.1.1 General Purpose Registers

The standard 64-bit CPU general purpose registers have been extended to 128-bit registers. New instructions have been defined to use the upper 64-bits of these registers.

Two of the CPU general purpose registers have special assigned functions:

- \( r0 \) is hardwired to a value of zero, and can be used as the target register for any instruction whose result is to be discarded. \( r0 \) can also be used as a source when a zero value is needed.
- \( r31 \) is the link register used by the Jump and Link instructions. In general, it should not be used by other instructions.

4.1.2 HI and LO Registers

The standard 64-bit \( HI \) and \( LO \) registers have been extended to 128-bit registers. New instructions have been defined to use the upper 64-bits of these registers. \( HI0 \) and \( LO0 \) are the standard 64-bit \( HI \) and \( LO \) registers. \( HI1 \) and \( LO1 \) are the upper 64 bits of the 128-bit \( HI \) and \( LO \) registers.

These four registers (\( HI0, LO0, HI1, LO1 \)) store:

- the product of integer multiply operations, or
- the accumulation of integer multiply-accumulate operations, or
- the quotient (in \( LO0 \) or \( LO1 \)) and remainder (in \( HI0 \) or \( HI1 \)) of integer divide operations.

4.1.3 Shift Amount (SA) Register

The SA register specifies the shift amount used by the funnel shift instruction. This is a new architecturally-visible register and it needs to be saved and restored as part of the processor state. New instructions have been defined to move values between this register and the general purpose registers.

4.1.4 Program Counter (PC)

The Program Counter (PC) holds the address of the instruction which is being executed. The PC is incremented automatically by 4 when a non-control-transfer instruction (that is: branch, jump, ERET, SYSCALL, or TRAP) is executed. Control-transfer instructions change the value of the PC to the target address specified by them. An exception also changes the contents of the PC to the specified exception vector address.
4.2 System Control Coprocessor (COP0) Registers

*COP0* registers are listed in Table 4-1.

Table 4-1. Coprocessor 0 Registers

<table>
<thead>
<tr>
<th>Register No.</th>
<th>Register Name</th>
<th>Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Index</td>
<td>Programmable register to select TLB entry for reading or writing</td>
<td>MMU</td>
</tr>
<tr>
<td>1</td>
<td>Random</td>
<td>Pseudo-random counter for TLB replacement</td>
<td>MMU</td>
</tr>
<tr>
<td>2</td>
<td>EntryLo0</td>
<td>Low half of TLB entry for even PFN (Physical page number)</td>
<td>MMU</td>
</tr>
<tr>
<td>3</td>
<td>EntryLo1</td>
<td>Low half of TLB entry for odd PFN (Physical page number)</td>
<td>MMU</td>
</tr>
<tr>
<td>4</td>
<td>Context</td>
<td>Pointer to kernel virtual PTE table in 32-bit addressing mode</td>
<td>Exception</td>
</tr>
<tr>
<td>5</td>
<td>PageMask</td>
<td>Mask that sets the TLB page size</td>
<td>MMU</td>
</tr>
<tr>
<td>6</td>
<td>Wired</td>
<td>Number of wired TLB entries</td>
<td>MMU</td>
</tr>
<tr>
<td>7</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>8</td>
<td>BadVAddr</td>
<td>Bad virtual address</td>
<td>Exception</td>
</tr>
<tr>
<td>9</td>
<td>Count</td>
<td>Timer compare</td>
<td>Exception</td>
</tr>
<tr>
<td>10</td>
<td>EntryHi</td>
<td>High half of TLB entry (Virtual page number and ASID)</td>
<td>MMU</td>
</tr>
<tr>
<td>11</td>
<td>Compare</td>
<td>Timer compare</td>
<td>Exception</td>
</tr>
<tr>
<td>12</td>
<td>Status</td>
<td>Processor Status Register</td>
<td>Exception</td>
</tr>
<tr>
<td>13</td>
<td>Cause</td>
<td>Cause of the last exception taken</td>
<td>Exception</td>
</tr>
<tr>
<td>14</td>
<td>EPC</td>
<td>Exception Program Counter</td>
<td>Exception</td>
</tr>
<tr>
<td>15</td>
<td>PRId</td>
<td>Processor Revision Identifier</td>
<td>MMU</td>
</tr>
<tr>
<td>16</td>
<td>Config</td>
<td>Configuration Register</td>
<td>MMU</td>
</tr>
<tr>
<td>17</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>18</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>19</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>20</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>21</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>22</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>23</td>
<td>BadPAddr</td>
<td>Bad physical address</td>
<td>Exception</td>
</tr>
<tr>
<td>24</td>
<td>Debug</td>
<td>This is used for Debug function</td>
<td>Debug</td>
</tr>
<tr>
<td>25</td>
<td>Perf</td>
<td>Performance Counter and Control Register</td>
<td>Exception</td>
</tr>
<tr>
<td>26</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>27</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>28</td>
<td>TagLo</td>
<td>Cache Tag register (low bits)</td>
<td>Cache</td>
</tr>
<tr>
<td>29</td>
<td>TagHi</td>
<td>Cache Tag register (high bits)</td>
<td>Cache</td>
</tr>
<tr>
<td>30</td>
<td>ErrorEPC</td>
<td>Error Exception Program Counter</td>
<td>Exception</td>
</tr>
<tr>
<td>31</td>
<td>(Reserved)</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
4.2.1 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 records 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.

Table 4-2 shows the format of the Index register; Table 4-2 describes the Index register fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>31</td>
<td>Probe failure. Set to 1 when the previous TLB Probe (TLBP) instruction was unsuccessful.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>Index</td>
<td>5:0</td>
<td>Index to the TLB entry affected by the TLB Read and TLB Write instructions.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>0</td>
<td>30:6</td>
<td>Reserved. Must be written as zeroes, and returns zeroes when read.</td>
<td>Read-only</td>
<td>0</td>
</tr>
</tbody>
</table>
4.2.2 Random Register (1)

Figure 4-3. Random Register

The Random register is a read-only register. The least significant six bits index an entry in the TLB. This register decrements every cycle an instruction is executed. Its value ranges between an upper and a lower bound, as follows:

- A lower bound is set by the number of TLB entries reserved for exclusive use by the operating system (the contents of the Wired register).
- An upper bound is set by the total number of TLB entries (47 maximum).

The Random register specifies the entry in the TLB that is affected by the TLB Write Random (TLBWR) 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 4-3 shows the format of the Random Register; Table 4-3 describes the Random Register fields.

Table 4-3. Random Register Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>5:0</td>
<td>TLB Random index.</td>
<td>Read-only</td>
<td>Upper bound (47)</td>
</tr>
<tr>
<td>0</td>
<td>31:6</td>
<td>Reserved. Must be written as zeros, and returns zeroes when read.</td>
<td>Read-only</td>
<td>0</td>
</tr>
</tbody>
</table>
4.2.3 EntryLo0 Register (2), and EntryLo1 Register (3)

The EntryLo0 and EntryLo1 registers consist of two registers that have similar format:
- **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 4-4 shows the format of the EntryLo0 and EntryLo1 Registers; Table 4-4 describes the EntryLo0 and EntryLo1 Register fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFN</td>
<td>25:6</td>
<td>Page frame number; the upper bits of the physical address.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
| C     | 5:3  | Specifies the TLB page coherency attribute.  
000(0): Reserved  
001(1): Reserved  
010(2): Uncached  
011(3): Cacheable, write-back, write allocate  
100(4): Reserved  
101(5): Reserved  
110(6): Reserved  
111(7): Uncached Accelerated | Read/Write | Undefined |
| D     | 2    | **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. | Read/Write | Undefined |
| V     | 1    | **Valid**. If this bit is set, it indicates that the TLB entry is valid; otherwise, a TLBL or TLBS miss will occur. | Read/Write | Undefined |
| G     | 0    | **Global**. If this bit is set in both EntryLo0 and EntryLo1, then the processor ignores the ASID during TLB look-up. | Read/Write | Undefined |
| 0     | 31:26 | Reserved. Must be written as zeroes, and returns zeroes when read.  
EntryLo0[31] is reserved for Kernel use. It contains the written value. This bit has no effect on any CPU or TLB operation. | Read-only | 0 |

Reserved codes in C field may not be written correctly into TLB entry by TLBWI or TLBWR instruction.
4.2.4 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 4-5 shows the format of the Context register; Table 4-5 describes the Context register fields.

Table 4-5. Context Register Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTEBase</td>
<td>31:23</td>
<td>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.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>BadVPN2</td>
<td>22:4</td>
<td>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.</td>
<td>Read-only</td>
<td>Undefined</td>
</tr>
<tr>
<td>0</td>
<td>3:0</td>
<td>Reserved. Must be written as zeros, and returns zeroes when read.</td>
<td>Read-only</td>
<td>0</td>
</tr>
</tbody>
</table>

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 KB 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.
### 4.2.5 PageMask Register (5)

The PageMask register is a read/write register used for reading or writing the TLB. It holds a comparison mask that sets the variable page size for each TLB entry, as shown in Table 4-6.

#### Table 4-6. PageMask Register Field

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
</table>
| MASK  | 24:13| Page comparison mask.  
0000 0000 0000: Page Size = 4 Kbytes  
0000 0000 0011: Page Size = 16 Kbytes  
0000 0000 1111: Page Size = 64 Kbytes  
0000 0011 1111: Page Size = 256 Kbytes  
0000 1111 1111: Page Size = 1 Mbyte  
0011 1111 1111: Page Size = 4 Mbytes  
1111 1111 1111: Page Size = 16 Mbytes | Read/Write | Undefined |
| 0     | 31:25, 12:0 | Reserved. Must be written as zeros, and returns zeroes when read. | Read-only | 0 |

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 4-6, the operation of the TLB is undefined.
4.2.6 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 4-8. Wired entries are fixed, non-replaceable entries which cannot be overwritten by a TLB write operation. Random entries can be overwritten. Figure 4-7 shows the format of the Wired register. Table 4-7 describes the register fields.

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 as shown in Figure 4-8.

Writing a value greater than 47 into this register produces undefined results.

Table 4-7. Wired Register Field Descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wired</td>
<td>5:0</td>
<td>TLB Wired boundary (the number of wired TLB entries)</td>
<td>Read/Write</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>31:6</td>
<td>Reserved. Must be written as zeros, and returns zeroes when read.</td>
<td>Read-only</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4-7. Wired Register

Figure 4-8. Wired Register Boundary
### 4.2.7 BadVAddr Register (8)

Figure 4-9. BadVAddr Register

The Bad Virtual Address register (BadVAddr) is a read-only register that displays the most recent virtual address that caused one of the following exceptions: TLB Invalid, TLB Modified, TLB Refill, or Address Error exceptions.

Figure 4-9 shows the format of the BadVAddr register; Table 4-8 describes the register fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BadVAddr</td>
<td>31:0</td>
<td>The most recent virtual address that cause a TLB Invalid, TLB Modified, TLB Refill, or Address Error exception.</td>
<td>Read-only</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

Note: The BadVAddr register does not save any information for bus errors, since bus errors are not addressing errors.
4.2.8 Count Register (9)

The Count register acts as a real-time timer. It is incremented every CPU clock cycle. The timer interrupt signaled through IP[7] can be disabled through the interrupt mask bit, IM[7]. This register can be read or written.

Figure 4-10 shows the format of the Count register. Table 4-9 describes the register fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>31:0</td>
<td>32-bit timer, incrementing at the CPU clock rate.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
4.2.9 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.

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.

Figure 4-11 shows the format of the EntryHi register. Table 4-10 describes the register fields.

Table 4-10. EntryHi Register Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN2</td>
<td>31:13</td>
<td>Virtual page number divided by two (maps to two pages).</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>ASID</td>
<td>7:0</td>
<td>Address space ID field. An 8-bit field that lets multiple processes share the TLB; each process can have a distinct mapping of otherwise identical virtual page numbers.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>0</td>
<td>12:8</td>
<td>Reserved. Must be written as zeroes, and returns zeroes when read.</td>
<td>Read-only</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4-11. EntryHi Register
4.2.10 Compare Register (11)

![Figure 4-12. Compare Register](image)

The Compare register acts as a timer (see also the Count register); it maintains a stable value that does not change on its own. When the value of the Count register equals the value of the Compare register, interrupt bit IP[7] in the Cause register is set. This causes an interrupt as soon as the interrupt is enabled. Writing a value to the Compare register, as a side effect, clears the timer interrupt.

For diagnostic purposes, the Compare register is a read/write register. In normal use, however, the Compare register is write-only. Figure 4-12 shows the format of the Compare register. Table 4-11 describes the register fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>31:0</td>
<td>The Compare register saves a stable value compared to the Count register. When the value of the Count register equals to the value of the Compare register, interrupt IP[7] occurs.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
4.2.11 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. Figure 4-13 shows the format of the Status register. The following paragraphs identify the more important Status register fields and describe the fields. Some of the important fields include:

- **The 3-bit Interrupt Mask (IM) field controls the enabling of three interrupt signals. Interrupts must be enabled before they can be asserted. Interrupts are recognized by the processor when the corresponding bits are set in both the Interrupt Mask and the Interrupt Enable fields of the Status register and the Interrupt Pending field of the Cause register. The C790 does not support software interrupts. IM[7] corresponds to the internal timer interrupt and IM[3:2] corresponds to Int[1:0] signals.**

- **The 4-bit Coprocessor Usability (CU) field (CU[3:0]) controls the usability of four possible coprocessors. Regardless of the CU[0] bit setting, COP0 is always usable in Kernel mode. For all other cases, an access to an unusable coprocessor causes an exception. C790 supports coprocessor 1 (FPU).**
### 4.2.11.1 Status Register Format

Table 4-12 describes the Status register fields. All bits in the Status register are readable and writable.

#### Table 4-12. Status Register Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU</td>
<td>31:28</td>
<td>Controls the usability of each of the four coprocessor unit numbers. COP0 is always usable when in Kernel mode, regardless of the setting of the CU[0] bit.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 → usable</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 → unusable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FR</td>
<td>26</td>
<td>Enable additional floating point registers</td>
<td>Read/Write</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 → 16 registers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 → 32 registers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEV</td>
<td>23</td>
<td>Controls the location of Performance counter and debug/SIO exception vectors.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 → normal</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 → bootstrap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEV</td>
<td>22</td>
<td>Controls the location of TLB refill and general exception vectors.</td>
<td>Read/Write</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 → normal</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 → bootstrap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CH</td>
<td>18</td>
<td>Cache Hit (tag match and valid state) or Miss indication for last CACHE Hit Invalidate and CACHE Hit Write-back Invalidate for the Data cache.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 → miss</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 → hit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDI</td>
<td>17</td>
<td>EI/DI instruction Enable: When this bit is set, the EI and DI instructions can operate in User, Supervisor and Kernel modes and as such set or clear the EIE bit to enable or disable all interrupts (except NMI). When this bit is cleared, EI and DI operate as NOPs in User and Supervisor modes and executes properly in Kernel mode.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>EIE</td>
<td>16</td>
<td>Enable IE: This bit enables or disables the IE (Interrupt Enable) bit. This bit is cleared by the DI instruction and set by the EI instruction.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 → disables all interrupts regardless of the value of the IE bit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 → enables the IE bit. (All interrupts are enabled if IE=1, EXL=0, and ERL=0.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IM[7,3:2]</td>
<td>15, 11:10</td>
<td>Interrupt Mask: controls the enabling of each of the external and internal 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.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 → disabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 → enabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note: The enabling of this bit is valid only when EIE=1, IE=1, EXL=0 and ERL=0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEM</td>
<td>12</td>
<td>Bus Error Mask: controls the updating of the BadPAddr register and signaling a bus error exception.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 → update BadPAddr and signal a bus error exception.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 → do not update BadPAddr and stop signaling a bus error exception. This bit is set to 1 when it is a 0 and a bus error is signaled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KSU</td>
<td>4:3</td>
<td>Kernel/Supervisor/User Mode bits:</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00 → Kernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 → Supervisor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 → User</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 → Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Field Bits Description Type Initial Value

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERL</td>
<td>2</td>
<td><strong>Error Level</strong>: set by the processor when Reset, NMI, performance counter, SIO or debug exception is taken. 0 → normal 1 → error</td>
<td>Read/Write</td>
<td>1</td>
</tr>
<tr>
<td>EXL</td>
<td>1</td>
<td><strong>Exception Level</strong>: set by the processor when any exception other than Reset, NMI, performance counter, or debug exception is taken. 0 → normal 1 → exception</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>IE</td>
<td>0</td>
<td><strong>Interrupt Enable</strong> 0 → disables all interrupts 1 → enables all interrupts (if EIE=1, ERL=0, and EXL=0)</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>0</td>
<td>27, 25:24, 21:19, 14:13, 9:5</td>
<td>Reserved. Must be written as zeroes, and returns zeroes when read.</td>
<td>Read-only</td>
<td>0</td>
</tr>
</tbody>
</table>

### 4.2.11.2 Status Register Modes and Access States

Fields of the Status register set the modes and access states below.

**Interrupt Enable**: Interrupts are enabled when all of the following conditions are true:
- Status.IE = 1,
- and Status.EIE = 1,
- and Status.EXL = 0,
- and Status.ERL = 0

If these conditions are met, setting the IM bits enable the appropriate interrupts.

**SIO Enable**: A level 2 exception by SIO is enabled when the following condition is true:
- Status.ERL = 0

If this condition is met, asserting the SIO signal causes a Debug exception to occur.

**Operating Modes**: The following CPU Status register bit settings are required for User, Kernel, and Supervisor modes.
- **The Processor is in User mode** when KSU = 10 and EXL = 0 and ERL = 0.
- **The processor is in Supervisor mode** when KSU = 01 and EXL = 0 and ERL = 0.
- **The processor is in Kernel mode** when KSU = 00 or EXL = 1 or ERL = 1.

**Kernel Address Space Accesses**: Access to the kernel address space is allowed when the processor is in Kernel mode.

**Supervisor Address Space Accesses**: Access to the supervisor address space is allowed when the processor is in Kernel mode or Supervisor mode, as described above.

**User Address Space Accesses**: Access to the user address space is allowed in Kernel, Supervisor, and User modes.
4.2.12 Cause Register (13)

The 32-bit read-only *Cause* register describes the cause of the most recent exception. Figure 4-14 shows the fields of this register. Table 4-13 describes the *Cause* register fields. All bits in the *Cause* register are read-only.

Table 4-13. Cause Register Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
</table>
| BD    | 31   | Set by the processor when any exception other than Reset, NMI, performance counter, or debug occurs and is taken in a branch delay slot.  
1 → delay slot  
0 → normal | Read-only | Undefined |
| BD2   | 30   | Indicates whether the last NMI, performance counter, debug, or SIO exception taken occurred in a branch delay slot.  
1 → delay slot  
0 → normal | Read-only | Undefined |
| CE    | 29:28| Coprocessor unit number referenced when a Coprocessor Unusable exception is taken. | Read-only | Undefined |
| EXC2  | 18:16| Indicates the exception codes for level 2 exceptions (Performance Counter, Reset, Debug, SIO and NMI exceptions)  
000 (0) : Res (Reset)  
001 (1) : NMI (Non-maskable Interrupt)  
010 (2) : PerfC (Performance Counter)  
011 (3) : Dbg (Debug) and SIO (SIO)  
1xx (4-7) : Reserved | Read-only | Undefined |
| IP[7:3:2] | 15, 11:10 | Indicates an interrupt is pending.  
1 → interrupt pending  
0 → no interrupt | Read-only | Undefined, Int[1:0] |
| SIOP  | 12   | Indicates an SIO signal is pending  
1 → SIO signal is pending  
0 → no SIO signal is pending | Read-only | SIO |
<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExcCode</td>
<td>6:2</td>
<td>Exception code filed.</td>
<td>Read-only</td>
<td>Undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00000  (0) : Int (Interrupt)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00001  (1) : Mod (TLB modification exception)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00010  (2) : TLBL (TLB exception (load or instruction fetch))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00011  (3) : TLBS (TLB exception (store))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00100  (4) : AdEL (Address error exception (load or instruction fetch))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00101  (5) : AdES (Address error exception (store))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00110  (6) : IBE (Bus error exception (instruction fetch))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00111  (7) : DBE (Bus error exception (data reference: load or store))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01000  (8) : Sys (Syscall exception)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01001  (9) : Bp (Breakpoint exception)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01010  (10): RI (Reserved instruction exception)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01011  (11): CpU (Coprocessor Unusable exception)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01100  (12): Ov (Arithmetic overflow exception)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01101  (13): Tr (Trap exception)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01110  (14): Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01111  (15): FPE (Floating-Point exception)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(16-31): (Reserved)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>27:19, 14:13, 9:7, 1:0</td>
<td>Reserved. Must be written as zeroes, and returns zeroes when read.</td>
<td>Read-only</td>
<td>0</td>
</tr>
</tbody>
</table>
4.2.13 EPC Register (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 BD bit in the Cause register is set).

On the occurrence of an exception, if the EXL bit in the Status register is set to a 1, the processor does not update the EPC register. Figure 4-15 shows the format of the EPC register. Table 4-14 describes the EPC register fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPC</td>
<td>31:0</td>
<td>Contains the address at which processing can resume after an exception has been serviced.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
4.2.14 PRId Register (15)

The 32-bit read-only Processor Revision Identifier (PRId) register contains information identifying the implementation and revision level of the C790 and COP0. Figure 4-16 shows the format of the PRId register; Table 4-15 describes the PRId register fields.

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 C790 processor is 0x38. 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 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, but 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, these values are not listed and software should not rely on the revision number in the PRId register to characterize the chip.

Table 4-15. PRId Register Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imp</td>
<td>15:8</td>
<td>Implementation number</td>
<td>Read-only</td>
<td>0x38</td>
</tr>
<tr>
<td>Rev</td>
<td>7:0</td>
<td>Revision number of each mask</td>
<td>Read-only</td>
<td>Revision number</td>
</tr>
<tr>
<td>0</td>
<td>31:16</td>
<td>Reserved. Must be written as zeroes, and returns zeroes when read.</td>
<td>Read-only</td>
<td></td>
</tr>
</tbody>
</table>
4.2.15 Config Register (16)

The Config register specifies various configuration options which can be selected. Figure 4-17 shows the format of the Config register; Table 4-16 describes the Config register fields.

Some configuration options, as defined by Config bits 30:28, 15 and 11:6, are set by the hardware during reset and are included in the Config register as read-only status bits for the software to access. Other configuration options like 18:16 and 13:12 are set by hardware during reset and can be modified by software. Other configuration options like bits 2:0 are read/write and controlled by software; on reset these fields are undefined.

Table 4-16. Config Register Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC</td>
<td>30:28</td>
<td>Bus clock ratio.</td>
<td>Read-only</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000: processor clock frequency divided by 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>001 ~ 111: (Reserved)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIE</td>
<td>18</td>
<td>Double issue enable</td>
<td>Read/Write</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 → Single issue 1 → Double issue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICE</td>
<td>17</td>
<td>Setting this bit to 1 enables the instruction cache.</td>
<td>Read/Write</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 → Instruction cache disable 1 → Instruction cache enable</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The CACHE instruction for the instruction cache is enabled regardless of the value of this bit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCE</td>
<td>16</td>
<td>Setting this bit to 1 enables the data cache.</td>
<td>Read/Write</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 → Data cache disable 1 → Data cache enable</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the cache is disabled, the PREF instruction becomes a NOP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BE</td>
<td>15</td>
<td>Big Edian</td>
<td>Read-only</td>
<td>Pin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 → Little Edian 1 → Big Edian</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NBE</td>
<td>13</td>
<td>Setting this bit to 1 enables non-blocking load.</td>
<td>Read/Write</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 → Disable Non-blocking load and hit under miss 1 → Enable Non-blocking load and hit under miss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BPE</td>
<td>12</td>
<td>Setting this bit to 1 enables branch prediction.</td>
<td>Read/Write</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 → Disable Branch Prediction 1 → Enable Branch Prediction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IC</td>
<td>11:9</td>
<td>Instruction cache Size (Instruction cache size = $2^{IC}$ bytes).</td>
<td>Read-only</td>
<td>011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>011 → 32 KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC</td>
<td>8:6</td>
<td>Data cache Size (Data cache size = $2^{DC}$ bytes).</td>
<td>Read-only</td>
<td>011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>011 → 32 KB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4-17. Config Register Format
<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0</td>
<td>2:0</td>
<td>kseg0 coherency algorithm. 000: Reserved 001: Reserved 010: Uncached 011: Cacheable, write-back, write allocate 100: Reserved 101: Reserved 110: Reserved 111: Uncached Accelerated</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>0</td>
<td>31, 27:19, 14, 5:3</td>
<td>Reserved. Must be written as zeroes, and returns zeroes when read.</td>
<td>Read-only</td>
<td>0</td>
</tr>
</tbody>
</table>

With single issue enabled (DIE = 0), the C790 always fetches two instructions but only issues a single instruction.
4.2.16 BadPAddr Register (23)

The Bad Physical Address register (BadPAddr) is a read-only register that contains the most recent physical address that caused a bus error. It is updated with a new value whenever Status.BEM is clear (0). Once this bit is set (on the occurrence of a bus error) the register holds the value.

Figure 4-18 shows BadPAddr register format; Table 4-17 describes the register fields.

**Table 4-17. BadPAddr Register Fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BdPAddr</td>
<td>31:4</td>
<td>Physical Address value</td>
<td>Read-Only</td>
<td>undefined</td>
</tr>
<tr>
<td>0</td>
<td>3:0</td>
<td>Reserved. Returns zeros when read.</td>
<td>Read-Only</td>
<td>0</td>
</tr>
</tbody>
</table>
4.2.17 Debug Registers (24)

There are seven separately addressable debug registers, which are all assigned to CP0, register 24.

Each of the seven registers is accessed by specifying subaccess code which is bit2 to bit0 of an instruction code.

Breakpoint Control Register (BPC) (subaccess code 0)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>I</td>
<td>D</td>
<td>B</td>
<td>D</td>
<td>D</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>R</td>
<td>W</td>
<td>V</td>
<td>0</td>
<td>U</td>
<td>S</td>
<td>K</td>
<td>E</td>
<td>0</td>
<td>U</td>
<td>S</td>
<td>K</td>
<td>X</td>
<td>T</td>
<td>T</td>
<td>E</td>
<td>0</td>
<td>W</td>
<td>R</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>D</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

See Table 13-3 for a detailed description of individual BPC register fields.
Instruction Address Breakpoint (IAB) (subaccess code 2)

```
  31  21  0
     IAB  0
```

Instruction Address Breakpoint Mask Register (IABM) (subaccess code 3)

```
  31  21  0
     IABM  0
```

Data Address Breakpoint Register (DAB) (subaccess code 4)

```
  31  0
     DAB
```

Data Address Breakpoint Mask Register (DABM) (subaccess code 5)

```
  31  0
     DABM
```

Data value Breakpoint Register (DVB) (subaccess code 6)

```
  31  0
     DVB
```

Data value Breakpoint Mask Register (DVBM) (subaccess code 7)

```
  31  0
     DVBM
```
4.2.18 Performance Counter Registers (25)

There are three separately addressable performance counter registers, which are all assigned to COP0, register 25.
Each of the three registers is accessed by specifying subaccess code which is bit1 to bit0 of an instruction code.
All performance counter registers are read/write registers.

Performance Counter Control Register (PCCR)

![PCCR Diagram]

Performance Counter Register 0 (PCR0)

![PCR0 Diagram]

Performance Counter Register 1 (PCR1)

![PCR1 Diagram]

Figure 4-19. Performance Counter Registers
Table 4-18 lists the field definitions for the Performance Counter Control register.

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTE</td>
<td>31</td>
<td>Enables event counting (CTR1, CTR0) and exception generation: 0 → Disable 1 → Enable</td>
<td>Read/Write</td>
<td>0</td>
</tr>
<tr>
<td>EVENT1</td>
<td>19:15</td>
<td>Set the event to be monitored by PCR1 00000 (0) Low-order branch issued 00001 (1) Processor cycle 00010 (2) Dual instruction issue 00011 (3) Branch miss predicted 00100 (4) TLB miss 00101 (5) DTLB miss 00110 (6) Data Cache miss 00111 (7) WBB single request unavailable 01000 (8) WBB burst request unavailable 01001 (9) WBB burst request almost full 01010 (10) WBB burst request full 01011 (11) CPU data bus busy 01100 (12) Instruction completed 01101 (13) Non-BDS instruction completed 01110 (14) COP1 instruction completed 01111 (15) Store completed 10000 (16) No event (17-31) Reserved</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>EVENT0</td>
<td>9:5</td>
<td>Set the event to be monitored by PCR0 00000 (0) Reserved 00001 (1) Processor cycle 00010 (2) Single instruction issue 00011 (3) Branch issue 00100 (4) BTAC miss 00101 (5) ITLB miss 00110 (6) Instruction Cache miss 00111 (7) DTLB accessed 01000 (8) Non-blocking load 01001 (9) WBB single request 01010 (10) WBB burst request 01011 (11) CPU address bus busy 01100 (12) Instruction completed 01101 (13) Non-BDS instruction completed 01110 (14) Reserved 01111 (15) Load completed 10000 (16) No event (17-31) Reserved</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>U1, U0</td>
<td>14, 4</td>
<td>Enables event counting (PCR1/PCR0) in the User mode. 0 → Disable 1 → Enable</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>S1, S0</td>
<td>13, 3</td>
<td>Enables event counting (PCR1/PCR0) in the Supervisor mode. 0 → Disable 1 → Enable</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>K1, K0</td>
<td>12, 2</td>
<td>Enables event counting (PCR1/PCR0) in the Kernel mode. 0 → Disable 1 → Enable</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>EXL1, EXL0</td>
<td>11, 1</td>
<td>Enables event counting (PCR1/PCR0) when EXL bit is set in the Status register. 0 → Disable 1 → Enable</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>0</td>
<td>30:20, 10, 0</td>
<td>Reserved. Must be written as zero, and returns zero when read.</td>
<td>Read-only</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 4-19 lists the field definitions for the *Performance Counter register 0 (PCR0)*.

**Table 4-19. Performance Counter Register 0 Fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVFL</td>
<td>31</td>
<td>Overflow flag</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>VALUE</td>
<td>30:0</td>
<td>The actual counter</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

Table 4-20 lists the field definitions for the *Performance Counter register 1 (PCR1)*.

**Table 4-20. Performance Counter Register 1 Fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVFL</td>
<td>31</td>
<td>Overflow flag</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>VALUE</td>
<td>30:0</td>
<td>The actual counter</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
### 4.2.19 TagLo (28) and TagHi (29) Registers

**TagLo**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>PTagLo</td>
</tr>
<tr>
<td>20</td>
<td>Special use</td>
</tr>
<tr>
<td>12</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>V</td>
</tr>
<tr>
<td>7</td>
<td>R</td>
</tr>
<tr>
<td>6</td>
<td>L</td>
</tr>
<tr>
<td>5</td>
<td>Su</td>
</tr>
</tbody>
</table>

**TagHi**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Special use</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

The TagLo and TagHi registers are 32-bit read/write registers used by the CACHE instruction. For writing to the data cache tags, the TagLo register contains the fields as shown above and the TagHi register is not used. For writing to the data cache data portion the TagLo register contains the data value. For writing to the instruction cache tags the TagLo register contains the fields as defined above except that bits three and six are also reserved bits. For writing to the instruction cache data portion, the TagLo register contains the data (instruction) and the TagHi register contains the steering bits and bits for the BHT as defined in Chapter 7. When reading from the caches, the values in the TagLo and TagHi register are the same as described above for writing. These registers are also used for manipulating the BTAC. See the description of the CACHE instruction in Appendix C for details. Figure 4-20 shows the format of these registers for some of the cache operations.
Table 4-21 lists the field definitions of the TagLo register.

### Table 4-21. TagLo Register Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTagLo [31:12]</td>
<td>31:12</td>
<td>PTagLo[31:12] specifies 20-bit physical address tag cache.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>Dirty: 0 → Clean 1 → Dirty</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>V</td>
<td>5</td>
<td>Valid: 0 → Invalid 1 → Valid</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>R</td>
<td>4</td>
<td>LRF Replacement: This bit participates in the calculation determining which cache way will be used for the next replacement. See Section 7.3.1 for details.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>L</td>
<td>3</td>
<td>Lock: This bit is only used for the data cache. For instruction cache operations this bit is treated as a reserved bit. 0 → For this line, this side is not locked. 1 → For this line, this side is locked.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>Special use, Su</td>
<td>11:7, 2:0</td>
<td>Used by the CACHE instruction to manipulate the branch target address cache. Refer to Chapter 7 for details.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

### Table 4-22. TagHi Register Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special use</td>
<td>31:0</td>
<td>The TagHi register is used by the CACHE instruction to manipulate some of the bits of the instruction cache. Refer to Chapter 7 for details.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
4.2.20 ErrorEPC (30)

The *ErrorEPC* register is similar to the *EPC* register, except that *ErrorEPC* is used on nonmaskable interrupt (NMI), debug, SIO, and performance counter exceptions.

The read/write *ErrorEPC* register contains the virtual address at which instruction processing can resume after servicing an error. This address can be:

- the virtual address of the instruction that caused the exception
- the virtual address of the immediately preceding branch or jump instruction (when the instruction is in a branch delay slot, and the BD2 bit in the Cause register is set).

Table 4-23 lists the field definition of the *ErrorEPC* register.

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ErrorEPC</td>
<td>31:0</td>
<td>Contains the virtual address at which instruction processing can resume after servicing an error.</td>
<td>Read/Write</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
5. Exception Processing and Reset

This chapter describes the exception processing, including level 1 and level 2 exceptions.
5.1 The Exception Handling Process

Exceptions can be recognized while the program is any of its three operating modes: User, Supervisor, or Kernel.

Exceptions are categorized into 2 groups which are level 1 exceptions and level 2 exceptions as shown in Table 5-1.

<table>
<thead>
<tr>
<th>Level 1 Exceptions</th>
<th>Level 2 Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>Reset</td>
</tr>
<tr>
<td>TLB Modified</td>
<td>NMI</td>
</tr>
<tr>
<td>TLB Refill</td>
<td>Performance Counter</td>
</tr>
<tr>
<td>TLB Invalid</td>
<td>Debug</td>
</tr>
<tr>
<td>Address Error</td>
<td>SIO</td>
</tr>
<tr>
<td>Syscall</td>
<td></td>
</tr>
<tr>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>Trap</td>
<td></td>
</tr>
<tr>
<td>Reserved Instruction</td>
<td></td>
</tr>
<tr>
<td>Coprocessor Usable</td>
<td></td>
</tr>
<tr>
<td>Integer Overflow</td>
<td></td>
</tr>
<tr>
<td>Bus Error</td>
<td></td>
</tr>
<tr>
<td>Floating Point Exception</td>
<td></td>
</tr>
</tbody>
</table>

Compatibility Note: Level 2 exceptions are a generalization of “error level” exception processing defined in earlier MIPS implementation.

5.1.1 Level 1 Exceptions

Exception Processing

When the processor takes a level 1 exception, the processor switches to Kernel mode. Rather than set the Status.KSU bits to effect the switch, the Status.EXL bit is set to 1. Whenever Status.EXL is 1, the operating mode is Kernel mode, regardless of the setting of Status.KSU.

Then the processor saves the virtual address of the instruction canceled by the exception. This address is saved in the EPC register. If the canceled instruction is in the delay slot of a branch instruction, the Cause.BD bit is set to 1 and EPC is set to the address of the branch instruction (rather than the delay slot). For non-delay-slot instructions, Cause.BD is set to 0. If Status.EXL bit was 1 before the exception is taken, EPC and Cause.BD aren’t set. The exception service routine examines Cause.BD to determine the true address of the instruction that raised the exception.

In addition to setting EPC, Cause.BD, and Status.EXL, the 5 bit field Cause.ExcCode is also set. This field specifies the cause of the exception; The Cause.CE fields may also get set when an Coprocessor unusable exception is raised.

After setting those bits, the processor jumps to the exception vector address.
The basic exception handling operation performed can be described using the Figure 5-1 Level 1 Exception Processing Flowchart.

(see next page)

**Disabled exceptions in level 1 exception handler**

Once a level 1 exception service routine is entered, interrupts and bus error are unconditionally disabled.

*C790 Programming Note:* The only level 1 exception that is unconditionally disabled within level 1 exceptions handler is external interrupts and bus errors. All other level 1 exceptions still occur and are recognized (if enabled). A software system that makes use of such exceptions must use extreme care. In particular, it must make sure that it has saved *EPC* and *Cause.BD* somewhere (e.g. in a software managed stack) before the exception occurs.
Set \textit{Cause.ExcCode}

\textit{Cause.CE} ← coprocessor number when CPU exception

Set \textit{BadVAddr} when AdES, AdEL or any TLB exception

Set \textit{Context} and \textit{EntryHi} when any TLB exception

Set \textit{BadPAddr} when Bus Error

\begin{itemize}
  \item \textbf{Status.EXL} ← 1
  \item \textbf{Instr.in Br.Dly.Slot ?}
    \begin{itemize}
      \item Yes
        \begin{itemize}
          \item \textbf{EPC} ← PC-4
          \item \textbf{Cause.BD} ← 1
          \item \textbf{Status.EXL} ← 1
        \end{itemize}
      \item No
        \begin{itemize}
          \item \textbf{EPC} ← PC
          \item \textbf{Cause.BD} ← 0
          \item \textbf{Status.EXL} ← 1
        \end{itemize}
    \end{itemize}
  \item = TLB Refill
  \item = Interrupt
  \item = Others
\end{itemize}

\begin{itemize}
  \item \textbf{Offset} ← 0x0
  \item \textbf{Offset} ← 0x180
  \item \textbf{Offset} ← 0x200
  \item \textbf{Offset} ← 0x180
\end{itemize}

\begin{itemize}
  \item = 0 (normal)
  \item = 1 (bootstrap)
\end{itemize}

\textbf{PC} ← 0x8000 0000+\textbf{Offset}

\textbf{PC} ← 0xBFC0 0200+\textbf{Offset}

\textbf{Figure 5-1. Level 1 Exception processing flowchart}
5.1.2 Level 2 Exceptions

Exception Processing

When the processor takes a level 2 exception, the processor switches to kernel mode, by setting Status.ERL to 1.

The address of the instruction where the Level 2 exception was recognized is stored in the ErrorEPC register. If the canceled instruction is in the delay slot of a branch instruction, the Cause.BD2 bit is set to 1 and ErrorEPC is set to the address of the branch instruction (rather than the delay slot). For non-delay-slot instructions, Cause.BD2 is set to 0. In addition, the cause of the exception is stored in Cause.EXC2.

After setting those bits, the processor jumps to the exception vector address.

The basic Level 2 exception handling operation performed can be described using the Figure 5-2 Level 2 Exception processing Flowchart.

(see next page)

Disabled Exceptions in level 2 exceptions

When executing a Level 2 exception service routine, following exceptions are disabled.

- NMI, Interrupt, and Bus error
- Debug, SIO and Performance counter

_C790 Implementation Note:_ Any external exception that is not level-sensitive (e.g. NMI) must be held until it is recognized; i.e. at least until the Level 2 handler is exited.

_C790 Programming Note:_ It is the programmer’s responsibility to ensure that all other internal exceptions (e.g. OVERFLOW) never occur within a Level 2 handler. If they do occur, the corresponding Level 1 exception handler will be entered. Since both Status.EXL and Status.ERL will be set when servicing this (nested) exception, the ERET used to exit the service routine will operate incorrectly.

_C790 Programming Note:_ When Status.ERL = 1, the user address, Kuseg, region becomes a $2^{31}$-byte unmapped, uncached address space (that is, mapped directly to physical address 0x0000 0000-0x7FFF FFFF).
Figure 5-2. Level 2 Exception processing flowchart
5.2 Exception Vector Locations

Exception vector addresses for level 1 exceptions are shown in Table 5-2. The vector address for TLB refill depends on the \textit{Status.EXL} bit. The vector addresses for level 1 exceptions also depend on the \textit{Status.BEV} bit.

Table 5-2. Exception Vectors for Level 1 exceptions

<table>
<thead>
<tr>
<th>Exceptions</th>
<th>BEV = 0</th>
<th>BEV = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB Refill (EXL = 0)</td>
<td>0x8000 0000</td>
<td>0xBFC0 0200</td>
</tr>
<tr>
<td>TLB Refill (EXL = 1)</td>
<td>0x8000 0180</td>
<td>0xBFC0 0380</td>
</tr>
<tr>
<td>Interrupt</td>
<td>0x8000 0200</td>
<td>0xBFC0 0400</td>
</tr>
<tr>
<td>Others</td>
<td>0x8000 0180</td>
<td>0xBFC0 0380</td>
</tr>
</tbody>
</table>

Exception vector addresses for level 2 exceptions are shown in Table 5-3. The vector addresses for level 2 exceptions also depend on the \textit{Status.DEV} bit.

Table 5-3. Exception Vectors for Level 2 exceptions

<table>
<thead>
<tr>
<th>Exceptions</th>
<th>Vector Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset, NMI</td>
<td>0xBFC0 0000</td>
</tr>
<tr>
<td>Performance Counter</td>
<td>0x8000 0080</td>
</tr>
<tr>
<td>Debug, SIO</td>
<td>0x8000 0100</td>
</tr>
</tbody>
</table>
5.3 Cause Register Setting

The Cause.ExcCode bits are set when a level 1 exception is taken. The Cause.ExcCode setting is shown in Table 5-4.

Table 5-4. Cause.ExcCode Field

<table>
<thead>
<tr>
<th>ExcCode</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Int (Interrupt)</td>
</tr>
<tr>
<td>1</td>
<td>Mod (TLB modification exception)</td>
</tr>
<tr>
<td>2</td>
<td>TLBL (TLB exception; load or inst fetch)</td>
</tr>
<tr>
<td>3</td>
<td>TLBS (TLB exception; store)</td>
</tr>
<tr>
<td>4</td>
<td>AdEL (Address error exception; load or inst fetch)</td>
</tr>
<tr>
<td>5</td>
<td>AdES (Address error exception; store)</td>
</tr>
<tr>
<td>6</td>
<td>IBE (Bus error exception; instruction fetch)</td>
</tr>
<tr>
<td>7</td>
<td>DBE (Bus error exception; load or store)</td>
</tr>
<tr>
<td>8</td>
<td>Sys (Syscall exception)</td>
</tr>
<tr>
<td>9</td>
<td>Bp (Breakpoint exception)</td>
</tr>
<tr>
<td>10</td>
<td>RI (Reserved instruction exception)</td>
</tr>
<tr>
<td>11</td>
<td>CpU (Coprocessor Unusable exection)</td>
</tr>
<tr>
<td>12</td>
<td>Ov (Integer Overflow exception)</td>
</tr>
<tr>
<td>13</td>
<td>Tr (Trap exception)</td>
</tr>
<tr>
<td>14</td>
<td>Reserved</td>
</tr>
<tr>
<td>15</td>
<td>FPE (Floating Point Exception)</td>
</tr>
<tr>
<td>16-31</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The Cause.EXC2 bits are set when a level 2 exception is taken. The Cause.EXC2 setting is shown in Table 5-5.

Table 5-5. Cause.EXC2 Field

<table>
<thead>
<tr>
<th>EXC2</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Res (Reset exception)</td>
</tr>
<tr>
<td>1</td>
<td>NMI (Non-Maskable Interrupt)</td>
</tr>
<tr>
<td>2</td>
<td>PerfC (Performance Counter exception)</td>
</tr>
<tr>
<td>3</td>
<td>Dbg (Debug exception), SIO (SIO exception)</td>
</tr>
<tr>
<td>4</td>
<td>SS (Single Step)</td>
</tr>
<tr>
<td>5-7</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
5.4 Masking an exception

The following exceptions can be masked by setting bits in Status register.

NMI, Performance counter, Debug, Bus error, Interrupt and SIO

The Table 5-6 shows whether the bits mask those exceptions. Exceptions which marked with “X” can be masked by setting (BEM, EXL or ERL) or clearing (IE or IM) the corresponding bit in the Status register.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Mask bit (in Status register)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IE</td>
</tr>
<tr>
<td>Reset</td>
<td></td>
</tr>
<tr>
<td>NMI</td>
<td></td>
</tr>
<tr>
<td>Performance Counter</td>
<td></td>
</tr>
<tr>
<td>Debug</td>
<td></td>
</tr>
<tr>
<td>SIO</td>
<td></td>
</tr>
<tr>
<td>Address error</td>
<td></td>
</tr>
<tr>
<td>TLB Refill/Invalid/Modify</td>
<td></td>
</tr>
<tr>
<td>Bus error</td>
<td></td>
</tr>
<tr>
<td>Syscall</td>
<td></td>
</tr>
<tr>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>Reserved instruction</td>
<td></td>
</tr>
<tr>
<td>Coprocessor Unusable</td>
<td></td>
</tr>
<tr>
<td>Interrupt</td>
<td></td>
</tr>
<tr>
<td>Integer overflow</td>
<td></td>
</tr>
<tr>
<td>Trap</td>
<td></td>
</tr>
</tbody>
</table>
5.5 Detailed Description

5.5.1 Exception Priority

Exception priority rules determine which exception is taken first, if multiple exceptions occur on the same instruction. The Table 5-7. Shows the priority order of the exceptions.

Table 5-7. Exception Priority Order

<table>
<thead>
<tr>
<th>Reset (highest priority)</th>
<th>NMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Counter</td>
<td></td>
</tr>
<tr>
<td>Instruction Breakpoint (debug)</td>
<td></td>
</tr>
<tr>
<td>Address error - Instruction fetch</td>
<td></td>
</tr>
<tr>
<td>TLB refill - Instruction fetch</td>
<td></td>
</tr>
<tr>
<td>TLB invalid - Instruction fetch</td>
<td></td>
</tr>
<tr>
<td>Bus Error - Instruction fetch</td>
<td></td>
</tr>
<tr>
<td>Single Step</td>
<td></td>
</tr>
<tr>
<td>syscall, BREAK, Reserved Instruction,*</td>
<td></td>
</tr>
<tr>
<td>Floating Point Exception or Coprocessor Unusable*</td>
<td></td>
</tr>
<tr>
<td>Interrupt</td>
<td></td>
</tr>
<tr>
<td>Data address/value breakpoint (debug)</td>
<td></td>
</tr>
<tr>
<td>SIO</td>
<td></td>
</tr>
<tr>
<td>Integer overflow, Trap</td>
<td></td>
</tr>
<tr>
<td>Address error - data access</td>
<td></td>
</tr>
<tr>
<td>TLB refill - data access</td>
<td></td>
</tr>
<tr>
<td>TLB invalid - data access</td>
<td></td>
</tr>
<tr>
<td>TLB modified - data access</td>
<td></td>
</tr>
<tr>
<td>Bus error - data access (lowest priority)</td>
<td></td>
</tr>
</tbody>
</table>

* The exception priority between Reserved Instruction exception(RI) and Coprocessor Unusable exception(CpU)

The exception priorities of the two exceptions are the same. However, when Status.CU[1] = 0, an attempt to execute any FPU (COP1) instruction causes a CpU exception. When Status.CU[1] = 1, the attempt is reported as an FPE(E):unimplemented FPU exception in the Cop1 sub-instructions.

On the other hand, an attempt to execute any COP0 class Reserved Instruction causes an RI exception regardless Status.CU[0].
5.5.2 Reset Exception

Cause

The RESET exception occurs when the Reset signal is asserted and then deasserted. This exception is not maskable.

Exception Level: 2

Vector Address: 0xBFC00000

Processing

The RESET exception vector is located within uncached and unmapped address space. Hence the cache and TLB need not be initialized in order to process the exception.

The contents of all registers in the CPU are undefined when this exception is recognized, except for the following register fields:

- In the Status register,
  Status.ERL and Status.BEV are set to 1.
  Status.BEM is set to 0.
  All other bits except for 0-fixed bits are undefined.
- In the Cause register,
  Cause.EXC2 is set to 0 (to indicate that a Reset occurred)
  All other bits except for 0-fixed bits are undefined.
- In the Config register,
  DIE, ICE, DCE, NBE, and BPE bits are set to 0.
  All other bits except for fixed-value, read-only bits are undefined.
- The Random register is initialized to the value of its upper bound (47).
- The Wired register is initialized to 0.
- The Counter Enable flag in the Performance Counter Control register (PCCR.CTE) is set to 0.
- The breakpoint address enable flags in the Breakpoint Control register, BPC.IAE, BPC.DRE, and BPC.DWE, are all set to 0.
- Valid, Dirty, LRF, and Lock bits of the data cache and the Valid and LRF bits of the instruction cache are initialized to 0 on reset.

Servicing

The RESET exception is serviced by:

- initializing all processor registers, coprocessor registers, caches, and the memory system
- performing diagnostic tests
- bootstrapping the operating system
5.5.3 Non-Maskable Interrupt (NMI) Exception

Cause

The Non-Maskable Interrupt (NMI) exception occurs in response to the falling edge of the \textit{NMI*} signal. The NMI exception is maskable by setting the \textit{Status.ERL} bit. It is recognized regardless of the settings of the \textit{Status.EXL}, and \textit{Status.IE} bits.

Exception Level: 2

Vector Address: 0xBFC00000

Processing

NMI and RESET exceptions share the same exception vector. This vector is located within uncached and unmapped address space; therefore, the cache and TLB need not be initialized in order to process the exception.

When the NMI exception is recognized, all register contents are preserved with the following exceptions:

- \textit{ErrorEPC} register, which contains the restart PC, and \textit{Cause.BD2} which records whether the NMI was recognized in a branch delay slot.
- \textit{Status.ERL} and \textit{Status.BEV} flags are both set to 1.
- \textit{Cause.EXC2} is set to 1 (NMI).

Servicing

Note that the NMI service routine entry address does not depend on the \textit{Status.BEV} flag. In fact, the \textit{Status.BEV} bit is unconditionally set to 1 before the NMI handler is entered. It is up to the NMI service routine to restore the setting of the \textit{Status.BEV} bit prior to exit.
5.5.4 Performance Counter Exception

Cause

A lower-case performance counter exception occurs when a Performance counter overflows and conditions are met as described in Section 9.3.2. This exception is maskable by setting Status.ERL bit.

Exception Level: 2

Vector Address: 0x8000 0080 (DEV = 0), 0xBFC0 0280 (DEV = 1)

Processing

The value of Cause.EXC2 is set to 2 (PerfC). The ErrorEPC register contains the address of the instruction where the Performance counter exception was detected unless it is in a branch delay slot, in which case the ErrorEPC register contains the address of the preceding branch instruction and the Cause.BD2 is set.

Servicing

When this exception is recognized, control is transferred to the applicable service routine.
5.5.5 Debug Exception

Cause

A DEBUG exception occurs whenever hardware breakpoint conditions as described in Chapter 13 are detected. This exception is maskable by setting Status.ERL bit.

Exception Level: 2

Vector Address: 0x8000 0100 (DEV = 0), 0xBFC0 0300 (DEV = 1)

Processing

The value of Cause.EXC2 is set to 3 (Dbg). The ErrorEPC register contains the address of the instruction where the debug exception was detected unless it is in a branch delay slot, in which case the ErrorEPC register contains the address of the preceding branch instruction and Cause.BD2 is set. Note that the Load data value breakpoint exception is imprecise. That is, the instruction where the breakpoint is detected is not the load instruction that triggers the breakpoint; see Chapter 13 for more details.

Servicing

When this exception is recognized, control is transferred to the applicable service routine.
5.5.6 Address Error Exception

Cause

The Address Error exception occurs when an attempt is made to execute one of the following:

- 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
- reference the kernel address space from User or Supervisor mode
- reference the supervisor address space from User mode

This exception is not maskable.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of Cause.ExcCode is set to 4 (AdEL) or 5 (AdES), depending on whether the exception was caused due to an instruction reference (AdEL), load operation (AdEL), or store operation (AdES).

When this exception is recognized, the virtual address that was not properly aligned or that referenced protected address space is stored in the BadVAddr register. This update occurs even if the exception occurs within a level 1 or level 2 exception handler. The contents of the VPN field of the Context 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 Cause.BD is set to indicate that the branch delay slot instruction actually caused the exception.
5.5.7 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.

**Exception Level:** 1

**Vector Address:**
- EXL = 0: 0x8000 0000 (BEV = 0), 0xBFC0 0200 (BEV = 1)
- EXL = 1: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

**Processing**

The value of Cause.ExcCode is set to either a value of 2 (TLBL) or 3 (TLBS). This code indicates whether the exception was caused due to an instruction reference, load operation, or store operation.

When this exception is recognized, the BadVAddr, Context and EntryHi registers are updated to hold the virtual address that failed address translation. The EntryHi register also contains the ASID for which the translation fault occurred. These actions take place even if the exception is recognized within a level 1 or level 2 exception handler. 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 Cause.BD is set.

The EPC register and BD bit in the Cause register point to the address of the instruction causing the exception.

**Servicing**

To service this exception, the contents of the Context 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 EntryLo0/EntryLo1 register; the EntryHi and EntryLo registers are then 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.
5.5.8 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.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of Cause.ExcCode is set to either 2 (TLBL) or 3 (TLBS). This code indicates whether the exception was caused due to an instruction reference, load operation, or store operation.

When this exception is recognized, the BadVAddr, Context, and EntryHi registers are loaded with the virtual address that failed address translation. The EntryHi register also contains the ASID for which the translation fault occurred. These actions occur even if the exception is recognized within a level 1 or level 2 exception handler. The Random register normally contains a valid location in which to put the replacement TLB entry. The contents of the EntryLo register is 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.
5.5.9 TLB Modified Exception

Cause

The TLB modified exception occurs when a store operation generates a virtual address that matches a TLB entry that is marked valid but is not dirty and therefore is not writable. This exception is not maskable.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of \texttt{Cause.ExcCode} is set to 1 (Mod) and the \texttt{BadVAddr}, \texttt{Context}, and \texttt{EntryHi} registers contain the virtual address that failed address translation. The \texttt{EntryHi} register also contains the ASID for which the translation fault occurred. These actions occur even if the exception is recognized within a level 1 or level 2 exception handler. The contents of the \texttt{EntryLo} register is undefined.

The \texttt{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 \texttt{EPC} register contains the address of the preceding branch instruction and the \texttt{BD} bit of the \texttt{Cause} register is set.

Servidng

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 \texttt{TLBP} instruction places the index of the TLB entry that must be altered into the \texttt{Index} register. The \texttt{EntryLo} register is loaded with a word containing the physical page frame and access control bits (with the \texttt{D} bit set), and the \texttt{EntryHi} and \texttt{EntryLo} registers are written into the TLB.
5.5.10 Bus Error Exception

Cause

A Bus Error exception is raised when BUSERR* signal is asserted during bus transactions. This exception is masked when Status.BEM, Status.EXL or Status.ERL are set to 1.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of Cause.ExcCode is set to 6 (IBE) or 7 (DBE), indicating whether the exception was caused due to an instruction reference (IBE), load operation (DBE), or store operation (DBE). The BadPAddr is set to the physical address which caused a bus error when Status.BEM bit is 0.

The EPC register and BD bit in the Cause register point to the address of the instruction currently being executed by the processor.

Note that there is no necessary relationship between a bus error and the instruction being executed currently. For example, a bus error may be caused by instruction prefetch, or by a data cache line operation that is unrelated to any instruction. Furthermore, it could be caused by a load or store that was issued several instructions prior to the instruction that was executing when the bus error was recognized.

If a bus error is caused by a load or store instruction, the instruction is retired. If the instruction is a store, the nature of how memory is updated depends on the memory subsystem’s design. If the instruction is a load, the value loaded into the destination register is indeterminate. If a data value breakpoint is pending for the memory address accessed, breakpoint recognition is implementation dependent.

Servicing

In the C790 the bus error exception is imprecise and as such difficult to recover from and continue processing. If a bus error occurs during instruction or data cache refills, the cache line loaded has undefined values in it. Since it is not possible in general to determine the offending address (from the EPC) the entire data and instruction cache contents should be invalidated by using Index Invalidate suboperation of the CACHE instruction. (See the CACHE instruction’s definition for details on how to do this.)
5.5.11 System Call Exception

**Cause**

A SYSCALL exception occurs as a result of executing the SYSCALL instruction. This exception is not maskable.

**Exception Level:** 1

**Vector Address:** 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

**Processing**

The value of Cause.ExcCode is set to 8 (Sys). 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 and Cause.BD is set.

**Servicing**

When this exception is recognized, control is transferred to the applicable system routine.

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.
5.5.12 BREAK Instruction Exception

Cause

A BREAK exception occurs as a result of executing the BREAK instruction. This exception is not maskable.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of Cause.ExcCode is set to 9 (Bp). 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 and Cause.BD is set.

Servicing

When a BREAK exception is recognized, control is transferred to the applicable system routine. Additional distinctions can be made by analyzing the unused bits 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.
5.5.13 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 a MMI instruction with an undefined minor opcode (bits 10:0)
- an attempt is made to execute a COPz instruction with an undefined minor opcode (bits 25:21)

Note: In the C790, 64-bit operations are always valid in User, Supervisor, and Kernel mode.

This exception is not maskable.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of Cause.ExcCode is set to 10 (RI). 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.
5.5.14 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 that has not been marked usable via the Status.Cu[ ] bits or
- COP0 instructions, when the unit has been marked not usable and the process executes in either User or Supervisor mode.

NOTE: COP0 instructions always execute in Kernel mode, regardless of the setting of Status.CU[0]. Also note that the operation of the COP0 instructions EI and DI is not controlled by Status.CU[0]. Instead, the Status.EDI bit specifies whether the EI and DI instructions execute in User and Supervisor modes. In case execution is suppressed, EI and DI behave as no-operations in User and Supervisor modes; they do not signal an exception.

The exception is not maskable.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of Cause.ExcCode is set to 11 (Cpu) and the field Cause.CE (Coprocessor Usage Error) is set to 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 CE (Coprocessor Usage Error) field, which result 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.
5.5.15 Interrupt Exception

Cause

The Interrupt exception occurs when one of the three interrupt signals is asserted. The significance of the interrupts is dependent upon the specific system implementation.

Each of the three interrupts can be masked by clearing the corresponding bit in the Int-Mask field of the Status register, and all of the three interrupts can be masked at once by clearing the IE bit or EIE bit of the Status register.

All three interrupts are also masked at once when the EXL or ERL bit of the Status register is set to 1.

Interrupt IP[7] is set when the Count register is equal to the Compare register.

Exception Level: 1

Vector Address: 0x8000 0200 (BEV = 0), 0xBFC0 0400 (BEV = 1)

Processing

The value of Cause.ExcCode is set to 0 (Int). 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.

Servicing

If the interrupt is hardware-generated, the interrupt condition is cleared by correcting the condition causing the interrupt pin to be asserted.

Due to the on-chip write buffer, a store to an external device (possibly clearing the interrupt) may not occur until after other instructions in the pipeline finish. Hence, the user must ensure that the store will occur before the return from exception instruction (ERET) is executed. This can be insured by executing a SYNC instruction. Otherwise the interrupt may be serviced again even though there is no actual interrupt pending.
5.5.16 SIO Exception

**Cause**

The SIO exception occurs when the `SIOInt` signal is asserted. This exception is maskable by setting `Status.ERL` bit.

**Exception Level:** 2

**Vector Address:** 0x8000 0100 (DEV = 0), 0xBFC0 0300 (DEV = 1)

**Processing**

The value of `Cause.EXC2` is set to 3(DBG). The `Cause.SIO` is set to 1. The `ErrorEPC` register contains the address of the instruction where the SIO exception was detected unless if is in a branch delay slot, in which case the `ErrorEPC` register contains the address of the preceding branch instruction and `Cause.BD2` is set.

**Servicing**

When this exception is recognized, control is transferred to the applicable service routine.
5.5.17 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.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of Cause.ExcCode is set to 12 (Ov). 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.
5.5.18 Trap Exception

Cause

The TRAP exception occurs when a TGE, TGEU, TLT, TLTU, TEQ, TNE, TGEI, TGEIU, TLTI, TLTIU, TEQI, or TNEI instruction results in a TRUE condition. This exception is not maskable.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of Cause.ExcCode is set to 13 (Tr). 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 Cause.BD is set.
5.5.19 Floating-Point Exception

**Cause**

The Floating-Point exception is used by the floating-point coprocessor. This exception is not maskable.

**Exception Level:** 1

**Vector Address:** 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

**Processing**

The common exception vector is used for this exception, and the FPE code in Cause register is set.

The contents of the Floating-Point Control/Status register indicate the cause of this exception.

This exception is cleared by clearing the appropriate bit in the Floating-Point Control/Status register.

For an unimplemented instruction exception, the kernel should emulate the instruction; for other exceptions, the kernel should pass the exception to the user program that caused the exception.
The C790 processor provides a memory management unit (MMU) which uses an on-chip translation look-aside buffer (TLB) to translate virtual addresses into physical addresses.

The C790 supports the MIPS compatible 32-bit address and 64-bit data mode. Only 32-bit virtual and physical addresses have been implemented. There is no requirement for address sign extension and address error exception checking will not be done on the “upper” 32-bits (which are ignored). The only condition that will generate the address error exception will be address alignment errors and segment protection errors. In Kernel mode, there will be address error exception free program counter wrap-around from kseg3 to kuseg.

Since there is only one addressing mode, all the four MIPS ISAs (I, II, III, IV) and the C790 specific ISA are available without any restrictions in all of the three processor modes (with the appropriate MIPS ISA coprocessor usable restrictions). As such the reserved instruction (RI) exception will occur only when the processor really tries to execute an undefined opcode.

This chapter describes the processor virtual and physical address spaces, the virtual-to-physical address translation, the operation of the TLB in making these translations, and those System Control Coprocessor (COP0) registers that provide the software interface to the TLB.
6.1 Translation Look-aside Buffer (TLB)

Mapped virtual addresses are translated into physical addresses using an on-chip TLB. The TLB is a fully associative memory that holds 48 entries, which provide mapping to 48 odd / even page pairs (96 pages). When address mapping is indicated, each TLB entry is checked simultaneously for a match with the virtual address that is extended with an ASID stored in the low 8 bits of the EntryHi register.

The address mapped to a page ranges in size from 4 KB to 16 MB, in multiples of four; that is, 4K, 16K, 64K, 256K, 1M, 4M, 16M.

6.1.1 Translation Status

In C790 processor, as the one implemented in R4000, each TLB entry holds two sets of mapping information for two odd/even page pair and therefore the translation result is categorized into three states, hit, miss and invalid.

Upon address translation, if there is no virtual address match in all 48 entries, the translation result is categorized as TLB miss. In this case, an exception is taken and software refills the TLB from the page table resident in memory. Software can write over a selected TLB entry or use a hardware mechanism to write into a random entry.

If there is a match on translation, the following takes place in the TLB hardware.

1. The translation information for odd page and even page is read out of the matching entry. Also the page size is extracted at the same time.

2. The TLB selects either of translation information in accordance with the page size information extracted above and the virtual address. This becomes the translation result in the TLB.

The translation result includes a valid flag to indicate the translation information is valid or not. If the flag is marked as ‘valid’, the translation is handled as TLB hit. The physical page number is extracted from the TLB and concatenated with the offset to form the physical address (see Figure 6-1).

If the flag is marked as ‘invalid’, the translation result is recognized as TLB invalid. In this case, an exception is taken to request the software to update the entry that got a match upon translation, by probing the TLB using TLBP operation.

6.1.2 Multiple Matches

Multiple match is the condition that there are two or more entries that match upon address translation. This is strictly prohibited and software is expected never to allow this to occur.

The C790 processor does NOT provide any meanings to detect this in hardware, such as TLB shutdown. The result of this condition is undefined and the further execution may provide incorrect result.
6.2 Address Spaces

This section describes the virtual and physical address spaces and the manner in which virtual addresses are converted or “translated” into physical addresses in the TLB.

6.2.1 Virtual Address Space

The C790 only implements 32 bits of virtual address space. There is no requirement for address sign extension and no checking will be done on the upper 32 bits of the address.

Figure 6-1 shows the translation of a virtual address into a physical address.

1. Virtual address (VA) represented by the virtual page number (VPN) is concatenated with the ASID and compared with the tags in the TLB.

2. If there is a match, the page frame number (PFN) representing the upper bits of the physical address (PA) is output from the TLB.

4. The Offset, which does not pass through the TLB, is then concatenated to the PFN.

As shown in Figure 6-2, the virtual address is extended with an 8-bit address space identifier (ASID), which reduces the frequency of TLB flushing when switching contexts. This 8-bit ASID is in the COP0 EntryHi register as described later in this chapter.
6.2.2 Physical Address Space

Using a 32-bit address, the processor physical address space encompasses 4 GB. The following section describes the translation of a virtual address to a physical address.

6.2.3 Virtual-to-Physical Address Translation

Converting a virtual address to a physical address begins by comparing the virtual address from the processor with the virtual addresses in the TLB; there is a match when the virtual page number (VPN) of the address is the same as the VPN field of the entry, and either:

- the Global (G) bit of the TLB entry is set, or
- the ASID field of the virtual address (taken from the 8-bit ASID field of the EntryHi register) is the same as the ASID field of the TLB entry.

If there is no match, a TLB Miss exception is taken by the processor and software can refill the TLB from a page table of virtual / physical addresses in memory.

If there is a virtual address match in the TLB, the physical address is output from the TLB and concatenated with the Offset, which represents an address within the page frame space. The Offset does not pass through the TLB. At the same time, the valid bit output from TLB is checked to qualify the translation. If this bit is not set, a TLB Invalid exception is taken by the processor and software can update the TLB.

Virtual-to-physical translation is described in greater detail throughout the remainder of this chapter. Figure 6-9, shown at the end of this chapter, is a detailed flow diagram of this process.
6.2.4 32-bit Address Translation Mode

The C790 supports only 32-bit address translation mode. 64-bit addressing mode is not supported.

Figure 6-2 shows the virtual-to-physical address translation of a 32-bit address.

- The top portion of Figure 6-2 shows a virtual address with a 12-bit, or 4-KB, page size, labeled *Offset*. The remaining 20 bits of the address represent the VPN, and index the 1M-entry page table.
- The bottom portion of Figure 6-2 shows a virtual address with a 24-bit, or 16-MB, page size, labeled *Offset*. The remaining 8 bits of the address represent the VPN, and index the 256-entry page table.

Virtual Address with 1M ($2^{20}$) 4-Kbyte pages

Virtual Address with 256 ($2^8$) 16-Mbyte pages

Figure 6-2. 32-bit Mode Virtual Address Translation
6.2.5 Operating Modes

The processor has the three standard MIPS operating modes:

- User mode
- Supervisor mode
- Kernel mode

Selection between the three modes can be made by the operating system (when in Kernel mode) by writing into Status register’s KSU field. The processor is forced into Kernel mode when the processor is handling a Level 1 exception (the EXL bit is set - also called the Exception Level mode in R-series processors) or a Level 2 exception (the ERL bit is set - also called the Error Level mode in R-series processors).

In the following table, dashes represent ‘don’t cares’.

<table>
<thead>
<tr>
<th>Description</th>
<th>KSU</th>
<th>ERL</th>
<th>EXL</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit User mode</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32-bit Supervisor mode</td>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32-bit Kernel mode</td>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32-bit Kernel mode (Level 1 exception)</td>
<td>-</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>32-bit Kernel mode (Level 2 exception)</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 6-3 shows a state transition among these three modes.

![Figure 6-3 State Transition among Operating Modes](image-url)
Table 6-2 summarizes address space for each operating mode.

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>32-bit User Mode</th>
<th>32-bit Supervisor Mode</th>
<th>32-bit Kernel Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF FFFF to 0xE000 0000</td>
<td>Address Error</td>
<td>Address Error</td>
<td>kseg3 (0.5 GB) Mapped</td>
</tr>
<tr>
<td>0xDFFF FFFF to 0xC000 0000</td>
<td>sseg (0.5 GB) Mapped</td>
<td>kseg3 (0.5 GB) Mapped</td>
<td></td>
</tr>
<tr>
<td>0xBFFF FFFF to 0xA000 0000</td>
<td>Address Error</td>
<td>Address Error</td>
<td>kseg1 (0.5 GB) Unmapped* Uncached</td>
</tr>
<tr>
<td>0x7FFF FFFF to 0x0000 0000</td>
<td>useg (2 GB) Mapped</td>
<td>suseg (2 GB) Mapped</td>
<td>kuseg (2 GB) Mapped (becomes unmapped if ERL is 1)</td>
</tr>
</tbody>
</table>

*Note: Virtual addresses of Kernel segments, kseg0 and kseg1, are not mapped through the TLB and always translated into physical addresses from 0x0000 0000 to 0x1FFF FFFF.

** Note: The kseg0 cache algorithm is controlled by the K0 field in the Config register.
6.2.6 User Mode Operations

In User mode, a single, uniform virtual address space, labeled User segment, is available; its size is:

- 2 GB ($2^{31}$ bytes) (useg)

Figure 6-4 shows User mode virtual address space.

![Virtual Address 32-bit Diagram](image)

The User segment starts at address 0x0000 0000 and the current active user process resides in useg. The TLB identically maps all references to useg from all modes, and controls cache accessibility.

The processor operates in User mode when the Status register contains the following bit-values:

- $KSU$ bits = $10_2$
- $and EXL = 0$
- $and ERL = 0$
Table 6-3 lists the characteristics of the User mode segment, \textit{useg}.

<table>
<thead>
<tr>
<th>Address Bit Values</th>
<th>Status Register Bit Values</th>
<th>Segment Name</th>
<th>Virtual Address Range</th>
<th>Segment Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[31] = 0</td>
<td>102</td>
<td>useg</td>
<td>0x0000 0000 through 0x7FFF FFFF</td>
<td>2 Gbyte (2^31 bytes)</td>
</tr>
</tbody>
</table>

\textbf{User Mode, User Space(\textit{useg})}

In User mode ($KSU = 102$ in the \textit{Status} register), when the most-significant bit of the 32-bit virtual address is set to 0, the \textit{useg} virtual address space is selected; it covers the $2^{31}$ bytes (2 GB) of the current user address space. 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 \textit{useg} through the TLB. Bit settings within the TLB entry for the page determine the cacheability of a reference. 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.
6.2.7 Supervisor Mode Operations

Supervisor mode is designed for layered operating systems in which a true kernel runs in C790 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 following bit-values:

- $KSU = 01_2$
- $and EXL = 0$
- $and ERL = 0$

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF FFFF</td>
<td>Address error</td>
</tr>
<tr>
<td>0xE000 0000</td>
<td>0.5 GB Mapped sseg</td>
</tr>
<tr>
<td>0xC000 0000</td>
<td>Address error</td>
</tr>
<tr>
<td>0xA000 0000</td>
<td>Address error</td>
</tr>
<tr>
<td>0x8000 0000</td>
<td>2 GB Mapped suseg</td>
</tr>
<tr>
<td>0x0000 0000</td>
<td>Address error</td>
</tr>
</tbody>
</table>

Figure 6-5. Supervisor Mode Virtual Address Space

<table>
<thead>
<tr>
<th>Address Bit Values</th>
<th>Status Register Bit Values</th>
<th>Segment Name</th>
<th>Virtual Address Range</th>
<th>Segment Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A[31] = 0$</td>
<td>$01_2$ EXL $00$</td>
<td>suseg</td>
<td>0x0000 0000 through 0x7FFF FFFF</td>
<td>2 Gbyte ($2^{31}$ bytes)</td>
</tr>
<tr>
<td>$A[31:29] = 110_2$</td>
<td>$01_2$ EXL $00$</td>
<td>sseg</td>
<td>0xC000 0000 through 0xDFFF FFFF</td>
<td>0.5 Gbyte ($2^{29}$ bytes)</td>
</tr>
</tbody>
</table>

Table 6-4. Supervisor Mode Segments

**Supervisor Mode, User Space (suseg)**

In Supervisor mode ($KSU = 01_2$ in the Status register), when the most-significant bit of the 32-bit virtual address is set to 0, the suseg virtual address space is selected; it covers the $2^{31}$ 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.

**Supervisor Mode, Supervisor Space (sseg)**

In Supervisor mode ($KSU = 01_2$ in the Status register), when 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.
6.2.8 Kernel Mode Operations

The processor operates in Kernel mode when the Status register contains one of the following values:

- \( KSU = 002 \)
- or \( EXL = 1 \)
- or \( ERL = 1 \)

The processor enters Kernel mode whenever an exception is detected and it remains in Kernel mode until an Exception Return (ERET) instruction is executed. The ERET instruction restores the processor to the mode existing prior to the exception.

Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 6-6.

Table 6-5 lists the characteristics of the kernel mode segments.

![Figure 6-6. Kernel Mode Address Space](image-url)
Table 6-5. Kernel Mode Segments

<table>
<thead>
<tr>
<th>Address Bit Values</th>
<th>Status Register Bit Values</th>
<th>Segment Name</th>
<th>Virtual Address Range</th>
<th>Segment Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[31] = 0</td>
<td>KSU = 002</td>
<td>kuseg</td>
<td>0x0000 0000 through 0x7FFF FFFF</td>
<td>2 Gbyte (2^31 bytes)</td>
</tr>
<tr>
<td>A[31:29] = 1002</td>
<td>or kseg0</td>
<td></td>
<td>0x8000 0000 through 0x9FFF FFFF</td>
<td>0.5 Gbyte (2^29 bytes)</td>
</tr>
<tr>
<td>A[31:29] = 1012</td>
<td>EXL = 1</td>
<td>kseg1</td>
<td>0xA000 0000 through 0xBFFF FFFF</td>
<td>0.5 Gbyte (2^29 bytes)</td>
</tr>
<tr>
<td>A[31:29] = 1102</td>
<td>or ksseg</td>
<td></td>
<td>0xC000 0000 through 0xDFFF FFFF</td>
<td>0.5 Gbyte (2^29 bytes)</td>
</tr>
<tr>
<td>A[31:29] = 1112</td>
<td>ERL = 1</td>
<td>kseg3</td>
<td>0xE000 0000 through 0xFFFF FFFF</td>
<td>0.5 Gbyte (2^29 bytes)</td>
</tr>
</tbody>
</table>

**Kernel Mode, User Space (kuseg)**

In Kernel mode (KSU = 002 or EXL = 1 or ERL = 1 in the Status register), when the most-significant bit of the virtual address, A[31], is a 0, the 32-bit kuseg virtual address space is selected; it covers the full 2^31 bytes (2 GB) 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.

When ERL = 1 in the Status register, the user address, kuseg, region becomes a 2^31-byte unmapped, uncached address space (that is, mapped directly to physical addresses 0x0000 0000 through 0x7FFF FFFF).

**Kernel Mode, Kernel Space 0 (kseg0)**

In Kernel mode (KSU = 002 or EXL = 1 or ERL = 1 in the Status register), when 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 MB) kernel physical space.

References to kseg0 are not mapped through the TLB; the physical address selected is defined by subtracting 0x8000 0000 from the virtual address. The K0 field of the Config register, described in this chapter, controls cacheability and coherency.

**Kernel Mode, Kernel Space 1 (kseg1)**

In Kernel mode (KSU = 002 or EXL = 1 or ERL = 1 in the Status register), when 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 MB) kernel physical space.

References to kseg1 are not mapped through the TLB; the physical address selected is defined 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) is accessed directly.

**Kernel Mode, Supervisor Space (ksseg)**

In Kernel mode (KSU = 002 in the Status register), when 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 MB) supervisor virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.
Kernel Mode, Kernel Space 3 (kseg3)

In Kernel mode (KSU = 002 in the Status register), when the most-significant three bits of the 32-bit virtual address are 111\textsubscript{2}, the kseg3 virtual address space is selected; it is the current $2^{39}$-byte (512 MB) kernel virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.
6.3 System Control Coprocessor

The System Control Coprocessor (COP0) is implemented as an integral part of the CPU, and supports memory management, address translation, exception handling, and other privileged operations. The COP0 registers shown in Figure 6-7 plus a 48-entry TLB make up the MMU.

Each COP0 register has a unique number that identifies it; this number is referred to as the register number. For instance, the PageMask register is register number 5.

Figure 6-7. COP0 Registers and the TLB
6.3.1 Format of a TLB Entry

Figure 6-8 shows the TLB entry formats for the 32-bit address translation modes. Each field of an entry has a corresponding field in the EntryHi, EntryLo0, EntryLo1, or PageMask registers. For example, the Mask field of the TLB entry is also held in the PageMask register.

![Figure 6-8. Format of a TLB Entry](image)

The format of the EntryHi, EntryLo, EntryLo1, and PageMask registers are nearly the same as the TLB entry. The one exception is the Global field (G bit), which is used in the TLB, but is reserved in the EntryHi register. The following register tables describe the TLB entry fields shown in Figure 6-8.
PageMask Register

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>24</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MASK</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MASK  Page comparison mask.
0  Reserved. Must be written as zeroes, and returns zeroes when read.

EntryHI Register

<table>
<thead>
<tr>
<th>31</th>
<th>13</th>
<th>12</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN2</td>
<td>0</td>
<td>ASID</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VPN2  Virtual page number divided by two (maps to two pages).
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.
0  Reserved. Must be written as zeroes, and returns zeroes when read.

EntryLo0 Register

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PFN</td>
<td>C</td>
<td>D</td>
<td>V</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PFN  Page frame number; the upper bits of the physical address.
C  Specifies the TLB page coherency attribute; see Table 6-7.
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 TLB 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.

EntryLo1 Register

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PFN</td>
<td>C</td>
<td>D</td>
<td>V</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PFN  Page frame number; the upper bits of the physical address.
C  Specifies the TLB page coherency attribute; see Table 6-7.
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 TLB 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 TLB page coherency attribute (C) bits specify whether references to the page should be either of cached, uncached, or uncache-accelerated. Table 6-6 shows the coherency attributes selected by the C bits.
Table 6-6  TLB Page Coherency (C) Bit Values

<table>
<thead>
<tr>
<th>C[5:3] Value</th>
<th>Page Coherency Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>Reserved</td>
</tr>
<tr>
<td>2</td>
<td>Uncached</td>
</tr>
<tr>
<td>3</td>
<td>Cacheable, write-back, write-allocate</td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
</tr>
<tr>
<td>5</td>
<td>Reserved</td>
</tr>
<tr>
<td>6</td>
<td>Reserved</td>
</tr>
<tr>
<td>7</td>
<td>Uncached, Accelerated</td>
</tr>
</tbody>
</table>

Write-back with allocate fetches the line with the missed data both on load misses and on store misses. Therefore, storing data to such pages is always performed to the data cache and will not be sent to the write buffer.

Uncached accelerated data provides a special kind of acceleration for handling uncached data. On a load of an uncached accelerated data item (which can range in size from a byte to a quadword) the C790 will always fetch an aligned 128-byte quantity from memory. These eight quadwords will be placed in a special 128-byte buffer called the uncache accelerated buffer, or UCAB in the CPU. Any subsequent loads which “hit” the UCAB will get the data from the UCAB. This process reduces bus traffic. The UCAB will be invalidated under the following conditions:

- Any load operation which doesn’t hit the buffer, or
- any store operation, or
- a SYNC (or SYNC.L) operation, or
- any exception.

For uncached accelerated stores, the C790 write-back buffer (128-bit x 8) also has some special features. On the first store of an uncached accelerated write the write-back buffer will mark the fact that this is an uncached accelerated write to a particular address. Subsequent uncached accelerated stores which hit within the same 128-bit address boundary will be accumulated (gathered) within the same write buffer entry. This process of data gathering reduces bus traffic. The gathering process will be terminated under the following conditions:

- Any store which can’t be gathered (different attribute or different address), or
- any load operation, or
- a SYNC (or SYNC.L) operation, or
- any exception.
6.4 Virtual-to-Physical Address Translation Process

In the supported 32-bit mode, the highest 8 to 20 bits of the virtual address (depending upon the page size) are compared to the contents of the TLB virtual page number. The 8-bit ASID is only compared if the global bit, G, is not set.

If a TLB entry matches, the physical address and access control bits (C, D, and V) are retrieved from the matching TLB entry. While the V bit of the entry must be set for a valid translation to take place, it is not involved in the determination of a matching TLB entry.

Figure 6-9 illustrates the TLB address translation process.
For valid address space, see the section describing Operating Modes in this chapter.

Virtual Address (Input)

Address Error → No

Access Allowed? → Yes

VPN and ASID

Access Allowed? → Yes

User Mode → No

Sup. Mode

Access Allowed? → No

Address Error → Exception

Unmapped Access → No

Mapped Area?

Yes

Address Error → Exception

VPN Match?

No

Yes

G=1?

No

ASID Match?

No

Yes

Match

Not Match

Match?

No match entry

V=1?

No

Yes

Dirty

Non-cacheable

Write?

No

Yes

TLB Mod

Exception

Physical Address (Output)

C = 010 or 111?

No

Yes

Access Main Memory

Access Cache

TLB Invalid

Exception

TLB Refill

D = 1?

Yes

Figure 6-9. TLB Address Translation
If there is no TLB entry that matches the virtual address, a TLB miss exception occurs. If the access control bits (D and V) indicate that the access is not valid, a TLB modified or TLB invalid exception occurs.

If the C bits equal 010₂ (Uncached) or 111₂ (Uncached Accelerated), the physical address that is generated directly accesses main memory, bypassing the cache.

6.5 TLB Instructions

Table 6-7 lists the instructions that the CPU provides for working with the TLB. See Appendix C for a detailed description on these instructions.

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Description of Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLBP</td>
<td>Translation Look-aside Buffer Probe</td>
</tr>
<tr>
<td>TLBR</td>
<td>Translation Look-aside Buffer Read</td>
</tr>
<tr>
<td>TLBI</td>
<td>Translation Look-aside Buffer Write Index</td>
</tr>
<tr>
<td>TLBWR</td>
<td>Translation Look-aside Buffer Write Random</td>
</tr>
</tbody>
</table>
7. Caches

The C790 core contains both an instruction cache and a separate data cache. The processor also contains a small size of read only cache memory for uncached accelerated area.

This chapter describes the cache structures, operation of the caches, and cache control.
7.1 Cache Features

The two caches are configured as shown in Table 7-1:

<table>
<thead>
<tr>
<th>Cache</th>
<th>Size</th>
<th>Organization</th>
<th>Line Size</th>
<th>Refill Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Cache</td>
<td>32 KB</td>
<td>2-Way</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>Data Cache</td>
<td>32 KB</td>
<td>2-Way</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
</tbody>
</table>

The following are the main features of the caches:

- Separate Instruction Cache and Data Cache
- Virtually indexed and physically tagged caches
- 64 Byte line size
- 64 Byte Refill size
- 2-way set-associative cache for higher performance
- Write-back policy for the Data Cache
- Missed quadword first sequential order burst refills for the Data Cache
- Data Cache line locking
- Non-Blocking Loads
- Data cache supports multiple Hits under a single miss
- No Snoop capability

No cache snoop capability has been provided. The user may choose to use \textit{CACHE} instructions to keep coherency between caches and main memory.
7.2 Organization of the Caches

Organization of the caches is illustrated in Figure 7-1 and Figure 7-2. Both the Instruction Cache and the Data Cacher are 2-way set-associative. Each cache line consists of a tag and data. Each cache has a data line size of 64 bytes.

7.2.1 Data Cache

The Data Cache is connected to the CPU via a 128-bit bus. Therefore, the Data Cache can supply to the CPU or the coprocessors up to a quadword of data per access.

The following diagram shows Data Cache structure. Tags are discussed in detail in a later section.

![Data Cache Diagram]

- **L**: Lock Bit  For description, see Section 7.3.7, *Data Cache Lock* Function
- **R**: LRF Bit  For description, see Section 7.3.1, *Line Replacement Algorithm*
- **V**: Valid Bit  For description, see Section 7.2.3, *Tag Structure*
- **D**: Dirty Bit  For description, see Section 7.2.3, *Tag Structure*

Figure 7-1. Organization of Data Cache
7.2.2 Instruction Cache

The Instruction Cache is connected to the CPU pipeline via a 64-bit bus. This enables the CPU to fetch two instructions per cycle from the Instruction Cache.

The following diagram shows Instruction Cache structure. Tags are discussed in detail in a later section.

![Instruction Cache Diagram](image-url)

Figure 7-2. Organization of Instruction Cache
7.2.3 Tag Structure

The general structure of a tag consists of a set of state bits and a physical page frame number or **PFN** field. The Data Cache and the Instruction Cache have different numbers of state bits; for more information, refer to the discussions in the following sections.

The size of the tag and the number of virtual address bits indexing the caches are dependent upon the size of the cache, address space, and set associativity. The C790 supports 32-bit virtual and physical addresses as shown in the figure below:

```
Virtual Address (VA)

  | 31 | 14 | 13 | 12 | 11 | 0 |
---|----|----|----|----|----|---|
   | VPN|    |    |    |    |   |

Physical Address (PA)

  | 31 | 14 | 13 | 12 | 11 | 0 |
---|----|----|----|----|----|---|
   | PFN|    |    |    |    |   |
```

Since the cache line size is fixed at 64 bytes, that is, four quadwords per entry, the Tag Cache associated with each way will have one tag for every four quadwords. Table 7-2 shows cache sizes, address bits and tag size.

<table>
<thead>
<tr>
<th>Cache</th>
<th>Size</th>
<th>Way</th>
<th>Size of Each Way</th>
<th>Cache Virtual Address Index Bits</th>
<th>Tag Cache Size of Each Way</th>
<th>Tag Virtual Address Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>32 K</td>
<td>2 WAY</td>
<td>256 x 64 Bytes</td>
<td>13:4</td>
<td>256 x 20 Bits</td>
<td>13:6</td>
</tr>
<tr>
<td>Instruction</td>
<td>32 K</td>
<td>2 WAY</td>
<td>256 x 64 Bytes</td>
<td>13:3</td>
<td>256 x 20 Bits</td>
<td>13:6</td>
</tr>
</tbody>
</table>

While the caches are indexed by the virtual address, the tag comparison is physical. This is possible because the caches and the TLB are accessed in parallel. So, when the tags have been accessed, the page frame number is ready to be compared against the translated virtual address for a cache hit or miss.

C790 Programming Note:

Overlapping of the cache index bit range and PFN bit range causes the “cache aliasing problem”. C790 does not have any hardware mechanisms to detect the cache aliasing. It is programmer’s responsibility to avoid the cache aliasing. When a physical page is mapped on the different virtual pages, VPN[13:12] have to be same in both virtual address. The conservative way to avoid this is that VPN[13:12] \(\equiv\) PFN[13:12] whenever a page is mapped.
7.2.3.1 Data Cache Tag Structure

In addition to the physical page frame number (PFN), each Data Cache Tag entry also contains additional Cache State bits as shown below. All lines in both ways of the Data Cache have these four state bits. Cache line state bits are also illustrated in Figure 7-1.

<table>
<thead>
<tr>
<th>Data Cache Tag Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dirty (D)</td>
</tr>
</tbody>
</table>

Two state bits, DIRTY and VALID, together identify which of three states the Data Cache is in: Valid Clean, Valid Dirty, or Invalid. Table 7-3 shows the state of the Data Cache line as a function of DIRTY and VALID bits.

<table>
<thead>
<tr>
<th>Dirty Bit (D)</th>
<th>Valid Bit (V)</th>
<th>Cache Line State</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>Invalid</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Valid Clean</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Valid Dirty</td>
</tr>
</tbody>
</table>

The LRF bit is the Least-Recently-Filled line replacement bit.

The LRF bits serve as a replacement algorithm between the two ways of the Data Cache. A refill access to a cache line in a way will flip the LRF bit to point to the other way as the least recently filled. For details of the LRF line update operation refer to Section 7.3.1.

As Figure 7-1 illustrates, Data Cache lines in each way have a LOCK bit. The LOCK bit, as explained in Section 7.3.7, Data Cache Lock Function, locks lines in one of the ways to keep data from being replaced.

7.2.3.2 Instruction Cache Tag Structure

In addition to the physical page frame number (PFN), each Instruction Cache Tag entry also contains two additional Cache State bits as shown below. All lines in both ways of the Instruction Cache have these two state bits.

<table>
<thead>
<tr>
<th>Instruction Cache Tag Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid (V)</td>
</tr>
</tbody>
</table>

The Instruction Cache VALID state bit defines whether each line is in the Valid or Invalid states.

The LRF bit is the Least-Recently-Filled line replacement bit. LRF bits serve as a replacement algorithm between the two ways of the Instruction Cache. A refill access to a cache line in a way will flip the LRF bit to point to the other way as the least recently filled. For details of LRF line update operation refer to Section 7.3.1.
7.2.4 State of Cache Tags After Reset

For all Data Cache tags the following fields are initialized to 0 upon reset:

- Valid
- Dirty
- LRF
- Lock

For all Instruction Cache tags the following fields are initialized to 0 upon reset:

- Valid
- LRF

All other fields in the Instruction Cache and the Data Cache contents are undefined upon reset.
Chapter 7  Caches

7.3 Cache Operations

This section describes cache operation in regard to read/write policies, coherency, write-back policy, and the lock function.

7.3.1 Line Replacement Algorithm

The line replacement policy for both the Instruction Cache and the Data Cache is based on the Least Recently Filled (LRF) algorithm. In this policy, the LRF bit of a way is modified (inverted) only when a cache line refill occurs to the corresponding way. Load/store accesses to the Data Cache do not modify the LRF bit. The bit indicating which way is the least recently filled way is the XOR of the two LRF bits of the two ways of the cache.

Table 7-4. LRF Line Replacement Algorithm

<table>
<thead>
<tr>
<th>Current Way0 LRF</th>
<th>Current Way1 LRF</th>
<th>XOR</th>
<th>Refill Way</th>
<th>New Way0 LRF</th>
<th>New Way1 LRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The column under XOR indicates the way which could be refilled (line replaced) on the next refill at that line location. Note that the table shown above is valid only when none of the ways of the cache line is locked. If a way of the cache line is locked, then regardless of the state of the LRF bits, the least recently filled way will always be the unlocked way.

The behavior is also slightly different for Instruction and Data Caches when one of the way is invalid. For the Data Cache the algorithm is followed exactly as given above irrespective of the ways being valid or invalid. For the Instruction Cache the algorithm given above is followed as long as both the ways are valid. Once a way becomes invalid, then that way gets priority of being filled over the valid way irrespective of the LRF bits.

7.3.2 Non-blocking Loads and Hit Under Miss

The Data Cache supports non-blocking load and hit under miss to improve performance. When a Data Cache miss occurs or an uncached load instruction is issued, Non-blocking load allows the pipeline to continue instruction execution until one of the following occurs:

1. A subsequent non-load/store/pref instruction has data dependency with the load that is pending (to be retired).
2. A pipeline0 stalls.
**Hit under miss** is a feature that allows access (load or store) to the Data Cache while a previous load miss (cached, uncached or uncached accelerated), a previous store miss (cached) or a previous prefetch miss (cached) is still pending. In this case, access to the cache proceeds and the pipe does not stall.

Uncached loads also do not stall the pipeline while they are pending (to be retired). The pipeline continues instruction execution until one of the following occurs:

1. A subsequent load/store/pref instruction has data dependency with the load that is pending (to be retired).
2. A Data Cache miss occurs or a miss occurs on the Uncached Accelerated Buffer.
3. An Uncached load instruction is issued.

To summarize, **Non-blocking load** and **Hit under miss** allow the pipeline to continue instruction execution until one of following occurs when a Data Cache miss occurs or an uncached load instruction is issued:

1. A subsequent instruction has data dependency with the load that is pending (to be retired).
2. A Data Cache miss occurs or a miss occurs on the Uncached Accelerated Buffer.
3. An uncached load instruction is issued.
4. A pipeline stalls.

Loads to the GPRs (IU) and FPRs (FPU) all follow the non-blocking protocol (when it is enabled). Loads to COP1 is **always** blocking.

### 7.3.3 Cache Miss and Hit Operations

In case of a Data Cache hit, the cache provides data to the CPU in 128-bit (single quadword) quantities. In case of an Instruction Cache hit, the cache provides data ("instruction") in 64-bit quantities. CPU reads or writes to the Data Cache in quantities less than 128 bits are specified by the least significant four bits of the address, bits 3:0.

Cache misses are processed by the cache controller in 64-byte quantities - one cache line. Since the caches are connected to the system bus via a 128-bit bus, cache refill takes a burst of 4 bus cycles (8 CPU cycles) that is, four quadwords are transferred in 4 bus cycles (actual transfer time can be more due to bus arbitration etc). These reads are performed in sequential order for both the Instruction Cache and the Data Cache. The quadword for which the address missed is always fetched first.

Table 7-5 indicates the sequential order. PA[5:4] are two least-significant address bits that are put out on the CPU Bus. Figure 7-3 illustrates the case where the second quadword, shaded area, missed and shows the order in which data are read from main memory.
Table 7-5. Quadword Retrieved Address PA[5:4]

<table>
<thead>
<tr>
<th>Bus Cycle</th>
<th>Starting Block Address PA[5:4]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>01</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
</tr>
</tbody>
</table>

128 bits 128 bits 128 bits 128 bits

Read order Third Second First Fourth

Figure 7-3. Read Missed Processed in Sequential Order

In case of a write miss to the Data Cache (for an allocate-on-write address), the cache controller will read in sequential order a cache line from main memory. Whether the cache line, being replaced, is first written out to memory or not - due to the DIRTY bit being set - is discussed in the next section.

The Instruction Cache processes cache misses in burst of 4 quadwords, just like the Data Cache. Furthermore, in case of an Instruction Cache miss, the pipeline starts in the same cycle the final quadword is stored into the Instruction Cache.

7.3.4 Data Cache Writeback Policy

Data cache lines are written back to the memory in the following cases:

1. The processor executes Index Write Back Invalidate CACHE instruction suboperation as defined in Appendix C and the line data are dirty. Or Hit Writeback Invalidate or Hit Writeback without Invalidate CACHE suboperations hit on Data Cache and the line data are dirty.

2. A read or write miss occurs and the line data are dirty. In this case the line has to be written to memory before it can be replaced by the miss data.
### 7.3.5 Data Cache State Transitions

As discussed previously, lines in the Data Cache can be in one of several states: **Invalid**, **Valid Clean** or **Valid Dirty**.

*Invalid* means the Data Cache entry does not contain valid data. Upon a miss, the cache can load data into this cache line with no further actions.

The *Valid Clean* state indicates that there are valid data in the Data Cache line and they are the same as memory. All writeback segments have their data in the *Valid Clean* state until they are written to by the processor.

The C790 supports the write-back protocol, hence the need for a *Valid Dirty* state. A Data Cache line transitions to the *Valid Dirty* state when the cache line is written to without reflecting the operation on the bus – the writeback protocol. In this case, the data in the cache does not match the data in memory.

Figure 7-4 shows the transition diagram of the Data Cache performing according to the writeback policy. For details on the CACHE operation, refer to Appendix C.

![Figure 7-4. Data Cache Transition Diagram, Writeback Protocol](image-url)
7.3.6 Instruction Cache State Transitions

Cache lines in the Instruction Cache can be in either of two states: *Invalid* or *Valid*. *Invalid* means the Instruction Cache entry does not contain valid instruction data. Upon a miss, the cache can load instructions into this cache line with no further actions.

The *Valid* state indicates that there are valid instructions in the cache line and so there is no need for miss processing.

The transition diagram for the Instruction Cache is simple; refer to Figure 7-5. For details on the CACHE instructions refer to Appendix C.

![Instruction Cache Transition Diagram](image)

7.3.7 Data Cache Lock Function

In a 2-way set-associative Data Cache, such as the one present in the C790, there is no explicit way of forcing data to be retained in the cache. The LRF-based mechanism dynamically determines which cache line should be replaced. A Data Cache lock function has been defined to aid in retaining critical pieces of data in the Data Cache under strict program control.

Each entry on each way of the Data Cache has a Lock (L) bit. The Lock bit aids in locking the line by writing directly into it. After locking the line, the LRF bit is no longer meaningful. Thus, if one of the ways for a particular line is locked, the other way is the only way available for caching. Thus, once a line is locked with a particular physical address tag, any other virtual address which maps onto the same cache line will have only a direct mapped location rather than a 2-way location.

To lock the Data Cache, the following two CACHE instruction suboperations can be used:

- INDEX STORE TAG (DCACHE)
- INDEX STORE DATA (DCACHE)

For details of the above CACHE instruction suboperation refer to Section 7.6. To lock a Data Cache line, the following code sequence can be used:
Chapter 7  Caches

7-13

li  t0,0x00010068 //PTagLo = 0x00010, D=V=L=1, R=0
mtc0 t0,$28   //t0 -> TagLo
sync.l
cache 18,0(r0) //TagLo -> Tag(way0)
sync.l
la  s0,0x00010000
sw t1,0(s0) //store contents of t1 into
//locked cache line

In this example, the tag has been modified using the CACHE instruction and the data has been updated using a Store instruction.

The following restrictions apply to line locking:

- The result of re-locking a locked line is undefined
- The results of locking both ways of a cache line are undefined

To unlock Data Cache lines, the following code sequence can be used:

li  t0,0x00010060 //D=V=1, L=R=0
mtc0 t0,$28   //t0 -> TagLo
sync.l
cache 18,0(r0) //TagLo -> Tag(way0)
sync.l

7.3.7.1  Operations During Lock

When the lock bit is set for cache line (index), only the other way is available for handling cache misses. The misses are blocking. A write access to a locked line in the Data Cache takes place only to the cache without affecting the state of memory. Writes to locked cache lines will not set the DIRTY (D) bit.

7.3.8  Relationship Between Cached and Uncached Operations

Uncached and Uncached Accelerated load and store operations are always executed in order on the CPU bus. Cached load operations can precede earlier store data present in buffers on the CPU bus. All store data present in buffers prevents a SYNC (or SYNC.L) instruction from completing until the store data has been sent either to the Data Cache or the CPU bus.

Stores with the uncached and uncached accelerated attributes bypass the Data Cache completely.
Chapter 7  Caches

7.4 Uncached Accelerated Buffer

The C790 has a small size of read only cache memory for uncached accelerated area to reduce bus traffic. This read only cache, the Uncached Accelerated Buffer (UCAB), can introduce data to itself only by refill process due to a load miss on the UCAB. Once load instructions hit on the UCAB, data are provided directly from the UCAB. The UCAB is invalidated under the following conditions:

- Any load operation which doesn’t hit the UCAB, or
- Any store operation, or
- A SYNC (or SYNC.L) operation, or
- Any exception

Snoop is not supported for the UCAB.

7.4.1 UCAB Configuration

The UCAB is configured as shown in Table 7-6.

<table>
<thead>
<tr>
<th>Uncached Accelerated Buffer</th>
<th>128 bytes</th>
<th>Direct Map</th>
<th>128 bytes</th>
<th>128 bytes</th>
</tr>
</thead>
</table>

7.4.2 Tag Structure

The UCAB is also indexed by the virtual address, the tag comparison is physical. Table 7-7 shows the UCAB size and access bits.

<table>
<thead>
<tr>
<th>UCAB</th>
<th>128 B</th>
<th>Direct Map</th>
<th>1x128 Bytes</th>
<th>6:4</th>
<th>1x25 Bits</th>
<th>—</th>
</tr>
</thead>
</table>

The least significant 5 bits of the UCAB Tag ([11:7]) is identical with the virtual address [11:7]. The UCAB Tag has one bit of valid bit. The UCAB Tag doesn’t have Ditty, LRF, Lock bits. The valid bit of UCAB Tag is initialized to 0 upon reset.

7.4.3 Non-blocking Loads and Hit under Miss

The UCAB also supports non-blocking load and hit under miss as well as the Data Cache. Non-blocking load and Hit under miss allow the pipeline to continue instruction execution until one of following occurs when an Uncached Accelerated Buffer miss occurs:

1. A subsequent instruction has data dependency with the load that is pending (to be retired).
2. A Data cache miss occurs or a miss occurs on the UCAB.
3. An uncached load instruction is issued.
4. A pipeline0 stalls.
7.5 Cache Control Registers

The operations of the caches are controlled by certain programmable bits in the Config register. These bits are:

- ICE: Instruction Cache Enable
- DCE: Data Cache Enable
- IC: Instruction Cache Size
- DC: Data Cache Size
- IB: Icache Line Size
- DB: Dcache Line Size

For details of these configuration bits refer to the COP0 register section.

The two cache tag registers TagLo and TagHi are 32-bit read/write registers that hold the tag and state of the cache line during initialization and diagnostics. The Tag registers are manipulated by MTC0 and CACHE instructions.

**TagLo**

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTagLo</td>
<td>0</td>
<td>D</td>
<td>V</td>
<td>R</td>
<td>L</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TagHi**

where

- PTagLo: Specifies physical address bits 31:12
- D: Cache State DIRTY bit (Not used for the Instruction Cache)
- V: Cache State VALID bit
- R: LRF Bit
- L: LOCK Bit (Not used for the Instruction Cache)
- 0: Must be written as zeros, will return zero on reads

The TagHi register contains instruction- and operation-specific items (see the next section).
7.6 CACHE Instruction

For information on the CACHE instruction, please refer to Appendix C.
8. CPU Bus

The C790 CPU core is connected to the rest of the system, and to external devices, through the group of on-chip C790 system bus signals called the CPU Bus. This chapter defines the architecture of the CPU Bus and describes it in the context of an overall system design.

This chapter describes the following:

- the CPU Bus architecture and agents on the CPU Bus
- the types of transactions possible between agents on the bus
- the bus protocols for transactions

1 The system consists of a DMA Controller (DMAC) as a master, and various slave devices.
Chapter 8  CPU Bus

8.1 Introduction

The CPU Bus is an on-chip bus in a highly integrated processor. All agents (see definitions section 8.1.1 below) on the CPU Bus are equipped with a CPU Bus interface unit connected via CPU Bus signals. An agent acts like a master when it initiates reads or writes on the bus. An agent acts like a slave when it responds to reads or writes initiated by a master. For the CPU Bus to operate properly, an arbiter is needed, to perform arbitration between the CPU and the other bus masters. The arbiter is located in the CPU, and CPU arbitration behavior is discussed in Section 8.5.1, Arbitration Operations.

The following are main features of the CPU Bus:
- Separate data and address buses (Demultiplexed operation)
- 128-bit data bus
- Clocked synchronous operations
- Peak transfer rate of 2.1GB/sec (@133 MHz bus clock)
- 8/16/32/64/128-bit and burst accesses
- Multimaster capability
- Pipelined operations
- No turn-around or dead cycles between transfers

The CPU Bus does not provide:
- Cache coherency support
- Split transactions
8.1.1 Terminology

**Address Phase** is the cycles during which an address is driven on the CPU Bus through the cycle the address is acknowledged.

**Agent** refers to different devices on the CPU Bus.

**Assert** means taking a signal to its active level. An active high signal is “1” when asserted, and an active low signal is “0” when asserted.

**CPU** means the C790 CPU. The terms CPU and C790 are used interchangeably in this chapter.

**Data Phase** is the cycles during which data are driven on the bus through the cycle they are acknowledged.

**DMAC** is the DMA Controller in the system.

**Master** means the current bus master on the CPU Bus.

**MEM** refers to the system memory controller.

**Negate/Deassert** means taking a signal to its inactive state. An active high signal is “0” when deasserted. An active low signal is “1” when negated.

* (after signal name) means active low signal.

8.1.2 Signal Naming Convention

Table 8-1 shows the prefixes used for naming signals in a system incorporating the C790 CPU Bus.

<table>
<thead>
<tr>
<th>Signal Prefix</th>
<th>Signal Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>Signals from the CPU multiplexed or logically combined with the DMAC signals to form the system signals. These signals include: CPUADDR, CPUBE*, CPURD*, CPUWR*, CPUSIZE, CPUASTART*, CPUDSTART*, CPUDATA.</td>
</tr>
<tr>
<td><strong>SYS</strong></td>
<td>The combined or multiplexed signals from any agents on the CPU Bus. These signals include: SYSADDR, SYSBE*, SYSRD*, SYSWR*, SYSTSIZE, SYSASTART*, SYSDSTART*, SYSAACK*, SYSDACK*, SYSDATA.</td>
</tr>
</tbody>
</table>
8.2 CPU Bus Architecture

The CPU Bus design is a synchronous pipelined bus with separate data (128-bit) and address buses running at half the clock frequency of the CPU. The CPU is connected to the rest of the system and external devices through this bus. Figure 8-1 illustrates the architecture of the bus and identifies different agents that can be on the bus.

Figure 8-1. CPU Bus Architecture
8.2.1 CPU Bus Connectivity for Address and Control Paths

Figure 8-2 illustrates the system-level interconnections for address paths of the CPU Bus.

Support logic is needed to handle the fact that the system contains multiple masters. AGNT* is used to control the multiplexer in the support logic that selects a master to be connected to the CPU Bus.

![Diagram of CPU Bus Address and Control Path Connections in System](image)

Figure 8-2. CPU Bus Address and Control Path Connections in System
8.2.2 CPU Bus Connectivity for Data Paths

Figure 8-3 illustrates the system-level interconnections for data paths of the CPU Bus.

For read cycles, the support logic must control the multiplexer so that the correct source of data is put on SYSDATA.

For write cycles, the support logic must detect whether the cycle is a CPU cycle or a DMA cycle, and use this to control the multiplexer.

Figure 8-3. CPU Bus Data Path Connections in System
8.3 CPU Bus Signal Descriptions

This section describes the CPU Bus signals and their usage in different bus operations.

8.3.1 Address Bus Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUADDR[31:4]</td>
<td>CPU address bus</td>
</tr>
<tr>
<td></td>
<td>CPUADDR[31:4] bits are valid during the address phase and can be sampled by the slave when CPUASTART* is sampled low.</td>
</tr>
<tr>
<td>SYSADDR[31:4]</td>
<td>System address bus</td>
</tr>
<tr>
<td></td>
<td>SYSADDR[31:4] are multiplexed outputs selecting between CPUADDR[31:4] and DMA address. They are valid during the address phase and can be sampled by the slave when SYSASTART* is sampled low.</td>
</tr>
<tr>
<td>CPUBE[15:0]*</td>
<td>CPU byte enables</td>
</tr>
<tr>
<td></td>
<td>CPUBE[i]<em>, driven during the address phase, indicates valid data on byte i of CPUDATA[127:0] during the data phase. CPU byte enables can be sampled by the slave when CPUASTART</em> is sampled low. CPU byte enables are used only in CPU single cycles.</td>
</tr>
<tr>
<td>SYSBE[15:0]*</td>
<td>System byte enables</td>
</tr>
<tr>
<td></td>
<td>SYSBE[i]<em>, driven during the address phase, indicates valid data on byte i of SYSDATA[127:0] during the data phase. System byte enables can be sampled by the slave when SYSASTART</em> is sampled low. System byte enables are used only in CPU single cycles.</td>
</tr>
</tbody>
</table>
**CPU TRANSTYPE[4:0]**

CPU transaction type

CPU TRANSTYPE[4:0], driven during the address phase, indicates the type of operation. CPU transaction type can be sampled by the slave when CPUASTART* is sampled low.

Table 8-2. Bus Transaction Types

<table>
<thead>
<tr>
<th>CPU TRANSTYPE</th>
<th>Type of Bus Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Not defined or miscellaneous</td>
</tr>
<tr>
<td>00001 - 00111</td>
<td>Reserved</td>
</tr>
<tr>
<td>01000</td>
<td>Data Cache Refill due to Load Miss</td>
</tr>
<tr>
<td>01001</td>
<td>Data Cache Refill due to Prefetch Instruction</td>
</tr>
<tr>
<td>01010</td>
<td>Data Cache Refill due to Store Miss</td>
</tr>
<tr>
<td>01011</td>
<td>Uncached Load</td>
</tr>
<tr>
<td>01100</td>
<td>Uncached Accelerated Load</td>
</tr>
<tr>
<td>01101 - 01111</td>
<td>Reserved</td>
</tr>
<tr>
<td>10000</td>
<td>Instruction Cache Miss Refill</td>
</tr>
<tr>
<td>10001</td>
<td>Cache Instruction - Fill Suboperation</td>
</tr>
<tr>
<td>10010</td>
<td>Uncached Execution</td>
</tr>
<tr>
<td>10011 - 10111</td>
<td>Reserved</td>
</tr>
<tr>
<td>11000</td>
<td>Data Cache Write-back due to Load/Store Miss</td>
</tr>
<tr>
<td>11001</td>
<td>Data Cache Write-back due to Cache Instruction</td>
</tr>
<tr>
<td>11010</td>
<td>Uncached Store</td>
</tr>
<tr>
<td>11011</td>
<td>Uncached Accelerated Store</td>
</tr>
<tr>
<td>11100</td>
<td>Non-allocated Store</td>
</tr>
<tr>
<td>11101 - 11111</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

**CPURD***

CPU read

The CPU asserts this signal to indicate a read operation. This signal can be sampled when CPUASTART* is sampled low. This signal is active during the address phase. CPURD* is used in transfers initiated by the CPU.

**CPUWR***

CPU write

The CPU asserts this signal to indicate a write operation. This signal can be sampled when CPUASTART* is sampled low. This signal is active during the address phase. CPUWR* is used in transfers initiated by the CPU.
CPUTSIZE[1:0]  CPU transfer size

While driven by the CPU, these signals indicate the size of the transfer in the current CPU initiated bus cycle. They are driven during the address phase and can be sampled starting at the edge where CPUASTART* is sampled low.

<table>
<thead>
<tr>
<th>CPUTSIZE[1:0]</th>
<th>Transfer Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1 Quadword (Single Cycle)</td>
</tr>
<tr>
<td>11</td>
<td>4 Quadwords</td>
</tr>
</tbody>
</table>

SYSTSIZE[2:0]  System transfer size

While driven by the system, these signals indicate the size of the transfer in the current system bus cycle. They are driven during the address phase and can be sampled starting at the edge where SYSASTART* is sampled low.

CPUASTART*  CPU address start

Driven by the CPU, it indicates the start of the address phase. Address, byte enable, and control signals (CPUADDR[31:4], CPUBE[15:0]*, CPURD*, CPUWR*, and CPUTSIZE) can be sampled to determine the type of cycle requested starting where CPUASTART* is sampled low. CPUASTART* is driven active for only one cycle.

SYSASTART*  System address start

SYSASTART* is driven by the system; it indicates the start of the address phase. Address, byte enable, and control signals can be sampled to determine the type of cycle requested starting where SYSASTART* is sampled low. SYSASTART* is driven active for only one cycle.

SYSAACK*  System address acknowledge

This signal is an input to all the agents on the CPU Bus indicating that address and control signals have been sampled by the slave. The master terminates the address phase one cycle after sampling SYSAACK* low.

CPUDATA[127:0]  CPU data bus

This is a 128-bit data bus output from the CPU.

SYSDATA[127:0]  System data bus

This is the 128-bit data bus input to all devices on the CPU Bus.
CPU data start

CPUDSTART*  

During read/write operations, this output from the CPU indicates the start of data phase. For CPU write operations, the slave can sample data from the bus one cycle after CPUDSTART* has been asserted. For CPU read operations, the slave can output data on the bus any cycle after the cycle CPUDSTART* has been asserted.

System data start

SYSDSTART*  

During read/write operations, this output from the system indicates the start of data phase. Data transfer can begin one cycle after SYSDSTART* has been asserted. For DMA cycles, if the slave, providing the data, cannot supply data in the next cycle after the assertion of SYSDSTART*, it is the responsibility of the designer to come up with a new DMA protocol.

System data acknowledge

SYSDACK*  

This signal is an input to all the agents on the bus indicating the valid status of data on the bus. During read cycles, it indicates read data are available on the bus to be sampled by the master. During write cycles, it indicates the slave has sampled the data. This signal should be asserted for each data transfer during burst operations. During read transactions, data are sampled one cycle after SYSDACK* has been asserted. During write transactions, the master drives new data on the bus one cycle after detecting SYSDACK* low.

Bus error

BUSERR*  

This signal is an input to the CPU and the DMAC which indicates that a bus error has occurred during the transaction. BUSERR* serves to terminate the bus protocol and return bus ownership to the CPU.

Interrupt request lines

INT[1:0]*  

These signals are interrupt inputs to the CPU.

Serial I/O interrupt request

SIOINT*  

This line provides the serial I/O interrupt from the I/O controller.

Non-maskable interrupt

NMI*  

Non-maskable interrupt input to the CPU.

Big Endian enable

SYSBIGENDIAN  

This input signal is sampled during cold reset and make CPU to operate as big endian when it is asserted. The input level of this signal must not be changed during the operation.
CPCOND0  Coprocessor conditions

These lines are an input to the CPU as test conditions for some of the branch instructions.

RESET*  Reset

Input to the CPU. When this line is asserted, the CPU, DMAC and slave devices execute a reset.

CPUCLK  CPU clock

BUSCLK  Bus clock

Bus clock: 1/2, 1/3 or 1/4 frequency of the CPUCLK.

AREQ*  Address bus request

This signal is an output from the DMAC to the CPU. When it is asserted, the DMAC requests the address bus mastership.

AGNT*  Address bus grant

This signal is an output from the CPU to grant the bus mastership to the DMAC. This signal is asserted in response to assertion of the AREQ* signal.

REL*  Bus release request

This signal is asserted by the CPU to request that the current bus owner release the CPU Bus.
8.4 Overview of CPU Bus Operations

This section discusses CPU Bus operations; it covers processor requests, DMA operations, and bus error operation.

In this section descriptions show CPU signals followed by the system lines, in parentheses, onto which they are asserted. For example: CPUASTART* (SYSASTART*) means CPUASTART* is asserted on the SYSASTART* line. Where a value is given, the bits output by the CPU are shown, followed by the bits, in parentheses, on the system lines. For example if we have 11 on CPUTSIZE[1:0], during a CPU bus cycle, then we will get 011 on the SYSTSIZE[2:0]. This will be shown as 11 (011).

8.4.1 CPU Bus Operations

The CPU Bus is different from conventional buses in that it allows pipeline operations. In this case, pipeline implies up to two outstanding requests before any data transaction has taken place. For instance, the CPU may issue two back-to-back read requests to main memory before any data have been returned. Note that at any time, there can only be two outstanding requests on the bus. The master requiring more than two operations has to wait until the first request has been serviced completely prior to issuing the third one.

8.4.2 Processor Requests

The CPU issues single requests, burst requests or a series of requests to other agents on the bus. These requests are referred to as processor requests initiated through the CPU Bus interface.

The processor requests are in response to the following system events:

- Load miss
- Store miss
- Write-back buffer writes (dirty data cache lines, uncached writes, etc.)
- Uncached loads and uncached accelerated loads
- Instruction miss and uncached instruction fetch

Processor read/write requests can be a burst, quadword, or partial quadword of data to and from the main memory or any other system resources. A processor-initiated burst is always 4 quadwords.

8.4.2.1 Read Requests

The CPU initiates read requests by driving address and control on the bus and asserting CPUASTART* (SYSASTART*) to indicate valid address and control. The CPU will keep driving address and control until the slave device has acknowledged the address phase by asserting address acknowledge, SYSAACK*. For burst reads, the CPU drives CPUTSIZE (SYSTSIZE) to 11 (011) to indicate burst reads. The CPU also indicates that it is ready to accept read data by asserting CPUĐSTART* (SYDSTART*). The slave device returns the requested data on the data bus by asserting SYSDACK*, data acknowledge.
8.4.2.2 Write Requests

The CPU initiates write requests by driving address and control on the bus and asserting CPUASTART* (SYSASTART*). The CPU also drives data on the bus and indicates that by asserting CPUDSTART* (SYSDSTART*). The slave device accepts the address and data by asserting SYSAACK* and SYSDACK*, respectively. Burst writes are indicated by driving CPUFSIZE (SYTFSIZE) to 11 (011) during the address phase.

8.4.3 Bus Error Operations

Bus error occurs when the CPU or DMA initiates cycles but there are no devices on the CPU Bus responding to the cycles. The absence of response to either the address phase or the data phase will cause the bus error condition. The bus error is always imprecise.

When bus error occurs, all the agents including the CPU, DMAC, and slave devices on the CPU Bus will terminate the current bus cycle.

In the case where CPU is the initiator of the cycle, there can be two types of bus error:

- Data load/store bus error
- Instruction fetch bus error

Bus error sets the corresponding exception bit in the CAUSE register. Subsequently, the CPU will jump to the proper error handler for the examination of the exception. However, the bus error exception is imprecise. There is no guarantee that the CPU can recover from this error condition.

In case the DMAC is the initiator of the cycle, the types of bus error depends on the implementation of the DMAC. After bus error occurs, the DMAC will release the bus mastership back to the CPU and assert interrupt or NMI to the CPU. The interrupt or NMI routine will then handle the bus error condition for the DMAC.
8.5 CPU Bus Transaction Protocols and Timing

This section describes transaction protocols and the timing for the following CPU Bus operations:

- Arbitration
- CPU single operations (one quadword)
- CPU burst operations (four quadwords)
- CPU non-pipelined single operations (one quadword)
- CPU non-pipelined burst operations (four quadwords)
- Bus error operations

8.5.1 Arbitration Operations

An arbiter is required to mediate between devices requesting the CPU Bus. The arbiter is located in the CPU. The CPU is the default bus master; AREQ* and AGNT* are both deasserted during RESET.

A master other than the CPU may request the bus by asserting the request signal, AREQ*. In response to the AREQ* signal, the CPU will issue the grant signal, AGNT*, to grant the address bus to the requesting master. In the cycle AGNT* is sampled active by the bus master, the master starts the address phases and deasserts AREQ* in the beginning of the last address phase. When the corresponding data phases commences, the CPU or the requesting master starts the data transfers depending on the DMA transfer. Data phases follow the exact order of address phases. The arbitration signals are shown in Figure 8-4.

Figure 8-4. Connection of Arbitration Signals

The arbitration priority in using the CPU Bus is that the DMAC always has higher priority than the CPU. When both the CPU and the DMAC arbitrate for the CPU Bus, the arbiter grants the bus mastership to the DMAC. The CPU can assert REL* to the DMAC in an effort to get the bus ownership back from the DMAC. The CPU will proceed with the transfer once the DMAC has released the CPU Bus.

The arbitration cycles and protocol are shown in Figure 8-5. In response to the DMAC asserting its request AREQ*, the arbiter asserts AGNT* in cycle 3 which is the arbitration cycle. The DMAC samples AGNT* asserted and begins its address phases. When the DMAC asserts to begin the last address phase, it deasserts its request line AREQ* in cycle 4. The arbiter then waits for the SYSAACK* cycle to deassert AGNT* to release bus mastership back to the CPU.
8.5.1.1 Cycle Stealing

Cycle stealing refers to the CPU’s ability to preempt a master in order to perform a bus operation. This operation could be either due to the write back buffer (WBB) being almost full (having more than 64 bytes filled up) or the CPU needing to perform an instruction or data read. These operations are collectively referred to as cycle stealing operations.

Figure 8-6 illustrates the cycle stealing protocol. The arbiter asserts the REL* (Release) signal in response to the CPU’s request cycles. The master deasserts its request after having finished its operations. When the master has begun the last address phase with the master deasserts the AREQ* signal indicating to the arbiter that the bus will be relinquished; as indicated in cycle 9. When the address phase ends, the address bus is returned to the CPU by the deassertion of AGNT* in cycle 12. The arbiter deasserts REL* at the same time AGNT* is deasserted. The data phases follow the same order as the address phases.
8.5.2 CPU Single Operations

CPU Single operations transfer one quadword.

In single operations, the CPU drives the address, byte enables, and the read/write signals and indicates their valid status by asserting CPUASTART* (SYSASTART*). The slave samples valid address and control lines and responds by asserting SYSAACK*. In single operations, CPUSIZE (SYSTSIZE) is always 00 (000).

When the CPU detects SYSAACK* active and is ready to put another address on the bus, it will start another address phase. The bus only supports two levels of address pipelining. That means only two address phases can be outstanding before any data phase begins.

The CPU indicates that it is ready to accept/supply data by asserting CPUDSTART* (SYSDSTART*) one cycle prior to actually accepting/supplying it. For read cycles, the slave supplies the data and indicates that the data is ready by asserting SYSDACK*. For write cycles, the CPU supplies data one cycle after CPUDSTART* (SYSDSTART*) is asserted, and the slave accepts the data by asserting SYSDACK*.

8.5.2.1 CPU Single Reads

The fastest CPU single read is 2 cycles. Address and data phases for AddrA illustrate the fastest CPU single read cycle. The CPU asserts CPUASTART* (SYSASTART*) to begin the address phase in cycle 1. The slave device asserts SYSAACK* in cycle 1 to indicate that it has sampled the address. The CPU then begin another address phase in cycle 3. The assertion of SYSDACK* by the slave device in cycle 1 triggers the CPU to sample SYSDATA at the end of cycle 2.

![Figure 8-7. CPU Single Reads](image-url)
8.5.2.2  CPU Single Writes

The fastest CPU single write is 2 cycles. Address and data phases for AddrA illustrate the fastest CPU single write cycle. The CPU always drives data onto CPUDATA one cycle after the assertion of CPUDSTART* (SYSDSTART*). For example, in cycle 2, the CPU drives CPUDATA in cycle 2 which is one cycle after the assertion of CPUDSTART* (SYSDSTART*) in cycle 1. The slave device samples SYSDATA one cycle after the assertion of SYSDACK*.

![Figure 8-8. CPU Single Writes](image)
8.5.2.3 CPU Single Read-Write-Read-Write Cycles

All adjacent address phases are read-write or write-read cycles. AddrA is a read address and AddrB is a write address, and so on.

![Diagram of CPU Single Read-Write-Read-Write Cycles]

Figure 8-9. CPU Single Read-Write-Read-Write Cycles
8.5.3 CPU Burst Operations

CPU Burst operations transfer four quadwords. In burst operations, the CPU drives the address and control signals and indicates their validity by asserting CPUASTART* (SYSASTART*). The slave samples valid address and control lines and asserts SYSAACK* to acknowledge the address phase. The address phase is the cycles from CPUASTART* (SYSASTART*) asserted to one cycle after SYSAACK* is asserted.

When the CPU detects SYSAACK* active and has another address ready, it will start another address phase.

The CPU indicates that it is ready to accept/supply data by asserting CPUDSTART* (SYSDSTART*) one cycle prior to actually accepting/supplying it. For read cycles, the slave supplies the data and indicates that data are valid by asserting SYSDACK* one cycle prior to the data being available. For write cycles, the CPU supplies data one cycle after CPUDSTART* (SYSDSTART*) is asserted, and the slave accepts the data by asserting SYSDACK*. For burst cycles, there are many SYSDACK* for data transfer.

The CPUTSIZE (SYSTSIZE) indicates the number of quadwords in the transfer. The CPU initiated cycles use only values of either 00 (for CPU Single operations) or 11 (for CPU Burst operations), which are single and burst of 4 quadwords respectively.

8.5.3.1 CPU Burst Reads

The fastest CPU burst read is 5 cycles. Address and data phases for AddrA illustrate the fastest CPU burst read cycle. There are four SYSDACK* sent by the slave device for every CPU burst read cycle. The slave device asserts SYSDACK* in cycle 1, 2, 3, and 4 to indicate that data can be sampled at the end of cycle 2, 3, 4, and 5 by the CPU.

![Figure 8-10. CPU Burst Reads](image-url)
8.5.3.2 CPU Burst Writes

The fastest CPU burst write is 5 cycles. Address and data phases for AddrA illustrate the fastest CPU burst write cycle. After assertion of CPUDSTART* (SYSDSTART*) in cycle 1, the CPU drives the first data on CPUDATA in cycle 2. As SYSDACK* is sampled asserted in cycles 1, 2, 3, and 4, the CPU drives a new data on CPUDATA at the end of cycles 2, 3, 4, and 5.

![Figure 8-11. CPU Burst Writes](image-url)
8.5.3.3 CPU Burst Read-Write Cycles

All adjacent address phases are read-write or write-read cycles. AddrA is a read address and AddrB is a write address, and so on.

Figure 8-12. CPU Burst Read-Write Cycles

8.5.3.4 CPU Burst Write-Read Cycles

All adjacent address phases are read-write or write-read cycles. AddrA is a write address and AddrB is a read address, and so on.

Figure 8-13. CPU Burst Write-Read Cycles
8.5.4 CPU Non-Pipeline Single Operations

The CPU Bus can support non-pipeline operations as well as pipeline operations. The non-pipeline operations are done simply by delaying the assertion of SYSAACK* until the last SYSDACK* of the bus transaction. The advantage of this is that the peripheral does not need to save the current address; it just decodes the address on the address bus for the current operation. Using this mode of operation simplifies the peripheral interfaces to the CPU Bus but it degrades the system performance.

8.5.4.1 CPU Non-Pipeline Single Reads

All adjacent address phases are read cycles.

Figure 8-14. CPU Non-Pipeline Single Reads
### 8.5.4.2 CPU Non-Pipeline Single Writes

All adjacent address phases are write cycles.

![Diagram of CPU Non-Pipeline Single Writes]

**Figure 8-15. CPU Non-Pipeline Single Writes**

### 8.5.5 CPU Non-Pipeline Burst Operations

#### 8.5.5.1 CPU Non-Pipeline Burst Reads

All adjacent address phases are read cycles.

![Diagram of CPU Non-Pipeline Burst Reads]

**Figure 8-16. CPU Non-Pipeline Burst Reads**
8.5.5.2 CPU Non-Pipeline Burst Writes

All adjacent address phases are write cycles.

```
<table>
<thead>
<tr>
<th>BUSCLK</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSADDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPUDATA</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td></td>
<td>B1</td>
<td>B2</td>
<td>B3</td>
<td>B4</td>
<td></td>
</tr>
<tr>
<td>SYSDATA</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td></td>
<td>B1</td>
<td>B2</td>
<td>B3</td>
<td>B4</td>
<td></td>
</tr>
<tr>
<td>SYSTSIZE</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SYSWR*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYSRD*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYSASTART*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYSAACK*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYSDSTART*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYSDACK*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 8-17. CPU Non-Pipeline Burst Writes
8.5.6  Bus Error Operations

Bus error occurs when there are no slave responding to the address or data phases of the bus cycle. When bus error occurs, the current bus operation is terminated, and the system proceeds with the next bus operation. Without bus error detection, the CPU Bus would remain waiting indefinitely for the SYSAACK* or SYSDACK* signals.

Bus error is generated by the CPU Bus monitor logic. The monitor logic basically makes sure that for both address and data phases in the current CPU Bus cycle, there are SYSAACK* and SYSDACK*, respectively. In the case, when there is no SYSAACK* or SYSDACK* or response to the address or data phase for a pre-defined period of time for the current CPU Bus cycle, bus error is generated by asserting BUSERR* for one CPU Bus clock. Bus error has higher priority than SYSAACK* or SYSDACK* if they are detected in the same cycle.

Bus error is always asserted in reference to the data phase of the cycle. The exact timing is the cycles from SYSDSTART* asserted to the cycle before the assertion of the next SYSDSTART*. The bus error signal is sampled when the system is waiting for the assertion of SYSDACK* and/or SYSAACK* of the operation corresponding to the current data phase. For example, if the address phase of a certain cycle has no response from the slave devices, the bus monitor logic will wait until the SYSDSTART* of the corresponding data phase before generating the bus error. The bus monitor logic can generate the bus error any time before the next data phase begins.

8.5.6.1  Bus Error Exceptions

As mentioned before, two operations can be pipelined on the CPU bus, and these two operations can be initiated from either the CPU as master or the DMAC as master.

If the bus error occurs in the CPU initiated operation, the following occurs:

- a bus error exception due to instruction fetch or data access is generated
- the bus error instruction or data address is recorded in the BadPAddr Register of COP0
- the Status.BEM bit is set (This bit is the bus error mask (BEM) in the COP0 Status Register).

Once a bus error occurs, any further bus errors are ignored until Status.BEM is cleared by the bus error exception handler.

If the bus error occurs in the DMA initiated operation (DMA cycle), the DMAC will finish the pending pipeline operations, disable itself, release the CPU Bus, and cause an interrupt. The interrupt routine will then service and re-enable the DMAC accordingly. Table 8-4 summarizes the exception generation:

<table>
<thead>
<tr>
<th>Operation with the Bus Error</th>
<th>Exception Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Initiated Instruction Fetch</td>
<td>Bus Error Exception - Instruction Fetch</td>
</tr>
<tr>
<td>CPU Initiated Data Access</td>
<td>Bus Error Exception - Data Access</td>
</tr>
<tr>
<td>DMA Cycle</td>
<td>Interrupt Exception</td>
</tr>
</tbody>
</table>

Table 8-4. Bus Error Exceptions
8.5.6.2 CPU Bus Cycle Termination

Two pipeline operations can be in progress at any time, but if a bus error occurs, only the operation with the bus error is terminated. That is, the occurrence of a bus error with one master does not affect the program execution of another master. For example, if bus error occurs when the first and second operations are initiated from the DMAC and CPU, respectively, the CPU Bus will terminate the DMA operation and continue with the CPU operation. Table 8-5 summarizes CPU Bus cycle sequence for all types of CPU Bus cycle termination.

Table 8-5. Operation Termination Sequence

<table>
<thead>
<tr>
<th>First Operation with Bus Error</th>
<th>Second Operation</th>
<th>CPU Bus Cycle Sequence</th>
</tr>
</thead>
</table>
| CPU Cycle #1                  | CPU Cycle #2    | 1. CPU Cycle #1 is terminated.  
|                               |                 | 2. Bus Error Exception occurs.  
|                               |                 | 3. CPU Cycle #2 continues on.    |
| CPU Cycle #1                  | DMA Cycle #2    | 1. CPU Cycle #1 is terminated.  
|                               |                 | 2. Bus Error Exception occurs.  
|                               |                 | 3. DMA Cycle #2 continues on.    |
| DMA Cycle #1                  | CPU Cycle #2    | 1. DMA Cycle #1 is terminated.  
|                               |                 | 2. CPU Cycle #2 continues on.    
|                               |                 | 3. DMA releases CPU Bus, disable itself (disable further requests until the interrupt routine re-enable the DMAC), and generate an interrupt.  
|                               |                 | 4. CPU cycles continues on.      |
| DMA Cycle #1                  | DMA Cycle #2    | 1. DMA Cycle #1 is terminated.  
|                               |                 | 2. DMA Cycle #2 continues on.    
|                               |                 | 3. DMAC releases CPU Bus, disable itself (disable further requests until the interrupt routine re-enable the DMAC), and generate an interrupt.  
|                               |                 | 4. CPU cycles continue on.       |

8.5.6.3 Bus Error Timing with No Pending Operation

If there are no pending operations on the bus, BUSERR* is ignored at all times.

8.5.6.4 Bus Error Timing with One Pending Operation

If there is one pending operation on the bus, BUSERR* is sampled while waiting for the assertion of SYSAACK* or SYSDACK*. If BUSERR* is asserted, the bus cycle will continue as if the SYSAACK* and/or the last SYSDACK* has been asserted. Figure 8-18, Figure 8-19, and Figure 8-20 illustrates the bus error associated with one pending operation. In these figures, BUSERR* is ignored before CPUDSTART* and after BUSERR* asserted because the bus is not waiting for the assertion of SYSAACK* nor SYSDACK*. 
Figure 8-18. One Operation with BUSERR* as the Last SYSDACK*
8.5.6.5 Bus Error Timing with Two Pending Operations

If there are two pending operations on the bus, BUSERR* is sampled while waiting for the assertion of SYSDACK*. If BUSERR* is asserted, the bus cycle will continue as if the last SYSDACK* has been asserted. The bus cycle will then proceed with the data phase of the next operation. The bus error that occurred is for the first pending operation.

Figure 8-21 illustrates the bus error associated with two pending operations. In this figure, BUSERR* is ignored after BUSERR* asserted because the bus is no longer waiting for the assertion of SYSDACK* corresponding to operation AddrA with the bus error, and detection of bus error for operation AddrB has not started until the assertion of CPUDSTART*.
Figure 8-21. Two Operations with Bus Error as the Last SYSDACK*
9. Performance Counter

The performance counter provides the means for gathering statistical information about the internal events of the CPU and the pipeline during program execution. The statistics gathered during program execution aid in tuning the performance of hardware and software systems based on the processor.
9.1 Overview

The performance counter consists of one control register and two counters. The control register controls the functions of the monitor while the counters count the number of events specified by the control register.

9.2 Performance Counters and Performance Control Registers

The Performance Counter Control Register, or PCCR, and Performance Counter Registers PCR0 and PCR1 are mapped into COP0 Register 25. Both the register and counters are read/write registers accessible by MTPC, MTPS, MTC0, MFPC, MFPS and MFC0 instructions. Each counter is capable of counting one event as specified by the control register.

The format of the PCCR is shown in Figure 9-1, and the format of PCR0 and PCR1 is shown in Figure 9-2.

```
<table>
<thead>
<tr>
<th>Field</th>
<th>Function</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTE</td>
<td>If 1, PCR0 and PCR1 counting and exception generation is enabled.</td>
<td>0</td>
</tr>
<tr>
<td>EVENT0/1</td>
<td>Event counted by PCR0/1; see Table 9-5 for details.</td>
<td>Undefined</td>
</tr>
<tr>
<td>U0/1</td>
<td>PCR0/1 counts event EVENT0/1 when in User mode.</td>
<td>Undefined</td>
</tr>
<tr>
<td>S0/1</td>
<td>PCR0/1 counts event EVENT0/1 when in Supervisor mode.</td>
<td>Undefined</td>
</tr>
<tr>
<td>K0/1</td>
<td>PCR0/1 counts event EVENT0/1 when in non-exception Kernel mode; i.e. with both STATUS.EXL and STATUS.ERL set to 0.</td>
<td>Undefined</td>
</tr>
<tr>
<td>EXL0/1</td>
<td>PCR0/1 counts event EVENT0/1 when in Level 1 exception handler.</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
```

Figure 9-1. Format of the Performance Counter Control Register PCCR

Figure 9-2. Format of Performance Counter Registers PCR0 and PCR1

The interpretation of the PCCR register bits is as follows:

The interpretation of the PCCR register bits is as follows:
9.2.1 Accessing Counters and Registers

The counter control register \( PCCR \) and the two performance counter registers \( PCR0 \) and \( PCR1 \) are accessed by using \( MTC0^* \) and \( MFC0^* \) instructions. All three registers are mapped to \( COP0 \) register 25. Table 9-2 illustrates how these registers are written by using the \( MTC0 \) instruction, and Table 9-3 illustrates the encoding of the \( MFC0 \) instructions used to read the registers.

Table 9-4 shows special mnemonics to access the performance counters and registers.

---

Table 9-2. Writing Performance Counters and Registers using MTC0

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11001</td>
<td>00</td>
<td>Move to Counter Control Register</td>
</tr>
<tr>
<td>11001</td>
<td>01</td>
<td>Move to Performance Counter Register 0</td>
</tr>
<tr>
<td>11001</td>
<td>10</td>
<td>unused</td>
</tr>
<tr>
<td>11001</td>
<td>11</td>
<td>Move to Performance Counter Register 1</td>
</tr>
</tbody>
</table>

Table 9-3. Reading Performance Counters and Registers using MFC0

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11001</td>
<td>00</td>
<td>Move from Counter Control Register</td>
</tr>
<tr>
<td>11001</td>
<td>01</td>
<td>Move from Performance Counter Register 0</td>
</tr>
<tr>
<td>11001</td>
<td>10</td>
<td>unused</td>
</tr>
<tr>
<td>11001</td>
<td>11</td>
<td>Move from Performance Counter Register 1</td>
</tr>
</tbody>
</table>

Table 9-4. Mnemonics to Access the Performance Counters and Registers

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTPC</td>
<td>Move to Performance Counter</td>
</tr>
<tr>
<td>MTPS</td>
<td>Move to Performance Event Specifies</td>
</tr>
<tr>
<td>MFPC</td>
<td>Move from Performance Counter</td>
</tr>
<tr>
<td>MFPS</td>
<td>Move from Performance Event Specifies</td>
</tr>
</tbody>
</table>

\* MTPC, MTPS, MFPC and MFPS are the special encoding of MTC0 and MFC0.
9.2.2 State of Performance Counter Control Registers Upon Reset

The CTE bit of the Performance Counter Control Register PCCR is initialized to 0 upon reset. This prevents event counting and interrupt generation until the control registers are initialized. It also allows a precise way for counters to be initialized by software; see the section 9.3.2 for more details. Note that the remaining bits of PCCR and both registers PCR0 and PCR1 must be initialized by software.
9.3 Counter Operation

The performance counters PCR0 and PCR1 increment by 1 whenever their corresponding count event occurs, and the counter is enabled. The count event for PCR0 is specified by PCCR.EVENT0 and the count event for PCR1 is specified by PCCR.EVENT1. The encoding of the EVENT field is specified in Table 9-5, and discussed in detail later. A counter is enabled only when both of the following conditions are satisfied:

1. The global counter enable flag PCCR.CTE is set to 1, and
2. The current privilege mode matches the permitted privilege mode for each counter. The values in PCCR.U0, PCCR.S0, PCCR.K0, and PCCR.EXL0 specify the permitted privilege modes for PCR0 and PCCR.U1. PCCR.S1, PCCR.K1, and PCCR.EXL1 specify the permitted privilege modes for PCR1. For example, if the current privilege mode is SUPERVISOR, PCR0 will operate only if PCCR.S0 is set to 1. Note that there is no “ERL0” or “ERL1” flag in PCCR. This is because counters are unconditionally disabled when in level 2 handlers.
### 9.3.1 Counter Events

A counter increments if it is enabled and its trigger event occurs. The permissible values for \texttt{PCCR.EVENT0} and \texttt{PCCR.EVENT1} are as shown in Table 9-5 below. The events are described in Section 9.3.1.1 Event Descriptions.

#### Table 9-5. Counter Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Counter 0</th>
<th>Counter 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>reserved</td>
<td>Low-order branch issued</td>
</tr>
<tr>
<td>1</td>
<td>Processor cycle</td>
<td>Processor cycle</td>
</tr>
<tr>
<td>2</td>
<td>Single instruction issue</td>
<td>Dual instruction issue</td>
</tr>
<tr>
<td>3</td>
<td>Branch issued</td>
<td>Branch mispredicted</td>
</tr>
<tr>
<td>4</td>
<td>BTAC miss</td>
<td>JTLB miss</td>
</tr>
<tr>
<td>5</td>
<td>ITLB miss</td>
<td>DTLB miss</td>
</tr>
<tr>
<td>6</td>
<td>I$ miss</td>
<td>D$ miss</td>
</tr>
<tr>
<td>7</td>
<td>DTLB accessed</td>
<td>WBB single request unavailable</td>
</tr>
<tr>
<td>8</td>
<td>Non-blocking load/store</td>
<td>WBB burst request unavailable</td>
</tr>
<tr>
<td>9</td>
<td>WBB single request</td>
<td>WBB burst request almost full</td>
</tr>
<tr>
<td>10</td>
<td>WBB burst request</td>
<td>WBB burst request full</td>
</tr>
<tr>
<td>11</td>
<td>CPU address bus busy</td>
<td>CPU data bus busy</td>
</tr>
<tr>
<td>12</td>
<td>Instruction completed</td>
<td>Instruction completed</td>
</tr>
<tr>
<td>13</td>
<td>Non-BDS instruction completed</td>
<td>Non-BDS instruction completed</td>
</tr>
<tr>
<td>14</td>
<td>reserved</td>
<td>COP1 instruction completed</td>
</tr>
<tr>
<td>15</td>
<td>Load completed</td>
<td>Store completed</td>
</tr>
<tr>
<td>16</td>
<td>No event</td>
<td>No event</td>
</tr>
<tr>
<td>17-31</td>
<td>reserved</td>
<td>reserved</td>
</tr>
</tbody>
</table>
9.3.1.1 Event Descriptions

In event descriptions, the word 'branch' (for example, 'branch issued', or 'branch mispredicted') means any 'transfer of control' instruction that is subject to prediction (that is, all the conditional branch instructions, J, and JAL). The JR, JALR, ERET, SYSCALL, BREAK, and TRAP instructions are not included.

**Branch issued** This event is triggered whenever a branch is issued to a functional pipe. Note that a branch that is issued in a pipelined implementation may get canceled if an instruction prior to it signals an exception.

**Branch mispredicted** This event is triggered whenever the predicted branch address (taken or not-taken) is incorrect. Note that a branch that is issued in a pipelined implementation may get canceled if an instruction prior to it signals an exception.

**BTAC miss** This event is triggered whenever the instruction address lookup into the BTAC fails. Counts low-order (even) branch instructions that miss the BTAC. Note that high-order (odd) branch does not refer the BTAC.

**COP1 instruction completed** This event is triggered when a COP1 instruction completes. The event is signaled even if the COP1 instruction completes successfully, but appears in the branch delay slot of a branch-likely instruction and is therefore nullified.

**CPU address bus busy** Generates a signal once every BUSCLK (not CPU clock) that the CPU address bus is unavailable. The CPU address bus is considered unavailable whenever it is busy, or when two addresses have been issued but the data for the first address has yet to return.

**Data cache miss** This event is triggered whenever a data cache miss is detected. See Table 9-6. for the D$ miss definition.

### Table 9-6. Definition of Data Cache Miss

<table>
<thead>
<tr>
<th>Access</th>
<th>DCE</th>
<th>Page Attr.</th>
<th>Hit/Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>0</td>
<td>Uncached, UCA, Cached</td>
<td>Miss</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Uncached, UCA</td>
<td>Miss</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cached</td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>0</td>
<td>Uncached, UCA, Cached</td>
<td>Hit</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Uncached, UCA</td>
<td>Hit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cached</td>
<td></td>
</tr>
<tr>
<td>Pref</td>
<td>0</td>
<td>Uncached, UCA, Cached</td>
<td>Uncount *</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Uncached, UCA</td>
<td>Uncount *</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cached</td>
<td></td>
</tr>
</tbody>
</table>

In this event, the data cache miss is defined as any load/store/pref instructions which may generate bus read operations to get missed data from external memory.

* Prefetch to the Uncached or UCA page is considered as nop.
DTLB accessed

Barring canceled instructions, this event counts the total number of executed loads and stores. Thus, ‘data cache miss’ divided by ‘DTLB accessed’ provide a good estimate of the D miss rate (assuming no uncached loads/stores occur). Also, ‘DTLB miss’ divided by ‘DTLB accessed’ provides the DTLB miss rate. DTLB is accessed even when unmapped page is accessed in case that minor revision number is 0x10 or later.

DTLB Miss

This event is triggered whenever a DTLB miss is detected. DTLB is accessed even when unmapped page is accessed in case that minor revision number is 0x10 or later.

Dual instruction issued

This event is signaled whenever both functional pipes of the C790 are issued instructions*. The event counter is incremented by 1.

Instruction cache miss

This event is triggered whenever an instruction cache miss is detected.

Instruction completed

This event triggers when an instruction completes. Note that some instructions (e.g. SYSCALL, TEQ, TEQI, etc) signal exceptions as a normal part of their operation. Such instructions are considered complete whether or not the "normal" exception was raised. Therefore, an “instruction complete” event is signaled even if a TEQ succeeds (i.e. raises a Trap exception). However, if a “true” exception occurs (e.g. a counter exception is signaled while the TEQ is executing), the instruction is canceled and no “instruction complete” signal is generated. Similarly, an instruction in the branch delay slot (BDS) of a branch-likely instruction is counted as complete even if the BDS instruction is nullified. If the BDS instruction is canceled because of a “true” exception, no “instruction completed” event is signaled.

C790 Implementation Note: Up to two instructions can complete every cycle in the C790. When two instructions do complete, the event counter is incremented by 2.

ITLB miss

This event is triggered whenever an ITLB miss is detected.

JTLB miss

This event is triggered whenever a JTLB miss is detected.

Load completed

This event triggers when a load instruction completes. Note that the event is signaled even if the load appears in the branch delay slot of a branch-likely instruction that is not taken and is therefore nullified.

Low-order branch issued

Counts the numbers of branches that were issued that appeared in the low-order (even) position of an instruction pair fetch. This count is needed since only these branches are subject to BTAC lookup.

No event

This “event” effectively disables the corresponding counter. It is useful principally if only one of the two counters need be activated.

Non-BDS instruction completed (for stepping)

This event triggers when an instruction that does not have a branch delay slot completes. In particular, it does not trigger when a branch or jump instruction completes. However, it does trigger when the instruction in the branch delay slot of the branch or jump completes. In the case of a branch-likely instruction, the instruction in the branch delay slot triggers the event even if this instruction is nullified. Note: this event is useful for stepping over instructions.

* (Dual instruction issued) * 2 + (Single instruction issued) = instruction issued
(Instruction issued) – (instruction completed) = instruction canceled
<table>
<thead>
<tr>
<th>Event Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Non-blocking load/store (1st cache miss):</strong></td>
<td>This event is signaled whenever a cached load/store/pref instruction misses on the Data Cache and there is no pending data cache miss, UCAB miss and uncached load.</td>
</tr>
<tr>
<td><strong>Processor cycle</strong></td>
<td>This event triggers on every processor clock cycle.</td>
</tr>
<tr>
<td><strong>Single instruction issued</strong></td>
<td>This event is signaled whenever only one of the functional pipes of the C790 is issued an instruction*.</td>
</tr>
<tr>
<td><strong>Store completed</strong></td>
<td>This event triggers when a store instruction completes. Note that the event is signaled even if the store appears in the branch delay slot of a branch-likely instruction that is not taken and is therefore nullified.</td>
</tr>
<tr>
<td><strong>WBB Single Request</strong></td>
<td>A non-burst request was made to the WBB.</td>
</tr>
<tr>
<td><strong>WBB Burst Request</strong></td>
<td>A burst request was made to the WBB.</td>
</tr>
<tr>
<td><strong>WBB Single Request unavailable</strong></td>
<td>A non-burst request was made to the WBB, but there were insufficient free entries in the WBB to service it. All 8 entries are used at that time.</td>
</tr>
<tr>
<td><strong>WBB Burst Request unavailable</strong></td>
<td>A burst request was made to the WBB, but, the WBB was completely full, or there were not enough to service the request. 5, 6, 7, 8 entries are used at that time.</td>
</tr>
<tr>
<td><strong>WBB Burst Request almost full</strong></td>
<td>A burst request was made to the WBB, and even though there were free entries, there were not enough to service the request. 5, 6, 7 entries are used at that time.</td>
</tr>
<tr>
<td><strong>WBB Burst Request full</strong></td>
<td>A burst request was made to the WBB, but the WBB was completely full. All 8 entries are used at that time.</td>
</tr>
</tbody>
</table>

* (Dual instruction issued) $\times 2 + $ (Single instruction issued) = instruction issued

(Instruction issued) − (instruction completed) = instruction canceled
9.3.2 Handling Performance Counter Exceptions

A performance counter exception is detected by an instruction if the following condition holds true:

~STATUS.ERL && PCCR.CTE && (CTR0.OVFL || CTR1.OVFL)

Note that software should not rely on the exception occurring if the instruction is nullified; i.e. it appears in the branch delay slot of a branch likely instruction that is not taken.

C790 Implementation Note: C790 implementation always counts events that occur within nullified instructions.

The instruction detecting a counter exception is canceled by the exception, and instruction execution continues as follows:

if (in branch delay slot) {
    ErrorEPC = PC - 4;
    CAUSE.BD2 = 1;
} else {
    ErrorEPC = PC;
    CAUSE.BD2 = 0;
}

if (STATUS.DEV)
    PC = 0xBFC00280;  // Uncached counter xcp handler
else
    PC = 0x80000080;  // “Normal” counter xcp handler
STATUS.ERL = 1;
CAUSE.EXC2 = 2;     // Counter exception

The description above makes use of the BD2 and EXC2 fields in the CAUSE register. Both are fields newly introduced in the C790 and occupy the bit positions shown below.

![Figure 9-3. CAUSE Register Fields](image)

C790 Programming Note: Note that the “normal” exception entry point is in kseg0 space. That is, the address is unmapped and the caching policy is determined by CONFIG.K0. If you don’t want to disturb the cache while counting and stepping, kseg0 should be configured in “uncached” mode. If cache data preservation is secondary to counter exception servicing performance counter overflow, kseg0 should be configured in “cached” mode.
9.3.3 Priority of Counter Exceptions

Counter exceptions have the highest priority after cold reset and NMI. If a cold reset occurs the processor is initialized - so a simultaneous counter exception is discarded. If an NMI occurs, the NMI handler is entered with either PCR0.OVFL or PCR1.OVFL (or both) set to 1, and ErrorEPC pointing at the instruction causing the counter overflow. (ErrorEPC is used because NMI is handled as a level 2 exception.) Once the NMI handler exits, the instruction that caused the overflow is re-executed. However, since PCR0.OVFL or PCR1.OVFL is 1, the instruction is canceled once more and the counter exception handler is entered.

9.3.4 Initializing Counters

Let us look at the code sequence needed to initialize counters and activate them. In the example below, PCR0 is set up to count clocks in all operating modes and report a counter exception after the count exceeds $2^{31}$. CTR1 is set up to count stores while in supervisor mode only, and report a counter exception after the count exceeds $2^{31}$. The code must be executed while in level 2 exception mode (ERL=1).

```plaintext
STATUS.ERL = 1; // Set ERL (to inhibit counting)
ErrorEPC = <target instruction where counting is to start>

PCR0 = 0; // Init CTR0, and ...
PCCR.EVENT0 = 1; // ... set up to count clocks ...
PCCR.U0 = 1; // ... in all privilege modes
PCCR.S0 = 1;
PCCR.K0 = 1;
PCCR.EXL0 = 1;

PCR1 = 0; // Init PCRT1, and ...
PCCR.EVENT1 = 15; // ... set up to count completed stores ...
PCCR.U1 = 0; // ... while in supervisor mode
PCCR.S1 = 1;
PCCR.K1 = 0;
PCCR.EXL1 = 0;

PCCR.CTE = 1; // Enable global counter flag
ERET // Execute ERET to clear ERL -
      // counting begins with ERET’s target
      // Note that the ERET instruction also
      // guarantees that the COP0 state
      // updated (e.g. CCR) is valid.
```
9.3.5 The Note to Read Counters

Whenever you want to read a counter by MTC0 or MTPC, be sure that any counting events must NOT occur, otherwise you may get wrong number. For example, counter for TLB event should be read in the unmapped area, that of instruction completion event should be read in the ERL=1 (level 2 exception) area or other disabled area.

It is a implement-dependent that when the event is counted. It depends on the number of the pipeline stages and so on.

To write a robust code among silicon versions and mask versions, you read the counters after flushing the pipeline by SYNC.P instruction. C790 is a pipeline processor. It is required for the instruction completion type event.

It is a nature of event counting that some inaccuracy exists. You don't need to be surprised if different number is observed in different version of silicon/mask.
10. Floating-Point Unit, CP1 (Option)

This chapter describes the floating-point operations, including the programming model, instruction set and formats.

Chapter 10  Floating-Point Unit, CP1

10.1 Overview

All floating-point instructions, as defined in the MIPS ISA for the floating-point coprocessor, CP1, are processed by the other hardware unit that executes integer instructions.

The floating point execution unit can be disabled by the coprocessor usability CU bit defined in the CP0 Status register.

10.2 Floating Point Register

10.2.1 Floating-Point General Registers (FGRs)

CP1 has a set of Floating-Point General Purpose registers (FGRs) that can be accessed in the following ways:

- As 32 general purpose registers (32 FGRs), each of which is 32 bits wide when the FR bit in the CPU Status register equals 0; or as 32 general purpose registers (32 FGRs), each of which is 64-bits wide when FR equals 1. The CPU accesses these registers through move, load, and store instructions.

- As 16 floating-point registers (see the next section for a description of FPRs), each of which is 64-bits wide, when the FR bit in the CPU Status register equals 0. The FPRs hold values in either single- or double-precision floating-point format. Each FPR corresponds to adjacent numbered FGRs as shown in Figure 10-1.

- As 32 floating-point registers (see the next section for a description of FPRs), each of which is 64-bits wide, when the FR bit in the CPU Status register equals 1. The FPRs hold values in either single- or double-precision floating-point format. Each FPR corresponds to an FGR as shown in Figure 10-1.
### Floating-Point Registers (FPR)

<table>
<thead>
<tr>
<th>FPR0</th>
<th>FGR0</th>
<th>FPR0</th>
<th>FGR0</th>
</tr>
</thead>
<tbody>
<tr>
<td>(least)</td>
<td>FGR0</td>
<td>(most)</td>
<td>FGR0</td>
</tr>
<tr>
<td>FPR1</td>
<td>FGR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(least)</td>
<td>FGR2</td>
<td>(most)</td>
<td>FGR2</td>
</tr>
<tr>
<td>FPR2</td>
<td>FGR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(least)</td>
<td>FGR28</td>
<td>(most)</td>
<td>FGR28</td>
</tr>
<tr>
<td>FPR28</td>
<td>FGR29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(least)</td>
<td>FGR30</td>
<td>(most)</td>
<td>FGR30</td>
</tr>
<tr>
<td>FPR30</td>
<td>FGR31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Floating-point Control Registers (FCR)

<table>
<thead>
<tr>
<th>31 (FCR31)</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control/Status Register</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31 (FCR0)</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation/Revision Register</td>
<td></td>
</tr>
</tbody>
</table>

### Floating-point General Purpose Registers

<table>
<thead>
<tr>
<th>31 (FGR)</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>63 (FGR)</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10-1. FP Registers
10.2.2 Floating-Point Registers (FPRs)

The FPU provides:

- 16 Floating-Point registers (FPRs) when the FR bit in the Status register equals 0, or
- 32 Floating-Point registers (FPRs) when the FR bit in the Status register equals 1.

These 64-bit registers hold floating-point values during floating-point operations and are physically formed from the General Purpose registers (FGRs). When the FR bit in the Status register equals 1, the FPR references a single 64-bit FGR.

The FPRs hold values in either single- or double-precision floating-point format. If the FR bit equals 0, only even numbers (the least register) can be used to address FPRs. When the FR bit is set to a 1, all FPR register numbers are valid.

If the FR bit equals 0 during a double-precision floating-point operation, the general registers are accessed in double pairs. Thus, in a double-precision operation, selecting Floating-Point Register 0 (FPR0) actually addresses adjacent Floating-Point General Purpose registers FGR0 and FGR1.

10.2.3 Floating-Point Control Registers

The MIPS RISC architecture defines 32 floating-point control registers (FCRs); the C790 processor implements two of these registers: FCR0 and FCR31. These FCRs are described below:

- The Implementation/Revision register (FCR0) holds revision information.
- The Control/Status register (FCR31) controls and monitors exceptions, holds the result of compare operations, and establishes rounding modes.
- FCR1 to FCR30 are reserved.

Table 10-1 lists the assignments of the FCRs.

<table>
<thead>
<tr>
<th>FCR Number</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCR0</td>
<td>Coprocessor implementation and revision register</td>
</tr>
<tr>
<td>FCR1 to FCR30</td>
<td>Reserved</td>
</tr>
<tr>
<td>FCR31</td>
<td>Rounding mode, cause, trap enables, and flags</td>
</tr>
</tbody>
</table>
Implementation and Revision Register (FCR0)

The read-only *Implementation and Revision* register (FCR0) specifies the implementation and revision number of CP1. This information can determine the coprocessor revision and performance level, and can also be used by diagnostic software.

Figure 10-2 shows the layout of the register; Table 10-2 describes the *Implementation and Revision* register (FCR0) fields.

![Implementation/Revision Register](image)

Table 10-2. FCR0 Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imp</td>
<td>Implementation number</td>
<td>0x38</td>
</tr>
<tr>
<td>Rev</td>
<td>Revision number in the form of y. x</td>
<td>Revision Number</td>
</tr>
<tr>
<td>0</td>
<td>Reserved. Returns zeroes when read.</td>
<td></td>
</tr>
</tbody>
</table>

The revision number is a value of the form y.x, where:

- y is a major revision number held in bits 7:4.
- x is a minor revision number held in bits 3:0.

The revision number distinguishes some chip revisions; however, there is not guarantee that changes to its chips are necessarily reflected by the revision number, or that changes to the revision number necessarily reflect real chip changes. For this reason revision number values are not listed, and software should not rely on the revision number to characterize the chip.

**IEEE Standard 754**

IEEE Standard 754 specifies that floating-point operations detect certain exceptional cases, raise flags, and can invoke an exception handler when an exception occurs. These features are implemented in the MIPS architecture with the *Cause*, *Enable*, and *Flag* fields of the *Control/Status* register. The *Flag* bits implement IEEE 754 exception status flags, and the *Cause* and *Enable* bits implement exception handling.
Control/Status Register (FCR31)

The Control/Status register (FCR31) contains control and status information that can be accessed by instructions in either Kernel or User mode. FCR31 also controls the arithmetic rounding mode and enables User mode traps, as well as identifying any exceptions that may have occurred in the most recently executed floating-point instruction, along with any exceptions that may have occurred without being trapped.

Figure 10-3 shows the format of the Control/Status register, and Table 10-3 describes the Control/Status register fields. Figure 10-4 shows the Control/Status register Cause, Flag, and Enable fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS</td>
<td>When set, denormalized results can be flushed instead of causing an unimplemented operation exception.</td>
</tr>
<tr>
<td>C</td>
<td>Condition bit. See description of Control/Status register Condition bit.</td>
</tr>
<tr>
<td>Cause</td>
<td>Cause bits. See Figure 10-4 and the description of Control/Status register Cause, Flag, and Enable bits.</td>
</tr>
<tr>
<td>Enables</td>
<td>Enable bits. See Figure 10-4 and the description of Control/Status register Cause, Flag, and Enable bits.</td>
</tr>
<tr>
<td>Flags</td>
<td>Flag bits. See Figure 10-4 and the description of Control/Status register Cause, Flag, and Enable bits.</td>
</tr>
<tr>
<td>RM</td>
<td>Rounding mode bits. See Table 10-5 and the description of Control/Status register Rounding Mode Control bits.</td>
</tr>
</tbody>
</table>
Chapter 10  Floating-Point Unit, CP1

Figure 10-4. Control/Status Register Cause, Flag, and Enable Fields

Control/Status Register FS Bit

The FS bit enables the flushing of denormalized values. When the FS bit is set and the Underflow and Inexact Enable bits are not set, denormalized results are flushed instead of causing an Unimplemented Operation exception. Results are flushed to either 0 or the minimum normalized value, depending upon the rounding mode (see Table 10-4 below), and the Underflow and Inexact of the Cause and Flag bits are set.

Table 10-4. Flush Values of Denormalized Results

<table>
<thead>
<tr>
<th>Denormalized Result</th>
<th>Flushed Result Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RN</td>
</tr>
<tr>
<td>Positive</td>
<td>+0</td>
</tr>
<tr>
<td>Negative</td>
<td>-0</td>
</tr>
</tbody>
</table>

Control/Status Register Condition Bit

When a floating-point Compare operation takes place, the result is stored at bit 23, the Condition bit. The C bit is set to 1 if the condition is true; the bit is cleared to 0 if the condition is false. Bit 23 is affected only by compare and CTC1 instructions.
Control/Status Register Cause, Flag, and Enable Fields

Figure 10-4 illustrates the Cause, Flag, and Enable fields of the Control/Status register. The Cause and Flag fields are updated by all conversion, computational (except MOV. fmt), CTC1, reserved, and unimplemented instructions. All other instructions have no affect on these fields.

**Cause Bits**

Bits 17:12 in the Control/Status register contain Cause bits, as shown in Figure 10-4, which reflect the results of the most recently executed floating-point instruction. The Cause bits are a logical extension of the CP0 Cause register; they identify the exceptions raised by the last floating-point operation. If the corresponding Enable bit is set at the time of the exception a floating-point exception is raised and trapped by CPU. If more than one exception occurs on a single instruction, each appropriate bit is set.

The Cause bits are updated by most floating-point operations. The Unimplemented Operation (E) bit is set to 1 if software emulation is required, otherwise it remains 0. The other bits are set to 0 or 1 to indicate the occurrence or non-occurrence (respectively) of an IEEE 754 exception. Within the set of floating-point instructions that update the Cause bits, the Cause field indicates the exceptions raised by the most-recently-executed instruction.

When a floating-point exception is taken, no results are stored, and the only state affected is the Cause bit.

**Enable Bits**

A floating-point exception is generated any time a Cause bit and the corresponding Enable bit are set. A floating-point operation that sets an enabled Cause bit forces an immediate floating-point exception, as does setting both Cause and Enable bits with CTC1.

There is no enable for Unimplemented Operation (E). An Unimplemented exception always generates a floating-point exception.

Before returning from a floating-point exception, software must first clear the enabled Cause bits with a CTC1 instruction to prevent a repeat of the exception trapping. Thus, User mode programs can never observe enabled Cause bits set; if this information is required in a User mode handler, it must be passed somewhere other than the Status register.

For a floating-point operation that sets only unenabled Cause bits, no floating-point exception occurs and the default result defined by IEEE 754 is stored. In this case, the exceptions that were caused by the immediately previous floating-point operation can be determined by reading the Cause field.
Flag Bits

The Flag bits are cumulative and indicate the exceptions that were raised by the operations that were executed since the bits were explicitly reset. Flag bits are set to 1 if an IEEE 754 exception is raised, otherwise they remain unchanged. The Flag bits are never cleared as a side effect of floating-point operations; however, they can be set or cleared by writing a new value into the Status register, using a CTC1 instruction.

When a floating-point exception is trapped, the flag bits are not set by the hardware; floating-point exception software is responsible for setting these bits before invoking a user handler.

Control/Status Register Rounding Mode Control Bits

Bits 1 and 0 in the Control/Status register constitute the Rounding Mode (RM) field.

As shown in Table 10-5, these bits specify the rounding mode that CP1 uses for all floating-point operations.

<table>
<thead>
<tr>
<th>Rounding ModeRM (1:0)</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RN</td>
<td>Round result to nearest representable value; round to value with least-significant bit 0 when the two nearest representable values are equally near.</td>
</tr>
<tr>
<td>1</td>
<td>RZ</td>
<td>Round toward 0: round to value closest to and not greater in magnitude than the infinitely precise result.</td>
</tr>
<tr>
<td>2</td>
<td>RP</td>
<td>Round toward +∞: round to value closest to and not less than the infinitely precise result.</td>
</tr>
<tr>
<td>3</td>
<td>RM</td>
<td>Round toward −∞: round to value closest to and not greater than the infinitely precise result.</td>
</tr>
</tbody>
</table>

10.2.4 Accessing the FP Control and Implementation/Revision Registers

The Control/Status and the Implementation/Revision registers are read by a Move Control From Coprocessor 1 (CFC1) instruction.

The bits in the Control/Status register can be set or cleared by writing to the register using a Move Control To Coprocessor 1 (CTCI) instruction. The Implementation/Revision register is a read-only register. There are no pipeline hazards (between any instructions) associated with floating-point control registers.
10.3 Floating-Point Formats

CP1 performs both 32-bit (single-precision) and 64-bit (double-precision) IEEE standard floating-point operations. The 32-bit single-precision format has a 24-bit signed-magnitude fraction field \( f_s \) and an 8-bit exponent \( e \), as shown in Figure 10-5.

![Figure 10-5. Single-Precision Floating-Point Format](image)

The 64-bit double-precision format has a 53-bit signed-magnitude fraction field \( f_s \) and an 11-bit exponent, as shown in Figure 10-6.

![Figure 10-6. Double-Precision Floating-Point Format](image)

As shown in the above figures, numbers in floating-point format are composed of three fields:

- sign field, \( s \)
- biased exponent, \( e = E + bias \)
- fraction, \( f = b_1b_2\ldots b_{p-1} \)

where \( bias = 127 \), \( p = 24 \) in single precision,

\[ bias = 1023, \; p = 53 \] in double precision

The range of the unbiased exponent \( E \) includes every integer between the two values \( E_{\min} \) and \( E_{\max} \) inclusive, together with two other reserved values:

- \( E_{\min} - 1 \) (to encode 0 and denormalized numbers)
- \( E_{\max} + 1 \) (to encode \( \infty \) and NaNs [Not a Number])

For single- and double-precision formats, each representable nonzero numerical value has just one encoding uniquely.

For single- and double-precision formats, the value of a number, \( v \), is determined by the equations shown in Table 10-6.
Table 10-6. Equations for Calculating Values in Single and Double-Precision Floating-Point Format

<table>
<thead>
<tr>
<th>Equation</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v = \text{NaN} )</td>
<td>( E = E_{\text{max}} + 1 ) and ( f \neq 0 ), regardless of ( s )</td>
</tr>
<tr>
<td>( v = (-1)^s \infty )</td>
<td>( E = E_{\text{max}} + 1 ) and ( f = 0 )</td>
</tr>
<tr>
<td>( v = (-1)^s 2^{E(1.f)} )</td>
<td>( E_{\text{min}} \leq E \leq E_{\text{max}} )</td>
</tr>
<tr>
<td>( v = (-1)^s 2^{E_{\text{min}}(0.f)} )</td>
<td>( E = E_{\text{min}} - 1 ) and ( f \neq 0 )</td>
</tr>
<tr>
<td>( v = (-1)^s 0 )</td>
<td>( E = E_{\text{min}} - 1 ) and ( f = 0 )</td>
</tr>
</tbody>
</table>

For all floating-point formats, if \( v \) is NaN, the most-significant bit of \( f \) determines whether the value is a signaling or quiet NaN: \( v \) is a signaling NaN if the most-significant bit of \( f \) is set, otherwise, \( v \) is a quiet NaN.

Table 10-7 defines the values for the format parameters; minimum and maximum floating-point values are given in Table 10-8.

Table 10-7. Floating-Point Format Parameter Values

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Single</td>
</tr>
<tr>
<td>( E_{\text{max}} )</td>
<td>+127</td>
</tr>
<tr>
<td>( E_{\text{min}} )</td>
<td>-126</td>
</tr>
<tr>
<td>Exponent bias</td>
<td>+127</td>
</tr>
<tr>
<td>Exponent width in bits</td>
<td>8</td>
</tr>
<tr>
<td>Integer bit</td>
<td>hidden</td>
</tr>
<tr>
<td>Fraction width in bits</td>
<td>23†</td>
</tr>
<tr>
<td>Format width in bits</td>
<td>32</td>
</tr>
</tbody>
</table>

† Excluding the sign bit.

Table 10-8. Minimum and Maximum Floating-Point Values

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float Minimum</td>
<td>1.40129846e-45</td>
</tr>
<tr>
<td>Float Minimum Norm</td>
<td>1.17549435e-38</td>
</tr>
<tr>
<td>Float Maximum</td>
<td>3.40282347e38</td>
</tr>
<tr>
<td>Double Minimum</td>
<td>4.9406564584124654e-324</td>
</tr>
<tr>
<td>Double Minimum Norm</td>
<td>2.2250738585072014e-308</td>
</tr>
<tr>
<td>Double Maximum</td>
<td>1.7976931348623157e308</td>
</tr>
</tbody>
</table>
10.4 Binary Fixed-Point Format

Binary fixed-point values are held in 2’s complement format. Unsigned fixed-point values are not directly provided by the floating-point instruction set. Figure 10-7 illustrates binary word fixed-point format and Figure 10-8 illustrates binary long fixed-point format; Table 10-9 lists the binary fixed-point format fields.

Field assignments of the binary fixed-point format are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>sign bit</td>
</tr>
<tr>
<td>integer</td>
<td>integer value (2’s complement)</td>
</tr>
</tbody>
</table>
10.5 Floating-Point Instruction Set Summary

Each instruction is 32 bits long, and aligned on a word boundary. This section describes the overview of instructions for floating-point unit. A detailed description of each instruction is provided in Appendix D.

10.5.1 Load, Store and Move Instructions (Table 10-10)

Load and Store instructions move data between memory and FPU general purpose registers (FGR), and Move instructions move data directly between CPU and FPU general purpose registers (FGR). These instructions are not perform format conversions and therefore never cause floating-point exceptions. The instruction immediately following a load can use the contents of the loaded register. However, in such case the hardware interlocks, requiring additional real cycles. Thus, the scheduling of load delay slots is required to avoid the interlocking.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWC1</td>
<td>Load Word to FPU (coprocessor 1)</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SWC1</td>
<td>Store Word from FPU (coprocessor 1)</td>
<td>MIPS I</td>
</tr>
<tr>
<td>MTC1</td>
<td>Move Word to FPU (coprocessor 1)</td>
<td>MIPS I</td>
</tr>
<tr>
<td>MFC1</td>
<td>Move Word from FPU (coprocessor 1)</td>
<td>MIPS I</td>
</tr>
<tr>
<td>CTC1</td>
<td>Move Control Word to FPU (coprocessor 1)</td>
<td>MIPS I</td>
</tr>
<tr>
<td>CFC1</td>
<td>Move Control Word from FPU (coprocessor 1)</td>
<td>MIPS I</td>
</tr>
<tr>
<td>LDC1</td>
<td>Load Doubleword to FPU (coprocessor 1)</td>
<td>MIPS II</td>
</tr>
<tr>
<td>SDC1</td>
<td>Store Doubleword from FPU (coprocessor 1)</td>
<td>MIPS II</td>
</tr>
<tr>
<td>DMTC1</td>
<td>Move Doubleword to FPU (coprocessor 1)</td>
<td>MIPS III</td>
</tr>
<tr>
<td>DMFC1</td>
<td>Move Doubleword from FPU (coprocessor 1)</td>
<td>MIPS III</td>
</tr>
</tbody>
</table>
10.5.2 Conversion Instructions (Table 10-11)

Conversion instructions perform conversion operations between the various data formats.

Table 10-11. FPU Instruction Set(Optional): Conversion Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVT.S.fmt</td>
<td>Floating-Point Convert to Single FP Format</td>
<td>MIPS I</td>
</tr>
<tr>
<td>CVT.W.fmt</td>
<td>Floating-Point Convert to Word Fixed-Point Format</td>
<td>MIPS I</td>
</tr>
<tr>
<td>CVT.D.fmt</td>
<td>Floating-Point Convert to Double FP Format</td>
<td>MIPS I</td>
</tr>
<tr>
<td>ROUND.W.fmt</td>
<td>Floating-point Round to Word Fixed-Point</td>
<td>MIPS II</td>
</tr>
<tr>
<td>TRUNC.W.fmt</td>
<td>Floating-point Truncate to Word Fixed-Point</td>
<td>MIPS II</td>
</tr>
<tr>
<td>CEIL.W.fmt</td>
<td>Floating-point Ceiling Convert to Word Fixed-Point</td>
<td>MIPS II</td>
</tr>
<tr>
<td>FLOOR.W.fmt</td>
<td>Floating-point Floor Convert to Word Fixed-Point</td>
<td>MIPS II</td>
</tr>
<tr>
<td>CVT.L.fmt</td>
<td>Floating-Point Convert to Long Fixed-Point Format</td>
<td>MIPS III</td>
</tr>
<tr>
<td>ROUND.L.fmt</td>
<td>Floating-point Round to Long Fixed-Point</td>
<td>MIPS III</td>
</tr>
<tr>
<td>TRUNC.L.fmt</td>
<td>Floating-point Truncate to Long Fixed-Point</td>
<td>MIPS III</td>
</tr>
<tr>
<td>CEIL.L.fmt</td>
<td>Floating-point Ceiling Convert to Long Fixed-Point</td>
<td>MIPS III</td>
</tr>
<tr>
<td>FLOOR.L.fmt</td>
<td>Floating-point Floor Convert to Long Fixed-Point</td>
<td>MIPS III</td>
</tr>
</tbody>
</table>

10.5.3 Computational Instructions (Table 10-12)

Computational instructions perform arithmetic operations on floating-point values in the FPU registers. These are two categories of computational instructions:

- 3-Operand Register-Type instructions, which perform floating-point addition, subtraction multiplication, and division operations
- 2-Operand Register-Type instructions, which perform floating-point absolute value, move, negate, and square root operations.

Table 10-12. FPU Instruction Set(Optional): Computational Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD.fmt</td>
<td>Floating-point Add</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SUB.fmt</td>
<td>Floating-point Subtract</td>
<td>MIPS I</td>
</tr>
<tr>
<td>MUL.fmt</td>
<td>Floating-point Multiply</td>
<td>MIPS I</td>
</tr>
<tr>
<td>DIV.fmt</td>
<td>Floating-point Divide</td>
<td>MIPS I</td>
</tr>
<tr>
<td>ABS.fmt</td>
<td>Floating-point Absolute Value</td>
<td>MIPS I</td>
</tr>
<tr>
<td>MOV.fmt</td>
<td>Floating-point Move</td>
<td>MIPS I</td>
</tr>
<tr>
<td>NEG.fmt</td>
<td>Floating-point Negate</td>
<td>MIPS I</td>
</tr>
<tr>
<td>SQRT.fmt</td>
<td>Floating-point Square root</td>
<td>MIPS II</td>
</tr>
</tbody>
</table>
10.5.4 Compare and Branch Instructions (Table 10-13)

Compare instructions perform comparisons of the contents of registers and set a conditional bit based on the results. Branch on FPU Condition instructions perform a branch to the specified target if the specified coprocessor condition is met.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.cond.fmt</td>
<td>Floating-point Compare</td>
<td>MIPS I</td>
</tr>
<tr>
<td>BC1T</td>
<td>Branch on FPU True</td>
<td>MIPS I</td>
</tr>
<tr>
<td>BC1F</td>
<td>Branch on FPU False</td>
<td>MIPS I</td>
</tr>
</tbody>
</table>
11. Floating-Point Exception (Option)

This chapter describes FPU floating-point exceptions, including FPU exception types, exception trap processing, exception flags, saving and restoring state when handling an exception, and trap handlers for IEEE Standard 754 exceptions.

A floating-point exception occurs whenever the FPU cannot handle either the operands or the results of a floating-point operation in its normal way. The FPU responds by generating an exception to initiate a software trap or by setting a status flag.
11.1 Introduction

This chapter describes floating-point exceptions, including FPU exception type, exception trap processing, exception flags, saving and restoring state when handling an exception, and trap handlers for IEEE Standard 754 exceptions.

11.2 Exception Types

The FP Control/Status register described in Chapter 10 contains an Enable bit for each exception type; exception Enable bits determine whether an exception will cause the FPU to initiate a trap or set a status flag.

- If a trap is taken, the FPU remains in the state found at the beginning of the operation and a software exception handling routine executes.
- If no trap is taken, an appropriate value is written into the FPU destination register and execution continues.

The FPU supports the five IEEE Standard 754 exceptions:

- Inexact (I)
- Underflow (U)
- Overflow (O)
- Division by Zero (Z)
- Invalid Operation (V)

Cause bits, Enables, and Flag bits (status flags) are used.

The FPU adds a sixth exception type, Unimplemented Operation (E). This exception indicates the use of a software implementation. The Unimplemented Operation exception has no Enable or Flag bit; whenever this exception occurs, an unimplemented exception trap is taken.

Figure 11-1 shows the Control/Status register bits that support exceptions.

Figure 11-1. Control/Status Register Exception/Flag/Trap/Enable Bits
11.3 Exception Trap Processing

When a floating-point exception trap is taken, the Cause register indicates the floating-point coprocessor is the cause of the exception trap.

The Floating-Point Exception (FPE) code is used, and the Cause bits of the floating-point Control/Status register indicate the reason for the floating-point exception. These bits are, in effect, an extension of the system coprocessor Cause register.

11.4 Flags

A Flag bit is provided for each IEEE exception. This Flag bit is set to a 1 on the assertion of its corresponding exception, without corresponding exception trap signaled.

The Flag bit is reset by writing a new value into the Status register; flags can be saved and restored by software either individually or as a group.

When no exception trap is signaled, floating-point coprocessor takes a default action, providing a substitute value for the exception-causing result of the floating-point operation. The particular default action taken depends upon the type of exception. Table 11-1 lists the default action taken by the FPU for each of the IEEE exceptions.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Rounding Mode</th>
<th>Default action</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Inexact exception</td>
<td>Any</td>
<td>Supply a rounded result</td>
</tr>
<tr>
<td>U</td>
<td>Underflow exception</td>
<td>RN</td>
<td>Modify underflow values to 0 with the sign of the intermediate result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RZ</td>
<td>Modify underflow values to 0 with the sign of the intermediate result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RP</td>
<td>Modify positive underflows to the format’s smallest positive finite number; modify negative underflows to −0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RM</td>
<td>Modify negative underflows to the format’s smallest negative finite number; modify positive underflows to 0.</td>
</tr>
<tr>
<td>O</td>
<td>Overflow exception</td>
<td>RN</td>
<td>Modify overflow values to +∞ with the sign of the intermediate result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RZ</td>
<td>Modify overflow values to the format’s largest finite number with the sign of the intermediate result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RP</td>
<td>Modify negative overflows to the format’s most negative finite number; modify positive overflows to +∞</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RM</td>
<td>Modify positive overflows to the format’s largest finite number; modify negative overflows to −∞</td>
</tr>
<tr>
<td>Z</td>
<td>Division by zero</td>
<td>Any</td>
<td>Supply a properly signed +∞</td>
</tr>
<tr>
<td>V</td>
<td>Invalid operation</td>
<td>Any</td>
<td>Supply 231−1 result (Word Fixed-Point); Supply 267−1 result (Long Fixed-Point); Otherwise supply a quiet Not a Number</td>
</tr>
</tbody>
</table>
The FPU detects the eight exception causes internally. When the FPU encounters one of these unusual situations, it causes either an IEEE exception or an Unimplemented Operation exception (E).

Table 11-2 lists the exception-causing situations and contrasts the behavior of the FPU with the requirements of the IEEE Standard 754.

<table>
<thead>
<tr>
<th>FPA Internal Result</th>
<th>IEEE Standard 754</th>
<th>Trap Enable</th>
<th>Trap Disable</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inexact result</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>Loss of accuracy</td>
</tr>
<tr>
<td>Exponent overflow</td>
<td>O, I (*1)</td>
<td>O, I</td>
<td>O, I</td>
<td>Normalized exponent &gt; $E_{\text{max}}$</td>
</tr>
<tr>
<td>Division by zero</td>
<td>Z</td>
<td>Z</td>
<td>Z</td>
<td>Zero is (exponent=$E_{\text{min}}-1$, mantissa=0)</td>
</tr>
<tr>
<td>Overflow on convert to Integer</td>
<td>V</td>
<td>V (*2)</td>
<td>V (*2)</td>
<td>Source out of integer range, $\infty$, NaN</td>
</tr>
<tr>
<td>Signaling NaN source</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>Invalid operation</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>0/0, etc.</td>
</tr>
<tr>
<td>Exponent underflow</td>
<td>U</td>
<td>E</td>
<td>E</td>
<td>Normalized exponent &lt; $E_{\text{min}}$</td>
</tr>
<tr>
<td>Denormalized or QNaN</td>
<td>None</td>
<td>E</td>
<td>E</td>
<td>Denormalized is (exponent=$E_{\text{min}}-1$ and mantissa &lt;&gt; 0)</td>
</tr>
</tbody>
</table>

(*1) The IEEE Standard 754 specifies an inexact exception on overflow only if the overflow trap is disabled.

(*2) Some implementations such as TX49 trap as (E) and SW support is required. In TX79 implementation there is NO SW support required.

(*3) Exponent underflow sets the U and I Cause bits if both the U and I Enable bits are not set and the FS bit is set; otherwise exponent underflow sets the E Cause bit.
11.5 FPU Exceptions

The following sections describe the conditions that cause the FPU to generate each of its exceptions, and details the FPU response to each exception-causing condition.

**Inexact Exception (I)**

The FPU generates the Inexact exception if one of the following occurs:

- the rounded result of an operation is not exact, or
- the rounded result of an operation overflows, or
- the rounded result of an operation underflows and both the Underflow and Inexact Enable bits are not set and the FS bit is set.

**Trap Enabled Results:** If Inexact exception traps are enabled, the result register is not modified and the source registers are preserved.

**Trap Disabled Results:** The rounded or overflowed result is delivered to the destination register if no other software trap occurs.
Invalid Operation Exception (V)

Floating-Point format operation

The Invalid Operation exception is signaled if one or both of the operands are invalid for an implemented operation. When the exception occurs without a trap, the MIPS ISA defines the result as a quiet Not a Number (QNaN) for Floating-Point format. The invalid operations are:

- Addition or subtraction: magnitude subtraction of infinities, such as: \((+\infty) + (-\infty)\) or \((-\infty) - (-\infty)\)
- Multiplication: 0 times \(\infty\), with any signs
- Division: 0/0, or \(\infty/\infty\), with any signs
- Comparison of predicates involving ‘<’ or ‘>’ without ‘?’, when the operands are unordered*
- Any arithmetic operation, when one or both operands is a signaling NaN. A move (MOV) operation is not considered to be an arithmetic operation, but absolute value (ABS) and negate (NEG) are considered to be arithmetic operations.
- Comparison or Conversion From Floating-point Format on a signaling NaN.
- Square root: \(\sqrt{x}\), where \(x\) is less than zero.

Software can simulate the Invalid Operation exception for other operations that are invalid for the given source operands. Examples of these operations include IEEE Standard 754-specified functions implemented in software, such as Remainder: \(x \text{ REM } y\), where \(y\) is 0 or \(x\) is infinite; conversion of a floating-point number to a decimal format whose value causes an overflow, is infinity, or is NaN; and transcendental functions, such as \(\ln(-5)\) or \(\cos^{-1}(3)\). Refer to Appendix D for examples or for routines to handle these cases.

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: A quiet NaN is delivered to the destination register if no other software trap occurs.

Conversion to Integer format

The Invalid Operation exception is also raised when the source operand is an Infinity (\(\infty\)) or NaN, or the correctly rounded integer result is outside of the representable range.

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disable Results: The result value \(2^{31} - 1\) (for Word Fixed-Point) or \(2^{63} - 1\) (for Long Fixed-Point) is delivered to the destination register if no other software trap occurs.

* `<`, `>` and `?` are the notation in IEEE std 754. `?` means ‘unordered.’ See Compare instruction in Appendix D.
Division-by-Zero Exception (Z)

The Division-by-Zero exception is signaled on an implemented divide operation if the divisor is zero and the dividend is a finite nonzero number. Software can simulate this exception for other operations that produce a signed infinity, such as \( \ln(0) \), \( \sec(\pi/2) \), \( \csc(0) \), or \( 0^{-1} \)

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: The result, when no trap occurs, is a correctly signed infinity.

Overflow Exception (O)

The Overflow exception is signaled when the magnitude of the rounded floating-point result, with an unbounded exponent range, is larger than the largest finite number of the destination format. (This exception also signals an Inexact exception.)

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: The result, when no trap occurs, is determined by the rounding mode and the sign of the intermediate result (see Table 11-3).

<table>
<thead>
<tr>
<th>Denormalized Result</th>
<th>Flushed result</th>
<th>Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive</td>
<td>+∞</td>
<td>+Emax</td>
</tr>
<tr>
<td>Negative</td>
<td>−∞</td>
<td>−Emax</td>
</tr>
</tbody>
</table>

Underflow Exception (U)

Two related events contribute to the Underflow exception:

- creation of a tiny nonzero result between \( \pm 2^{\text{emin}} \) which can cause some later exception because it is so tiny
- extraordinary loss of accuracy during the approximation of such tiny numbers by denormalized numbers.

IEEE Standard 754 allows a variety of ways to detect these events, but requires they be detected the same way for all operations.

Tininess can be detected by one of the following methods:

- after rounding (when a nonzero result, computed as though the exponent range were unbounded, would lie strictly between \( \pm 2^{\text{emin}} \))
- before rounding (when a nonzero result, computed as though the exponent range and the precision were unbounded, would lie strictly between \( \pm 2^{\text{emin}} \)).

The MIPS architecture requires that tininess be detected after rounding.

Loss of accuracy can be detected by one of the following methods:
denormalization loss (when the delivered result differs from what would have been computed if the exponent range were unbounded)

inexact result (when the delivered result differs from what would have been computed if the exponent range and precision were both unbounded).

The MIPS architecture requires that loss of accuracy be detected as an inexact result.

Trap Enabled Results: If Underflow or Inexact traps are enabled, or if the FS bit is not set, then an Unimplemented exception (E) is generated, and the result register is not modified and the source registers are preserved.

Trap Disabled Results: If Underflow and Inexact traps are not enabled and the FS bit is set, the result is determined by the rounding mode and the sign of the intermediate result (See Table 10-4).

**Unimplemented Instruction Exception (E)**

Any attempt to execute an instruction with an operation code or format code that has been reserved for future definition sets the *Unimplemented* bit in the *Cause* field in the FPU Control/Status register and traps. The operand and destination registers remain undisturbed and the instruction is emulated in software. Any of the IEEE Standard 754 exceptions can arise from the emulated operation, and these exceptions are simulated.

The Unimplemented Instruction exception can also be signaled when unusual operands or result conditions are detected that the implemented hardware cannot handle properly. These include:

- Denormalized operand, except for Compare instruction
- Quiet Not a Number operand, except for Compare instruction
- Denormalized result or Underflow, when either Underflow or Inexact *Enable* bit is set or the FS bit is not set.
- Reserved opcodes
- Unimplemented formats
- Operations which are invalid for their format (for instance, CVT.S.S)

**NOTE:** Denormalized and NaN operands are only trapped if the instruction is a convert or a computational operation. A move operation does not trap if their operands are either denormalized or NaNs.

The use of this exception for such conditions is optional; most of these conditions are newly developed and are not expected to be widely used in early implementations. Loopholes are provided in the architecture so that these conditions can be implemented with assistance provided by software, maintaining full compatibility with the IEEE Standard 754.

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: This trap cannot be disabled.
11.6 Saving and Restoring State

Sixteen doubleword\(^1\) coprocessor load or store operations save or restore the coprocessor floating-point register state in memory. The remainder of control and status information can be saved or restored through CFC1/CTC1 instructions, and saving and restoring the processor registers. Normally, the Control/Status register is saved first and restored last.

When state is restored, state information in the Control/Status register indicates the exceptions that are pending. Writing a zero value to the Cause field of Control/Status register clears all pending exceptions, permitting normal processing to restart after the floating-point register state is restored.

11.7 Trap Handlers for IEEE Standard 754 Exceptions

The IEEE Standard 754 strongly recommends that users be allowed to specify a trap handler for any of the five standard exceptions so that a software subroutine can return a value to be used in stead of the exceptional operation's result; the trap handler can either compute or specify a substitute result to be placed in the destination register of the operation.

By retrieving an instruction using the processor Exception Program Counter (EPC) register, the trap handler determines:

- exceptions occurred during the operation
- the operation being performed
- the destination format

On Overflow or Underflow exceptions (except for conversions), and on Inexact exceptions, the trap handler gains access to the correctly rounded result by decoding source register field of the instruction code and simulating the operation in software.

On Overflow or Underflow exceptions caused by a floating-point conversion, on Invalid Operation and on Division-by-Zero exceptions, the trap handler gains access to the operand values by decoding the source register field of the instruction code.

The IEEE Standard 754 recommends that, if enabled, the overflow and underflow traps take precedence over a separate inexact trap. This prioritization is accomplished in software; hardware sets the bits for both the Inexact exception and the Overflow or Underflow exception.

\(^1\) 32 doublewords if the FR bit is set to 1.
This chapter describes the trace functions present on the C790.

The C790 supports real-time PC tracing. Pipeline status, target addresses of indirect jumps, and exception vectors are made available on special signals. The executed instruction sequence can be restored from signals and the source program.

The C790 also supports hardware breakpoints. The breakpoint facility is described in Chapter 13.
12.1 Real-Time PC Tracing

Trace information and non-sequential Program Counters are made available on special signal lines of the CPU.

The following trace information is made available:

- Instruction being executed in pipeline 0
- Instruction being executed in pipeline 1
- Current execution status (Normal (sequential), Branch Taken, Jump Target, Exception Target)

For Indirect jumps, the target address is also made available. For exception vectors, a code for the exception vector address is made available.

12.1.1 Classification of Branch and Jump Instructions

In this chapter, branches and jumps are classified into three categories which are direct jump, indirect jump and branch in order to explain the function of PC trace. The classification is shown in Table 12-1.

<table>
<thead>
<tr>
<th>Class</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump</td>
<td>Direct or Indirect Jump</td>
</tr>
<tr>
<td>Direct Jump</td>
<td>J or JAL Instruction</td>
</tr>
<tr>
<td>Indirect Jump</td>
<td>JR, JALR or ERET Instruction</td>
</tr>
<tr>
<td>Branch</td>
<td>Any of conditional branch Instruction</td>
</tr>
</tbody>
</table>
12.1.2 PC Trace Signals

All PC trace signals operate at half the C790 CPU clock frequency using the BUSCLK clock signal. Because of the half frequency operation there are pairs of signals which indicate the status of execution within the CPU pipelines. Phase A signals show the status corresponding to the even CPU clock cycle and Phase B signals show the status corresponding to the odd CPU clock cycle.

As can be seen from the following figure the execution status of the CPU pipeline during time 0 (all time references are in relation to the CPU clock) is put on the phase A signals at the next rising edge of BUSCLK during time 2. Similarly the execution status of the CPU pipeline during time 1 is put on the phase B signals.

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

The following signals are made available for real-time PC tracing.

- P0EXEA* (Phase A Pipeline 0 Execution Status) Output
- P1EXEA* (Phase A Pipeline 1 Execution Status) Output
- JMPA* (Phase A Jump) Output
- P0EXEB* (Phase B Pipeline 0 Execution Status) Output
- P1EXEB* (Phase B Pipeline 1 Execution Status) Output
- JMPB* (Phase B Jump) Output
- TPCE* (Target PC Enable) Output
- TPC[3:0] (Target PC Bus) Output

(1) P0EXEA* (Phase A Pipeline 0 Execution Status) Output

P0EXEA indicates whether an instruction has completed execution without generating an exception (retired) via Pipeline 0 during phase A.

- 0: An instruction was retired.
- 1: No instruction was retired.
(2) P1EXEA*  (Phase A Pipeline 1 Execution Status)  Output
P1EXEA indicates whether an instruction retired via Pipeline 1 during phase A. Note if this signal is asserted at the same time as P0EXEA* then two instructions were retired simultaneously during phase A via pipelines 0 and 1 but there is no indication as to which specific instruction was retired via which pipeline.

0: An instruction was retired.
1: No instruction was retired.

(3) JMPA*  (Jump Phase A)  Output
A jump was retired during phase A or a conditional branch instruction was retired and the branch was taken during phase A. Note that exceptions do not assert this signal.

0: Jump or conditional branch instruction was retired.
1: No jump or conditional branch instruction was retired.

(4) P0EXEB*  (Phase B Pipeline 0 Execution Status)  Output
P0EXEB indicates whether an instruction retired via Pipeline 0 during phase B.

0: An instruction was retired.
1: No instruction was retired.

(5) P1EXEB*  (Phase B Pipeline 1 Execution Status)  Output
P1EXEB indicates whether an instruction retired via Pipeline 1 during phase B. Note if this signal is asserted at the same time as P0EXEB* then two instructions were retired simultaneously during phase B via pipelines 0 and 1 but there is no indication as to which specific instruction was retired via which pipeline.

0: An instruction was retired.
1: No instruction was retired.

(6) JMPB*  (Jump Phase B)  Output
A jump was retired during phase B or a conditional branch instruction was retired and the branch was taken during phase B. Note that exceptions do not assert this signal.

0: Jump or conditional branch instruction was retired.
1: No jump or conditional branch instruction was retired.
(7) **TPCE**

**Target PC Enable**

Output

When this signal is asserted the TPC bus indicates the type of target PC that will be made available.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TPC bus indicates type of target PC.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>TPC bus has either the target PC or the exception vector address code or has no information.</td>
<td></td>
</tr>
</tbody>
</table>

The normal sequence of operation for the TPCE* and the TPC[3:0] signals is as follows: First TPCE* is asserted and simultaneously TPC[3:0] contains information about the type of the target PC (non-sequential PC). Next TPCE* is deasserted and either the target PC for indirect jumps is made available on the TPC[3:0] bus or for exceptions an exception vector address code is made available on the TPC[3:0] bus.

(8) **TPC[3:0]**

**Target PC**

Output

TPC[3:0] either indicates the type of the target PC address or the target address of indirect jump instructions or exception vector address codes.

**TPC[3:0] when TPCE* is asserted**

When TPCE* is asserted the type of the target PC address is made available on TPC[3:0]. Each bit of TPC[3:0] indicates a different type and multiple bits can be active at the same time.

- **TPC[0]: Jump Target during Phase A**
  
  When this signal is asserted it indicates that the target instruction of an Indirect Jump instruction (includes JR, JALR and ERET) is retired during Phase A. The target address is made available on TPC[3:0] in the next cycle if neither TPC[2] or TPC[3] are asserted simultaneously with this signal.

- **TPC[1]: Exception Target during Phase A**
  
  When this signal is asserted it indicates that the first instruction of an exception handler is retired during Phase A. The exception vector address is made available on TPC[3:0] in the next cycle if neither TPC[2] nor TPC[3] are asserted simultaneously with this signal.

- **TPC[2]: Jump Target during Phase B**
  
  When this signal is asserted it indicates that the target instruction of an Indirect Jump instruction is retired during Phase B. The target address is made available on TPC[3:0] in the next cycle.

- **TPC[3]: Exception Target during Phase B**
  
  When this signal is asserted it indicates that the first instruction of an exception handler is retired during Phase B. The exception vector address is made available on TPC[3:0] in the next cycle.
**TPC[3:0] when TPCE* is deasserted**

When TPCE* is not asserted TPC[3:0] can be carrying the following three type of information:

1. There is no meaningful information on TPC. This happens most of the time when the program is executing sequentially.
2. The target address is made available because in the previous cycle TPCE* was asserted and TPC[0] or TPC[2] were equal to 0. The target address starts with the least significant four bits of the target instruction address (bits[5:2]).
3. An exception vector address code is made available because in the previous cycle TPCE* was asserted and TPC[1] or TPC[3] were equal to 0. The exception vector address code are shown in Table 12-2.

<table>
<thead>
<tr>
<th>Exception</th>
<th>STATUS.BEV</th>
<th>STATUS.DEV</th>
<th>STATUS.EXL</th>
<th>Vector Address</th>
<th>Code (TPC[3:0])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset, NMI</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0xBFC0 0000</td>
<td>8 (1000)</td>
</tr>
<tr>
<td>TLB Miss</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>0xBFC0 0200</td>
<td>12 (1100)</td>
</tr>
<tr>
<td>TLB Miss</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>0x8000 0000</td>
<td>0 (0000)</td>
</tr>
<tr>
<td>TLB Miss</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>0xBFC0 0380</td>
<td>15 (1111)</td>
</tr>
<tr>
<td>TLB Miss</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>0x8000 0180</td>
<td>3 (0011)</td>
</tr>
<tr>
<td>Debug &amp; SIO</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>0xBFC0 0300</td>
<td>14 (1110)</td>
</tr>
<tr>
<td>Debug &amp; SIO</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>0x8000 0100</td>
<td>2 (0010)</td>
</tr>
<tr>
<td>Performance Counter</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>0xBFC0 0280</td>
<td>13 (1101)</td>
</tr>
<tr>
<td>Performance Counter</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>0x8000 0080</td>
<td>1 (0001)</td>
</tr>
<tr>
<td>Interrupt</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>0xBFC0 0400</td>
<td>9 (1001)</td>
</tr>
<tr>
<td>Interrupt</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0x8000 0200</td>
<td>4 (0100)</td>
</tr>
<tr>
<td>Common</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>0xBFC0 0380</td>
<td>15 (1111)</td>
</tr>
<tr>
<td>Common</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0x8000 0180</td>
<td>3 (0011)</td>
</tr>
</tbody>
</table>

Table 12-2. Exception Vector Address Codes
12.1.3 Priority of Target Addresses

The target address for an indirect jump instruction or an exception vector address code is made available on TPC[3:0]. For an indirect jump instruction it takes multiple cycles (8 BUSCLK cycles or 16 CPU clock cycles) for the complete target address to be made available on the TPC[3:0] bus. As such multiple conditions can occur simultaneously and there are certain priorities associated with putting out the target address. The rules governing what is made available on the TPC[3:0] bus are listed below:

1. If a new indirect jump instruction is retired while the target address PC for a previous indirect instruction is still being put out on TPC[3:0], the new indirect jump instruction's target PC will be signaled and start coming out on the TPC[3:0] bus and the previous target PC output will be terminated.

2. If an exception is taken while the target address PC for a previous indirect instruction is still being put out on TPC[3:0], the exception vector address code will be signaled and start coming out on the TPC[3:0] bus and the previous target PC output will be terminated.

The rules are also described in the following flowchart.

---

**Figure 12-1. Priority of Outputting Jump or Exception Target**
12.1.4 Examples of PC Tracing

The following sections contain examples of program execution and the corresponding waveforms of the PC trace signals. Note that when two instructions are retired simultaneously, just for the sake of illustration, it is indicated which instruction is executed in which pipeline. In reality, in this case, it is not known which instruction is retired from which pipeline.
12.1.4.1 Sequential Execution

This is an example of sequential program execution. The program fragment is as follows:

mul
add
sub
lw r1
add
sub ,,r1
add
add

The PC trace signals for the program fragment are shown below:

Figure 12-2. Waveform for Sequential Execution
12.1.4.2 Conditional Branch

This is an example of program with conditional branch instructions. Both the branch taken and not taken case is illustrated. The program fragment is as follows:

```
add
add
beq L0  # Not Taken
lw
add
beq L1  # Taken
add
....
```

```
L1:
add
beq L2  # Taken
sll
....
L2:
sub
sub
```

The PC trace signals for the program fragment are shown below:

```
Phase  | A | B | A | B | A | B | A | B | A | B |
CPUCLK       |   |   |   |   |   |   |   |   |   |   |
BUSCLK       |   |   |   |   |   |   |   |   |   |   |
Pipe 0       | add | add | add | - | - | add | bne | sub |
Pipe 1       | - | beq | lw | - | beq | add | sll | sub |
```

```
P0EXEA* add add bne |
P1EXEA* lw beq sll |
P0EXEB* add add sub |
P1EXEB* beq add sub |
JMPA* beq bne |
JMPB* |
TPCE* |
TPC[3:0]        |
```

Figure 12-3. Waveform for Conditional Branch
12.1.4.3 Indirect Jump (Target in Phase A)

This is an example of a program with an indirect jump instruction which is retired during phase B. The program fragment is as follows:

```
add
add
jr   L1
lw
....
L1: xor
add
ori
ori
sw
sll
sub
sub
```

The PC trace signals for the program fragment are shown below:

```
<table>
<thead>
<tr>
<th>Phase</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUCLK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUSCLK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pipe 0</td>
<td>add</td>
<td>add</td>
<td>–</td>
<td>–</td>
<td>xor</td>
<td>ori</td>
<td>sll</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pipe 1</td>
<td>–</td>
<td>jr</td>
<td>lw</td>
<td>–</td>
<td>add</td>
<td>ori</td>
<td>sw</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P0EXEA*</td>
<td>add</td>
<td>xor</td>
<td>sll</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P0EXEB*</td>
<td>add</td>
<td>ori</td>
<td>sub</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1EXEA*</td>
<td>lw</td>
<td>add</td>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1EXEB*</td>
<td>jr</td>
<td>ori</td>
<td>sub</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMPA*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMPB*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TPCE*</td>
<td></td>
<td>xor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TPC[3:0]</td>
<td></td>
<td>TA[5:2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TA[x:y] = Target address bit x to y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 12-4. Waveform for Indirect Jump (Target in Phase A)
12.1.4.4 Indirect Jump (Target in Phase B)

This is an example of a program with an indirect jump instruction which is retired during phase A. The program fragment is as follows:

```
add
add
jr L1
lw
....
L1:
oxr
add
ori
ori
sw
sll
sub
sub
```

The PC trace signals for the program fragment are shown below:

```
<table>
<thead>
<tr>
<th>Phase</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUCLK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUSCLK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pipe 0</td>
<td>add</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Ori</td>
<td>sll</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pipe 1</td>
<td>jr</td>
<td>lw</td>
<td>-</td>
<td>xor</td>
<td>add</td>
<td>ori</td>
<td>sw</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
P0EXEA* add sll
P1EXEA* jr add sw
P0EXEB* ori sub
P1EXEB* lw xor ori sub
JMPA* jr
JMPB*
TPCE*
xor
```

![Figure 12-5. Waveform for Indirect Jump (Target in Phase B)](image)
12.1.4.5 Indirect Jump (During Target PC Output)

This is an example of a program with two indirect jump instructions. While the target address PC associated with the first indirect jump instruction is being put out the second indirect jump instruction is retired. Thus the first target PC output is terminated and the second target PC output is signaled and then made available. The program fragment is as follows:

```
add
data
jr    L1
lw
....
L1:
xor
add
ejr   L2
add
....
L2
sw
sll
sub
sub
```

The PC trace signals for the program fragment are shown below:

```
Phase  | A | B | A | B | A | B | A | B | A | B |
CPUCLK |   |   |   |   |   |   |   |   |   |   |
BUSCLK |   |   |   |   |   |   |   |   |   |   |
Pipe 0  | add | add | - | - | Target | xor | jr | - | - | Target | sll | sub |
P0EXEA* |     | add | xor | sll |
P1EXEA* |     | lw  | add | sw |
P0EXEB* |     | add | jr  | sub |
P1EXEB* |     | jr  | add | sub |
JMPA*   |     |     |     |     |
JMPB*   |     | jr  | jr  |     |
TPCE*   |     |     |     |     |
```

Figure 12-6. Waveform for Indirect Jump (During Target PC Output)
12.1.4.6 Exception (Target in Phase B)

This is an example of a program which generates an exception. The target instruction (first instruction of the exception handler) retires in phase B. The program fragment is shown below. The label *ExHnd* identifies the first instruction of the exception handler.

```
add
add
add
lw
teq # Generates exception
....
ExHnd: xor
      add
      sw
      sll
      sub
      sub
```

The PC trace signals for the program fragment are shown below:

```
More stall cycles might be inserted.

<table>
<thead>
<tr>
<th>Phase</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUCLK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUSCLK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipe 0:
- add  add  -  -  -  xor  sll  sub
Pipe 1:
- -  add  lw  -  -  add  sw  sub
```

```
P0EXEA*:
  add  sll

P1EXEA*:
  lw  sw

P0EXEB*:
  add  xor  sub

P1EXEB*:
  add  add  sub

JMPA*

JMPB*

TPCE*:
  xor

TPC[3:0]:
  0111  E.Code
```

Figure 12-7. Waveform for Exception (Target in Phase B)
12.1.4.7 Exception (During Target PC Output)

This is an example of a program which generates an exception while a target PC from an earlier indirect jump instruction is being made available. The target PC output is terminated and the exception vector address code is signaled and then made available. The target instruction (first instruction of the exception handler) retires in phase B. The program fragment is shown below. The label ExHnd identifies the first instruction of the exception handler.

```
add
add
add
lw
teq  # Generates exception
....
ExHnd: xor
add
sw
sll
sub
sub
```

The PC trace signals for the program fragment are shown below:

![PC Trace Signals](image)

More stall cycles might be inserted.

Figure 12-8. Waveform for Exception (During Target PC Output)
12.1.4.8 Exception Generated by Branch or Jump Instruction

This is an example of a program in which an indirect jump instruction generates an exception. As such the program jumps to the exception handler and the only thing indicated is the exception vector address code and not the jump. The target instruction (first instruction of the exception handler) retires in phase B. The program fragment is shown below. The label ExHnd identifies the first instruction of the exception handler.

```
add
add
add
lw
jr # Generates an exception
nop # Branch delay slot
....
ExHnd: xor
add
sw
sll
sub
sub
```

The PC trace signals for the program fragment are shown below:

```
More stall cycles might be inserted.

<table>
<thead>
<tr>
<th>Phase</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUCLK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUSCLK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipe 0
| add | add | - | - | - | xor | sll | sub |
Pipe 1
| - | add | lw | - | - | add | sw | sub |

P0EXEA* add sll
P1EXEA* lw sw
P0EXEB* add xor sub
P1EXEB* add add sub
JMPA* 
JMPB* 
TPCE* xor
TPC[3:0] 0111 E.Code
```

E.Code = Exception Vector Code

Figure 12-9. Waveform for Exception Generated by Branch or Jump Instruction
12.1.4.9 Exception Generated by Branch Delay Slot Instruction

This is an example of a program in which the branch delay slot instruction generates an exception. As such the program jumps to the exception handler and the only thing indicated is the exception vector address code and not the jump. The target instruction (first instruction of the exception handler) retires in phase B. The program fragment is shown below. The label ExHnd identifies the first instruction of the exception handler.

```
add
add
add
lw
jr
lw      # Generates an exception
....
ExHnd: xor
add
sw
sll
sub
sub
```

The PC trace signals for the program fragment are shown below:

![PC Trace Waveform]

More stall cycles might be inserted.

Figure 12-10. Waveform for Exception Generated by Branch Delay Slot Instruction
12.1.4.10 Exception Generated by Target Instruction

This is an example of a program in which the target instruction of an indirect jump generates an exception. As such the program jumps to the exception handler and the only thing indicated is the exception vector address code and not the jump. The target instruction (first instruction of the exception handler) retires in phase B. The program fragment is shown below. The label ExHnd identifies the first instruction of the exception handler.

```
add
add
add
lw
jr
nop
....
L1: lw  # Generates an exception
     and
     ....
ExHnd: xor
       add
       sw
       sll
       sub
       sub
```

The PC trace signals for the program fragment are shown below:

![PC trace signals](image)

Figure 12-11. Waveform for Exception Generated by Target Instruction
12.1.4.11 Back to Back Exceptions (Case I)

This is an example of a program in which two back to back exceptions are generated. The program jumps to the first exception handler but then immediately jumps to the second exception handler. The target instruction (first instruction of the second exception handler) retires in phase A. The exception vector address code for the first handler is never made available. The program fragment is shown below. The label ExHnd1 identifies the first instruction of the first exception handler and the label ExHnd2 identifies the first instruction of the second exception handler.

```
add
add
ExHnd1: xor
....
# Generates the first exception
xor
....
ExHnd2: sw
sll
sub
sub
```

The PC trace signals for the program fragment are shown below:

```
E.Code
A
BABABA
B
Phase
CPUCLK
BUSCLK
Pipe 0
add
Pipe 1
- - - - - - - - - - - - - - - - sw
P0EXEA*
add
sll
P1EXEA*
sw
P0EXEB*
sub
P1EXEB*
sub
JMPA*
JMPB*
TPCE*
sw
TPC[3:0]
1101 E.Code
```

More stall cycles might be inserted.

Figure 12-12. Waveform for Back to Back Exceptions (Case I)
12.1.4.12 Back to Back Exceptions (Case II)

This is an example of a program in which two (all most) back to back exceptions are generated. The program jumps to the first exception handler and then generates an exception when executing the second instruction of the exception handler. It then jumps to the second exception handler. The target instruction (first instruction of the first exception handler) retires in phase A. As compared to the case discussed above the exception vector address code for the both the handlers are made available. The program fragment is shown below. The label ExHnd1 identifies the first instruction of the first exception handler and the label ExHnd2 identifies the first instruction of the second exception handler.

```assembly
add
add # Generates the first exception
....
ExHnd1: xor
xor # Generates the second exception
....
ExHnd2: sw
sll
sub
sub
```

The PC trace signals for the program fragment are shown below:

```
A  B  A  B  A  B  A  B  A  B  A  B

Pipe 0 | add | - | - | - | xor | - | - | - | sll | sub |
Pipe 1 | - | - | - | - | - | - | - | sw | sub |
```

More stall cycles might be inserted.

```
P0EXEA* add xor sll
P1EXEA* sw
P0EXEB* sub
P1EXEB* sub
JMPA* ...
JMPB* ...
TPCE* xor sw
```

E.Code = Exception Vector Code

Figure 12-13. Waveform for Back to Back Exceptions (Case II)
13. Hardware Breakpoint

This chapter describes hardware breakpoint functions for debugging present on the C790.
13.1 Hardware Breakpoint

C790 provides hardware breakpoint mechanism for debugging purpose. (In this section, hardware breakpoint is sometimes referred to as “breakpoint”.) This function allows users to set an instruction breakpoint and a data address/value breakpoint with signaling the breakpoint event occurrence to external probe. The following summarizes the features of the breakpoint function.

- Provides both instruction and data breakpointing in virtual address.
- Instruction address breakpoint with address masking.
- Data breakpoint with masking. Data breakpoint can be set by the following events:
  - Address with masking
  - Value with masking
  - Read/write
- Independent exception event control for instruction and data.
- Individual event control by processor operating mode/exception level.
- Provides a trigger signal to external probes synchronized with the breakpointing event.

Hardware breakpointing is implemented as a part of Coprocessor 0. Configuring the breakpoint is done by setting 7 Breakpoint registers by special MTC0/MFC0 instructions. Figure 13-1 shows the basic structure of the breakpoint hardware.

Breakpoint can generate breakpoint exception which is categorized in Level2 exception, and has a dedicated exception vector. (See 5. Exception) This exception is only masked in Level2 mode, and exception generation itself can be controlled by the Breakpoint Control Register mentioned in the following section. Note that some of breakpoint exceptions are imprecise, for instance, setting value breakpoint for load instruction is basically imprecise because the load instruction may retire from the pipeline before actual acquisition of memory contents. The following summarizes imprecise cases:

- All data value breakpoint on load instruction
- Data value breakpoint on SWC1 instruction

13.1.1 Hardware Breakpoint signal

To signal a breakpoint occurrence, the C790 activates a signal called TRIG, whenever a trigger condition is met.

- TRIG (Trigger Output) Output

This signal is asserted for two BUSCLK cycles when a trigger condition is met.
13.2 Breakpoint Registers

Hardware breakpoint is comprised of 3 pairs of breakpoint registers and one control register listed below. Each of breakpoint register pair includes one breakpoint value register and one breakpoint mask register.

- **Breakpoint Control Register (BPC)**
- Instruction Address Breakpoint Registers
  - Instruction Address Breakpoint Register (IAB)
  - Instruction Address Breakpoint Mask Register (IABM)
- Data Address Breakpoint Registers
  - Data Address Breakpoint Register (DAB)
  - Data Address Breakpoint Mask Register (DABM)
- Data Value Breakpoint Registers
  - Data Value Breakpoint Register (DVB)
  - Data Value Breakpoint Mask Register (DVBM)
All 7 registers are 32-bit read/write and assigned to Coprocessor0 register 24. Therefore, C790 provides extended MTC0 instructions for accessing these registers and it is necessary to use these instructions to access these registers instead of the conventional MTC0/MFC0 instructions. Table 13-1 and Table 13-2 summarizes the instructions for accessing the registers.

Table 13-1. Set a new value into breakpoint registers

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTBPC</td>
<td>Move to Breakpoint Control Register</td>
</tr>
<tr>
<td>MTIAB</td>
<td>Move to Instruction Address Breakpoint Register</td>
</tr>
<tr>
<td>MTIABM</td>
<td>Move to Instruction Address Breakpoint Mask Register</td>
</tr>
<tr>
<td>MTDAB</td>
<td>Move to Data Address Breakpoint Register</td>
</tr>
<tr>
<td>MTDABM</td>
<td>Move to Data Address Breakpoint Mask Register</td>
</tr>
<tr>
<td>MTDVB</td>
<td>Move to Data Value Breakpoint Register</td>
</tr>
<tr>
<td>MTDVBM</td>
<td>Move to Data Value Breakpoint Mask Register</td>
</tr>
</tbody>
</table>

Table 13-2. Get the value from breakpoint registers

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFBPC</td>
<td>Move from Breakpoint Control Register</td>
</tr>
<tr>
<td>MFIAB</td>
<td>Move from Instruction Address Breakpoint Register</td>
</tr>
<tr>
<td>MFIABM</td>
<td>Move from Instruction Address Breakpoint Mask Register</td>
</tr>
<tr>
<td>MFDAB</td>
<td>Move from Data Address Breakpoint Register</td>
</tr>
<tr>
<td>MFDABM</td>
<td>Move from Data Address Breakpoint Mask Register</td>
</tr>
<tr>
<td>MFDVB</td>
<td>Move from Data Value Breakpoint Register</td>
</tr>
<tr>
<td>MFDVBM</td>
<td>Move from Data Value Breakpoint Mask Register</td>
</tr>
</tbody>
</table>

13.2.1 Breakpoint Control Register (BPC)

The BPC register contains enable bits and status bits for controlling the breakpointing of both instruction and data. This register consists of 5 parts of bit fields:

- **Breakpoint overall control** (bit [31:28])
  These bits control the operation mode of the breakpointing.

- **Instruction breakpoint control** (bit [26:23])
  These bits specify the processor mode that the instruction breakpoint is enabled.

- **Data breakpoint control** (bit[21:18])
  These bits specify the processor mode that the data breakpoint is enabled.

- **Signaling Control** (bit[17:15])
  These bits control the occurrence of breakpoint exception / trigger generation upon the breakpoint event.

- **Breakpoint Status** (bit[2:0])
  These bits indicate the type of breakpoint event. This part is used to identify which breakpoint event occurred in the breakpoint exception handler.
The following shows the detailed bitmap of BPC register.

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAE</td>
<td>31</td>
<td>Instruction Enable. This bit enables/disables instruction address breakpointing.</td>
<td>Read / Write</td>
<td>0</td>
</tr>
<tr>
<td>DRE</td>
<td>30</td>
<td>Data Read Enable. This bit enables data load address breakpointing.</td>
<td>Read / Write</td>
<td>0</td>
</tr>
<tr>
<td>DWE</td>
<td>29</td>
<td>Data Write Enable. This bit enables data store address breakpointing.</td>
<td>Read / Write</td>
<td>0</td>
</tr>
<tr>
<td>DVE</td>
<td>28</td>
<td>Data Value Enable. This bit is valid only when DRE and/or DWE are set to 1. When DVE is set to 1 data read breakpoints (DRE == 1) are further qualified by the value of the data read, and data write breakpoints (DWE == 1) are further qualified by the value of the data written. Note that data value breakpoints for data reads are imprecise. See section 13.1 (&quot;Hardware Breakpoint&quot;) for more details.</td>
<td>Read / Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>rsvd</td>
<td>27</td>
<td>Reserved - must be written as zeros by software. The processor returns zeros in these bit positions when read.</td>
<td>Read</td>
<td>0</td>
</tr>
<tr>
<td>IUE</td>
<td>26</td>
<td>Instruction break - User Enable. This bit enables instruction address breakpointing in (standard) user mode. This bit is only valid if IAE is set to 1.</td>
<td>Read / Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>ISE</td>
<td>25</td>
<td>Instruction break - Supervisor Enable. This bit enables instruction address breakpointing in Supervisor mode. This bit is only valid if IAE is set to 1.</td>
<td>Read / Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>IKE</td>
<td>24</td>
<td>Instruction break - Kernel Enable. This bit enables instruction address breakpointing in non-exception kernel mode - i.e. when both STATUS.EXL and STATUS.ERL are 0. This bit is only valid if IAE is set to 1.</td>
<td>Read / Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>IXE</td>
<td>23</td>
<td>Instruction break - EXL mode Enable. This bit enables instruction address breakpointing in exception kernel mode - i.e. when STATUS.EXL is 1 and STATUS.ERL is 0. This bit is only valid if IAE is set to 1.</td>
<td>Read / Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>rsvd</td>
<td>22</td>
<td>Reserved - must be written as zeros by software. The processor returns zeros in these bit positions when read.</td>
<td>Read</td>
<td>0</td>
</tr>
</tbody>
</table>
### Chapter 13 Hardware Breakpoint

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUE</td>
<td>21</td>
<td>Data break - User Enable. This bit enables data breakpointing in User mode. This bit is only valid if DWE or DRE is set to 1. 0: disable data breakpointing in User mode 1: enable data breakpointing in User mode</td>
<td>Read / Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>DSE</td>
<td>20</td>
<td>Data break - Supervisor Enable. This bit enables data breakpointing in Supervisor mode. This bit is only valid if DWE or DRE is set to 1. 0: disable data breakpointing in Supervisor mode 1: enable data breakpointing in Supervisor mode</td>
<td>Read / Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>DKE</td>
<td>19</td>
<td>Data break - Kernel Enable. This bit enables data breakpointing in Kernel mode - i.e. when both STATUS.EXL and STATUS.ERL are 0. This bit is only valid if DWE or DRE is set to 1. 0: disable data breakpointing in Kernel mode 1: enable data breakpointing in Kernel mode</td>
<td>Read / Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>DXE</td>
<td>18</td>
<td>Data break - EXL mode Enable. This bit enables data breakpointing in Exception Kernel mode - i.e. when STATUS.EXL is 1 and STATUS.ERL is 0. This bit is only valid if at least one of DRE or DWE are set to 1. 0: disable data breakpointing in EXL mode 1: enable data breakpointing in EXL mode</td>
<td>Read / Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>ITE</td>
<td>17</td>
<td>Instruction Trigger Enable. This bit enables the generation of the trigger signal when an instruction breakpoint occurs. 0: disable instruction breakpoint trigger 1: enable instruction breakpoint trigger</td>
<td>Read / Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>DTE</td>
<td>16</td>
<td>Data Trigger Enable. This bit enables the generation of the trigger signal when an data breakpoint occurs. 0: disable data breakpoint trigger 1: enable data breakpoint trigger</td>
<td>Read / Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>BED</td>
<td>15</td>
<td>Breakpoint Exception Disable. This bit disables the entry into the debug exception handler. Note that the setting of this bit does not affect trigger signal generation. 0: enable entry into debug exception handler 1: disable entry into debug exception handler</td>
<td>Read / Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>rsvd</td>
<td>14 - 3</td>
<td>Reserved - must be written as zeros by software. The processor returns zeros in these bit positions when read.</td>
<td>Read</td>
<td>0</td>
</tr>
<tr>
<td>DWB</td>
<td>2</td>
<td>Data Write Breakpoint. This status bit indicates whether a data breakpoint has occurred on a write or not. 0: no data breakpoint has occurred on a write 1: data breakpoint has occurred on a write</td>
<td>Read / Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>DRB</td>
<td>1</td>
<td>Data Read Breakpoint. This status bit indicates whether a data breakpoint has occurred on a read or not. 0: no data breakpoint has occurred on a read 1: data breakpoint has occurred on a read</td>
<td>Read / Write</td>
<td>Undefined</td>
</tr>
<tr>
<td>IAB</td>
<td>0</td>
<td>Instruction Address Breakpoint. This status bit indicates whether an instruction address breakpoint has occurred or not. 0: no instruction address breakpoint has occurred on a read 1: instruction address breakpoint has occurred on a read</td>
<td>Read / Write</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
13.2.2 Instruction Address Breakpoint Register (IAB) / Instruction Address Breakpoint Mask Register (IABM)

This register pair holds the instruction breakpointing address. Both the value in IAB register and the current fetch PC are masked by the value in IABM. If the values are equal, condition for instruction address breakpoint becomes true. As fetch PC is always word-aligned, the bit 0 and bit 1 of these registers are fixed to zeros.

13.2.3 Data Address Breakpoint Register (DAB) / Data Address Breakpoint Mask Register (DABM)

This register pair holds the data breakpointing address. Both the value in DAB register and the destination for load/store operation are masked by the value in DABM. If the values are equal, condition for data address breakpoint becomes true. These registers are 32-bit wide readable/writable.
13.2.4 Data Value Breakpoint Register (DVB) / Data Value Breakpoint Mask Register (DVBM)

This register pair holds the value for data value breakpointing. Both the value in DVB and the lower 32 bits of load/store data are masked with the value in DVBM. If the values are equal, condition for data value breakpoint becomes true. Note that enabling data value breakpoint implies activating the data address breakpointing (setting either/both of DRE/DWE bit in BPC), and therefore breakpoint event for data value only happens if both condition for data address breakpoint and data value breakpoint becomes true. Note that the comparison of data value is always performed in 32bit regardless of the width of load/store operation: the store value comes from GPR is truncated to 32bit value for comparison and the load value is appropriately signextended or merged with the contents of GPR (unaligned cases) and then the least significant 32-bits are used for comparison. For instance, most significant (64+32) bits/32-bits are truncated on data value comparison for LQ/SQ/LD/SD instructions, while the value from memory is sign-extended to comprise a 32bit value for LB/LH instructions.

13.3 Setting Breakpoint

The following sections mention the details of breakpoint controls with some sample codes. As C790 is a pipelined superscalar processor, several restrictions are applied in setting breakpoint registers. The following is the main topic that has to be taken care of:

- Upon changing the configuration of breakpointing, it is very likely that 3 or more registers must be updated. However, the change is performed in pipelined manner as C790 is pipelined processor. This potentially has possibility to create a hazardous area in generating exception unconsciously.
- C790 does NOT wait for the data arrival on load operation. The instruction itself may retire from the pipeline before storing the data into the registers, and the occurrence of breakpointing event delays from the instruction completion. This not only make some data value breakpoints imprecise, but also temporally masks an occurrence of breakpointing event as following case: a data load instruction that should cause data value breakpoint exception results in cache miss. But in the next cycle, other level2 exception such as SIO interrupt had been detected and the processor entered level2 before the acquisition of the data. Under this scenario, data value exception will be delayed until the processor returns from Level2 mode.
13.3.1 Sequence of Setting Breakpoint

In order to prevent spurious exception during reconfiguring the breakpoint, managing breakpointing enable before and after the change is mandatory. One easy way is to change the processor mode into Level2 to mask breakpoint exception unconditionally, but, this has an side effect that the user segment becomes unmapped. Therefore, this section mainly focuses on changing the configuration without changing the processor mode. The following summarizes the sequence of changing breakpointing configuration.

1. Synchronize the pipeline
2. Disable the breakpoint exception that is going to be reconfigured
3. Synchronize the pipeline
4. Set appropriate data in Breakpoint register pairs
5. Set appropriate configuration into Breakpoint Control Register, including enabling the breakpoint exception.
6. Synchronize the pipeline

There are three synchronization points in the sequence: the first one is to ensure that there is no pending breakpoint exception for consistency in the breakpoint exception handler. The second one is right after disabling the breakpoint that is going to be reconfigured. This separates the change in the control register from the change for other breakpoint register so that programmer can safely change the breakpoint. The third synchronization is after updating breakpoint control register. Since C790 issues the instructions in in-ordered manner, changes for breakpoint register pair always precedes the change in the control register. In this sense, there is no spurious exception without this synchronization. However, in order to catch the breakpointing event right after updating the control register, flushing the pipeline at this point is strongly recommended. The first synchronized operation must be either of SYNC.P or SYNC.L operation depending on the breakpoint that is going to be reconfigured. If it is instruction breakpoint, SYNC.P is to be used and otherwise SYNC.L is to be used. For second and third synchronization, SYNC.P is to be used.

The flow generating TRIG* and exception is shown in Figure 13-8, Figure 13-9, Figure 13-10. Figure 13-8 describes the flow hardware breakpoint encounters the breakpointing event. Figure 13-9, and Figure 13-10 describe the flow how the exception and TRIG* signal is asserted.

The following shows some simple sample codes for configuring breakpoint registers. Several programming notes/issues are put in the comments.
Figure 13-8. Hardware Breakpoint detection flow (Setting)
Figure 13-9. Hardware Breakpoint detection flow (IAB)
Figure 13-10. Hardware Breakpoint detection flow (DAB/DVB) (1/2)
Figure 13-10. Hardware Breakpoint detection flow (IAB) (2/2)
13.3.2 Instruction Breakpointing

The following code sets an instruction breakpoint from 0x1234_5600 to 0x1234_56ff, and traps if the processor is either in user mode or in supervisor mode.

```assembly
# Setting Instruction address breakpoint from 0x1234_5600 to 0x1234_56ff
# in user mode and supervisor mode

# 1st sync.
sync.p                   # A barrier to ensure there is no pending instruction address breakpoint in pipe.
# pipeline flushing works for this purpose.

# At first, disable instruction breakpointing to avoid spurious exceptions.
# The following uses conservative way not to break the configuration for data breakpointing.
# mfbpc $4                # get the value in BPC
bgez $4, 1f            # skip following if ( BPC[31] == 0 )
nop                      # (bds)
li     $5, (1 << 31)     # IAE is in 31st bit of BPC
xor    $4, $5, $4        # Resetting IAE bit to zero.
mtbpc  $4                # reload BPC.

# 2nd sync.
sync.p                   # barrier to ensure the configuration change of breakpoint function
1:

# Reconfigure instruction breakpoint address.
# Note that least significant 8 bits can be anything because it is masked by IABM register anyway
li     $4, 0x12345678
mtiab  $4

# Setting mask register. Masked if corresponding bit in mask register is reset to zero.
li     $5, 0xffffffff
mtiabm $5

# Reconfigure instruction breakpoint. For better understanding, once resetting all the bits for instruction breakpoint, and then sets new config.
mfbpc $4

# Reset IUE/ISE/IKE/ITE/IAB. Especially resetting IAB is important to know the cause of next breakpoint exception correctly.
li      $5, ~( 1 << 26 )    # IUE    
       | ( 1 << 25 )    # ISE    
       | ( 1 << 24 )    # IKE    
       | ( 1 << 23 )    # IXE    
       | ( 1 << 17 )    # ITE    
       | ( 1 <<  0 )    # IAB
and    $4, $5, $5

# Set new configuration to BPC register.
# Note that setting BPC after IAB/IABM is so important to avoid spurious exception.
```

---
li $6, $6,
  
  ( 1 << 31 ) # IAE = 1 to enable Inst. B.P.
  | ( 1 << 26 ) # IUE = 1 to enable Inst. B.P in user mode.
  | ( 1 << 20 ) # IUE = 1 to enable Inst. B.P in superv. mode.
  | ( 1 << 15 ) # BED = 1 to enable generating exception.
)
or $5, $4, $6
mtbpc $5

# 3rd sync.
Sync.p # Barrier to ensure the configuration change
------------------------------------------------------------------
13.3.3 Data Address Breakpointing

The following code sets a data address breakpoint from 0x1230_0000 to 0x1233_ffff for both reading and writing, and traps if the processor is either in kernel mode (including under level 1).

------------------------------------------------------------------
# Setting data address breakpoint from 0x1230_0000 to 0x1233_ffff
# in kernel (normal, L1) mode
# 1st sync.
sync.l                    # A barrier to ensure there is no pending
# data address breakpoint in pipe.
# Must flush all buffers for load/store for this
# purpose by SYNC.L
#
# At first, reset data-breakpoint related bits to zeros.
# Resetting DWB/DRB is important so that the handler can recognize the
# next breakpoint exception correctly.
#
mbpc  $4                # load current configuration
li     $5, ~(                         
( 1 << 30 )    # DRE  
( 1 << 29 )    # DWE  
( 1 << 28 )    # DVE  
( 1 << 21 )    # DUE  
( 1 << 20 )    # DSE  
( 1 << 19 )    # DKE  
( 1 << 18 )    # DXE  
( 1 << 16 )    # DTE  
( 1 << 2 )    # DWB  
( 1 << 1 )    # DRB  
)
and    $4, $4, $5
mtbpc  $4                # reload BPC.
#
# 2nd sync.
sync.p                   # barrier to ensure the configuration change
# of breakpoint function
#
# Reconfigure data breakpoint address.
# Note that least significant 18 bits can be anything because it is masked
# by DABM register anyway
#
li     $6, 0x12305678
mtdab  $6
#
# Setting mask register. Masked if corresponding bit in mask register
# is reset to zero.
#
li     $5, 0xffffc0000
mtdabm $5
#
# Set new configuration to BPC register.
# Note that setting BPC after DAB/DABM is so important to avoid spurious
# exception.
#
li     $6, $6,                                                           
| ( 1 << 30 ) # DRE = 1 to enable Data B.P on read          
| ( 1 << 29 ) # DWE = 1 to enable Data B.P on write         
| ( 1 << 19 ) # DKE = 1 to enable Data B.P in kern. mode.   
| ( 1 << 15 ) # BED = 1 to enable generating exception.     
| ( 1 << 2 )  # DWB = 1 to enable B.P under L1.             
| ( 1 << 1 )  # DRB = 1 to enable generating exception.     
)                                           
or     $5, $4, $6           # Note that $4 still holds the value used
# on MTBPC.
mtbpc  $5
# 3rd sync.
sync.p  # Barrier to ensure the configuration change
13.3.4 Breakpointing by Data Address and Value

Setting Data Address and Value breakpoint is the same as Data Address breakpoint. The following example is the same as the previous example except in that the trap only happens if the data contains 0xCAFE in least significant 16 bits, and traps only on loading data.

```
# Setting data address/value breakpoint from 0x1230_0000 to 0x1233_ffff
# with data that contains 0xCAFE in kernel(normal, L1) mode.
# 1st sync.
sync.l                      # A barrier to ensure there is no pending
# data address breakpoint in pipe.
# Must flush all buffers for load/store for this
# purpose by SYNC.L

# At first, reset data-breakpoint related bits to zeros.
# Resetting DWB/DRB is important so that the hander can recognize the
# next breakpoint exception correctly.
mfbpcl $4                   # load current configuration
li   $5, ~(                  |
    ( 1 << 30 )    # DRE  |
    ( 1 << 29 )    # DWE  |
    ( 1 << 28 )    # DVE  |
    ( 1 << 21 )    # DUE  |
    ( 1 << 20 )    # DSE  |
    ( 1 << 19 )    # DKE  |
    ( 1 << 18 )    # DXE  |
    ( 1 << 16 )    # DTE  |
    ( 1 << 2 )    # DWE  |
    ( 1 << 1 )    # DRB  |
)                           |
and  $4, $4, $5             # reload BPC.

# 2nd sync.
sync.p                      # barrier to ensure the configuration change
# of breakpoint function

# Reconfigure data breakpoint address.
# Note that least significant 18 bits can be anything because it is masked
# by DABM register anyway
li   $6, 0x1233ffff          |
mtdb $6                      |

# Setting mask register. Masked if corresponding bit in mask register
# is reset to zero.
li   $5, 0xfffc0000           |
mtdbm $5                      |

# Configure data value address.
# Note that least significant 8 bits can be anything because it is masked
# by DVBM register anyway
li   $6, 0xbabecafe           |
mtdv $6                        |

# Setting mask register. Masked if corresponding bit in mask register
# is reset to zero.
li   $5, 0x0000000000000000 |
mtdvbm $5                      |
```

# Set new configuration to BPC register.
# Note that setting BPC after DAB/DABM is so important to avoid spurious
# exception.
#
li     $6,  \
     ( 1 << 30 ) # DRE = 1 to enable Data B.P on read  \
     ( 1 << 28 ) # DVE = 1 to enable Data value B.P    \
     ( 1 << 19 ) # DKE = 1 to enable Data B.P in kern. mode.  \
     ( 1 << 18 ) # DXE = 1 to enable Data B.P under L1.   \
     ( 1 << 15 ) # BED = 1 to enable generating exception. \
) or     $5, $4, $6            # Note that $4 still holds the value used \
     # on MTBPC.
mtbpc  $5

# 3rd sync.
sync.p  # Barrier to ensure the configuration change

13.3.5  Data Value Breakpointing

Data value breakpoint can be configured so that it traps only by data value, by setting
zero to \textit{DABM} register and configuring the data breakpoint to “Data Address and Value”
mode.
13.4 Triggering External Probes

There is one dedicated pad to make breakpoint visible outside of C790. This pad, TRIG* signal, is asserted for two cycles whenever break point event is detected. This trigger signal generation is enabled by setting ITE/DTE bit in BPC register to 1. Note that assertion of TRIG* signal is not completely synchronized with the occurrence of exception: TRIG signal is directly connected to the internal breakpoint detect logic while exception including breakpoint always occurs along with retirement of instruction. Therefore, timing of the assertion of TRIG* signal and that of occurrence of exception may differs. Especially, if the breakpoint is detected right before entering Level2 mode, and if the breakpoint exception is taken imprecisely, exception may be masked because of processor’s mode change although TRIG* signal has already been asserted.

13.5 Important notice on using hardware breakpoint

One important issue not mentioned in this section is that breakpointing does not take care of ASID on detecting breakpoint. This implies not only that software has to take care of it on context switching to apply breakpointing for a specific process, but also that imprecise breakpoint exception may be detected after or in the middle of context switching. In such condition, it may become difficult to identify which process the breakpoint exception belongs to. This can be avoided by executing SYNC.L instruction right before changing ASID. (Since all imprecise breakpoint events relates to load/store instructions, executing SYNC.L works as a barrier).

Relating to this issue, as briefly described in section 13.3, issuing breakpoint exception may delay because of other level2 exception handling, although the breakpoint exception is actual precedent from instruction ordering point of view. In such condition, because C790 generates breakpoint exception after the processor returns from Level2, there is no possibility to miss encountering the breakpoint. However, if the program need to insure the order of occurrence between level2 exceptions, software has to take care of it (i.e. all level2 handler has to check the occurrence of breakpointing first). Similarly, if a level2 exception DOES NOT return to where the exception was detected, software has to insure to reset the condition of breakpoint.

---

1 C790 tracks the occurrence of breakpoint exception until the breakpoint exception is taken.
# INDEX

## A

<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>2-18, 11-6, D-4</td>
</tr>
<tr>
<td>ABS.fmt</td>
<td>3-21, 10-14, D-41</td>
</tr>
<tr>
<td>AbsoluteValue</td>
<td>D-4</td>
</tr>
<tr>
<td>ADD</td>
<td>2-18, 3-15, 5-26, A-11, A-141</td>
</tr>
<tr>
<td>ADD.fmt</td>
<td>D-5</td>
</tr>
<tr>
<td>ADDI</td>
<td>3-14, 5-26, A-12, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>ADDIU</td>
<td>3-14, A-12, A-13, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>ADDU</td>
<td>3-15, A-11, A-14, A-141</td>
</tr>
<tr>
<td>AdEL</td>
<td>4-20, 5-8, 5-15</td>
</tr>
<tr>
<td>AdES</td>
<td>4-20, 5-8, 5-15</td>
</tr>
<tr>
<td>AGNT</td>
<td>8-5, 8-11, 8-14, 8-15</td>
</tr>
<tr>
<td>ALU</td>
<td>2-3, 2-10, 2-11, 2-12, 2-13, 3-14</td>
</tr>
<tr>
<td>ANDI</td>
<td>3-14, A-16, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>arbiter</td>
<td>8-2, 8-14, 8-15</td>
</tr>
<tr>
<td>AREQ</td>
<td>8-11, 8-14, 8-15</td>
</tr>
<tr>
<td>ASID</td>
<td>2-15, 4-5, 4-8, 4-14, 5-16, 5-17, 5-18, 6-2, 6-3, 6-4, 6-9, 6-10, 6-12, 6-13, 6-16, 6-18, 13-20, C-38</td>
</tr>
<tr>
<td>Associativity</td>
<td>2-17</td>
</tr>
</tbody>
</table>

## B

<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>BadPAddr</td>
<td>2-15, 4-5, 4-17, 4-25, 5-19, 8-25</td>
</tr>
<tr>
<td>BadVAddr</td>
<td>2-15, 4-5, 4-9, 4-12, 5-15, 5-16, 5-17, 5-18</td>
</tr>
<tr>
<td>BadVPN2</td>
<td>4-9</td>
</tr>
<tr>
<td>BC0</td>
<td>C-41, C-42</td>
</tr>
<tr>
<td>BC0F</td>
<td>3-20, C-2, C-41, C-42</td>
</tr>
<tr>
<td>BC0FL</td>
<td>3-20, C-3, C-42</td>
</tr>
<tr>
<td>BC0T</td>
<td>3-20, C-4, C-42</td>
</tr>
<tr>
<td>BC0TL</td>
<td>3-20, C-5, C-42</td>
</tr>
<tr>
<td>BC1</td>
<td>D-40</td>
</tr>
<tr>
<td>BC1F</td>
<td>3-21, 10-15, D-6, D-8, D-40</td>
</tr>
<tr>
<td>BC1T</td>
<td>3-21, 10-15, D-7, D-8, D-40</td>
</tr>
<tr>
<td>BD2</td>
<td>4-19, 4-33, 5-5, 5-12, 5-13, 5-14, 5-25, 9-10</td>
</tr>
<tr>
<td>Instruction</td>
<td>Page Numbers</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>BdPAddr</td>
<td>4-25</td>
</tr>
<tr>
<td>BDS</td>
<td>4-29, 9-6, 9-8</td>
</tr>
<tr>
<td>BE</td>
<td>13-6, 13-15, 13-16, 13-19</td>
</tr>
<tr>
<td>BEM</td>
<td>3-17, A-17, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>BEQ</td>
<td>3-17, A-17, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>BEQL</td>
<td>3-17, A-18, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>BEV</td>
<td>4-16, 4-17, 5-7, 5-11, 5-12, 5-15, 5-16, 5-17, 5-18, 5-19, 5-20, 5-21, 5-22, 5-23, 5-24, 5-26, 5-27, 5-28, 12-6</td>
</tr>
<tr>
<td>BFH</td>
<td>C-6</td>
</tr>
<tr>
<td>BGEZ</td>
<td>3-18, A-19, A-142</td>
</tr>
<tr>
<td>BGEZAL</td>
<td>3-18, A-20, A-142</td>
</tr>
<tr>
<td>BGEZALL</td>
<td>3-18, A-21, A-142</td>
</tr>
<tr>
<td>BGEZL</td>
<td>3-18, A-22, A-142</td>
</tr>
<tr>
<td>BGTL</td>
<td>3-17, A-23, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>BGTLZL</td>
<td>3-17, A-24, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>BHINBT</td>
<td>C-6</td>
</tr>
<tr>
<td>BHT</td>
<td>1-2, 2-3, 2-6, 2-7, 4-31, C-10</td>
</tr>
<tr>
<td>BIU</td>
<td>2-4</td>
</tr>
<tr>
<td>BLEZ</td>
<td>3-17, A-25, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>BLEZL</td>
<td>3-17, A-26, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>BLTZ</td>
<td>3-18, A-27, A-142</td>
</tr>
<tr>
<td>BLTZAL</td>
<td>3-18, A-28, A-142</td>
</tr>
<tr>
<td>BLTZALL</td>
<td>3-18, A-29, A-142</td>
</tr>
<tr>
<td>BLTZL</td>
<td>3-18, A-30, A-142</td>
</tr>
<tr>
<td>BNE</td>
<td>3-17, A-31, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>BNEL</td>
<td>3-17, A-32, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>bootstrapping</td>
<td>5-11</td>
</tr>
<tr>
<td>BPC</td>
<td>4-26, 5-11, 13-3, 13-4, 13-5, 13-8, 13-14, 13-16, 13-18, 13-19, 13-20</td>
</tr>
<tr>
<td>BPE</td>
<td>4-23, 5-11, C-9</td>
</tr>
<tr>
<td>BR</td>
<td>2-3, 2-11, 2-12, 3-26</td>
</tr>
<tr>
<td>branch likely</td>
<td>2-13, 9-10</td>
</tr>
<tr>
<td>breakpoint</td>
<td>12-1, 13-5, 13-8, A-2</td>
</tr>
<tr>
<td>BTAC</td>
<td>1-2, 2-3, 2-6, 2-7, 4-29, 4-31, 9-6, 9-7, 9-8, C-6, C-7, C-9, C-10, C-11, C-13, C-28</td>
</tr>
<tr>
<td>BUSERR</td>
<td>5-19, 8-10, 8-25, 8-26, 8-27, 8-28, 8-29</td>
</tr>
<tr>
<td>BXLBT</td>
<td>C-6</td>
</tr>
</tbody>
</table>
BXSBT............................................................................................................................................................C-6
C.........................................................................................................................................................D-8
C.cond.D.........................................................................................................................................................3-21, 10-15, D-41
C.cond.fmt ...................................................................................................................................3-21, 10-15, D-41
C.cond.S.........................................................................................................................................................D-8
Cache.................................................................1-2, 2-1, 2-3, 2-6, 2-7, 2-15, 2-17, 2-18, 3-20, 4-5, 4-17, 4-29, 8-2, 8-8, 9-7, 9-9, A-6, A-7, C-6, C-7, C-8, C-9, C-13
CACHE ...............................................................................................................................2-11, 2-13, 2-17, 3-20, 4-17, 4-23, 4-31, 4-32, 5-19, A-141, B-163, C-6, C-7, C-8, C-9, C-10, C-11, C-12, C-13, C-41, D-40
CacheOp.........................................................................................................................................................C-7
CAUSE................................................................................................................................................. 8-13, 9-10
CCR .............................................................................................................................................. 9-2, 9-5, 9-10, 9-11, A-3
CE....................................................................................................................................... 4-19, 4-23, 5-2, 5-23
CEIL. .............................................................................................................................................................D-12
CEIL.L.fmt................................................................................................................................ 3-21, 10-14, D-41
CEIL.W..........................................................................................................................................................D-13
CEIL.W.fmt............................................................................................................................... 3-21, 10-14, D-41
CFC1.............................................................................................................................................. 3-21, 10-13, 11-9, D-14, D-40
CH........................................................................................................................................................ 4-16, 4-17
coherecy................................................................................................................................................... 2-18, 4-8, 4-24, 6-12, 6-16, 8-2
Coherency.............................................................................................................................................. 6-17
Config.............................................................................................................................................. 2-15, 4-5, 4-23, 5-11, 6-7, 6-12, C-9
CONFIG.............................................................................................................................................. 9-10, C-28
consistency ................................................................................................................................................... 13-9
Context.............................................................................................................................................. 2-15, 4-5, 4-9, 5-15, 5-16, 5-17, 5-18
contexts....................................................................................................................................................... 6-3
ConvertFmt ........................................................................................................................................ D-2, D-16, D-17, D-18, D-19, D-23, D-24
COP0 ......................................................................................................................................................... 2-7, 2-11, 2-12, 2-13, 2-15, 3-2, 3-20, 4-1, 4-5, 4-16, 4-17, 4-22, 4-28, 5-23, 6-1, 6-3, 6-14, 8-25, 9-2, 9-3, 9-11, A-4, A-141, A-142, B-163, C-1, C-7, C-9, C-10, C-11, C-12, C-14, C-15, C-17, C-18, C-19, C-20, C-21, C-22, C-23, C-24, C-25, C-26, C-27, C-28, C-29, C-30, C-31, C-32, C-33, C-34, C-35, C-36, C-41, C-42, D-40
COPI ......................................................................................................................................................... 2-3, 2-4, 2-7, 2-8, 2-10, 2-11, 2-12, 2-13, 2-14, 3-2, 3-21, 4-29, 9-6, 9-7, A-8, A-125, A-141, A-142, B-163, C-16, C-41, D-1, D-2, D-27, D-29, D-40, D-41
coprocessor ........................................................................................................................................ 2-4, 2-7, 2-8, 2-16, 3-5, 3-21, 4-16, 4-17, 5-11, 5-23, 6-1, 10-2, A-4, A-5, A-142, C-1, C-2, C-3, C-4, C-5, C-14, C-15, C-18, C-28, C-41, D-14, D-15, D-21, D-26
Coprocessor ........................................................................................................................................ 1-1, 1-5, 2-11, 2-15, 3-2, 3-5, 3-16, 3-20, 3-21, 4-1, 4-5, 4-16, 4-19, 4-20, 5-2, 5-8, 5-9, 5-10, 5-23, 6-1, 6-14, 8-10, 8-11, 13-2, A-3, A-4, A-5, A-8, A-141, A-142, C-1, C-2, C-3, C-4, C-5, C-7, C-16, C-17, C-18, C-19, C-20, C-21, C-22, C-23, C-24, C-25, C-26, C-27, C-28, C-29, C-30, C-31, C-32, C-33, C-34, C-35, C-36, C-37, C-38, C-39, C-40, D-4, D-5,

Coprocessor0 ............................................................................................................................................... 13-4
Count ............................................................................................................................................. 2-15, 3-25, 4-5, 4-13, 4-15, 5-24, B-4, B-5
counter............................................................................ 2-15, 2-16, 2-19, 3-17, 4-5, 4-17, 4-18, 4-19, 4-28, 4-30, 4-33, 5-5, 5-9, 5-13, 6-1, 9-1, 9-2, 9-3, 9-5, 9-6, 9-8, 9-10, 9-11, C-28, C-35
Counter ............................................................................ 2-3, 2-15, 2-19, 3-20, 4-1, 4-2, 4-3, 4-4, 4-5, 4-19, 4-21, 4-28, 4-29, 4-30, 5-2, 5-7, 5-8, 5-9, 5-10, 5-11, 5-13, 9-1, 9-2, 9-3, 9-4, 9-5, 9-6, 9-10, 9-11, 12-6, A-4, C-25, C-26, C-35
CPCOND ........................................................................................................................................................ A-3
CPCOND0 ............................................................................................................................................... 8-10, 8-11, C-2, C-3, C-4, C-5
CPR ..............................................................A-3, C-17, C-18, C-19, C-20, C-21, C-22, C-23, C-24, C-25, C-26, C-27, C-28, C-29, C-30, C-31, C-32, C-33, C-34, C-35, C-36

CPUADDR ........................................................................................................................................... 8-3, 8-7, 8-9
CPUASTART ....................................................................................... 8-3, 8-7, 8-8, 8-12, 8-13, 8-16, 8-19
CPUBE .............................................................................................................................................. 8-3, 8-7, 8-9
CPUCLK ........................................................................................................................................................ 8-11
CPUDATA .............................................................................................................................................. 8-3, 8-7, 8-9, 8-17, 8-20
CPUDSTART ............................................................... 8-3, 8-10, 8-12, 8-13, 8-16, 8-17, 8-19, 8-20, 8-26, 8-28
CPURD ............................................................................................................................................. 8-3, 8-8, 8-9
CPUTRANSTYPE ........................................................................................................................................... 8-8
CPUTSIZE ........................................................................................................................................... 8-3, 8-9, 8-12, 8-13, 8-16, 8-19
CPUWR ......................................................................................................................................... 8-3, 8-8, 8-9
CTC1 ......................................................................................... 3-21, 10-7, 10-8, 10-9, 10-13, 11-9, D-15, D-40
CTE ...................................................................................................................................................... 4-28, 4-29, 5-11, 9-2, 9-4, 9-5, 9-10, 9-11
CTR0 ........................................................................................................................................... 4-29, 9-10, 9-11
CTR1 ........................................................................................................................................... 4-29, 9-10, 9-11
CU......................................................................................................................................................... 1-5, 3-5, 3-20, 3-21, 4-16, 4-17, C-1, C-14, C-15
CU0......................................................................................................................................................... 5-23, C-7
CVT ...................................................................................................................................................... 3-26
CVTD ................................................................................................................................................... D-16
CVTD.fmt ........................................................................................................................................ 3-21, 10-14, D-41
CVTL ................................................................................................................................................... D-17
CVTL.fmt ........................................................................................................................................ 3-21, 10-14, D-41
CVTS ................................................................................................................................................... D-18
CVT.S.fmt ........................................................................................................................................ 3-21, 10-14, D-41
CVTW.fmt ........................................................................................................................................ 3-21, 10-14, D-41
CVTW.S ........................................................................................................................................... D-19

D
DAB............................................................................................................................................. 4-27, 13-3, 13-7, 13-12, 13-16, 13-19
TOSHIBA

Divide ........................................................ 1-1, 2-6, 3-14, 3-16, 3-21, 3-22, 3-23, 3-24, 3-26, 4-1, B-3, B-5, B-8
DIV .................................................................................................. 2-14, 3-23, 3-26, 4-2, B-3, B-9, B-163
DIVU ........................................................... 3-16, 3-26, A-40, A-141
DIVU1 .......................................................... 2-14, 3-23, 3-26, 4-2, B-3, B-9, B-163
Divide ........................................................ 1-1, 2-6, 3-14, 3-16, 3-21, 3-22, 3-23, 3-24, 3-26, 4-1, B-3, B-5, B-8
DIVU ........................................................... 3-16, 3-26, A-40, A-141
DIVU1 .......................................................... 2-14, 3-23, 3-26, 4-2, B-3, B-9, B-163
DKE ..................................................................................................... 13-6, 13-16, 13-18, 13-19
DMA .............................................................. 8-1, 8-3, 8-6, 8-7, 8-10, 8-12, 8-13, 8-14, 8-25, 8-26
DMAC .............................................................. 8-1, 8-3, 8-10, 8-11, 8-13, 8-14, 8-25, 8-26
DMFC1 ........................................................... 3-21, 10-13, D-21, D-40
DMTC1 ........................................................... 3-21, 10-13, D-22, D-40
DMULT .......................................................... 3-4, 3-14, A-142, B-165, C-42, D-41
DMULTU .......................................................... 3-4, 3-14, A-142, B-165, C-42, D-41

X-5
A-118, A-122, B-2, B-64, B-65, B-72, B-74, B-78, B-79, B-80, B-81, B-82, B-83, B-89, B-93, B-95, B-113, B-120, B-122, B-128, B-129, B-130

DRB ................................................................................................................................................................13-6, 13-16, 13-18
DRE ................................................................................................................................................................5-11, 13-5, 13-6, 13-8, 13-16, 13-18, 13-19
DSE .................................................................................................................................................................... 13-6, 13-16, 13-18
DSLL .................................................................................................................................................................. 3-15, A-41, A-141
DSLL32 .......................................................................................................................................................... 3-15, A-42, A-141
DSLLV .......................................................................................................................................................... 3-15, A-43, A-141
DSRA ............................................................................................................................................................. 3-15, A-44, A-141
DSRA32 ...................................................................................................................................................... 3-15, A-45, A-141
DSRL .............................................................................................................................................................. 3-15, A-46, A-141
DSRL32 ...................................................................................................................................................... 3-15, A-47, A-141
DSRLV .......................................................................................................................................................... 3-15, A-48, A-141
DSUB ............................................................................................................................................................. 3-15, 5-26, A-50, A-141
DTE ................................................................................................................................................................13-6, 13-16, 13-18
DTLB ................................................................................................................................................................. 2-3, 2-6, 2-16, 4-29, 9-6, 9-8
DUE ................................................................................................................................................................13-6, 13-16, 13-18
DVB ................................................................................................................................................................4-27, 13-3, 13-8, 13-12
DVM ................................................................................................................................................................4-27, 13-3, 13-8, 13-18
DVE ................................................................................................................................................................13-5, 13-16, 13-18, 13-19
DWB ................................................................................................................................................................13-6, 13-16, 13-18
DWE ................................................................................................................................................................5-11, 13-5, 13-6, 13-8, 13-16, 13-18
DXE ................................................................................................................................................................13-6, 13-16, 13-18, 13-19
DXIN .................................................................................................................................................................. C-6
DXLD ......................................................................................................................................................... C-6
DXLTG ............................................................................................................................................................ C-6
DXSDT ............................................................................................................................................................ C-6
DXSTG ............................................................................................................................................................ C-6
DXWBIN ........................................................................................................................................................ C-6

E

EC ....................................................................................................................................................................... 4-23
EDI .................................................................................................................................................................... 4-16, 4-17, 5-23, C-1, C-14, C-15
Edian .............................................................................................................................................................. 4-23
El ..................................................................................................................................................................... 3-20, 4-16, 4-17, 5-23, C-1, C-14, C-15, C-42
EIE .................................................................................................................................................................. 4-16, 4-17, 4-18, 5-24, C-14, C-15
endianess .................................................................................................................................................. 3-9
Endianness .............................................................................................................................................. 1-2, 3-5
EntryHi ......................................................................................................................................................... 2-15, 4-5, 4-14, 5-15, 5-16, 5-17, 5-18, 6-2, 6-3, 6-4, 6-15, C-28, C-37, C-38, C-39, C-40
EntryHi7 .......................................................................................................................................................... C-37
EntryLo ............................................................................................................................................................ 5-15, 5-16, 5-17, 5-18, 6-15, C-38, C-39, C-40
EntryHi .......................................................................................................................................................... 6-16
EntryLo ........................................................................................................................................................... 5-15, 5-16, 5-17, 5-18, 6-15, C-38, C-39, C-40
EPC................................................................. 2-6, 2-15, 4-5, 4-21, 4-33, 5-2, 5-3, 5-15, 5-16, 5-17, 5-18, 5-19, 5-20, 5-21, 5-22, 5-23,
7-26, 5-27, 11-9, C-16
ERE.............................................................. 2-11, 2-12, 2-13, 3-20, 4-4, 5-5, 5-24, 6-11, 9-7, 9-11, 12-2, 12-5, C-16, C-38, C-39, C-40, C-42
ERL.............................................................. 4-16, 4-17, 4-18, 5-5, 5-9, 5-11, 5-12, 5-13, 5-14, 5-19, 5-24, 5-25, 6-6, 6-7, 6-8, 6-9, 6-10,
6-11, 6-12, 9-2, 9-10, 9-11, 13-5, 13-6, C-14, C-15, C-16
ERL0 ............................................................................................................................................................... 9-5
ERL1 ............................................................................................................................................................... 9-5
Error ............................................................ 2-6, 2-15, 4-5, 4-12, 4-17, 4-18, 5-2, 5-10, 5-15, 5-19, 5-23, 6-6, 6-7, 6-9, 8-13, 8-25, 8-26,
D-37
ErrorEPC .............................................................. 4-33, 5-5, 5-12, 5-13, 5-14, 5-25, 9-10, 9-11, C-16
ErrorPC .................................................................................................................................................. 2-15, 4-5
EVENT ............................................................................................................................................................ 9-5
EVENT0 ....................................................................................................................................................4-28, 4-29, 9-2, 9-5, 9-6, 9-11
EVENT1 ....................................................................................................................................................4-28, 4-29, 9-5, 9-6, 9-11
EXC2 .................................................................................................................................................... 4-19, 5-5, 5-8, 5-11, 5-12, 5-13, 5-14, 5-25, 9-10
ExcCode .............................................. 2-15, 2-16, 2-18, 2-19, 3-2, 3-5, 3-16, 3-18, 3-20, 4-4, 4-5, 4-9, 4-12, 4-14, 4-16, 4-17, 4-18,
4-19, 4-20, 4-21, 4-29, 4-33, 5-1, 5-2, 5-3, 5-5, 5-8, 5-9, 5-10, 5-11, 5-12, 5-13, 5-14, 5-15,
5-16, 5-17, 5-18, 5-19, 5-20, 5-21, 5-22, 5-23, 5-24, 5-25, 5-26, 5-27, 6-1, 6-2, 6-4, 6-6,
6-9, 6-11, 6-14, 6-15, 6-16, 6-17, 6-20, 8-13, 8-25, 9-2, 9-7, 9-8, 9-9, 9-10, 9-11, 10-8, 11-2, 11-3,
12-1, 12-2, 12-3, 12-5, 12-6, 12-7, 12-14, 12-15, 12-16, 12-17, 12-18, 12-19, 12-20, 13-2,
B-14, B-22, B-23, B-25, B-27, B-28, B-66, B-67, B-68, B-70, B-71, B-84, B-86,
B-91, B-93, B-95, B-111, B-113, B-118, B-120, B-122, B-165, C-1, C-2, C-3, C-4, C-5, C-7,
C-8, C-16, C-17, C-18, C-19, C-20, C-21, C-22, C-23, C-24, C-25, C-26, C-27, C-28, C-29,
C-30, C-31, C-32, C-33, C-34, C-35, C-36, C-37, C-38, C-39, C-40, C-42, D-26, D-37, D-41
Exception ....................................................... 2-6, 2-11, 2-15, 2-19, 3-18, 3-20, 3-21, 4-5, 4-18, 4-20, 4-21, 5-1, 5-2, 5-3, 5-4, 5-5, 5-6, 5-7,
<table>
<thead>
<tr>
<th>Exceptions</th>
<th>11-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>execution pipeline</td>
<td>2-3, 2-5, 2-10, 2-11, 2-12, 3-26, C-16</td>
</tr>
<tr>
<td>ExHnd</td>
<td>12-14, 12-15, 12-16, 12-17, 12-18</td>
</tr>
<tr>
<td>ExHnd1</td>
<td>12-19, 12-20</td>
</tr>
<tr>
<td>ExHnd2</td>
<td>12-19, 12-20</td>
</tr>
<tr>
<td>EXL</td>
<td>4-16, 4-17, 4-18, 4-21, 4-29, 5-2, 5-5, 5-7, 5-9, 5-12, 5-16, 5-19, 5-24, 6-6, 6-8, 6-9, 6-10, 6-11, 6-12, 9-2, 12-6, 13-5, 13-6, C-14, C-15, C-16</td>
</tr>
<tr>
<td>EXL0</td>
<td>4-29, 9-2, 9-5, 9-11</td>
</tr>
<tr>
<td>EXL1</td>
<td>4-29, 9-5, 9-11</td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
<tr>
<td>FCR</td>
<td>D-14</td>
</tr>
<tr>
<td>FCR0</td>
<td>10-4</td>
</tr>
<tr>
<td>FCR31</td>
<td>10-4, 10-6, D-15</td>
</tr>
<tr>
<td>FCRs</td>
<td>10-4</td>
</tr>
<tr>
<td>FetchAddress</td>
<td>C-10, C-11</td>
</tr>
<tr>
<td>FGR</td>
<td>10-13</td>
</tr>
<tr>
<td>FGRs</td>
<td>10-2</td>
</tr>
<tr>
<td>FLOOR.L</td>
<td>D-23</td>
</tr>
<tr>
<td>FLOOR.L.fmt</td>
<td>3-21, 10-14, D-41</td>
</tr>
<tr>
<td>FLOOR.W</td>
<td>D-24</td>
</tr>
<tr>
<td>FLOOR.W.fmt</td>
<td>3-21, 10-14, D-41</td>
</tr>
<tr>
<td>FP_Control</td>
<td>D-14, D-15</td>
</tr>
<tr>
<td>FPE</td>
<td>4-20, 5-8, 5-28, 11-3</td>
</tr>
<tr>
<td>FPRs</td>
<td>10-2, D-10, D-16, D-17, D-28</td>
</tr>
<tr>
<td>FPU</td>
<td>1-2, 2-3, 2-7, 2-8, 2-14, 2-18, 4-16, 10-13, 10-14, 11-2, 11-5, 11-8, D-1, D-2, D-3, D-14, D-15, D-27, D-29</td>
</tr>
<tr>
<td>FR</td>
<td>4-16, 4-17, 10-2</td>
</tr>
<tr>
<td>funnel shift</td>
<td>2-3, 2-14, 4-1, 4-2, 4-4, B-17, B-20, B-21, B-22, B-161</td>
</tr>
<tr>
<td>Funnel shift</td>
<td>2-11</td>
</tr>
<tr>
<td>G</td>
<td></td>
</tr>
<tr>
<td>gathering</td>
<td>2-4, 2-19, 6-17, 9-1, A-8, A-125</td>
</tr>
<tr>
<td>General Purpose Registers</td>
<td>2-3, 4-1, 4-2, 4-3, 4-4, A-3</td>
</tr>
<tr>
<td>global bit</td>
<td>6-18</td>
</tr>
<tr>
<td>GPR</td>
<td>D-21</td>
</tr>
<tr>
<td>GPR10</td>
<td>B-21, B-22</td>
</tr>
<tr>
<td>Index</td>
<td>Page Numbers</td>
</tr>
<tr>
<td>-------</td>
<td>--------------</td>
</tr>
<tr>
<td>X-10</td>
<td>X-10</td>
</tr>
<tr>
<td>Issues</td>
<td>2-3, 4-24, 8-12, 13-9</td>
</tr>
<tr>
<td>ITE</td>
<td>13-6, 13-14, 13-20</td>
</tr>
<tr>
<td>ITLB</td>
<td>2-3, 2-6, 2-16, 9-6, 9-8</td>
</tr>
<tr>
<td>J</td>
<td>1-1, 1-2, 1-3, 2-16, 3-2, 3-4, 3-19, 6-1, A-82, A-83, A-91, A-141</td>
</tr>
<tr>
<td>K0</td>
<td>4-24, 4-29, 6-7, 6-12, 9-2, 9-5, 9-10, 9-11, C-28</td>
</tr>
<tr>
<td>Kernel</td>
<td>2-16, 2-19, 3-20, 3-26, 4-16, 4-17, 4-18, 4-29, 5-2, 5-22, 5-23, 6-1, 6-6, 6-7, 6-10, 6-11, 6-12, 6-13, 9-2, 13-5, 13-6, C-1, C-7, C-14, C-15</td>
</tr>
<tr>
<td>kseg0</td>
<td>4-24, 6-7, 6-12, 9-10, C-28</td>
</tr>
<tr>
<td>kseg1</td>
<td>6-7, 6-12</td>
</tr>
<tr>
<td>kseg3</td>
<td>2-16, 4-9, 6-1, 6-7, 6-12, 6-13</td>
</tr>
<tr>
<td>kseg3</td>
<td>6-7, 6-12</td>
</tr>
<tr>
<td>kseg3</td>
<td>4-16, 4-17, 4-18, 5-2, 6-6, 6-8, 6-9, 6-10, 6-11, 6-12, 6-13, C-14, C-15</td>
</tr>
<tr>
<td>kseg3</td>
<td>2-16, 6-1, 6-7, 6-12</td>
</tr>
<tr>
<td>kseg3</td>
<td>3-4, 13-8, A-56, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>kseg3</td>
<td>3-4, A-57, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>kseg3</td>
<td>3-4, 13-8, A-58, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>kseg3</td>
<td>3-5, 3-21, 3-26, 10-13, A-141, B-163, C-41, D-25, D-40</td>
</tr>
</tbody>
</table>
Index

LDR .......................................................... 3-4, 3-8, A-59, A-63, A-64, A-141, B-163, C-41, D-40
LHU .......................................................... 3-4, A-68, A-141, B-163, C-41, D-40
Link ....................................................... 2-11, 3-17, 3-18, 4-4
LL ............................................................... 1-2, 3-4, A-142, B-165, C-42, D-41
LLD ......................................................... 1-2, 3-4, A-142, B-165, C-42, D-41
LO ......................................................... 2-11, 2-14, 3-2, 3-23, 3-24, 3-26, 4-1, 4-2, 4-3, 4-4, A-38, A-39, A-40, A-81, A-85, A-86, A-87, B-2, B-5, B-11, B-13, B-23, B-25, B-66, B-67, B-68, B-70, B-84, B-85, B-86, B-87, B-91, B-92, B-93, B-95, B-102, B-106, B-111, B-113, B-116, B-117, B-118, B-120, B-122
LO0 ......................................................... 4-2, 4-3, 4-4, 6-16, B-2
LO1 ................................................. 2-11, 2-14, 4-2, 4-3, 4-4, 6-16, B-2, B-3, B-7, B-8, B-9, B-12, B-14, B-16, B-19, B-24, B-26
Lock ..................................................... 2-17, 4-32, 5-11, C-11, C-12, C-13
Locking .................................................. 2-17
logical pipe .............................................. 2-10, 2-12, 2-13
LQ .......................................................... 3-5, 3-25, 13-8, A-141, B-4, B-10, B-163, C-41, D-40
LRF ......................................................... 4-32, 5-11, C-9, C-10, C-11, C-12, C-13
LUI ........................................................ 3-14, 3-26, A-69, A-141, B-163, C-41, D-40
LWC1 ..................................................... 3-5, 3-21, 3-26, 10-13, A-141, B-163, C-41, D-26, D-40
LWC2 ..................................................... A-142, B-165, C-42, D-41
LZC ....................................................... 2-13, B-4, B-90

M
MAC ......................................................... 2-11, 3-16, 3-22
MAC0 ..................................................... 2-11, 3-12, 2-13
MAC1 ..................................................... 2-11, 2-12, 2-13
MADD .................................................. 3-23, 3-26, B-3, B-11, B-13, B-163
MADD1 ............................................... 2-14, 3-23, 3-26, 4-2, B-3, B-12, B-14, B-163
MADDU .................................................. 3-23, 3-26, B-3, B-13, B-163
MADDU1 .............................................. 2-14, 3-23, 3-26, 4-2, B-3, B-14, B-163
Mask ................................................. 2-15, 2-19, 3-20, 4-5, 4-10, 4-16, 4-17, 5-9, 5-24, 6-15, 13-3, 13-4, 13-7, 13-8, C-20, C-22, C-24, C-30, C-32, C-34, C-39, C-40
MASK .................................................. 4-10, 6-16
Maskable .............................................. 5-8, 5-12
MAX ...................................................... 2-18
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB</td>
<td>6-2, 6-5, 6-12, 6-13, A-52, A-53</td>
</tr>
<tr>
<td>MF0</td>
<td></td>
</tr>
<tr>
<td>MFBPC</td>
<td>3-20, 13-4, C-17, C-41</td>
</tr>
<tr>
<td>MFC0</td>
<td>3-20, 4-1, 9-3, 13-2, 13-4, C-18</td>
</tr>
<tr>
<td>MFC1</td>
<td>3-21, 10-13, D-27, D-40</td>
</tr>
<tr>
<td>MFDAB</td>
<td>3-20, 13-4, C-19, C-41</td>
</tr>
<tr>
<td>MFDABM</td>
<td>3-20, 13-4, C-20, C-41</td>
</tr>
<tr>
<td>MFDVBM</td>
<td>3-20, 13-4, C-22, C-41</td>
</tr>
<tr>
<td>MFHI</td>
<td>2-11, 3-16, A-80, A-81, A-141</td>
</tr>
<tr>
<td>MFH1</td>
<td>2-11, 2-14, 3-23, 4-2, B-3, B-15, B-163</td>
</tr>
<tr>
<td>MFIAB</td>
<td>3-20, 13-4, C-23, C-41</td>
</tr>
<tr>
<td>MFIABM</td>
<td>3-20, 13-4, C-24, C-41</td>
</tr>
<tr>
<td>MFLO</td>
<td>3-16, 3-23, A-81, A-141</td>
</tr>
<tr>
<td>MFLO1</td>
<td>2-14, 3-23, 4-2, B-3, B-16, B-163</td>
</tr>
<tr>
<td>MFPC</td>
<td>3-20, 9-2, 9-3, C-25, C-41</td>
</tr>
<tr>
<td>MFPS</td>
<td>3-20, 9-2, 9-3, C-26, C-41</td>
</tr>
<tr>
<td>MFSAD</td>
<td>3-25, A-141, B-5, B-17, B-20, B-21, B-22</td>
</tr>
<tr>
<td>MIN</td>
<td>2-18</td>
</tr>
<tr>
<td>Misaligned</td>
<td>3-8</td>
</tr>
<tr>
<td>misalignment</td>
<td></td>
</tr>
<tr>
<td>mispredicted</td>
<td>9-6, 9-7</td>
</tr>
<tr>
<td>Miss</td>
<td>2-17, 4-17, 6-4, 8-8, 9-7, 9-8, 12-6</td>
</tr>
<tr>
<td>misses</td>
<td>1-1, 6-17, 9-9</td>
</tr>
<tr>
<td>MMI</td>
<td>5-22, A-141, B-163, B-164, B-165, C-41, D-40</td>
</tr>
<tr>
<td>MMI0</td>
<td>B-163</td>
</tr>
<tr>
<td>MMI1</td>
<td>B-163, B-164</td>
</tr>
<tr>
<td>MMI2</td>
<td>B-163, B-165</td>
</tr>
<tr>
<td>MMI3</td>
<td>B-163, B-165</td>
</tr>
<tr>
<td>MMU</td>
<td>2-3, 2-15, 2-16, 4-5, 6-1, 6-14</td>
</tr>
<tr>
<td>mod</td>
<td>A-38, A-40, B-7, B-9, B-66, B-68, B-70</td>
</tr>
<tr>
<td>MOV</td>
<td>11-6, D-28</td>
</tr>
<tr>
<td>MOV, fmt</td>
<td>10-8</td>
</tr>
<tr>
<td>MOV.fmt</td>
<td>3-21, 10-14, D-41</td>
</tr>
<tr>
<td>Move1</td>
<td>2-11</td>
</tr>
<tr>
<td>MOVN</td>
<td>3-19, A-82, A-141</td>
</tr>
<tr>
<td>MOVZ</td>
<td>3-19, A-83, A-141</td>
</tr>
<tr>
<td>MT0</td>
<td>C-41</td>
</tr>
<tr>
<td>MTBPC</td>
<td>3-20, 13-4, 13-16, 13-19, C-27, C-41</td>
</tr>
<tr>
<td>MTC0</td>
<td>3-20, 4-1, 9-3, 13-2, 13-4, C-28</td>
</tr>
</tbody>
</table>

X-12
<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
<th>Index Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTC1</td>
<td>3-21, 3-26, 10-13, D-29, D-40</td>
<td>3-21</td>
</tr>
<tr>
<td>MTDAB</td>
<td>3-20, 13-4, C-29, C-41</td>
<td>10-13</td>
</tr>
<tr>
<td>MTDABM</td>
<td>3-20, 13-4, C-30, C-41</td>
<td>10-13</td>
</tr>
<tr>
<td>MTDVB</td>
<td>3-20, 13-4, C-31, C-41</td>
<td>10-13</td>
</tr>
<tr>
<td>MTDVBM</td>
<td>3-20, 13-4, C-32, C-41</td>
<td>10-13</td>
</tr>
<tr>
<td>MTHI1</td>
<td>2-11, 2-14, 3-23, 4-2, B-3, B-18, B-163</td>
<td>3-23, 4-2</td>
</tr>
<tr>
<td>MTIAB</td>
<td>3-20, 13-4, C-33, C-41</td>
<td>10-13</td>
</tr>
<tr>
<td>MTIABM</td>
<td>3-20, 13-4, C-34, C-41</td>
<td>10-13</td>
</tr>
<tr>
<td>MTLO</td>
<td>3-16, A-85, A-141</td>
<td>10-13</td>
</tr>
<tr>
<td>MTLO1</td>
<td>2-14, 3-23, 4-2, B-3, B-19, B-163</td>
<td>3-23, 4-2</td>
</tr>
<tr>
<td>MTPC</td>
<td>3-20, 9-2, 9-3, C-35, C-41</td>
<td>10-13</td>
</tr>
<tr>
<td>MTPS</td>
<td>3-20, 9-2, 9-3, C-36, C-41</td>
<td>10-13</td>
</tr>
<tr>
<td>MTSAB</td>
<td>2-13, 3-25, A-141, B-5, B-17, B-20</td>
<td>3-25, A-141</td>
</tr>
<tr>
<td>MTSABH</td>
<td>2-13, 3-25, A-141, A-142, B-5, B-20, B-21, B-22, B-161</td>
<td>3-25, A-141</td>
</tr>
<tr>
<td>MTSAX</td>
<td>2-13, 3-25, A-141, A-142, B-5, B-20, B-22, B-161</td>
<td>3-25, A-141</td>
</tr>
<tr>
<td>Multi</td>
<td>2-18, B-20</td>
<td>3-25, A-141</td>
</tr>
<tr>
<td>Multimedia</td>
<td>1-1, 1-2, 2-3, 2-6, 3-2, 3-4, 3-5, 3-23</td>
<td>1-1, 1-2, 2-3, 2-6, 3-2, 3-4, 3-5, 3-23</td>
</tr>
<tr>
<td>Multimaster</td>
<td>2-18, B-2</td>
<td>1-1, 1-2, 2-3, 2-6, 3-2, 3-4, 3-5, 3-23</td>
</tr>
<tr>
<td>multiply</td>
<td>2-14, 3-2, 3-4, 3-16, 3-22, 3-23, 4-1, 4-2, 4-4, A-8, A-86, A-87, A-125, B-11, B-12, B-13, B-14, B-23, B-24, B-25, B-26, B-84, B-85, B-86, B-87, B-91, B-92, B-93, B-95, B-111, B-113, B-118, B-120, B-122, C-16, D-30</td>
<td>1-1, 1-2, 2-3, 2-6, 3-2, 3-4, 3-16, 3-21, 3-22, 3-23, 3-24, 3-26, 4-1, B-1, B-3, B-5</td>
</tr>
<tr>
<td>Multiply</td>
<td>1-1, 1-2, 2-3, 2-6, 2-9, 2-11, 3-2, 3-14, 3-16, 3-21, 3-22, 3-23, 3-24, 3-26, 4-1, B-1, B-3, B-5</td>
<td>1-1, 1-2, 2-3, 2-6, 2-9, 2-11, 3-2, 3-14, 3-16, 3-21, 3-22, 3-23, 3-24, 3-26, 4-1, B-1, B-3, B-5</td>
</tr>
<tr>
<td>MULTU</td>
<td>3-16, 3-23, 3-26, A-87, A-141, B-3, B-25</td>
<td>3-16, 3-23, 3-26, A-87, A-141, B-3, B-25</td>
</tr>
<tr>
<td>MULTU1</td>
<td>2-14, 3-23, 3-26, 4-2, B-3, B-26, B-163</td>
<td>2-14, 3-23, 3-26, 4-2, B-3, B-26, B-163</td>
</tr>
<tr>
<td>N</td>
<td>10-11, 11-6, D-8, D-10, D-11, D-12, D-13</td>
<td>10-11, 11-6, D-8, D-10, D-11, D-12, D-13</td>
</tr>
<tr>
<td>NaNs</td>
<td>2-18</td>
<td>2-18</td>
</tr>
<tr>
<td>NBE</td>
<td>4-23, 5-11, C-28</td>
<td>4-23</td>
</tr>
<tr>
<td>NEG</td>
<td>2-18, 11-6, D-31</td>
<td>2-18</td>
</tr>
<tr>
<td>NEG.fmt</td>
<td>3-21, 10-14, D-41</td>
<td>3-21</td>
</tr>
<tr>
<td>Negate</td>
<td>3-21, 8-3, D-2, D-31, D-32, D-33</td>
<td>3-21, 8-3, D-2, D-31, D-32, D-33</td>
</tr>
<tr>
<td>NMI</td>
<td>4-17, 4-19, 4-33, 5-2, 5-5, 5-7, 5-8, 5-9, 5-10, 5-12, 8-10, 8-13, 9-11, 12-6, C-14</td>
<td>4-17, 4-19, 4-33, 5-2, 5-5, 5-7, 5-8, 5-9, 5-10, 5-12, 8-10, 8-13, 9-11, 12-6, C-14</td>
</tr>
</tbody>
</table>
### Index

- **nonmaskable** ........................................................................................................................................ 4-33
- **Normalization** ..................................................................................................................................... 2-9
- **NOT** .................................................................................................................................................... 6-2, 13-8, 13-20, A-3, A-88, B-124

### O
- **opCode** .......................................................................................................................................... 2-16, 3-9, 5-22, 6-1, A-2
- **OpCode** .............................................................................................................................................. 3-23, 3-24, 3-25, 6-20, 9-3, A-141, A-142, B-163, B-164, B-165, C-6, C-25, C-26, C-35, C-36, C-41, C-42, D-40, D-41
- **operand** ............................................................................................................................................... 1-2, 3-14, 3-22, 3-23, A-104, B-1, B-3, D-1, D-4, D-31, D-35
- **Operand** ............................................................................................................................................. 2-4, 3-14, 3-15, 3-23, B-3
- **ORI** .................................................................................................................................................... 3-14, A-90, A-141, B-163, C-41, D-40
- **Ov** ....................................................................................................................................................... 4-20, 5-8, 5-26
- **OVERFLOW** ..................................................................................................................................... 5-5
- **OVFL** .................................................................................................................................................. 4-28, 4-30, 9-2, 9-10, 9-11

### P
- **P0EXEA** .............................................................................................................................................. 12-3, 12-4
- **P0EXEB** .............................................................................................................................................. 12-3, 12-4
- **P1EXEA** .............................................................................................................................................. 12-3, 12-4
- **P1EXEB** .............................................................................................................................................. 12-3, 12-4
- **PA** ...................................................................................................................................................... C-6, C-7, C-9, C-10, C-11, C-12
- **PABSH** .............................................................................................................................................. 3-24, B-4, B-27, B-164
- **PABSW** .............................................................................................................................................. 3-24, B-4, B-28, B-164
- **PADDB** .............................................................................................................................................. 3-24, B-3, B-29, B-164
- **PADDH** .............................................................................................................................................. 3-24, B-3, B-30, B-164
- **PADDSB** ............................................................................................................................................ 3-24, B-3, B-31, B-164
- **PADDSH** ............................................................................................................................................ 3-24, B-3, B-35, B-164
- **PADDSW** ............................................................................................................................................ 3-24, B-3, B-37, B-164
- **PADDUB** ............................................................................................................................................ 3-24, B-3, B-39, B-164
- **PADDUH** ............................................................................................................................................ 3-24, B-3, B-42, B-164
- **PADDUW** ............................................................................................................................................ 3-24, B-3, B-44, B-164
- **PADDW** .............................................................................................................................................. 3-24, B-3, B-46, B-164
- **PADSBH** ............................................................................................................................................ 3-24, B-3, B-47, B-164

---

**X-14**
PINTEH .......................................................... 3-25, B-5, B-88, B-165
PINTH .......................................................... 3-25, B-5, B-89, B-165
PLZCW .......................................................... 3-25, B-4, B-90, B-165
PMADDH ...................................................... 3-24, B-5, B-91, B-94, B-96, B-112, B-114, B-119, B-121, B-123, B-165
PMADDUW .................................................. 3-24, B-5, B-93, B-165
PMADDW ...................................................... 3-24, B-5, B-95, B-165
PMAXH .......................................................... 3-24, B-4, B-97, B-164
PMAXW .......................................................... 3-24, B-4, B-99, B-164
PMFHI .......................................................... 3-24, B-5, B-101, B-165
PMFHL .......................................................... 3-24, B-5, B-102, B-163
PMFLO .......................................................... 3-24, B-5, B-106, B-165
PMINH .......................................................... 3-24, B-4, B-107, B-164
PMINW .......................................................... 3-24, B-4, B-109, B-164
PMSUBH ...................................................... 3-24, B-5, B-111, B-165
PMSUBW ...................................................... 3-24, B-5, B-113, B-165
PMTHI .......................................................... 3-24, B-5, B-115, B-165
PMTHL .......................................................... 3-24, B-5, B-116, B-163
PMTLO .......................................................... 3-24, B-5, B-117, B-165
PMULTH ...................................................... 3-24, B-5, B-118, B-165
PMULTUW .................................................. 3-24, B-5, B-120, B-165
PMULTW ...................................................... 3-24, B-5, B-122, B-165
PNOR ........................................................... 3-25, B-4, B-124, B-165
pointer ........................................................... 4-9, A-92
POR ............................................................ 3-25, B-4, B-125, B-165
PPAC5 .......................................................... 3-25, B-5, B-126, B-164
PPACB .......................................................... 3-25, B-5, B-128, B-164
PPACH .......................................................... 3-25, B-5, B-129, B-164
PPACW .......................................................... 3-25, B-5, B-130, B-164
precise ...................................................................... 9-4
prediction ................................................................ 1-2, 2-3, 4-23, 9-7
Prediction .................................................................. 4-23
PREF ............................................................. 3-19, 4-23, A-2, A-91, A-141, B-163, C-41, D-40
prefetch .................................................................... 5-19, A-91, A-92
Prefetch ............................................................... 1-1, 1-2, 2-11, 2-17, 3-19, 8-8, 9-7, A-7, A-92
Prefix ........................................................................ 8-3
PREVH .......................................................... 3-25, B-5, B-131, B-165
PRId ............................................................... 2-15, 4-5, 4-22
priorities .................................................................. 12-7
privilege ..................................................................... 9-5, 9-11, C-8
privilege mode ......................................................... 9-5, 9-11
<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probe</td>
<td>3-20, 4-6, 4-14, 5-17, 6-20</td>
</tr>
<tr>
<td>PROT3W</td>
<td>3-25, B-5, B-132, B-165</td>
</tr>
<tr>
<td>Pseudo</td>
<td>2-15, 4-5</td>
</tr>
<tr>
<td>PSLH</td>
<td>A-3, A-4, A-6, B-2, D-2</td>
</tr>
<tr>
<td>PSLLVW</td>
<td>3-25, B-4, B-134, B-165</td>
</tr>
<tr>
<td>PSLLW</td>
<td>3-25, B-4, B-135, B-163</td>
</tr>
<tr>
<td>PSRAH</td>
<td>3-25, B-4, B-136, B-163</td>
</tr>
<tr>
<td>PSRAVW</td>
<td>3-25, B-4, B-137, B-165</td>
</tr>
<tr>
<td>PSRAW</td>
<td>3-25, B-4, B-138, B-163</td>
</tr>
<tr>
<td>PSRLH</td>
<td>3-25, B-4, B-139, B-163</td>
</tr>
<tr>
<td>PSRLVW</td>
<td>3-25, B-4, B-140, B-165</td>
</tr>
<tr>
<td>PSRLW</td>
<td>3-25, B-4, B-141, B-163</td>
</tr>
<tr>
<td>PSUBB</td>
<td>3-24, B-3, B-142, B-164</td>
</tr>
<tr>
<td>PSUBH</td>
<td>3-24, B-3, B-143, B-164</td>
</tr>
<tr>
<td>PSUBSB</td>
<td>3-24, B-3, B-144, B-164</td>
</tr>
<tr>
<td>PSUBSH</td>
<td>3-24, B-3, B-148, B-164</td>
</tr>
<tr>
<td>PSUBSW</td>
<td>3-24, B-3, B-150, B-164</td>
</tr>
<tr>
<td>PSUBUB</td>
<td>3-24, B-3, B-152, B-164</td>
</tr>
<tr>
<td>PSUBUH</td>
<td>3-24, B-3, B-155, B-164</td>
</tr>
<tr>
<td>PSUBUW</td>
<td>3-24, B-3, B-157, B-164</td>
</tr>
<tr>
<td>PSUBW</td>
<td>3-24, B-3, B-159, B-164</td>
</tr>
<tr>
<td>PTagLo</td>
<td>4-31, 4-32</td>
</tr>
<tr>
<td>PTE</td>
<td>2-15, 4-5, 4-9</td>
</tr>
<tr>
<td>PTEBase</td>
<td>4-9</td>
</tr>
<tr>
<td>PTEs</td>
<td>4-9</td>
</tr>
<tr>
<td>PXOR</td>
<td>3-25, B-4, B-160, B-165</td>
</tr>
<tr>
<td>QFSRV</td>
<td>3-25, B-5, B-20, B-21, B-22, B-161, B-164</td>
</tr>
<tr>
<td>qNaN</td>
<td>11-6</td>
</tr>
<tr>
<td>Quadword</td>
<td>1-2, 3-5, 3-8, 3-10, 3-12, 3-25, 8-9, B-4, B-5</td>
</tr>
<tr>
<td>QUADWORD</td>
<td>A-7, B-10, B-162</td>
</tr>
<tr>
<td>Quintibyte</td>
<td>3-10, 3-12</td>
</tr>
<tr>
<td>quotient</td>
<td>4-4, A-38, A-40, B-7, B-9</td>
</tr>
<tr>
<td>R10000</td>
<td>1-3</td>
</tr>
<tr>
<td>R4000</td>
<td>1-3, 6-2</td>
</tr>
<tr>
<td>random</td>
<td>2-15, 4-5, 4-11, 6-2</td>
</tr>
<tr>
<td>Random</td>
<td>2-15, 3-20, 4-5, 4-7, 4-11, 4-14, 5-11, 5-16, 5-17, 6-20, C-40</td>
</tr>
</tbody>
</table>

**Q**

<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>QFSRV</td>
<td>3-25, B-5, B-20, B-21, B-22, B-161, B-164</td>
</tr>
</tbody>
</table>

**R**

<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>R10000</td>
<td>1-3</td>
</tr>
<tr>
<td>R4000</td>
<td>1-3, 6-2</td>
</tr>
<tr>
<td>random</td>
<td>2-15, 4-5, 4-11, 6-2</td>
</tr>
<tr>
<td>Random</td>
<td>2-15, 3-20, 4-5, 4-7, 4-11, 4-14, 5-11, 5-16, 5-17, 6-20, C-40</td>
</tr>
<tr>
<td>Term</td>
<td>Pages</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>Random5</td>
<td>C-40</td>
</tr>
<tr>
<td>REGIMM</td>
<td>5-22, A-141, A-142, B-163, C-41, D-40</td>
</tr>
<tr>
<td>register</td>
<td>10-2, 10-6, 11-2, 11-3, 11-8, 11-9</td>
</tr>
<tr>
<td>Register</td>
<td>2-5, 2-6, 2-8, 2-15, 3-14, 3-15, 3-17, 3-20, 3-25, 4-3, 4-4, 4-5, 4-6, 4-7, 4-8, 4-9, 4-10, 4-11, 4-12, 4-13, 4-14, 4-15, 4-16, 4-17, 4-18, 4-19, 4-21, 4-22, 4-23, 4-25, 4-26, 4-27, 4-28, 4-29, 4-30, 4-32, 4-33, 5-8, 6-9, 6-10, 6-12, 6-16, 8-25, 9-2, 9-3, 9-4, 9-10, 10-7, 10-8, 10-9, 13-2, 13-3, 13-4, 13-5, 13-7, 13-8, 13-9, A-3, A-4, A-5, A-9, A-54, B-3, B-5, B-161</td>
</tr>
<tr>
<td>registers</td>
<td>10-4</td>
</tr>
<tr>
<td>Registers</td>
<td>2-1, 2-3, 2-14, 2-15, 3-17, 4-1, 4-2, 4-3, 4-4, 4-5, 4-8, 4-26, 4-28, 4-31, 6-14, 9-2, 9-3, 9-4, 13-3</td>
</tr>
<tr>
<td>REL</td>
<td>8-11, 8-14, 8-15</td>
</tr>
<tr>
<td>Request</td>
<td>9-9</td>
</tr>
<tr>
<td>Res</td>
<td>4-19, 5-8</td>
</tr>
<tr>
<td>Reset</td>
<td>4-18, 4-19, 5-1, 5-2, 5-7, 5-8, 5-9, 5-10, 5-11, 8-11, 9-4, 12-6, 13-14</td>
</tr>
<tr>
<td>RESET</td>
<td>5-11, 5-12, 8-11, 8-14</td>
</tr>
<tr>
<td>RI</td>
<td>2-16, 4-20, 5-8, 5-22, 6-1</td>
</tr>
<tr>
<td>Root</td>
<td>3-21</td>
</tr>
<tr>
<td>Rotate</td>
<td>3-25, B-5</td>
</tr>
<tr>
<td>ROUND.L</td>
<td>D-32</td>
</tr>
<tr>
<td>ROUND.L.fmt</td>
<td>3-21, 10-14, D-41</td>
</tr>
<tr>
<td>ROUND.W</td>
<td>D-33</td>
</tr>
<tr>
<td>ROUND.W.fmt</td>
<td>3-21, 10-14, D-41</td>
</tr>
<tr>
<td>R SQRT</td>
<td>2-18, 3-26</td>
</tr>
</tbody>
</table>

**S**

<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>4-29, 9-2, 9-5, 9-11</td>
</tr>
<tr>
<td>S1</td>
<td>4-29, 9-5, 9-11</td>
</tr>
<tr>
<td>SA</td>
<td>2-3, 2-11, 2-12, 2-13, 2-14, 3-25, 4-1, 4-2, 4-3, 4-4, B-17, B-20, B-21, B-22, B-161</td>
</tr>
<tr>
<td>Saturate</td>
<td>B-34, B-36, B-38, B-41, B-43, B-45, B-147, B-149, B-151, B-154, B-156, B-158</td>
</tr>
<tr>
<td>saturation</td>
<td>B-3, B-31, B-35, B-37, B-39, B-42, B-44, B-144, B-148, B-150, B-152, B-155, B-157</td>
</tr>
<tr>
<td>Saturation</td>
<td>3-24, B-3</td>
</tr>
<tr>
<td>SB</td>
<td>3-4, A-93, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>SC</td>
<td>1-2, 3-4, A-142, B-165, C-42, D-41</td>
</tr>
<tr>
<td>SCD</td>
<td>1-2, 3-4, A-142, B-165, C-42, D-41</td>
</tr>
<tr>
<td>SDC1</td>
<td>3-5, 3-21, 10-13, A-141, B-163, C-41, D-34, D-40</td>
</tr>
<tr>
<td>Term</td>
<td>References</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>SDR</td>
<td>3-4, 3-8, A-95, A-99, A-100, A-141, B-163, C-41, D-40</td>
</tr>
<tr>
<td>segment</td>
<td>2-16, 4-9, 6-1, 6-8, 6-9, 13-9</td>
</tr>
<tr>
<td>Segment</td>
<td>6-9, 6-10, 6-12</td>
</tr>
<tr>
<td>Semaphore</td>
<td>3-4</td>
</tr>
<tr>
<td>Septibyte</td>
<td>3-10, 3-12</td>
</tr>
<tr>
<td>Serialization</td>
<td>3-19</td>
</tr>
<tr>
<td>Sextibyte</td>
<td>3-10, 3-12</td>
</tr>
<tr>
<td>SH</td>
<td>3-4, A-103, A-141, B-102, B-163, C-41, D-40</td>
</tr>
<tr>
<td>Shift</td>
<td>2-3, 2-11, 3-14, 3-15, 3-25, 3-26, 4-2, 4-4, B-4, B-5</td>
</tr>
<tr>
<td>Shifter</td>
<td>2-3</td>
</tr>
<tr>
<td>shutdown</td>
<td>6-2</td>
</tr>
<tr>
<td>Signal</td>
<td>8-3, 8-7, A-8</td>
</tr>
<tr>
<td>SIO</td>
<td>4-17, 4-18, 4-19, 4-33, 5-2, 5-5, 5-7, 5-8, 5-9, 5-10, 5-25, 8-10, 12-6, 13-8, C-14</td>
</tr>
<tr>
<td>SIOMP</td>
<td>8-10</td>
</tr>
<tr>
<td>SIOP</td>
<td>4-19, 5-25</td>
</tr>
<tr>
<td>sll</td>
<td>12-10, 12-11, 12-12, 12-13, 12-14, 12-15, 12-16, 12-17, 12-18, 12-19, 12-20</td>
</tr>
</tbody>
</table>
SLW .......................................................................................................................... B-102
Snooping ................................................................................................................... 2-17
SPECIAL ................................................................................................................ 5-22, A-9, A-141, B-163, C-41, D-40
SQ .......................................................................................................................... 3-5, 3-25, 13-8, A-141, B-4, B-162, B-163, C-41, D-40
SQRT ................................................................................................................... 2-18, 3-26, D-35
SORT fmt .................................................................................................................. 3-21, 10-14, D-41
Square .................................................................................................................... 3-21
Square Root ............................................................................................................ D-35
SR .......................................................................................................................... 1-5, 4-16
SRA ....................................................................................................................... 3-15, A-110, A-141
SRAV .................................................................................................................. 3-15, A-111, A-141
SRL ....................................................................................................................... 3-15, A-112, A-141
SRLV ................................................................................................................... 3-15, A-113, A-141
sseg ........................................................................................................................ 6-7, 6-10
State ....................................................................................................................... 6-6, 9-4
Status ................................................................................................................... 1-5, 2-15, 3-5, 3-20, 3-21, 4-5, 4-16, 4-17, 4-18, 4-21, 4-25, 4-29, 5-2, 5-5, 5-7, 5-9, 5-11, 5-12, 5-13, 5-14, 5-16, 5-19, 5-23, 5-24, 5-25, 6-2, 6-6, 6-8, 6-9, 6-10, 6-11, 6-12, 6-13, 8-25, 10-2, 10-4, 10-7, 10-8, 10-9, 11-2, 11-8, 11-9, 12-3, 12-4, 13-4, C-1, C-7, C-9, C-13, C-14, C-15, C-16
STATUS .................................................................................................................. 9-2, 9-10, 9-11, 12-6, 13-5, 13-6
steering ................................................................................................................. 2-6, 4-31
Steering Bits ........................................................................................................... C-10
stepping .................................................................................................................. 1-2, 9-8, 9-10, B-20, B-21, B-22
SUB ....................................................................................................................... 2-18, 3-15, 5-26, A-114, A-141, D-36
SUB fmt ............................................................................................................... 3-21, 10-14, D-41
Subroutine .......................................................................................................... 3-17
Subsequent .......................................................................................................... 2-4, 6-17
Subtract ............................................................................................................... 3-15, 3-21, 3-24, B-3, B-5
supervisor ........................................................................................................... 4-18, 5-15, 6-10, 6-12, 9-11, 13-5, 13-14
Supervisor .......................................................................................................... 2-16, 2-19, 4-17, 4-18, 4-29, 5-2, 5-15, 5-22, 5-23, 6-6, 6-7, 6-10, 6-12, 9-2, 13-5, 13-6, C-1, C-14, C-15
SUPERVISOR ................................................................................................... 9-5
suseg ..................................................................................................................... 6-7, 6-10
SWC1 .................................................................................................................. 3-5, 3-21, 10-13, 13-2, A-141, B-163, C-41, D-37, D-40
SWC2 .................................................................................................................. A-142, B-165, C-42, D-41
<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNC</td>
<td>2-11, 2-12, 2-13, 3-19, 5-24, 6-17, 13-9, 13-16, 13-18, 13-20, A-125, A-141, C-13, C-27, C-28, C-29, C-30, C-31, C-32, C-33, C-34, C-35, C-36, C-38, C-39, C-40</td>
</tr>
<tr>
<td>Synchronization</td>
<td>2-11, 3-19</td>
</tr>
<tr>
<td>Sys</td>
<td>4-20, 5-8, 5-20</td>
</tr>
<tr>
<td>SYSAACK</td>
<td>8-3, 8-9, 8-12, 8-13, 8-14, 8-16, 8-19, 8-22, 8-25, 8-26, 8-27, 8-28, 8-29</td>
</tr>
<tr>
<td>SYSADDR</td>
<td>8-3, 8-7</td>
</tr>
<tr>
<td>SYSASTART</td>
<td>8-3, 8-7, 8-9, 8-12, 8-13, 8-16, 8-19</td>
</tr>
<tr>
<td>SYSBE</td>
<td>8-3, 8-7</td>
</tr>
<tr>
<td>Syscall</td>
<td>4-20, 5-2, 5-8, 5-9</td>
</tr>
<tr>
<td>SYSCALL</td>
<td>2-11, 3-18, 4-4, 5-10, 5-20, 9-7, 9-8, A-126, A-141</td>
</tr>
<tr>
<td>SYSACK</td>
<td>8-3, 8-10, 8-12, 8-13, 8-16, 8-17, 8-19, 8-20, 8-22, 8-25, 8-26, 8-27, 8-28, A-125</td>
</tr>
<tr>
<td>SYSDATA</td>
<td>8-3, 8-6, 8-7, 8-9, 8-16, 8-17</td>
</tr>
<tr>
<td>SYSDSTART</td>
<td>8-3, 8-10, 8-12, 8-13, 8-16, 8-17, 8-19, 8-20, 8-25</td>
</tr>
<tr>
<td>SYSRD</td>
<td>8-3</td>
</tr>
<tr>
<td>SYSTSIZE</td>
<td>8-3, 8-9, 8-12, 8-13, 8-16, 8-19</td>
</tr>
<tr>
<td>SYSSIZE</td>
<td>8-3</td>
</tr>
<tr>
<td>SYSSWR</td>
<td>8-3</td>
</tr>
<tr>
<td>T</td>
<td>2-6, 2-7, 2-15, 4-5, C-9, C-11, C-12, C-13</td>
</tr>
<tr>
<td>Tag</td>
<td>2-15, 4-5, 4-31, 4-32</td>
</tr>
<tr>
<td>TagHi</td>
<td>C-6</td>
</tr>
<tr>
<td>TagHi</td>
<td>C-10, C-11</td>
</tr>
<tr>
<td>TagLo</td>
<td>C-10, C-11</td>
</tr>
<tr>
<td>TagLo</td>
<td>C-9, C-10, C-11, C-12</td>
</tr>
<tr>
<td>tags</td>
<td>C-9, C-10, C-11, C-12</td>
</tr>
<tr>
<td>TargetAddress</td>
<td>C-10, C-11</td>
</tr>
<tr>
<td>TEQ</td>
<td>3-18, 5-27, 9-8, A-127, A-141</td>
</tr>
<tr>
<td>TEQI</td>
<td>3-18, 5-27, 9-8, A-128, A-142</td>
</tr>
<tr>
<td>TGE</td>
<td>3-18, 5-27, 9-8, A-129, A-141</td>
</tr>
<tr>
<td>TGEI</td>
<td>3-18, 5-27, A-130, A-142</td>
</tr>
<tr>
<td>TGEIU</td>
<td>3-18, 5-27, A-131, A-142</td>
</tr>
<tr>
<td>TGEU</td>
<td>3-18, 5-27, A-132, A-141</td>
</tr>
<tr>
<td>timer</td>
<td>4-13, 4-15, 4-16</td>
</tr>
<tr>
<td>TLB</td>
<td>1-2, 2-3, 2-6, 2-7, 2-15, 2-16, 3-20, 4-5, 4-6, 4-7, 4-8, 4-9, 4-10, 4-11, 4-12, 4-14, 4-17, 4-20, 4-29, 5-2, 5-7, 5-8, 5-9, 5-10, 5-11, 5-12, 5-16, 5-17, 5-18, 6-1, 6-2, 6-3, 6-4, 6-6, 6-7, 6-8, 6-9, 6-12, 6-14, 6-15, 6-16, 6-17, 6-18, 6-19, 6-20, 12-6, A-6, A-56, A-57, A-58, A-62, A-66, A-67, A-68, A-70, A-74, A-78, A-79, A-92, A-93, A-94, A-98, A-102, A-103, A-116, A-120, A-124, B-10, B-162, C-6, C-7, C-8, C-28, C-37, C-38, C-39, C-40, D-26, D-37</td>
</tr>
<tr>
<td>TLBEntries</td>
<td>C-37</td>
</tr>
<tr>
<td>------------</td>
<td>-----</td>
</tr>
<tr>
<td>TLBL</td>
<td>4-8, 4-20, 5-8, 5-16, 5-17</td>
</tr>
<tr>
<td>TLBP</td>
<td>3-20, 4-6, 5-17, 5-18, 6-2, 6-20, C-37, C-42</td>
</tr>
<tr>
<td>TLBR</td>
<td>2-13, 3-20, 4-6, 6-20, C-38, C-42</td>
</tr>
<tr>
<td>TLBS</td>
<td>4-8, 4-20, 5-8, 5-16, 5-17</td>
</tr>
<tr>
<td>TLBWRI</td>
<td>2-13, 3-20, 4-6, 4-8, 6-20, C-28, C-38, C-39, C-42</td>
</tr>
<tr>
<td>TLBWR</td>
<td>2-13, 3-20, 4-7, 4-8, 6-20, C-28, C-38, C-40, C-42</td>
</tr>
<tr>
<td>TLT</td>
<td>3-18, 5-27, A-133, A-141</td>
</tr>
<tr>
<td>TLTI</td>
<td>3-18, 5-27, A-134, A-142</td>
</tr>
<tr>
<td>TLTIU</td>
<td>3-18, 5-27, A-135, A-142</td>
</tr>
<tr>
<td>TLTU</td>
<td>3-18, 5-27, A-136, A-141</td>
</tr>
<tr>
<td>TNE</td>
<td>3-18, 5-27, A-137, A-141</td>
</tr>
<tr>
<td>TNEI</td>
<td>3-18, 5-27, A-138, A-142</td>
</tr>
<tr>
<td>TPC</td>
<td>12-3, 12-5, 12-6, 12-7</td>
</tr>
<tr>
<td>TPCE</td>
<td>12-3, 12-5, 12-6</td>
</tr>
<tr>
<td>Trace</td>
<td>12-1, 12-2, 12-3</td>
</tr>
<tr>
<td>transaction</td>
<td>8-8, 8-10, 8-12, 8-14, 8-22</td>
</tr>
<tr>
<td>Translation</td>
<td>2-3, 6-2, 6-3, 6-4, 6-5, 6-18, 6-19, 6-20</td>
</tr>
<tr>
<td>translations</td>
<td>4-9, 6-1, A-92</td>
</tr>
<tr>
<td>TRAP</td>
<td>4-4, 5-27, 9-7</td>
</tr>
<tr>
<td>TRIG</td>
<td>13-9, 13-20</td>
</tr>
<tr>
<td>Trigger</td>
<td>2-19, 13-6</td>
</tr>
<tr>
<td>Triplebyte</td>
<td>3-10, 3-12</td>
</tr>
<tr>
<td>TRUNC.L</td>
<td>D-38</td>
</tr>
<tr>
<td>TRUNC.L.fmt</td>
<td>3-21, 10-14, D-41</td>
</tr>
<tr>
<td>TRUNC.W</td>
<td>D-39</td>
</tr>
<tr>
<td>TRUNC.W.fmt</td>
<td>3-21, 10-14, D-41</td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>U0</td>
<td>4-29, 9-2, 9-5, 9-11</td>
</tr>
<tr>
<td>U1</td>
<td>4-29, 9-5, 9-11</td>
</tr>
<tr>
<td>UCA</td>
<td>9-7</td>
</tr>
<tr>
<td>UCAB</td>
<td>2-4, 2-6, 2-7, 6-17, 9-9</td>
</tr>
<tr>
<td>Uncached</td>
<td>2-4, 4-8, 4-24, 6-7, 6-17, 6-20, 8-8, 8-12, 9-7, 9-10</td>
</tr>
</tbody>
</table>
underflow ........................................ 2-9, B-29, B-30, B-31, B-35, B-37, B-46, B-47, B-142, B-143, B-144, B-148, B-150, B-152, B-155, B-157, B-159

Underflow ............................................................ B-31, B-35, B-37, B-144, B-148, B-150, B-152, B-155, B-157
UNIX .......................................................................................... A-39, B-8, B-67
unmapped ............................................................. 5-11, 5-12, 6-7, 6-12, 9-8, 9-10, 13-9, A-6, C-28, C-38, C-39, C-40
Unmapped .............................................................. 6-7
Unsigned .......................................................... 3-4, 3-14, 3-15, 3-16, 3-18, 3-23, 3-24, B-3, B-5, B-158
useg .................................................................................. 6-7, 6-8, 6-9
UW ........................................................................................................... B-102

V
VA .................................................................................. C-6, C-7, C-8, C-9, C-10, C-11, C-12
VALID .......................................................................................... C-9
VALUE .................................................................................. 4-28, 4-30, 9-2
Value FPR .......................................................................................... D-10
ValueFPR .......................................................................................... D-4, D-12, D-13, D-16
VAX .......................................................................................... 3-6
VPN ................................................................................... 4-9, 5-15, 6-4, 6-5
VPN2 ................................................................................... 4-14, 6-16, C-39, C-40

W
WBB ................................................................. 2-4, 4-29, 8-15, 9-6, 9-9
Wide ................................................................................ 2-10, 2-11, 2-12, 2-13
wired .................................................................................. 2-15, 4-5, 4-11
Wired .................................................................................. 2-15, 4-5, 4-7, 4-11, 5-11
writeback .................................................................................. A-91
Writeback .................................................................................. 2-4, C-7, C-8, C-11, C-12, C-13
WRITEBACK .................................................................................. C-6, C-13

X
XORI ................................................................................ 3-14, A-140, A-141, B-163, C-41, D-40
A. CPU Instruction Set Details

This appendix provides a detailed description of the operation of each instruction. The instructions are listed in alphabetical order.

Exceptions that may occur due to the execution of each instruction are listed after the description of each instruction. Descriptions of the immediate cause and manner of handling exceptions are omitted from the instruction descriptions in this appendix.

Descriptions use a pseudocode notation explained in Section A.2.

For an overview of the instruction set, refer to Chapter 3 of the User's Manual.
A.1 Description of an Instruction

Each instruction description contains several sections that contain specific information about the instruction. The following sections describe the contents of each section in detail.

A.1.1 Instruction Mnemonic and Name

The instruction mnemonic and name are printed as page headings for each page in the instruction description.

A.1.2 Instruction Encoding Picture

The instruction word encoding is shown in pictorial form at the top of the instruction description. The picture shows the values of all constant fields and the opcode names for opcode fields in upper-case. It labels all variable fields with lower-case names that are used in the instruction description. Fields that contain zeroes but are not named are unused fields that are required to be zero.

A.1.3 Format

The assembler formats for the instruction and the architecture level at which the instruction was originally defined are shown.

A.1.4 Purpose

This is a very short statement of the purpose of the instruction.

A.1.5 Description

If a one-line symbolic description of the instruction is feasible, it will appear immediately to the right of the Description heading. The body of the section is a description of the operation of the instruction in text, tables, and figures. This description complements the high-level language description in the Operation section.

A.1.6 Restrictions

This section documents the restrictions on the instructions. Most restrictions fall in the category of alignment requirements for memory addresses, valid values of operands, and order of instructions necessary to guarantee correct execution.

A.1.7 Operation

This section describes the operation as pseudocode in a high-level language notation resembling Pascal. The purpose of this section is to describe the operation of the instruction clearly in a form with less ambiguity than prose.

A.1.8 Exceptions

This section lists the exceptions that can be caused by the operation of the instruction. It omits exceptions that can be caused by instruction fetch, performance counters, and breakpoints. It also omits exceptions that can be caused by asynchronous external events, e.g. interrupts. Although the Bus Error exception may be caused by the operation of a load, store or PREF instruction this section does not list Bus Error for load, store or PREF instructions because the relationship between these instructions and external error conditions, like Bus Error is asynchronous and implementation specific.
### A.1.9 Programming Notes, Implementation Notes

These sections contain material that is useful for programmers and implementors respectively but is not necessary to describe the instruction and does not belong in the description sections.

### A.2 Instruction Description Notation and Functions

The *Operation* sections of the instruction descriptions describe the operation performed by each instruction using a high-level language notation, or pseudocode. Symbols, functions, and structures used in the *Operation* sections are described here.

#### A.2.1.1 Pseudocode Language Statement Execution

Each of the high-level language statements in an operation description is executed in sequential order (as modified by conditional and loop constructs).

#### A.2.1.2 Pseudocode Symbols

Special symbols used in the notation are described in Table A-1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>←</td>
<td>Assignment.</td>
</tr>
<tr>
<td>=, ≠</td>
<td>Tests for equality and inequality.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>X^y</td>
<td>A y-bit string formed by y copies of the single-bit value x.</td>
</tr>
<tr>
<td>Xy..z</td>
<td>Selection of bits y through z of bit string x.</td>
</tr>
<tr>
<td>+, −</td>
<td>Two's complement or floating point arithmetic: addition, subtraction.</td>
</tr>
<tr>
<td>*, ×</td>
<td>Two’s complement or floating point multiplication (both used for either).</td>
</tr>
<tr>
<td>div</td>
<td>Two’s complement integer division.</td>
</tr>
<tr>
<td>Mod</td>
<td>Two’s complement modulo.</td>
</tr>
<tr>
<td>/</td>
<td>Floating point division.</td>
</tr>
<tr>
<td>&lt;</td>
<td>Two’s complement less than comparison.</td>
</tr>
<tr>
<td>Not</td>
<td>Bit-wise logical NOT.</td>
</tr>
<tr>
<td>Nor</td>
<td>Bit-wise logical NOR.</td>
</tr>
<tr>
<td>Xor</td>
<td>Bit-wise logical XOR.</td>
</tr>
<tr>
<td>And</td>
<td>Bit-wise logical AND.</td>
</tr>
<tr>
<td>or</td>
<td>Bit-wise logical OR.</td>
</tr>
<tr>
<td>GPRLEN</td>
<td>The length in bits (64 in the C790), of the CPU General Purpose Registers.</td>
</tr>
<tr>
<td>GPR[x]</td>
<td>CPU General Purpose Register x. The content of GPR[0] is always zero.</td>
</tr>
<tr>
<td>CPR[z, x]</td>
<td>Coprocessor unit z, general register x.</td>
</tr>
<tr>
<td>CCR[z, x]</td>
<td>Coprocessor unit z, control register x.</td>
</tr>
<tr>
<td>CPCOND[z]</td>
<td>Coprocessor unit z condition signal.</td>
</tr>
<tr>
<td>BigEndian</td>
<td>Big-endian made as configured at reset (0→Little, 1→Big) from core boundary signal.</td>
</tr>
</tbody>
</table>
### Symbol | Meaning
--- | ---
$I_i$ | This occurs as a prefix to operation description lines and functions as a label. It indicates the instruction time during which the effects of the pseudocode lines appears to occur (i.e., when the pseudocode is "executed"). Unless otherwise indicated, all effects of the current instruction appear to occur during the instruction time of the current instruction.
No label is equivalent to a time label of "$I_i$"
Sometimes effects of an instruction appear to occur either earlier or later during the instruction time of another instruction. When that happens, the instruction operation is written in sections labeled with the instruction time, relative to the current instruction $I_i$, in which the effect of that pseudocode appears to occur. For example, an instruction may have a result that is not available until after the next instruction. Such an instruction will have the portion of the instruction operation description that writes the result register in a section labeled "$I_i+1$".
The effect of pseudocode statements for the current instruction labeled "$I_i+1$" appears to occur "at the same time" as the effect of pseudocode statements labeled "$I_i$" for the following instruction. Within one pseudocode sequence the effects of the statements takes place in order. However, between sequences of statements for different instructions that occur "at the same time", there is no order defined. Programs must not depend on a particular order of evaluation between such sections.

$PC$ | The Program Counter value. During the instruction time of an instruction this is the address of the instruction word. The address of the instruction that occurs during the next instruction time is determined by assigning a value to PC during an instruction time. If no value is assigned to PC during instruction time by any pseudocode statement, it is automatically incremented by 4 before the next instruction time. A taken branch assigns the target address to PC during the instruction time of the instruction in the branch delay slot.

$PSIZE$ | The SIZE, number of bits, of Physical address in an implementation.

### A.2.2 Definitions of Pseudocode Functions Used in Instruction Descriptions

A variety of functions are used in the pseudocode employed in the instruction descriptions. These functions are used to make the pseudocode more readable and also to abstract implementation-specific behavior. These functions are defined in this section. Certain additional functions specific to a particular coprocessor are described at the beginning of the appendix for that coprocessor.

### A.2.2.1 Coprocessor General Register Access Pseudocode Functions

Defined coprocessors, except for COP0, have instructions to exchange words and doublewords and quadwords between coprocessor general registers and the rest of the system. What a coprocessor does with a word or doubleword supplied to it, and how a coprocessor supplies a word or doubleword, is defined by the coprocessor itself. The functions are listed in Table A-2.
Table A-2. Coprocessor General Register Access Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>COP_LW(z, rt, memword)</strong></td>
<td>( z ): The coprocessor unit number. ( rt ): Coprocessor general register specifier. Memword: A 32-bit word value supplied to the coprocessor. This is the action taken by coprocessor ( z ) when supplied with a word from memory during a load word operation. The action is coprocessor-specific. The typical action would be to store the contents of memword in coprocessor general register ( rt ).</td>
</tr>
<tr>
<td><strong>COP_LD(z, rt, memdouble)</strong></td>
<td>( z ): The coprocessor unit number. ( rt ): Coprocessor general register specifier. Memdouble: 64-bit doubleword value supplied to the coprocessor. This is the action taken by coprocessor ( z ) when supplied with a doubleword from memory during a load doubleword operation. The action is coprocessor-specific. The typical action would be to store the contents of memdouble in coprocessor general register ( rt ).</td>
</tr>
<tr>
<td><strong>Dataword ( \leftarrow ) COP_SW(z, rt)</strong></td>
<td>( z ): The coprocessor unit number. ( rt ): Coprocessor general register specifier. Dataword: 32-bit word value. This defines the action taken by coprocessor ( z ) to supply a word of data during a store word operation. The action is coprocessor-specific. The typical action would be to supply the contents of low-order word in coprocessor general register ( rt ).</td>
</tr>
<tr>
<td><strong>Datadouble ( \leftarrow ) COP_SD(z, rt)</strong></td>
<td>( z ): The coprocessor unit number. ( rt ): Coprocessor general register specifier. Datadouble: 64-bit doubleword value. This defines the action taken by coprocessor ( z ) to supply a doubleword of data during a store doubleword operation. The action is coprocessor-specific. The typical action would be to supply the contents of the doubleword coprocessor general register ( rt ).</td>
</tr>
</tbody>
</table>
A.2.2.2 Load and Store Memory Pseudocode Functions

Regardless of byte-numbering order (endianness), the address of a halfword, word, or
doubleword is the smallest byte address among the bytes in the object. For a big-endian
ordering this is the most-significant byte; for a little-endian ordering this is the least-
significant byte.

In the operation description pseudocode for load and store operations, the functions listed
in Table A-3 are used to summarize the handling of virtual addresses and accessing
physical memory.

The size of the data item to be loaded or stored is passed in the AccessLength field. The
valid constant names and values are shown in Table A-4. The bytes within the addressed
unit of memory (quadword for 128-bit processors) which are used can be determined
directly from the AccessLength and the four low-order bits of the address.

Table A-3. Load and Store Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pAddr, CCA) ← AddressTranslation (vAddr, IorD, LorS)</td>
<td>Translate a virtual address to a physical address and a cache coherence algorithm describing the mechanism used to resolve the memory reference. Given the virtual address vAddr, and whether the reference is to Instructions or Data (IorD), find the corresponding physical address (pAddr) and the cache coherence algorithm (CCA) used to resolve the reference. If the virtual address is in one of the unmapped address spaces the physical address and CCA are determined directly by the virtual address. If the virtual address is in one of the mapped address spaces then the TLB is used to determine the physical address and access type; if the required translation is not present in the TLB or the desired access is not permitted the function fails and an exception is taken.</td>
</tr>
<tr>
<td>MemElem ← LoadMemory (CCA, AccessLength, pAddr, vAddr, IorD)</td>
<td>Load a value from memory. Uses the cache and main memory as specified in the Cache Coherence Algorithm (CCA) and the sort of access (IorD) to find the contents of AccessLength memory bytes starting at physical location pAddr. The data is returned in the fixed width naturally-aligned memory element (MemElem). The low-order two, three, or four bits of the address and the AccessLength indicate which of the bytes within MemElem needs to be given to the processor. If the memory access type of the reference is uncached then only the referenced bytes are read from memory and valid within the memory element. If the access type is cached, and the data is not present in cache, an implementation specific size and alignment block of memory is read and loaded into the cache to satisfy a load reference. At a minimum, the block is the entire memory element.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pAddr</td>
<td>Physical Address.</td>
</tr>
<tr>
<td>CCA</td>
<td>Cache Coherence Algorithm: the method used to access caches and memory and resolve the reference.</td>
</tr>
<tr>
<td>vAddr</td>
<td>Virtual Address.</td>
</tr>
<tr>
<td>IorD</td>
<td>Indicates whether access is for Instruction or Data.</td>
</tr>
<tr>
<td>LorS</td>
<td>Indicates whether access is for Load or Store</td>
</tr>
<tr>
<td>AccessLength</td>
<td>Length, in bytes, of access.</td>
</tr>
<tr>
<td>pAddr</td>
<td>Physical Address.</td>
</tr>
<tr>
<td>vAddr</td>
<td>Virtual Address.</td>
</tr>
<tr>
<td>IorD</td>
<td>Indicates whether access is for Instructions or Data.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemElem</td>
<td>Data is returned in a fixed width with a natural alignment. The width is the same size as the CPU general purpose register.</td>
</tr>
<tr>
<td>CCA</td>
<td>Cache Coherence Algorithm: the method used to access caches and memory and resolve the reference.</td>
</tr>
<tr>
<td>AccessLength</td>
<td>Length, in bytes, of access.</td>
</tr>
<tr>
<td>pAddr</td>
<td>Physical Address.</td>
</tr>
<tr>
<td>vAddr</td>
<td>Virtual Address.</td>
</tr>
<tr>
<td>IorD</td>
<td>Indicates whether access is for Instructions or Data.</td>
</tr>
</tbody>
</table>
StoreMemory (CCA, AccessLength, MemElem, pAddr, vAddr)

CCA: Cache Coherence Algorithm: the method used to access caches and memory and resolve the reference.

AccessLength: Length, in bytes, of access.

MemElem: Data in the width and alignment of a memory element. The width is the same size as the CPU general purpose register. For a partial-memory-element store, only the bytes that will be stored must be valid.

pAddr: Physical Address.

vAddr: Virtual Address.

Store a value to memory.
The specified data is stored into the physical location pAddr using the memory hierarchy (data caches and main memory) as specified by the Cache Coherence Algorithm (CCA). The MemElem contains the data for an aligned, fixed-width memory element, though only the bytes that will actually be stored to memory need to be valid. The low-order four bits of pAddr and the AccessLength field indicates which of the bytes within the MemElem data should actually be stored; only these bytes in memory will be changed.

Prefetch (CCA, pAddr, vAddr, DATA, hint)

CCA: Cache Coherence Algorithm: the method used to access caches and memory and resolve the reference.

pAddr: Physical Address.

vAddr: Virtual Address.

DATA: Indicates that access is for DATA.

hint: Hint that indicates the possible use of the data

Prefetch data from memory.
Prefetch is an advisory instruction for which an implementation specific action is taken. The action taken may increase performance but must not change the meaning of the program or alter architecturally-visible state.

Table A-4. AccessLength Specifications for Loads / Stores

<table>
<thead>
<tr>
<th>AccessLength name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUADWORD</td>
<td>15</td>
<td>16 bytes (128 bits)</td>
</tr>
<tr>
<td>DOUBLEWORD</td>
<td>7</td>
<td>8 bytes (64 bits)</td>
</tr>
<tr>
<td>SEPTIBYTE</td>
<td>6</td>
<td>7 bytes (56 bits)</td>
</tr>
<tr>
<td>SEXTIBYTE</td>
<td>5</td>
<td>6 bytes (48 bits)</td>
</tr>
<tr>
<td>QUINTIBYTE</td>
<td>4</td>
<td>5 bytes (40 bits)</td>
</tr>
<tr>
<td>WORD</td>
<td>3</td>
<td>4 bytes (32 bits)</td>
</tr>
<tr>
<td>TRIPLEBYTE</td>
<td>2</td>
<td>3 bytes (24 bits)</td>
</tr>
<tr>
<td>HALFWORD</td>
<td>1</td>
<td>2 bytes (16 bits)</td>
</tr>
<tr>
<td>BYTE</td>
<td>0</td>
<td>1 byte (8 bits)</td>
</tr>
</tbody>
</table>
A.2.2.3 Miscellaneous Functions

Table A-5 describes additional miscellaneous functions for CPU instruction descriptions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SyncOperation (stype)</td>
<td>Type of synchronization operation to be performed. Based on the value of stype either a memory barrier operation is performed or a pipeline barrier operation is performed. In case of a memory barrier all pending loads and stores are retired. Loads are retired when the destination register is written. Stores are retired when the stored data (in store buffers or write buffers) is either stored in the data cache, or sent on the processor bus. All uncached accelerated data gathering operation is terminated. The uncached accelerated buffer is invalidated. All bus read processes due to load/store/pref/cache instructions are completed. All pending bus write processes in the write back buffer are completed. In case of pipeline barrier all instructions prior to the barrier are completed before the instructions following the barrier operation are fetched. Note that the barrier operation does not wait for any instruction which was issued prior to the barrier operation but not retired (e.g., multiply, divide, multicycle COP1 operations or a pending load which were issued prior to the pipeline barrier operation).</td>
</tr>
<tr>
<td>SignalException (Exception)</td>
<td>The exception condition that exists. Signal an exception condition. This will result in an exception that aborts the instruction. The instruction operation pseudocode will never see a return from this function call.</td>
</tr>
<tr>
<td>UndefinedResult()</td>
<td>This function indicates that the result of the operation is undefined.</td>
</tr>
<tr>
<td>NullifyCurrentInstruction()</td>
<td>Nullify the current instruction. This occurs during the instruction time for some instruction and that instruction is not executed further. This appears for branch-likely instructions during the execution of the instruction in the delay slot and it kills the instruction in the delay slot.</td>
</tr>
<tr>
<td>CoprocessorOperation (z, cop_fun)</td>
<td>Coprocessor unit number Coprocessor function from function field of instruction Perform the specified Coprocessor operation.</td>
</tr>
</tbody>
</table>
A.3 CPU Instruction Formats

A CPU instruction is a single 32-bit aligned word. There are three instruction formats: Immediate (I-type), Jump (J-type), and Register (R-type). These formats are shown in Figure A-1 below:

### I-Type (Immediate)

```
31 26 25 21 20 16 15 0
op  rs  rt  immediate
6  5  5  16
```

### J-Type (Jump)

```
31 26 25 0
op  target
6  26
```

### R-Type (Register)

```
31 26 25 21 20 16 15 11 10 6 5 0
op  rs  rt  rd  sa  funct
6  5  5  5  5  6
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>6-bit primary operation code</td>
</tr>
<tr>
<td>rd</td>
<td>5-bit destination register specifier</td>
</tr>
<tr>
<td>rs</td>
<td>5-bit source register specifier</td>
</tr>
<tr>
<td>rt</td>
<td>5-bit target (source/destination) register specification or branch condition</td>
</tr>
<tr>
<td>immediate</td>
<td>16-bit signed immediate used for: logical operands, arithmetic signed operands, load/store address byte offsets, PC-relative branch signed instruction displacement</td>
</tr>
<tr>
<td>target</td>
<td>26-bit index shifted left two bits to supply the low-order 28 bits of the jump target address.</td>
</tr>
<tr>
<td>sa</td>
<td>5-bit shift amount</td>
</tr>
<tr>
<td>funct</td>
<td>6-bit function field used to specify functions within the primary operation code value SPECIAL</td>
</tr>
</tbody>
</table>

Figure A-1. CPU Instruction Formats
A.4 Instruction Descriptions

The user-level CPU instructions are described in alphabetical order in this section.
Appendix A  CPU Instruction Set Details

ADD

<table>
<thead>
<tr>
<th>SPECIAL</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
<th>ADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td>000000</td>
<td>100000</td>
</tr>
</tbody>
</table>

### MIPS I

**Format:** ADD rd, rs, rt

**Purpose:** To add 32-bit integers. If overflow occurs, then trap.

**Description:**

\[ rd \leftarrow rs + rt \]

The 32-bit word value in GPR rt is added to the 32-bit value in GPR rs to produce a 32-bit result. If the addition results in 32-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR rd.

**Restrictions:**

If either GPR rt or GPR rs do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

**Operation:**

\[
\begin{align*}
\text{If } \neg \text{WordValue(GPR[rs]_{63.0}) or } \neg \text{WordValue(GPR[rt]_{63.0}) then UndefinedResult();} \\
\text{endif} \\
\text{temp } \leftarrow \text{GPR[rs]_{63.0} + GPR[rt]_{63.0}} \\
\text{if 32_bit_arithmetic_overflow then} \\
\quad \text{SignalException(IntegerOverflow)} \\
\text{else} \\
\quad \text{GPR[rd]_{63.0} } \leftarrow \text{sign_extend(temp31.0)} \\
\text{endif}
\end{align*}
\]

**Exceptions:**

Integer Overflow

**Programming Notes:**

ADDU performs the same arithmetic operation but, does not trap on overflow.
ADDI

Add Immediate Word

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format: ADDI rt, rs, immediate

Purpose: To add a constant to a 32-bit integer. If overflow occurs, then trap.

Description: rt ← rs + immediate

The 16-bit signed immediate is added to the 32-bit value in GPR rs to produce a 32-bit result. If the addition results in 32-bit 2’s complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR rt.

Restrictions:

If GPR rs does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is undefined.

Operation:

if (NotWordValue (GPR[rs] 63..0)) then UndefinedResult() endif

temp ← GPR[rs] 63..0 + sign_extend (immediate)

if (32_bit_arithmetic_overflow) then
    SignalException (IntegerOverflow)
else
    GPR[rt] 63..0 ← sign_extend (temp 31..0)
endif

Exceptions:

Integer Overflow

Programming Notes:

ADDIU performs the same arithmetic operation but, does not trap on overflow.
ADDIU

Add Immediate Unsigned Word

MIPS I

Format: ADDIU rt, rs, immediate
Purpose: To add a constant to a 32-bit integer.
Description: rt ← rs + immediate

The 16-bit signed immediate is added to the 32-bit value in GPR rs and the 32-bit arithmetic result is placed into GPR rt.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

If GPR rs does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is undefined.

Operation:

if (NotWordValue (GPR[rs]63..0)) then UndefinedResult( ) endif

temp ← GPR[rs]63..0 + sign_extend (immediate)
GPR[rt]63..0 ← sign_extend (temp31..0)

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.
### ADDU

#### Add Unsigned Word

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>ADDU</td>
<td>000000</td>
<td>100001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### MIPS I

- **Format:** ADDU rd, rs, rt
- **Purpose:** To add 32-bit integers.
- **Description:** rd ← rs + rt
  
  The 32-bit word value in GPR rt is added to the 32-bit value in GPR rs and the 32-bit arithmetic result is placed into GPR rd. No Integer Overflow exception occurs under any circumstances.

- **Restrictions:**
  
  If either GPR rt or GPR rs do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

- **Operation:**
  
  \[
  \text{temp} \leftarrow \text{GPR[rs]}_{63.0} + \text{GPR[rt]}_{63.0} \\
  \text{GPR[rt]}_{63.0} \leftarrow \text{sign\_extend} (\text{temp}_{31.0})
  \]

- **Exceptions:**
  
  None

- **Programming Notes:**
  
  The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.
AND

MIPS I

Format: AND rd, rs, rt
Purpose: To do a bitwise logical AND.
Description: rd ← rs AND rt

The contents of GPR rs are combined with the contents of GPR rt in a bitwise logical AND operation. The result is placed into GPR rd.
Restrictions:
None
Operation:
GPR[rd]_{63..0} ← GPR[rs]_{63..0} and GPR[rt]_{63..0}
Exceptions:
None
Programming Notes:
None
ANDI
And Immediate

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANDI</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001100</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

**Format:**
ANDI rt, rs, immediate

**Purpose:**
To do a bitwise logical AND with a constant.

**Description:**
rt ← rs AND immediate

The 16-bit immediate is zero-extended to the left and combined with the contents of GPR rs in a bitwise logical AND operation. The result is placed into GPR rt.

**Restrictions:**
None

**Operation:**

\[
\text{GPR}[rt]_{63..0} \leftarrow \text{zeroextend}(\text{immediate}) \text{ and } \text{GPR}[rs]_{63..0}
\]

**Exceptions:**
None

**Programming Notes:**
None
BEQ Branch on Equal

<table>
<thead>
<tr>
<th>BEQ 000100</th>
<th>rs</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 26 25 21 20 16 15 0</td>
<td>6 5 5 16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format: BEQ rs, rt, offset

Purpose: To compare GPRs then do a PC-relative conditional branch.

Description: if (rs = rt) then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs and GPR rt are equal, branch to the effective target address after the instruction in the delay slot is executed.

Restriction:
None

Operation:

I: tgt_offset ← sign_extend (offset || 0^2)
condition ← (GPR[rs]_63..0 = GPR[rt]_63..0)
I+1: if condition then
    PC ← PC + tgt_offset
endif

Exceptions:
None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.
**BEQL**

**Branch on Equal Likely**

### MIPS II

**Format:**

BEQL rs, rt, offset

**Purpose:**

To compare GPRs then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

**Description:**

if (rs = rt) then branch_likely

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs and GPR rt are equal, branch to the target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**

None

**Operation:**

I: tgt_offset ← sign_extend (offset || 0²)
  condition ← (GPR[rs]63..0 = GPR[rt]63..0)
  I+1: if condition then
  PC ← PC + tgt_offset
  else
  NullifyCurrentInstruction()
  endif

**Exceptions:**

None

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.
### BGEZ

**Branch on Greater Than or Equal to Zero**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGIMM</td>
<td>rs</td>
<td>BGEZ</td>
<td>00001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### MIPS I

**Format:** BGEZ rs, offset

**Purpose:** To test a GPR then do a PC-relative conditional branch.

**Description:**

if (rs ≥ 0) then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**

None

**Operation:**

I: tgt_offset ← sign_extend (offset || 0²)

condition ← GPR[rs]₂⁶⁻¹ ≥ 0²⁶⁻¹

I+1: if condition then

PC ← PC + tgt_offset

endif

**Exceptions:**

None

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.
**BGEZAL**

**Branch on Greater Than or Equal to Zero and Link**

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>BGEZAL 10001</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:**

BGEZAL rs, offset

**Purpose:**

To test a GPR then do a PC-relative conditional procedure call.

**Description:**

if (rs ≥ 0) then procedure_call

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution would continue after a procedure call.

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed.

**Restriction:**

GPR 31 must not be used for the source register rs, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

**Operation:**

I: tgt_offset ← sign_extend (offset || 0^2)
   condition ← GPR[rs]_63..0 ≥ 0^GPRLEN
   GPR[31]_63..0 ← zero_extend (PC+8)
   I+1: if condition then
         PC ← PC + tgt_offset
         endif

**Exceptions:**

None

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to more distant addresses.
**BGEZALL**  
Branch on Greater Than or Equal to Zero and Link Likely

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>BGEZALL</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td>10011</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS II**

**Format:**  
BGEZALL rs, offset

**Purpose:**  
To test a GPR then do a PC-relative conditional procedure call; execute the delay slot only if the branch is taken.

**Description:**  
if (rs ≥ 0) then procedure_call_likely

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution would continue after a procedure call.

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**

GPR 31 must not be used for the source register rs, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

**Operation:**

I:  
tgt_offset ← sign_extend (offset || 0^2)  
condition ← GPR[rs]_63..0 ≥ 0^GPRLEN  
GPR[31]_63..0 ← zero_extend (PC+8)

I+1: if condition then  
PC ← PC + tgt_offset  
else  
NullifyCurrentInstruction()  
endif

**Exceptions:**

None

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump and link (JAL) or jump and link register (J ALR) instructions for procedure calls to more distant addresses.
BGEZL  Branch on Greater Than or Equal to Zero Likely

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>BGEZL</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td></td>
<td>00011</td>
<td></td>
</tr>
</tbody>
</table>

MIPS II

Format: BGEZL  rs, offset

Purpose: To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if (rs $\geq$ 0) then branch Likely

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:
None

Operation:
I: tgt_offset $\leftarrow$ sign_extend (offset $||$ 0^2)
   condition $\leftarrow$ GPR[rs]_{63..0} $\geq$ 0^{GPRLEN}
I+1: if condition then
      PC $\leftarrow$ PC + tgt_offset
   else
      NullifyCurrentInstruction()
endif

Exceptions:
None

Programming Notes:
With the 18-bit signed instruction offset, the conditional branch range is $\pm$ 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.
**BGTZ**  
Branch on Greater Than Zero

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGTZ</td>
<td>rs</td>
<td>0</td>
<td>00000</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:**  
BGTZ rs, offset

**Purpose:**  
To test a GPR then do a PC-relative conditional branch.

**Description:**  
if (rs > 0) then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are greater than zero (sign bit is 0 but value not zero), branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**

None

**Operation:**

I: \( tgt\_offset \leftarrow \text{sign}\_\text{extend} (\text{offset} \mid 0^2) \)  
condition \( \leftarrow \text{GPR}[rs]_{63..0} > 0^{\text{GPRLEN}} \)  
I+1: if condition then  
\( \text{PC} \leftarrow \text{PC} + tgt\_offset \)  
endif

**Exceptions:**

None

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch range is \( \pm 128 \) KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.
Branch on Greater Than Zero Likely

**BGTZL**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>16</th>
<th>15</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGTZL</td>
<td>rs</td>
<td>00000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BGTZL 010111 offset rs 0

**MIPS II**

**Format:**

BGTZL  rs, offset

**Purpose:**

To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

**Description:**

if (rs > 0) then branch _likely_

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are greater than zero (sign bit is 0 but value not zero), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**

None

**Operations:**

1: tgt_offset ← sign_extend (offset || 0^2)  
   condition ← GPR[rs]_{63..0} > 0^{GPRLEN}  
   if condition then  
   PC ← PC + tgt_offset  
   else  
   NullifyCurrentInstruction()  
   endif

**Exceptions:**

None

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch is ±128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.
BLEZ  Branch on Less Than or Equal to Zero  BLEZ

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:** BLEZ rs, offset

**Purpose:** To test a GPR then do a PC-relative conditional branch.

**Description:**

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of the GPR rs are less than or equal to zero (sign bit is 1 or value is zero), branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**

None

**Operation:**

I: $\text{tgt.offset} \leftarrow \text{sign.extend} (\text{offset} || 0^2)$

condition $\leftarrow \text{GPR[rs]}_{63..0} \leq 0^{\text{GPRLEN}}$

I+1: if condition then

$\text{PC} \leftarrow \text{PC} + \text{tgt.offset}$

endif

**Exceptions:**

None

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch range is $\pm 128$ KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.
**BLEZL**

**Branch on Less Than or Equal to Zero Likely**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLEZL</td>
<td>rs</td>
<td>0</td>
<td>00000</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS II**

**Format:** BLEZL rs, offset

**Purpose:** To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

**Description:** if (rs ≤ 0) then branch_likely

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are less than or equal to zero (sign bit is 1 or value is zero), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**

None

**Operation:**

I:  
\[ \text{tgt\_offset} \leftarrow \text{sign\_extend} (\text{offset} \| 0^2) \]
\[ \text{condition} \leftarrow \text{GPR}[\text{rs}]_{63..0} \leq 0^{\text{GPRLEN}} \]

I+1: if condition then

\[ \text{PC} \leftarrow \text{PC} + \text{tgt\_offset} \]
else

NullifyCurrentInstruction()
endif

**Exceptions:**

None

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.
Appendix A  CPU Instruction Set Details

BLTZ  
Branch on Less Than Zero

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGIMM</td>
<td>rs</td>
<td>BLTZ</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001</td>
<td>00000</td>
<td>00000</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format: BLTZ  rs, offset

Purpose: To test a GPR then do a PC-relative conditional branch.

Description: if (rs < 0) then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:
None

Operation:

I: tgt_offset ← sign_extend (offset || 0^2)
   condition ← GPR[rs]63..0 < 0^GPRLEN

I+1: if condition then
      PC ← PC + tgt_offset
      endif

Exceptions:
None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.
BLTZAL

Branch on Less Than Zero and Link

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>BLTZAL</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td>10000</td>
<td>000001</td>
<td>10000</td>
</tr>
</tbody>
</table>

MIPS I

Format: BLTZAL rs, offset
Purpose: To test a GPR then do a PC-relative conditional procedure call.
Description: if (rs < 0) then procedure_call

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch (not the branch itself), where execution would continue after a procedure call.

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch, in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

GPR 31 must not be used for the source register rs, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

Operation:

I: tgt_offset ← sign_extend (offset || 0^2)
   condition ← GPR[rs]_{63..0} < 0_{GPRLEN}
   GPR[31]_{63..0} ← zero_extend (PC+8)
I+1: if condition then
      PC ← PC + tgt_offset
      endif

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to more distant addresses.
### BLTZALL

**Branch on Less Than Zero and Link Likely**

<table>
<thead>
<tr>
<th></th>
<th>REGIMM</th>
<th>rs</th>
<th>BLTZALL</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>000001</td>
<td>0</td>
<td>10010</td>
<td>0000</td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>25</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>21</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>20</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>16</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>15</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>14</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>13</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>12</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>11</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>10</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>9</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>8</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>7</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>6</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>4</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

**MIPS II**

**Format:**

```
BLTZALL  rs, offset
```

**Purpose:**

To test a GPR then do a PC-relative conditional procedure call; execute the delay slot only if the branch is taken.

**Description:**

if (rs < 0) then procedure_call_likely

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch (not the branch itself), where execution would continue after a procedure call.

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch, in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**

GPR 31 must not be used for the source register rs, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

**Operation:**

```
I:   tgt_offset ← sign_extend (offset || 02)
     condition ← GPR[rs]63..0 < 0GPRLEN
     GPR[31]63..0 ← zero_extend (PC+8)
I+1: if condition then
     PC ← PC + tgt_offset
     else
     NullifyCurrentInstruction()
     endif
```

**Exceptions:**

None

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch range ± 128 KB. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to more distant addresses.
Appendix A  CPU Instruction Set Details

BLTZL  Branch on Less Than Zero Likely  BLTZL

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGIMM</td>
<td>rs</td>
<td>BLTZL</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td>00010</td>
<td>000001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6  5  5  16

MIPS II

Format:  BLTZL  rs, offset

Purpose:  To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description:  if (rs < 0) then branch_likely

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

None

Operation:

1:  tgt_offset ← sign_extend (offset || 0^2)
    condition ← GPR[rs]_63..0 < 0^GPRLEN

1+1:  if condition then
       PC ← PC + tgt_offset
   else
       NullifyCurrentInstruction()
   endif

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.
BNE Branch on Not Equal

BNE

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNE</td>
<td>rs</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000101</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format: BNE rs, rt, offset

Purpose: To compare GPRs then do a PC-relative conditional branch.

Description: if (rs ≠ rt) then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs and GPR rt are not equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

None

Operation:

I: tgt_offset ← sign_extend (offset || 0²)
   condition ← (GPR[rs]₆₃..₀ ≠ GPR[rt]₆₃..₀)
I+1: if condition then
      PC ← PC + tgt_offset
      endif

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.
BNEL

Branch on Not Equal Likely

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNEL</td>
<td>rs</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010101</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS II

Format: BNEL rs, rt, offset

Purpose: To compare GPRs then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if (rs ≠ rt) then branch likely

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs and GPR rt are not equal, branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:
None

Operation:

I: tgt_offset ← sign_extend (offset || 0^2)
   condition ← (GPR[rs]63..0 ≠ GPR[rt]63..0)
I+1: if condition then
      PC ← PC + tgt_offset
   else
      NullifyCurrentInstruction()
   endif

Exceptions:
None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.
### MIPS I

**Format:** BREAK

**Purpose:** To cause a Breakpoint exception.

**Description:**

A breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

**Restrictions:**

None

**Operation:**

SignalException (Breakpoint)

**Exceptions:**

Breakpoint

**Programming Notes:**

None
**DADD**

**Doubleword Add**

<table>
<thead>
<tr>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
<th>DADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>00000</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>101100</td>
</tr>
</tbody>
</table>

**MIPS III**

**Format:**

DADD  rd, rs, rt

**Purpose:**

To add 64-bit integers. If overflow occurs, then trap.

**Description:**

rd ← rs + rt

The 64-bit doubleword value in GPR rt is added to the 64-bit value in GPR rs to produce a 64-bit result. If the addition results in 64-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 64-bit result is placed into GPR rd.

**Restrictions:**

None

**Operation:**

```
temp ← GPR[rs]_{63..0} + GPR[rt]_{63..0}  
if (64_bit_arithmetic_overflow) then  
    SignalException (IntegerOverflow)  
else  
    GPR[rd]_{63..0} ← temp  
endif
```

**Exceptions:**

Integer Overflow

**Programming Notes:**

DADDU performs the same arithmetic operation but, does not trap on overflow.
**DADDI**

**Doubleword Add Immediate**

<table>
<thead>
<tr>
<th>DADDI</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>011000</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

**MIPS III**

**Format:**
DADDI rt, rs, immediate

**Purpose:**
To add a constant to a 64-bit integer. If overflow occurs, then trap.

**Description:**
rt $\leftarrow$ rs + immediate

The 16-bit signed *immediate* is added to the 64-bit value in GPR rs to produce a 64-bit result. If the addition results in 64-bit 2’s complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 64-bit result is placed into GPR rt.

**Restrictions:**
None

**Operation:**

```plaintext```
temp $\leftarrow$ GPR[rs] 63..0 + sign_extend (immediate)
if (64_bit_arithmetic_overflow) then
    SignalException (IntegerOverflow)
else
    GPR[rt] 63..0 $\leftarrow$ temp
endif
```

**Exceptions:**
Integer Overflow

**Programming Notes:**

DADDIU performs the same arithmetic operation but, does not trap on overflow.
Appendix A  CPU Instruction Set Details

DADDIU  Doubleword Add Immediate Unsigned

<table>
<thead>
<tr>
<th>DADDIU 011001</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 26 25 21 20 16 15 0</td>
<td>6 5 5 16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS III

Format: DADDIU  rt, rs, immediate

Purpose: To add a constant to a 64-bit integer.

Description: rt ← rs + immediate

The 16-bit signed immediate is added to the 64-bit value in GPR rs and the 64-bit arithmetic result is placed into GPR rt.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

GPR[rt]_{63..0} ← GPR[rs]_{63..0} + sign_extend (immediate)

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 64-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.
**DADDU**

Doubleword Add Unsigned

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPECIAL</strong></td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td><strong>DADDU</strong></td>
<td>101101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS III**

**Format:**
DADDU rd, rs, rt

**Purpose:**
To add 64-bit integers.

**Description:**
rd ← rs + rt

The 64-bit doubleword value in GPR rt is added to the 64-bit value in GPR rs and the 64-bit arithmetic result is placed into GPR rd.

No Integer Overflow exception occurs under any circumstances.

**Restrictions:**
None

**Operation:**
GPR[rd]_{63..0} ← GPR[rs]_{63..0} + GPR[rt]_{63..0}

**Exception:**
None

**Programming Notes:**
The term “unsigned” in the instruction name is a misnomer; this operation is 64-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.
DIV
Divide Word

<table>
<thead>
<tr>
<th>DIV</th>
<th>SPECIAL</th>
<th>rs</th>
<th>rt</th>
<th>0</th>
<th>00 0000 0000</th>
<th>011010</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
<td>15</td>
</tr>
</tbody>
</table>

MIPS I

Format: \text{DIV rs, rt}

Purpose: To divide 32-bit signed integers.

Description: \((LO, HI) \leftarrow rs / rt\)

The 32-bit word value in GPR \(rs\) is divided by the 32-bit value in GPR \(rt\), treating both operands as signed values. The 32-bit quotient is placed into special register \(LO\) and the 32-bit remainder is placed into special register \(HI\).

No arithmetic exception occurs under any circumstances.

Restrictions:

If either GPR \(rt\) or GPR \(rs\) do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

If the divisor in GPR \(rt\) is zero, the arithmetic result value is undefined.

Operation:

\[
\begin{align*}
\text{if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif} \\
q & \leftarrow \text{GPR}[rs]_{31.0} \div \text{GPR}[rt]_{31.0} \\
\text{LO}_{63.0} & \leftarrow \text{sign\_extend (q}_{31.0}) \\
r & \leftarrow \text{GPR}[rs]_{31.0} \mod \text{GPR}[rt]_{31.0} \\
\text{HI}_{63.0} & \leftarrow \text{sign\_extend (r}_{31.0})
\end{align*}
\]

Exceptions:

None

Supplementary Explanation:

Normally, when 0x80000000 (-2147483648) the signed minimum value is divided by 0xFFFFFFFF (-1), the operation will result in an overflow. However, in this instruction an overflow exception doesn’t occur and the result will be as follows:

Quotient is 0x80000000 (-2147483648), and remainder is 0x00000000 (0).

This sign of the quotient and the remainder is based on the signs of the dividend and the divisor as shown in the table below:
Programming Notes:

In the C790, the integer divide operation proceeds asynchronously and allows other CPU instructions to execute before it is retired. An attempt to read LO or HI before the results are written will wait (interlock) until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the divide so that other instructions can execute in parallel.

No arithmetic exception occurs under any circumstances. If divide-by-zero or overflow conditions should be detected and some action taken, then the divide instruction is typically followed by additional instructions to check for a zero divisor and / or for overflow. If the divide is asynchronous then the zero-divisor check can execute in parallel with the divide. The action taken on either divide-by-zero or overflow is either a convention within the program itself or more typically, the system software; one possibility is to take a BREAK exception with a code field value to signal the problem to the system software.

As an example, the C programming language in a UNIX environment expects division by zero to either terminate the program or execute a program-specified signal handler. C does not expect overflow to cause any exceptional condition. If the C compiler uses a divide instruction, it also emits code to test for a zero divisor and execute a BREAK instruction to inform the operating system if one is detected.

In the C790, sign-extended 32-bit values (bits 63..31) are ignored on divide operation.
### DIVU

**Divide Unsigned Word**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>0</td>
<td>00 0000 0000</td>
<td>DIVU</td>
<td>011011</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:**

```
DIVU  rs, rt
```

**Purpose:**

To divide 32-bit unsigned integers.

**Description:**

```
(LO, HI) ← rs / rt
```

The 32-bit word value in GPR rs is divided by the 32-bit value in GPR rt, treating both operands as unsigned values. The 32-bit quotient is placed into special register LO and the 32-bit remainder is placed into special register HI.

No arithmetic exception occurs under any circumstances.

**Restrictions:**

- If either GPR rt or GPR rs do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.
- If the divisor in GPR rt is zero, the arithmetic result is undefined.

**Operation:**

```
if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
q ← (0 || GPR[rs]31..0) div (0 || GPR[rt]31..0)
LO63..0 ← sign_extend (q31..0)
r ← (0 || GPR[rs]31..0) mod (0 || GPR[rt]31..0)
HI63..0 ← sign_extend (r31..0)
```

**Exceptions:**

None

**Programming Notes:**

See the Programming Notes for the DIV instruction.
Appendix A  CPU Instruction Set Details

DSLL
Doubleword Shift Left Logical

<table>
<thead>
<tr>
<th>SPECIAL</th>
<th>000000</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>DSLL 110000</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

MIPS III

Format: \[\text{DSLL \ rd, rt, sa}\]

Purpose: To left shift a doubleword by a fixed amount — 0 to 31 bits.

Description: \[\text{rd} \leftarrow \text{rt} \ll \text{sa}\]

The 64-bit doubleword contents of GPR \(rt\) are shifted left, inserting zeros into the emptied bits; the result is placed in GPR \(rd\). The bit shift count in the range 0 to 31 is specified by \(sa\).

Restrictions:

None

Operation:

\[s \leftarrow 0 \ll \text{sa}\]
\[\text{GPR}[rd]_{63..0} \leftarrow \text{GPR}[rt]_{(63-s)..0} \ll 0^s\]

Exceptions:

None

Programming Notes:

None
DSLL32
Doubleword Shift Left Logical Plus 32

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>0</td>
<td>00000</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>DSLL32</td>
<td>111100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS III

Format: DSLL32 rd, rt, sa

Purpose: To left shift a doubleword by a fixed amount — 32 to 63 bits.

Description: rd ← rt << (sa + 32)

The 64-bit doubleword contents of GPR rt are shifted left, inserting zeros into the emptied bits; the result is placed in GPR rd. The bit shift count in the range 32 to 63 is specified by sa + 32.

Restrictions:
None

Operation:
s ← 1 || sa /* 32 + sa */
GPR[rd]63..0 ← GPR[rt](63-s)..0 || 0s

Exceptions:
None

Programming Notes:
None
**DSLLV**  
**Doubleword Shift Left Logical Variable**  

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>000000</td>
<td>DSLLV</td>
<td>010100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### MIPS III

**Format:** DSLLV rd, rt, rs  
**Purpose:** To left shift a doubleword by a variable number of bits.  
**Description:**  
\[ \text{rd} \leftarrow \text{rt} \ll \text{rs} \]  
The 64-bit doubleword contents of GPR rt are shifted left, inserting zeros into the emptied bits; the result is placed in GPR rd. The bit shift count in the range 0 to 63 is specified by the low-order six bits in GPR rs.  

**Restrictions:**  
None  

**Operation:**  
\[ s \leftarrow 0 \parallel \text{GPR}[rs]_{5:0} \] \[ \text{GPR}[rd]_{63:0} \leftarrow \text{GPR}[rt]_{63:5} \parallel 0^6 \]  

**Exceptions:**  
None  

**Programming Notes:**  
None
Appendix A  CPU Instruction Set Details

DSRA  Doubleword Shift Right Arithmetic  DSRA

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rt</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sa</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSRA</td>
<td>111011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS III

Format: DSRA  rd, rt, sa
Purpose: To arithmetic right shift a doubleword by a fixed amount — 0 to 31 bits.
Description: rd ← rt >> sa  (arithmetic)

The 64-bit doubleword contents of GPR rt are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR rd. The bit shift count in the range 0 to 31 is specified by sa.

Restrictions:
None

Operation:
s ← 0 || sa
GPR[rd]_{63..0} ← (GPR[rt]_{63})^s || GPR[rt]_{63..5}

Exceptions:
None

Programming Notes:
None
### DSRA32

**Doubleword Shift Right Arithmetic Plus 32**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>DSRA32</td>
<td>111111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS III**

**Format:**
DSRA32 rd, rt, sa

**Purpose:**
To arithmetic right shift a doubleword by a fixed amount — 32-63 bits.

**Description:**
rd ← rt >> (sa + 32)  (arithmetic)

The doubleword contents of GPR rt are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR rd. The bit shift count in the range 32 to 63 is specified by sa + 32.

**Restrictions:**
None

**Operation:**
\[
\text{s} \leftarrow 1 \parallel \text{sa} \quad /!* 32+\text{sa} */\n\]
\[
\text{GPR}[\text{rd}]_{63..0} \leftarrow (\text{GPR}[\text{rt}]_{63})^s \parallel \text{GPR}[\text{rt}]_{63..5}
\]

**Exceptions:**
None

**Programming Notes:**
None
### DSRAV

**Doubleword Shift Right Arithmetic Variable**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs</td>
<td></td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td></td>
<td>DSRAV</td>
<td>010111</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

#### MIPS III

**Format:**

DSRAV rd, rt, rs

**Purpose:**

To arithmetic right shift a doubleword by a variable number of bits.

**Description:**

rd ← rt >> rs (arithmetic)

The doubleword contents of GPR rt are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR rd. The bit shift count in the range 0 to 63 is specified by the low-order six bits in GPR rs.

**Restrictions:**

None

**Operation:**

s ← GPR[rs]5..0
GPR[rd]63..0 ← (GPR[rt]63)\(^s\) || GPR[rt]63..s

**Exceptions:**

None

**Programming Notes:**

None
DSRL Doubleword Shift Right Logical

<table>
<thead>
<tr>
<th>SPECIAL</th>
<th>000000</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>DSRL</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>111010</td>
</tr>
</tbody>
</table>

MIPS III

Format: DSRL rd, rt, sa

Purpose: To logical right shift a doubleword by a fixed amount — 0 to 31 bits.

Description: rd ← rt >> sa (logical)

The doubleword contents of GPR rt are shifted right, inserting zeros into the emptied bits; the result is placed in GPR rd. The bit shift count in the range 0 to 31 is specified by sa.

Restrictions:
None

Operation:

s ← 0 || sa
GPR[rd]_{63..0} ← 0^s || GPR[rt]_{63..s}

Exceptions:
None

Programming Notes:
None
DSRL32  Doubleword Shift Right Logical Plus 32

MIPS III

Format: DSRL32 rd, rt, sa

Purpose: To logical right shift a doubleword by a fixed amount — 32 to 63 bits.

Description: rd ← rt >> (sa + 32) (logical)

The 64-bit doubleword contents of GPR rt are shifted right, inserting zeros into the emptied bits; the result is placed in GPR rd. The bit shift count in the range 32 to 63 is specified by sa + 32.

Restrictions:
None

Operation:

s ← 1 || sa /* 32 + sa */

GPR[rd]_{63..0} ← 0^s || GPR[rt]_{63..5}

Exceptions:
None

Programming Notes:
None
**DSRLV**

Doubleword Shift Right Logical Variable

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL 000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0 00000</td>
<td>DSRLV 010110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS III**

**Format:**

DSRLV rd, rt, rs

**Purpose:**

To logical right shift a doubleword by a variable number of bits.

**Description:**

\[
rd \leftarrow rt \gg rs \quad \text{(logical)}
\]

The 64-bit doubleword contents of GPR rt are shifted right, inserting zeros into the emptied bits; the result is placed in GPR rd. The bit shift count in the range 0 to 63 is specified by the low-order six bits in GPR rs.

**Restrictions:**

None

**Operation:**

\[
s \leftarrow \text{GPR}[rs]_{5..0}
\]

\[
\text{GPR}[rd]_{63..0} \leftarrow 0^s \parallel \text{GPR}[rt]_{63..s}
\]

**Exceptions:**

None

**Programming Notes:**

None
### DSUB

#### Doubleword Subtract

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>000000</td>
<td>DSUB</td>
<td>101110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### MIPS III

**Format:**

DSUB rd, rs, rt

**Purpose:**

To subtract 64-bit integers; trap if overflow.

**Description:**

\[ \text{rd} \leftarrow \text{rs} - \text{rt} \]

The 64-bit doubleword value in GPR rt is subtracted from the 64-bit value in GPR rs to produce a 64-bit result. If the subtraction results in 64-bit 2’s complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 64-bit result is placed into GPR rd.

**Restrictions:**

None

**Operation:**

\[
\text{temp} \leftarrow \text{GPR[rs]}_{63.0} - \text{GPR[rt]}_{63.0}
\]

if (64_bit_arithmetic_overflow) then
  SignalException (IntegerOverflow)
else
  \[ \text{GPR[rd]}_{63.0} \leftarrow \text{temp} \]
endif

**Exceptions:**

Integer Overflow

**Programming Notes:**

DSUBU performs the same arithmetic operation but, does not trap on overflow.
## DSUBU

**Doubleword Subtract Unsigned**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPECIAL</strong></td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>rs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>rt</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>rd</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>0</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000000</td>
</tr>
<tr>
<td><strong>DSUBU</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>101111</td>
</tr>
</tbody>
</table>

### MIPS III

**Format:**

DSUBU  rd, rs, rt

**Purpose:**

To subtract 64-bit integers.

**Description:**

rd ← rs - rt

The 64-bit doubleword value in GPR rt is subtracted from the 64-bit value in GPR rs and the 64-bit arithmetic result is placed into GPR rd.

**No Integer Overflow exception occurs under any circumstances.**

**Restrictions:**

None

**Operation:**

GPR[rd]_{63..0} ← GPR[rs]_{63..0} - GPR[rt]_{63..0}

**Exceptions:**

None

**Programming Notes:**

The term “unsigned” in the instruction name is a misnomer; this operation is 64-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.
Appendix A  CPU Instruction Set Details

J
Jump

\[
\begin{array}{c|c|c|c|c}
31 & 26 & 25 & 0 \\
\hline
J & 000010 & instr_index \\
6 & 26 \\
\end{array}
\]

MIPS I

Format: J target

Purpose: To branch within the current 256 MB aligned region.

Description:
This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256 MB aligned region. The low 28 bits of the target address is the instr_index field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the jump itself).

Jump to the effective target address. Execute the instruction following the jump, in the branch delay slot, before jumping.

Restrictions:
None

Operation:
I:
I+1: PC ← PC_{31..28} || instr_index || 0^2

Exceptions:
None

Programming Notes:

Forming the branch target address by concatenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch to anywhere in the region from anywhere in the region which a signed relative offset would not allow.

This definition creates the boundary case where the branch instruction is in the last word of a 256 MB region and can therefore only branch to the following 256 MB region containing the branch delay slot.
JAL

Jump and Link

```
JAL
000011
```

**MIPS I**

**Format:** JAL target

**Purpose:** To procedure call within the current 256 MB aligned region.

**Description:**

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution would continue after a procedure call.

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256 MB aligned region. The low 28 bits of the target address is the `instr_index` field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the jump itself).

Jump to the effective target address. Execute the instruction following the jump, in the branch delay slot, before jumping.

**Restrictions:**

None

**Operation:**

I: \[ \text{GPR}[31]_{63..0} \leftarrow \text{zero\_extend} (\text{PC} + 8) \]

I+1: \[ \text{PC} \leftarrow \text{PC}_{31..28} || \text{instr\_index} || 0^2 \]

**Exceptions:**

None

**Programming Notes:**

Forming the branch target address by concatenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch to anywhere in the region from anywhere in the region which a signed relative offset would not allow.

This definition creates the boundary case where the branch instruction is in the last word of a 256 MB region and can therefore only branch to the following 256 MB region containing the branch delay slot.
### JALR

**Jump and Link Register**

**Format:**
- JALR rs
  - (rd = 31 implied)
- JALR rd, rs

**Purpose:** To procedure call to an instruction address in a register.

**Description:**
- \( \text{rd} \leftarrow \text{return_addr} \)
- \( \text{PC} \leftarrow \text{rs} \)

Place the return address link in GPR \( \text{rd} \). The return link is the address of the second instruction following the branch, where execution would continue after a procedure call.

Jump to the effective target address in GPR \( \text{rs} \). Execute the instruction following the jump, in the branch delay slot, before jumping.

**Restrictions:**
- Register specifiers \( \text{rs} \) and \( \text{rd} \) must not be equal, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.
- The effective target address in GPR \( \text{rs} \) must be naturally aligned. If either of the two least-significant bits are not -zero, then an Address Error exception occurs, not for the jump instruction, but when the branch target is subsequently fetched as an instruction.

**Operation:**
- \( \text{I} \)
  - \( \text{temp} \leftarrow \text{GPR[rs]_31.0} \)
  - \( \text{GPR[rd]_63.0} \leftarrow \text{zero_extend (PC + 8)} \)
- \( \text{I+1} \)
  - \( \text{PC} \leftarrow \text{temp} \)

**Exceptions:**
- None

**Programming Notes:**
- This is the only branch-and-link instruction that can select a register for the return link; all other link instructions use GPR 31. The default register for GPR \( \text{rd} \), if omitted in the assembly language instruction, is GPR 31.
### JR

**Jump Register**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>6</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>0</td>
<td>000000</td>
<td>000000 0000 0000</td>
<td>001000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>15</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:** JR rs

**Purpose:** To branch to an instruction address in a register.

**Description:**

PC ← rs

Jump to the effective target address in GPR rs. Execute the instruction following the jump, in the branch delay slot, before jumping.

**Restrictions:**

The effective target address in GPR rs must be naturally aligned. If either of the two least-significant bits are not-zero, then an Address Error exception occurs, not for the jump instruction, but when the branch target is subsequently fetched as an instruction.

**Operation:**

I: temp ← GPR[rs]_{31..0}
I+1: PC ← temp

**Exceptions:**

None

**Programming Notes:**

None
LB  Load Byte

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100000</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format:   LB  rt, offset (base)
Purpose:  To load a byte from memory as a signed value.
Description:  rt ← memory [base + offset]

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, sign-extended, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

Restrictions:  None

Operation:  (128-bit bus)

\[
\text{vAddr} \leftarrow \text{sign}_\text{extend}(\text{offset}) + \text{GPR[base]}_{31..0} \\
(\text{pAddr, uncached}) \leftarrow \text{AddressTranslation}(\text{vAddr, DATA, LOAD}) \\
\text{pAddr} \leftarrow \text{pAddr}_{(\text{PSIZE}-1)\ldots 4} || (\text{pAddr}_{3..0} \text{xor BigEndian}^4) \\
\text{memquad} \leftarrow \text{LoadMemory}(\text{uncached, BYTE, pAddr, vAddr, DATA}) \\
\text{byte} \leftarrow \text{vAddr}_{3..0} \text{xor BigEndian}^4 \\
\text{GPR[rt]}_{63..0} \leftarrow \text{sign}_\text{extend}(\text{memquad}_{(7+8\text{-byte})..8\text{-byte}})
\]

Exceptions:

- TLB Refill
- TLB Invalid
- Address Error

Programming Notes:

None
Appendix A  CPU Instruction Set Details

LBU
Load Byte Unsigned

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBU</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100100</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format:
LBU  rt, offset (base)

Purpose:
To load a byte from memory as an unsigned value.

Description:
rt ← memory [base + offset]

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, zero-extended, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

Restrictions:
None

Operation:  (128-bit bus)

vAddr ← sign_extend (offset) + GPR[base] 31..0
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr[P.SIZE-1..4 || (pAddr 3..0 xor BigEndian4)]
memquad ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
byte ← vAddf 3..0 xor BigEndian4
GPR[rt]63..0 ← zero_extend (memquad(7+8-byte) 8-byte)

Exceptions:
TLB Refill
TLB Invalid
Address Error

Programming Notes:
None
**LD**  Load Doubleword

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>110111</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS III**

**Format:**  LD  rt, offset (base)

**Purpose:**  To load a doubleword from memory.

**Description:**  rt ← memory [base + offset]

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

**Restrictions:**

The effective address must be naturally aligned. If any of the three least-significant bits of the effective address are non-zero, an Address Error exception occurs.

**Operation: (128-bit bus)**

\[ v\text{Addr} \leftarrow \text{sign}_{-}\text{extend}(\text{offset}) + \text{GPR}[\text{base}]_{31..0} \]

\[ \text{if} \ (v\text{Addr}_{2..0}) \neq 0^3 \ \text{then SignalException (AddressError) endif} \]

\[ p\text{Addr} \leftarrow \text{AddressTranslation}(v\text{Addr}, \text{DATA, LOAD}) \]

\[ p\text{Addr} \leftarrow p\text{Addr}^{\text{PSIZE}-1..4} \ || \ (p\text{Addr}_{3.0} \ \text{xor} \ (\text{BigEndian} \ || \ 0^3)) \]

\[ \text{byte} \leftarrow v\text{Addr}^{3.0} || (\text{BigEndian} || 0^3) \]

\[ \text{memquad} \leftarrow \text{LoadMemory}(\text{uncached}, \text{DOUBLEWORD}, p\text{Addr}, v\text{Addr}, \text{DATA}) \]

\[ \text{GPR}[rt]_{63..0} \leftarrow \text{memquad}[63\text{+}\text{byte}..8\text{+}\text{byte}] \]

**Exceptions:**

- TLB Refill
- TLB Invalid
- Address Error

**Programming Notes:**

None
Appendix A  CPU Instruction Set Details

LDL

**Load Doubleword Left**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDL</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011010</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS III**

**Format:**
LDL  rt, offset (base)

**Purpose:**
To load the more-significant part of a doubleword from an unaligned memory address.

**Description:**
rt ← rt MERGE memory [base + offset]

Paired LDL and LDR instructions are used to load a register with a doubleword from eight consecutive bytes in memory starting at an arbitrary byte address. LDL loads the left (most-significant) bytes and LDR loads the right (least-significant) bytes.

The instruction adds the 16-bit signed offset to the contents of GPR base to form the effective address. This is the address of the most-significant byte of a doubleword composed of eight consecutive bytes in memory. LDL loads from one to eight bytes, the most-significant bytes of the doubleword, into the corresponding bytes of GPR rt. It loads the bytes that are in the target doubleword that are also in the aligned doubleword which contains the byte specified by the effective address.

Conceptually, it starts at the specified byte in memory and loads that byte into the high-order (left-most) byte of the register; then it loads bytes from memory into the register until it reaches the low-order byte of the doubleword in memory. The least-significant (right-most) byte(s) of the register will not be changed.

The contents of GPR rt are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register rt and a following LDL (or LDR) instruction which also specifies register rt.
No address exceptions due to alignment are possible.

Restrictions:
None

Operation: (128-bit bus)

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign_extend (offset)} + \text{GPR[base]}_{31..0} \\
\text{pAddr}, \text{uncached} & \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \\
\text{pAddr} & \leftarrow \text{pAddr}_{(PSIZE-1)..4} \parallel (\text{pAddr}_{3..0} \oplus \text{BigEndian}^4) \\
\text{if (BigEndian = 0) then} & \\
\quad \text{pAddr} & \leftarrow \text{pAddr}_{(PSIZE-1)..3} \parallel 0^3 \\
\quad \text{endif} \\
\text{byte} & \leftarrow 0 \parallel (\text{vAddr}_{2..0} \oplus \text{BigEndian}^3) \\
\text{doubleword} & \leftarrow \text{vAddr}_{3} \oplus \text{BigEndian} \\
\text{memquad} & \leftarrow \text{LoadMemory (uncached, byte, pAddr, vAddr, DATA)} \\
\text{GPR[rt]}_{63..0} & \leftarrow \text{memquad}_{(7+8\cdot\text{byte}+64\cdot\text{doubleword})..(64\cdot\text{doubleword})} \parallel \text{GPR[rt]}_{(55-8\cdot\text{byte})..0}
\end{align*}
\]

Given a doubleword in a register and a doubleword in memory, the operation of LDL is as follows:
### LDL

<table>
<thead>
<tr>
<th>vAddr&lt;sub&gt;3..0&lt;/sub&gt;</th>
<th>Little-endian byte ordering (BigEndianCPU = 0)</th>
<th>Destination register contents after instruction (shaded is unchanged)</th>
<th>Type offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(63----------------------------------------32 31------------------------------------------0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>X b c d e f g h</td>
<td>0 0 15</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>W X c d e f g h</td>
<td>1 0 14</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>V W X d e f g h</td>
<td>2 0 13</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>U V W X e f g h</td>
<td>3 0 12</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>T U V W X f g h</td>
<td>4 0 11</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S T U V W X g h</td>
<td>5 0 10</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>R S T U V W X h</td>
<td>6 0 9</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Q R S T U V W X h</td>
<td>7 0 8</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>P b c d e f g h</td>
<td>0 8 7</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>O P c d e f g h</td>
<td>1 8 6</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>N O P d e f g h</td>
<td>2 8 5</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>M N O P e f g h</td>
<td>3 8 4</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>L M N O P f g h</td>
<td>4 8 3</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>K L M N O P g h</td>
<td>5 8 2</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>J K L M N O P h</td>
<td>6 8 1</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>I J K L M N O P h</td>
<td>7 8 0</td>
<td></td>
</tr>
</tbody>
</table>
### LDL

<table>
<thead>
<tr>
<th>MSB 63</th>
<th>0</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Big-endian</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Memory</td>
<td>I</td>
<td>J</td>
</tr>
<tr>
<td>Little-endian</td>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>

#### Big-endian byte ordering (BigEndianCPU = 0)

<table>
<thead>
<tr>
<th>vAddr_{3,0}</th>
<th>63</th>
<th>32</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination register contents after instruction (shaded is unchanged)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>offset</td>
<td>LEM</td>
<td>BEM</td>
</tr>
<tr>
<td>0</td>
<td>I</td>
<td>J</td>
<td>K</td>
</tr>
<tr>
<td>1</td>
<td>J</td>
<td>K</td>
<td>L</td>
</tr>
<tr>
<td>2</td>
<td>K</td>
<td>L</td>
<td>M</td>
</tr>
<tr>
<td>3</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>5</td>
<td>N</td>
<td>O</td>
<td>P</td>
</tr>
<tr>
<td>6</td>
<td>O</td>
<td>P</td>
<td>c</td>
</tr>
<tr>
<td>7</td>
<td>P</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>8</td>
<td>Q</td>
<td>R</td>
<td>S</td>
</tr>
<tr>
<td>9</td>
<td>R</td>
<td>S</td>
<td>T</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>11</td>
<td>T</td>
<td>U</td>
<td>V</td>
</tr>
<tr>
<td>12</td>
<td>U</td>
<td>V</td>
<td>W</td>
</tr>
<tr>
<td>13</td>
<td>V</td>
<td>W</td>
<td>X</td>
</tr>
<tr>
<td>14</td>
<td>W</td>
<td>X</td>
<td>c</td>
</tr>
<tr>
<td>15</td>
<td>X</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

- **LEM** Little-endian memory (BigEndian = 0)
- **BEM** BigEndian = 1
- **Type** AccessLength sent to memory
- **Offset** pAddr_{3,0} sent to memory

#### Exceptions:
- TLB Refill
- TLB Invalid
- Address Error

#### Programming Notes:
- None
Appendix A  CPU Instruction Set Details

### LDR

**Load Doubleword Right**

<table>
<thead>
<tr>
<th>LDR</th>
<th>base</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>011011</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Format:** LDR rt, offset (base)

**Purpose:** To load the less-significant part of a doubleword from an unaligned memory address.

**Description:**

\[ rt \leftarrow rt \text{ MERGE memory } [\text{base + offset}] \]

Paired LDL and LDR instructions are used to load a register with a doubleword from eight consecutive bytes in memory starting at an arbitrary byte address. LDL loads the left (most-significant) bytes and LDR loads the right (least-significant) bytes.

The instruction adds the 16-bit signed offset to the contents of GPR base to form the effective address. This is the address of the least-significant bytes of a doubleword composed of eight consecutive bytes in memory. LDR loads from one to eight bytes, the least-significant bytes of the doubleword, into the corresponding bytes of GPR rt. It loads the bytes that are in the target doubleword that are also in the aligned doubleword which contains the byte specified by the effective address.

Conceptually, it starts at the specified byte in memory and loads that byte into the low-order (right-most) byte of the register; then it loads bytes from memory into the register until it reaches the high-order byte of the doubleword in memory. The most significant (left-most) byte(s) of the register will not be changed.

**MIPS III**

<table>
<thead>
<tr>
<th>LDR</th>
<th>Load Doubleword Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26 25 21 20 16 15 0</td>
</tr>
</tbody>
</table>

The contents of GPR rt are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register rt and a following LDR (or LDL) instruction which also specifies register rt.
No address exceptions due to alignment are possible.

Restrictions:
None

Operation: (128-bit bus)

\[ vAddr \leftarrow \text{sign}\_\text{extend}(\text{offset}) + \text{GPR[base]}_{31:0} \]

\[ (\text{pAddr}, \text{uncached}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{LOAD}) \]

\[ \text{pAddr} \leftarrow \text{pAddr}_{(\text{PSIZE}-1):0} \text{||} (\text{pAddr}_{3:0} \text{xor BigEndian})^4 \]

if (BigEndian = 1) then

\[ \text{pAddr} \leftarrow \text{pAddr}_{(\text{PSIZE}-1):3} \text{||} 0^3 \]
endif

\[ \text{byte} \leftarrow 0 \text{||} (\text{vAddr}_{2:0} \text{xor BigEndian})^3 \]

\[ \text{doubleword} \leftarrow \text{vAddr}_{3} \text{xor BigEndian} \]

\[ \text{memquad} \leftarrow \text{LoadMemory}(\text{uncached, byte, pAddr, vAddr, DATA}) \]

\[ \text{GPR[rt]}_{63:0} \leftarrow \text{GPR[rt]}_{63:(64-8\times \text{byte})} \text{|| memquad}_{(64-8\times \text{byte})} \text{|| memquad}_{(64-8\times \text{doubleword})} \]

Given a doubleword in a register and a doubleword in memory, the operation of LDR is as follows:
### LDR

**Register**

<table>
<thead>
<tr>
<th>MSB</th>
<th>63</th>
<th>0</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
</tbody>
</table>

**Little-endian Memory**

```
I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X
```

**Little-endian byte ordering (BigEndianCPU = 0)**

<table>
<thead>
<tr>
<th>vAddr_{3,0}</th>
<th>Destination register contents after instruction (shaded is unchanged)</th>
<th>Type</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(63----------------------------------------32 31------------------------------------------0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Q R S T U V W X</td>
<td>7</td>
<td>0 0</td>
</tr>
<tr>
<td>1</td>
<td>a Q R S T U V W</td>
<td>6</td>
<td>1 0</td>
</tr>
<tr>
<td>2</td>
<td>a b Q R S T U V</td>
<td>5</td>
<td>2 0</td>
</tr>
<tr>
<td>3</td>
<td>a b c Q R S T U</td>
<td>4</td>
<td>3 0</td>
</tr>
<tr>
<td>4</td>
<td>a b c d Q R S T</td>
<td>3</td>
<td>4 0</td>
</tr>
<tr>
<td>5</td>
<td>a b c d e Q R S</td>
<td>2</td>
<td>5 0</td>
</tr>
<tr>
<td>6</td>
<td>a b c d e f Q R</td>
<td>1</td>
<td>6 0</td>
</tr>
<tr>
<td>7</td>
<td>a b c d e f g Q</td>
<td>0</td>
<td>7 0</td>
</tr>
<tr>
<td>8</td>
<td>I J K L M N O P</td>
<td>7</td>
<td>8 0</td>
</tr>
<tr>
<td>9</td>
<td>a I J K L M N</td>
<td>6</td>
<td>9 0</td>
</tr>
<tr>
<td>10</td>
<td>a b I J K L M</td>
<td>5</td>
<td>10 0</td>
</tr>
<tr>
<td>11</td>
<td>a b c I J K L</td>
<td>4</td>
<td>11 0</td>
</tr>
<tr>
<td>12</td>
<td>a b c d I J K L</td>
<td>3</td>
<td>12 0</td>
</tr>
<tr>
<td>13</td>
<td>a b c d e I J K</td>
<td>2</td>
<td>13 0</td>
</tr>
<tr>
<td>14</td>
<td>a b c d e f I J</td>
<td>1</td>
<td>14 0</td>
</tr>
<tr>
<td>15</td>
<td>a b c d e f g I</td>
<td>0</td>
<td>15 0</td>
</tr>
</tbody>
</table>
### LDR

<table>
<thead>
<tr>
<th>vAddr3..0</th>
<th>Big-endian byte ordering (BigEndianCPU = 1)</th>
<th>Destination register contents after instruction (shaded is unchanged)</th>
<th>Type</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(63----------------------------------------32 31------------------------------------------0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>a b c d e f g I</td>
<td>0</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>a b c d e f I J</td>
<td>1</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>a b c d e I J K</td>
<td>2</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>a b c d I J K L</td>
<td>3</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>a b c I J K L M</td>
<td>4</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>a b I J K L M N</td>
<td>5</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>a I J K L M N O</td>
<td>6</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>I J K L M N O P</td>
<td>7</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>a b c d e f g Q</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>a b c d e f Q R</td>
<td>1</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>a b c d e Q R S</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>a b c d Q R S T</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>a b c Q R S T U</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>a b Q R S T U V</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>a Q R S T U V W</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>Q R S T U V W X</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**LEM** Little-endian memory (BigEndianMem = 0)

**BEM** BigEndianMem = 1

**Type** AccessLength sent to memory

**Offset** pAddr3..0 sent to memory

### Exceptions:
- TLB Refill
- TLB Invalid
- Address Error

### Programming Notes:
- None
### Load Halfword

**Format:**

LH  rt, offset (base)

**Purpose:**

To load a halfword from memory as a signed value.

**Description:**

\[ rt \leftarrow \text{memory \[base + offset\]} \]

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, sign-extended, and placed in GPR \( rt \). The 16-bit signed \( offset \) is added to the contents of GPR \( base \) to form the effective address.

**Restrictions:**

The effective address must be naturally aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

**Operation:** (128-bit bus)

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend (offset)} + \text{GPR[base]}_{31..0} \\
\text{if (vAddr}_0 \neq 0 \text{ then SignalException (AddressError) endif} \\
\text{pAddr} & \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \\
\text{memquad} & \leftarrow \text{LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)} \\
\text{byte} & \leftarrow \text{vAddr}_{3..0} \text{ xor (BigEndian^3 \| 0)} \\
\text{GPR[rt]}_{63..0} & \leftarrow \text{sign\_extend (memquad(15+8-byte)..8-byte)}
\end{align*}
\]

**Exceptions:**

TLB Refill  
TLB Invalid  
Address Error

**Programming Notes:**

None
### LHU

**Load Halfword Unsigned**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LHU</td>
<td>offset</td>
<td>100101</td>
<td>base</td>
<td>rt</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:**
LHU rt, offset (base)

**Purpose:**
To load a halfword from memory as an unsigned value.

**Description:**
rt ← memory [base + offset]

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, zero-extended, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

**Restrictions:**
The effective address must be naturally aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

**Operation:** (128-bit bus)

\[
vAddr \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}]_{31..0}
\]

if (vAddr\_0 ≠ 0) then SignalException (AddressError) endif

\[
\begin{align*}
pAddr & \leftarrow \text{AddressTranslation}(vAddr, \text{DATA, LOAD}) \\
pAddr & \leftarrow pAddr(\text{PSIZE}-1..4 || (pAddr_{3..0} \xor \text{BigEndian}_{3\ldots0})) \\
\text{memquad} & \leftarrow \text{LoadMemory}(\text{uncached, HALFWORD, pAddr, vAddr, DATA}) \\
\text{byte} & \leftarrow vAddr_{3..0} \xor \text{BigEndian}_{3\ldots0} \\
\text{GPR}[rt]_{63..0} & \leftarrow \text{zero\_extend}(\text{memquad}(15+8\cdot\text{byte},1..8\cdot\text{byte})
\end{align*}
\]

**Exceptions:**

- TLB Refill
- TLB Invalid
- Address Error

**Programming Notes:**

None
LUI  Load Upper Immediate

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td>0</td>
<td>00000</td>
<td>rt</td>
<td></td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format:  
LUI  rt, immediate

Purpose:  
To load a constant into the upper half of a word.

Description:  
rt ← immediate || 0^16

The 16-bit immediate is shifted left 16 bits and concatenated with 16 bits of low-order zeros. The 32-bit result is sign-extended and placed into GPR rt.

Restrictions:  
None

Operation:  
GPR [rt]_{63..0} ← sign_extend (immediate || 0^16)

Exceptions:  
None

Programming Notes:  
None
## LW

**Load Word**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100011</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### MIPS I

**Format:**

LW rt, offset (base)

**Purpose:** To load a word from memory as a signed value.

**Description:**

\[
rt \leftarrow \text{memory} [\text{base} + \text{offset}]
\]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

**Restrictions:**

The effective address must be naturally aligned. If either of the two least-significant bits of the address are non-zero, an Address Error exception occurs.

**Operation:** (128-bit bus)

\[
\begin{align*}
vAddr & \leftarrow \text{sign}_\text{extend} (\text{offset}) + \text{GPR} [\text{base}]_{31..0} \\
\text{if} (vAddr_{1..0}) \neq 0^2 \text{ then SignalException (AddressError) endif} \\
pAddr & \leftarrow \text{AddressTranslation} (vAddr, \text{DATA, LOAD}) \\
pAddr & \leftarrow \text{pAddr}^{[\text{PSIZE}-1..4]} \oplus (\text{pAddr}_{3..0} \oplus (\text{BigEndian}^2 \oplus 0^2)) \\
\text{memquad} & \leftarrow \text{LoadMemory} (\text{uncached, WORD, pAddr, vAddr, DATA}) \\
\text{byte} & \leftarrow vAddr_{3..0} \oplus (\text{BigEndian}^2 \oplus 0^2) \\
\text{GPR} [rt]_{63..0} & \leftarrow \text{sign}_\text{extend} (\text{memquad}_{31+8*\text{byte}}..8*\text{byte})
\end{align*}
\]

**Exceptions:**

- TLB Refill
- TLB Invalid
- Address Error

**Programming Notes:**

None
### LWL

**Load Word Left**

<table>
<thead>
<tr>
<th>LWL</th>
<th>base</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>100010</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:** LWL rt, offset (base)

**Purpose:** To load the more-significant part of a word from an unaligned memory address as a signed value.

**Description:**

\[
rt \leftarrow rt \text{ MERGE memory } [\text{base + offset}]
\]

Paired LWL and LWR instructions are used to load a register with a word from four consecutive bytes in memory starting at an arbitrary byte address. LWL loads the left (most-significant) bytes and LWR loads the right (least-significant) bytes.

The instruction adds the 16-bit signed offset to the contents of GPR base to form the effective address. This is the address of the most-significant byte of a word composed of four consecutive bytes in memory. LWL loads from one to four bytes, the most-significant bytes of the word, into the corresponding bytes of GPR rt. It loads the bytes that are in the target word that are also in the aligned word which contains the byte specified by the effective address.

Bit 31 of the register is loaded so the loaded word is sign-extended.

Conceptually, it starts at the specified byte in memory and loads that byte into the high-order (left-most) byte of the register; then it loads bytes from memory into the register until it reaches the low-order byte of the word in memory. The least-significant (right-most) byte(s) of the register will not be changed.
The contents of GPR \( rt \) are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register \( rt \) and a following LWL (or LWR) instruction which also specifies register \( rt \).

No address exceptions due to alignment are possible.

**Restrictions:**

None

**Operation:** (128-bit bus)

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign} \_ \text{extend (offset)} + \text{GPR} [\text{base}]_{31..0} \\
(\text{pAddr, uncached}) & \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \\
\text{pAddr} & \leftarrow \text{pAddr}_{(\text{PSIZE}-1..4} || (\text{pAddr}_{3..0} \text{xor BigEndian})^4 \\
\text{if (BigEndian = 0) then} & \\
\text{pAddr}_{(\text{PSIZE}-1..3} & \mid 0^3 \\
\text{endif} \\
\text{byte} & \leftarrow 0^2 || (\text{vAddr}_{1..0} \text{xor BigEndian})^2 \\
\text{word} & \leftarrow \text{vAddr}_{3..2} \text{xor BigEndian}^2 \\
\text{memquad} & \leftarrow \text{LoadMemory (uncached, byte, pAddr, vAddr, DATA)} \\
\text{temp} & \leftarrow \text{memquad}_{(32 \cdot \text{word} + 8 \cdot \text{byte} + 7..32 \cdot \text{word} || \text{GPR}[rt]_{(23..8 \cdot \text{byte}..0}} \\
\text{GPR} [rt]_{63..0} & \leftarrow (\text{temp}_{31})^{32} || \text{temp}
\end{align*}
\]

Given a doubleword in a register and a doubleword in memory, the operation of LWL is as follows:
### LWL

**Register**

<table>
<thead>
<tr>
<th>MSB 63</th>
<th>0 LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

**Little-endian Memory**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
</tr>
</tbody>
</table>

---

**Little-endian byte ordering (BigEndianCPU = 0)**

<table>
<thead>
<tr>
<th>vAddr3,0</th>
<th>Destination register contents after instruction (shaded is unchanged)</th>
<th>Type</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sign bit(31) extended X f g h 0 0 15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Sign bit(31) extended W X g h 1 0 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Sign bit(31) extended V W X h 2 0 13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Sign bit(31) extended U V W X 3 0 12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Sign bit(31) extended T f g h 0 4 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Sign bit(31) extended S T g h 1 4 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Sign bit(31) extended R S T h 2 4 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Sign bit(31) extended Q R S T 3 4 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Sign bit(31) extended P f g h 0 8 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Sign bit(31) extended O P g h 1 8 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Sign bit(31) extended N O P h 2 8 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Sign bit(31) extended M N O P 3 8 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Sign bit(31) extended L f g h 0 12 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Sign bit(31) extended K L g h 1 12 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Sign bit(31) extended J K L h 2 12 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Sign bit(31) extended I J K L 3 12 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**LWL**

Big-endian Little-endian

<table>
<thead>
<tr>
<th>vAddr_{3,0}</th>
<th>Sign bit(31) extended</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>Type</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sign bit(31) extended</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>3</td>
<td>12 0</td>
</tr>
<tr>
<td>1</td>
<td>Sign bit(31) extended</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>h</td>
<td>2</td>
<td>12 1</td>
</tr>
<tr>
<td>2</td>
<td>Sign bit(31) extended</td>
<td>K</td>
<td>L</td>
<td>g</td>
<td>h</td>
<td>1</td>
<td>12 2</td>
</tr>
<tr>
<td>3</td>
<td>Sign bit(31) extended</td>
<td>L</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>0</td>
<td>12 3</td>
</tr>
<tr>
<td>4</td>
<td>Sign bit(31) extended</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>3</td>
<td>8 4</td>
</tr>
<tr>
<td>5</td>
<td>Sign bit(31) extended</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>h</td>
<td>2</td>
<td>8 5</td>
</tr>
<tr>
<td>6</td>
<td>Sign bit(31) extended</td>
<td>O</td>
<td>P</td>
<td>g</td>
<td>h</td>
<td>1</td>
<td>8 6</td>
</tr>
<tr>
<td>7</td>
<td>Sign bit(31) extended</td>
<td>P</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>0</td>
<td>8 7</td>
</tr>
<tr>
<td>8</td>
<td>Sign bit(31) extended</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>3</td>
<td>4 8</td>
</tr>
<tr>
<td>9</td>
<td>Sign bit(31) extended</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>h</td>
<td>2</td>
<td>4 9</td>
</tr>
<tr>
<td>10</td>
<td>Sign bit(31) extended</td>
<td>S</td>
<td>T</td>
<td>g</td>
<td>h</td>
<td>1</td>
<td>4 10</td>
</tr>
<tr>
<td>11</td>
<td>Sign bit(31) extended</td>
<td>T</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>0</td>
<td>4 11</td>
</tr>
<tr>
<td>12</td>
<td>Sign bit(31) extended</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>3</td>
<td>0 12</td>
</tr>
<tr>
<td>13</td>
<td>Sign bit(31) extended</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>h</td>
<td>2</td>
<td>0 13</td>
</tr>
<tr>
<td>14</td>
<td>Sign bit(31) extended</td>
<td>W</td>
<td>X</td>
<td>g</td>
<td>h</td>
<td>1</td>
<td>0 14</td>
</tr>
<tr>
<td>15</td>
<td>Sign bit(31) extended</td>
<td>X</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>0</td>
<td>0 15</td>
</tr>
</tbody>
</table>

**LEM**  Little-endian memory (BigEndianMem = 0)

**BEM**  BigEndianMem = 1

**Type**  AccessLength sent to memory

**Offset**  pAddr_{2,0} sent to memory

**Exceptions:**

TLB Refill
TLB Invalid
Address Error

**Programming Notes:**

The architecture provides no direct support for treating unaligned words as unsigned values, i.e. zeroing bits 63-32 of the destination register when bit 31 is loaded. See SLL or SLLV for a single-instruction method of propagating the word sign bit in a register into the upper half of a 64-bit register.
LWR
Load Word Right

<table>
<thead>
<tr>
<th>Base</th>
<th>Rt</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>100110</td>
<td>6</td>
<td>16</td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:**
LWR rt, offset (base)

**Purpose:**
To load the less-significant part of a word from an unaligned memory address as a signed value.

**Description:**
rt ← rt MERGE memory [base + offset]

Paired LWL and LWR instructions are used to load a register with a word from four consecutive bytes in memory starting at an arbitrary byte address. LWL loads the left (most-significant) bytes and LWR loads the right (least-significant) bytes.

The instruction adds the 16-bit signed offset to the contents of GPR base to form the effective address. This is the address of the least-significant byte of a word composed of four consecutive bytes in memory. LWR loads from one to four bytes, the least-significant bytes of the word, into the corresponding bytes of GPR rt. It loads the bytes that are in the target word that are also in the aligned word which contains the byte specified by the effective address.

If the word sign bit (bit 31) is loaded from memory into the register by the instruction, then the loaded word is sign-extended. If the sign bit is not loaded from memory by the LWR, then bits 63..32 of the destination are unchanged.

Conceptually, it starts at the specified byte in memory and loads that byte into the low-order (right-most) byte of the register; then it loads bytes from memory into the register until it reaches the high-order byte of the word in memory. The most significant (left-most) byte(s) of the register will not be changed.
The contents of GPR \( rt \) are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register \( rt \) and a following LWR (or LWL) instruction which also specifies register \( rt \).

No address exceptions due to alignment are possible.

Restrictions:

None

Operation: \((128\text{-bit bus})\)

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign} \_ \text{extend} \text{ (offset)} + \text{GPR} \text{ [base]}31..0 \\
(p\text{Addr, uncached}) & \leftarrow \text{AddressTranslation} \text{ (vAddr, DATA, LOAD)} \\
p\text{Addr} & \leftarrow p\text{Addr}(\text{PSIZE}-1)\ldots4 \ || (p\text{Addr}.0 \ xor \ \text{BigEndian}^4) \\
& \text{if (BigEndian = 1) then} \\
& \ p\text{Addr}(\text{PSIZE}-31)\ldots3 \ || 0^3 \\
& \text{endif} \\
\text{byte} & \leftarrow 0 \ || (\text{vAddr}.1..0 \ xor \ \text{BigEndian}^2) \\
\text{word} & \leftarrow \text{vAddr}.3..2 \ xor \ \text{BigEndian}^2 \\
\text{memquad} & \leftarrow \text{LoadMemory} \text{ (uncached, byte, pAddr, vAddr, DATA)} \\
\text{temp} & \leftarrow \text{GPR} \text{ [rt]}31..(32\text{-byte} \ || \ \text{memquad}(31+32\text{-word}..(32\text{-word}+8\text{-byte}) \\
& \text{if (byte = 4) then} \\
& \ utemp \leftarrow (\text{temp}31)^{32} \quad \text{/* loaded bit 31, must sign extend */} \\
& \text{else} \\
& \quad \text{one of the following two behaviors:} \\
& \quad \quad \text{utemp} \leftarrow \text{GPR} \text{ [rt]}63..32 \quad \text{/* leave what was there alone */} \\
& \quad \quad \text{utemp} \leftarrow (\text{GPR} \text{ [rt]}31)^{32} \quad \text{/* sign-extend bit 31 */} \\
& \text{endif} \\
\text{GPR} \text{ [rt]}63..0 & \leftarrow \text{utemp} \ || \ \text{temp}
\end{align*}
\]

Given a word in a register and a word in memory, the operation of LWR is as follows:
## LWR

### Register

<table>
<thead>
<tr>
<th>MSB 63</th>
<th>0 LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c d e f g h</td>
<td></td>
</tr>
</tbody>
</table>

### Little-endian Memory

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
</tr>
</tbody>
</table>

### Little-endian byte ordering (BigEndianCPU = 0)

<table>
<thead>
<tr>
<th>vAddr&lt;sub&gt;3:0&lt;/sub&gt;</th>
<th>Destination register contents after instruction (shaded is unchanged)</th>
<th>Type</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sign bit (31) extended</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>Sign bit (31) extended or unchanged</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>Sign bit (31) extended or unchanged</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>Sign bit (31) extended or unchanged</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>Sign bit (31) extended</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>Sign bit (31) extended or unchanged</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>Sign bit (31) extended or unchanged</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>Sign bit (31) extended or unchanged</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>Sign bit (31) extended</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>Sign bit (31) extended or unchanged</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>Sign bit (31) extended or unchanged</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>Sign bit (31) extended or unchanged</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>Sign bit (31) extended</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>Sign bit (31) extended or unchanged</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>Sign bit (31) extended or unchanged</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>Sign bit (31) extended or unchanged</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>
### LWR

<table>
<thead>
<tr>
<th>Register</th>
<th>MSB 63</th>
<th>0 LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Big-endian Memory</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th>N</th>
<th>O</th>
<th>P</th>
<th>Q</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>U</th>
<th>V</th>
<th>W</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Little-endian Memory</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Big-endian byte ordering (BigEndianCPU = 1)

<table>
<thead>
<tr>
<th>vAddr_{2,0}</th>
<th>Destination register contents after instruction (shaded is unchanged)</th>
<th>Type</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>(63-----------------------------32 31------------------------------------------0)</td>
<td>LEM</td>
<td>BEM</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Sign bit (31) extended or unchanged</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>1</td>
<td>Sign bit (31) extended or unchanged</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>2</td>
<td>Sign bit (31) extended or unchanged</td>
<td>e</td>
<td>I</td>
</tr>
<tr>
<td>3</td>
<td>Sign bit (31) extended</td>
<td>I</td>
<td>J</td>
</tr>
<tr>
<td>4</td>
<td>Sign bit (31) extended or unchanged</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>5</td>
<td>Sign bit (31) extended or unchanged</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td>Sign bit (31) extended or unchanged</td>
<td>e</td>
<td>M</td>
</tr>
<tr>
<td>7</td>
<td>Sign bit (31) extended</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>8</td>
<td>Sign bit (31) extended or unchanged</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>9</td>
<td>Sign bit (31) extended or unchanged</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>10</td>
<td>Sign bit (31) extended or unchanged</td>
<td>e</td>
<td>Q</td>
</tr>
<tr>
<td>11</td>
<td>Sign bit (31) extended</td>
<td>Q</td>
<td>R</td>
</tr>
<tr>
<td>12</td>
<td>Sign bit (31) extended or unchanged</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>13</td>
<td>Sign bit (31) extended or unchanged</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>14</td>
<td>Sign bit (31) extended or unchanged</td>
<td>e</td>
<td>U</td>
</tr>
<tr>
<td>15</td>
<td>Sign bit (31) extended</td>
<td>U</td>
<td>V</td>
</tr>
</tbody>
</table>

- **LEM**: Little-endian memory (BigEndian = 0)
- **BEM**: BigEndianMem = 1
- **Type**: AccessLength sent to memory
- **Offset**: pAddr_{2,0} sent to memory

**Exceptions:**
- TLB Refill
- TLB Invalid
- Address Error

**Programming Notes:**

The architecture provides no direct support for treating unaligned words as unsigned values, i.e. zeroing bits 63..32 of the destination register when bit 31 is loaded. See SLL or SLLV for a single-instruction method of propagating the word sign bit in a register into the upper half of a 64-bit register.
### LWU

**Load Word Unsigned**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWU</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100111</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### MIPS III

**Format:**
LWU  rt, offset (base)

**Purpose:**
To load a word from memory as an unsigned value.

**Description:**

\[
rt \leftarrow \text{memory}[\text{base} + \text{offset}]
\]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, zero-extended, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

**Restrictions:**

The effective address must be naturally aligned. If either of the two least-significant bits of the address are non-zero, an Address Error Exception occurs.

**Operation:** (128-bit bus)

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign}\_\text{extend}(\text{offset}) + \text{GPR}[\text{base}]_{31..0} \\
\text{if} \ (\text{vAddr}_{1..0}) \neq 0^2 \ & \text{then} \ \text{SignalException (AddressError)} \ \text{endif} \\
\text{pAddr} & \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \\
\text{pAddr} & \leftarrow \text{pAddr}_{(\text{PSIZE} - 1..4)} \ || \ (\text{pAddr}_{3..0} \ \text{xor} \ (\text{BigEndian}^2 \ || \ 0^2)) \\
\text{memquad} & \leftarrow \text{LoadMemory (uncached, WORD, pAddr, vAddr, DATA)} \\
\text{byte} & \leftarrow \text{vAddr}_{3..0} \ \text{xor} \ (\text{BigEndian}^2 \ || \ 0^2) \\
\text{GPR}[rt]_{63..0} & \leftarrow 0^{32} \ || \ \text{memquad}_{(31\text{-byte}\_\text{byte})..8\text{-byte}}
\end{align*}
\]

**Exceptions:**

- TLB Refill
- TLB Invalid
- Address Error

**Programming Notes:**

None
### MFHI

**Move from HI Register**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>0</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>MFHI</td>
<td>010000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### MIPS I

- **Format:** MFHI rd
- **Purpose:** To copy the special purpose HI register to a GPR.
- **Description:** rd ← HI
  
  The contents of special register HI are loaded into GPR rd.
- **Restrictions:** None
- **Operation:**
  
  GPR [rd]63..0 ← HI63..0
- **Exceptions:** None
- **Programming Notes:**
  
  No restriction is needed because C790 has an interlock mechanism for MULT or DIV instructions.
**MFLO**

**Move from LO Register**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>0</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>00000</td>
<td>010010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:** MFLO rd

**Purpose:** To copy the special purpose LO register to a GPR.

**Description:** rd ← LO

The contents of special register LO are loaded into GPR rd.

**Restrictions:** None

**Operation:**

GPR [rd]_{63..0} ← LO_{63..0}

**Exceptions:** None

**Programming Notes:**

(Same as MFHI)
Appendix A  CPU Instruction Set Details

**MOVN**  
Move Conditional on Not Zero

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>MOVN</td>
<td>001011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 6 | 5 | 5 | 5 | 5 | 6 |

**MIPS IV**

**Format:**  
MOVN rd, rs, rt

**Purpose:**  
To conditionally move a GPR after testing a GPR value.

**Description:**  
if (rt ≠ 0) then rd ← rs

If the value in GPR rt is not equal to zero, then the contents of GPR rs are placed into GPR rd.

**Restrictions:**  
None

**Operation:**

if GPR [rt]₆₃..₀ ≠ 0 then
  GPR [rd]₆₃..₀ ← GPR [rs]₆₃..₀
endif

**Exceptions:**  
None

**Programming Notes:**

The nonzero value tested here is the “condition true” result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

---

A-82
MOVZ Move Conditional on Zero

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>MOVZ</td>
<td>000000</td>
<td>001010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS IV**

**Format:** MOVZ rd, rs, rt

**Purpose:** To conditionally move a GPR after testing a GPR value.

**Description:** if (rt = 0) then rd ← rs

If the value in GPR rt is equal to zero, then the contents of GPR rs are placed into GPR rd.

**Restrictions:**
None

**Operation:**

if GPR [rt]_{63..0} = 0 then
  GPR [rd]_{63..0} ← GPR [rs]_{63..0}
endif

**Exceptions:**
None

**Programming Notes:**

The zero value tested here is the “condition false” result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.
**MTHI**  Move to HI Register

<table>
<thead>
<tr>
<th>SPECIAL</th>
<th>rs</th>
<th>0</th>
<th>MTHI</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>010001</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:** MTHI rs

**Purpose:** To copy a GPR to the special purpose HI register.

**Description:**

HI ← rs

The contents of GPR rs are loaded into special register HI.

**Restrictions:**

None

**Operation:**

HI_{63..0} ← GPR [rs]_{63..0}

**Exceptions:**

None

**Programming Notes:**

None
Appendix A  CPU Instruction Set Details

MTLO: Move to LO Register

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>rs</td>
<td>0</td>
<td>000 0000 0000 0000</td>
<td>MTLO</td>
<td>010011</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format: MTLO rs

Purpose: To copy a GPR to the special purpose LO register.

Description: LO ← rs

The contents of GPR rs are loaded into special register LO.

Restrictions: None

Operation: LO63.0 ← GPR [rs]63.0

Exceptions: None

Programming Notes: None
Appendix A  CPU Instruction Set Details

MULT

<table>
<thead>
<tr>
<th>MULT</th>
<th>Multiply Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 26 25 21 20 16 15</td>
<td>0 6 5 0</td>
</tr>
<tr>
<td>SPECIAL</td>
<td>rs</td>
</tr>
<tr>
<td>000000</td>
<td>00 0000 0000</td>
</tr>
</tbody>
</table>

MIPS I

Format: MULT rs, rt

Purpose: To multiply 32-bit signed integers.

Description: (LO, HI) ← rs × rt

The 32-bit word value in GPR rt is multiplied by the 32-bit value in GPR rs, treating both operands as signed values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register LO, and the high-order 32-bit word is placed into special register HI.

No arithmetic exception occurs under any circumstances.

Restrictions:

If either GPR rt or GPR rs do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:

\[
\text{if (NotWordValue (GPR [rs]) or NotWordValue (GPR [rt])) then UndefinedResult() endif}
\]

\[
\text{prod} \leftarrow \text{GPR [rs]31..0} \times \text{GPR [rt]31..0}
\]

\[
\text{LO63..0} \leftarrow (\text{prod 31})^{32} || \text{prod31..0}
\]

\[
\text{HI63..0} \leftarrow (\text{prod 63})^{32} || \text{prod63..32}
\]

Exceptions:

None

Programming Notes:

In the C790, the integer multiply operation proceeds asynchronously and allows other CPU instructions to execute before it is retired. An attempt to read LO or HI before the results are written will wait (interlock) until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.
## MULTU

### Multiply Unsigned Word

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>16</th>
<th>15</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL 000000</td>
<td>rs</td>
<td>rt</td>
<td>0</td>
<td>00 0000 0000</td>
<td>MULTU 011001</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:**

MULTU rs, rt

**Purpose:**

To multiply 32-bit unsigned integers.

**Description:**

\[(LO, HI) \leftarrow rs \times rt\]

The 32-bit word value in GPR rt is multiplied by the 32-bit value in GPR rs, treating both operands as unsigned values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register LO, and the high-order 32-bit word is placed into special register HI.

No arithmetic exception occurs under any circumstances.

**Restrictions:**

If either GPR rt or GPR rs do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

**Operation:**

if (NotWordValue(GPR [rs]) or NotWordValue(GPR [rt])) then UndefinedResult() endif

prod \leftarrow (0 || GPR [rs]_{31..0}) \times (0 || GPR [rt]_{31..0})

LO_{63..0} \leftarrow \left( \text{prod}_{31} \right)_{32} \parallel \text{prod}_{31..0}

HI_{63..0} \leftarrow \left( \text{prod}_{63} \right)_{32} \parallel \text{prod}_{63..32}

**Exceptions:**

None

**Programming Notes:**

See the Programming Notes for the MULT instruction.
### NOR
Not Or

<table>
<thead>
<tr>
<th></th>
<th>SPECIAL</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
<th>NOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

#### MIPS I

**Format:** NOR rd, rs, rt

**Purpose:** To do a bitwise logical NOT OR.

**Description:**

\[
\text{rd} \leftarrow \text{rs} \text{ NOR rt}
\]

The contents of GPR rs are combined with the contents of GPR rt in a bitwise logical NOR operation. The result is placed into GPR rd.

**Restrictions:**

None

**Operation:**

\[
\text{GPR [rd]}_{63..0} \leftarrow \text{GPR [rs]}_{63..0} \text{ NOR GPR [rt]}_{63..0}
\]

**Exceptions:**

None

**Programming Notes:**

None
OR

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>000000</td>
<td>OR</td>
<td>100101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format: OR rd, rs, rt

Purpose: To do a bitwise logical OR.

Description: rd ← rs OR rt

The contents of GPR rs are combined with the contents of GPR rt in a bitwise logical OR operation. The result is placed into GPR rd.

Restrictions:

None

Operation:

GPR [rd]_{63..0} ← GPR [rs]_{63..0} or GPR [rt]_{63..0}

Exceptions:

None

Programming Notes:

None
### ORI

#### Or Immediate

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORI</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### MIPS I

**Format:** ORI rt, rs, immediate

**Purpose:** To do a bitwise logical OR with a constant.

**Description:**

rt ← rs OR immediate

The 16-bit immediate is zero-extended to the left and combined with the contents of GPR rs in a bitwise logical OR operation. The result is placed into GPR rt.

**Restrictions:**

None

**Operation:**

GPR [rt]_{63..0} ← zero_extend (immediate) or GPR [rs]_{63..0}

**Exceptions:**

None

**Programming Notes:**

None
PREF

Prefetch

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREF</td>
<td>base</td>
<td>hint</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110011</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS IV

Format:  PREF  hint, offset (base)

Purpose: To prefetch data from memory.

Description: prefetch_memory (base+offset)

PREF adds the 16-bit signed offset to the contents of GPR base to form an effective byte address. It advises that data at the effective address may be used in the near future.

If the hint field is 00000₂, this instruction prefetches a block of data from main memory into cache.

PREF is an advisory instruction. It may change the performance of the program. For all hint values and all effective addresses, it neither changes architecturally-visible state nor alters the meaning of the program.

PREF does not cause addressing-related exceptions. If it raises an exception condition, the exception conditions ignored. If an addressing-related exception condition is raised and ignored, no data will be prefetched. Even if no data is prefetched in such a case, some action that is not architecturally-visible, such as writeback of a dirty cache line, might take place.

PREF will never generate a memory operation for a location with an uncached memory access type.

The defined hint values are shown in the table below. The C790 only supports hint = 0. The hint table may be extended in future implementations.

Values of hint field for prefetch instruction

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Data use and desired prefetch action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>load</td>
<td>Data is expected to be loaded (not modified). Fetch data as if for a load.</td>
</tr>
<tr>
<td>1-31</td>
<td>(Reserved)</td>
<td>(Reserved)</td>
</tr>
</tbody>
</table>
Restrictions:

None

Operation:

\[ v\text{Addr} \leftarrow \text{sign\_extend (offset)} + \text{GPR [base] }_{31:0} \]

\( (p\text{Addr, uncached}) \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \)

Prefetch (uncached, pAddr, vAddr, DATA, hint)

Exceptions:

None

Programming Notes:

Prefetch can not prefetch data from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch on C790 may not prefetch data when there is outstanding bus read process due to a data cache miss, an uncached load or a miss on the uncached accelerated buffer.

Prefetch does not cause addressing exceptions. It will not cause an exception to prefetch using an address pointer value before the validity of a pointer determined.

Implementation Notes:

A reserved \textit{hint} field value causes a default prefetch action, the load \textit{hint}. 
SB

Store Byte

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB</td>
<td></td>
<td></td>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td>offset</td>
</tr>
<tr>
<td>101000</td>
<td>base</td>
<td></td>
<td></td>
<td></td>
<td>rt</td>
<td></td>
<td>offset</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td></td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format: SB rt, offset (base)

Purpose: To store a byte to memory.

Description: memory [base + offset] ← rt

The least-significant 8-bit byte of GPR rt is stored in memory at the location specified by the effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

Restrictions:

None

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + GPR [base]_{31..0}
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr[(PSIZE-1)..4] || (pAddr_{3..0} xor BigEndian^4)
byte ← vAddr_{3..0} xor BigEndian^4
dataquad ← GPR [rt]_{127-8-byte..0} || 0^byte
StoreMemory (uncached, BYTE, dataquad, pAddr, vAddr, DATA)

Exceptions:

TLB Refill
TLB Invalid
TLB Modified
Address Error

Programming Notes:

None
Store Doubleword

### Format:

```
SD  rt, offset (base)
```

### Purpose:

To store a doubleword to memory.

### Description:

```
memory [base + offset] ← rt
```

The 64-bit doubleword in GPR `rt` is stored in memory at the location specified by the aligned effective address. The 16-bit signed `offset` is added to the contents of GPR `base` to form the effective address.

### Restrictions:

The effective address must be naturally aligned. If any of the three least-significant bits of the effective address are non-zero, an Address Error exception occurs.

### Operation: (128-bit bus)

```
vAddr ← sign_extend (offset) + GPR [base] 31..0
if (vAddr2..0) ≠ 03 then SignalException (AddressError) endif
pAddr ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr (PSIZE-1..4 || (pAddr 3..0 xor (BigEndian || 03)))
byte ← vAddr 3..0 || (BigEndian || 03)
dataquad ← GPR [rt] 127-8..0 || 08*byte
StoreMemory (uncached, DOUBLEWORD, dataquad, pAddr, vAddr, DATA)
```

### Exceptions:

- TLB Refill
- TLB Invalid
- TLB Modified
- Address Error

### Programming Notes:

None
Appendix A  CPU Instruction Set Details

SDL

Store Doubleword Left

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDL</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101100</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS III

Format: SDL rt, offset (base)

Purpose: To store the more-significant part of a doubleword to an unaligned memory address.

Description: memory [base + offset] ← rt

Paired SDL and SDR instructions are used to store a doubleword from a register into eight consecutive bytes in memory starting at an arbitrary byte address. SDL stores the left (most-significant) bytes and SDR stores the right (least-significant) bytes.

The 16-bit signed offset is added to the contents of GPR base to form the effective address of the most-significant byte of the contiguous doubleword in memory. It alters only the doubleword in memory which contains that byte. From one to eight bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the most-significant byte of the register and copies it to the specified byte in memory; then it copies bytes from register to memory until it reaches the low-order byte of the word in memory.

No address exceptions due to alignment are possible.
Appendix A  CPU Instruction Set Details

Restrictions:

None

Operation:  (128-bit bus)

vAddr ← sign_extend (offset) + GPR [base]_{31..0}
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_{(PSIZE-1)..4} || (pAddr_{3..0} xor BigEndian^4)
If (BigEndian = 0) then
    pAddr ← pAddr_{(PSIZE-1)..3} || 0^3
endif
byte ← 0 || (vAddr_{2..0} xor BigEndian^3)
if (vAddr_{3} xor BigEndian = 0) then
    dataquad ← 0^64 || 0^{(56-8-byte)} || GPR [rt]_{63..(56-8-byte)}
else
    dataquad ← 0^{(56-8-byte)} || GPR [rt]_{63..(56-8-byte)} || 0^64
endif
StoreMemory (uncached, byte, dataquad, pAddr, vAddr, DATA)

Given a doubleword in a register and a doubleword in memory, the operation of SDL is as follows:
## SDL

| Register | MSB 63 | 0 | LSB |       | A | B | C | D | E | F | G | H |
|----------|--------|---|-----|-------|---|---|---|---|---|---|---|---|---|
| Little-endian | Memory | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |

### Little-endian byte ordering (BigEndianCPU = 1)

<table>
<thead>
<tr>
<th>vAddr3_0</th>
<th>Destination memory contents after instruction (shaded is unchanged)</th>
<th>Type</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(127---------------------------------------64 63------------------------------------------0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>i j k l m n o p q r s t u v w A</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>i j k l m n o p q r s t u v A B</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>i j k l m n o p q r s t u A B C</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>i j k l m n o p q r s t A B C D</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>i j k l m n o p q r s A B C D E</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>i j k l m n o p q r A B C D E F</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>i j k l m n o p q A B C D E F G</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>i j k l m n o p A B C D E F G H</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>i j k l m n o A q r s t u v w x</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>i j k l m n A B q r s t u v w x</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>i j k l m n A B C q r s t u v w x</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>i j k l m A B C D q r s t u v w x</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>i j k l m A B C D E q r s t u v w x</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>i j A B C D E F q r s t u v w x</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>i A B C D E F G q r s t u v w x</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>A B C D E F G q r s t u v w x</td>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>
### SDL

**Register**

<table>
<thead>
<tr>
<th>MSB 63</th>
<th>LSB 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

### Big-endian Memory

<table>
<thead>
<tr>
<th>Memory</th>
<th>ABCDEFGH</th>
<th>ijklnopqrstuvwxyz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Big-endian byte ordering (BigEndianCPU = 0)

<table>
<thead>
<tr>
<th>vAddr_{3..0}</th>
<th>Destination memory contents after instruction (shaded is unchanged)</th>
<th>Type AccessLength sent to memory</th>
<th>Offset pAddr_{3..0} sent to memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ABCDEFGH qrsstuvwxyz</td>
<td>LEM</td>
<td>BEM</td>
</tr>
<tr>
<td>1</td>
<td>iABCDDEFG qrsstuvwxyz</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>ijABCDDEF qrsstuvwxyz</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>ijkABCDqrsstuvwxyz</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>ijkIlABCDqrsstuvwxyz</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>ijkIlmABCDqrsstuvwxyz</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>ijkIlmnABqrsstuvwxyz</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>ijkIlmnopAqrsstuvwxyz</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>ijkIlmnopABCDqrsstuvwxyz</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>ijkIlmnopABCDqrsstuvwxyz</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>ijkIlmnopABCDqrsstuvwxyz</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>ijkIlmnopABCDqrsstuvwxyz</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>ijkIlmnopABCDqrsstuvwxyz</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>13</td>
<td>ijkIlmnopABCDqrsstuvwxyz</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>ijkIlmnopABCDqrsstuvwxyz</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>15</td>
<td>ijkIlmnopABCDqrsstuvwxyz</td>
<td>14</td>
<td>14</td>
</tr>
</tbody>
</table>

**LEM** Little-endian memory (BigEndianMem = 0)

**BEM** BigEndianMem = 1

**Type** AccessLength sent to memory

**Offset** pAddr_{3..0} sent to memory

### Exceptions:

- TLB Refill
- TLB Invalid
- TLB Modified
- Address Error

### Programming Notes:

None
Appendix A  CPU Instruction Set Details

SDR

Store Doubleword Right

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDR</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101101</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS III

Format:  SDR  rt, offset (base)

Purpose: To store the less-significant part of a doubleword to an unaligned memory address.

Description: memory [base + offset] ← rt

Paired SDL and SDR instructions are used to store a doubleword from a register into eight consecutive bytes in memory starting at an arbitrary byte address. SDL stores the left (most-significant) bytes and SDR stores the right (least-significant) bytes.

The SDR instruction adds its sign-extended 16-bit offset to the contents of GPR base to form an effective address which may specify an arbitrary byte. It alters only the doubleword in memory which contains that byte. From one to eight bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the least-significant (rightmost) byte of the register and copies it to the specified byte in memory; then it copies bytes from register to memory until it reaches the high-order byte of the word in memory. No address exceptions due to alignment are possible.

Restrictions:

None
Operation: (128-bit bus)

\[
vAddr \leftarrow \text{sign\_extend (offset)} + \text{GPR [base]}_{31..0} \\
(pAddr, \text{uncached}) \leftarrow \text{AddressTranslation} (vAddr, \text{DATA, STORE}) \\
pAddr \leftarrow pAddr_{\text{PSIZE-1..4}} \| (pAddr_{3..0} \text{ xor BigEndian}^4) \\
\text{If } (\text{BigEndian} = 0) \text{ then} \\
pAddr \leftarrow pAddr_{\text{PSIZE-31..3}} \| 0^3 \\
\text{endif} \\
\text{byte } \leftarrow vAddr_{2..0} \text{ xor BigEndian}^4 \\
\text{if}(vAddr_3 \text{ xor BigEndian }= 0) \text{ then} \\
dataquad \leftarrow 0^64 \| \text{GPR [rt]}_{63-8\times\text{byte}} \| 0^8\text{byte} \\
\text{else} \\
dataquad \leftarrow \text{GPR [rt]}_{63-8\times\text{byte}} \| 0^8\text{byte} \| 0^64 \\
\text{endif} \\
\text{StoreMemory (uncached, DOUBLEWORD-byte, dataquad, pAddr, vAddr, DATA)}
\]

Given a doubleword in a register and a doubleword in memory, the operation of SDR is as follows:
### SDR

**Register**

<table>
<thead>
<tr>
<th>MSB 63</th>
<th>0</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
<td></td>
</tr>
</tbody>
</table>

**Little-endian Memory**

| i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x |

**Little-endian byte ordering (BigEndianCPU = 0)**

<table>
<thead>
<tr>
<th>vAddr3_0</th>
<th>Destination memory contents after instruction (shaded is unchanged)</th>
<th>Type</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(127-------------------------------64 63--------------------------------------0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>i j k l m n o p A B C D E F G H 7 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>i j k l m n o p B C D E F G H x 6 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>i j k l m n o p C D E F G H w x 5 2 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>i j k l m n o p D E F G H v w x 4 3 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>i j k l m n o p E F G H u v w x 3 4 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>i j k l m n o p F G H t u v w x 2 5 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>i j k l m n o p G H s t u v w x 1 6 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>i j k l m n o p H r s t u v w x 0 7 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>A B C D E F G H q r s t u v w x 7 8 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>B C D E F G H p q r s t u v w x 6 9 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>C D E F G H o p q r s t u v w x 5 10 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>D E F G H n o p q r s t u v w x 4 11 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>E F G H m n o p q r s t u v w x 3 12 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>F G H l m n o p q r s t u v w x 2 13 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>G H k l m n o p q r s t u v w x 1 14 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>H j k l m n o p q r s t u v w x 0 15 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# SDR

## Register

<table>
<thead>
<tr>
<th>Big-endian</th>
<th>Memory</th>
<th>Little-endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</td>
<td>i j k l m n o p q r s t u v w x</td>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

## Big-endian byte ordering (BigEndianCPU = 0)

<table>
<thead>
<tr>
<th>vAddr&lt;sub&gt;3..0&lt;/sub&gt;</th>
<th>Destination memory contents after instruction (shaded is unchanged)</th>
<th>Type</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>14 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>13 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>11 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>6 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>5 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>4 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>3 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>2 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LEM** Little-endian memory (BigEndianMem = 0)

**BEM** BigEndianMem = 1

**Type** AccessLength sent to memory

**Offset** pAddr<sub>3..0</sub> sent to memory

**Exceptions:**

- TLB Refill
- TLB Invalid
- TLB Modified
- Address Error

**Programming Notes:**

None
Appendix A  CPU Instruction Set Details

SH

Store Halfword

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SH</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101001</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format: SH rt, offset (base)

Purpose: To store a halfword to memory.

Description: memory [base + offset] ← rt

The least-significant 16-bit halfword if register rt is stored in memory at the location specified by the aligned effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

Restrictions:

The effective address must be naturally aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + GPR [base]_31..0
if (vAddr_0) ≠ 0 then SignalException (AddressError) endif
pAddr, uncached ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr[PSIZE-1..4] || (pAddr_3..0 xor (BigEndian_3 || 0))
byte ← vAddr_3..0 xor (BigEndian_3 || 0)
dataquad ← GPR [rt]_127-8-byte..0 || 08-byte
StoreMemory (uncached, HALFWORD, dataquad, pAddr, vAddr, DATA)

Exceptions:

TLB Refill
TLB Invalid
TLB Modified
Address Error

Programming Notes:

None
SLL

Shift Word Left Logical

<table>
<thead>
<tr>
<th>0</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>SLL 000000</td>
</tr>
</tbody>
</table>

Format:
SLL rd, rt, sa

Purpose:
To left shift a word by a fixed number of bits.

Description:
rd ← rt << sa

The contents of the low-order 32-bit word of GPR rt are shifted left, inserting zeroes into the emptied bits; the word result is placed in GPR rd. The bit shift count is specified by sa. The result word is sign-extended.

Restrictions:
None

Operation:
s ← sa
temp ← GPR [rt][31-s..0] || 0s
GPR [rd][63..0] ← sign_extend (temp[31..0])

Exceptions:
None

Programming Notes:
Unlike nearly all other word operations the input operand does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign extended into a 64-bit destination register; this instruction with a zero shift amount truncates a 64-bit value to 32 bits and sign extends it and stores it in the destination register.
**SLLV**  
Shift Word Left Logical Variable  

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>SLLV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:**  
SLLV  rd, rt, rs

**Purpose:**  
To left shift a word by a variable number of bits.

**Description:**  
rd ← rt << rs

The contents of the low-order 32-bit word of GPR rt are shifted left, inserting zeroes into the emptied bits; the result word is placed in GPR rd. The bit shift count is specified by the low-order five bits of GPR rs. The result word is sign-extended.

**Restrictions:**
None

**Operation:**

\[
\begin{align*}
    s & \leftarrow \text{GP}[rs]_{4..0} \\
    \text{temp} & \leftarrow \text{GPR}[rt]_{31..0} \parallel 0^5 \\
    \text{GPR}[rd]_{63..0} & \leftarrow \text{sign\_extend}(\text{temp}_{31..0})
\end{align*}
\]

**Exceptions:**
None

**Programming Notes:**
None
**SLT**

Set on Less Than

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>SLT</td>
<td>101010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS I**

Format: SLT rd, rs, rt

Purpose: To record the result of a less-than comparison.

Description:

\[ \text{rd} \leftarrow (\text{rs} < \text{rt}) \]

Compare the contents of GPR rs and GPR rt as signed integers and record the Boolean result of the comparison in GPR rd. If GPR rs is less than GPR rt the result is 1 (true), otherwise 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

\[
\begin{align*}
\text{if } \text{GPR}[\text{rs}]_{63..0} & < \text{GPR}[\text{rt}]_{63..0} \text{ then} \\
\text{GPR}[\text{rd}]_{63..0} & \leftarrow \text{0}^{\text{GPRLEN}-1} \| 1 \\
\text{else} & \\
\text{GPR}[\text{rd}]_{63..0} & \leftarrow \text{0}^{\text{GPRLEN}}
\end{align*}
\]

Exceptions:

None

Programming Notes:

None
### SLTI

**Set on Less Than Immediate**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLTI</td>
<td>001010</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### MIPS I

**Format:** SLTI rt, rs, immediate  
**Purpose:** To record the result of a less-than comparison with a constant.  
**Description:** rt ← (rs < immediate)

Compare the contents of GPR rs and the 16-bit signed immediate as signed integers and record the Boolean result of the comparison in GPR rt. If GPR rs is less than immediate the result is 1 (true), otherwise 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:** None

**Operation:**

\[
\text{if GPR [rs]}_{63..0} < \text{sign\_extend (immediate)}\text{ then}
\]

\[
\text{GPR [rd]}_{63..0} \leftarrow 0^{\text{GPRLEN-1}} || 1
\]

\[\text{else}
\]

\[
\text{GPR [rd]}_{63..0} \leftarrow 0^{\text{GPRLEN}}
\]

\[\text{endif}
\]

**Exceptions:** None

**Programming Notes:** None
SLTIU

Set on Less Than Immediate Unsigned

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLTIU</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001011</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format: SLTIU rt, rs, immediate

Purpose: To record the result of an unsigned less-than comparison with a constant.

Description: rt ← (rs < immediate)

Compare the contents of GPR rs and the sign-extended 16-bit immediate as unsigned integers and record the Boolean result of the comparison in GPR rt. If GPR rs is less than immediate the result is 1 (true), otherwise 0 (false).

Because the 16-bit immediate is sign-extended before comparison, the instruction is able to represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

if (0 || GPR [rs] 63..0) < (0 || sign_extend (immediate)) then
    GPR [rd] 63..0 ← 0^GPRLEN-1 || 1
else
    GPR [rd] 63..0 ← 0^GPRLEN
endif

Exceptions:

None

Programming Notes:

None
### SLTU

Set on Less Than Unsigned

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>SLTU</td>
<td>101011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### MIPS I

**Format:**

SLTU rd, rs, rt

**Purpose:**

To record the result of an unsigned less-than comparison.

**Description:**

rd ← (rs < rt)

Compare the contents of GPR rs and GPR rt as unsigned integers and record the Boolean result of the comparison in GPR rd. If GPR rs is less than GPR rt the result is 1 (true), otherwise 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:**

None

**Operation:**

if (0 || GPR [rs]_{63..0} < (0 || GPR [rt]_{63..0}) then
    GPR [rd]_{63..0} ← 0^{GPRLEN-1} || 1
else
    GPR [rd]_{63..0} ← 0^{GPRLEN}
endif

**Exceptions:**

None

**Programming Notes:**

None
### SRA

**Shift Word Right Arithmetic**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>0</td>
<td>00000</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>SRA</td>
<td>000011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:** SRA rd, rt sa

**Purpose:** To arithmetic right shift a word by a fixed number of bits.

**Description:**

rd ← rt >> sa (arithmetic)

The contents of the low-order 32-bit word of GPR rt are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR rd. The bit shift count is specified by sa. The result word is sign-extended.

**Restrictions:**

If GPR rt does not contain a sign-extended 32-bit value (bit 63..31 equal) then the result of the operation is undefined.

**Operation:**

if (NotWordValue (GPR [rt] 63..0 )) then UndefinedResult () endif  
s ← sa  
temp ← (GPR [rt] 31) || GPR [rt] 31..5  
GPR [rd] 63..0 ← sign_extend (temp 31..0)

**Exceptions:**

None

**Programming Notes:**

None
Appendix A  CPU Instruction Set Details

SRAV  Shift Word Right Arithmetic Variable

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>00000</td>
<td>SRAV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format:  SRAV  rd, rt, rs
Purpose:  To arithmetic right shift a word by a variable number of bits.
Description:  rd ← rt >> rs  (arithmetic)

The contents of the low-order 32-bit word of GPR rt are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR rd. The bit shift count is specified by the low-order five bits of GPR rs. The result word is sign-extended.

Restrictions:

If GPR rt does not contain a sign-extended 32-bit value (bit 63..31 equal) then the result of the operation is undefined.

Operation:

if (NotWordValue (GPR [rt]63..0)) then UndefinedResult () endif
s ← GPR [rs]4..0
temp ← (GPR [rt]31)∥ (GPR [rt]31..5)
GPR [rd]63..0 ← sign_extend (temp31..0)

Exceptions:

None

Programming Notes:

None
Appendix A  CPU Instruction Set Details

SRL  Shift Word Right Logical

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>SRL</td>
<td>000010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format: SRL rd, rt, sa
Purpose: To logical right shift a word by a fixed number of bits.
Description: rd ← rt >> sa (logical)

The contents of the low-order 32-bit word of GPR rt are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR rd. The bit shift count is specified by sa. The result word is sign-extended.

Restrictions:
If GPR rt does not contain a sign-extended 32-bit value (bit 63..31 equal) then the result of the operation is undefined.

Operation:
if (NotWordValue (GPR [rt] 63..0)) then UndefinedResult () endif
s ← sa
temp ← 0s || GPR [rt]31..5
GPR [rd]63..0 ← sign_extend(temp31..0)

Exceptions:
None

Programming Notes:
None
### SRLV

**Shift Word Right Logical Variable**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>000000</td>
<td>SRLV</td>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:**

SRLV rd, rt, rs

**Purpose:**

To logical right shift a word by a variable number of bits.

**Descriptions:**

rd ← rt >> rs (logical)

The contents of the low-order 32-bit word of GPR rt are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR rd. The bit shift count is specified by the low-order five bits of GPR rs. The result word is sign-extended.

**Restrictions:**

If GPR rt does not contain a sign-extended 32-bit value (bits 63..31 equal) then the result of the operation is undefined.

**Operation:**

if (NotWordValue (GPR[rt] 63..0)) then UndefinedResult () endif

s ← GPR [rs]4..0

temp ← 0s || GPR [rt]31..s

GPR [rd] 63..0 ← sign_extend (temp31..0)

**Exceptions:**

None

**Programming Notes:**

None
SUB

Subtract Word

<table>
<thead>
<tr>
<th>SUB</th>
<th>SUB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SPECIAL</td>
<td>rs</td>
</tr>
<tr>
<td>000000</td>
<td>000000</td>
</tr>
</tbody>
</table>

MIPS I

Format: SUB rd, rs, rt

Purpose: To subtract 32-bit integers. If overflow occurs, then trap.

Description: rd ← rs - rt

The 32-bit word value in GPR rt is subtracted from the 32-bit value in GPR rs to produce a 32-bit result. If the subtraction results in 32-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR rd.

Restrictions:

If either GPR rt or GPR rs do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:

if (NotWordValue (GPR[rs] 63..0) or NotWordValue (GPR[rt] 63..0)) then UndefinedResult () endif

temp ← GPR[rs] 63..0 - GPR[rt] 63..0

if (32_bit_arithmetic_overflow) then
    SignalException (IntegerOverflow)
else
    GPR [rd] 63..0 ← sign_extend (temp31..0)
endif

Exceptions:

Integer Overflow

Programming Notes:

SUBU performs the same arithmetic operation but, does not trap on overflow.
**SUBU** Subtract Unsigned Word **SUBU**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>000000</td>
<td>SUBU</td>
<td>100011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:** 
SUBU rd, rs, rt

**Purpose:** 
To subtract 32-bit integers.

**Description:** 
rd ← rs - rt

The 32-bit word value in GPR rt is subtracted from the 32-bit value in GPR rs and the 32-bit arithmetic result is placed into GPR rd.

**Restrictions:** 
If either GPR rt or GPR rs do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

**Operation:**

if (NotWordValue (GPR[rs] 63..0) or NotWordValue (GPR[rt] 63..0)) then UndefinedResult () endif

temp ← GPR [rs] 63..0 - GPR [rt] 63..0

GPR [rd] 63..0 ← sign_extend (temp31..0)

**Exceptions:**
None

**Programming Notes:**

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.
### SW

**Store Word**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101011</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:**

SW  rt, offset (base)

**Purpose:**

To store a word to memory.

**Description:**

memory [base + offset] ← rt

The least-significant 32-bit word of register rt is stored in memory at the location specified by the aligned effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

**Restrictions:**

The effective address must be naturally aligned. If either of the two least-significant bits of the address are non-zero, an Address Error exception occurs.

**Operation:** (128-bit bus)

\[
\text{vAddr} \leftarrow \text{sign}_\text{extend (offset)} + \text{GPR [base]}_\text{31..0}
\]

\[
\text{if (vAddr}_{1..0} \neq 0^2 \text{ then SignalException (AddressError) endif}
\]

\[
\text{pAddr} \leftarrow \text{AddressTranslation (vAddr, DATA, STORE)}
\]

\[
\text{pAddr} \leftarrow \text{pAddr}_{(\text{PSIZE}-1..4)} || (\text{pAddr}_{3..0} \text{xor (BigEndian}_2 || 0^2))
\]

\[
\text{byte} \leftarrow \text{vAddr}_{3..0} \text{xor (BigEndian}_2 || 0^2)
\]

\[
\text{dataquad} \leftarrow \text{GPR [rt]}_{(127\text{-byte}.0} || 0^8\text{byte}
\]

\[
\text{StoreMemory (uncached, WORD, dataquad, pAddr, vAddr, DATA)}
\]

**Exceptions:**

- TLB Refill
- TLB Invalid
- TLB Modified
- Address Error

**Programming Notes:**

None
### SWL

**Store Word Left**

<table>
<thead>
<tr>
<th>CPU Instruction</th>
<th>Format</th>
<th>Purpose</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWL</td>
<td>SWL rt, offset (base)</td>
<td>To store the more-significant part of a word to an unaligned memory address.</td>
<td>memory [base + offset] ← rt</td>
</tr>
</tbody>
</table>

Paired SWL and SWR instructions are used to store a word from a register into four consecutive bytes in memory starting at an arbitrary byte address. SWL stores the left (most-significant) bytes and SWR stores the right (least-significant) bytes.

The SWL instruction adds its sign-extended 16-bit offset to the contents of GPR base to form an effective address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the most-significant byte of the register and copies it to the specified byte in memory; then it copies bytes from register to memory until it reaches the low-order byte of the word in memory.

No address exceptions due to alignment are possible.
Restrictions:

None

Operation:

\[ vAddr \leftarrow \text{sign extend (offset)} + \text{GPR [base]}_{31..0} \]
\[ (pAddr, \text{uncached}) \leftarrow \text{AddressTranslation (vAddr, DATA, STORE)} \]
\[ pAddr \leftarrow pAddr^{PSIZE-1..4} \| (pAddr^{3..0} \oplus \text{BigEndian}^4) \]
If (BigEndian = 0) then
\[ pAddr \leftarrow pAddr^{PSIZE-1..2} \| 0^2 \]
endif
byte \leftarrow vAddr^{1..0} \oplus \text{BigEndian}^2
if (vAddr^{3..2} \oplus \text{BigEndian}^2) = 00^2 then
\[ \text{dataquad} \leftarrow 0^{96} \| 0^{24-8 \times \text{byte}} \| \text{GPR[rt]}_{31..(24-8 \times \text{byte})} \]
elseif (vAddr^{3..2} \oplus \text{BigEndian}^2) = 01^2 then
\[ \text{dataquad} \leftarrow 0^{64} \| 0^{24-8 \times \text{byte}} \| \text{GPR [rt]}_{31..(24-8 \times \text{byte})} \| 0^{32} \]
elseif (vAddr^{3..2} \oplus \text{BigEndian}^2) = 10^2 then
\[ \text{dataquad} \leftarrow 0^{32} \| 0^{24-8 \times \text{byte}} \| \text{GPR [rt]}_{31..(24-8 \times \text{byte})} \| 0^{32} \]
elseif (vAddr^{3..2} \oplus \text{BigEndian}^2) = 11^2 then
\[ \text{dataquad} \leftarrow 0^{16} \| \text{GPR [rt]}_{31..(24-8 \times \text{byte})} \| 0^{64} \]
endif
StoreMemory (uncached, byte, dataquad, pAddr, vAddr, DATA)

Given a doubleword in a register and a doubleword in memory, the operation of SWL is as follows:
### SWL

#### Register

<table>
<thead>
<tr>
<th>MSB</th>
<th>63</th>
<th>0</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
</tbody>
</table>

#### Memory

|     |     |   |     |   |     |   |     |   |     |   |     |   |     |   |     |   |     |   |
|-----|-----|---|-----|---|-----|---|-----|---|-----|---|-----|---|-----|---|-----|---|-----|
|     | 15  | 14| 13  | 12 | 11  | 10|  9  |  8 |  7  |  6 |  5  |  4 |  3  |  2 |  1  |  0 |
| i   | j   | k | l   | m | n   | o | p   | q | r   | s | t   | u | v   | w | x   |   |     |

#### Little-endian byte ordering (BigEndianCPU = 0)

<table>
<thead>
<tr>
<th>vAddr3..0</th>
<th>Destination memory contents after instruction(shaded is unchanged)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(127---------------------------------------64  63------------------------------------------0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEM</td>
<td>BEM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vAddr3..0</th>
<th>Little-endian byte ordering (BigEndianCPU = 0)</th>
<th>Type</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Destination memory contents after instruction(shaded is unchanged)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(127---------------------------------------64  63------------------------------------------0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>i j k l m n o p q r s t u v w E</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>i j k l m n o p q r s t u v E F</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>i j k l m n o p q r s t u E F G</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>i j k l m n o p q r s t E F G H</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>i j k l m n o p q r s E u v w x</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>i j k l m n o p q r E F u v w x</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>i j k l m n o p q E F G u v w x</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>i j k l m n o p E F G H u v w x</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>i j k l m n o E q r s t u v w x</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>i j k l m n E F q r s t u v w x</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>i j k l m E F G q r s t u v w x</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>i j k l E F G H q r s t u v w x</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>i j k E m n o p q r s t u v w x</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>i j E F m n o p q r s t u v w x</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>i E F G m n o p q r s t u v w x</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>15</td>
<td>E F G H m n o p q r s t u v w x</td>
<td>3</td>
<td>12</td>
</tr>
</tbody>
</table>
### SWL

**Register**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB 63</td>
<td>0</td>
<td>LSB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Big-endian Memory**

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>k</th>
<th>l</th>
<th>m</th>
<th>n</th>
<th>o</th>
<th>p</th>
<th>q</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>u</th>
<th>v</th>
<th>w</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

**Little-endian Memory**

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>k</th>
<th>l</th>
<th>m</th>
<th>n</th>
<th>o</th>
<th>p</th>
<th>q</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>u</th>
<th>v</th>
<th>w</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Big-endian byte ordering (BigEndianCPU = 1)**

<table>
<thead>
<tr>
<th>vAddr_{3..0}</th>
<th>Destination memory contents after instruction (shaded is unchanged)</th>
<th>Type</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>E F G H m n o p q r s t u v w x</td>
<td>3</td>
<td>12 0</td>
</tr>
<tr>
<td>1</td>
<td>i E G H m n o p q r s t u v w x</td>
<td>2</td>
<td>12 1</td>
</tr>
<tr>
<td>2</td>
<td>i j E F m n o p q r s t u v w x</td>
<td>1</td>
<td>12 2</td>
</tr>
<tr>
<td>3</td>
<td>i j k E m n o p q r s t u v w x</td>
<td>0</td>
<td>12 3</td>
</tr>
<tr>
<td>4</td>
<td>i j k l E F G H q r s t u v w x</td>
<td>3</td>
<td>8 4</td>
</tr>
<tr>
<td>5</td>
<td>i j k l m E F G q r s t u v w x</td>
<td>2</td>
<td>8 5</td>
</tr>
<tr>
<td>6</td>
<td>i j k l m n E F q r s t u v w x</td>
<td>1</td>
<td>8 6</td>
</tr>
<tr>
<td>7</td>
<td>i j k l m n o E q r s t u v w x</td>
<td>0</td>
<td>8 7</td>
</tr>
<tr>
<td>8</td>
<td>i j k l m n o p E F G H u v w x</td>
<td>3</td>
<td>4 8</td>
</tr>
<tr>
<td>9</td>
<td>i j k l m n o p q E F G u v w x</td>
<td>2</td>
<td>4 9</td>
</tr>
<tr>
<td>10</td>
<td>i j k l m n o p q r E F u v w x</td>
<td>1</td>
<td>4 10</td>
</tr>
<tr>
<td>11</td>
<td>i j k l m n o p q r s F u v w x</td>
<td>0</td>
<td>4 11</td>
</tr>
<tr>
<td>12</td>
<td>i j k l m n o p q r s t E F G H</td>
<td>3</td>
<td>0 12</td>
</tr>
<tr>
<td>13</td>
<td>i j k l m n o p q r s t u E F G</td>
<td>2</td>
<td>0 13</td>
</tr>
<tr>
<td>14</td>
<td>i j k l m n o p q r s t u v E F</td>
<td>1</td>
<td>0 14</td>
</tr>
<tr>
<td>15</td>
<td>i j k l m n o p q r s t u v w F</td>
<td>0</td>
<td>0 15</td>
</tr>
</tbody>
</table>

**Exceptions:**

- TLB Refill
- TLB Invalid
- TLB Modified
- Address Error

**Programming Notes:**

None
SWR
Store Word Right

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWR</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101110</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:**  
SWR rt, offset (base)

**Purpose:**  
To store the less-significant part of a word to an unaligned memory address.

**Description:**  
memory [base + offset] ← rt

Paired SWL and SWR instructions are used to store a word from a register into four consecutive bytes in memory starting at an arbitrary byte address. SWL stores the left (most-significant) bytes and SWR stores the right (least-significant) bytes.

The SWR instruction adds its sign-extended 16-bit offset to the contents of GPR base to form an effective address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the least-significant (rightmost) byte of the register and copies it to the specified byte in memory; then copies bytes from register to memory until it reaches the high-order byte of the word in memory.

No address exceptions due to alignment are possible.
Restrictions:
None

Operation:
- vAddr ← sign_extend (offset) + GPR [base]_{31..0}
- (pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
- pAddr ← pAddr_{PSIZE-1..4} || (pAddr_{3..0} xor BigEndian^4)
- If (BigEndian = 0) then
  - pAddr ← pAddr_{PSIZE-1..2} || 0^2
- endif
- byte ← vAddr_{1..0} xor BigEndian^2
- if (vAddr_{3..2} xor BigEndian^2) = 00^2 then
  - dataquad ← 0^96 || GPR [rt]_{31-8-byte}..0 || 0^{3-byte} || 0^byte
- else if (vAddr_{3..2} xor BigEndian^2) = 01^2 then
  - dataquad ← 0^64 || GPR [rt]_{31-8-byte}..0 || 0^{3-byte} || 0^{32-byte}
- else if (vAddr_{3..2} xor BigEndian^2) = 10^2 then
  - dataquad ← 0^{32-byte} || GPR [rt]_{31-8-byte}..0 || 0^{3-byte} || 0^{64-byte}
- else if (vAddr_{3..2} xor BigEndian^2) = 11^2 then
  - dataquad ← GPR [rt]_{31-8-byte}..0 || 0^{3-byte} || 0^{96-byte}
- endif
- StoreMemory (uncached, WORD-byte, dataquad, pAddr, vAddr, DATA)

Given a doubleword in a register and a doubleword in memory, the operation of SWR is as follows:
Appendix A CPU Instruction Set Details

**SWR**

<table>
<thead>
<tr>
<th>Register</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
</table>

**Little-endian Memory**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td>o</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
<td>x</td>
</tr>
</tbody>
</table>

**Little-endian byte ordering (BigEndianCPU = 0)**

<table>
<thead>
<tr>
<th>vAddr3_0</th>
<th>Destination memory contents after instruction(shaded is unchanged)</th>
<th>Type offset</th>
<th>LEM</th>
<th>BEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>(127------..64 63------.0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>i j k l m n o p q r s t E F G H</td>
<td>3</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>i j k l m n o p q r s t F G H x</td>
<td>2</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>i j k l m n o p q r s t G H w x</td>
<td>1</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>i j k l m n o p q r s t H v w x</td>
<td>0</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>i j k l m n o p E F G H u v w x</td>
<td>3</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>i j k l m n o p F G H t u v w x</td>
<td>2</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>i j k l m n o p G H s t u v w x</td>
<td>1</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>i j k l m n o p H r s t u v w x</td>
<td>0</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>i j k l E F G H q r s t u v w x</td>
<td>3</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>i j k l F G H p q r s t u v w x</td>
<td>2</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>i j k l G H o p q r s t u v w x</td>
<td>1</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>i j k l H n o p q r s t u v w x</td>
<td>0</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>E F G H m n o p q r s t u v w x</td>
<td>3</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>F G H l m n o p q r s t u v w x</td>
<td>2</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>G H k l m n o p q r s t u v w x</td>
<td>1</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>H j k l m n o p q r s t u v w x</td>
<td>0</td>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>
## SWR

**Register**

<table>
<thead>
<tr>
<th>MSB 63</th>
<th>0</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
<td></td>
</tr>
</tbody>
</table>

**Big-endian Memory**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td>o</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
<td>x</td>
</tr>
</tbody>
</table>

**Little-endian Memory**

| 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |

**Big-endian byte ordering (BigEndianCPU = 1)**

<table>
<thead>
<tr>
<th>vAddr3..0</th>
<th>Destination memory contents after instruction (shaded is unchanged)</th>
<th>Type</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H ij kl mn op qr rs tu uv w x</td>
<td>0</td>
<td>15 0</td>
</tr>
<tr>
<td>1</td>
<td>G H ij kl mn op qr rs tu uv w x</td>
<td>1</td>
<td>14 0</td>
</tr>
<tr>
<td>2</td>
<td>F G H ij kl mn op qr rs tu uv w x</td>
<td>2</td>
<td>13 0</td>
</tr>
<tr>
<td>3</td>
<td>E F G H ij kl mn op qr rs tu uv w x</td>
<td>3</td>
<td>12 0</td>
</tr>
<tr>
<td>4</td>
<td>i j k l H ij kl mn op qr rs tu uv w x</td>
<td>0</td>
<td>11 4</td>
</tr>
<tr>
<td>5</td>
<td>i j k l G H ij kl mn op qr rs tu uv w x</td>
<td>1</td>
<td>10 4</td>
</tr>
<tr>
<td>6</td>
<td>i j k l F G H ij kl mn op qr rs tu uv w x</td>
<td>2</td>
<td>9 4</td>
</tr>
<tr>
<td>7</td>
<td>i j k l E F G H ij kl mn op qr rs tu uv w x</td>
<td>3</td>
<td>8 4</td>
</tr>
<tr>
<td>8</td>
<td>i j k l H ij kl mn op qr rs tu uv w x</td>
<td>0</td>
<td>7 8</td>
</tr>
<tr>
<td>9</td>
<td>i j k l G H ij kl mn op qr rs tu uv w x</td>
<td>1</td>
<td>6 8</td>
</tr>
<tr>
<td>10</td>
<td>i j k l F G H ij kl mn op qr rs tu uv w x</td>
<td>2</td>
<td>5 8</td>
</tr>
<tr>
<td>11</td>
<td>i j k l E F G H ij kl mn op qr rs tu uv w x</td>
<td>3</td>
<td>4 8</td>
</tr>
<tr>
<td>12</td>
<td>i j k l H ij kl mn op qr rs tu uv w x</td>
<td>0</td>
<td>3 12</td>
</tr>
<tr>
<td>13</td>
<td>i j k l G H ij kl mn op qr rs tu uv w x</td>
<td>1</td>
<td>2 12</td>
</tr>
<tr>
<td>14</td>
<td>i j k l F G H ij kl mn op qr rs tu uv w x</td>
<td>2</td>
<td>1 12</td>
</tr>
<tr>
<td>15</td>
<td>i j k l E F G H ij kl mn op qr rs tu uv w x</td>
<td>3</td>
<td>0 12</td>
</tr>
</tbody>
</table>

**LEM** Little-endian memory (BigEndianMem = 0)

**BEM** BigEndianMem = 1

**Type** AccessLength sent to memory

**Offset** pAddr3..0 sent to memory

**Exceptions:**

- TLB Refill
- TLB Invalid
- TLB Modified
- Address Error

**Programming Notes:**

None
SYNC.stype  
Synchronize Shared Memory  
SYNC.stype

| SPECIAL  | 0 | stype | 01111 | SYNC  
|---------|---|-------|-------|------  
| 000000  | 000 0000 0000 0000 | 000 0000 0000 0000 | 00111  | 00111 |

MIPS II

Format:
SYNC  (stype = 0xxxx)
SYNC.L  (stype = 0xxxx)
SYNC.P  (stype = 1xxxx)

Purpose: To perform either a memory barrier operation or a pipeline barrier operation.

Description:
This instruction either interlocks the pipeline until all pending loads and stores are completed or all earlier issued instructions are completed.

In case of the SYNC or the SYNC.L instructions (memory barrier) all pending loads and stores are retired. Loads are retired when the destination register is written. Stores are retired when the stored data (in store buffers or write buffers) is either stored in the data cache, or sent on the processor bus and SYSDACK* has been asserted. All uncached accelerated data gathering operation is terminated. The uncached accelerated buffer is invalidated. All bus read processes due to load/store/pref/cache instructions are completed. All pending bus write processes in the write back buffer are completed.

In case of the SYNC.P instruction (pipeline barrier) all instructions prior to the barrier are completed before the instructions following the barrier operation are fetched. Note that the barrier operation does not wait for any instruction which was issued prior to the barrier operation but not retired (e.g., multiply, divide, multicycle COP1 operations or a pending load which were issued prior to the barrier operation).

Operation:
SyncOperation (stype)

Exceptions:
None

Programming Notes:
The SYNC instruction (SYNC.P or SYNC.L) is not allowed in the branch delay slot of instructions which have branch delay slots.
SYSCALL System Call

MIPS I

**Format:** SYSCALL

**Purpose:** To cause a System Call exception.

**Description:**

A system call exception occurs, immediately and unconditionally transferring control to the exception handler.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

**Restrictions:**

None

**Operation:**

SignalException (SystemCall)

**Exceptions:**

System Call

**Programming Notes:**

None
### TEQ

**Trap if Equal**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>code</td>
<td>TEQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>110100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### MIPS II

**Format:** TEQ rs, rt

**Purpose:** To compare GPRs and do a conditional Trap.

**Description:**

if (rs = rt) then Trap

Compare the contents of GPR rs and GPR rt as signed integers; if GPR rs is equal to GPR rt then take a Trap exception.

The contents of the code field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**

None

**Operation:**

if GPR[rs]_{63..0} = GPR[rt]_{63..0} then
    SignalException (Trap)
endif

**Exceptions:**

Trap

**Programming Notes:**

None
Appendix A  CPU Instruction Set Details

**TEQI**  
Trap if Equal Immediate  

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGIMM 000001</td>
<td>rs</td>
<td>TEQI 01100</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS II**

Format: TEQI rs, immediate

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs = immediate) then Trap

Compare the contents of GPR rs and the 16-bit signed immediate as signed integer; if GPR rs is equal to immediate then taken a Trap exception.

Restrictions:
   None

Operation:
   if GPR [rs]63..0 = sign_extend (immediate) then
     SignalException (Trap)
   endif

Exceptions:
   Trap

Programming Notes:
   None
### TGE

**Trap if Greater or Equal**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL 000000</td>
<td>rs</td>
<td>rt</td>
<td>code</td>
<td>TGE 110000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### MIPS II

**Format:**
TGE rs, rt

**Purpose:**
To compare GPRs and do a conditional Trap.

**Description:**
if (rs ≥ rt) then Trap

Compare the contents of GPR rs and GPR rt as signed integers; if GPR rs is greater than or equal to GPR rt then take a Trap exception.

The contents of the code field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**
None

**Operation:**
if GPR [rs]63.0 ≥ GPR [rt]63.0 then
    SignalException (Trap)
endif

**Exceptions:**
Trap

**Programming Notes:**
None
### TGEI

#### Trap if Greater or Equal Immediate

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>TGEI</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td>0</td>
<td>01000</td>
<td>0100000</td>
</tr>
</tbody>
</table>

**MIPS II**

**Format:**
TGEI  rs, immediate

**Purpose:**
To compare a GPR to a constant and do a conditional Trap.

**Description:**
if (rs \(\geq\) immediate) then Trap

Compare the contents of GPR rs and the 16-bit signed immediate as signed integers; if GPR rs is greater than or equal to immediate then take a Trap exception.

**Restrictions:**
None

**Operation:**
if GPR [rs]_{63..0} \(\geq\) sign_extend (immediate) then
SignalException (Trap)
endif

**Exceptions:**
Trap

**Programming Notes:**
None
Appendix A  CPU Instruction Set Details

**TGEIU**  
Trap if Greater or Equal Immediate Unsigned

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>TGEIU</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td>7</td>
<td>01001</td>
<td></td>
</tr>
</tbody>
</table>

**MIPS II**

**Format:**  
TGEIU rs, immediate

**Purpose:**  
To compare a GPR to a constant and do a conditional Trap.

**Description:**  
if (rs \geq\ immediate) then Trap

Compare the contents of GPR rs and the 16-bit sign-extended immediate as unsigned integers; if GPR rs is greater than or equal to immediate then take a Trap exception.

Because the 16-bit immediate is sign-extended before comparison, the instruction is able to represent the smallest or largest unsigned numbers. The representable values are at the minimum [0,32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

**Restrictions:**

None

**Operation:**

\[
\text{if } (0 \parallel \text{GPR}[rs]_{63:0}) \geq (0 \parallel \text{sign_extend}(\text{immediate})) \text{ then }
\]

SignalException (Trap)

**Exceptions:**

Trap

**Programming Notes:**

None
TGEU          Trap if Greater or Equal Unsigned

|   | 31 | 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  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|
|   | SPECIAL | rs | rt | code | TGEU | 110001 |
|   | 6 | 5 | 5 | 10 | 6 |

**MIPS II**

**Format:**
TGEU rs, rt

**Purpose:**
To compare GPRs and do a conditional Trap.

**Description:**
if (rs ≥ rt) then Trap

Compare the contents of GPR rs and GPR rt as unsigned integers; if GPR rs is greater than or equal to GPR rt then take a Trap exception.

The contents of the code field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**
None

**Operation:**
if (0 || GPR[rs]₆₃..₀) ≥ (0 || GPR[rt]₆₃..₀) then
  SignalException (Trap)
endif

**Exceptions:**
Trap

**Programming Notes:**
None
Appendix A  CPU Instruction Set Details

**TLT**  
Trap if Less Than

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>code</th>
<th>TLT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL 000000</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>110010</td>
<td></td>
</tr>
</tbody>
</table>

**MIPS II**

**Format:** TLT  rs, rt

**Purpose:** To compare GPRs and do a conditional Trap.

**Description:**

if (rs < rt) then Trap

Compare the contents of GPR rs and GPR rt as signed integers; if GPR rs is less than GPR rt then take a Trap exception.

The contents of the code field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**

None

**Operation:**

if GPR [rs]$_{63..0}$ < GPR [rt]$_{63..0}$ then

SignalException (Trap)

endif

**Exceptions:**

Trap

**Programming Notes:**

None
Appendix A  CPU Instruction Set Details

TLTI  Trap if Less Than Immediate

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>TLTI 01010</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

MIPS II

Format: TLTI rs, immediate

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs < immediate) then Trap

Compare the contents of GPR rs and the 16-bit signed immediate as signed integers; if GPR rs is less than immediate then take a Trap exception.

Restrictions:
None

Operation:
if GPR[rs]_{63..0} < sign_extend (immediate) then
  SignalException (Trap)
endif

Exceptions:
Trap

Programming Notes:
None
**TLTIU**

Traps if Less Than Immediate Unsigned

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>TLTIU</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td>01011</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS II**

Format: TLTIU rs, immediate

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs < immediate) then Trap

Compare the contents of GPR rs and the 16-bit sign-extended `immediate` as unsigned integers; if GPR rs is less than `immediate` then take a Trap exception.

Because the 16-bit `immediate` is sign-extended before comparison, the instruction is able to represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max(unsigned)-32767, max(unsigned)] end of the unsigned range.

Restrictions:

None

Operation:

if (0 || GPR[rs]_63..0) < (0 || sign_extend (immediate)) then
    SignalException (Trap)
endif

Exceptions:

Trap

Programming Notes:

None
### TLTU - Trap if Less Than Unsigned

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL 000000</td>
<td>rs</td>
<td>rt</td>
<td>code</td>
<td>TLTU 110011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### MIPS II

**Format:** TLTU rs, rt

**Purpose:** To compare GPRs and do a conditional Trap.

**Description:**

if (rs < rt) then Trap

Compare the contents of GPR rs and GPR rt as unsigned integers; if GPR rs is less than GPR rt then take a Trap exception.

The contents of the code field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:** None

**Operation:**

if (0 || GPR[rs]_{63..0}) < (0 || GPR[rt]_{63..0}) then
  SignalException (Trap)
endif

**Exceptions:** Trap

**Programming Notes:** None
TNE

Trap if Not Equal

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>rs</td>
<td>rt</td>
<td>code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS II**

**Format:**

TNE rs, rt

**Purpose:**

To compare GPRs and do a conditional Trap.

**Description:**

if (rs \neq rt) then Trap

Compare the contents of GPR rs and GPR rt as signed integers; if GPR rs is not equal to GPR rt then take a Trap exception.

The contents of the code field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**

None

**Operation:**

if GPR[rs]_{63..0} \neq GPR[rt]_{63..0} then
    SignalException (Trap)
endif

**Exceptions:**

Trap

**Programming Notes:**

None
TNEI  Trap if Not Equal Immediate

TNEI

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGIMM</td>
<td>000001</td>
<td>rs</td>
<td>TNEI</td>
<td>01110</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS II

Format: TNEI rs, immediate

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs ≠ immediate) then Trap

Compare the contents of GPR rs and the 16-bit signed immediate as signed integers; if GPR rs is not equal to immediate then take a Trap exception.

Restriction: None

Operation:

\[
\text{if } \text{GPR}[rs]_{63..0} \neq \text{sign\_extend} (\text{immediate}) \text{ then}
\]

\[
\text{SignalException (Trap)}
\]

Exceptions: Trap

Programming Notes: None
Appendix A  CPU Instruction Set Details

XOR

<table>
<thead>
<tr>
<th>XOR</th>
<th>Exclusive OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd</td>
<td>rs, rt</td>
</tr>
</tbody>
</table>

MIPS I

Format: XOR rd, rs, rt

Purpose: To do a bitwise logical EXCLUSIVE OR.

Description: rd ← rs XOR rt

Combine the contents of GPR rs and GPR rt in a bitwise logical exclusive OR operation and place the result into GPR rd.

Restrictions:
None

Operation:
GPR[rd]_{63..0} ← GPR[rs]_{63..0} xor GPR[rt]_{63..0}

Exceptions:
None

Programming Notes:
None
### XORI

**Exclusive OR Immediate**

<table>
<thead>
<tr>
<th>XORI</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>001110</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

#### MIPS I

**Format:**

XORI  rt, rs, immediate

**Purpose:**

To do a bitwise logical EXCLUSIVE OR with a constant.

**Description:**

rt ← rs XOR immediate

Combine the contents of GPR rs and the 16-bit zero-extended immediate in a bitwise logical exclusive OR operation and place the result into GPR rt.

**Restrictions:**

None

**Operation:**

GPR[rt]_{63..0} ← GPR[rs]_{63..0} xor zero_extend (immediate)

**Exceptions:**

None

**Programming Notes:**

None
### A.5 CPU Instruction Encoding

The following table shows the OpCode encoding of CPU instructions for the MIPS IV architecture. This architecture level includes all MIPS I, MIPS II, MIPS III and some MIPS IV instructions. Even though the OpCodes for MTSAB, MTSAH, MFSA, MTSA, LQ, and SQ are shown in this OpCode table, these instructions are described in Appendix B since they are C790-specific instructions.

Coprocessor 0 (COP0 - System Control Processor), Coprocessor 1 (COP1 - Floating-point Processor) and C790 specific instructions are described in separate sections.

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Instructions encoded by OpCode field</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 28..26</td>
<td></td>
</tr>
<tr>
<td>bits 31..29</td>
<td>000 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td>0 000</td>
<td>SPECIAL REGIMM δ</td>
</tr>
<tr>
<td>1 001</td>
<td>ADDI</td>
</tr>
<tr>
<td>2 010</td>
<td>COP0 α λ COP1 α λ</td>
</tr>
<tr>
<td>3 011</td>
<td>DADDI</td>
</tr>
<tr>
<td>4 100</td>
<td>LB</td>
</tr>
<tr>
<td>5 101</td>
<td>SB</td>
</tr>
<tr>
<td>6 110</td>
<td>η</td>
</tr>
<tr>
<td>7 111</td>
<td>η</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>function</th>
<th>Instructions encoded by function field when OpCode field = SPECIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 5..0</td>
<td>000 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td>0 000</td>
<td>SLL</td>
</tr>
<tr>
<td>1 001</td>
<td>JR</td>
</tr>
<tr>
<td>2 010</td>
<td>MFHI</td>
</tr>
<tr>
<td>3 011</td>
<td>MULT</td>
</tr>
<tr>
<td>4 100</td>
<td>ADD</td>
</tr>
<tr>
<td>5 101</td>
<td>MFSA µ</td>
</tr>
<tr>
<td>6 110</td>
<td>TGE</td>
</tr>
<tr>
<td>7 111</td>
<td>DSLL</td>
</tr>
</tbody>
</table>
Appendix A  CPU Instruction Set Details

<table>
<thead>
<tr>
<th>31 26 20 16 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode =</td>
</tr>
<tr>
<td>REGIMM rt</td>
</tr>
</tbody>
</table>

**rt**

Instructions encoded by rt field when OpCode field = REGIMM

<table>
<thead>
<tr>
<th>bits 20..19</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>BLTZ</td>
<td>BGEZ</td>
<td>BLTZL</td>
<td>BGEZL</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>0 01</td>
<td>TGEI</td>
<td>TGEIU</td>
<td>TLTI</td>
<td>TLTIU</td>
<td>TEQI</td>
<td>*</td>
<td>TNEI</td>
<td>*</td>
</tr>
<tr>
<td>2 10</td>
<td>BLTZAL</td>
<td>BGEZAL</td>
<td>BLTZALL</td>
<td>BGEZALL</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>3 11</td>
<td>MTSABµ</td>
<td>MTSAHµ</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

* This OpCode is reserved for future use. An attempt to execute it causes a Reserved Instruction exception.

η This OpCode is reserved for one of the following instructions which are currently not supported: DMULT, DMULTU, DDIV, DDIVU, LL, LLD, SC, SCD, LWC2, SWC2. An attempt to execute it causes a Reserved Instruction exception.

δ This OpCode indicates an instruction class. The instruction word must be further decoded by examining additional tables that show the values for another instruction field.

µ This OpCode indicates C790 specific instructions. It is included in the table because it uses a primary OpCode in the instruction encoding map.

α This OpCode is a coprocessor operation, not a CPU operation. If the processor state does not allow access to the specified coprocessor, the instruction causes a Coprocessor Unusable exception. It is included in the table because it uses a primary OpCode in the instruction encoding map.

λ This OpCode indicates the class of Coprocessor 0 (System Control Processor) instructions. If the processor state does not allow access to the coprocessor 0, the instruction causes a Coprocessor Unusable exception. Further encoding information for this instruction class is in the COP0 Instruction Encoding tables.

π This OpCode indicates the class of Coprocessor 1 (Floating-Point Processor) instructions. If the processor state does not allow access to the coprocessor 1, the instruction causes a Coprocessor Unusable exception. Further encoding information for this instruction class is in the COP1 Instruction Encoding tables.
B. C790-Specific Instruction Set Details

This appendix provides a detailed description of the operation of each C790-specific instruction. The C790's instruction set is extended from the original MIPS ISA in order to support embedded applications. There are three classes of C790-specific instructions:

- Three-operand Multiply and Multiply-Add instructions
- Multiply and Multiply-Add instructions for pipeline 1
- Multimedia instructions
B.1 Conventions Used in This Chapter

The HI and LO registers are 128 bits wide. Some instructions operate on either the lower or the upper doublewords of these registers, and there are also instructions which operate on the complete registers.

The following terminology is used for these registers.

- Strictly speaking, a reference to the least-significant doubleword of the HI and LO register should use the names HI0 and LO0. However, to be consistent with existing MIPS terminology, these registers are just called HI and LO.
- Reference to the upper doublewords of the HI and LO registers is made by using the names HI1 and LO1.
- Occasionally, based on context, the complete 128-bit registers are referred to as HI and LO.
- Any portion of these registers can use the names HI and LO with the appropriate bit width specifications. Thus HI1 can be referred to as HI127..64 and LO1 can be referred to as LO127..64, etc.

B.1.1 Instruction Description Notation and Functions

The Operation sections of the instruction descriptions describe the operation performed by each instruction using a high-level language notation, or pseudocode. Symbols, functions, and structures used in the Operation sections are described here.

B.1.2 Pseudocode Language Statement Execution

Each of the high-level language statements in an operation description is executed in sequential order (as modified by conditional and loop constructs).

B.1.3 Pseudocode Symbols

Special symbols used in the notation are described in Appendix A.

B.2 Definitions for Pseudocode Functions Used in Operation Descriptions

A variety of functions are used in the pseudocode descriptions to make the pseudocode more readable and also to abstract implementation-specific behavior. These functions are defined in Appendix A.
B.3 Summary of C790-Specific Instructions

B.3.1 Multiply and Multiply-Add Instructions

- Three-Operand Multiply and Multiply-Add (4 instructions)
  - MADD Multiply/Add
  - MADDU Multiply/Add Unsigned
  - MULT Multiply (3-operand)
  - MULTU Multiply Unsigned (3-operand)

- Multiply Instructions for Pipeline 1 (10 instructions)
  - MULT1 Multiply Pipeline 1
  - MULTU1 Multiply Unsigned Pipeline 1
  - DIV1 Divide Pipeline 1
  - DIVU1 Divide Unsigned Pipeline 1
  - MADD1 Multiply-Add Pipeline 1
  - MADDU1 Multiply-Add Unsigned Pipeline 1
  - MFHI1 Move From HI1 Register
  - MFLO1 Move From LO1 Register
  - MTHI1 Move To HI1 Register
  - MTLO1 Move To LO1 Register

B.3.2 Multimedia Instructions

- Arithmetic (19 instructions)
  - PADDB Parallel Add Byte
  - PSUBB Parallel Subtract Byte
  - PADDH Parallel Add Halfword
  - PSUBH Parallel Subtract Halfword
  - PADDW Parallel Add Word
  - PSUBW Parallel Subtract Word
  - PADSBH Parallel Add/Subtract Halfword
  - PADDSSB Parallel Add with Signed Saturation Byte
  - PSUBSSB Parallel Subtract with Signed Saturation Byte
  - PADDSSH Parallel Add with Signed Saturation Halfword
  - PSUBSSH Parallel Subtract with Signed Saturation Halfword
  - PADDSSW Parallel Add with Signed Saturation Word
  - PSUBSSW Parallel Subtract with Signed Saturation Word
  - PADDUB Parallel Add with Unsigned saturation Byte
  - PSUBUB Parallel Subtract with Unsigned saturation Byte
  - PADDUH Parallel Add with Unsigned saturation Halfword
  - PSUBUH Parallel Subtract with Unsigned saturation Halfword
  - PADDUW Parallel Add with Unsigned saturation Word
  - PSUBUW Parallel Subtract with Unsigned saturation Word
• Min/Max (4 instructions)
  PMAXH Parallel Maximum Halfword
  PMINH Parallel Minimum Halfword
  PMAXW Parallel Maximum Word
  PMINW Parallel Minimum Word

• Absolute (2 instructions)
  PABSH Parallel Absolute Halfword
  PABSW Parallel Absolute Word

• Logical (4 instructions)
  PAND Parallel AND
  POR Parallel OR
  PXOR Parallel XOR
  PNOR Parallel NOR

• Shift (9 instructions)
  PSLLH Parallel Shift Left Logical Halfword
  PSRLH Parallel Shift Right Logical Halfword
  PSRAH Parallel Shift Right Arithmetic Halfword
  PSLLW Parallel Shift Left Logical Word
  PSRLW Parallel Shift Right Logical Word
  PSRAW Parallel Shift Right Arithmetic Word
  PSLLVW Parallel Shift Left Logical Variable Word
  PSRLVW Parallel Shift Right Logical Variable Word
  PSRAVW Parallel Shift Right Arithmetic Variable Word

• Compare (6 instructions)
  PCGTB Parallel Compare for Greater Than Byte
  PCEQB Parallel Compare for Equal Byte
  PCGTH Parallel Compare for Greater Than Halfword
  PCEQH Parallel Compare for Equal Halfword
  PCGTW Parallel Compare for Greater Than Word
  PCEQW Parallel Compare for Equal Word

• LZC (1 instruction)
  PLZCW Parallel Leading Zero or One Count Word

• Quadword Load and Store (2 instructions)
  LQ Load Quadword
  SQ Store Quadword
Appendix B  C790-Specific Instruction Set Details

- **Multiply and Divide (19 instructions)**
  - PMULTW Parallel Multiply Word
  - PMULTUW Parallel Multiply Unsigned Word
  - PDIVW Parallel Divide Word
  - PDIVUW Parallel Divide Unsigned Word
  - PMADDW Parallel Multiply-Add Word
  - PMADDUW Parallel Multiply-Add Unsigned Word
  - PMSUBW Parallel Multiply-Subtract Word
  - PMULTH Parallel Multiply Halfword
  - PMADDH Parallel Multiply-Add Halfword
  - PMSUBH Parallel Multiply-Subtract Halfword
  - PHMADH Parallel Horizontal Multiply-Add Halfword
  - PHMSBH Parallel Horizontal Multiply-Subtract Halfword
  - PDIVBW Parallel Divide Broadcast Word
  - PMFHI Parallel Move From HI Register
  - PMFLO Parallel Move From LO Register
  - PMTHI Parallel Move To HI Register
  - PMTLO Parallel Move To LO Register
  - PMFHL Parallel Move From HI/LO Register
  - PMTHL Parallel Move To HI/LO Register

- **Pack/Extend (11 instructions)**
  - PPAC5 Parallel Pack to 5 bits
  - PPACB Parallel Pack to Byte
  - PPACH Parallel Pack to Halfword
  - PPACW Parallel Pack to Word
  - PEX5 Parallel Extend Upper from 5 bits
  - PEXTUB Parallel Extend Upper from Byte
  - PEXTLB Parallel Extend Lower from Byte
  - PEXTUH Parallel Extend Upper from Halfword
  - PEXTLH Parallel Extend Lower from Halfword
  - PEXTUW Parallel Extend Upper from Word
  - PEXTLW Parallel Extend Lower from Word

- **Others (16 instructions)**
  - PCPYH Parallel Copy Halfword
  - PCPYLD Parallel Copy Lower Doubleword
  - PCPYUD Parallel Copy Upper Doubleword
  - PREVH Parallel Reverse Halfword
  - PINTH Parallel Interleave Halfword
  - PINTEH Parallel Interleave Even Halfword
  - PEXEH Parallel Exchange Even Halfword
  - PEXCH Parallel Exchange Center Halfword
  - PEXEW Parallel Exchange Even Word
  - PEXCW Parallel Exchange Center Word
  - QFSRV Quadword Funnel Shift Right Variable
  - MFSA Move from Shift Amount Register
  - MTA Move to Shift Amount Register
  - MTSA Move Byte Count to Shift Amount Register
  - MTSAH Move Halfword Count to Shift Amount Register
  - PROT3W Parallel Rotate 3 Words
B.4 Instruction Set Details

In the following sections, details are provided for each of the C790-specific instructions.

Exceptions that may occur due to the execution of each instruction are listed after the description of each instruction. Descriptions of the immediate cause and manner of handling exceptions are omitted from the instruction descriptions in this appendix.
Appendix B  C790-Specific Instruction Set Details

DIV1
Divide Word Pipeline 1

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>rt</th>
<th>0</th>
<th>DIV1</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td></td>
<td>00000000000</td>
<td>011010</td>
<td></td>
</tr>
</tbody>
</table>

Format: DIV1 rs, rt

Purpose: To divide 32-bit signed integers using pipeline 1.

Description: (LO1, HI1) ← rs / rt

The 32-bit value in GPR rs is divided by the 32-bit value in GPR rt, treating both operands as signed values. The 32-bit quotient is placed into special register LO1 (= LO127..64) and the 32-bit remainder is placed into special register HI1 (= HI127..64).

No arithmetic exception occurs under any circumstances.

Restrictions:

If either GPR rt or GPR rs do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation will be undefined.

If the divisor in GPR rt is zero, the arithmetic result value will be undefined.

Operation:

if (NotWordValue(GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif

q ← GPR[rs]31..0 div GPR[rt]31..0
r ← GPR[rs]31..0 mod GPR[rt]31..0

LO127..64 ← (q 31)32 || q 31..0
HI127..64 ← (r 31)32 || r 31..0

Supplementary Explanation:

Normally, when 0x80000000 (-2147483648) the signed minimum value is divided by 0xFFFFFFFF (-1), the operation will result in an overflow. However, in this instruction an overflow exception doesn’t occur and the result will be as follows:

Quotient is 0x80000000 (-2147483648), and remainder is 0x00000000 (0).

This sign of the quotient and the remainder is based on the signs of the dividend and the divisor as shown in the table below:
Table B-1. Quotient and Remainder Signs

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Divisor</th>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive</td>
<td>Positive</td>
<td>Positive</td>
<td>Positive</td>
</tr>
<tr>
<td>Positive</td>
<td>Negative</td>
<td>Negative</td>
<td>Positive</td>
</tr>
<tr>
<td>Negative</td>
<td>Positive</td>
<td>Negative</td>
<td>Negative</td>
</tr>
<tr>
<td>Negative</td>
<td>Negative</td>
<td>Positive</td>
<td>Negative</td>
</tr>
</tbody>
</table>

Exceptions:

None

Programming Notes:

In C790, the integer divide operation proceeds asynchronously and allows other CPU instructions to execute before it is retired. An attempt to read LO1 or HI1 registers before the results are written will cause an interlock until the results are ready. Out-of-order execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the divide so that other instructions can execute in parallel.

No arithmetic exception occurs under any circumstances. Divide-by-zero or overflow conditions should be detected by instructions preceding the divide instruction. If the divide is asynchronous then the zero-divisor check can execute in parallel with the divide. The action taken on either divide-by-zero or overflow is either a convention within the program itself or more typically, the system software; one possibility is to take a BREAK exception with a code field value to signal the problem to the system software.

As an example, the C programming language in a UNIX environment expects division by zero to either terminate the program or execute a program-specified signal handler. C does not expect overflow to cause any exceptional condition. If the C compiler uses a divide instruction, it also emits code to test for a zero divisor and execute a BREAK instruction to inform the operating system if one is detected.
DIVU1  Divide Unsigned Word Pipeline 1

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>rt</th>
<th>0</th>
<th>DIVU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>0</td>
<td>0</td>
<td>0000000000</td>
<td>011011</td>
</tr>
</tbody>
</table>

Format:  
DIVU1 rs, rt

Purpose:  
To divide 32-bit unsigned integers using pipeline 1.

Description:  
(LO1, HI1) ← rs / rt

The 32-bit value in GPR rs is divided by the 32-bit value in GPR rt, treating both operands as unsigned values. The 32-bit quotient is placed into special register LO1 (= LO_{127..64}) and the 32-bit remainder is placed into special register HI1 (= HI_{127..64}).

No arithmetic exception occurs under any circumstances.

Restrictions:

If either GPR rt or GPR rs do not contain zero-extended 32-bit values (bits 63..32 equal zero), then the result of the operation is undefined.

If the divisor in GPR rt is zero, the arithmetic result will be undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif

q ← (0 || GPR[rs]_{31..0}) div (0 || GPR[rt]_{31..0})

r ← (0 || GPR[rs]_{31..0}) mod (0 || GPR[rt]_{31..0})

LO_{127..64} ← (q_{31})_{32} || q_{31..0}

HI_{127..64} ← (r_{31})_{32} || r_{31..0}

Exceptions:

None

Programming Notes:

See the Programming Notes for the DIV1 instruction.
### C790

<table>
<thead>
<tr>
<th>LQ</th>
<th>Load Quadword</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
</tr>
<tr>
<td>LQ</td>
<td>base</td>
</tr>
<tr>
<td>011110</td>
<td>6</td>
</tr>
</tbody>
</table>

**Format:** LQ rt, offset (base)

**Purpose:** To load a quadword from memory.

**Description:**

The contents of the 128-bit quadword at the memory location specified by the effective address are fetched and placed in the 128-bit GPR rt. The 16-bit signed offset is added to the contents of GPR base register to form the effective address. The least-significant four bits of the effective address are masked to zero (effectively creating an aligned address) before being used to access memory. No address exceptions due to alignment are possible.

**Restriction:**

The effective address doesn't have to be naturally aligned. The least significant 4 bits of the effective address are ignored.

**Operations:**

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend (offset)} + \text{GPR [base]}_{31..0} \\
\text{vAddr}_{3..0} & = 0^4 \\
(p\text{Addr, uncached}) & \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \\
\text{memquad} & \leftarrow \text{LoadMemory (uncached, QUADWORD, pAddr, vAddr, DATA)} \\
\text{GPR[rt]}_{127..0} & \leftarrow \text{memquad}
\end{align*}
\]

**Exceptions:**

- TLB Refill
- TLB Invalid
- Address Error
C790

Format:  
MADD rs, rt  
MADD rd, rs, rt

Purpose:  
To multiply 32-bit signed integers and add.

Description:  
\[(rd, HI, LO) \leftarrow (HI, LO) + rs \times rt\]

The 32-bit word value in GPR rt is multiplied by the 32-bit value in GPR rs, treating both operands as signed values, to produce a 64-bit multiply result. The 64-bit multiply result is added to the contents in special registers HI and LO. The low-order 32-bit word of the result is placed into special register LO and GPR rd, and the high-order 32-bit word of the result is placed into special register HI.

No arithmetic exception occurs under any circumstances.

If GPR rd is omitted in assembly language, 0 is used as the default value.

Restrictions:  
If either GPR rt or GPR rs do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation will be undefined.

Operation:  
if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
prod \(\leftarrow (HI_{31.0} || LO_{31.0}) + GPR[rs]_{31.0} * GPR[rt]_{31.0}\)
LO_{63.0} \(\leftarrow (prod_{31})^{32} || prod_{31}\)
HI_{63.0} \(\leftarrow (prod_{63})^{32} || prod_{63.32}\)
GPR[rd]_{63.0} \(\leftarrow (prod_{31})^{32} || prod_{31.0}\)

Exceptions:  
None

Programming Notes:  
In C790, the integer multiply accumulate operation proceeds asynchronously and allows other CPU instructions to execute before it is retired. An attempt to read LO or HI registers before the results are written will cause an interlock until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.
### MADD1

#### Multiply-Add word Pipeline 1

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>MADD1</td>
<td>100000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Format:
MADD1 rs, rt
MADD1 rd, rs, rt

#### Purpose:
To multiply 32-bit signed integers and add in Pipeline 1.

#### Description:

\[
(\text{rd}, \text{HI1}, \text{LO1}) \leftarrow (\text{HI1}, \text{LO1}) + \text{rs} \times \text{rt}
\]

The 32-bit word value in GPR rt is multiplied by the 32-bit value in GPR rs, treating both operands as signed values, to produce a 64-bit multiply result. The 64-bit multiply result is added to the contents in special registers HI1 (= HI127..64) and LO1 (= LO127..64). The low-order 32-bit word of the result is placed into special register LO1 and GPR rd, and the high-order 32-bit word of the result is placed into special register HI1.

No arithmetic exception occurs under any circumstances.

If GPR rd is omitted in assembly language, 0 is used as the default value.

#### Restrictions:

If either GPR rt or GPR rs do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation will be undefined.

#### Operation:

\[
\begin{align*}
\text{prod} & \leftarrow (\text{HI95..64} \ || \ \text{LO95..64}) + \text{GPR[rs]31..0} \times \text{GPR[rt]31..0} \\
\text{LO127..64} & \leftarrow (\text{prod31})^{32} \ || \ \text{prod}_{31..0} \\
\text{HI127..64} & \leftarrow (\text{prod63})^{32} \ || \ \text{prod}_{63..32} \\
\text{GPR[rd]}^{63..0} & \leftarrow (\text{prod31})^{32} \ || \ \text{prod}_{31..0}
\end{align*}
\]

#### Exceptions:
None

#### Programming Notes:

In the C790, the integer multiply accumulate operation proceeds asynchronously and allows other CPU instructions to execute before it is retired. An attempt to read LO1 or HI1 registers before the results are written will cause an interlock until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.
**MADDU**

**Multiply-Add Unsigned word**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>MADDU</td>
<td>000001</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

MADDU rs, rt

MADDU rd, rs, rt

**Purpose:**

To multiply 32-bit unsigned integers and add.

**Description:**

\[(rd, HI, LO) \leftarrow (HI, LO) + rs \times rt\]

The 32-bit word value in GPR rt is multiplied by the 32-bit value in GPR rs, treating both operands as unsigned values, to produce a 64-bit multiply result. The 64-bit multiply result is added to the contents in special registers HI and LO. The low-order 32-bit word of the result is placed into special register LO and GPR rd, and the high-order 32-bit word of the result is placed into special register HI.

No arithmetic exception occurs under any circumstances.

If GPR rd is omitted in assembly language, 0 is used as the default value.

**Restrictions:**

If either GPR rt or GPR rs do not contain zero-extended 32-bit values (bits 63..32 equal zero), then the result of the operation will be undefined.

**Operation:**

\[
\text{if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif}
\]

\[
\text{prod} \leftarrow (\text{HI}_31..0 || \text{LO}_31..0) + (0 || \text{GPR[rs]}_{31..0}) \times (0 || \text{GPR[rt]}_{31..0})
\]

\[
\text{LO}_{63..0} \leftarrow (\text{prod}_{31})^{32} || \text{prod}_{31..0}
\]

\[
\text{HI}_{63..0} \leftarrow (\text{prod}_{63})^{32} || \text{prod}_{63..32}
\]

\[
\text{GPR[rd]}_{63..0} \leftarrow (\text{prod}_{31})^{32} || \text{prod}_{31..0}
\]

**Exceptions:**

None

**Programming Notes:**

See the Programming Notes for the MADD instruction
MADDU1 Multiply-Add Unsigned word Pipeline 1  

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>MADDU1</td>
<td>100001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: 
MADDU1 rs, rt  
MADDU1 rd, rs, rt  

Purpose: To multiply 32-bit unsigned integers and add in Pipeline 1.  

Description: 
(rd, HI1, LO1) ← (HI1, LO1) + rs × rt  
The 32-bit value in GPR rt is multiplied by the 32-bit value in GPR rs, treating both operands as unsigned values, to produce a 64-bit multiply result. The 64-bit multiply result is added to the contents in special registers HI1 (= HI127..64) and LO1 (= LO127..64). The low-order 32-bit word of the result is placed into special register LO1 and GPR rd, and the high-order 32-bit word of the result is placed into special register HI1.  

No arithmetic exception occurs under any circumstances.  

If GPR rd is omitted in assembly language, 0 is used as the default value.  

Restrictions:  
If either GPR rt or GPR rs do not contain zero-extended 32-bit values (bits 63..32 equal zero), then the result of the operation will be undefined.  

Operation:  
if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif  
prod ← (HI195..64 || LO195..64) + (0 || GPR[rs]31..0) * (0 || GPR[rt]31..0)  
LO127..64 ← (prod 31)32 || prod31..0  
HI127..64 ← (prod 63)32 || prod63..32  
GPR[rd]63..0 ← (prod 31)32 || prod31..0  

Exceptions:  
None  

Programming Notes:  
See the Programming Notes for the MADD1 instruction.
### MFHI1

**Move From HI1 Register**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>0</td>
<td>0000000000</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>MFHI1</td>
<td>010000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** MFHI1 rd

**Purpose:** To copy the special purpose register HI1 to a GPR.

**Description:**

\[
\text{rd} \leftarrow \text{HI1}
\]

The contents of special register \( HI1 (= HI_{127..64}) \) are loaded into GPR \( rd \).

**Restrictions:**

None

**Operation:**

\[
\text{GPR}[rd]_{63..0} \leftarrow \text{HI}_{127..64}
\]

**Exceptions:**

None
Format: MFLO1 rd

Purpose: To copy the special purpose LO1 register to a GPR.

Description:

\[
\text{rd} \leftarrow \text{LO1}
\]

The contents of special register \( LO1 (= LO_{127..64} ) \) are loaded into GPR \( rd \).

Restrictions:

None

Operation:

\[
\text{GPR}[\text{rd}]_{63..0} \leftarrow \text{LO}_{127..64}
\]

Exceptions:

None
Appendix B  C790-Specific Instruction Set Details

MFSA  Move from Shift Amount Register

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>0</td>
<td>0000000000</td>
<td>rd</td>
<td>0</td>
<td>000000</td>
<td>MFSA</td>
<td>101000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C790

Format: MFSA  rd

Purpose: To copy the shift amount register SA to a GPR.

Description: rd ← SA

The contents of SA, the special register storing the funnel shift amount, is loaded into GPR rd. Note that the shift amount is encoded in SA in an implementation-defined manner. Therefore, it is not meaningful for software to operate on the value returned in rd. The sole purpose of this instruction is to permit the shift amount to be saved during a context switch. The MTSA instruction should be used to restore the state of SA.

Restrictions:
None

Operation:
GPR[rd]_{63..0} ← SA

Exceptions:
None

Implementation Note:
This instruction executes only in pipeline 0.
**MTHI1**

**Move To HI1 Register**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>0</td>
<td>0000000000000000</td>
<td>MTHI1</td>
<td>010001</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6 5           15  6

**Format:** MTHI1 rs

**Purpose:** To copy a GPR to the special purpose register HI1.

**Description:**

\[
\text{HI1} \leftarrow \text{rs}
\]

The contents of GPR rs are loaded into special register HI1 (= HI127..64).

**Restrictions:**

None

**Operation:**

\[
\text{HI1}_{127..64} \leftarrow \text{GPR}[\text{rs}]_{63..0}
\]

**Exceptions:**

None

**Programming Notes:**

None
MTLO1
Move To LO1 Register

Format: MTLO1 rs
Purpose: To copy a GPR to the special purpose register LO1.
Description: LO1 ← rs
  The contents of GPR rs are loaded into special register LO1 (= LO127..64).
Restrictions: None
Operation:
  LO127..64 ← GPR[rs]63..0
Exceptions: None
Programming Notes: None
**MTSA**  
**Move to Shift Amount Register**

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>0</td>
<td>000 0000 0000 0000</td>
<td>MTSA</td>
<td>101001</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Format:**  
MTSA  rs

**Purpose:**  
To copy a GPR to the shift amount register SA.

**Description:**  
SA ← rs  
The contents of GPR rs are loaded into SA, the special register storing the funnel shift amount. Note that rs must contain a value that was originally generated by MFSA. If some other user-generated value is in rs, the shifting action performed by the funnel shifter is not defined; that is, MTSA cannot be used to by a program to set a new funnel shift amount. This is because the shift amount is encoded in SA in an implementation-defined manner. The sole purpose of this instruction is to permit the shift amount to be restored during a context switch.

**Restrictions:**

*Note that the three instructions statically preceding a MTSA instruction must not read or write the SA register; that is, they cannot be either of the instructions MFSA, QFSRV, or MTSAx.*

Use the MTSAB and MTSAH instructions to set a new funnel shift amount.

**Operation:**

SA ← GPR[rs]_{63..0}

**Exceptions:**

None

**Implementation Note:**

1. MTSA updates the SA register in the A Stage. To keep exception processing simple, this requires that the cycle prior to MTSA not read the SA register. Also, when single stepping, making sure that SA always contains the value of the SA write instruction, just single stepped, requires that the cycle after MTSA not write the SA register. Both these rules are enforced by the architectural requirement that the three instructions prior to MTSA not read SA.

2. The MTSA instruction executes only in pipeline 0.
MTSAB  Move Byte Count to Shift Amount Register  MTSAB

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>MTSAB</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td></td>
<td>11000</td>
<td></td>
</tr>
</tbody>
</table>

### Format:
MTSAB rs, immediate

### Purpose:
To copy a GPR to the shift amount register SA.

### Description:
$$SA \leftarrow (rs \oplus \text{immediate}) \times 8$$

The least-significant four bits of GPR rs are XORed with the least-significant four bits of the immediate value. The resulting four bits are interpreted as a byte shift amount and stored into SA, the special register storing the funnel shift amount.

### Restrictions:
The three instructions statically preceding a MTSAB instruction must not read the SA register; that is, they cannot be either of the instructions MFSA or QFSRV.

### Operation:
$$SA \leftarrow (\text{GPR}[rs]_{3..0} \oplus \text{immediate}_{3..0}) \times 8$$

### Exceptions:
None

### Implementation Note:
1. MTSAB updates the SA register in the A Stage. To keep exception processing simple, this requires that the cycle prior to MTSAB not read the SA register. Also, when single stepping, making sure that SA always contains the value of the SA write instruction, just single stepped, requires that the cycle after the MTSAB not write the SA register. Both these rules are enforced by the architectural requirement that the three instructions prior to MTSAB not read SA.

2. The MTSAB instruction executes only in pipeline 0.

### Programming Note:
MTSAB allows the user to load either a variable shift amount or a fixed shift amount, as follows:

- `mtsab 0, 5 // Set shift amount to “5 bytes”`
- `mtsab 10, 0 // Set byte shift amount to contents of GPR10`
**MTSAH**

Move Halfword Count to Shift Amount Register

---

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>MTSAH</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td></td>
<td>11001</td>
<td></td>
</tr>
</tbody>
</table>

### Format:

MTSAH rs, immediate

### Purpose:

To copy a GPR to the shift amount register SA.

### Description:

\[
SA \leftarrow (rs \text{ xor immediate}) \times 16
\]

The least-significant three bits of GPR rs are XORed with the least-significant three bits of the immediate value. The resulting three bits are interpreted as a halfword shift amount and stored into SA, the special register storing the funnel shift amount.

### Restrictions:

The three instructions statically preceding a MTSAB instruction must not read the SA register; that is, they cannot be either of the instructions MFSA or QFSRV.

### Operation:

\[
SA \leftarrow (\text{GPR}[rs]_{2..0} \text{ xor immediate}_{2..0}) \times 16
\]

### Exceptions:

None

### Implementation Note:

1. MTSAH updates the SA register in the A Stage. To keep exception processing simple, this requires that the cycle prior to MTSAH not read the SA register. Also, when single stepping, making sure that SA always contains the value of the SA write instruction, just single stepped, requires that the cycle after MTSAH not write the SA register. Both these rules are enforced by the architectural requirement that the three instructions prior to MTSAH not read SA.

2. The MTSAH instruction executes only in pipeline 0.

### Programming Note:

MTSAH allows the user to load either a variable shift amount or a fixed shift amount, as follows:

```assembly
mtsah 0, 5 // Set shift amount to “5 halfwords”
mtsah 10, 0 // Set halfword shift amount to value of GPR10
```
### MULT

**Multiply Word**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>MULT</td>
<td>011000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Format:

- MULT rd, rs, rt
- MULT rs, rt

#### Purpose:

To multiply 32-bit signed integers.

#### Description:

\[(rd, LO, HI) \leftarrow rs \times rt\]

The 32-bit value in GPR \( rt \) is multiplied by the 32-bit value in GPR \( rs \), treating both operands as signed values, to produce a 64-bit result. The low-order 32-bits of the result is placed into special register \( LO \) and GPR \( rd \), and the high-order 32-bit of the result is placed into special register \( HI \).

No arithmetic exception occurs under any circumstances.

If GPR \( rd \) is omitted in assembly language, 0 is used as the default value.

#### Restrictions:

- If either GPR \( rt \) or GPR \( rs \) do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation will be undefined.

#### Operation:

\[
\text{if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif}
\]

\[
\text{prod} \leftarrow \text{GPR[rs]}31..0 \times \text{GPR[rt]}31..0
\]

\[
\text{LO}63..0 \leftarrow (\text{prod}32 || \text{prod}31..0
\]

\[
\text{HI}63..0 \leftarrow (\text{prod}63 || \text{prod}63..32
\]

\[
\text{GPR[rd]}63..0 \leftarrow (\text{prod}31 || \text{prod}31..0
\]

#### Exceptions:

None

#### Programming Notes:

In the C790, the integer multiply operation allows other CPU instructions to execute out-of-order. An attempt to read \( LO \) or \( HI \) registers before the results are written will cause an interlock until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.
MULT1 Multiply Word Pipeline 1

\[
\begin{array}{ccccccc}
31 & 26 & 25 & 21 & 20 & 15 & 11 & 10 & 6 & 5 & 0 \\
\hline
\text{MMI} & & & & & & & & & & \text{MULT1} \\
011100 & rs & rt & rd & 0 & 00000 & \\
6 & 5 & 5 & 5 & 5 & 6 &
\end{array}
\]

Format: MULT1 rd, rs, rt
MULT1 rs, rt

Purpose: To multiply 32-bit signed integers in Pipeline 1.

Description:

\[(rd, HI1, LO1) \leftarrow rs \times rt\]

The 32-bit value in GPR rt is multiplied by the 32-bit value in GPR rs, treating both operands as signed values, to produce a 64-bit result. The low-order 32-bits of the result is placed into special register LO1 (= LO127..64) and GPR rd, and the high-order 32-bits of the result is placed into special register HI1 (= HI127..64).

No arithmetic exceptions occurs under any circumstances.

If GPR rd is omitted in assembly language, 0 is used as the default value.

Restrictions:

If either GPR rt or GPR rs do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation will be undefined.

Operation:

\[
\begin{align*}
\text{prod} & \leftarrow \text{GPR}[rs]_{31.0} \times \text{GPR}[rt]_{31.0} \\
\text{LO127..64} & \leftarrow (\text{prod}^{32} \parallel \text{prod}^{31.0}) \\
\text{HI127..64} & \leftarrow (\text{prod}^{63} \parallel \text{prod}^{63.32}) \\
\text{GPR}[rd]_{63.0} & \leftarrow (\text{prod}^{31} \parallel \text{prod}^{31.0})
\end{align*}
\]

Exceptions:

None

Programming Notes:

In the C790 the integer multiply operation allows other CPU instructions to execute out-of-order. An attempt to read LO1 or HI1 before the results are written will cause an interlock until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.
**Appendix B  C790-Specific Instruction Set Details**

**MULTU**  
Multiply Unsigned Word

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>000000</td>
<td>MULTU</td>
<td>011001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

MULTU rd, rs, rt  
MULTU rs, rt

**Purpose:**

To multiply 32-bit unsigned integers.

**Description:**

(rd, HI, LO) ← rs × rt

The 32-bit value in GPR rt is multiplied by the 32-bit value in GPR rs, treating both operands as unsigned values, to produce a 64-bit result. The low-order 32-bit of the result is placed into special register LO and GPR rd, and the high-order 32-bits of the result is placed into special register HI.

No arithmetic exception occurs under any circumstances.

If GPR rd is omitted in assembly language, 0 is used as the default value.

**Restrictions:**

If either GPR rt or GPR rs do not contain zero-extended 32-bit values (bits 63..32 equal zero), then the result of the operation will be undefined.

**Operation:**

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif

prod ← (0 || GPR[rs]31..0) * (0 || GPR[rt]31..0)

LO63..0 ← (prod 32) || prod31..0

HI 63..0 ← (prod 63) || prod63..32

GPR[rd] 63..0 ← (prod 31) || prod31..0

**Exceptions:**

None

**Programming Notes:**

See the Programming Notes for the MULT instruction.
MULTU1

Multiply Unsigned Word Pipeline 1

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>MULTU1</td>
<td>011001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:
MULTU1 rd, rs, rt
MULTU1 rs, rt

Purpose:
To multiply 32-bit unsigned integers in Pipeline 1.

Description:
(rd, HI1, LO1) ← rs × rt

The 32-bit value in GPR rt is multiplied by the 32-bit value in GPR rs, treating both operands as unsigned values, to produce a 64-bit result. The low-order 32-bit of the result is placed into special register LO1 (= LO127..64) and GPR rd, and the high-order 32-bit of the result is placed into special register HI1 (= HI127..64).

No arithmetic exceptions occurs under any circumstances.

If GPR rd is omitted in assembly language, 0 is used as the default value.

Restrictions:
If either GPR rt or GPR rs do not contain zero-extended 32-bit values (bits 63..32 equal zero), then the result of the operation will be undefined.

Operation:
if (NotWordValue (GPR[r]) or NotWordValue (GPR[r])) then UndefinedResult() endif
prod ← ( 0 || GPR[r][31..0]) * ( 0 || GPR[r][31..0])
LO127..64 ← (prod 31)32 || prod 31..0
HI127..64 ← (prod 63)32 || prod 63..32
GPR[rd][63..0] ← (prod 31)32 || prod 31..0

Exceptions:
None

Programming Notes:
See the Programming Notes for the MULT1 instruction.
**PABSH**  Parallel Absolute Halfword  **PABSH**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>0</td>
<td>00000</td>
<td>rt</td>
<td>rd</td>
<td>PABSH</td>
<td>00101</td>
<td>MMI1</td>
<td>101000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**Format:**  
PABSH rd, rt

**Purpose:**  To calculate the absolute value of 8 16-bit integers in parallel.

**Description:**  
rd ← | rt |

The absolute value of the eight signed halfword values in GPR rt are placed into the corresponding eight halfwords in GPR rd.  
This instruction operates on 128-bit registers.

**Operation:**  
GPR[rd]₁₅.₀ ← GPR[rt]₁₅.₀  
GPR[rd]₃₁.₁₆ ← GPR[rt]₃₁.₁₆  
GPR[rd]₄₇.₃₂ ← GPR[rt]₄₇.₃₂  
GPR[rd]₆₃.₄₈ ← GPR[rt]₆₃.₄₈  
GPR[rd]₇₉.₆₄ ← GPR[rt]₇₉.₆₄  
GPR[rd]₉₅.₈₀ ← GPR[rt]₉₅.₈₀  
GPR[rd]₁₁₁.₉₆ ← GPR[rt]₁₁₁.₉₆  
GPR[rd]₁₂₇.₁₂₂ ← GPR[rt]₁₂₇.₁₂₂

**Supplementary explanation:**  
When the halfword value in GPR rt is 0x8000 (-32768), the smallest negative value, the operation will result in an overflow. However, overflow exception doesn't occur; the result is truncated to the largest positive number - 0x7FFF (+32767).

**Exceptions:**  
None
PABSW
Parallel Absolute Word

Format: PABSW rd, rt

Purpose: To calculate the absolute value of 4 32-bit integers in parallel.

Description: rd ← |rt|

The absolute value of the four signed word values in GPR rt are placed into the corresponding four words in GPR rd.

This instruction operates on 128-bit registers.

Operation:

GPR[rd]_{31..0} ← |GPR[rt]_{31..0}|
GPR[rd]_{63..32} ← |GPR[rt]_{63..32}|
GPR[rd]_{95..64} ← |GPR[rt]_{95..64}|
GPR[rd]_{127..96} ← |GPR[rt]_{127..96}|

Supplementary explanation:

When the word value of the GPR rt is equal to 0x80000000 (-2147483648), the smallest negative number, the operation will result in an overflow. However, if an overflow exception doesn't occur; the result is truncated to the largest positive value - 0x7FFFFFFF (+2147483647).

Exceptions:

None
## PADDB

**Parallel Add Byte**

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>PADDB</th>
<th>MMI0</th>
<th>001000</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

### Format:

PADDB rd, rs, rt

### Purpose:
To add 16 pairs of 8-bit integers in parallel.

### Description:

\[ rd \leftarrow rs + rt \]

The sixteen byte values in GPR \( rs \) are added to the corresponding sixteen byte values in GPR \( rt \) in parallel. The results are placed into the corresponding sixteen bytes in GPR \( rd \).

No overflow or underflow exceptions are generated under any circumstances. This instruction operates on 128-bit registers.

### Operation:

\[
\begin{align*}
GPR[rd]_{7..0} & \leftarrow (GPR[rs]_{7..0} + GPR[rt]_{7..0})_{7..0} \\
GPR[rd]_{15..8} & \leftarrow (GPR[rs]_{15..8} + GPR[rt]_{15..8})_{7..0} \\
GPR[rd]_{23..16} & \leftarrow (GPR[rs]_{23..16} + GPR[rt]_{23..16})_{7..0} \\
GPR[rd]_{31..24} & \leftarrow (GPR[rs]_{31..24} + GPR[rt]_{31..24})_{7..0} \\
GPR[rd]_{39..32} & \leftarrow (GPR[rs]_{39..32} + GPR[rt]_{39..32})_{7..0} \\
GPR[rd]_{47..40} & \leftarrow (GPR[rs]_{47..40} + GPR[rt]_{47..40})_{7..0} \\
GPR[rd]_{55..48} & \leftarrow (GPR[rs]_{55..48} + GPR[rt]_{55..48})_{7..0} \\
GPR[rd]_{63..56} & \leftarrow (GPR[rs]_{63..56} + GPR[rt]_{63..56})_{7..0} \\
GPR[rd]_{71..64} & \leftarrow (GPR[rs]_{71..64} + GPR[rt]_{71..64})_{7..0} \\
GPR[rd]_{79..72} & \leftarrow (GPR[rs]_{79..72} + GPR[rt]_{79..72})_{7..0} \\
GPR[rd]_{87..80} & \leftarrow (GPR[rs]_{87..80} + GPR[rt]_{87..80})_{7..0} \\
GPR[rd]_{95..88} & \leftarrow (GPR[rs]_{95..88} + GPR[rt]_{95..88})_{7..0} \\
GPR[rd]_{103..96} & \leftarrow (GPR[rs]_{103..96} + GPR[rt]_{103..96})_{7..0} \\
GPR[rd]_{111..104} & \leftarrow (GPR[rs]_{111..104} + GPR[rt]_{111..104})_{7..0} \\
GPR[rd]_{119..112} & \leftarrow (GPR[rs]_{119..112} + GPR[rt]_{119..112})_{7..0} \\
GPR[rd]_{127..120} & \leftarrow (GPR[rs]_{127..120} + GPR[rt]_{127..120})_{7..0}
\end{align*}
\]

### Exceptions:
None
PADDH Parallel Add Halfword

**Format:**

PADDH rd, rs, rt

**Purpose:**

To add 8 pairs of 16-bit integers in parallel.

**Description:**

\[ \text{rd} \leftarrow \text{rs} + \text{rt} \]

The eight halfword values in GPR rs are added to the corresponding eight halfword values in GPR rt in parallel. The results are placed into the corresponding eight halfwords in GPR rd.

No overflow or underflow exceptions are generated under any circumstances.

This instruction operates on 128-bit registers.

**Operation:**

\[
\begin{align*}
\text{GPR}[\text{rd}]_{15.0} & \leftarrow (\text{GPR}[\text{rs}]_{15.0} + \text{GPR}[\text{rt}]_{15.0})_{15.0} \\
\text{GPR}[\text{rd}]_{31.16} & \leftarrow (\text{GPR}[\text{rs}]_{31.16} + \text{GPR}[\text{rt}]_{31.16})_{15.0} \\
\text{GPR}[\text{rd}]_{47.32} & \leftarrow (\text{GPR}[\text{rs}]_{47.32} + \text{GPR}[\text{rt}]_{47.32})_{15.0} \\
\text{GPR}[\text{rd}]_{63.48} & \leftarrow (\text{GPR}[\text{rs}]_{63.48} + \text{GPR}[\text{rt}]_{63.48})_{15.0} \\
\text{GPR}[\text{rd}]_{79.64} & \leftarrow (\text{GPR}[\text{rs}]_{79.64} + \text{GPR}[\text{rt}]_{79.64})_{15.0} \\
\text{GPR}[\text{rd}]_{95.80} & \leftarrow (\text{GPR}[\text{rs}]_{95.80} + \text{GPR}[\text{rt}]_{95.80})_{15.0} \\
\text{GPR}[\text{rd}]_{111.96} & \leftarrow (\text{GPR}[\text{rs}]_{111.96} + \text{GPR}[\text{rt}]_{111.96})_{15.0} \\
\text{GPR}[\text{rd}]_{127.112} & \leftarrow (\text{GPR}[\text{rs}]_{127.112} + \text{GPR}[\text{rt}]_{127.112})_{15.0}
\end{align*}
\]

**Exceptions:**

None
**PADDSB**  Parallel Add with Signed saturation Byte  **PADDSB**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PADDSB</td>
<td>11000</td>
<td>MMI0</td>
<td>001000</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Format:**  
PADDSB rd, rs, rt

**Purpose:**  
To add 16 pairs of 8-bit signed integers with saturation in parallel.

**Description:**  
rd ← rs + rt

The sixteen signed byte values in GPR rs are added to the corresponding sixteen signed byte values in GPR rt in parallel. The results are placed into the corresponding sixteen bytes in GPR rd.

No overflow or underflow exceptions are generated under any circumstances. Results beyond the range of a signed byte value are saturated according to the following:

- **Overflow:** 0x7F
- **Underflow:** 0x80

This instruction operates on 128-bit registers.

**Operation:**

```plaintext```
if ((GPR[rs]7..0 + GPR[rt]7..0) > 0x7F) then
    GPR[rd]7..0 ← 0x7F
else if (0x100 <= (GPR[rs]7..0 + GPR[rt]7..0) < 0x180) then
    GPR[rd]7..0 ← 0x80
else
    GPR[rd]7..0 ← (GPR[rs]7..0 + GPR[rt]7..0)7..0
endif

if ((GPR[rs]15..8 + GPR[rt]15..8) > 0x7F) then
    GPR[rd]15..8 ← 0x7F
else if (0x100 <= (GPR[rs]15..8 + GPR[rt]15..8) < 0x180) then
    GPR[rd]15..8 ← 0x80
else
    GPR[rd]15..8 ← (GPR[rs]15..8 + GPR[rt]15..8)7..0
endif

if ((GPR[rs]23..16 + GPR[rt]23..16) > 0x7F) then
    GPR[rd]23..16 ← 0x7F
else if (0x100 <= (GPR[rs]23..16 + GPR[rt]23..16) < 0x180) then
    GPR[rd]23..16 ← 0x80
else
    GPR[rd]23..16 ← (GPR[rs]23..16 + GPR[rt]23..16)7..0
endif

if ((GPR[rs]31..24 + GPR[rt]31..24) > 0x7F) then
    GPR[rd]31..24 ← 0x7F
else if (0x100 <= (GPR[rs]31..24 + GPR[rt]31..24) < 0x180) then
```

---

C790
GPR[rd]31..24 ← 0x80 else
GPR[rd]31..24 ← (GPR[rs]31..24 + GPR[rt]31..24)7..0 endif

if ((GPR[rs]39..32 + GPR[rt]39..32) >= 0x7F) then
GPR[rd]39..32 ← 0x7F else if (0x100 <= (GPR[rs]39..32 + GPR[rt]39..32) < 0x180) then
GPR[rd]39..32 ← 0x80 else
GPR[rd]39..32 ← (GPR[rs]39..32 + GPR[rt]39..32)7..0 endif

if ((GPR[rs]47..40 + GPR[rt]47..40) >= 0x7F) then
GPR[rd]47..40 ← 0x7F else if (0x100 <= (GPR[rs]47..40 + GPR[rt]47..40) < 0x180) then
GPR[rd]47..40 ← 0x80 else
GPR[rd]47..40 ← (GPR[rs]47..40 + GPR[rt]47..40)7..0 endif

if ((GPR[rs]55..48 + GPR[rt]55..48) >= 0x7F) then
GPR[rd]55..48 ← 0x7F else if (0x100 <= (GPR[rs]55..48 + GPR[rt]55..48) < 0x180) then
GPR[rd]55..48 ← 0x80 else
GPR[rd]55..48 ← (GPR[rs]55..48 + GPR[rt]55..48)7..0 endif

if ((GPR[rs]63..56 + GPR[rt]63..56) >= 0x7F) then
GPR[rd]63..56 ← 0x7F else if (0x100 <= (GPR[rs]63..56 + GPR[rt]63..56) < 0x180) then
GPR[rd]63..56 ← 0x80 else
GPR[rd]63..56 ← (GPR[rs]63..56 + GPR[rt]63..56)7..0 endif

if ((GPR[rs]71..64 + GPR[rt]71..64) >= 0x7F) then
GPR[rd]71..64 ← 0x7F else if (0x100 <= (GPR[rs]71..64 + GPR[rt]71..64) < 0x180) then
GPR[rd]71..64 ← 0x80 else
GPR[rd]71..64 ← (GPR[rs]71..64 + GPR[rt]71..64)7..0 endif

if ((GPR[rs]79..72 + GPR[rt]79..72) >= 0x7F) then
GPR[rd]79..72 ← 0x7F else if (0x100 <= (GPR[rs]79..72 + GPR[rt]79..72) < 0x180) then
GPR[rd]79..72 ← 0x80 else
GPR[rd]79..72 ← (GPR[rs]79..72 + GPR[rt]79..72)7..0 endif

if ((GPR[rs]87..80 + GPR[rt]87..80) >= 0x7F) then
GPR[rd]87..80 ← 0x7F
else if (0x100 <= (GPR[rs]87..80 + GPR[rt]87..80) < 0x180) then
  GPR[rd]87..80 ← 0x80
else
  GPR[rd]87..80 ← (GPR[rs]87..80 + GPR[rt]87..80)7..0
endif

if ((GPR[rs]95..88 + GPR[rt]95..88) > 0x7F) then
  GPR[rd]95..88 ← 0x7F
else if (0x100 <= (GPR[rs]95..88 + GPR[rt]95..88) < 0x180) then
  GPR[rd]95..88 ← 0x80
else
  GPR[rd]95..88 ← (GPR[rs]95..88 + GPR[rt]95..88)7..0
endif

if ((GPR[rs]103..96 + GPR[rt]103..96) > 0x7F) then
  GPR[rd]103..96 ← 0x7F
else if (0x100 <= (GPR[rs]103..96 + GPR[rt]103..96) < 0x180) then
  GPR[rd]103..96 ← 0x80
else
  GPR[rd]103..96 ← (GPR[rs]103..96 + GPR[rt]103..96)7..0
endif

if ((GPR[rs]111..104 + GPR[rt]111..104) > 0x7F) then
  GPR[rd]111..104 ← 0x7F
else if (0x100 <= (GPR[rs]111..104 + GPR[rt]111..104) < 0x180) then
  GPR[rd]111..104 ← 0x80
else
  GPR[rd]111..104 ← (GPR[rs]111..104 + GPR[rt]111..104)7..0
endif

if ((GPR[rs]119..112 + GPR[rt]119..112) > 0x7F) then
  GPR[rd]119..112 ← 0x7F
else if (0x100 <= (GPR[rs]119..112 + GPR[rt]119..112) < 0x180) then
  GPR[rd]119..112 ← 0x80
else
  GPR[rd]119..112 ← (GPR[rs]119..112 + GPR[rt]119..112)7..0
endif

if ((GPR[rs]127..120 + GPR[rt]127..120) > 0x7F) then
  GPR[rd]127..120 ← 0x7F
else if (0x100 <= (GPR[rs]127..120 + GPR[rt]127..120) < 0x180) then
  GPR[rd]127..120 ← 0x80
else
  GPR[rd]127..120 ← (GPR[rs]127..120 + GPR[rt]127..120)7..0
endif
### Appendix B  C790-Specific Instruction Set Details

#### Exceptions:
None

---

<table>
<thead>
<tr>
<th>rs</th>
<th>A15</th>
<th>A14</th>
<th>A13</th>
<th>A12</th>
<th>A11</th>
<th>A10</th>
<th>A9</th>
<th>A8</th>
<th>A7</th>
<th>A6</th>
<th>A5</th>
<th>A4</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt</td>
<td>B15</td>
<td>B14</td>
<td>B13</td>
<td>B12</td>
<td>B11</td>
<td>B10</td>
<td>B9</td>
<td>B8</td>
<td>B7</td>
<td>B6</td>
<td>B5</td>
<td>B4</td>
<td>B3</td>
<td>B2</td>
<td>B1</td>
<td>B0</td>
</tr>
</tbody>
</table>

* Saturate to signed byte
### PADDSH
Parallel Add with Signed saturation Halfword

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PADDSH</td>
<td>MMIO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>001000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**
PADDSH rd, rs, rt

**Purpose:**
To add 8 pairs of 16-bit signed integers with saturation in parallel.

**Description:**
rd ← rs + rt

The eight signed halfword values in GPR rs are added to the corresponding eight signed halfword values in GPR rt in parallel. The results are placed into the corresponding eight halfwords in GPR rd.

No overflow or underflow exceptions are generated under any circumstances. Results beyond the range of a signed halfword value are saturated according to the following:

- **Overflow:** 0x7FFF
- **Underflow:** 0x8000

This instruction operates on 128-bit registers.

**Operation:**

```plaintext
if ((GPR[rs]15..0 + GPR[rt]15..0) > 0x7FFF) then
  GPR[rd]15..0 ← 0x7FFF
else if (0x10000 <= (GPR[rs]15..0 + GPR[rt]15..0) < 0x18000) then
  GPR[rd]15..0 ← 0x8000
else
  GPR[rd]15..0 ← (GPR[rs]15..0 + GPR[rt]15..0)15..0
endif

if ((GPR[rs]31..16 + GPR[rt]31..16) > 0x7FFF) then
  GPR[rd]31..16 ← 0x7FFF
else if (0x10000 <= (GPR[rs]31..16 + GPR[rt]31..16) < 0x18000) then
  GPR[rd]31..16 ← 0x8000
else
  GPR[rd]31..16 ← (GPR[rs]31..16 + GPR[rt]31..16)15..0
endif

if ((GPR[rs]47..32 + GPR[rt]47..32) > 0x7FFF) then
  GPR[rd]47..32 ← 0x7FFF
else if (0x10000 <= (GPR[rs]47..32 + GPR[rt]47..32) < 0x18000) then
  GPR[rd]47..32 ← 0x8000
else
  GPR[rd]47..32 ← (GPR[rs]47..32 + GPR[rt]47..32)15..0
endif
```
if ((GPR[rs]_63..48 + GPR[rt]_63..48) > 0x7FFF) then
    GPR[rd]_63..48 ← 0x7FFF
else if (0x10000 ≤ (GPR[rs]_63..48 + GPR[rt]_63..48) < 0x18000) then
    GPR[rd]_63..48 ← 0x8000
else
    GPR[rd]_63..48 ← (GPR[rs]_63..48 + GPR[rt]_63..48)_{15..0}
endif

if ((GPR[rs]_79..64 + GPR[rt]_79..64) > 0x7FFF) then
    GPR[rd]_79..64 ← 0x7FFF
else if (0x10000 ≤ (GPR[rs]_79..64 + GPR[rt]_79..64) < 0x18000) then
    GPR[rd]_79..64 ← 0x8000
else
    GPR[rd]_79..64 ← (GPR[rs]_79..64 + GPR[rt]_79..64)_{15..0}
endif

if ((GPR[rs]_95..80 + GPR[rt]_95..80) > 0x7FFF) then
    GPR[rd]_95..80 ← 0x7FFF
else if (0x10000 ≤ (GPR[rs]_95..80 + GPR[rt]_95..80) < 0x18000) then
    GPR[rd]_95..80 ← 0x8000
else
    GPR[rd]_95..80 ← (GPR[rs]_95..80 + GPR[rt]_95..80)_{15..0}
endif

if ((GPR[rs]_111..96 + GPR[rt]_111..96) > 0x7FFF) then
    GPR[rd]_111..96 ← 0x7FFF
else if (0x10000 ≤ (GPR[rs]_111..96 + GPR[rt]_111..96) < 0x18000) then
    GPR[rd]_111..96 ← 0x8000
else
    GPR[rd]_111..96 ← (GPR[rs]_111..96 + GPR[rt]_111..96)_{15..0}
endif

if ((GPR[rs]_127..112 + GPR[rt]_127..112) > 0x7FFF) then
    GPR[rd]_127..112 ← 0x7FFF
else if (0x10000 ≤ (GPR[rs]_127..112 + GPR[rt]_127..112) < 0x18000) then
    GPR[rd]_127..112 ← 0x8000
else
    GPR[rd]_127..112 ← (GPR[rs]_127..112 + GPR[rt]_127..112)_{15..0}
endif

**Exceptions:**

None
PADDSW Parallel Add with Signed saturation Word

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI 011100</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PADDSW 10000</td>
<td>MMI0 001000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** PADDSW rd, rs, rt

**Purpose:** To add 4 pairs of 32-bit signed integers with saturation in parallel.

**Description:** rd ← rs + rt

The four signed word values in GPR rs are added to the corresponding four signed word values in GPR rt in parallel. The results are placed into the corresponding four words in GPR rd.

No overflow or underflow exceptions are generated under any circumstances. Results beyond the range of a signed word value are saturated according to the following:

- **Overflow:** 0x7FFFFFFF
- **Underflow:** 0x80000000

This instruction operates on 128-bit registers.

**Operation:**

```plaintext
if ((GPR[rs]31..0 + GPR[rt]31..0) > 0x7FFFFFFF) then
    GPR[rd]31..0 ← 0x7FFFFFFF
else if (0x100000000 <= (GPR[rs]31..0 + GPR[rt]31..0) < 0x180000000) then
    GPR[rd]31..0 ← 0x80000000
else
    GPR[rd]31..0 ← (GPR[rs]31..0 + GPR[rt]31..0)31..0
endif

if ((GPR[rs]63..32 + GPR[rt]63..32) > 0x7FFFFFFF) then
    GPR[rd]63..32 ← 0x7FFFFFFF
else if (0x100000000 <= (GPR[rs]63..32 + GPR[rt]63..32) < 0x180000000) then
    GPR[rd]63..32 ← 0x80000000
else
    GPR[rd]63..32 ← (GPR[rs]63..32 + GPR[rt]63..32)31..0
endif

if ((GPR[rs]95..64 + GPR[rt]95..64) > 0x7FFFFFFF) then
    GPR[rd]95..64 ← 0x7FFFFFFF
else if (0x100000000 <= (GPR[rs]95..64 + GPR[rt]95..64) < 0x180000000) then
    GPR[rd]95..64 ← 0x80000000
else
    GPR[rd]95..64 ← (GPR[rs]95..64 + GPR[rt]95..64)31..0
endif
```
if \((\text{GPR}[rs]_{127..96} + \text{GPR}[rt]_{127..96}) > 0x7FFFFFFF\) then
\[
\text{GPR}[rd]_{127..96} \leftarrow 0x7FFFFFFF
\]
else if \((0x100000000 \leq (\text{GPR}[rs]_{127..96} + \text{GPR}[rt]_{127..96}) < 0x180000000)\) then
\[
\text{GPR}[rd]_{127..96} \leftarrow 0x80000000
\]
else
\[
\text{GPR}[rd]_{127..96} \leftarrow (\text{GPR}[rs]_{127..96} + \text{GPR}[rt]_{127..96})_{31..0}
\]
endif

Saturate to signed word

Exceptions:
None
PADDUB — Parallel Add with Unsigned saturation Byte

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>PADDUB</th>
<th>MMI1</th>
</tr>
</thead>
<tbody>
<tr>
<td>01100</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>16</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Format: PADDUB rd, rs, rt

Purpose: To add 16 pairs of 8-bit unsigned integers with saturation in parallel.

Description: rd ← rs + rt

The sixteen unsigned byte values in GPR rs are added to the corresponding sixteen unsigned byte values in GPR rt in parallel. The results are placed into the corresponding sixteen bytes in GPR rd.

No overflow exceptions are generated under any circumstances. Results beyond the range of an unsigned byte value are saturated according to the following:

Overflow: 0xFF

This instruction operates on 128-bit registers.

Operation:

if ((GPR[rs]7..0 + GPR[rt]7..0) > 0xFF) then
    GPR[rd]7..0 ← 0xFF
else
    GPR[rd]7..0 ← (GPR[rs]7..0 + GPR[rt]7..0)7..0
endif

if ((GPR[rs]15..8 + GPR[rt]15..8) > 0xFF) then
    GPR[rd]15..8 ← 0xFF
else
    GPR[rd]15..8 ← (GPR[rs]15..8 + GPR[rt]15..8)7..0
endif

if ((GPR[rs]23..16 + GPR[rt]23..16) > 0xFF) then
    GPR[rd]23..16 ← 0xFF
else
    GPR[rd]23..16 ← (GPR[rs]23..16 + GPR[rt]23..16)7..0
endif

if ((GPR[rs]31..24 + GPR[rt]31..24) > 0xFF) then
    GPR[rd]31..24 ← 0xFF
else
    GPR[rd]31..24 ← (GPR[rs]31..24 + GPR[rt]31..24)7..0
endif

if ((GPR[rs]39..32 + GPR[rt]39..32) > 0xFF) then
    GPR[rd]39..32 ← 0xFF
else
    GPR[rd]39..32 ← (GPR[rs]39..32 + GPR[rt]39..32)7..0
endif
if ((GPR[rs]47..40 + GPR[rt]47..40) > 0xFF) then
    GPR[rd]47..40 ← 0xFF
else
    GPR[rd]47..40 ← (GPR[rs]47..40 + GPR[rt]47..40)7..0
endif

if ((GPR[rs]55..48 + GPR[rt]55..48) > 0xFF) then
    GPR[rd]55..48 ← 0xFF
else
    GPR[rd]55..48 ← (GPR[rs]55..48 + GPR[rt]55..48)7..0
endif

if ((GPR[rs]63..56 + GPR[rt]63..56) > 0xFF) then
    GPR[rd]63..56 ← 0xFF
else
    GPR[rd]63..56 ← (GPR[rs]63..56 + GPR[rt]63..56)7..0
endif

if ((GPR[rs]71..64 + GPR[rt]71..64) > 0xFF) then
    GPR[rd]71..64 ← 0xFF
else
    GPR[rd]71..64 ← (GPR[rs]71..64 + GPR[rt]71..64)7..0
endif

if ((GPR[rs]79..72 + GPR[rt]79..72) > 0xFF) then
    GPR[rd]79..72 ← 0xFF
else
    GPR[rd]79..72 ← (GPR[rs]79..72 + GPR[rt]79..72)7..0
endif

if ((GPR[rs]87..80 + GPR[rt]87..80) > 0xFF) then
    GPR[rd]87..80 ← 0xFF
else
    GPR[rd]87..80 ← (GPR[rs]87..80 + GPR[rt]87..80)7..0
endif

if ((GPR[rs]95..88 + GPR[rt]95..88) > 0xFF) then
    GPR[rd]95..88 ← 0xFF
else
    GPR[rd]95..88 ← (GPR[rs]95..88 + GPR[rt]95..88)7..0
endif

if ((GPR[rs]103..96 + GPR[rt]103..96) > 0xFF) then
    GPR[rd]103..96 ← 0xFF
else
    GPR[rd]103..96 ← (GPR[rs]103..96 + GPR[rt]103..96)7..0
endif

if ((GPR[rs]111..104 + GPR[rt]111..104) > 0xFF) then
    GPR[rd]111..104 ← 0xFF
else
    GPR[rd]111..104 ← (GPR[rs]111..104 + GPR[rt]111..104)7..0
endif

if ((GPR[rs]119..112 + GPR[rt]119..112) > 0xFF) then
    GPR[rd]119..112 ← 0xFF
else
    GPR[rd]119..112 ← (GPR[rs]119..112 + GPR[rt]119..112)7..0
endif
Appendix B  C790-Specific Instruction Set Details

GPR[rd]119..112 ← 0xFF
else
  GPR[rd]119..112 ← (GPR[rs]119..112 + GPR[rt]119..112)7..0
endif

if ((GPR[rs]127..120 + GPR[rt]127..120) > 0xFF) then
  GPR[rd]127..120 ← 0xFF
else
  GPR[rd]127..120 ← (GPR[rs]127..120 + GPR[rt]127..120)7..0
endif

Exceptions:
None
Appendix B  C790-Specific Instruction Set Details

Parallel Add with Unsigned saturation Halfword

**Format:**

PADDUH  rd, rs, rt

**Purpose:**
To add 8 pairs of 16-bit unsigned integers with saturation in parallel.

**Description:**

rd ← rs + rt

The eight unsigned halfword values in GPR rs are added to the corresponding eight unsigned halfword values in GPR rt in parallel. The results are placed into the corresponding eight halfwords in GPR rd.

No overflow exceptions are generated under any circumstances. Results beyond the range of an unsigned halfword value are saturated according to the following:

**Overflow:** 0xFFFF

This instruction operates on 128-bit registers.

**Operation:**

if ((GPR[rs]15..0 + GPR[rt]15..0) > 0xFFFF) then
  GPR[rd]15..0 ← 0xFFFF
else
  GPR[rd]15..0 ← (GPR[rs]15..0 + GPR[rt]15..0)15..0
endif

if ((GPR[rs]31..16 + GPR[rt]31..16) > 0xFFFF) then
  GPR[rd]31..16 ← 0xFFFF
else
  GPR[rd]31..16 ← (GPR[rs]31..16 + GPR[rt]31..16)15..0
endif

if ((GPR[rs]47..32 + GPR[rt]47..32) > 0xFFFF) then
  GPR[rd]47..32 ← 0xFFFF
else
  GPR[rd]47..32 ← (GPR[rs]47..32 + GPR[rt]47..32)15..0
endif

if ((GPR[rs]63..48 + GPR[rt]63..48) > 0xFFFF) then
  GPR[rd]63..48 ← 0xFFFF
else
  GPR[rd]63..48 ← (GPR[rs]63..48 + GPR[rt]63..48)15..0
endif
if \(((\text{GPR}[\text{rs}]_{79..64} + \text{GPR}[\text{rt}]_{79..64}) > 0xFFFF)\) then
\(\text{GPR}[\text{rd}]_{79..64} \leftarrow 0xFFFF\)
else
\(\text{GPR}[\text{rd}]_{79..64} \leftarrow (\text{GPR}[\text{rs}]_{79..64} + \text{GPR}[\text{rt}]_{79..64})_{15..0}\)
endif

if \(((\text{GPR}[\text{rs}]_{95..80} + \text{GPR}[\text{rt}]_{95..80}) > 0xFFFF)\) then
\(\text{GPR}[\text{rd}]_{95..80} \leftarrow 0xFFFF\)
else
\(\text{GPR}[\text{rd}]_{95..80} \leftarrow (\text{GPR}[\text{rs}]_{95..80} + \text{GPR}[\text{rt}]_{95..80})_{15..0}\)
endif

if \(((\text{GPR}[\text{rs}]_{111..96} + \text{GPR}[\text{rt}]_{111..96}) > 0xFFFF)\) then
\(\text{GPR}[\text{rd}]_{111..96} \leftarrow 0xFFFF\)
else
\(\text{GPR}[\text{rd}]_{111..96} \leftarrow (\text{GPR}[\text{rs}]_{111..96} + \text{GPR}[\text{rt}]_{111..96})_{15..0}\)
endif

if \(((\text{GPR}[\text{rs}]_{127..112} + \text{GPR}[\text{rt}]_{127..112}) > 0xFFFF)\) then
\(\text{GPR}[\text{rd}]_{127..112} \leftarrow 0xFFFF\)
else
\(\text{GPR}[\text{rd}]_{127..112} \leftarrow (\text{GPR}[\text{rs}]_{127..112} + \text{GPR}[\text{rt}]_{127..112})_{15..0}\)
endif

### Exceptions:

None
Appendix B  C790-Specific Instruction Set Details

PADDUW  Parallel Add with Unsigned saturation Word

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PADDUW</td>
<td>10000</td>
<td>MMI1</td>
<td>101000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: PADDUW  rd, rs, rt

Purpose: To add 4 pairs of 32-bit unsigned integers with saturation in parallel.

Description: rd ← rs + rt

The four unsigned word values in GPR rs are added to the corresponding four unsigned word values in GPR rt in parallel. The results are placed into the corresponding four words in GPR rd.

No overflow exceptions are generated under any circumstances. Results beyond the range of an unsigned word value are saturated according to the following:

Overflow: 0xFFFFFFFF

This instruction operates on 128-bit registers.

Operation:

if ((GPR[rs]31..0 + GPR[rt]31..0) > 0xFFFFFFFF) then
    GPR[rd]31..0 ← 0xFFFFFFFF
else
    GPR[rd]31..0 ← (GPR[rs]31..0 + GPR[rt]31..0)31..0
endif

if ((GPR[rs]63..32 + GPR[rt]63..32) > 0xFFFFFFFF) then
    GPR[rd]63..32 ← 0xFFFFFFFF
else
    GPR[rd]63..32 ← (GPR[rs]63..32 + GPR[rt]63..32)31..0
endif

if ((GPR[rs]95..64 + GPR[rt]95..64) > 0xFFFFFFFF) then
    GPR[rd]95..64 ← 0xFFFFFFFF
else
    GPR[rd]95..64 ← (GPR[rs]95..64 + GPR[rt]95..64)31..0
endif

if ((GPR[rs]127..96 + GPR[rt]127..96) > 0xFFFFFFFF) then
    GPR[rd]127..96 ← 0xFFFFFFFF
else
    GPR[rd]127..96 ← (GPR[rs]127..96 + GPR[rt]127..96)31..0
endif
### C790-Specific Instruction Set Details

<table>
<thead>
<tr>
<th>rs</th>
<th>127</th>
<th>96</th>
<th>95</th>
<th>64</th>
<th>63</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A3</td>
<td>A2</td>
<td>A1</td>
<td>A0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rt</th>
<th>127</th>
<th>96</th>
<th>95</th>
<th>64</th>
<th>63</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B3</td>
<td>B2</td>
<td>B1</td>
<td>B0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rd*</th>
<th>127</th>
<th>96</th>
<th>95</th>
<th>64</th>
<th>63</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3+B3</td>
<td>A2+B2</td>
<td>A1+B1</td>
<td>A0+B0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Saturate to unsigned word

### Exceptions:
None
PADDW  Parallel Add Word  PADDW

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PADDW</td>
<td>00000</td>
<td>MMI0</td>
<td>00100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C790

Format: PADDW rd, rs, rt

Purpose: To add 4 pairs of 32-bit integers in parallel.

Description: rd ← rs + rt

The four word values in GPR rs are added to the corresponding four word values in GPR rt in parallel. The results are placed into the corresponding four words in GPR rd.

No overflow or underflow exceptions are generated under any circumstances.

This instruction operates on 128-bit registers.

Operation:

| GPR[rd]31..0 | ← (GPR[rs]31..0 + GPR[rt]31..0)31..0 |
| GPR[rd]63..32 | ← (GPR[rs]63..32 + GPR[rt]63..32)31..0 |
| GPR[rd]95..64 | ← (GPR[rs]95..64 + GPR[rt]95..64)31..0 |
| GPR[rd]127..96 | ← (GPR[rs]127..96 + GPR[rt]127..96)31..0 |

Exceptions:

None
## PADSBH
### Parallel Add/Subtract Halfword

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>PADSBH</th>
<th>MMI1</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>00100</td>
<td>101000</td>
</tr>
</tbody>
</table>

### Format:
PADSBH rd, rs, rt

### Purpose:
To add/subtract 8 pairs of 16-bit integers in parallel.

### Description:
rd ← rs +/- rt

The high-order four halfword values in GPR rs are added to the corresponding four halfword values in GPR rt and the low-order four halfword values in GPR rt are subtracted from the corresponding four halfword values in GPR rs in parallel. The results are placed into the corresponding eight halfword values in GPR rd.

No overflow or underflow exceptions are generated under any circumstances.

This instruction operates on 128-bit registers.

### Operation

\[
\begin{align*}
\text{GPR}[rd]_{15..0} & \leftarrow (\text{GPR}[rs]_{15..0} - \text{GPR}[rt]_{15..0})_{15..0} \\
\text{GPR}[rd]_{31..16} & \leftarrow (\text{GPR}[rs]_{31..16} - \text{GPR}[rt]_{31..16})_{15..0} \\
\text{GPR}[rd]_{47..32} & \leftarrow (\text{GPR}[rs]_{47..32} - \text{GPR}[rt]_{47..32})_{15..0} \\
\text{GPR}[rd]_{63..48} & \leftarrow (\text{GPR}[rs]_{63..48} - \text{GPR}[rt]_{63..48})_{15..0} \\
\text{GPR}[rd]_{79..64} & \leftarrow (\text{GPR}[rs]_{79..64} + \text{GPR}[rt]_{79..64})_{15..0} \\
\text{GPR}[rd]_{95..80} & \leftarrow (\text{GPR}[rs]_{95..80} + \text{GPR}[rt]_{95..80})_{15..0} \\
\text{GPR}[rd]_{111..96} & \leftarrow (\text{GPR}[rs]_{111..96} + \text{GPR}[rt]_{111..96})_{15..0} \\
\text{GPR}[rd]_{127..112} & \leftarrow (\text{GPR}[rs]_{127..112} + \text{GPR}[rt]_{127..112})_{15..0}
\end{align*}
\]

### Exceptions:
None
Parallel And

**PAND**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PAND</td>
<td>MMI2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>10010</td>
<td>001001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \[ \text{PAND} \text{ rd, rs, rt} \]

**Purpose:** To perform a bitwise logical AND.

**Description:** \[ \text{rd} \leftarrow \text{rs AND rt} \]

The contents of GPR \( rs \) are combined with the contents of GPR \( rt \) in a bitwise logical AND operation. The result is placed into GPR \( rd \).

This instruction operates on 128-bit registers.

**Operation:**

\[
\text{GPR[rd]}_{127..0} \leftarrow \text{GPR[rs]}_{127..0} \text{ and GPR[rt]}_{127..0}
\]

**Exceptions:**

None
**Appendix B  C790-Specific Instruction Set Details**

**PCEQB**

Parallel Compare for Equal Byte

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PCEQB</td>
<td>MMI1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**
PCEQB  rd, rs, rt

**Purpose:**
To record the result of 16 equality comparisons in parallel.

**Description:**
rd ← (rs = rt)

The sixteen signed byte values in GPR rs are compared to the corresponding sixteen signed byte values in GPR rt, in parallel. The results of the comparison are placed into GPR rd as follows:

If the signed byte value in GPR rs is equal to the corresponding signed byte value in GPR rt, then the corresponding byte in GPR rd is set to 0xFF otherwise it is set to 0x00.

This instruction operates on 128-bit registers.

**Operation:**

if (GPR[rs]7..0 = GPR[rt]7..0) then
  GPR[rd]7..0 ← 18
else
  GPR[rd]7..0 ← 08
endif

if (GPR[rs]15..8 = GPR[rt]15..8) then
  GPR[rd]15..8 ← 18
else
  GPR[rd]15..8 ← 08
endif

if (GPR[rs]23..16 = GPR[rt]23..16) then
  GPR[rd]23..16 ← 18
else
  GPR[rd]23..16 ← 08
endif

if (GPR[rs]31..24 = GPR[rt]31..24) then
  GPR[rd]31..24 ← 18
else
  GPR[rd]31..24 ← 08
endif
if (GPR[rs]39..32 = GPR[rt]39..32) then
    GPR[rd]39..32 ← 18
else
    GPR[rd]39..32 ← 08
endif

if (GPR[rs]47..40 = GPR[rt]47..40) then
    GPR[rd]47..40 ← 18
else
    GPR[rd]47..40 ← 08
endif

if (GPR[rs]55..48 = GPR[rt]55..48) then
    GPR[rd]55..48 ← 18
else
    GPR[rd]55..48 ← 08
endif

if (GPR[rs]63..56 = GPR[rt]63..56) then
    GPR[rd]63..56 ← 18
else
    GPR[rd]63..56 ← 08
endif

if (GPR[rs]71..64 = GPR[rt]71..64) then
    GPR[rd]71..64 ← 18
else
    GPR[rd]71..64 ← 08
endif

if (GPR[rs]79..72 = GPR[rt]79..72) then
    GPR[rd]79..72 ← 18
else
    GPR[rd]79..72 ← 08
endif

if (GPR[rs]87..80 = GPR[rt]87..80) then
    GPR[rd]87..80 ← 18
else
    GPR[rd]87..80 ← 08
endif

if (GPR[rs]95..88 = GPR[rt]95..88) then
    GPR[rd]95..88 ← 18
else
    GPR[rd]95..88 ← 08
endif

if (GPR[rs]103..96 = GPR[rt]103..96) then
    GPR[rd]103..96 ← 18
else
    GPR[rd]103..96 ← 08
endif

if (GPR[rs]111..104 = GPR[rt]111..104) then
GPR[rd]111..104 ← $1^8$
else
  GPR[rd]111..104 ← $0^8$
endif

if (GPR[rs]119..112 = GPR[rt]119..112) then
  GPR[rd]119..112 ← $1^8$
else
  GPR[rd]119..112 ← $0^8$
endif

if (GPR[rs]127..120 = GPR[rt]127..120) then
  GPR[rd]127..120 ← $1^8$
else
  GPR[rd]127..120 ← $0^8$
endif

Exceptions:
None
Parallel Compare for Equal Halfword

**Format:**
PCEQH rd, rs, rt

**Purpose:**
To record the results of 8 equality comparisons in parallel.

**Description:**
rd ← (rs = rt)

The eight signed halfword values in GPR rs are compared to the corresponding eight signed halfword values in GPR rt, in parallel. The results of the comparison are placed into GPR rd as follows:

If the signed halfword value in GPR rs is equal to the corresponding signed halfword value in GPR rt, then the corresponding halfword in GPR rd is set to 0xFFFF otherwise it is set to 0x0000.

This instruction operates on 128-bit registers.

**Operation:**

if (GPR[rs]15..0 = GPR[rt]15..0) then
    GPR[rd]15..0 ← 116
else
    GPR[rd]15..0 ← 016
endif

if (GPR[rs]31..16 = GPR[rt]31..16) then
    GPR[rd]31..16 ← 116
else
    GPR[rd]31..16 ← 016
endif

if (GPR[rs]47..32 = GPR[rt]47..32) then
    GPR[rd]47..32 ← 116
else
    GPR[rd]47..32 ← 016
endif

if (GPR[rs]63..48 = GPR[rt]63..48) then
    GPR[rd]63..48 ← 116
else
    GPR[rd]63..48 ← 016
endif
if (GPR[rs]79..64 = GPR[rt]79..64) then
  GPR[rd]79..64 ← 116
else
  GPR[rd]79..64 ← 016
endif

if (GPR[rs]95..80 = GPR[rt]95..80) then
  GPR[rd]95..80 ← 116
else
  GPR[rd]95..80 ← 016
endif

if (GPR[rs]111..96 = GPR[rt]111..96) then
  GPR[rd]111..96 ← 116
else
  GPR[rd]111..96 ← 016
endif

if (GPR[rs]127..112 = GPR[rt]127..112) then
  GPR[rd]127..112 ← 116
else
  GPR[rd]127..112 ← 016
endif

Exceptions:
None
C790

Appendix B  C790-Specific Instruction Set Details

Parallel Compare for Equal Word

PCEQW

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PCEQW</td>
<td>MMI1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>00010</td>
<td>101000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: PCEQW  rd, rs, rt
Purpose: To record the result of 4 equality comparisons in parallel.
Description: rd ← (rs = rt)

The four signed word values in GPR rs are compared to the corresponding four signed word values in GPR rt, in parallel. The results of the comparison are placed into GPR rd as follows:

If the signed word value in GPR rs is equal to the corresponding signed word value in GPR rt, then the corresponding word in GPR rd is set to 0xFFFFFFFF otherwise it is set to 0x00000000.

This instruction operates on 128-bit registers.

Operation:

if (GPR[rs]31..0 = GPR[rt]31..0) then
    GPR[rd]31..0 ← 1
else
    GPR[rd]31..0 ← 0
endif

if (GPR[rs]63..32 = GPR[rt]63..32) then
    GPR[rd]63..32 ← 1
else
    GPR[rd]63..32 ← 0
endif

if (GPR[rs]95..64 = GPR[rt]95..64) then
    GPR[rd]95..64 ← 1
else
    GPR[rd]95..64 ← 0
endif

if (GPR[rs]127..96 = GPR[rt]127..96) then
    GPR[rd]127..96 ← 1
else
    GPR[rd]127..96 ← 0
endif
### Appendix B  C790-Specific Instruction Set Details

<table>
<thead>
<tr>
<th>rs</th>
<th>[A3]</th>
<th>[A2]</th>
<th>[A1]</th>
<th>[A0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt</td>
<td>[B3]</td>
<td>[B2]</td>
<td>[B1]</td>
<td>[B0]</td>
</tr>
</tbody>
</table>

#### Exceptions:

None
**PCGTB**

**Parallel Compare for Greater Than Byte**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PCGTB</td>
<td>01010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: PCGTB rd, rs, rt

Purpose: To record the result of 16 greater-than comparisons in parallel.

Description: rd ← (rs > rt)

The sixteen signed byte values in GPR rs are compared to the corresponding sixteen signed byte values in GPR rt in parallel. The results of the comparison are placed into GPR rd as follows:

If the signed byte value in GPR rs is greater than the corresponding signed byte value in GPR rt, then the corresponding byte in GPR rd is set to 0xFF otherwise it is set to 0x00.

This instruction operates on 128-bit registers.

Operation:

```plaintext
if (GPR[rs]7..0 > GPR[rt]7..0) then
    GPR[rd]7..0 ← 18
else
    GPR[rd]7..0 ← 08
endif

if (GPR[rs]15..8 > GPR[rt]15..8) then
    GPR[rd]15..8 ← 18
else
    GPR[rd]15..8 ← 08
endif

if (GPR[rs]23..16 > GPR[rt]23..16) then
    GPR[rd]23..16 ← 18
else
    GPR[rd]23..16 ← 08
endif

if (GPR[rs]31..24 > GPR[rt]31..24) then
    GPR[rd]31..24 ← 18
else
    GPR[rd]31..24 ← 08
endif
```
if (GPR[rs]39..32 > GPR[rt]39..32) then
    GPR[rd]39..32 ← 18
else
    GPR[rd]39..32 ← 08
endif

if (GPR[rs]47..40 > GPR[rt]47..40) then
    GPR[rd]47..40 ← 18
else
    GPR[rd]47..40 ← 08
endif

if (GPR[rs]55..48 > GPR[rt]55..48) then
    GPR[rd]55..48 ← 18
else
    GPR[rd]55..48 ← 08
endif

if (GPR[rs]63..56 > GPR[rt]63..56) then
    GPR[rd]63..56 ← 18
else
    GPR[rd]63..56 ← 08
endif

if (GPR[rs]71..64 > GPR[rt]71..64) then
    GPR[rd]71..64 ← 18
else
    GPR[rd]71..64 ← 08
endif

if (GPR[rs]79..72 > GPR[rt]79..72) then
    GPR[rd]79..72 ← 18
else
    GPR[rd]79..72 ← 08
endif

if (GPR[rs]87..80 > GPR[rt]87..80) then
    GPR[rd]87..80 ← 18
else
    GPR[rd]87..80 ← 08
endif

if (GPR[rs]95..88 > GPR[rt]95..88) then
    GPR[rd]95..88 ← 18
else
    GPR[rd]95..88 ← 08
endif
if (GPR[rs]103..96 > GPR[rt]103..96) then
    GPR[rd]103..96 ← 18
else
    GPR[rd]103..96 ← 08
endif

if (GPR[rs]111..104 > GPR[rt]111..104) then
    GPR[rd]111..104 ← 18
else
    GPR[rd]111..104 ← 08
endif

if (GPR[rs]119..112 > GPR[rt]119..112) then
    GPR[rd]119..112 ← 18
else
    GPR[rd]119..112 ← 08
endif

if (GPR[rs]127..120 > GPR[rt]127..120) then
    GPR[rd]127..120 ← 18
else
    GPR[rd]127..120 ← 08
endif

Exceptions:

None
### Parallel Compare for Greater Than Halfword (PCGTH)

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PCGTH</td>
<td>MMI0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>00110</td>
<td>00100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Format:
PCGTH rd, rs, rt

#### Purpose:
To record the results of 8 greater-than comparisons in parallel.

#### Description:
r
\( \leftarrow (rs > rt) \)

The eight signed halfword values in GPR rs are compared to the corresponding eight signed halfword values in GPR rt in parallel. The results of the comparison are placed into GPR rd as follows:

If the signed halfword value in GPR rs is greater than the corresponding signed halfword value in GPR rt, then the corresponding halfword in GPR rd is set to 0xFFFF otherwise it is set to 0x0000.

This instruction operates on 128-bit registers.

#### Operation:

\[
\begin{align*}
    &\text{if } (\text{GPR}[rs]_{15..0} > \text{GPR}[rt]_{15..0}) \text{ then} \\
    &\quad \text{GPR}[rd]_{15..0} \leftarrow 1_{16} \\
    &\quad \text{else} \\
    &\quad \text{GPR}[rd]_{15..0} \leftarrow 0_{16} \\
    &\quad \text{endif} \\
    &\text{if } (\text{GPR}[rs]_{31..16} > \text{GPR}[rt]_{31..16}) \text{ then} \\
    &\quad \text{GPR}[rd]_{31..16} \leftarrow 1_{16} \\
    &\quad \text{else} \\
    &\quad \text{GPR}[rd]_{31..16} \leftarrow 0_{16} \\
    &\quad \text{endif} \\
    &\text{if } (\text{GPR}[rs]_{47..32} > \text{GPR}[rt]_{47..32}) \text{ then} \\
    &\quad \text{GPR}[rd]_{47..32} \leftarrow 1_{16} \\
    &\quad \text{else} \\
    &\quad \text{GPR}[rd]_{47..32} \leftarrow 0_{16} \\
    &\quad \text{endif} \\
    &\text{if } (\text{GPR}[rs]_{63..48} > \text{GPR}[rt]_{63..48}) \text{ then} \\
    &\quad \text{GPR}[rd]_{63..48} \leftarrow 1_{16} \\
    &\quad \text{else} \\
    &\quad \text{GPR}[rd]_{63..48} \leftarrow 0_{16} \\
    &\quad \text{endif}
\end{align*}
\]
if (GPR[rs]_{79..64} > GPR[rt]_{79..64}) then
    GPR[rd]_{79..64} ← 1_{16}
else
    GPR[rd]_{79..64} ← 0_{16}
endif

if (GPR[rs]_{95..80} > GPR[rt]_{95..80}) then
    GPR[rd]_{95..80} ← 1_{16}
else
    GPR[rd]_{95..80} ← 0_{16}
endif

if (GPR[rs]_{111..96} > GPR[rt]_{111..96}) then
    GPR[rd]_{111..96} ← 1_{16}
else
    GPR[rd]_{111..96} ← 0_{16}
endif

if (GPR[rs]_{127..112} > GPR[rt]_{127..112}) then
    GPR[rd]_{127..112} ← 1_{16}
else
    GPR[rd]_{127..112} ← 0_{16}
endif

Exceptions:
None
**PCGTW**

**Parallel Compare for Greater Than Word**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PCGTW</td>
<td>00010</td>
<td>MMI0</td>
<td>001000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

PCGTW  rd, rs, rt

**Purpose:**

To record the results of 4 greater-than comparisons in parallel.

**Description:**

rd ← (rs > rt)

The four signed word values in GPR rs are compared to the corresponding four signed word values in GPR rt in parallel. The results of the comparison are placed into GPR rd as follows:

If the signed word value in GPR rs is greater than the corresponding signed word value in GPR rt, then the corresponding word in GPR rd is set to 0xFFFFFFFF otherwise it is set to 0x00000000.

This instruction operates on 128-bit registers.

**Operation:**

\[
\begin{align*}
\text{if } (GPR[rs]_{31..0} > GPR[rt]_{31..0}) \text{ then} \\
& \quad \text{GPR}[rd]_{31..0} \leftarrow 1_{32} \\
\text{else} \\
& \quad \text{GPR}[rd]_{31..0} \leftarrow 0_{32} \\
\text{endif} \\
\text{if } (GPR[rs]_{63..32} > GPR[rt]_{63..32}) \text{ then} \\
& \quad \text{GPR}[rd]_{63..32} \leftarrow 1_{32} \\
\text{else} \\
& \quad \text{GPR}[rd]_{63..32} \leftarrow 0_{32} \\
\text{endif} \\
\text{if } (GPR[rs]_{95..64} > GPR[rt]_{95..64}) \text{ then} \\
& \quad \text{GPR}[rd]_{95..64} \leftarrow 1_{32} \\
\text{else} \\
& \quad \text{GPR}[rd]_{95..64} \leftarrow 0_{32} \\
\text{endif} \\
\text{if } (GPR[rs]_{127..96} > GPR[rt]_{127..96}) \text{ then} \\
& \quad \text{GPR}[rd]_{127..96} \leftarrow 1_{32} \\
\text{else} \\
& \quad \text{GPR}[rd]_{127..96} \leftarrow 0_{32} \\
\text{endif}
\end{align*}
\]
### Appendix B  C790-Specific Instruction Set Details

<table>
<thead>
<tr>
<th>rs</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt</td>
<td>B3</td>
<td>B2</td>
<td>B1</td>
<td>B0</td>
</tr>
<tr>
<td>rd</td>
<td>0^32</td>
<td>1^32</td>
<td>0^32</td>
<td>1^32</td>
</tr>
</tbody>
</table>

#### Exception:

None
### Parallel Copy Halfword

<table>
<thead>
<tr>
<th>MMI</th>
<th>011100</th>
<th>rt</th>
<th>rd</th>
<th>PCPYH</th>
<th>110111</th>
<th>MMI3</th>
<th>101001</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

PCPYH rd, rt

**Purpose:**

To copy halfword.

**Description:**

rd ← copy (rt)

The contents of the low-order halfword of the two doublewords in GPR rt are copied to each of the halfwords of the two doublewords in GPR rd.

This instruction operates on 128-bit registers.

**Operation:**

- \( \text{GPR}[rd]_{15.0} \leftarrow \text{GPR}[rt]_{15.0} \)
- \( \text{GPR}[rd]_{31.16} \leftarrow \text{GPR}[rt]_{15.0} \)
- \( \text{GPR}[rd]_{47.32} \leftarrow \text{GPR}[rt]_{15.0} \)
- \( \text{GPR}[rd]_{63.48} \leftarrow \text{GPR}[rt]_{15.0} \)
- \( \text{GPR}[rd]_{79.64} \leftarrow \text{GPR}[rt]_{79.64} \)
- \( \text{GPR}[rd]_{95.80} \leftarrow \text{GPR}[rt]_{79.64} \)
- \( \text{GPR}[rd]_{111.96} \leftarrow \text{GPR}[rt]_{79.64} \)
- \( \text{GPR}[rd]_{127.112} \leftarrow \text{GPR}[rt]_{79.64} \)

**Exceptions:**

None
Appendix B  C790-Specific Instruction Set Details

**PCPYLD**  Parallel Copy Lower Doubleword  **PCPYLD**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PCPYLD</td>
<td>MMI2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>001001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  
PCPYLD  rd, rs, rt

**Purpose:**  
To copy doubleword.

**Description:**  
rd ← copy (rs, rt)

The contents of the low-order doubleword in GPR rs are combined with the contents of the low-order doubleword in GPR rt. The quadword result is placed into GPR rd.

This instruction operates on 128-bit registers.

**Operation:**

GPR[rd]_{63..0} ← GPR[rt]_{63..0}
GPR[rd]_{127..64} ← GPR[rs]_{63..0}

**Exceptions:**

None
Parallel Copy Upper Doubleword

**PCPYUD**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PCPYUD</td>
<td>MMI3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>011100</td>
<td>101001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C790

**Format:**

PCPYUD rd, rs, rt

**Purpose:**

To copy doubleword.

**Description:**

rd ← copy (rs, rt)

The contents of the high-order doubleword in GPR rs are combined with the contents of the high-order doubleword in GPR rt. The quadword result is placed into GPR rd.

This instruction operates on 128-bit registers.

**Operation**

GPR[rd]63..0 ← GPR[rs]127..64
GPR[rd]127..64 ← GPR[rt]127..64

**Exceptions:**

None
**PDIVBW**

**Parallel Divide Broadcast Word**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMO</td>
<td>rs</td>
<td>rt</td>
<td>00000</td>
<td>PDIVBW</td>
<td>11101</td>
<td>MMO2</td>
<td>001001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**
PDIVBW rs, rt

**Purpose:** To divide 4 32-bit signed integers by a 16-bit signed integer in parallel.

**Description:**
\[(\text{LO}, \text{HI}) \leftarrow rs / rt\]

The four signed words in GPR rs are divided by the low-order signed halfword in GPR rt, in parallel. The four 32-bit quotients are placed into special register LO. The four 16-bit remainders are placed into special register HI.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

**Restrictions:**
If the divisor in GPR rt is zero, the arithmetic result value is undefined.

**Operation:**
\[
\begin{align*}
q_0 & \leftarrow \text{GPR}[rs]_{31..0} \text{ div GPR}[rt]_{15..0} \\
r_0 & \leftarrow \text{GPR}[rs]_{31..0} \text{ mod GPR}[rt]_{15..0} \\
q_1 & \leftarrow \text{GPR}[rs]_{63..32} \text{ div GPR}[rt]_{15..0} \\
r_1 & \leftarrow \text{GPR}[rs]_{63..32} \text{ mod GPR}[rt]_{15..0} \\
q_2 & \leftarrow \text{GPR}[rs]_{95..64} \text{ div GPR}[rt]_{15..0} \\
r_2 & \leftarrow \text{GPR}[rs]_{95..64} \text{ mod GPR}[rt]_{15..0} \\
q_3 & \leftarrow \text{GPR}[rs]_{127..96} \text{ div GPR}[rt]_{15..0} \\
r_3 & \leftarrow \text{GPR}[rs]_{127..96} \text{ mod GPR}[rt]_{15..0} \\
\text{LO}_{31..0} & \leftarrow q_0_{31..0} \\
\text{HI}_{31..0} & \leftarrow (r_0_{15})_{16} || r_{015..0} \\
\text{LO}_{63..32} & \leftarrow q_1_{31..0} \\
\text{HI}_{63..32} & \leftarrow (r_1_{15})_{16} || r_{115..0} \\
\text{LO}_{95..64} & \leftarrow q_2_{31..0} \\
\text{HI}_{95..64} & \leftarrow (r_2_{15})_{16} || r_{215..0} \\
\text{LO}_{127..96} & \leftarrow q_3_{31..0} \\
\text{HI}_{127..96} & \leftarrow (r_3_{15})_{16} || r_{315..0}
\end{align*}
\]
Supplementary explanation:

When 0x80000000 (-2147483648), the most negative value, is divided by 0xFFFF (-1), the operation will result in an overflow. However, overflow exception doesn't occur and the operation results in the following:

Quotient is 0x80000000 (-2147483648), and remainder is 0x00000000 (0).

Exceptions:

None

Programming Notes:

In the C790 the integer divide operation proceeds asynchronously and allows other CPU instructions to execute before it is retired. An attempt to read LO or HI before the results are written will cause an interlock until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the divide so that other instructions can execute in parallel.

No arithmetic exception occurs under any circumstances. If divide-by-zero or overflow conditions should be detected and some action taken, then the divide instruction is typically followed by additional instructions to check for a zero divisor and / or for overflow. If the divide is asynchronous then the zero-divisor check can execute in parallel with the divide. The action taken on either divide-by-zero or overflow is either a convention within the program itself or more typically, the system software; one possibility is to take a BREAK exception with a code field value to signal the problem to the system software.

As an example, the C programming language in a UNIX environment expects division by zero to either terminate the program or execute a program-specified signal handler. C does not expect overflow to cause any exceptional condition. If the C compiler uses a divide instruction, it also emits code to test for a zero divisor and execute a BREAK instruction to inform the operating system if one is detected.
Parallel Divide Unsigned Word

**Format:**
PDIVUW rs, rt

**Purpose:**
To divide 2 pairs of 32-bit unsigned integers in parallel.

**Description:**
(LO, HI) ← rs / rt

The low-order unsigned word of the two doublewords in GPR rs are divided by the low-order unsigned word of the two doublewords in GPR rt in parallel. The two 32 bit quotients are placed into special register LO. The two 32-bit remainders are placed into special register HI.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

**Restrictions:**
If neither GPR rt nor GPR rs contain a zero-extended 32-bit value (bits 127..96 and 63..32 equal zero), the result of the operation will be undefined.

If the divisor in GPR rt is zero, the result will be undefined.

**Operation:**
\[
\text{if } (\text{NotWordValue}(\text{GPR}[rs]) \text{ or } \text{NotWordValue}(\text{GPR}[rt])) \text{ then UndefinedResult() endif}
\]
\[
q0 \leftarrow (0 || \text{GPR}[rs]_{31.0}) \text{ div } (0 || \text{GPR}[rt]_{31.0})
\]
\[
r0 \leftarrow (0 || \text{GPR}[rs]_{31.0}) \text{ mod } (0 || \text{GPR}[rt]_{31.0})
\]
\[
q1 \leftarrow (0 || \text{GPR}[rs]_{95.64}) \text{ div } (0 || \text{GPR}[rt]_{95.64})
\]
\[
r1 \leftarrow (0 || \text{GPR}[rs]_{95.64}) \text{ mod } (0 || \text{GPR}[rt]_{95.64})
\]
\[
\text{LO}_{63..0} \leftarrow (q0_{32} || q0_{31..0})
\]
\[
\text{HI}_{63..0} \leftarrow (r0_{32} || r0_{31..0})
\]
\[
\text{LO}_{127..64} \leftarrow (q1_{32} || q1_{31..0})
\]
\[
\text{HI}_{127..64} \leftarrow (r1_{32} || r1_{31..0})
\]
Exceptions:

None

Programming Notes:

See the Programming Notes for the PDIVBW instruction.
**PDIVW**  
**Parallel Divide Word**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>rs</td>
<td>rt</td>
<td>0</td>
<td>00000</td>
<td>PDIVW</td>
<td>01101</td>
<td>MMI2</td>
<td>001001</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  
**PDIVW** rs, rt

**Purpose:**  
To divide 2 pairs of 32-bit signed integers in parallel.

**Description:**  
\[(LO, HI) \leftarrow rs / rt\]

The low-order signed word of the two doublewords in GPR rs are divided by the low-order signed word of the two doublewords in GPR rt in parallel. The two 32 bit quotients are placed into special register LO. The two 32-bit remainders are placed into special register HI.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

**Restrictions:**

- If neither GPR rt nor GPR rs contain a sign-extended 32-bit value (bits 127..95 equal and 63..31 equal), the result of the operation will be undefined.
- If the divisor in GPR rt is zero, the result will be undefined.

**Operation:**

```plaintext
defined = NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])
if defined then UndefinedResult() else
q0 ← GPR[rs]31..0 div GPR[rt]31..0
r0 ← GPR[rs]31..0 mod GPR[rt]31..0
q1 ← GPR[rs]95..64 div GPR[rt]95..64
r1 ← GPR[rs]95..64 mod GPR[rt]95..64
LO63..0 ← (q0 31)32 || q031..0
HI63..0 ← (r0 31)32 || r031..0
LO127..64 ← (q1 31)32 || q131..0
HI127..64 ← (r1 31)32 || r131..0
```

<table>
<thead>
<tr>
<th></th>
<th>127</th>
<th>96</th>
<th>95</th>
<th>64</th>
<th>63</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs</td>
<td></td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>127</td>
<td>96</td>
<td>95</td>
<td>÷</td>
<td>64</td>
<td>63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rt</td>
<td></td>
<td>B1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>127</th>
<th>96</th>
<th>95</th>
<th>64</th>
<th>63</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>HI</td>
<td>sign ext</td>
<td>A1 mod B1</td>
<td></td>
<td>sign ext</td>
<td>A0 mod B0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>127</td>
<td>96</td>
<td>95</td>
<td>64</td>
<td>63</td>
<td>32</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>LO</td>
<td>sign ext</td>
<td>A1 div B1</td>
<td></td>
<td>sign ext</td>
<td>A0 div B0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Supplementary explanation:

When 0x80000000 (-2147483648), the most negative value, is divided by 0xFFFFFFFF (-1), the operation results in an overflow. However, overflow exception doesn’t occur; the operation results in the followings:

Quotient (q) is 0x80000000 (-2147483648), and remainder (r) is 0x00000000(0).

Exceptions:

None

Programming Notes:

See the Programming Notes for the PDIVBW instruction.
PEXCH Parallel Exchange Center Halfword

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>0</td>
<td>00000</td>
<td>rt</td>
<td>rd</td>
<td>PEXCH</td>
<td>11010</td>
<td>MMI3</td>
<td>101001</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: PEXCH rd, rt

Purpose: To exchange halfwords.

Description: rd ← exchange (rt)

The two central halfwords of the high-order doubleword in GPR rt are exchanged and the two central halfwords of the low-order doubleword in GPR rt are exchanged. The results are copied to GPR rd while other halfwords are copied directly to the corresponding halfwords.

This instruction operates on 128-bit registers.

Operation:
- GPR[rd]15..0 ← GPR[rt]15..0
- GPR[rd]31..16 ← GPR[rt]47..32
- GPR[rd]47..32 ← GPR[rt]31..16
- GPR[rd]63..48 ← GPR[rt]63..48
- GPR[rd]79..64 ← GPR[rt]79..64
- GPR[rd]95..80 ← GPR[rt]111..96
- GPR[rd]111..96 ← GPR[rt]95..80
- GPR[rd]127..112 ← GPR[rt]127..112

Exceptions:
None
## PEXCW

**Parallel Exchange Center Word**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>000000</td>
<td>rt</td>
<td>rd</td>
<td>PEXCW</td>
<td>11110</td>
<td>MMI3</td>
<td>101001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

PEXCW  rd, rt

**Purpose:**

To exchange words.

**Description:**

rd ← exchange (rt)

The two central words in GPR rt are exchanged. The results are copied to GPR rd while other words are copied directly to the corresponding words.

This instruction operates on 128-bit registers.

**Operation:**

- GPR[rd]31..0 ← GPR[rt]31..0
- GPR[rd]63..32 ← GPR[rt]95..64
- GPR[rd]95..64 ← GPR[rt]63..32
- GPR[rd]127..96 ← GPR[rt]127..96

**Exceptions:**

None
Appendix B  C790-Specific Instruction Set Details

PEXEH  Parallel Exchange Even Halfword  PEXEH

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>000000</td>
<td>rt</td>
<td>rd</td>
<td>PEXEH</td>
<td>11010</td>
<td>MMI2</td>
<td>001001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C790

Format:  PEXEH  rd, rt
Purpose:  To exchange halfwords.
Description:  rd ← exchange (rt)

The two low-order halfwords of the two words of the high-order doubleword in GPR  rt are exchanged and the two low-order halfwords of the two words of the low-order doubleword in GPR  rt are exchanged. The results are copied to GPR  rd while other halfwords are copied directly to the corresponding halfwords.

This instruction operates on 128-bit registers.

Operation:

GPR[rd]15..0 ← GPR[rt]47..32
GPR[rd]31..16 ← GPR[rt]31..16
GPR[rd]47..32 ← GPR[rt]15..0
GPR[rd]63..48 ← GPR[rt]63..48
GPR[rd]79..64 ← GPR[rt]111..96
GPR[rd]95..80 ← GPR[rt]95..80
GPR[rd]111..96 ← GPR[rt]79..64
GPR[rd]127..112 ← GPR[rt]127..112

Exceptions:

None
PEXEW  Parallel Exchange Even Word  PEXEW

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>0</td>
<td>00000</td>
<td>rt</td>
<td>rd</td>
<td>PEXEW</td>
<td>11110</td>
<td>MMI2</td>
<td>001001</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C790

Format: PEXEW rd, rt

Purpose: To exchange word.

Description: rd ← exchange (rt)

The two low-order words of the two doublewords in GPR rt are exchanged. The results are copied to GPR rd while other words are copied directly to the corresponding words.

This instruction operates on 128-bit registers.

Operation:

GPR[rd]_{31..0} ← GPR[rt]_{95..64}
GPR[rd]_{63..32} ← GPR[rt]_{63..32}
GPR[rd]_{95..64} ← GPR[rt]_{31..0}
GPR[rd]_{127..96} ← GPR[rt]_{127..96}

Exceptions:

None
**PEXT5**

**Parallel Extend from 5-bits**

<table>
<thead>
<tr>
<th>rd</th>
<th>rt</th>
<th>PEXT5</th>
<th>MMI0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>11110</td>
<td>001000</td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

`PEXT5 rd, rt`

**Purpose:**

To extend bytes from 5-bits.

**Description:**

rd ← extend (rt)

The four low-order 16-bits (1, 5, 5, 5 bit) of the four words in GPR rt are extended to four 32-bits (8, 8, 8, 8 bit). The quadword result is placed into GPR rd.

This instruction operates on 128-bit registers.

**Operation**

<table>
<thead>
<tr>
<th>GPR[rd]</th>
<th>←</th>
</tr>
</thead>
<tbody>
<tr>
<td>2..0</td>
<td>03</td>
</tr>
<tr>
<td>7..3</td>
<td>GPR[rt]</td>
</tr>
<tr>
<td>10..8</td>
<td>03</td>
</tr>
<tr>
<td>15..11</td>
<td>GPR[rt]</td>
</tr>
<tr>
<td>18..16</td>
<td>03</td>
</tr>
<tr>
<td>23..19</td>
<td>GPR[rt]</td>
</tr>
<tr>
<td>30..24</td>
<td>07</td>
</tr>
<tr>
<td>31</td>
<td>GPR[rt]</td>
</tr>
<tr>
<td>34..32</td>
<td>03</td>
</tr>
<tr>
<td>39..35</td>
<td>GPR[rt]</td>
</tr>
<tr>
<td>42..40</td>
<td>03</td>
</tr>
<tr>
<td>47..43</td>
<td>GPR[rt]</td>
</tr>
<tr>
<td>50..48</td>
<td>03</td>
</tr>
<tr>
<td>55..51</td>
<td>GPR[rt]</td>
</tr>
<tr>
<td>62..56</td>
<td>07</td>
</tr>
<tr>
<td>63</td>
<td>GPR[rt]</td>
</tr>
<tr>
<td>66..64</td>
<td>03</td>
</tr>
<tr>
<td>71..67</td>
<td>GPR[rt]</td>
</tr>
<tr>
<td>74..72</td>
<td>03</td>
</tr>
<tr>
<td>79..75</td>
<td>GPR[rt]</td>
</tr>
<tr>
<td>82..80</td>
<td>03</td>
</tr>
<tr>
<td>87..83</td>
<td>GPR[rt]</td>
</tr>
<tr>
<td>94..88</td>
<td>07</td>
</tr>
<tr>
<td>95</td>
<td>GPR[rt]</td>
</tr>
<tr>
<td>98..96</td>
<td>03</td>
</tr>
<tr>
<td>103..99</td>
<td>GPR[rt]</td>
</tr>
<tr>
<td>106..104</td>
<td>03</td>
</tr>
<tr>
<td>111..107</td>
<td>GPR[rt]</td>
</tr>
<tr>
<td>114..112</td>
<td>03</td>
</tr>
<tr>
<td>119..115</td>
<td>GPR[rt]</td>
</tr>
<tr>
<td>126..120</td>
<td>07</td>
</tr>
<tr>
<td>127</td>
<td>GPR[rt]</td>
</tr>
</tbody>
</table>
[Overview]

[Detail of word region (31..0)]

Exceptions:

None
PEXTLB  Parallel Extend Lower from Byte  PEXTLB

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PEXTLB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMI0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:  PEXTLB  rd, rs, rt

Purpose:  To extend halfwords from bytes.

Description:  rd ← extend (rs, rt)

The contents of the low-order doubleword in GPR rs are combined with the contents of the low-order doubleword in GPR rt in a byte wide Interleaved operation. The quadword result is placed into GPR rd.

This instruction operates on 128-bit registers.

Operation

GPR[rd]7..0 ← GPR[rt]7..0
GPR[rd]15..8 ← GPR[rs]7..0
GPR[rd]23..16 ← GPR[rt]15..8
GPR[rd]31..24 ← GPR[rs]15..8
GPR[rd]39..32 ← GPR[rt]23..16
GPR[rd]47..40 ← GPR[rs]23..16
GPR[rd]55..48 ← GPR[rt]31..24
GPR[rd]63..56 ← GPR[rs]31..24
GPR[rd]71..64 ← GPR[rt]39..32
GPR[rd]79..72 ← GPR[rs]39..32
GPR[rd]87..80 ← GPR[rt]47..40
GPR[rd]95..88 ← GPR[rs]47..40
GPR[rd]103..96 ← GPR[rt]55..48
GPR[rd]111..104 ← GPR[rs]55..48
GPR[rd]119..112 ← GPR[rt]63..56
GPR[rd]127..120 ← GPR[rs]63..56

Exceptions:

None
PEXTLH  Parallel Extend Lower from Halfword

**Format:**
PEXTLH rd, rs, rt

**Purpose:**
To extend words from halfwords.

**Description:**
rd ← extend (rs, rt)

The contents of the low-order doubleword in GPR rs are combined with the contents of the low-order doubleword in GPR rt in a halfword wide Interleaved operation. The quadword result is placed into GPR rd.

This instruction operates on 128-bit registers.

**Operation**

\[
\begin{align*}
\text{GPR}[rd]_{15..0} & \leftarrow \text{GPR}[rt]_{15..0} \\
\text{GPR}[rd]_{31..16} & \leftarrow \text{GPR}[rs]_{15..0} \\
\text{GPR}[rd]_{47..32} & \leftarrow \text{GPR}[rt]_{31..16} \\
\text{GPR}[rd]_{63..48} & \leftarrow \text{GPR}[rs]_{31..16} \\
\text{GPR}[rd]_{79..64} & \leftarrow \text{GPR}[rt]_{47..32} \\
\text{GPR}[rd]_{95..80} & \leftarrow \text{GPR}[rs]_{47..32} \\
\text{GPR}[rd]_{111..96} & \leftarrow \text{GPR}[rt]_{63..48} \\
\text{GPR}[rd]_{127..112} & \leftarrow \text{GPR}[rs]_{63..48}
\end{align*}
\]

**Exceptions:**
None
PEXTLW  Parallel Extend Lower from Word  PEXTLW

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PEXTLW</td>
<td>MMI0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>10010</td>
<td>001000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**C790**

**Format:**  PEXTLW  rd, rs, rt

**Purpose:**  To extend doublewords from words.

**Description:**  rd ← extend (rs, rt)

The contents of the low-order doubleword in GPR rs are combined with the contents of the low-order doubleword in GPR rt in a word wide Interleaved operation. The quadword result is placed into GPR rd.

This instruction operates on 128-bit registers.

**Operation:**

- GPR[rd]_{31..0} ← GPR[rt]_{31..0}
- GPR[rd]_{63..32} ← GPR[rs]_{31..0}
- GPR[rd]_{95..64} ← GPR[rt]_{63..32}
- GPR[rd]_{127..96} ← GPR[rs]_{63..32}

**Exceptions:**

None
Appendix B  C790-Specific Instruction Set Details

PEXTUB  Parallel Extend Upper from Byte  PEXTUB

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PEXTUB</td>
<td>MMI1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C790

Format:  PEXTUB  rd, rs, rt
Purpose:  To extend halfwords from bytes.
Description:  rd ← extend (rs, rt)

The contents of the high-order doubleword in GPR rs are combined with the contents of the high-order doubleword in GPR rt in a byte wide Interleaved operation. The quadword result is placed into GPR rd.

This instruction operates on 128-bit registers.

Operation:

GPR[rd]7..0 ← GPR[rt]71..64
GPR[rd]15..8 ← GPR[rs]71..64
GPR[rd]23..16 ← GPR[rt]79..72
GPR[rd]31..24 ← GPR[rs]79..72
GPR[rd]39..32 ← GPR[rt]87..80
GPR[rd]47..40 ← GPR[rs]87..80
GPR[rd]55..48 ← GPR[rt]95..88
GPR[rd]63..56 ← GPR[rs]95..88
GPR[rd]71..64 ← GPR[rt]103..96
GPR[rd]79..72 ← GPR[rs]103..96
GPR[rd]87..80 ← GPR[rt]111..104
GPR[rd]95..88 ← GPR[rs]111..104
GPR[rd]103..96 ← GPR[rt]119..112
GPR[rd]111..104 ← GPR[rs]119..112
GPR[rd]119..112 ← GPR[rt]127..120
GPR[rd]127..120 ← GPR[rs]127..120

Exceptions:

None
**PEXTUH**

Parallel Extend Upper from Halfword

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PEXTUH</td>
<td>10110</td>
<td>MMI1</td>
<td>101000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C790

**Format:**

PEXTUH rd, rs, rt

**Purpose:**

To extend words from halfwords.

**Description:**

rd ← extend (rs, rt)

The contents of the high-order doubleword in GPR rs are combined with the contents of the high-order doubleword in GPR rt in a halfword wide Interleaved operation. The quadword result is placed into GPR rd.

This instruction operates on 128-bit registers.

**Operation:**

\[
\begin{align*}
\text{GPR}[rd]_{15..0} & \leftarrow \text{GPR}[rt]_{79..64} \\
\text{GPR}[rd]_{31..16} & \leftarrow \text{GPR}[rs]_{79..64} \\
\text{GPR}[rd]_{47..32} & \leftarrow \text{GPR}[rt]_{95..80} \\
\text{GPR}[rd]_{63..48} & \leftarrow \text{GPR}[rs]_{95..80} \\
\text{GPR}[rd]_{79..64} & \leftarrow \text{GPR}[rt]_{111..96} \\
\text{GPR}[rd]_{95..80} & \leftarrow \text{GPR}[rs]_{111..96} \\
\text{GPR}[rd]_{111..96} & \leftarrow \text{GPR}[rt]_{127..112} \\
\text{GPR}[rd]_{127..112} & \leftarrow \text{GPR}[rs]_{127..112}
\end{align*}
\]

**Exceptions:**

None
**PEXTUW**

Parallel Extend Upper from Word

<table>
<thead>
<tr>
<th>MMI 011100</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>PEXTUW 10010</th>
<th>MMI1 101000</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

**Format:**

PEXTUW rd, rs, rt

**Purpose:**

To extend doublewords from words.

**Description:**

rd ← extend (rs, rt)

The contents of the high-order doubleword in GPR rs are combined with the contents of the high-order doubleword in GPR rt in a word wide Interleaved operation. The quadword result is placed into GPR rd.

This instruction operates on 128-bit registers.

**Operation:**

\[
\begin{align*}
\text{GPR}[rd]_{31..0} & \leftarrow \text{GPR}[rt]_{95..64} \\
\text{GPR}[rd]_{63..32} & \leftarrow \text{GPR}[rs]_{95..64} \\
\text{GPR}[rd]_{95..64} & \leftarrow \text{GPR}[rt]_{127..96} \\
\text{GPR}[rd]_{127..96} & \leftarrow \text{GPR}[rs]_{127..96}
\end{align*}
\]

**Exceptions:**

None
Parallel Horizontal Multiply-Add Halfword

**PHMADH**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PHMADH</td>
<td>10001</td>
<td>MMI2</td>
<td>001001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** PHMADH rd, rs, rt

**Purpose:** To multiply 8 pairs of 16-bit signed integers and horizontally add.

**Description:**

\[(rd, HI, LO) \leftarrow rs \times rt + rs \times rt\]

The eight signed halfwords in GPR rs are multiplied by the eight signed halfwords in GPR rt in parallel. The four word multiply results are added to the other four word multiply results, and the four word results are placed into the corresponding words in special registers HI, LO and GPR rd.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

**Restrictions:**

None

**Operation:**

- \(\text{prod0} \leftarrow \text{GPR}[rs]_{31..16} \times \text{GPR}[rt]_{31..16} + \text{GPR}[rs]_{15..0} \times \text{GPR}[rt]_{15..0}\)
- \(\text{prod1} \leftarrow \text{GPR}[rs]_{63..48} \times \text{GPR}[rt]_{63..48} + \text{GPR}[rs]_{47..32} \times \text{GPR}[rt]_{47..32}\)
- \(\text{prod2} \leftarrow \text{GPR}[rs]_{95..80} \times \text{GPR}[rt]_{95..80} + \text{GPR}[rs]_{79..64} \times \text{GPR}[rt]_{79..64}\)
- \(\text{prod3} \leftarrow \text{GPR}[rs]_{127..112} \times \text{GPR}[rt]_{127..112} + \text{GPR}[rs]_{111..96} \times \text{GPR}[rt]_{111..96}\)
- \(\text{LO}_{31..0} \leftarrow \text{prod0}_{31..0}\)
- \(\text{LO}_{63..32} \leftarrow \text{Undefined}\)
- \(\text{HI}_{31..0} \leftarrow \text{prod1}_{31..0}\)
- \(\text{HI}_{63..32} \leftarrow \text{Undefined}\)
- \(\text{LO}_{95..64} \leftarrow \text{prod2}_{31..0}\)
- \(\text{LO}_{127..96} \leftarrow \text{Undefined}\)
- \(\text{HI}_{95..64} \leftarrow \text{prod3}_{31..0}\)
- \(\text{HI}_{127..96} \leftarrow \text{Undefined}\)
- \(\text{GPR}[rd]_{31..0} \leftarrow \text{prod0}_{31..0}\)
- \(\text{GPR}[rd]_{63..32} \leftarrow \text{prod1}_{31..0}\)
- \(\text{GPR}[rd]_{95..64} \leftarrow \text{prod2}_{31..0}\)
- \(\text{GPR}[rd]_{127..96} \leftarrow \text{prod3}_{31..0}\)
Appendix B  C790-Specific Instruction Set Details

Exceptions:
None

Programming Notes:
In the C790, the integer multiply operation allows other CPU instructions to execute out-of-order. An attempt to read LO or HI registers before the results are written will cause an interlock until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.
## PHMSBH

**Parallel Horizontal Multiply-Subtract Halfword**

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>PHMSBH</th>
<th>MMI2</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**Format:** PHMSBH rd, rs, rt

**Purpose:** To multiply 8 pairs of 16-bit signed integers and horizontally subtract.

**Description:**

\[(rd, HI, LO) \leftarrow rs \times rt - rs \times rt\]

The eight signed halfwords in GPR rs are multiplied by the eight signed halfwords in GPR rt in parallel. The four word multiply results are subtracted from the other four word multiply results, and the four word results are placed into the corresponding words in special registers HI, LO and GPR rd.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

**Restrictions:**

None

**Operation:**

- \(prod0 \leftarrow GPR[rs]_{31..16} \times GPR[rt]_{31..16} - GPR[rs]_{15..0} \times GPR[rt]_{15..0}\)
- \(prod1 \leftarrow GPR[rs]_{63..48} \times GPR[rt]_{63..48} - GPR[rs]_{47..32} \times GPR[rt]_{47..32}\)
- \(prod2 \leftarrow GPR[rs]_{95..80} \times GPR[rt]_{95..80} - GPR[rs]_{79..64} \times GPR[rt]_{79..64}\)
- \(prod3 \leftarrow GPR[rs]_{127..112} \times GPR[rt]_{127..112} - GPR[rs]_{111..96} \times GPR[rt]_{111..96}\)
- \(LO_{31..0} \leftarrow prod0_{31..0}\)
- \(LO_{63..32} \leftarrow Undefined\)
- \(HI_{31..0} \leftarrow prod1_{31..0}\)
- \(HI_{63..32} \leftarrow Undefined\)
- \(LO_{95..64} \leftarrow prod2_{31..0}\)
- \(LO_{127..96} \leftarrow Undefined\)
- \(HI_{95..64} \leftarrow prod3_{31..0}\)
- \(HI_{127..96} \leftarrow Undefined\)
- \(GPR[rd]_{31..0} \leftarrow prod0_{31..0}\)
- \(GPR[rd]_{63..32} \leftarrow prod1_{31..0}\)
- \(GPR[rd]_{95..64} \leftarrow prod2_{31..0}\)
- \(GPR[rd]_{127..96} \leftarrow prod3_{31..0}\)
Exceptions:

None

Programming Notes:

In the C790, the integer multiply operation allows other CPU instructions to execute out-of-order. An attempt to read LO or HI registers before the results are written will wait (interlock) until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.
### PINTEH Parallel Interleave Even Halfword

<table>
<thead>
<tr>
<th>rs</th>
<th>rt</th>
<th>rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td></td>
<td>01010</td>
</tr>
</tbody>
</table>

#### Format:
PINTEH rd, rs, rt

#### Purpose:
To combine halfwords in a halfword wide interleaved operation.

#### Description:
rd ← interleave (rs, rt)

The low-order halfword of the four words in GPR rs are combined with the low-order halfword of the four words in GPR rt in a halfword wide Interleaved operation. The quadword result is placed into GPR rd.

This instruction operates on 128-bit registers.

#### Operation:
- \( \text{GPR[rd]}_{15..0} \leftarrow \text{GPR[rt]}_{15..0} \)
- \( \text{GPR[rd]}_{31..16} \leftarrow \text{GPR[rs]}_{15..0} \)
- \( \text{GPR[rd]}_{47..32} \leftarrow \text{GPR[rt]}_{47..32} \)
- \( \text{GPR[rd]}_{63..48} \leftarrow \text{GPR[rs]}_{47..32} \)
- \( \text{GPR[rd]}_{79..64} \leftarrow \text{GPR[rt]}_{79..64} \)
- \( \text{GPR[rd]}_{95..80} \leftarrow \text{GPR[rs]}_{79..64} \)
- \( \text{GPR[rd]}_{111..96} \leftarrow \text{GPR[rt]}_{111..96} \)
- \( \text{GPR[rd]}_{127..112} \leftarrow \text{GPR[rs]}_{111..96} \)

#### Exceptions:
None
PINTH
Parallel Interleave Halfword

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>PINTH</th>
<th>MMI2</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>01010</td>
<td>001001</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: PINTH rd, rs, rt
Purpose: To combine doublewords in a halfword wide interleaved operation.
Description: rd ← interleave (rs, rt)

The contents of the high-order doubleword in GPR rs are combined with the contents of the low-order doubleword in GPR rt in a halfword wide Interleaved operation. The quadword result is placed into GPR rd.

This instruction operates on 128-bit registers.

Operation:
GPR[rd]15.0 ← GPR[rt]15.0
GPR[rd]31.16 ← GPR[rs]79.64
GPR[rd]47.32 ← GPR[rt]31.16
GPR[rd]63.48 ← GPR[rs]95.80
GPR[rd]79.64 ← GPR[rt]47.32
GPR[rd]95.80 ← GPR[rs]111.96
GPR[rd]111.96 ← GPR[rt]63.48
GPR[rd]127.112 ← GPR[rs]127..112

Exceptions:
None
**PLZCW**  
Parallel Leading Zero or one Count Word

### Format:
```
PLZCW  rd, rs
```

### Purpose:
To count leading zero (s) or one (s) (2 parallel operations).

### Description:
```
rd ← LZC (rs) − 1
```

The number of leading zeros or ones of the two words in GPR `rs` are counted. The results of the leading counts minus one are loaded in the corresponding words in GPR `rd`.

### Operation:
- GPR[rd]31..0 ← Leading zero or one count (GPR[rs]31..0) − 1
- GPR[rd]63..32 ← Leading zero or one count (GPR[rs]63..32) − 1

### Example:
```
rs

```

```
0x000FFFFF  0xFF000000
```

```
rd

```

```
0x0000000B  0x00000007
```

### Exceptions:
None
PMADDH Parallel Multiply-Add Halfword

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>PMADDH</th>
<th>MMI2</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>10000</td>
<td>001001</td>
</tr>
</tbody>
</table>

Format: PMADDH rd, rs, rt

Purpose: To multiply 8 pairs of 16-bit signed integers and accumulate, in parallel.

Description: \((rd, HI, LO) \leftarrow (HI, LO) + rs \times rt\)

The eight signed halfwords in GPR rs are multiplied by the eight signed halfwords in GPR rt in parallel. The eight word multiply results are added to the corresponding words in special registers HI and LO, and the word results are placed into the corresponding words in special registers HI, LO and GPR rd.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

Restrictions:

None

Operation:

\[
\begin{align*}
\text{prod0} & \leftarrow \text{LO}_{31.0} + \text{GPR[rs]}_{15.0} \times \text{GPR[rt]}_{15.0} \\
\text{prod1} & \leftarrow \text{LO}_{63.32} + \text{GPR[rs]}_{31.16} \times \text{GPR[rt]}_{31.16} \\
\text{prod2} & \leftarrow \text{HI}_{31.0} + \text{GPR[rs]}_{47.32} \times \text{GPR[rt]}_{47.32} \\
\text{prod3} & \leftarrow \text{HI}_{63.32} + \text{GPR[rs]}_{63.48} \times \text{GPR[rt]}_{63.48} \\
\text{prod4} & \leftarrow \text{LO}_{95.64} + \text{GPR[rs]}_{79.64} \times \text{GPR[rt]}_{79.64} \\
\text{prod5} & \leftarrow \text{LO}_{127.96} + \text{GPR[rs]}_{95.80} \times \text{GPR[rt]}_{95.80} \\
\text{prod6} & \leftarrow \text{HI}_{95.64} + \text{GPR[rs]}_{111.96} \times \text{GPR[rt]}_{111.96} \\
\text{prod7} & \leftarrow \text{HI}_{127.96} + \text{GPR[rs]}_{127.112} \times \text{GPR[rt]}_{127.112} \\
\text{LO}_{31.0} & \leftarrow \text{prod0}_{31.0} \\
\text{LO}_{63.32} & \leftarrow \text{prod1}_{31.0} \\
\text{HI}_{31.0} & \leftarrow \text{prod2}_{31.0} \\
\text{HI}_{63.32} & \leftarrow \text{prod3}_{31.0} \\
\text{LO}_{95.64} & \leftarrow \text{prod4}_{31.0} \\
\text{LO}_{127.96} & \leftarrow \text{prod5}_{31.0} \\
\text{HI}_{95.64} & \leftarrow \text{prod6}_{31.0} \\
\text{HI}_{127.96} & \leftarrow \text{prod7}_{31.0} \\
\text{GPR[rd]}_{31.0} & \leftarrow \text{prod0}_{31.0} \\
\text{GPR[rd]}_{63.32} & \leftarrow \text{prod1}_{31.0} \\
\text{GPR[rd]}_{95.64} & \leftarrow \text{prod4}_{31.0} \\
\text{GPR[rd]}_{127.96} & \leftarrow \text{prod7}_{31.0}
\end{align*}
\]
Appendix B  C790-Specific Instruction Set Details

<table>
<thead>
<tr>
<th>rs</th>
<th>127</th>
<th>112</th>
<th>111</th>
<th>96</th>
<th>95</th>
<th>80</th>
<th>79</th>
<th>64</th>
<th>63</th>
<th>48</th>
<th>47</th>
<th>32</th>
<th>31</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A7</td>
<td>A6</td>
<td>A5</td>
<td>A4</td>
<td>A3</td>
<td>A2</td>
<td>A1</td>
<td>A0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rt</th>
<th>127</th>
<th>112</th>
<th>111</th>
<th>96</th>
<th>95</th>
<th>80</th>
<th>79</th>
<th>64</th>
<th>63</th>
<th>48</th>
<th>47</th>
<th>32</th>
<th>31</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>B7</td>
<td>B6</td>
<td>B5</td>
<td>B4</td>
<td>B3</td>
<td>B2</td>
<td>B1</td>
<td>B0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HI</th>
<th>127</th>
<th>96</th>
<th>95</th>
<th>64</th>
<th>63</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C7</td>
<td>C6</td>
<td>C3</td>
<td>C2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LO</th>
<th>127</th>
<th>96</th>
<th>95</th>
<th>64</th>
<th>63</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C5</td>
<td>C4</td>
<td>C1</td>
<td>C0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rd</th>
<th>127</th>
<th>96</th>
<th>95</th>
<th>64</th>
<th>63</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A6 × B6 + C6</td>
<td>A4 × B4 + C4</td>
<td>A2 × B2 + C2</td>
<td>A0 × B0 + C0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HI</th>
<th>127</th>
<th>96</th>
<th>95</th>
<th>64</th>
<th>63</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A7 × B7 + C7</td>
<td>A6 × B6 + C6</td>
<td>A3 × B3 + C3</td>
<td>A2 × B2 + C2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LO</th>
<th>127</th>
<th>96</th>
<th>95</th>
<th>64</th>
<th>63</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A5 × B5 + C5</td>
<td>A4 × B4 + C4</td>
<td>A1 × B1 + C1</td>
<td>A0 × B0 + C0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Exceptions:**

None

**Programming Notes:**

In the C790, the integer multiply operation allow other CPU instructions to execute out-of-order. An attempt to read LO or HI registers before the results are written will cause an interlock until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.
PMADDUW Parallel Multiply-Add Unsigned Word

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PMADDUW</td>
<td>MMI3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>00000</td>
<td>101001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** PMADDUW rd, rs, rt

**Purpose:** To multiply 2 pairs of 32-bit unsigned integers and accumulate in parallel.

**Description:**
\[(rd, HI, LO) \leftarrow (HI, LO) + rs \times rt\]

The low-order unsigned word of the two doublewords in GPR rs are multiplied by the low-order unsigned word of the two doublewords in GPR rt in parallel. The two 64-bit multiply results are added to the contents of special registers HI and LO. The low-order word of the two doubleword results are placed into special register LO, and the high-order word of the two doubleword results are placed into special register HI. The two doubleword results are placed into GPR rd.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

**Restrictions:**
If either GPR rt or GPR rs do not contain zero-extended 32-bit values (bits 127..96 and 63..32 equal zero) then the result of the equation will be undefined.

**Operation:**
if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
prod0 ⟵ (HI\_{31..0} || LO\_{31..0}) + (0 || GPR[rs]_{31..0}) \times (0 || GPR[rt]_{31..0})
prod1 ⟵ (HI\_{95..64} || LO\_{95..64}) + (0 || GPR[rs]_{95..64}) \times (0 || GPR[rt]_{95..64})
LO\_{63..0} ⟵ (prod0pery01..32 || prod0_{31..0})
HI\_{63..0} ⟵ (prod0_{63..32} || prod0_{63..32})
LO\_{127..64} ⟵ (prod1_{31..32} || prod1_{31..0})
HI\_{127..64} ⟵ (prod1_{63..32} || prod1_{63..32})
GPR[rd]_{63..0} ⟵ prod0_{63..0}
GPR[rd]_{127..64} ⟵ prod1_{63..0}
### Exceptions:

None

### Programming Notes:

See the Programming Notes for the PMADDH instruction.
PMADDW

Parallel Multiply-Add Word

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PMADDW</td>
<td>MMI2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>00000</td>
<td>001001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: PMADDW rd, rs, rt

Purpose: To multiply 2 pairs of 32-bit signed integers and accumulate in parallel.

Description: 

(rd, HI, LO) ← (HI, LO) + rs × rt

The low-order signed word of the two doublewords in GPR rs are multiplied by the low-
order signed word of the two doublewords in GPR rt in parallel. The two 64-bit multiply
results are added to the contents of special registers HI and LO. The low-order word of the
two doubleword results are placed into special register LO, and the high-order word of the
two doubleword results are placed into special register HI. The two doubleword results are
placed into GPR rd.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

Restrictions:

If either GPR rt or GPR rs do not contain sign-extended 32-bit values (bits 127..95 and
63..31 equal) then the result of the equation will be undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
prod0 ← (HI31.0 || LO31.0) + GPR[rs]31..0 × GPR[rt]31..0
prod1 ← (HI95.64 || LO95.64) + GPR[rs]95..64 × GPR[rt]95..64
LO63.0 ← (prod0 31)|| prod031..0
HI63.0 ← (prod0 63)|| prod063..32
LO127.64 ← (prod1 31)|| prod131..0
HI127.64 ← (prod1 63)|| prod163..32
GPR[rd]63..0 ← prod063.0
GPR[rd]127..64 ← prod163.0
### Appendix B  C790-Specific Instruction Set Details

#### pmaddh

<table>
<thead>
<tr>
<th>rs</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
<th>64</th>
<th>63</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt</td>
<td>B3</td>
<td>B2</td>
<td>B1</td>
<td>B0</td>
<td>64</td>
<td>63</td>
<td>32</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>HI</td>
<td>C7</td>
<td>C6</td>
<td>C3</td>
<td>C2</td>
<td>64</td>
<td>63</td>
<td>32</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>LO</td>
<td>C5</td>
<td>C4</td>
<td>C1</td>
<td>C0</td>
<td>64</td>
<td>63</td>
<td>32</td>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Exceptions:

None

#### Programming Notes:

See the Programming Notes for the PMADDH instruction.
### PMAXH

**Parallel Maximum Halfword**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PMAXH</td>
<td>00111</td>
<td>MI0</td>
<td>001000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** PMAXH rd, rs, rt

**Purpose:** To select maximum 16-bit signed integers (8 parallel operations).

**Description:**

rd ← max (rs, rt)

The eight signed halfword values in GPR rt are subtracted from the corresponding eight signed halfword values in GPR rs in parallel. If the result of subtraction is larger than zero, the corresponding signed halfword value in GPR rs is placed into the corresponding halfword in GPR rd otherwise the corresponding signed halfword value in GPR rt is placed into the corresponding halfword of the GPR rd.

This instruction operates on 128-bit registers.

**Operation:**

if ((GPR[rs]15..0 − GPR[rt]15..0) > 0) then
    GPR[rd]15..0 ← GPR[rs]15..0
else
    GPR[rd]15..0 ← GPR[rt]15..0
endif

if ((GPR[rs]31..16 − GPR[rt]31..16) > 0) then
    GPR[rd]31..16 ← GPR[rs]31..16
else
    GPR[rd]31..16 ← GPR[rt]31..16
endif

if ((GPR[rs]47..32 − GPR[rt]47..32) > 0) then
    GPR[rd]47..32 ← GPR[rs]47..32
else
    GPR[rd]47..32 ← GPR[rt]47..32
endif

if ((GPR[rs]63..48 − GPR[rt]63..48) > 0) then
    GPR[rd]63..48 ← GPR[rs]63..48
else
    GPR[rd]63..48 ← GPR[rt]63..48
endif

if ((GPR[rs]79..64 − GPR[rt]79..64) > 0) then
    GPR[rd]79..64 ← GPR[rs]79..64
else
    GPR[rd]79..64 ← GPR[rt]79..64
endif
if ((GPR[rs]₉₅..₈₀ − GPR[rt]₉₅..₈₀) > 0) then
  GPR[rd]₉₅..₈₀ ← GPR[rs]₉₅..₈₀
else
  GPR[rd]₉₅..₈₀ ← GPR[rt]₉₅..₈₀
endif

if ((GPR[rs]₁₁₁..₉₆ − GPR[rt]₁₁₁..₉₆) > 0) then
  GPR[rd]₁₁₁..₉₆ ← GPR[rs]₁₁₁..₉₆
else
  GPR[rd]₁₁₁..₉₆ ← GPR[rt]₁₁₁..₉₆
endif

if ((GPR[rs]₁₂₇..₁₁₂ − GPR[rt]₁₂₇..₁₁₂) > 0) then
  GPR[rd]₁₂₇..₁₁₂ ← GPR[rs]₁₂₇..₁₁₂
else
  GPR[rd]₁₂₇..₁₁₂ ← GPR[rt]₁₂₇..₁₁₂
endif

**Exceptions:**

None
PMAXW  Parallel Maximum Word  PMAXW

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>PMAXW</th>
<th>MMI0</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>00011</td>
<td>001000</td>
</tr>
</tbody>
</table>

Format:  PMAXW  rd, rs, rt
Purpose:  To select maximum 32-bit signed integers (4 parallel operations).
Description:  rd ← max (rs, rt)

The four signed word values in GPR rt are subtracted from the corresponding four signed word values in GPR rs in parallel. If the result of subtraction is larger than zero, the corresponding signed word value in GPR rs is placed into the corresponding word in GPR rd otherwise the corresponding signed word value in GPR rt is placed into the corresponding word of the GPR rd.

This instruction operates on 128-bit registers.

Operation:
if ((GPR[rs]31..0 − GPR[rt]31..0) > 0) then
    GPR[rd]31..0 ← GPR[rs]31..0
else
    GPR[rd]31..0 ← GPR[rt]31..0
endif

if ((GPR[rs]63..32 − GPR[rt]63..32) > 0) then
    GPR[rd]63..32 ← GPR[rs]63..32
else
    GPR[rd]63..32 ← GPR[rt]63..32
endif

if ((GPR[rs]95..64 − GPR[rt]95..64) > 0) then
    GPR[rd]95..64 ← GPR[rs]95..64
else
    GPR[rd]95..64 ← GPR[rt]95..64
endif

if ((GPR[rs]127..96 − GPR[rt]127..96) > 0) then
    GPR[rd]127..96 ← GPR[rs]127..96
else
    GPR[rd]127..96 ← GPR[rt]127..96
endif
#### Appendix B  C790-Specific Instruction Set Details

<table>
<thead>
<tr>
<th></th>
<th>127</th>
<th>96</th>
<th>95</th>
<th>64</th>
<th>63</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rs</strong></td>
<td>A3</td>
<td></td>
<td></td>
<td>A2</td>
<td></td>
<td></td>
<td>A1</td>
<td>A0</td>
</tr>
<tr>
<td><strong>rt</strong></td>
<td></td>
<td>B3</td>
<td></td>
<td>B2</td>
<td></td>
<td></td>
<td>B1</td>
<td>B0</td>
</tr>
<tr>
<td><strong>rd</strong></td>
<td>max (A3, B3)</td>
<td>max (A2, B2)</td>
<td>max (A1, B1)</td>
<td>max (A0, B0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Exceptions:**

None
**PMFHI**

Parallel Move From HI Register

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>0</td>
<td>rd</td>
<td>PMFHI</td>
<td>01000</td>
<td>MMI2</td>
<td>001001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** PMFHI rd

**Purpose:** To copy the special purpose register HI to a GPR.

**Description:** rd ← HI

The contents of special register HI are loaded into GPR rd.

This instruction operates on 128-bit registers.

**Restrictions:** None

**Operation:**

\[
\text{GPR}[\text{rd}]_{127..0} \leftarrow \text{HI}_{127..0}
\]

**Exceptions:** None
PMFHL.fmt  Parallel Move From HI / LO Register  PMFHL.fmt

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>PMFHL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>0</td>
<td>0000000000</td>
<td>rd</td>
<td>fmt</td>
<td>110000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C790

Format: 
PMFHL.LW  rd (fmt = 0)
PMFHL.UW  rd (fmt = 1)
PMFHL.SLW rd (fmt = 2)
PMFHL.LH  rd (fmt = 3)
PMFHL.SH  rd (fmt = 4)

Purpose: 
To copy the special purpose registers HI / LO to a GPR.

Description: 
rd ← HI / LO

The contents of special registers HI / LO are loaded into GPR rd.

This instruction operates on 128-bit registers.

Restrictions:
None

Operation:

if (fmt = 0) then
    GPR[rd]31..0 ← LO31..0
    GPR[rd]63..32 ← HI31..0
    GPR[rd]95.64 ← LO95.64
    GPR[rd]127.96 ← HI95.64
else if (fmt = 1) then
    GPR[rd]31..0 ← LO63..32
    GPR[rd]63..32 ← HI63..32
    GPR[rd]95.64 ← LO127..96
    GPR[rd]127.96 ← HI127.96
else if (fmt = 2) then
    if ((HI31..0 || LO31..0) > 0x0000000007FFFFFF) then
        GPR[rd]63.0 ← 0x0000000007FFFFFF
    else if ((HI31..0 || LO31..0) < -0x0000000080000000) then
        GPR[rd]63.0 ← 0xFFFFFFFF80000000
    else
        GPR[rd]63.0 ← HI31..0 || LO31..0
    endif
    if ((HI95..64 || LO95..64) > 0x0000000007FFFFFF) then
        GPR[rd]127..64 ← 0x0000000007FFFFFF
    else if ((HI95..64 || LO95..64) < -0x000000008000000000) then
        GPR[rd]127..64 ← -0x000000008000000000
    else
        GPR[rd]127..64 ← (LO95)32 || LO95.64
    endif
else if (fmt = 3) then
    GPR[rd]15..0 ← LO15..0
else if (fmt = 4) then
  if (0x7FFFFFF >= LO31..0 > 0x00007FFF) then
    GPR[rd]15..0 ← 0x7FFF
  else if (0x80000000 <= LO31..0 < 0xFFFF8000) then
    GPR[rd]15..0 ← 0x8000
  else
    GPR[rd]15..0 ← LO15..0
  endif
  if (LO63..32 > 0x00007FFF) then
    GPR[rd]31..16 ← 0x7FFF
  else if (LO63..32 < 0xFFFF8000) then
    GPR[rd]31..16 ← 0x8000
  else
    GPR[rd]31..16 ← LO47..32
  endif
  if (HI31..0 > 0x00007FFF) then
    GPR[rd]47..32 ← 0x7FFF
  else if (HI31..0 < 0xFFFF8000) then
    GPR[rd]47..32 ← 0x8000
  else
    GPR[rd]47..32 ← HI15..0
  endif
  if (HI63..32 > 0x00007FFF) then
    GPR[rd]63..48 ← 0x7FFF
  else if (HI63..32 < 0xFFFF8000) then
    GPR[rd]63..48 ← 0x8000
  else
    GPR[rd]63..48 ← HI47..32
  endif
  if (LO95..64 > 0x00007FFF) then
    GPR[rd]79..64 ← 0x7FFF
  else if (LO95..64 < -0xFFFF8000) then
    GPR[rd]79..64 ← 0x8000
  else
    GPR[rd]79..64 ← LO79..64
  endif
  if (LO127..96 > 0x00007FFF) then
    GPR[rd]95..80 ← 0x7FFF
  else if (LO127..96 < 0xFFFF8000) then
    GPR[rd]95..80 ← 0x8000
  else
    GPR[rd]95..80 ← LO111..96
  endif
  if (HI95..64 > 0x00007FFF) then
    GPR[rd]111..96 ← 0x7FFF
  else if (HI95..64 < 0xFFFF8000) then
    GPR[rd]111..96 ← 0x8000
  else
    GPR[rd]111..96 ← HI79..64
  endif

else
    GPR[rd]_{111..96} ← HI_{79..64}
endif
if (HI_{127..96} > 0x00007FFF) then
    GPR[rd]_{127..112} ← 0x7FFF
else if (HI_{127..96} < 0xFFFF8000) then
    GPR[rd]_{127..112} ← 0x8000
else
    GPR[rd]_{127..112} ← HI_{111..96}
endif
endif

(fmt = 0)

(fmt = 1)

(fmt = 2)
Exceptions:

None
PMFLO  Parallel Move From LO Register  PMFLO

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>0</td>
<td>0000000000</td>
<td>rd</td>
<td>PMFLO</td>
<td>01001</td>
<td>MMI2</td>
<td>001001</td>
<td></td>
</tr>
</tbody>
</table>

C790

Format: PMFLO rd

Purpose: To copy the special purpose register LO to a GPR.

Description: rd ← LO

The contents of special register LO are loaded into GPR rd.

This instruction operates on 128-bit registers.

Restrictions:
None

Operation:

GPR[rd]_{127..0} ← LO_{127..0}

Exceptions:
None
**PMINH**

**Parallel Minimum Halfword**

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>6 5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PMINH</td>
<td>MMI1</td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td></td>
<td></td>
<td></td>
<td>001111</td>
<td>101000</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** PMINH rd, rs, rt

**Purpose:** To select the minimum of two 16-bit signed integers (8 parallel operations).

**Description:** rd ← min (rs, rt)

The eight signed halfword values in GPR rt are subtracted from the corresponding eight signed halfword values in GPR rs in parallel. If the result of each subtraction is larger than zero, the corresponding signed halfword in GPR rt is placed into the corresponding halfword in GPR rd otherwise the corresponding signed halfword in GPR rs is placed into the corresponding halfword of GPR rd.

This instruction operates on 128-bit registers.

**Operation:**

```plaintext
if ((GPR[rs]15..0 - GPR[rt]15..0) > 0) then
gpr[rd]15..0 ← GPR[rt]15..0
else
  gpr[rd]15..0 ← GPR[rs]15..0
endif
if ((GPR[rs]31..16 - GPR[rt]31..16) > 0) then
  gpr[rd]31..16 ← GPR[rt]31..16
else
  gpr[rd]31..16 ← GPR[rs]31..16
endif
if ((GPR[rs]47..32 - GPR[rt]47..32) > 0) then
  gpr[rd]47..32 ← GPR[rt]47..32
else
  gpr[rd]47..32 ← GPR[rs]47..32
endif
if ((GPR[rs]63..48 - GPR[rt]63..48) > 0) then
  gpr[rd]63..48 ← GPR[rt]63..48
else
  gpr[rd]63..48 ← GPR[rs]63..48
endif
if ((GPR[rs]79..64 - GPR[rt]79..64) > 0) then
  gpr[rd]79..64 ← GPR[rt]79..64
else
  gpr[rd]79..64 ← GPR[rs]79..64
endif
if ((GPR[rs]95..80 - GPR[rt]95..80) > 0) then
  gpr[rd]95..80 ← GPR[rt]95..80
else
  gpr[rd]95..80 ← GPR[rs]95..80
endif
```
if ((GPR[rs]_{111..96} − GPR[rt]_{111..96}) > 0) then
    GPR[rd]_{111..96} ← GPR[rt]_{111..96}
else
    GPR[rd]_{111..96} ← GPR[rs]_{111..96}
endif

if ((GPR[rs]_{127..112} − GPR[rt]_{127..112}) > 0) then
    GPR[rd]_{127..112} ← GPR[rt]_{127..112}
else
    GPR[rd]_{127..112} ← GPR[rs]_{127..112}
endif

<table>
<thead>
<tr>
<th>rs</th>
<th>A7</th>
<th>A6</th>
<th>A5</th>
<th>A4</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt</td>
<td>B7</td>
<td>B6</td>
<td>B5</td>
<td>B4</td>
<td>B3</td>
<td>B2</td>
<td>B1</td>
<td>B0</td>
</tr>
<tr>
<td>rd</td>
<td>min (A7, B7)</td>
<td>min (A6, B6)</td>
<td>min (A5, B5)</td>
<td>min (A4, B4)</td>
<td>min (A3, B3)</td>
<td>min (A2, B2)</td>
<td>min (A1, B1)</td>
<td>min (A0, B0)</td>
</tr>
</tbody>
</table>

Exceptions:
None
PMINW  Parallel Minimum Word  PMINW

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>6  5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PMINW</td>
<td>MMI1</td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td></td>
<td></td>
<td></td>
<td>00011</td>
<td>101000</td>
<td></td>
</tr>
</tbody>
</table>

6  5  5  5  5  6

C790

Format:  PMINW rd, rs, rt
Purpose:  To select the minimum of two 32-bit signed integers (4 parallel operations).
Description:  rd ← min (rs, rt)

The four signed word values in GPR rt are subtracts from the corresponding four signed word values in GPR rs, in parallel. If the result of each subtraction is larger than zero, the corresponding signed word value in GPR rt is placed into the corresponding word of GPR rd otherwise the corresponding signed word value in GPR rs is placed into the corresponding word of GPR rd.

This instruction operates on 128-bit registers.

Operation:
if ((GPR[rs]31..0 − GPR[rt]31..0) > 0) then
  GPR[rd]31..0 ← GPR[rt]31..0
else
  GPR[rd]31..0 ← GPR[rs]31..0
endif

if ((GPR[rs]63..32 − GPR[rt]63..32) > 0) then
  GPR[rd]63..32 ← GPR[rt]63..32
else
  GPR[rd]63..32 ← GPR[rs]63..32
endif

if ((GPR[rs]95..64 − GPR[rt]95..64) > 0) then
  GPR[rd]95..64 ← GPR[rt]95..64
else
  GPR[rd]95..64 ← GPR[rs]95..64
endif

if ((GPR[rs]127..96 − GPR[rt]127..96) > 0) then
  GPR[rd]127..96 ← GPR[rt]127..96
else
  GPR[rd]127..96 ← GPR[rs]127..96
endif
### Appendix B  C790-Specific Instruction Set Details

#### Exceptions:

None

<table>
<thead>
<tr>
<th>rs</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>96</td>
<td>95</td>
<td>64</td>
<td>63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rt</th>
<th>B3</th>
<th>B2</th>
<th>B1</th>
<th>B0</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>96</td>
<td>95</td>
<td>64</td>
<td>63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rd</th>
<th>min (A3, B3)</th>
<th>min (A2, B2)</th>
<th>min (A1, B1)</th>
<th>min (A0, B0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>96</td>
<td>95</td>
<td>64</td>
<td>63</td>
</tr>
</tbody>
</table>
### PMSUBH

**Parallel Multiply-Subtract Halfword**

| Format: | PMSUBH rd, rs, rt |
| Purpose: | To multiply 8 pairs of 16-bit signed integers and subtract in parallel. |
| Description: | (rd, HI, LO) ← (HI, LO) − rs × rt |

The eight signed halfwords in GPR rs are multiplied by the eight signed halfwords in GPR rt in parallel. The eight word multiply results are subtracted from the corresponding words in special registers HI and LO, and the word results are placed into the corresponding words in special registers HI, LO and GPR rd.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

### Restrictions:

None

#### Operation:

- prod0 $\leftarrow$ LO\_31..0 − GPR[rs]\_15..0 × GPR[rt]\_15..0
- prod1 $\leftarrow$ LO\_63..32 − GPR[rs]\_31..16 × GPR[rt]\_31..16
- prod2 $\leftarrow$ HI\_31..0 − GPR[rs]\_47..32 × GPR[rt]\_47..32
- prod3 $\leftarrow$ HI\_63..32 − GPR[rs]\_63..48 × GPR[rt]\_63..48
- prod4 $\leftarrow$ LO\_95..64 − GPR[rs]\_79..64 × GPR[rt]\_79..64
- prod5 $\leftarrow$ LO\_127..96 − GPR[rs]\_95..80 × GPR[rt]\_95..80
- prod6 $\leftarrow$ HI\_95..64 − GPR[rs]\_111..96 × GPR[rt]\_111..96
- prod7 $\leftarrow$ HI\_127..96 − GPR[rs]\_127..112 × GPR[rt]\_127..112
- LO\_31..0 $\leftarrow$ prod0\_31..0
- LO\_63..32 $\leftarrow$ prod1\_31..0
- HI\_31..0 $\leftarrow$ prod2\_31..0
- HI\_63..32 $\leftarrow$ prod3\_31..0
- LO\_95..64 $\leftarrow$ prod4\_31..0
- LO\_127..96 $\leftarrow$ prod5\_31..0
- HI\_95..64 $\leftarrow$ prod6\_31..0
- HI\_127..96 $\leftarrow$ prod7\_31..0
- GPR[rd]\_31..0 $\leftarrow$ prod0\_31..0
- GPR[rd]\_63..32 $\leftarrow$ prod1\_31..0
- GPR[rd]\_95..64 $\leftarrow$ prod4\_31..0
- GPR[rd]\_127..96 $\leftarrow$ prod7\_31..0

C790
**Appendix B  C790-Specific Instruction Set Details**

<table>
<thead>
<tr>
<th>rs</th>
<th>A7</th>
<th>A6</th>
<th>A5</th>
<th>A4</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>127</td>
<td>112</td>
<td>111</td>
<td>96</td>
<td>95</td>
<td>80</td>
<td>79</td>
<td>64</td>
</tr>
<tr>
<td>rt</td>
<td>B7</td>
<td>B6</td>
<td>B5</td>
<td>B4</td>
<td>B3</td>
<td>B2</td>
<td>B1</td>
<td>B0</td>
</tr>
<tr>
<td></td>
<td>127</td>
<td>112</td>
<td>111</td>
<td>96</td>
<td>95</td>
<td>80</td>
<td>79</td>
<td>64</td>
</tr>
<tr>
<td>HI</td>
<td>C7</td>
<td>C6</td>
<td>C3</td>
<td>C2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>127</td>
<td>96</td>
<td>95</td>
<td>64</td>
<td>63</td>
<td>32</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>LO</td>
<td>C5</td>
<td>C4</td>
<td>C1</td>
<td>C0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Exceptions:**

None

**Programming Notes:**

See the Programming Notes for the PMADDH instruction.
Appendix B  C790-Specific Instruction Set Details

PMSUBW  Parallel Multiply-Subtract Word

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PMSUBW</td>
<td>00100</td>
<td>MMI2</td>
<td>0010001</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6 | 5 | 5 | 5 | 5 | 6 |

C790

Format:  
PMSUBW  rd, rs, rt

Purpose:  
To multiply 2 pairs of 32-bit signed integers and subtract in parallel.

Description:

(rd, HI, LO) ← (HI, LO) − rs × rt

The low-order signed words of the two doublewords in GPR rs are multiplied by the low-order signed words of the two doublewords in GPR rt in parallel. The two 64-bit multiply results are subtracted from the contents of special registers HI and LO. The low-order word of the two doubleword results are placed into special register LO, and the high-order word of the two doubleword results are placed into special register HI. The two doubleword results are placed into GPR rd.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

Restrictions:

If either GPR rt or GPR rs do not contain sign-extended 32-bit values (bits 127..95 and 63..31 equal) then the result of the equation will be undefined.

Operation:

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
prod0 ← (HI31..0 || LO31..0) − GPR[rs]31..0 × GPR[rt]31..0
prod1 ← (HI95..64 || LO95..64) − GPR[rs]95..64 × GPR[rt]95..64
LO63..0 ← (prod031)32 || prod031..0
HI63..0 ← (prod032 || prod063..32
LO127..64 ← (prod131)32 || prod131..0
HI127..64 ← (prod132 || prod163..32
GPR[rd]63..0 ← prod063..0
GPR[rd]127..64 ← prod163..0
Appendix B  C790-Specific Instruction Set Details

rs                   A3                                   A2                                   A1                                    A0
      |                    |                                   |                                   |                                    |  
57 96 95                                  64 63                                 32 31                                    0

rt                    B3                                   B2                                   B1                                    B0
      |                    |                                   |                                   |                                    |  
57 96 95                                  64 63                                 32 31                                    0

HI                   C7                                   C6                                   C3                                    C2
      |                    |                                   |                                   |                                    |  
57 96 95                                  64 63                                 32 31                                    0

LO                   C5                                   C4                                   C1                                    C0
      |                    |                                   |                                   |                                    |  

rd                          (C6 || C4) − A2 × B2                                  (C2 || C0) − A0 × B0
      |                                           |                                           |                                    |  
57 96 95                                  64 63                                    0

HI                   sign ext                ((C6 || C4) − A2 × B2)63..32                        sign ext             ((C2 || C0) − A0 × B0)63..32
      |                                           |                                           |                                    |  
57 96 95                                  64 63                                    0

LO                   sign ext                ((C6 || C4) − A2 × B2)31..0                        sign ext             ((C2 || C0) − A0 × B0)31..0
      |                                           |                                           |                                    |  

Exceptions:
None

Programming Notes:
See the Programming Notes for the PMADDH instruction.
PMTHI  Parallel Move To HI Register  PMTHI

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>0</td>
<td>0000000000</td>
<td>PMTHI</td>
<td>01000</td>
<td>MMI3</td>
<td>101001</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6  5  10  5  6  

Format:       PMTHI  rs
Purpose:      To copy a GPR to the special purpose register HI.
Description:  HI ← rs

The contents of GPR rs are loaded into special register HI.

This instruction operates on 128-bit registers.

Restrictions: None

Operation:

HI_{127..0} ← GPR[rs]_{127..0}

Exceptions:

None
**PMTHL.fmt**

*Parallel Move To HI / LO Register*

| 31 | 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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| MMI | rs | 0  | 0000000000 | fmt | PMTHL | 110001 |

**Format:**

PMTHL.LW rs (fmt = 0)

**Purpose:**

To copy a GPR to the special registers HI / LO.

**Description:**

\[
\text{HI / LO} \leftarrow \text{rs}
\]

The contents of GPR \( r_d \) are loaded into special register \( HI / LO \).

This instruction operates on 128-bit registers.

**Restrictions:**

None

**Operation:**

\[
\text{if (fmt = 0) then}
\]

\[
\begin{align*}
\text{LO}_{31.0} & \leftarrow \text{GPR}[rs]_{31.0} \\
\text{LO}_{63.32} & \leftarrow \text{LO}_{63.32} \\
\text{HI}_{31.0} & \leftarrow \text{GPR}[rs]_{63.32} \\
\text{HI}_{63.32} & \leftarrow \text{HI}_{63.32} \\
\text{LO}_{95.64} & \leftarrow \text{GPR}[rs]_{95.64} \\
\text{LO}_{127.96} & \leftarrow \text{LO}_{127.96} \\
\text{HI}_{95.64} & \leftarrow \text{GPR}[rs]_{127.96} \\
\text{HI}_{127.96} & \leftarrow \text{HI}_{127.96}
\end{align*}
\]

\[
\text{endif}
\]

**Exceptions:**

None
Parallel Move To LO Register

**PMTLO**

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>0</th>
<th>PMTLO</th>
<th>MMI3</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>0000000000</td>
<td>01001</td>
<td>101001</td>
<td></td>
</tr>
</tbody>
</table>

**Format:** PMTLO rs

**Purpose:** To copy a GPR to the special register LO.

**Description:**

\[
\text{LO} \leftarrow \text{rs}
\]

The contents of GPR rs are loaded into special register LO.

This instruction operates on 128-bit registers.

**Restrictions:** None

**Operation:**

\[
\text{LO}_{127..0} \leftarrow \text{GPR}[\text{rs}]_{127..0}
\]

**Exceptions:** None
PMULTH  Parallel Multiply Halfword  PMULTH

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PMULTH</td>
<td>11100</td>
<td>MMI2</td>
<td>001001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** PMULTH  rd, rs, rt

**Purpose:** To multiply 8 pairs of 16-bit signed integers in parallel.

**Description:**

\[(rd, LO, HI) \leftarrow rs \times rt\]

The eight signed halfwords in GPR rs are multiplied by the eight signed halfwords in GPR rt, in parallel. The eight word results are placed into special register HI, LO and GPR rd.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

**Restrictions:**

None

**Operation:**

\[
\begin{align*}
prod0 & \leftarrow GPR[rs]_{15.0} \times GPR[rt]_{15.0} \\
nod0 & \leftarrow GPR[rs]_{31.16} \times GPR[rt]_{31.16} \\
prod2 & \leftarrow GPR[rs]_{47.32} \times GPR[rt]_{47.32} \\
prod3 & \leftarrow GPR[rs]_{63.48} \times GPR[rt]_{63.48} \\
prod4 & \leftarrow GPR[rs]_{79.64} \times GPR[rt]_{79.64} \\
prod5 & \leftarrow GPR[rs]_{95.80} \times GPR[rt]_{95.80} \\
prod6 & \leftarrow GPR[rs]_{111.96} \times GPR[rt]_{111.96} \\
prod7 & \leftarrow GPR[rs]_{127.112} \times GPR[rt]_{127.112}
\end{align*}
\]

LO \[31.0\] \leftarrow prod0_{31.0}
LO \[63.32\] \leftarrow prod1_{31.0}
HI \[31.0\] \leftarrow prod2_{31.0}
HI \[63.32\] \leftarrow prod3_{31.0}
LO \[95.64\] \leftarrow prod4_{31.0}
LO \[127.96\] \leftarrow prod5_{31.0}
HI \[95.64\] \leftarrow prod6_{31.0}
HI \[127.96\] \leftarrow prod7_{31.0}
GPR[rd]_{31.0} \leftarrow prod0_{31.0}
GPR[rd]_{63.32} \leftarrow prod1_{31.0}
GPR[rd]_{95.64} \leftarrow prod2_{31.0}
GPR[rd]_{127.96} \leftarrow prod3_{31.0}
<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A7</td>
<td>A6</td>
<td>A5</td>
<td>A4</td>
<td>A3</td>
<td>A2</td>
<td>A1</td>
<td>A0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B7</td>
<td>B6</td>
<td>B5</td>
<td>B4</td>
<td>B3</td>
<td>B2</td>
<td>B1</td>
<td>B0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Exceptions:
None

Programming Notes:
See the Programming Notes of the PMADDH instruction.
# PMULTUW

**Parallel Multiply Unsigned Word**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PMULTUW</td>
<td>01100</td>
<td>MMI3</td>
<td>101001</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Format:

PMULTUW rd, rs, rt

### Purpose:
To multiply 2 pairs of 32-bit unsigned integers in parallel.

### Description:

The low-order unsigned words of the two doublewords in GPR rs are multiplied by the low-order unsigned words of the two doublewords in GPR rt in parallel. The low-order word of the two doubleword result is placed into special register LO, and the high-order word of the two doubleword result is placed into special register HI. The two doubleword results are placed into GPR rd.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

### Restrictions:

If either GPR rt or GPR rs do not contain zero-extended 32-bit values (bits 127..96 and 63..32 equal zero) then the result of the equation will be undefined.

### Operation:

```plaintext
if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
prod0 ← (0 || GPR[rs]31.0) × (0 || GPR[rt]31.0)
prod1 ← (0 || GPR[rs]35.64) × (0 || GPR[rt]35.64)
LO63.0 ← (prod0 32) || prod031.0
HI63.0 ← (prod0 63) || prod063.32
LO127.64 ← (prod1 32) || prod131.0
HI127.64 ← (prod1 63) || prod163.32
GPR[rd]63.0 ← prod0
GPR[rd]127.64 ← prod1
```

### Example:

<table>
<thead>
<tr>
<th>rs</th>
<th>rt</th>
<th>rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3 96 95</td>
<td>B3 96 95</td>
<td>(0</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>LO83 63</th>
<th>HI83 63</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign ext</td>
<td>sign ext</td>
</tr>
<tr>
<td>(0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LO127 63</th>
<th>HI127 63</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign ext</td>
<td>sign ext</td>
</tr>
<tr>
<td>(0</td>
<td></td>
</tr>
</tbody>
</table>
```
Exceptions:
   None

Programming Notes:
   See the Programming Notes of the PMADDH instruction.
PMULTW  Parallel Multiply Word

PMULTW

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>6 5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PMULTW</td>
<td>01100</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

C790

Format:  PMULTW  rd, rs, rt

Purpose:  To multiply 2 pairs of 32-bit signed integers in parallel.

Description:  (rd, LO, HI) ← rs × rt

The low-order signed words of the two doublewords in GPR rs are multiplied by the low-order signed words of the two doublewords in GPR rt in parallel. The low-order word of the two doubleword results is placed into special register LO, and the high-order word of the two doubleword results is placed into special register HI. The two doubleword results are placed into GPR rd.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

Restrictions:

If either GPR rt or GPR rs do not contain sign-extended 32-bit values (bits 127..95 and 63..31 equal) then the result of the equation will be undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif

prod0 ← GPR[rs]31.0 × GPR[rt]31.0
prod1 ← GPR[rs]95..64 × GPR[rt]95..64

LO[63.0] ← (prod0 31)32 || prod0[31.0]
HI[63.0] ← (prod0 63)32 || prod0[63.32]

LO[127..64] ← (prod1 31)32 || prod1[31.0]
HI[127..64] ← (prod1 63)32 || prod1[63.32]

GPR[rd][63.0] ← prod0
GPR[rd][127.64] ← prod1
Exceptions:
None

Programming Notes:
See the Programming Notes of the PMADDH instruction.
The contents of GPR \( rs \) are combined with the contents of GPR \( rt \) in a bitwise logical NOR operation. The result is placed into GPR \( rd \).

This instruction operates on 128-bit registers.

**Operation:**

\[
GPR[rd]_{127..0} \leftarrow GPR[rs]_{127..0} \text{ nor } GPR[rt]_{127..0}
\]

**Exceptions:**

None
Appendix B  C790-Specific Instruction Set Details

## POR  Parallel Or

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>POR</th>
<th>MMI3</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>10010</td>
<td>101001</td>
</tr>
</tbody>
</table>

### Format:
POR rd, rs, rt

### Purpose:
To do a bitwise logical OR.

### Description:
rd ← rs OR rt

The contents of GPR rs are combined with the contents of GPR rt in a bitwise logical OR operation. The result is placed into GPR rd.

This instruction operates on 128-bit registers.

### Operation:
GPR[rd]_{127..0} ← GPR[rs]_{127..0} or GPR[rt]_{127..0}

### Exceptions:
None
### PPAC5

**Parallel Pack to 5-bits**

<table>
<thead>
<tr>
<th>MMI</th>
<th>011100</th>
<th>rt</th>
<th>00000</th>
<th>rd</th>
<th>111111</th>
<th>PPAC5</th>
<th>MMI0</th>
<th>001000</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** PPAC5 rd, rt

**Purpose:** To truncate and pack data into consecutive 5-bits.

**Description:** rd ← pack (rt)

The four 32-bit words (8, 8, 8, 8 bit) in GPR rt are packed into the four 16-bit halfwords (1, 5, 5, 5 bit). The results are placed into GPR rd. See diagram on next page.

This instruction operates on 128-bit registers.

**Operation**

\[
\begin{align*}
\text{GPR}[rd]_{4.0} & \leftarrow \text{GPR}[rt]_{7.3} \\
\text{GPR}[rd]_{9.5} & \leftarrow \text{GPR}[rt]_{15.11} \\
\text{GPR}[rd]_{14.10} & \leftarrow \text{GPR}[rt]_{23.19} \\
\text{GPR}[rd]_{15} & \leftarrow \text{GPR}[rt]_{31} \\
\text{GPR}[rd]_{31.16} & \leftarrow 0^{16} \\
\text{GPR}[rd]_{36.32} & \leftarrow \text{GPR}[rt]_{39.35} \\
\text{GPR}[rd]_{41.37} & \leftarrow \text{GPR}[rt]_{47.43} \\
\text{GPR}[rd]_{46.42} & \leftarrow \text{GPR}[rt]_{55.51} \\
\text{GPR}[rd]_{47} & \leftarrow \text{GPR}[rt]_{63} \\
\text{GPR}[rd]_{63.48} & \leftarrow 0^{16} \\
\text{GPR}[rd]_{68.64} & \leftarrow \text{GPR}[rt]_{71.67} \\
\text{GPR}[rd]_{73.69} & \leftarrow \text{GPR}[rt]_{79.75} \\
\text{GPR}[rd]_{78.74} & \leftarrow \text{GPR}[rt]_{87.83} \\
\text{GPR}[rd]_{79} & \leftarrow \text{GPR}[rt]_{95} \\
\text{GPR}[rd]_{95.80} & \leftarrow 0^{16} \\
\text{GPR}[rd]_{100.96} & \leftarrow \text{GPR}[rt]_{103.99} \\
\text{GPR}[rd]_{105.101} & \leftarrow \text{GPR}[rt]_{111.107} \\
\text{GPR}[rd]_{110.106} & \leftarrow \text{GPR}[rt]_{119.115} \\
\text{GPR}[rd]_{111} & \leftarrow \text{GPR}[rt]_{127} \\
\text{GPR}[rd]_{127.112} & \leftarrow 0^{16}
\end{align*}
\]
[Overview]

[Detail of word region (31..0)]

Exceptions:
None
Appendix B  C790-Specific Instruction Set Details

PPACB
Parallel Pack to Byte

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>Format: PPACB  rd, rs, rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td></td>
<td></td>
<td>110111</td>
<td>Purpose: To pack into consecutive bytes.</td>
</tr>
<tr>
<td>6  5</td>
<td>5  5</td>
<td>5  5</td>
<td>6  6</td>
<td>Description: rd ← pack (rs, rt)</td>
</tr>
</tbody>
</table>

The low-order bytes of the eight halfwords in GPR rs are packed into consecutive bytes of the high-order doubleword in GPR rd. Similarly, the low-order bytes of the eight halfwords in GPR rt are packed into consecutive bytes of the low-order doubleword in GPR rd.

This instruction operates on 128-bit registers.

Operation:

\[
\begin{align*}
GPR[rd]_{7..0} & \leftarrow GPR[rt]_{7..0} \\
GPR[rd]_{15..8} & \leftarrow GPR[rt]_{23..16} \\
GPR[rd]_{23..16} & \leftarrow GPR[rt]_{39..32} \\
GPR[rd]_{31..24} & \leftarrow GPR[rt]_{55..48} \\
GPR[rd]_{39..32} & \leftarrow GPR[rt]_{71..64} \\
GPR[rd]_{47..40} & \leftarrow GPR[rt]_{87..80} \\
GPR[rd]_{55..48} & \leftarrow GPR[rt]_{103..96} \\
GPR[rd]_{63..56} & \leftarrow GPR[rt]_{119..112} \\
GPR[rd]_{71..64} & \leftarrow GPR[rs]_{7..0} \\
GPR[rd]_{79..72} & \leftarrow GPR[rs]_{23..16} \\
GPR[rd]_{87..80} & \leftarrow GPR[rs]_{39..32} \\
GPR[rd]_{95..88} & \leftarrow GPR[rs]_{55..48} \\
GPR[rd]_{103..96} & \leftarrow GPR[rs]_{71..64} \\
GPR[rd]_{111..104} & \leftarrow GPR[rs]_{87..80} \\
GPR[rd]_{119..112} & \leftarrow GPR[rs]_{103..96} \\
GPR[rd]_{127..120} & \leftarrow GPR[rs]_{119..112}
\end{align*}
\]

Exceptions:

None
### PPACH

**Parallel Pack to Halfword**

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>PPACH</th>
<th>MMI0</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>10111</td>
<td>00100</td>
</tr>
</tbody>
</table>

**Format:** PPACH rd, rs, rt

**Purpose:** To pack into consecutive halfwords.

**Description:**

The low-order halfwords of the four words in GPR rs are packed into consecutive halfwords of the high-order doubleword in GPR rd. Similarly, the low-order halfwords of the four words in GPR rt are packed into consecutive halfwords of the low-order doubleword in GPR rd.

This instruction operates on 128-bit registers.

**Operation:**

- GPR[rd]15..0 ← GPR[rt]15..0
- GPR[rd]31..16 ← GPR[rt]47..32
- GPR[rd]47..32 ← GPR[rt]79..64
- GPR[rd]63..48 ← GPR[rt]111..96
- GPR[rd]79..64 ← GPR[rs]15..0
- GPR[rd]95..80 ← GPR[rs]47..32
- GPR[rd]111..96 ← GPR[rs]79..64
- GPR[rd]127..112 ← GPR[rs]111..96

**Exceptions:**

None
### PPACW

**Parallel Pack to Word**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PPACW</td>
<td>MMI0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>00</td>
<td>5</td>
<td>5</td>
<td>10011</td>
<td>001000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** PPACW rd, rs, rt

**Purpose:** To pack into consecutive words.

**Description:**

The low-order words of the two doublewords in GPR rs are packed into consecutive words of the high-order doubleword in GPR rd. Similarly, the low-order words of the two doublewords in GPR rt are packed into consecutive words of the low-order doubleword in GPR rd.

This instruction operates on 128-bit registers.

**Operation:**

\[
\begin{align*}
\text{GPR}[rd]_{31..0} & \leftarrow \text{GPR}[rt]_{31..0} \\
\text{GPR}[rd]_{63..32} & \leftarrow \text{GPR}[rt]_{95..64} \\
\text{GPR}[rd]_{95..64} & \leftarrow \text{GPR}[rs]_{31..0} \\
\text{GPR}[rd]_{127..96} & \leftarrow \text{GPR}[rs]_{95..64}
\end{align*}
\]

**Exceptions:**

None
## PREVH

**Parallel Reverse Halfword**

<table>
<thead>
<tr>
<th>MMI</th>
<th>rt</th>
<th>rd</th>
<th>PREVH</th>
<th>MMI2</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>00000</td>
<td></td>
<td>11011</td>
<td>001001</td>
</tr>
</tbody>
</table>

### Format:

```
PREVH rd, rt
```

### Purpose:

To reverse halfwords.

### Description:

```
rd ← reverse (rt)
```

The four high-order halfwords in GPR rt are reversed and the four low-order halfwords in GPR rt are reversed. The results are placed into GPR rd.

This instruction operates on 128-bit registers.

### Operation:

- `GPR[rd][15..0] ← GPR[rt][63..48]`
- `GPR[rd][31..16] ← GPR[rt][47..32]`
- `GPR[rd][47..32] ← GPR[rt][31..16]`
- `GPR[rd][63..48] ← GPR[rt][15..0]`
- `GPR[rd][79..64] ← GPR[rt][127..112]`
- `GPR[rd][95..80] ← GPR[rt][111..96]`
- `GPR[rd][111..96] ← GPR[rt][95..80]`
- `GPR[rd][127..112] ← GPR[rt][79..64]`

### Exceptions:

None
**PROT3W**

Parallel Rotate 3 Words Left

<table>
<thead>
<tr>
<th>MMI</th>
<th>011100</th>
<th>rt</th>
<th>rd</th>
<th>PROT3W</th>
<th>111111</th>
<th>MMI2</th>
<th>001001</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**Format:** PROT3W rd, rt

**Purpose:** To rotate words.

**Description:** rd ← rotate (rt)

The three low-order words in GPR rt are rotated to the right. The results are placed into GPR rd while the other word is copied directly to the corresponding word in GPR rd.

This instruction operates on 128-bit registers.

**Operation:**

\[
\begin{align*}
\text{GPR}[rd]_{31..0} & \leftarrow \text{GPR}[rt]_{63..32} \\
\text{GPR}[rd]_{63..32} & \leftarrow \text{GPR}[rt]_{95..64} \\
\text{GPR}[rd]_{95..64} & \leftarrow \text{GPR}[rt]_{31..0} \\
\text{GPR}[rd]_{127..96} & \leftarrow \text{GPR}[rt]_{127..96}
\end{align*}
\]

**Exceptions:**

None
PSLLH  Parallel Shift Left Logical Halfword  PSLLH

<table>
<thead>
<tr>
<th>MMI</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>PSLLH</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>110100</td>
</tr>
</tbody>
</table>

Format: \( \text{PSLLH} \) \( rd, rt, sa \)

Purpose: To logically shift left 8 halfwords by a fixed number of bits, in parallel.

Description: \( rd \leftarrow rt \ll sa \) (logical)

The eight halfwords in GPR \( rt \) are shifted left in parallel, inserting zeros into the emptied bits; the results are placed into the corresponding eight halfwords in GPR \( rd \). The bit shift count is specified by the low-order four bits of \( sa \).

This instruction operates on 128-bit registers.

Operation:
\[
\begin{align*}
  s &\leftarrow sa_{3..0} \\
  GPR[rd]_{15..0} &\leftarrow GPR[rt]_{15..0} \ | 0^6 \\
  GPR[rd]_{31..16} &\leftarrow GPR[rt]_{31..16} \ | 0^6 \\
  GPR[rd]_{47..32} &\leftarrow GPR[rt]_{47..32} \ | 0^6 \\
  GPR[rd]_{63..48} &\leftarrow GPR[rt]_{63..48} \ | 0^6 \\
  GPR[rd]_{79..64} &\leftarrow GPR[rt]_{79..64} \ | 0^6 \\
  GPR[rd]_{95..80} &\leftarrow GPR[rt]_{95..80} \ | 0^6 \\
  GPR[rd]_{111..96} &\leftarrow GPR[rt]_{111..96} \ | 0^6 \\
  GPR[rd]_{127..112} &\leftarrow GPR[rt]_{127..112} \ | 0^6 \\
\end{align*}
\]

Exceptions:
None
### PSLLVW

**Parallel Shift Left Logical Variable Word**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PSLLVW</td>
<td>MMI2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>00010</td>
<td>001001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** PSLLVW rd, rt, rs

**Purpose:** To logically shift left 2 words by a variable number of bits, in parallel.

**Description:**

rd ← rt << rs (logical)

The low-order words of the two doublewords in GPR rt are shifted left in parallel, inserting zeros into the emptied bits; the results are placed into the corresponding two words in GPR rd. The bit shift counts are specified by the low-order five bits of the two doublewords in GPR rs.

This instruction operates on 128-bit registers.

**Operation:**

\[
\begin{align*}
\text{s0} & \leftarrow \text{GPR}[rs]_{4..0} \\
\text{s1} & \leftarrow \text{GPR}[rs]_{68..64} \\
\text{temp0} & \leftarrow \text{GPR}[rt]_{31..0} \| 0^5 \\
\text{temp1} & \leftarrow \text{GPR}[rt]_{95..64} \| 0^5 \\
\text{GPR}[rd]_{63..0} & \leftarrow (\text{temp0}_{31})^{32} \| \text{temp0}_{31..0} \\
\text{GPR}[rd]_{127..64} & \leftarrow (\text{temp1}_{31})^{32} \| \text{temp1}_{31..0}
\end{align*}
\]

**Exceptions:** None
PSLLW

Parallel Shift Left Logical Word

PSLLW

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>00000</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>PSLLW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>0</td>
<td>011100</td>
<td>0</td>
<td>011100</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: PSLLW rd, rt, sa

Purpose: To logically shift left 4 words by a fixed number of bits, in parallel.

Description: rd ← rt << sa (logical)

The four words in GPR rt are shifted left by five bits of sa in parallel, inserting zeros into the emptied bits; the results are placed into the corresponding four words in GPR rd.

This instruction operates on 128-bit registers.

Operation:

\[ s \leftarrow sa_{4..0} \]
\[ GPR[rd]_{31..0} \leftarrow GPR[rt]_{31-s..0} \]
\[ GPR[rd]_{63..32} \leftarrow GPR[rt]_{63-s..32} \]
\[ GPR[rd]_{95..64} \leftarrow GPR[rt]_{95-s..64} \]
\[ GPR[rd]_{127..96} \leftarrow GPR[rt]_{127-s..96} \]

Exceptions:

None
**PSRAH**

**Parallel Shift Right Arithmetic Halfword**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>0</td>
<td>00000</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>PSRAH</td>
<td>110111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

PSRAH rd, rt, sa

**Purpose:**

To arithmetically shift right 8 halfwords by a fixed number of bits, in parallel.

**Description:**

rd ← rt >> sa (arithmetic)

The eight halfwords in GPR rt are shifted right by sa bits in parallel sign extending the high order bits; the results are placed into the corresponding eight halfwords in GPR rd. The bit shift count is specified by the low-order four bits of sa.

This instruction operates on 128-bit registers.

**Operation:**

\[ s \leftarrow sa_{3..0} \]

\[ GPR[rd]_{15..0} \leftarrow (GPR[rt]_{15})^s \| GPR[rt]_{15..5} \]

\[ GPR[rd]_{31..16} \leftarrow (GPR[rt]_{31})^s \| GPR[rt]_{31..(16+s)} \]

\[ GPR[rd]_{47..32} \leftarrow (GPR[rt]_{47})^s \| GPR[rt]_{47..(32+s)} \]

\[ GPR[rd]_{63..48} \leftarrow (GPR[rt]_{63})^s \| GPR[rt]_{63..(48+s)} \]

\[ GPR[rd]_{79..64} \leftarrow (GPR[rt]_{79})^s \| GPR[rt]_{79..(64+s)} \]

\[ GPR[rd]_{95..80} \leftarrow (GPR[rt]_{95})^s \| GPR[rt]_{95..(80+s)} \]

\[ GPR[rd]_{111..96} \leftarrow (GPR[rt]_{111})^s \| GPR[rt]_{111..(96+s)} \]

\[ GPR[rd]_{127..112} \leftarrow (GPR[rt]_{127})^s \| GPR[rt]_{127..(112+s)} \]

**Exceptions:**

None
PSRAVW Parallel Shift Right Arithmetic Variable Word

C790

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>PSRAVW</th>
<th>MMI3</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>00011</td>
<td>101001</td>
</tr>
</tbody>
</table>

Format: PSRAVW rd, rt, rs

Purpose: To arithmetically shift right 2 words by a variable number of bits, in parallel.

Description: rd ← rt >> rs (arithmetic)

The low-order words of the two doublewords in GPR rt are shifted right in parallel, sign extending the high order bits; the results are placed into the corresponding two words in GPR rd. The bit shift counts are specified by the low-order five bits of the two doublewords in GPR rs.

This instruction operates on 128-bit registers.

Operation:

s0 ← GPR[rs]4..0
s1 ← GPR[rs]68..64

temp0 ← (GPR[rt]31)s0 || GPR[rt]31..s0

temp1 ← (GPR[rt]95)s1 || GPR[rt]95..(64+s1)

GPR[rd]63..0 ← (temp031)32 || temp031..0
GPR[rd]127..64 ← (temp131)32 || temp131..0

Exceptions:

None
Appendix B  C790-Specific Instruction Set Details

**PSRAW** Parallel Shift Right Arithmetic Word

<table>
<thead>
<tr>
<th>MMI</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>PSRAW</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>00000</td>
<td>111111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** PSRAW rd, rt, sa

**Purpose:** To arithmetically shift right 4 word by a fixed number of bits, in parallel.

**Description:**

\[ \text{rd} \leftarrow \text{rt} >> \text{sa} \]  (arithmetic)

The four words in GPR rt are shifted right by five bits of sa in parallel, sign extending the high order bits; the results are placed into the corresponding four words in GPR rd.

This instruction operates on 128-bit registers.

**Operation:**

\[ s \leftarrow s_{a4..0} \]
\[ \text{GPR}[\text{rd}]_{31..0} \leftarrow (\text{GPR}[\text{rt}]_{31})^s \parallel \text{GPR}[\text{rt}]_{31..s} \]
\[ \text{GPR}[\text{rd}]_{63..32} \leftarrow (\text{GPR}[\text{rt}]_{63})^s \parallel \text{GPR}[\text{rt}]_{63..(32+s)} \]
\[ \text{GPR}[\text{rd}]_{95..64} \leftarrow (\text{GPR}[\text{rt}]_{95})^s \parallel \text{GPR}[\text{rt}]_{95..(64+s)} \]
\[ \text{GPR}[\text{rd}]_{127..96} \leftarrow (\text{GPR}[\text{rt}]_{127})^s \parallel \text{GPR}[\text{rt}]_{127..(96+s)} \]

**Exceptions:** None
### Appendix B C790-Specific Instruction Set Details

#### Parallel Shift Right Logical Halfword

**Format:**

```
PSRLH rd, rt, sa
```

**Purpose:**

To logically shift right 8 halfwords by a fixed number of bits, in parallel.

**Description:**

```
rd ← rt >> sa (logical)
```

The eight halfwords in GPR `rt` are shifted right by `sa` bits, in parallel, inserting zeros into the high order bits; the results are placed into the corresponding eight halfwords in GPR `rd`. The bit shift count is specified by the low-order four bits of `sa`.

This instruction operates on 128-bit registers.

**Operation:**

```
s ← s3..0
GPR[rd]15..0 ← 0s || GPR[rt]15..s
GPR[rd]31..16 ← 0s || GPR[rt]31..(16+s)
GPR[rd]47..32 ← 0s || GPR[rt]47..(32+s)
GPR[rd]63..48 ← 0s || GPR[rt]63..(48+s)
GPR[rd]79..64 ← 0s || GPR[rt]79..(64+s)
GPR[rd]95..80 ← 0s || GPR[rt]95..(80+s)
GPR[rd]111..96 ← 0s || GPR[rt]111..(96+s)
GPR[rd]127..122 ← 0s || GPR[rt]127..(112+s)
```

**Exceptions:**

None
**PSRLVW**

**Parallel Shift Right Logical Variable Word**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PSRLVW</td>
<td>00011</td>
<td>MMI2</td>
<td>001001</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** PSRLVW rd, rt, rs

**Purpose:** To logically shift right 2 words by a variable number of bits, in parallel.

**Description:** rd ← rt >> rs (logical)

The low-order words of the two doublewords in GPR rt are shifted right in parallel, inserting zeros into the high order bits. The results are sign extended; the results are placed into the corresponding two words in GPR rd. The bit shift counts are specified by the low-order five bits of the two doublewords in GPR rs.

This instruction operates on 128-bit registers.

**Operation:**

\[
s0 \leftarrow GPR[rs]_{4..0} \\
s1 \leftarrow GPR[rs]_{68..64} \\
temp0 \leftarrow 0^{s0} \| GPR[rt]_{31..s0} \\
temp1 \leftarrow 0^{s1} \| GPR[rt]_{95..(64+s1)} \\
GPR[rd]_{63..0} \leftarrow (temp0_{31})^{32} \| temp0_{31..0} \\
GPR[rd]_{127..64} \leftarrow (temp1_{31})^{32} \| temp1_{31..0}
\]

**Exceptions:** None
**PSRLW**

Parallel Shift Right Logical Word

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MMI</strong></td>
<td>011100</td>
<td>0</td>
<td>00000</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>PSRLW</td>
<td>111110</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**C790**

**Format:** PSRLW rd, rt, sa

**Purpose:** To logically shift right 4 words by a fixed number of bits, in parallel.

**Description:**

rd ← rt >> sa (logical)

The four words in GPR rt are shifted right by five bits of sa, in parallel, inserting zeros into the high order bits; the results are placed into the corresponding four words in GPR rd.

This instruction operates on 128-bit registers.

**Operation:**

\[
s \leftarrow sa_{4..0} \\
GPR[rd]_{31..0} \leftarrow 0^s \parallel GPR[rt]_{31..s} \\
GPR[rd]_{63..32} \leftarrow 0^s \parallel GPR[rt]_{63..(32+s)} \\
GPR[rd]_{95..64} \leftarrow 0^s \parallel GPR[rt]_{95..(64+s)} \\
GPR[rd]_{127..96} \leftarrow 0^s \parallel GPR[rt]_{127..(96+s)}
\]

**Exceptions:**

None
Appendix B  C790-Specific Instruction Set Details

PSUBB  Parallel Subtract Byte  PSUBB

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>PSUBB</th>
<th>MMI0</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>01001</td>
<td>001000</td>
</tr>
</tbody>
</table>

C790

Format:  PSUBB  rd, rs, rt

Purpose:  To subtract 16 pairs of 8-bit integers in parallel.

Description:  rd ← rs − rt

The sixteen signed byte values in GPR rt are subtracted from the corresponding sixteen byte values in GPR rs in parallel. The results are placed into the corresponding sixteen bytes in GPR rd.

No overflow or underflow exceptions are generated under any circumstances.

This instruction operates on 128-bit registers.

Operation:

\[
\begin{align*}
GPR[rd]_{7.0} & \leftarrow (GPR[rs]_{7.0} - GPR[rt]_{7.0})_{7.0} \\
GPR[rd]_{15.8} & \leftarrow (GPR[rs]_{15.8} - GPR[rt]_{15.8})_{7.0} \\
GPR[rd]_{23.16} & \leftarrow (GPR[rs]_{23.16} - GPR[rt]_{23.16})_{7.0} \\
GPR[rd]_{31.24} & \leftarrow (GPR[rs]_{31.24} - GPR[rt]_{31.24})_{7.0} \\
GPR[rd]_{39.32} & \leftarrow (GPR[rs]_{39.32} - GPR[rt]_{39.32})_{7.0} \\
GPR[rd]_{47.40} & \leftarrow (GPR[rs]_{47.40} - GPR[rt]_{47.40})_{7.0} \\
GPR[rd]_{55.48} & \leftarrow (GPR[rs]_{55.48} - GPR[rt]_{55.48})_{7.0} \\
GPR[rd]_{63.56} & \leftarrow (GPR[rs]_{63.56} - GPR[rt]_{63.56})_{7.0} \\
GPR[rd]_{71.64} & \leftarrow (GPR[rs]_{71.64} - GPR[rt]_{71.64})_{7.0} \\
GPR[rd]_{79.72} & \leftarrow (GPR[rs]_{79.72} - GPR[rt]_{79.72})_{7.0} \\
GPR[rd]_{87.80} & \leftarrow (GPR[rs]_{87.80} - GPR[rt]_{87.80})_{7.0} \\
GPR[rd]_{95.88} & \leftarrow (GPR[rs]_{95.88} - GPR[rt]_{95.88})_{7.0} \\
GPR[rd]_{103.96} & \leftarrow (GPR[rs]_{103.96} - GPR[rt]_{103.96})_{7.0} \\
GPR[rd]_{111.104} & \leftarrow (GPR[rs]_{111.104} - GPR[rt]_{111.104})_{7.0} \\
GPR[rd]_{119.112} & \leftarrow (GPR[rs]_{119.112} - GPR[rt]_{119.112})_{7.0} \\
GPR[rd]_{127.120} & \leftarrow (GPR[rs]_{127.120} - GPR[rt]_{127.120})_{7.0}
\end{align*}
\]

Exceptions:

None

B-142
### PSUBH

**Parallel Subtract Halfword**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PSUBH</td>
<td>MMI0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Format: | PSUBH rd, rs, rt |
| Purpose: | To subtract 8 pairs of 16-bit integers in parallel. |
| Description: | rd ← rs − rt |

The eight signed halfwords in GPR rt are subtracted from the corresponding eight halfwords in GPR rs in parallel. The results are placed into the corresponding eight halfwords in GPR rd.

No overflow or underflow exceptions are generated under any circumstances.

This instruction operates on 128-bit registers.

**Operation:**

- \( \text{GPR}[\text{rd}]_{15.0} \leftarrow (\text{GPR}[\text{rs}]_{15.0} - \text{GPR}[\text{rt}]_{15.0})_{15.0} \)
- \( \text{GPR}[\text{rd}]_{31.16} \leftarrow (\text{GPR}[\text{rs}]_{31.16} - \text{GPR}[\text{rt}]_{31.16})_{15.0} \)
- \( \text{GPR}[\text{rd}]_{47.32} \leftarrow (\text{GPR}[\text{rs}]_{47.32} - \text{GPR}[\text{rt}]_{47.32})_{15.0} \)
- \( \text{GPR}[\text{rd}]_{63.48} \leftarrow (\text{GPR}[\text{rs}]_{63.48} - \text{GPR}[\text{rt}]_{63.48})_{15.0} \)
- \( \text{GPR}[\text{rd}]_{79.64} \leftarrow (\text{GPR}[\text{rs}]_{79.64} - \text{GPR}[\text{rt}]_{79.64})_{15.0} \)
- \( \text{GPR}[\text{rd}]_{95.80} \leftarrow (\text{GPR}[\text{rs}]_{95.80} - \text{GPR}[\text{rt}]_{95.80})_{15.0} \)
- \( \text{GPR}[\text{rd}]_{111.96} \leftarrow (\text{GPR}[\text{rs}]_{111.96} - \text{GPR}[\text{rt}]_{111.96})_{15.0} \)
- \( \text{GPR}[\text{rd}]_{127.112} \leftarrow (\text{GPR}[\text{rs}]_{127.112} - \text{GPR}[\text{rt}]_{127.112})_{15.0} \)

**Exceptions:**

None
PSUBSB

Parallel Subtract with Signed saturation Byte

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>0111</td>
<td>00</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PSUBSB</td>
<td>11001</td>
<td>MMI0</td>
<td>0010</td>
<td>00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: PSUBSB rd, rs, rt

Purpose: To subtract 16 pairs of 8-bit signed integers with saturation in parallel.

Description:

\[
\text{rd} \leftarrow \text{rs} - \text{rt}
\]

The sixteen signed bytes in GPR \( rt \) are subtracted from the corresponding sixteen signed bytes in GPR \( rs \) in parallel. The results are placed into the corresponding sixteen bytes in GPR \( rd \).

No overflow or underflow exceptions are generated under any circumstances. Results beyond the range of a signed byte value are saturated according to the following:

Overflow: \( 0x7F \)

Underflow: \( 0x80 \)

This instruction operates on 128-bit registers.

Operation:

\[
\text{if } ((\text{GPR}[rs]_{7..0} - \text{GPR}[rt]_{7..0}) > 0x7F) \text{ then}
\]

\[
\text{GPR}[rd]_{7..0} \leftarrow 0x7F
\]

\[
\text{else if } (0x100 \leq (\text{GPR}[rs]_{7..0} - \text{GPR}[rt]_{7..0}) < 0x180) \text{ then}
\]

\[
\text{GPR}[rd]_{7..0} \leftarrow 0x80
\]

\[
\text{else}
\]

\[
\text{GPR}[rd]_{7..0} \leftarrow (\text{GPR}[rs]_{7..0} - \text{GPR}[rt]_{7..0})_{7..0}
\]

\[
\text{endif}
\]

\[
\text{if } ((\text{GPR}[rs]_{15..8} - \text{GPR}[rt]_{15..8}) > 0x7F) \text{ then}
\]

\[
\text{GPR}[rd]_{15..8} \leftarrow 0x7F
\]

\[
\text{else if } (0x100 \leq (\text{GPR}[rs]_{15..8} - \text{GPR}[rt]_{15..8}) < 0x180) \text{ then}
\]

\[
\text{GPR}[rd]_{15..8} \leftarrow 0x80
\]

\[
\text{else}
\]

\[
\text{GPR}[rd]_{15..8} \leftarrow (\text{GPR}[rs]_{15..8} - \text{GPR}[rt]_{15..8})_{7..0}
\]

\[
\text{endif}
\]

\[
\text{if } ((\text{GPR}[rs]_{23..16} - \text{GPR}[rt]_{23..16}) > 0x7F) \text{ then}
\]

\[
\text{GPR}[rd]_{23..16} \leftarrow 0x7F
\]

\[
\text{else if } (0x100 \leq (\text{GPR}[rs]_{23..16} - \text{GPR}[rt]_{23..16}) < 0x180) \text{ then}
\]

\[
\text{GPR}[rd]_{23..16} \leftarrow 0x80
\]

\[
\text{else}
\]

\[
\text{GPR}[rd]_{23..16} \leftarrow (\text{GPR}[rs]_{23..16} - \text{GPR}[rt]_{23..16})_{7..0}
\]

\[
\text{endif}
\]
if ((GPR[rs]31..24 − GPR[rt]31..24) > 0x7F) then
    GPR[rd]31..24 ← 0x7F
else if (0x100 <= (GPR[rs]31..24 − GPR[rt]31..24) < 0x180) then
    GPR[rd]31..24 ← 0x80
else
    GPR[rd]31..24 ← (GPR[rs]31..24 − GPR[rt]31..24)7..0
endif

if ((GPR[rs]39..32 − GPR[rt]39..32) > 0x7F) then
    GPR[rd]39..32 ← 0x7F
else if (0x100 <= (GPR[rs]39..32 − GPR[rt]39..32) < 0x180) then
    GPR[rd]39..32 ← 0x80
else
    GPR[rd]39..32 ← (GPR[rs]39..32 − GPR[rt]39..32)7..0
endif

if ((GPR[rs]47..40 − GPR[rt]47..40) > 0x7F) then
    GPR[rd]47..40 ← 0x7F
else if (0x100 <= (GPR[rs]47..40 − GPR[rt]47..40) < 0x180) then
    GPR[rd]47..40 ← 0x80
else
    GPR[rd]47..40 ← (GPR[rs]47..40 − GPR[rt]47..40)7..0
endif

if ((GPR[rs]55..48 − GPR[rt]55..48) > 0x7F) then
    GPR[rd]55..48 ← 0x7F
else if (0x100 <= (GPR[rs]55..48 − GPR[rt]55..48) < 0x180) then
    GPR[rd]55..48 ← 0x80
else
    GPR[rd]55..48 ← (GPR[rs]55..48 − GPR[rt]55..48)7..0
endif

if ((GPR[rs]63..56 − GPR[rt]63..56) > 0x7F) then
    GPR[rd]63..56 ← 0x7F
else if (0x100 <= (GPR[rs]63..56 − GPR[rt]63..56) < 0x180) then
    GPR[rd]63..56 ← 0x80
else
    GPR[rd]63..56 ← (GPR[rs]63..56 − GPR[rt]63..56)7..0
endif

if ((GPR[rs]71..64 − GPR[rt]71..64) > 0x7F) then
    GPR[rd]71..64 ← 0x7F
else if (0x100 <= (GPR[rs]71..64 − GPR[rt]71..64) < 0x180) then
    GPR[rd]71..64 ← 0x80
else
    GPR[rd]71..64 ← (GPR[rs]71..64 − GPR[rt]71..64)7..0
endif
if ((GPR[rs]72..72 − GPR[rt]72..72) > 0x7F) then
    GPR[rd]72..72 ← 0x7F
else if (0x100 <= (GPR[rs]72..72 − GPR[rt]72..72) < 0x180) then
    GPR[rd]72..72 ← 0x80
else
    GPR[rd]72..72 ← (GPR[rs]72..72 − GPR[rt]72..72)7..0
endif

if ((GPR[rs]80..80 − GPR[rt]80..80) > 0x7F) then
    GPR[rd]80..80 ← 0x7F
else if (0x100 <= (GPR[rs]80..80 − GPR[rt]80..80) < 0x180) then
    GPR[rd]80..80 ← 0x80
else
    GPR[rd]80..80 ← (GPR[rs]80..80 − GPR[rt]80..80)7..0
endif

if ((GPR[rs]96..96 − GPR[rt]96..96) > 0x7F) then
    GPR[rd]96..96 ← 0x7F
else if (0x100 <= (GPR[rs]96..96 − GPR[rt]96..96) < 0x180) then
    GPR[rd]96..96 ← 0x80
else
    GPR[rd]96..96 ← (GPR[rs]96..96 − GPR[rt]96..96)7..0
endif

if ((GPR[rs]104..104 − GPR[rt]104..104) > 0x7F) then
    GPR[rd]104..104 ← 0x7F
else if (0x100 <= (GPR[rs]104..104 − GPR[rt]104..104) < 0x180) then
    GPR[rd]104..104 ← 0x80
else
    GPR[rd]104..104 ← (GPR[rs]104..104 − GPR[rt]104..104)7..0
endif

if ((GPR[rs]112..112 − GPR[rt]112..112) > 0x7F) then
    GPR[rd]112..112 ← 0x7F
else if (0x100 <= (GPR[rs]112..112 − GPR[rt]112..112) < 0x180) then
    GPR[rd]112..112 ← 0x80
else
    GPR[rd]112..112 ← (GPR[rs]112..112 − GPR[rt]112..112)7..0
endif
if ((GPR[rs]127..120 - GPR[rt]127..120) > 0x7F) then
    GPR[rd]127..120 ← 0x7F
else if (0x100 <= (GPR[rs]127..120 - GPR[rt]127..120) < 0x180) then
    GPR[rd]127..120 ← 0x80
else
    GPR[rd]127..120 ← (GPR[rs]127..120 - GPR[rt]127..120)7..0
endif

* Saturate to signed byte

Exceptions:

None
PSUBSH
Parallel Subtract with Signed Saturation Halfword

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>PSUBSH</th>
<th>MMI0</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>001000</td>
<td>10101</td>
<td>001000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: PSUBSH rd, rs, rt

Purpose: To subtract 8 pairs of 16-bit signed integers with saturation in parallel.

Description:
rd ← rs − rt

The eight signed halfwords in GPR rt are subtracted from the corresponding eight signed halfwords in GPR rs in parallel. The results are placed into the corresponding eight halfwords in GPR rd.

No overflow or underflow exceptions are generated under any circumstances. Results beyond the range of a signed halfword value are saturated according to the following:

Overflow: 0x7FFF
Underflow: 0x8000

This instruction operates on 128-bit registers.

Operation:

if ((GPR[rs]15..0 − GPR[rt]15..0) > 0x7FFF) then
    GPR[rd]15..0 ← 0x7FFF
else if (0x10000 <= (GPR[rs]15..0 − GPR[rt]15..0) < 0x18000) then
    GPR[rd]15..0 ← 0x8000
else
    GPR[rd]15..0 ← (GPR[rs]15..0 − GPR[rt]15..0)15..0
endif

if ((GPR[rs]31..16 − GPR[rt]31..16) > 0x7FFF) then
    GPR[rd]31..16 ← 0x7FFF
else if (0x10000 <= (GPR[rs]31..16 − GPR[rt]31..16) < 0x18000) then
    GPR[rd]31..16 ← 0x8000
else
    GPR[rd]31..16 ← (GPR[rs]31..16 − GPR[rt]31..16)15..0
endif

if ((GPR[rs]47..32 − GPR[rt]47..32) > 0x7FFF) then
    GPR[rd]47..32 ← 0x7FFF
else if (0x10000 <= (GPR[rs]47..32 − GPR[rt]47..32) < 0x18000) then
    GPR[rd]47..32 ← 0x8000
else
    GPR[rd]47..32 ← (GPR[rs]47..32 − GPR[rt]47..32)15..0
endif

if ((GPR[rs]63..48 − GPR[rt]63..48) > 0x7FFF) then
    GPR[rd]63..48 ← 0x7FFF
else if (0x10000 <= (GPR[rs]63..48 − GPR[rt]63..48) < 0x18000) then
    GPR[rd]63..48 ← 0x8000
else
    GPR[rd]63..48 ← (GPR[rs]63..48 − GPR[rt]63..48)15..0
endif
if ((GPR[rs]79..64 − GPR[rt]79..64) > 0x7FFF) then
  GPR[rd]79..64 ← 0x7FFF
else if (0x10000 <= (GPR[rs]79..64 − GPR[rt]79..64) < 0x18000) then
  GPR[rd]79..64 ← 0x8000
else
  GPR[rd]79..64 ← (GPR[rs]79..64 − GPR[rt]79..64)\text{15..0}
endif

if ((GPR[rs]95..80 − GPR[rt]95..80) > 0x7FFF) then
  GPR[rd]95..80 ← 0x7FFF
else if (0x10000 <= (GPR[rs]95..80 − GPR[rt]95..80) < 0x18000) then
  GPR[rd]95..80 ← 0x8000
else
  GPR[rd]95..80 ← (GPR[rs]95..80 − GPR[rt]95..80)\text{15..0}
endif

if ((GPR[rs]111..96 − GPR[rt]111..96) > 0x7FFF) then
  GPR[rd]111..96 ← 0x7FFF
else if (0x10000 <= (GPR[rs]111..96 − GPR[rt]111..96) < 0x18000) then
  GPR[rd]111..96 ← 0x8000
else
  GPR[rd]111..96 ← (GPR[rs]111..96 − GPR[rt]111..96)\text{15..0}
endif

if ((GPR[rs]127..112 − GPR[rt]127..112) > 0x7FFF) then
  GPR[rd]127..112 ← 0x7FFF
else if (0x10000 <= (GPR[rs]127..112 − GPR[rt]127..112) < 0x18000) then
  GPR[rd]127..112 ← 0x8000
else
  GPR[rd]127..112 ← (GPR[rs]127..112 − GPR[rt]127..112)\text{15..0}
endif

* Saturate to signed halfword

Exceptions:
None
**PSUBSW**  Parallel Subtract with Signed Saturation Word

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PSUBSW</td>
<td>10001</td>
<td>MMI0</td>
<td>001000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  
PSUBSW rd, rs, rt

**Purpose:**  
To subtract 4 pairs of 32-bit signed integers with saturation in parallel.

**Description:**  
rd ← rs – rt  

The four signed words in GPR rt are subtracted from the corresponding four signed words in GPR rs in parallel. The results are placed into the corresponding four words in GPR rd.

No overflow or underflow exceptions are generated under any circumstances. Results beyond the range of a signed word value are saturated according to the following:

**Overflow:**  
0xFFFFFFF

**Underflow:**  
0x80000000

This instruction operates on 128-bit registers.

**Operation:**

if ((GPR[rs]31..0 - GPR[rt]31..0) > 0xFFFFFFF) then  
GPR[rd]31..0 ← 0xFFFFFFF
else if (0x100000000 <= (GPR[rs]31..0 - GPR[rt]31..0) < 0x180000000) then  
GPR[rd]31..0 ← 0x80000000
else  
GPR[rd]31..0 ← (GPR[rs]31..0 - GPR[rt]31..0)31..0
endif

if ((GPR[rs]63..32 - GPR[rt]63..32) > 0xFFFFFFF) then  
GPR[rd]63..32 ← 0xFFFFFFF
else if (0x100000000 <= (GPR[rs]63..32 - GPR[rt]63..32) < 0x180000000) then  
GPR[rd]63..32 ← 0x80000000
else  
GPR[rd]63..32 ← (GPR[rs]63..32 - GPR[rt]63..32)31..0
endif

if ((GPR[rs]95..64 - GPR[rt]95..64) > 0xFFFFFFF) then  
GPR[rd]95..64 ← 0xFFFFFFF
else if (0x100000000 <= (GPR[rs]95..64 - GPR[rt]95..64) < 0x180000000) then  
GPR[rd]95..64 ← 0x80000000
else  
GPR[rd]95..64 ← (GPR[rs]95..64 - GPR[rt]95..64)31..0
endif
if (($GPR[rs]_{127..96} - GPR[rt]_{127..96}) > 0x7FFFFFFF) then
    $GPR[rd]_{127..96} \leftarrow 0x7FFFFFFF$
else if ($0x100000000 \leq (GPR[rs]_{127..96} - GPR[rt]_{127..96}) < 0x180000000$) then
    $GPR[rd]_{127..96} \leftarrow 0x80000000$
else
    $GPR[rd]_{127..96} \leftarrow (GPR[rs]_{127..96} - GPR[rt]_{127..96})_{31..0}$
endif

---

**Exceptions:**

None
PSUBUB
Parallel Subtract withUnsigned Saturation Byte

<table>
<thead>
<tr>
<th>MM</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>PSUBUB</th>
<th>MM</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>11</td>
<td>00</td>
<td>11</td>
<td>00111</td>
<td>01</td>
</tr>
</tbody>
</table>

Format: PSUBUB rd, rs, rt
Purpose: To subtract 16 pairs of 8-bit unsigned integers with saturation in parallel.
Description: rd ← rs − rt

The sixteen unsigned bytes in GPR rt are subtracted from the corresponding sixteen unsigned bytes in GPR rs in parallel. The results are placed into the corresponding sixteen bytes in GPR rd.

No underflow exceptions are generated under any circumstances. Results beyond the range of an unsigned byte value are saturated according to the following:

Underflow: 0x00

This instruction operates on 128-bit registers.

Operation:
if ((GPR[rs]7..0 − GPR[rt]7..0) < 0x00) then
GPR[rd]7..0 ← 0x00
else
GPR[rd]7..0 ← (GPR[rs]7..0 − GPR[rt]7..0)7..0
endif

if ((GPR[rs]15..8 − GPR[rt]15..8) < 0x00) then
GPR[rd]15..8 ← 0x00
else
GPR[rd]15..8 ← (GPR[rs]15..8 − GPR[rt]15..8)7..0
endif

if ((GPR[rs]23..16 − GPR[rt]23..16) < 0x00) then
GPR[rd]23..16 ← 0x00
else
GPR[rd]23..16 ← (GPR[rs]23..16 − GPR[rt]23..16)7..0
endif

if ((GPR[rs]31..24 − GPR[rt]31..24) < 0x00) then
GPR[rd]31..24 ← 0x00
else
GPR[rd]31..24 ← (GPR[rs]31..24 − GPR[rt]31..24)7..0
endif

if ((GPR[rs]39..32 − GPR[rt]39..32) < 0x00) then
GPR[rd]39..32 ← 0x00
else
GPR[rd]39..32 ← (GPR[rs]39..32 − GPR[rt]39..32)7..0
endif
if ((GPR[rs]47..40 - GPR[rt]47..40) < 0x00) then
    GPR[rd]47..40 ← 0x00
else
    GPR[rd]47..40 ← (GPR[rs]47..40 - GPR[rt]47..40)7..0
endif

if ((GPR[rs]55..48 - GPR[rt]55..48) < 0x00) then
    GPR[rd]55..48 ← 0x00
else
    GPR[rd]55..48 ← (GPR[rs]55..48 - GPR[rt]55..48)7..0
endif

if ((GPR[rs]63..56 - GPR[rt]63..56) < 0x00) then
    GPR[rd]63..56 ← 0x00
else
    GPR[rd]63..56 ← (GPR[rs]63..56 - GPR[rt]63..56)7..0
endif

if ((GPR[rs]71..64 - GPR[rt]71..64) < 0x00) then
    GPR[rd]71..64 ← 0x00
else
    GPR[rd]71..64 ← (GPR[rs]71..64 - GPR[rt]71..64)7..0
endif

if ((GPR[rs]79..72 - GPR[rt]79..72) < 0x00) then
    GPR[rd]79..72 ← 0x00
else
    GPR[rd]79..72 ← (GPR[rs]79..72 - GPR[rt]79..72)7..0
endif

if ((GPR[rs]87..80 - GPR[rt]87..80) < 0x00) then
    GPR[rd]87..80 ← 0x00
else
    GPR[rd]87..80 ← (GPR[rs]87..80 - GPR[rt]87..80)7..0
endif

if ((GPR[rs]95..88 - GPR[rt]95..88) < 0x00) then
    GPR[rd]95..88 ← 0x00
else
    GPR[rd]95..88 ← (GPR[rs]95..88 - GPR[rt]95..88)7..0
endif

if ((GPR[rs]103..96 - GPR[rt]103..96) < 0x00) then
    GPR[rd]103..96 ← 0x00
else
    GPR[rd]103..96 ← (GPR[rs]103..96 - GPR[rt]103..96)7..0
endif

if ((GPR[rs]111..104 - GPR[rt]111..104) < 0x00) then
    GPR[rd]111..104 ← 0x00
else
    GPR[rd]111..104 ← (GPR[rs]111..104 - GPR[rt]111..104)7..0
endif
if ($(GPR[rs]_{119..112} - GPR[rt]_{119..112}) < 0x00)$ then
    $GPR[rd]_{119..112} \leftarrow 0x00$
else
    $GPR[rd]_{119..112} \leftarrow (GPR[rs]_{119..112} - GPR[rt]_{119..112})_{7..0}$
endif

if ($(GPR[rs]_{127..120} - GPR[rt]_{127..120}) < 0x00)$ then
    $GPR[rd]_{127..120} \leftarrow 0x00$
else
    $GPR[rd]_{127..120} \leftarrow (GPR[rs]_{127..120} - GPR[rt]_{127..120})_{7..0}$
endif

* Saturate to unsigned byte

Exceptions:

None
PSUBUH
Parallel Subtract with Unsigned Saturation Halfword

<table>
<thead>
<tr>
<th>MMI</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>PSUBUH</th>
<th>MMI1</th>
</tr>
</thead>
<tbody>
<tr>
<td>011100</td>
<td>10101</td>
<td>101000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: PSUBUH rd, rs, rt

Purpose: To subtract 8 pairs of 16-bit unsigned integers with saturation in parallel.

Description: rd ← rs − rt

The eight unsigned halfwords in GPR rt are subtracted from the corresponding eight unsigned halfwords in GPR rs in parallel. The results are placed into the corresponding eight halfwords in GPR rd.

No underflow exceptions are generated under any circumstances. Results beyond the range of an unsigned halfword value are saturated according to the following:

Underflow: 0x0000

This instruction operates on 128-bit registers.

Operation:
if ((GPR[rs]15..0 − GPR[rt]15..0) < 0x0000) then
    GPR[rd]15..0 ← 0x0000
else
    GPR[rd]15..0 ← (GPR[rs]15..0 − GPR[rt]15..0)15..0
endif

if ((GPR[rs]31..16 − GPR[rt]31..16) < 0x0000) then
    GPR[rd]31..16 ← 0x0000
else
    GPR[rd]31..16 ← (GPR[rs]31..16 − GPR[rt]31..16)15..0
endif

if ((GPR[rs]47..32 − GPR[rt]47..32) < 0x0000) then
    GPR[rd]47..32 ← 0x0000
else
    GPR[rd]47..32 ← (GPR[rs]47..32 − GPR[rt]47..32)15..0
endif

if ((GPR[rs]63..48 − GPR[rt]63..48) < 0x0000) then
    GPR[rd]63..48 ← 0x0000
else
    GPR[rd]63..48 ← (GPR[rs]63..48 − GPR[rt]63..48)15..0
endif

if ((GPR[rs]79..64 − GPR[rt]79..64) < 0x0000) then
    GPR[rd]79..64 ← 0x0000
else
    GPR[rd]79..64 ← (GPR[rs]79..64 − GPR[rt]79..64)15..0
endif
if \((GPR[rs]_{95..80} - GPR[rt]_{95..80}) < 0x0000\) then
   \(GPR[rd]_{95..80} \leftarrow 0x0000\)
else
   \(GPR[rd]_{95..80} \leftarrow (GPR[rs]_{95..80} - GPR[rt]_{95..80})_{15..0}\)
endif

if \((GPR[rs]_{111..96} - GPR[rt]_{111..96}) < 0x0000\) then
   \(GPR[rd]_{111..96} \leftarrow 0x0000\)
else
   \(GPR[rd]_{111..96} \leftarrow (GPR[rs]_{111..96} - GPR[rt]_{111..96})_{15..0}\)
endif

if \((GPR[rs]_{127..112} - GPR[rt]_{127..112}) < 0x0000\) then
   \(GPR[rd]_{127..112} \leftarrow 0x0000\)
else
   \(GPR[rd]_{127..112} \leftarrow (GPR[rs]_{127..112} - GPR[rt]_{127..112})_{15..0}\)
endif

* Saturate to unsigned halfword

Exceptions:

None
Appendix B  C790-Specific Instruction Set Details

PSUBUW  Parallel Subtract with Unsigned Saturation Word

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PSUBUW</td>
<td>MMI1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td></td>
<td></td>
<td></td>
<td>10001</td>
<td>101000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:  PSUBUW  rd, rs, rt
Purpose: To subtract 4 pairs of 32-bit unsigned integers with saturation in parallel.
Description:  rd ← rs − rt

The four unsigned words in GPR rt are subtracted from the corresponding four unsigned words in GPR rs in parallel. The results are placed into the corresponding four words in GPR rd.

No underflow exceptions are generated under any circumstances. Results beyond the range of an unsigned word value are saturated according to the following:

Underflow:  0x00000000

This instruction operates on 128-bit registers.

Operation:

if ((GPR[rs]31..0 − GPR[rt]31..0) < 0x00000000) then
  GPR[rd]31..0 ← 0x00000000
else
  GPR[rd]31..0 ← (GPR[rs]31..0 − GPR[rt]31..0)31..0
endif

if ((GPR[rs]63..32 − GPR[rt]63..32) < 0x00000000) then
  GPR[rd]63..32 ← 0x00000000
else
  GPR[rd]63..32 ← (GPR[rs]63..32 − GPR[rt]63..32)31..0
endif

if ((GPR[rs]95..64 − GPR[rt]95..64) < 0x00000000) then
  GPR[rd]95..64 ← 0x00000000
else
  GPR[rd]95..64 ← (GPR[rs]95..64 − GPR[rt]95..64)31..0
endif

if ((GPR[rs]127..96 − GPR[rt]127..96) < 0x00000000) then
  GPR[rd]127..96 ← 0x00000000
else
  GPR[rd]127..96 ← (GPR[rs]127..96 − GPR[rt]127..96)31..0
endif
### Exceptions:

None
PSUBW

Parallel Subtract Word

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI</td>
<td>011100</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>PSUBW</td>
<td>00001</td>
<td>MMI0</td>
<td>00100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6  5  5  5  5  6

C790

Format: PSUBW  rd, rs, rt

Purpose: To subtract 4 pairs of 32-bit integers in parallel.

Description: rd ← rs − rt

The four signed words in GPR rt are subtracted from the corresponding four words in GPR rs in parallel. The results are placed into the corresponding four words in GPR rd.

No overflow or underflow exceptions are generated under any circumstances.

This instruction operates on 128-bit registers.

Operation:

- GPR[rd]31..0 ← (GPR[rs]31..0 − GPR[rt]31..0)
- GPR[rd]63..32 ← (GPR[rs]63..32 − GPR[rt]63..32)
- GPR[rd]95..64 ← (GPR[rs]95..64 − GPR[rt]95..64)
- GPR[rd]127..96 ← (GPR[rs]127..96 − GPR[rt]127..96)

Exceptions:

None
PXOR
Parallel Exclusive OR

Format: PXOR rd, rs, rt
Purpose: To do a bitwise logical EXCLUSIVE OR.
Description: rd ← rs XOR rt

The contents of GPR rs are combined with the contents of GPR rt in a bitwise logical
exclusive OR operation. The result is placed into GPR rd.

This instruction operates on 128-bit registers.

Operation:
GPR[rd]_{127..0} ← GPR[rs]_{127..0} XOR GPR[rt]_{127..0}

Exceptions:
None
Quadword Funnel Shift Right Variable

Format: QFSRV rd, rs, rt

Purpose: To right shift a quadword by a variable number of bits.

Description: rd ← (rs, rt) >> SA

The content of GPR rt is concatenated with the content of GPR rs producing the intermediate result rs:rt. This value is shifted right by the number of bits specified in the shift amount register SA. The least significant 16 bytes (i.e. quadword) of the shifted result is placed into GPR rd.

Restriction:

Note that SA can be loaded only with byte shift values (MTSAB) or halfword shift values (MTSAH); i.e. with bit shift amounts that are multiples of 8 or 16.

This instruction operates on 128-bit registers.

Operation:

if ( SA == 0 ) then
   GPR[rd]127..0 ← GPR[rt]127..0
else
   GPR[rd]127..0 ← GPR[rs](SA − 1)..0 || GPR[rt]127..SA
endif

Programming Note:

1. A left funnel shift by an amount of s bytes can be done by setting SA to 16-s using the MTSAB instruction, provided that s is not 0. Similarly, a left funnel shift by s halfwords can be done by setting SA to 8-s using the MTSAH instruction, provided that s is not 0. A quick way to perform this computation is as follows:
   // Register %sal contains the left shift amount
   subi %samt, %sal, 1
   mtsab%samt, -1
   // Following QFSRV does a shift left by %sal bytes
   qfsrv %dst, %src1, %src2

2. QFSRV can be used to rotate a 128-bit quantity r by setting both source operands rs and rt to register r. For example, the following code sequence rotates right the value in wide register %5 by 3 halfwords (i.e. 48 bits), and deposits the result in wide register %6.

   mtsah %0, 3
   qfsrv %6, %5, %5
SQ  Store Quadword  SQ

31  26  25  21  20  16  15  0

SQ  base  rt  offset

011111

6  5  5  16

Format:  SQ  rt, offset (base)
Purpose:  To store a quadword to memory.
Description:  memory [base + offset] ← rt

The 128-bit quadword in GPR rt is stored in memory at the location specified by the effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address. The least significant four bits of the effective address are masked to zero (effectively creating an aligned address) before being used to access memory. No address exceptions due to alignment are possible.

Restrictions:

The effective address doesn't have to be naturally aligned. The least significant 4 bits of the effective address are ignored.

Operation:

\[
\text{vAddr} \leftarrow \text{sign\_extend}\ (\text{offset}) + \text{GPR}\[\text{base}\]_{31..0}
\text{vAddr}^{3..0} = 0^4
\]

(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
quadword ← GPR[rt]_{127..0}
StoreMemory (uncached, QUADWORD, quadword, pAddr, vAddr, DATA)

Exceptions:

- TLB Refill
- TLB Invalid
- Address Error

Programming Notes:

None
## B.5 C790-Specific Instruction Encoding

**OpCode**

<table>
<thead>
<tr>
<th>bits 28..26</th>
<th>Instructions encoded by <strong>OpCode</strong> field (MMI, LQ, SQ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 26 5 0</td>
<td></td>
</tr>
<tr>
<td>0 000</td>
<td>SPECIAL, REGIMM, J, JAL, BEQ, BNE, BLEZ, BGTZ</td>
</tr>
<tr>
<td>0 001</td>
<td>ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI, LUI</td>
</tr>
<tr>
<td>2 010</td>
<td>COP0, COP1, *BEQ, *BNE, *BLEZ, *BGTZ</td>
</tr>
<tr>
<td>3 011</td>
<td>DADDI, DADDIU, LDL, LDR, *MMI, *LQ, *SQ</td>
</tr>
<tr>
<td>4 100</td>
<td>LB, LH, LWL, LW, LBU, LHU, LWR, LWU</td>
</tr>
<tr>
<td>5 101</td>
<td>SB, SH, SWL, SW, SDL, SDR, SWR, CACHE</td>
</tr>
<tr>
<td>6 110</td>
<td>η, LWC1, *PREF, *LD, *LDC1</td>
</tr>
<tr>
<td>7 111</td>
<td>η, SWC1, *SDF, *SD</td>
</tr>
</tbody>
</table>

**function**

<table>
<thead>
<tr>
<th>bits 2..0</th>
<th>Instructions encoded by <strong>function</strong> field when OpCode field = MMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 3 0 000</td>
<td>MADD, MADDU, *PLZCW, *PLZCW, *PLZCW</td>
</tr>
<tr>
<td>1 001</td>
<td>*MMI0, *MMI2</td>
</tr>
<tr>
<td>2 010</td>
<td>MFHI1, MTHI1, MFLO1, MTLO1, *DIV1, *DIV1</td>
</tr>
<tr>
<td>3 011</td>
<td>*MULT1, *MULTU1, *DIV1, *DIV1</td>
</tr>
<tr>
<td>4 100</td>
<td>*MADD1, *MADDU1, *MADDU1, *MADDU1</td>
</tr>
<tr>
<td>5 101</td>
<td>*MMI1, *MMI2</td>
</tr>
<tr>
<td>7 111</td>
<td>*PSLLW, *PSRLW, *PSRAW</td>
</tr>
</tbody>
</table>
### Instructions encoded by `function` field when `OpCode field = MMI` & bit 5..0 = MMI0

<table>
<thead>
<tr>
<th>bits 0..3</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>10..8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 000</td>
<td>PADDW</td>
<td>PSUBW</td>
<td>PCGTW</td>
<td>PMAXW</td>
</tr>
<tr>
<td>1 001</td>
<td>PADDH</td>
<td>PSUBH</td>
<td>PCGTH</td>
<td>PMAXH</td>
</tr>
<tr>
<td>2 010</td>
<td>PADDB</td>
<td>PSUBB</td>
<td>PCGTB</td>
<td>*</td>
</tr>
<tr>
<td>3 011</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>4 100</td>
<td>PADDSW</td>
<td>PSUBSW</td>
<td>PEXTLW</td>
<td>PPACW</td>
</tr>
<tr>
<td>5 101</td>
<td>PADDSH</td>
<td>PSUBSH</td>
<td>PEXTLH</td>
<td>PPACH</td>
</tr>
<tr>
<td>6 110</td>
<td>PADDSB</td>
<td>PSUBSB</td>
<td>PEXTLB</td>
<td>PPACB</td>
</tr>
<tr>
<td>7 111</td>
<td>*</td>
<td>*</td>
<td>PEXT5</td>
<td>PPAC5</td>
</tr>
</tbody>
</table>

### Instructions encoded by `function` field when `OpCode field = MMI` & bit 5..0 = MMI1

<table>
<thead>
<tr>
<th>bits 0..3</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>10..8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 000</td>
<td>*</td>
<td>PABSW</td>
<td>PCEQW</td>
<td>PMINW</td>
</tr>
<tr>
<td>1 001</td>
<td>PADS BH</td>
<td>PABSH</td>
<td>PCEQH</td>
<td>PMINH</td>
</tr>
<tr>
<td>2 010</td>
<td>*</td>
<td>*</td>
<td>PCEQB</td>
<td>*</td>
</tr>
<tr>
<td>3 011</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>4 100</td>
<td>PADDUW</td>
<td>PSUBUW</td>
<td>PEXTUW</td>
<td>*</td>
</tr>
<tr>
<td>5 101</td>
<td>PADDUH</td>
<td>PSUBUH</td>
<td>PEXTUH</td>
<td>*</td>
</tr>
<tr>
<td>6 110</td>
<td>PADDUB</td>
<td>PSUBUB</td>
<td>PEXTUB</td>
<td>OFSRV</td>
</tr>
<tr>
<td>7 111</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
### Appendix B  C790-Specific Instruction Set Details

#### Function

<table>
<thead>
<tr>
<th>bits 7..6</th>
<th>Instructions encoded by function field when OpCode field = MMI &amp; bit 5..0 = MMI2</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 26</td>
<td>10 6 5 0</td>
</tr>
<tr>
<td>OpCode</td>
<td>function</td>
</tr>
<tr>
<td>MMI</td>
<td>MMI2</td>
</tr>
</tbody>
</table>

#### Function

<table>
<thead>
<tr>
<th>bits 7..6</th>
<th>Instructions encoded by function field when OpCode field = MMI &amp; bit 5..0 = MMI3</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 26</td>
<td>10 6 5 0</td>
</tr>
<tr>
<td>OpCode</td>
<td>function</td>
</tr>
<tr>
<td>MMI</td>
<td>MMI3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bits 10..8</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>PMADDW</td>
<td>*</td>
<td>*</td>
<td>PSLLVW</td>
</tr>
<tr>
<td>1 001</td>
<td>PMSUBW</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>2 010</td>
<td>PMFHI</td>
<td>PMFLO</td>
<td>PINTH</td>
<td>*</td>
</tr>
<tr>
<td>3 011</td>
<td>PMULTW</td>
<td>PDIVW</td>
<td>PCPYLD</td>
<td>*</td>
</tr>
<tr>
<td>4 100</td>
<td>PMADDH</td>
<td>PHMADH</td>
<td>PAND</td>
<td>PXOR</td>
</tr>
<tr>
<td>5 101</td>
<td>PMSUBH</td>
<td>PHMSBH</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>6 110</td>
<td>*</td>
<td>*</td>
<td>PEXEH</td>
<td>PREVH</td>
</tr>
<tr>
<td>7 111</td>
<td>PMULTH</td>
<td>PDIVBW</td>
<td>PEXEW</td>
<td>PROT3W</td>
</tr>
</tbody>
</table>

* This OpCode is reserved for future use. An attempt to execute it causes a Reserved Instruction exception.

δ This OpCode indicates an instruction class. The instruction word must be further decoded by examining additional tables that show the values for another instruction fields.

η This OpCode is reserved for one of the following instructions which are currently not supported: DMULT, DMULTU, DDIV, DDIVU, LL, LLD, SC, SCD, LWC2, SWC2. An attempt to execute it causes a Reserved Instruction exception.
C. COP0 System Control Coprocessor Instruction Set Details

This appendix provides a detailed description of the operation of each System Control Coprocessor (COP0) instruction.

COP0 instructions perform operations specifically on the System Control Coprocessor registers to manipulate the memory management and exception handing facilities of the processor.

COP0 Coprocessor instructions are enabled if the processor is in Kernel mode, or if bit 28 (CU[0]) is set in the Status register. Otherwise, executing one of these instructions generates a Coprocessor Unusable exception. The only exception to this rule are the EI and the DI instructions which never generate Coprocessor Unusable exceptions.

When the EDI bit in the Status register is set, the EI and DI instructions operate in User, Supervisor, and Kernel modes independent of whether COP0 coprocessor usable bit (Status.CU[0]) is set or not. When the EDI bit is cleared EI and DI work as NOPs in User and Supervisor modes independent of whether COP0 coprocessor usable bit (Status.CU[0]) is set or not, and executes properly in Kernel mode.
Branch on Coprocessor 0 False

Format: BC0F offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and 16-bit offset, shifted left two bits and sign-extended. If coprocessor 0’s condition signal, as sampled during the previous instruction, is false, then the program branches to the target address with a delay of one instruction.

Restrictions:

Because the coprocessor 0 condition is externally supplied, there is no way to synchronize the change/update of the condition and the execution of this instruction.

Operation:

I:   tgt_offset ← sign_extend (offset || 0²)
     condition ← not CPCOND0

I+1: if condition then
     PC ← PC + tgt_offset
     endif

Exceptions:

Coprocessor Unusable exception
BC0FL Branch on Coprocessor 0 False Likely

MIPS II

Format: BC0FL offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the contents of coprocessor 0’s condition signal, as sampled during the previous instruction, is false, the program branches to the target address with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Restrictions:

Because the coprocessor 0 condition is externally supplied, there is no way to synchronize the change/update of the condition and the execution of this instruction.

Operation:

I: tgt_offset ← sign_extend (offset || 0^2)
   condition ← not CPCOND0

I+1: if condition then
      PC ← PC + tgt_offset
      endif

Exceptions:

Coprocessor Unusable exception
**BC0T**  
**Branch on Coprocessor 0 True**

<table>
<thead>
<tr>
<th></th>
<th>COP0</th>
<th>BC0</th>
<th>BC0T</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>010000</td>
<td>01000</td>
<td>00001</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td>0</td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:**

BC0T  offset

**Description:**

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit **offset**, shifted left two bits and sign-extended. If the coprocessor 0'z condition signal is true, then the program branches to the target address, with a delay of one instruction.

**Restrictions:**

Because the coprocessor 0 condition is externally supplied, there is no way to synchronize the change/update of the condition and the execution of this instruction.

**Operation:**

1: \[ tgt\_offset \leftarrow \text{sign\_extend}(\text{offset} \parallel 0^2) \]
   \[ \text{condition} \leftarrow \neg\text{CPCOND0} \]

1+1: if condition then

   \[ PC \leftarrow PC + tgt\_offset \]

   endif

**Exceptions:**

Coprocessor Unusable exception
### Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of coprocessor 0’s condition signal, as sampled during the previous instruction, is true, the program branches to target address with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

### Restrictions:

Because the coprocessor 0 condition is externally supplied, there is no way to synchronize the change/update of the condition and the execution of this instruction.

### Operation:

\[
\begin{align*}
\text{I:} & \quad \text{tgt\_offset} \leftarrow \text{sign\_extend} (\text{offset} \| 0^2) \\
& \quad \text{condition} \leftarrow \text{not CPCOND}0 \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \quad \text{PC} \leftarrow \text{PC} + \text{tgt\_offset} \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{NullifyCurrentInstruction()} \\
& \quad \quad \text{endif}
\end{align*}
\]

### Exceptions:

- Coprocessor Unusable exception

---

**BC0TL**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>BC0</td>
<td>BC0TL</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000</td>
<td>01000</td>
<td>00011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS II**

Format: BC0TL offset

Description: A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of coprocessor 0’s condition signal, as sampled during the previous instruction, is true, the program branches to target address with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Restrictions: Because the coprocessor 0 condition is externally supplied, there is no way to synchronize the change/update of the condition and the execution of this instruction.

Operation:

\[
\begin{align*}
\text{I:} & \quad \text{tgt\_offset} \leftarrow \text{sign\_extend} (\text{offset} \| 0^2) \\
& \quad \text{condition} \leftarrow \text{not CPCOND}0 \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \quad \text{PC} \leftarrow \text{PC} + \text{tgt\_offset} \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{NullifyCurrentInstruction()} \\
& \quad \quad \text{endif}
\end{align*}
\]

Exceptions:

- Coprocessor Unusable exception
CACHE

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>OpCode</th>
<th>CACHE Instruction</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>IXIN</td>
<td>00111</td>
<td>INDEX INVALIDATE</td>
<td>Instruction Cache</td>
</tr>
<tr>
<td>IXLTG</td>
<td>00000</td>
<td>INDEX LOAD TAG</td>
<td>Instruction Cache</td>
</tr>
<tr>
<td>IXSTG</td>
<td>00100</td>
<td>INDEX STORE TAG</td>
<td>Instruction Cache</td>
</tr>
<tr>
<td>IHIN</td>
<td>01011</td>
<td>HIT INVALIDATE</td>
<td>Instruction Cache</td>
</tr>
<tr>
<td>IFL</td>
<td>01110</td>
<td>FILL</td>
<td>Instruction Cache</td>
</tr>
<tr>
<td>IXLDT</td>
<td>00001</td>
<td>INDEX LOAD DATA</td>
<td>Instruction Cache</td>
</tr>
<tr>
<td>IXSDT</td>
<td>00101</td>
<td>INDEX STORE DATA</td>
<td>Instruction Cache</td>
</tr>
<tr>
<td>BXLBT</td>
<td>00010</td>
<td>INDEX LOAD BTAC</td>
<td>BTAC</td>
</tr>
<tr>
<td>BXSBT</td>
<td>00110</td>
<td>INDEX STORE BTAC</td>
<td>BTAC</td>
</tr>
<tr>
<td>BFH</td>
<td>01100</td>
<td>BTAC FLUSH</td>
<td>BTAC</td>
</tr>
<tr>
<td>BHNBT</td>
<td>01010</td>
<td>HIT INVALIDATE BTAC</td>
<td>BTAC</td>
</tr>
<tr>
<td>DXWBIN</td>
<td>10100</td>
<td>INDEX WRITE BACK INVALIDATE</td>
<td>Data Cache</td>
</tr>
<tr>
<td>DXLTG</td>
<td>10000</td>
<td>INDEX LOAD TAG</td>
<td>Data Cache</td>
</tr>
<tr>
<td>DXSTG</td>
<td>10010</td>
<td>INDEX STORE TAG</td>
<td>Data Cache</td>
</tr>
<tr>
<td>DXIN</td>
<td>10110</td>
<td>INDEX INVALIDATE</td>
<td>Data Cache</td>
</tr>
<tr>
<td>DHIN</td>
<td>11010</td>
<td>HIT INVALIDATE</td>
<td>Data Cache</td>
</tr>
<tr>
<td>DHWBIN</td>
<td>11000</td>
<td>HIT WRITEBACK INVALIDATE</td>
<td>Data Cache</td>
</tr>
<tr>
<td>DXLDT</td>
<td>10001</td>
<td>INDEX LOAD DATA</td>
<td>Data Cache</td>
</tr>
<tr>
<td>DXSDT</td>
<td>10011</td>
<td>INDEX STORE DATA</td>
<td>Data Cache</td>
</tr>
<tr>
<td>DHWOIN</td>
<td>11100</td>
<td>HIT WRITEBACK W/O INVALIDATE</td>
<td>Data Cache</td>
</tr>
</tbody>
</table>
Appendix C  COP0 System Control Coprocessor Instruction Set Details

Operation:
\[ \text{vAddr} \leftarrow (\text{offset}_{15})_{16} \|	ext{offset}_{15..0} + \text{GPR[base]}_{31..0} \]
\[ (\text{pAddr, uncached}) \leftarrow \text{AddressTranslation (vAddr, DATA)} \]
\[ \text{CacheOp (op, vAddr, pAddr)} \]

Exceptions:
- Coprocessor Unusable exception
- TLB Refill
- TLB Invalid
- Address Error

C.1.1 Notes on the CACHE Instruction Sub-operations

Cache Virtual Address
The CACHE instruction uses the following portions of the Virtual Address (VA) computed by adding the offset to the base to specify a cache block and way:
- VA[13:6] defines a 64-byte line in the data cache array
- VA[13:6] defines a 64-byte line in the instruction cache array
- In both cases, VA[0] defines the way needed by Index sub-operations

When accessing data in the caches, VA[13:2] is used to read or write a specific data word in the data cache and VA[13:2] is used to read or write a specific instruction in the instruction cache.

Cache Physical Address
The CACHE instruction computes the Physical Address (PA) to access memory for cache Hit Invalidate (I) and Fill (I) sub-operations in the following manner:
- VA[31:6] is computed from the CACHE instruction by adding the offset to the base and then the result is translated to produce PA[31:6]

The CACHE instruction computes the Physical Address (PA) to access memory for cache Hit Invalidate (D), Hit Writeback Invalidate (D), Hit Writeback Without Invalidate (D) sub-operations in the following manner:
- VA[31:6] is computed from the CACHE instruction by adding the offset to the base and then the result is translated to produce PA[31:6]

BTAC Virtual Address
The CACHE instruction uses the following portions of the Virtual Address (VA) computed by adding the offset to the base to check if there is an entry that matches the VA:
- VA[31:3] defines an entry in the BTAC

BTAC Index Bits
Since the BTAC has 64 entries the VA[5:0] computed from the CACHE instruction by adding the offset to the base is used to index the BTAC.

COP0 Not Usable
If COP0 is not usable (if not in Kernel mode, Status.CU0 must be set for COP0 to be usable), a Coprocessor unusable exception is taken.
TLB Exceptions on Cache Operations

TLB Refill and TLB Invalid exceptions can occur only for the following sub-operations:

1. Hit Invalidate (I)
2. Fill (I)
3. Hit Invalidate (D)
4. Hit Writeback Invalidate (D)
5. Hit Writeback without Invalidate (D)

The TLB Modified exception is never generated.

Hit Sub-operation Accesses

A Hit sub-operation accesses the specified cache as a normal data reference, and performs the specified operation if the cache line contains valid data at the specified physical address (a hit). The operation is undefined if a CACHE sub-operation hit occurs in both ways of the cache.

Breakpoint Exception

Breakpoint exceptions can not be generated by any of the CACHE sub-operations (note that an Instruction Address Breakpoint can still be done on the CACHE instruction itself).

Address Error Exception

None of the CACHE sub-operations will generate an Address Error exception due to misalignment of the VA created by the CACHE instruction as described above. The following CACHE sub-operations can generate privilege-type Address Error exceptions:

1. Hit Invalidate (I)
2. Fill (I)
3. Hit Invalidate (D)
4. Hit Writeback Invalidate (D)
5. Hit Writeback without Invalidate (D)
C.1.2 Sub-Operation Descriptions

Note on Cache Enable Status

All Instruction cache related suboperations perform their function regardless of the value of the ICE bit of the Config register. (i.e., regardless of whether the Instruction cache is enabled or not.)

All data cache related suboperations perform their function regardless of the value of the DCE bit of the Config register. (i.e., regardless of whether the data cache is enabled or not.)

All BTAC-related suboperations perform their function regardless of the value of the BPE bit of the Config register.

Op = 00111 Index Invalidate (I)

Index Invalidate (I) sets a line in the instruction cache to Invalid. VA[13:6] defines the index of the line and VA[0] defines the way to be invalidated. The LRF bit does not change.

Op = 00000 Index Load Tag (I)

Index Load Tag (I) reads the instruction cache tag array fields into the COP0 TagLO register. VA[13:6] defines the index and VA[0] defines the way of the tag to be read. The following mapping defines the sub-operation:

- TagLO[4] = LRF bit
- TagLO[5] = VALID bit
- TagLO[31:12] = Tag[19:0]

All other TagLO bits are undefined.

Op = 00100 Index Store Tag (I)

Index Store Tag (I) stores the COP0 TagLO register into the instruction cache tag array. VA[13:6] defines the index and VA[0] defines the way of the tag to be read. The following mapping defines the sub-operation:

- LRF bit = TagLO[4]
- VALID bit = TagLO[5]
- Tag[19:0] = TagLO[31:12]

Note that it is perfectly feasible to invalidate the cache line using this sub-operation.

Op = 01011 Hit Invalidate (I)

Hit Invalidate (I) invalidates a line in the instruction cache which matches the PA[31:6] computed from the CACHE instruction. Both way tags at VA[13:6] are read from the instruction cache.

If the Valid bit of one of the entries is a 1 and the PA of the CACHE instruction matches the Tag from that entry of the instruction cache tag array, the Valid bit of the entry is changed to a 0 (Invalid). The LRF bit does not change. This sub-operation also invalidates BTAC entries which match VA[31:6].
Op = 01110  Fill (I)

Fill (I) brings in a cache line from memory and stores it in the instruction cache. The following sequence is followed:
1. The PA computed from the CACHE instruction is used to fetch the cache line from memory.
2. The line is loaded into the cache line addressed by VA[13:6] and the way of cache is defined by the rules of the LRF bits.
3. The corresponding instruction cache tag is loaded with the PFN and the entry is validated.

Op = 00001  Index Load Data (I)

Index Load Data (I) reads a single instruction from the instruction cache data array and stores it into the COP0 TagLO and TagHI registers. VA[13:2] defines the index and VA[0] defines the way of the instruction cache to be read. The following mapping defines the sub-operation:
- TagLO[31:0] = 32-bit instruction
- TagHI[3:0] = SteeringBits[3:0]
- TagHI[5:4] = BHT[1:0]

All other TagHI bits are undefined.

Op = 00101  Index Store Data (I)

Index Store Data (I) stores the COP0 TagLO and TagHI registers into the instruction cache data array.

VA[13:2] defines the index and VA[0] defines the way of the instruction cache to be written. The following mapping defines the sub-operation:
- 32-bit instruction = TagLO[31:0]
- SteeringBits[3:0] = TagHI[3:0]

The BHT[1:0] bits are associated with the instruction pair at VA[13:3]. This sub-operation invalidates all BTAC entries.

Op = 00010  Index Load BTAC (B)

Index Load BTAC (B) reads a single BTAC entry and stores it into the COP0 TagLO registers. VA[5:0] defines the index of the BTAC entry to be read. The following mapping defines the sub-operation:
- TagLO[0] = Valid Bit
- TagLO[31:3] = FetchAddress[28:0]
- TagHI[31:2] = TargetAddress[29:0]

All other TagLO and TagHI bits are undefined.
Op = 00110  Index Store BTAC (B)

Index Store BTAC (B) stores the COP0 TagLO and TagHI registers into a single BTAC entry. VA[5:0] defines the index of the BTAC entry to be written. The following mapping defines the sub-operation:

- Valid Bit = TagLO[0]
- FetchAddress[28:0] = TagLO[31:3]
- TargetAddress[29:0] = TagHI[31:2]

Op = 01100  BTAC Flush (B)

This sub-operation invalidates the complete BTAC by writing a 0 into the valid bits of all the entries of the BTAC.

Op = 01010  Hit Invalidate BTAC (B)

Hit Invalidate BTAC (B) invalidates an entry in the BTAC which matches the VA[31:3] computed from the CACHE instruction. If the VA[31:3] matches an entry in the BTAC and its Valid bit is equal to 1 then the Valid bit is changed to a 0. The result is undefined if there are plural of entries that matches the VA.

Op = 10100  Index Writeback Invalidate (D)

Index Writeback Invalidate (D) sub-operation sets a cache line in the data cache to Invalid and writes back any dirty data to the CPU bus. VA[13:6] defines the index and VA[0] defines the way of the data cache line to be invalidated. The invalidation takes place by writing a 0 to the Valid bit. The LRF bit does not change.

The PA where the cache line will be written to is calculated by appending VA[11:6] to the 20-bit PFN field from the data cache tag to form PA[31:6]. This address represents a cache line address.

Op = 10000  Index Load Tag (D)

Index Load Tag (D) reads the data cache tag array fields into the COP0 TagLO register. VA[13:6] defines the index and VA[0] defines the way of the tag to be read. The following mapping defines the sub-operation:

- TagLO[3] = Lock bit
- TagLO[4] = LRF bit
- TagLO[31:12] = Tag[31:12]

All other TagLO bits are undefined.

Op = 10010  Index Store Tag (D)

Index Store Tag (D) stores the COP0 TagLO register into the data cache tag array. VA[13:6] defines the index and VA[0] defines the way of the tag to be written. The following mapping defines the sub-operation:

- Lock bit = TagLO[3]
- LRF bit = TagLO[4]
- Valid bit = TagLO[5]
- Tag[19:0] = TagLO[31:12]
Op = 10110  Index Invalidate (D)

Index Invalidate (D) sets a line in the data cache to Invalid. VA[13:6] defines the index of the line and VA[0] defines the way to be invalidated. The Lock bit, Dirty bit, and Valid bit are changed to zero. The LRF bit doesn’t change.

Op = 11010  Hit Invalidate (D)

Hit Invalidate (D) invalidates an entry in the data cache which matches the PA computed from the CACHE instruction. Both way tags at VA[13:6] are read from the data cache.

If the Valid bit of the entry is one and the PA of the CACHE instruction matches the Tag from the data cache tag array, the Valid bit of the entry is changed to zero (Invalid). The Lock bit and Dirty bit are also changed to zero. The LRF bit does not change.

Op = 11000  Hit Writeback Invalidate (D)

Hit Writeback Invalidate (D) sub-operation invalidates an entry in the data cache which matches the PA computed from the CACHE instruction. Additionally it writes back any dirty data to the CPU bus. Both way tags at VA[13:6] are read from the data cache. The Lock bit, Dirty bit, and Valid bit are changed to zero. The LRF bits are not modified.

If the PA computed from the CACHE instruction matches the tag from the data cache tag array and the Valid bit is 1 then the Valid bit is changed to 0. Furthermore if the Dirty bit is 1 then the cache line is written to the physical address calculated by appending VA[11:6] to the 20-bit PFN field from the data cache tag to form PA[31:6]. This address represents a cache line physical address.

Op = 10001  Index Load Data (D)

Index Load Data (D) reads a single word from the data cache data array and stores it into the COP0 TagLO register. VA[13:2] defines the index and VA[0] defines the way of the data cache to be read. The following mapping defines the sub-operation:

- TagLO[31:0] = 32-bit data

Op = 10011  Index Store Data (D)

Index Store Data (D) stores the COP0 TagLO register into the data cache data array. VA[13:2] defines the index and VA[0] defines the way of the data cache to be written. The following mapping defines the sub-operation:

- 32-bit data = TagLO[31:0]

Op = 11100  Hit Writeback Without Invalidate (D)

Hit Writeback Without Invalidate (D) sub-operation writes back any dirty data to the CPU bus. Both way tags at VA[13:6] are read from the data cache. The Dirty bit is changed to zero. The LRF bits are not modified.

If the PA computed from the CACHE instruction matches the tag from the data cache tag array and the Valid and Dirty bits are 1 then the cache line is written to the physical address calculated by appending VA[11:6] to the 20-bit PFN field from the data cache tag to form PA[31:6]. This address represents a cache line physical address.
Programming Notes:

For all CACHE sub-operations which operate on the instruction cache the following programming restrictions have to be followed:

1. A sequence of CACHE instructions has to be directly preceded and followed by a SYNC.P instruction.
2. Each individual FILL sub-operation has to be followed by a SYNC.L instruction.

For all CACHE sub-operations which operate on the data cache the following programming restrictions have to be followed:

1. A sequence of CACHE instructions have to be directly preceded and followed by a SYNC.L instruction.
2. Each of the three WRITEBACK sub-operations have to be individually followed by a SYNC.L instruction.

For all CACHE sub-operations which operate on the BTAC the following programming restrictions have to be followed:

1. A sequence of CACHE instructions have to be directly preceded and followed by a SYNC.P instruction.

C.1.3 Updates of Data Tag Status Bits

The following table summarizes the updates of Data Tag status bits for various Cache sub-operations. The values in the table for Hit Writeback Invalidate, Hit Writeback Without Invalidate, and Hit Invalidate only apply if there is a hit in the data cache. If there is no hit, the status bits are unchanged.

<table>
<thead>
<tr>
<th>Cache Instruction</th>
<th>LRF Bit</th>
<th>Lock Bit</th>
<th>Dirty Bit</th>
<th>Valid Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Load Data</td>
<td>unchanged</td>
<td>unchanged</td>
<td>unchanged</td>
<td>unchanged</td>
</tr>
<tr>
<td>Index Store Data</td>
<td>unchanged</td>
<td>unchanged</td>
<td>unchanged</td>
<td>unchanged</td>
</tr>
<tr>
<td>Index Load Tag</td>
<td>unchanged</td>
<td>unchanged</td>
<td>unchanged</td>
<td>unchanged</td>
</tr>
<tr>
<td>Index Store Tag</td>
<td>loaded</td>
<td>loaded</td>
<td>loaded</td>
<td>loaded</td>
</tr>
<tr>
<td>Index Writeback Invalidate</td>
<td>unchanged</td>
<td>cleared</td>
<td>cleared</td>
<td>cleared</td>
</tr>
<tr>
<td>Index Invalidate</td>
<td>unchanged</td>
<td>cleared</td>
<td>cleared</td>
<td>cleared</td>
</tr>
<tr>
<td>Hit Invalidate</td>
<td>unchanged</td>
<td>cleared</td>
<td>cleared</td>
<td>cleared</td>
</tr>
<tr>
<td>Hit Writeback Invalidate</td>
<td>unchanged</td>
<td>cleared</td>
<td>cleared</td>
<td>cleared</td>
</tr>
<tr>
<td>Hit Writeback Without Invalidate</td>
<td>unchanged</td>
<td>cleared</td>
<td>cleared</td>
<td>unchanged</td>
</tr>
</tbody>
</table>
DI

**Format:**

DI

**Description:**

DI instruction clears the EIE bit in the Status register and disable all interrupts (except NMI and SIO). When the EIE bit is cleared, all interrupts are disabled regardless of the value of IE bit in the Status register.

When the EDI bit in the Status register is set, the DI instruction operates in User, Supervisor, and Kernel modes independent of whether COP0 coprocessor usable bit (Status.CU[0]) is set or not. When this bit is cleared EI and DI work as NOPs in User and Supervisor modes independent of whether COP0 coprocessor usable bit (Status.CU[0]) is set or not, and executes properly in Kernel mode.

**Operation:**

\[
\text{If (Status.EDI = 1) || (Status.EXL = 1) || (Status.ERL = 1) || (Status.KSU = 002) then}
\]

\[
\text{Status.EIE } \leftarrow 0
\]

**Exceptions:**

None
EI

Enable Interrupt

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>C0</td>
<td>0</td>
<td>000 0000 0000 0000</td>
<td>El</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000</td>
<td>10000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6 5 15 6

C790

Format: EI

Description:

EI instruction sets the EIE bit in the Status register. When the EIE bit is set, all interrupts are enabled if the IE bit in the Status register is 1, EXL bit is 0, and ERL bit is 0.

When the EDI bit in the Status register is set, the EI instruction operates in User, Supervisor, and Kernel modes independent of whether COP0 coprocessor usable bit (Status.CU[0]) is set or not. When this bit is cleared EI and DI work as NOPs in User and Supervisor modes independent of whether COP0 coprocessor usable bit (Status.CU[0]) is set or not, and executes properly in Kernel mode.

Operation:

If (Status.EDI = 1) || (Status.EXL = 1) || (Status.ERL = 1) || (Status.KSU = 002) then

Status.EIE ← 1

endif

Exceptions:

None
ERET Exception Return ERET

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>C0</td>
<td>0</td>
<td>000 0000 0000 0000</td>
<td>ERET</td>
<td>011000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R4000

Format: ERET

Description:

ERET is the instruction for returning from an interrupt, exception, or error trap. Unlike a branch or jump instruction, ERET does not execute the next instruction.

ERET must not itself be placed in a branch delay slot.

If the processor is servicing a Level 2 exception, then load the PC from the ErrorEPC and clear the ERL bit of the Status register (bit 2 in Status register). Otherwise (ERL = 0), load the PC from the EPC, and clear the EXL bit of the Status register (bit 1 in Status register).

Operation:

\[
\begin{align*}
\text{if Status.ERL} &= 1 \text{ then} \\
\quad & \quad \text{PC } \leftarrow \text{ErrorEPC} \\
\quad & \quad \text{Status.ERL } \leftarrow 0 \\
\text{else} \\
\quad & \quad \text{PC } \leftarrow \text{EPC} \\
\quad & \quad \text{Status.EXL } \leftarrow 0 \\
\text{endif}
\end{align*}
\]

Exceptions:

Coprocessor Unusable exception

Implementation Note:

ERET flushes the execution pipelines of the CPU before fetching the instruction from the target. Any pending loads, stores, ongoing multiplies, divides, multiply-accumulates and COP1 instructions are not flushed.

Programming Notes:

Any Reserved Instruction must not be placed in a branch delay slot just after ERET instruction. Please pay careful attention if any instruction is placed in the branch delay slot, because the instruction in the branch delay slot may be executed incompletely before flushing. It is commended that NOP is placed in the branch delay slot.
**MFBPC**

**Move from Breakpoint Control Register**

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>MF0</td>
<td>rt</td>
<td>Debug</td>
<td>0</td>
<td>MFBPC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000</td>
<td>00000</td>
<td>0000</td>
<td>0000000</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  
MFBPC rt

**Description:**  
The contents of the Breakpoint Control register of the COP0 are loaded into general register rt.

**Operation:**  
\[
data \leftarrow \text{CPR}[0, \text{Breakpoint Control}]
\]
\[
\text{GPR}[rt] \leftarrow (data_{31})^{32} \parallel data_{31.0}
\]

**Exceptions:**  
Coprocessor Unusable exception
**MFC0**

**Move from System Control Coprocessor**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>MF0</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Format:**

MFC0  rt, rd

**Description:**

The contents of coprocessor register rd of the COP0 are loaded into general register rt.

**Operation:**

1. data ← CPR[0, rd]
2. GPR[rt] ← (data31)\text{32} || data31.0

**Exceptions:**

Coprocessor Unusable exception
The contents of the Data Address Breakpoint register of the COP0 are loaded into general register rt.

Operation:

\[
data \leftarrow \text{CPR}[0, \text{Data Address Breakpoint}]
\]
\[
\text{GPR}[rt] \leftarrow (\text{data}_{31})^{32} \| \text{data}_{31.0}
\]

Exceptions:

Coprocessor Unusable exception
Appendix C  COP0 System Control Coprocessor Instruction Set Details

MFDABM  Move from Data Address Breakpoint Mask Register  MFDABM

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>3 2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>MF0</td>
<td>rt</td>
<td>Debug</td>
<td>0</td>
<td>0000 0000</td>
<td>MFDABM 101</td>
</tr>
<tr>
<td>010000</td>
<td>00000</td>
<td>Debug 11000</td>
<td>0</td>
<td>0000 0000</td>
<td>MFDABM 101</td>
<td></td>
</tr>
</tbody>
</table>

C790

Format:  MFDABM  rt

Description:

The contents of Data Address Breakpoint Mask register of the COP0 are loaded into general register rt.

Operation:

data ← CPR[0, Data Address Breakpoint Mask]
GPR[rt] ← (data31)32 || data31.0

Exceptions:

Coprocessor Unusable exception
MFDVB  Move from Data value Breakpoint Register  MFDVB

<table>
<thead>
<tr>
<th>COP0</th>
<th>MF0</th>
<th>rt</th>
<th>Debug</th>
<th>000000</th>
<th>MFDVB</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>00000</td>
<td>11000</td>
<td></td>
<td>000000</td>
<td>110</td>
</tr>
</tbody>
</table>

Format: MFDVB rt

Description:
The contents of Data Value Breakpoint register of the COP0 are loaded into general register rt.

Operation:
\[
data \leftarrow \text{CPR}[0, \text{Data Value Breakpoint}]
\]
\[
\text{GPR}[rt] \leftarrow (data_{31})^{32} || data_{31.0}
\]

Exceptions:

Coprocessor Unusable exception
### Format

MFDVBM rt

### Description

The contents of *Data Value Breakpoint Mask* register of the COP0 are loaded into general register *rt*.

### Operation

\[
\text{data} \leftarrow \text{CPR}[0, \text{Data Value Breakpoint Mask}] \\
\text{GPR}[rt] \leftarrow (\text{data}_{31})^{32} \| \text{data}_{31.0}
\]

### Exceptions

- Coprocessor Unusable exception
Move from Instruction Address Breakpoint Register

<table>
<thead>
<tr>
<th>COP0</th>
<th>MF0</th>
<th>rt</th>
<th>Debug</th>
<th>0</th>
<th>MFIAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>00000</td>
<td>11000</td>
<td>00000000</td>
<td>010</td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

MFIAB rt

**Description:**

The contents of Instruction Address Breakpoint register of the COP0 are loaded into general register rt.

**Operation:**

\[
\text{data} \leftarrow \text{CPR}[0, \text{Instruction Address Breakpoint}]
\]

\[
\text{GPR}[rt] \leftarrow (\text{data}_{31})^{32} \parallel \text{data}_{31..0}
\]

**Exceptions:**

Coprocessor Unusable exception
Format: MFIABM rt

Description:

The contents of Instruction Address Breakpoint Mask register of the COP0 are loaded into general register rt.

Operation:

\[
\text{data} \leftarrow \text{CPR}[0, \text{Instruction Address Breakpoint Mask}]
\]

\[
\text{GPR}[rt] \leftarrow (\text{data}_{31})^{32} \| \text{data}_{31.0}
\]

Exceptions:

Coprocessor Unusable exception
The contents of Performance Counter register of the COP0 are loaded into general register rt.

The reg OpCode bit indicates the number of Performance Counters. Only register 0 and 1 are valid in the C790 implementation.

**Operation:**

\[
\text{data} \leftarrow \text{CPR}[0, \text{Performance Counter (reg)}] \\
\text{GPR}[rt] \leftarrow (\text{data}_{31})^{32} \| \text{data}_{31.0}
\]

**Exceptions:**

Coprocessor Unusable exception
Appendix C  COP0 System Control Coprocessor Instruction Set Details

**MFPS**  
Move from Performance Event Specifier

<table>
<thead>
<tr>
<th>COP0</th>
<th>MF0</th>
<th>rt</th>
<th>Perf</th>
<th>0</th>
<th>reg</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>00000</td>
<td>11001</td>
<td>00000</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6 5 5 5 1

**Format:** MFPS rt, reg

**Description:**

The contents of *Performance Control* register of the COP0 are loaded into general register rt.

The reg OpCode bit indicates the number of *Performance Counter Control* registers. Only register 0 is valid in the C790 implementation.

**Operation:**

\[
\text{data} \leftarrow \text{CPR}[0, \text{Performance Control (reg)}]
\]

\[
\text{GPR}[rt] \leftarrow (\text{data}_{31})^{32} \|	ext{data}_{31:0}
\]

**Exceptions:**

Coprocessor Unusable exception
MTBPC  Move to Breakpoint Control Register  MTBPC

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>MT0</td>
<td>rt</td>
<td>Debug</td>
<td>0</td>
<td>0000 0000</td>
<td>MTBPC</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000</td>
<td>00100</td>
<td>0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:  MTBPC  rt

Description:
The contents of general register rt are loaded into Breakpoint Control register of COP0.

Operation:
\[
data \leftarrow \text{GPR}[rt]
\]
\[
\text{CPR}[0, \text{Breakpoint Control}] \leftarrow \text{data}
\]

Programming Notes:
All MTBPC instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

Exceptions:
Coprocessor Unusable exception
**MTC0**  
**Move to System Control Coprocessor**

<table>
<thead>
<tr>
<th>COP0</th>
<th>MT0</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>00100</td>
<td>000</td>
<td>00000000</td>
<td></td>
</tr>
</tbody>
</table>

**R4000**

**Format:**  
MTC0 rt, rd

**Description:**  
The contents of general register rt are loaded into coprocessor register rd of COP0.

**Operation:**

\[
data \leftarrow \text{GPR}[rt] \\
\text{CPR}[0, rd] \leftarrow data
\]

**Programming Notes:**

1. All MTC0 instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update. There is one exception to this rule:
   a) An MTC0 instruction which loads the EntryHi COP0 register can be followed by a TLBWI or a TLBWR instruction without having an intervening SYNC.P instruction. This special case is handled by a hardware interlock.

2. It is required that the MTC0 instruction to EntryHi register MUST be executed either from unmapped space or from global mapped space (mapped space with a TLB entry which has the G bit set). Furthermore, the BTAC is flushed whenever the EntryHi register is updated.

3. Modifying CONFIG.K0 via a MTC0 instruction should not occur from kseg0 space.

4. A SYNC.L instruction is needed before executing a MTC0 instruction which modifies CONFIG.NBE or CONFIG.DCE.

5. Updating the performance counter registers via a MTC0 instruction while the performance counters are enabled will result in undefined counter values.

**Exceptions:**

- Coprocessor Unusable exception
MTDAB  Move to Data Address Breakpoint Register  MTDAB

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>3 2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>MT0</td>
<td>rt</td>
<td>Debug</td>
<td>0</td>
<td>MTDAB</td>
<td>100</td>
</tr>
<tr>
<td>010000</td>
<td>00100</td>
<td>0000 0000</td>
<td>11000</td>
<td>0000 0000</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C790

Format:   MTDAB rt

Description:
The contents of general register rt are loaded into Data Address Breakpoint register of COP0.

Operation:
\[
data \leftarrow \text{GPR}[rt] \\
\text{CPR}[0, \text{Data Address Breakpoint}] \leftarrow \text{data}
\]

Programming Notes:
All MTDAB instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

Exceptions:
Coprocessor Un usable exception
MTDABM  Move to Data Address Breakpoint Mask Register  MTDABM

<table>
<thead>
<tr>
<th>COP0</th>
<th>MT0</th>
<th>rt</th>
<th>Debug</th>
<th>Data Address Breakpoint Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>00100</td>
<td>11000</td>
<td>0000000</td>
<td>101</td>
</tr>
</tbody>
</table>

**Format**

MTDABM  rt

**Description:**

The contents of general register rt are loaded into Data Address Breakpoint Mask register of COP0.

**Operation:**

\[
data \leftarrow \text{GPR}[rt] \\
\text{CPR}[0, \text{Data Address Breakpoint Mask}] \leftarrow \text{data}
\]

**Programming Notes:**

All MTDABM instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

**Exceptions:**

Coprocessor Unusable exception
MTDVB

Move to Data Value Breakpoint Register

C790

Format: MTDVB rt

Description:

The contents of general register rt are loaded into Data Value Breakpoint register of COP0.

Operation:

\[
\text{data} \leftarrow \text{GPR}[rt] \\
\text{CPR}[0, \text{Data Value Breakpoint}] \leftarrow \text{data}
\]

Programming Notes:

All MTDVB instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

Exceptions:

Coprocessor Unusable exception
### MTDVBM

**Move to Data Value Breakpoint Mask Register**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>010000</td>
<td>MT0</td>
<td>00100</td>
<td>rt</td>
<td>Debug</td>
<td>11000</td>
<td>0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** MTDVBM rt

**Description:**

The contents of general register rt are loaded into Data Value Breakpoint Mask register of COP0.

**Operation:**

\[
\text{data} \leftarrow \text{GPR}[rt] \\
\text{CPR}[0, \text{Data Value Breakpoint Mask}] \leftarrow \text{data}
\]

**Programming Notes:**

All MTDVBM instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

**Exceptions:**

Coprocessor Unusable exception
MTIAB  
Move to Instruction Address Breakpoint Register  

<table>
<thead>
<tr>
<th>COP0</th>
<th>MT0</th>
<th>rt</th>
<th>Debug</th>
<th>0</th>
<th>MTIAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>00100</td>
<td>rt</td>
<td>11000</td>
<td>0000 0000</td>
<td>010</td>
</tr>
</tbody>
</table>

Format: MTIAB rt

Description:
The contents of general register rt are loaded into Instruction Address Breakpoint register of COP0.

Operation:

\[
data \leftarrow \text{GPR}[rt] \\
\text{CPR}[0, \text{Instruction Address Breakpoint}] \leftarrow \text{data}
\]

Programming Notes:
All MTIAB instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

Exceptions:
Coprocessor Unusable exception
## MTIABM

**Move to Instruction Address Mask Breakpoint Register**

<table>
<thead>
<tr>
<th></th>
<th>COP0</th>
<th>MT0</th>
<th>rt</th>
<th>Debug</th>
<th>11000</th>
<th>00000000</th>
<th>MTIABM</th>
<th>011</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>010000</td>
<td>00100</td>
<td></td>
<td></td>
<td>11000</td>
<td>000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** MTIABM rt

**Description:**

The contents of general register rt are loaded into Instruction Address Mask Breakpoint register of COP0.

**Operation:**

\[
\text{data} \leftarrow \text{GPR}[rt] \\
\text{CPR}[0, \text{Instruction Address Mask Breakpoint}] \leftarrow \text{data}
\]

**Programming Notes:**

All MTIABM instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

**Exceptions:**

Coprocessor Unusable exception
**MTPC**  
Move to Performance Counter

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>6 5</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0 010000</td>
<td>MT0 00100</td>
<td>rt</td>
<td>Perf 11001</td>
<td>0 00000</td>
<td>reg</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

**Format:** MTPC rt, reg

**Description:**

The contents of general register rt are loaded into Performance Counter register.

The reg OpCode bit indicates the number of Performance Counters. Only register 0 and 1 are valid in the C790 implementation.

**Operation:**

data ← GPR[rt]  
CPR[0, Performance Counter (reg)] ← data

**Programming Notes:**

All MTPC instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

Updating the performance counters via a MTPC instruction while the performance counters are enabled will result in undefined counter values.

**Exceptions:**

Coprocessor unusable exception
**MTPS**

Move to Performance Event Specifier

<table>
<thead>
<tr>
<th>COP0</th>
<th>MT0</th>
<th>rt</th>
<th>Perf</th>
<th>reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>00100</td>
<td>11001</td>
<td>00000</td>
<td>0</td>
</tr>
</tbody>
</table>

| 6 | 5 | 5 | 5 | 5 | 1 |

**C790**

**Format:** MTPS rt, reg

**Description:**

The contents of general register rt are loaded into Performance Control register.

The reg OpCode bit indicates the number of Performance Control registers. Only register 0 is valid in the C790 implementation.

**Operation:**

\[
data \leftarrow \text{GPR}[rt] \\
\text{CPR}[0, \text{Performance Control (reg)}] \leftarrow \text{data}
\]

**Programming Notes:**

All MTPS instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

**Exceptions:**

Coprocessor unusable exception
**TLBP**

Probe TLB for Matching Entry

<table>
<thead>
<tr>
<th>COP0</th>
<th>C0</th>
<th>0</th>
<th>TLBP</th>
<th>TLBP</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>10000</td>
<td>0000000000000000</td>
<td>001000</td>
<td>001000</td>
</tr>
</tbody>
</table>

**Format:** TLBP

**Description:**

The Index register is loaded with the address of the TLB entry whose contents match the contents of the EntryHi register. If no TLB entry matches, the high-order bit of the Index register is set to 1. Note that the virtual address in the EntryHi register is masked with the corresponding mask field of the TLB entry prior to the comparison.

The architecture does not specify the operation of memory references associated with the instruction immediately after a TLBP instruction, nor is the operation specified if more than one TLB entry matches.

**Operation:**

```
Index ← 1 || 0^25 || undefined^6
for i in 0..TLBEntries-1
  if (TLB[i]_95..77 = (not TLB[i]_127..109) and EntryHi_31..13) and (TLB[i]_76 or (TLB[i]_71..64 = EntryHi_7..0)) then
    Index ← 0^26 || i_5..0
  endif
endfor
```

**Programming Notes:**

The TLBP instruction MUST be immediately followed by SYNC.P or ERET instruction

**Exceptions:**

Coprocessor Unusable exception
TLBR

Read Indexed TLB Entry

<table>
<thead>
<tr>
<th>COP0</th>
<th>C0</th>
<th>0</th>
<th>TLBR</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>10000</td>
<td>000 0000 0000 0000</td>
<td>000001</td>
</tr>
</tbody>
</table>

R4000

Format: TLBR

Description:

The EntryHi, EntryLo, and PageMask registers are loaded with the contents of the TLB entry pointed at by the contents of the TLB Index register.

The G bit (which controls ASID matching) read from the TLB is written into both of the EntryLo0 and EntryLo1 registers. Depending the value in PageMask register used for a TLB write instruction, the value read out from TLB may not retrieve what was originally written. See Description for TLBWI/TLBWR instruction.

Operation:

\[
\begin{align*}
\text{PageMask} & \leftarrow \text{TLB[Index5..0]}_{127..96} \\
\text{EntryHi} & \leftarrow (\text{TLB[Index5..0]}_{95..77} \parallel 0^6 \parallel \text{TLB[Index5..0]}_{71..64} ) \text{ and (not TLB[Index5..0]}_{127..96}) \\
\text{EntryLo0} & \leftarrow \text{TLB[Index5..0]}_{63..33} \parallel \text{TLB[Index5..0]}_{76} \\
\text{EntryLo1} & \leftarrow \text{TLB[Index5..0]}_{31..1} \parallel \text{TLB[Index5..0]}_{76}
\end{align*}
\]

Programming Notes:

The TLBR instruction MUST be executed from either unmapped space or global mapped space (mapped space with a TLB entry which has the G bit set).

The TLBR instruction MUST be immediately followed by SYNC.P or ERET instruction.

Exceptions:

Coprocessor Unusable exception
Appendix C  COP0 System Control Coprocessor Instruction Set Details

TLBWI  Write Index TLB Entry

| COP0 | C0  | 0 | 0000000000000000 | TLBWI |
| 010000 | 10000 | 0 | 000010 | 000010 |

R4000

Format:  TLBWI

Description:

The TLB entry pointed at by the contents of the TLB Index register is loaded with the contents of the PageMask, EntryHi, EntryLo0 and EntryLo1 registers.

The G bit of the TLB is written with the logical AND of the G bits in the EntryLo0 and EntryLo1 registers. The virtual address in the EntryHi register is modified by the Mask field of the PageMask register before being written into the TLB.

The operation is invalid (and the results are unspecified) if contents of the TLB Index register are greater than the number of TLB entries in the processor.

In the C790 processor, a TLB write instruction is used to write the whole page frame number from the EntryLo registers to the TLB entry. Depending on the page size specified in the corresponding PageMask register, the lower bits of PFN may not be used for address translation and lower bits of VPN2 in EntryHi register which is masked by the content of PageMask register are forced to zeros during a TLB write. This does not affect TLB address translation, however, a TLB read may not retrieve what was originally written.

Operation:

\[
\text{TLB}[\text{Index5..0}] \leftarrow \text{PageMask} \| (\text{EntryHi31..13} \| (\text{EntryLo0} \text{ and EntryLo1}) \| \text{EntryHi31..0} \text{ and } \text{not PageMask})) \| \text{EntryLo031..1} \| 0 \| \text{EntryLo131..1} \| 0
\]

Programming Notes:

The TLBWI instruction MUST be executed from either unmapped space or global mapped space (mapped space with a TLB entry which has the G bit set).

The TLBWI instruction MUST be followed by a ERET or a SYNC.P instruction to insure TLB update.

Exceptions:

Coprocessor Unusable exception
### TLBWR

**Format:** TLBWR

**Description:**

The TLB entry pointed at by the contents of the TLB Random register is loaded with the contents of the PageMask, EntryHi, EntryLo0 and EntryLo1 registers.

The G bit of the TLB is written with the logical AND of the G bits in the EntryLo0 and EntryLo1 registers. The virtual address in the EntryHi register is modified by the Mask field of the PageMask register before being written into the TLB.

In the C790 processor, a TLB write instruction is used to write the whole page frame number from the EntryLo registers to the TLB entry. Depending on the page size specified in the corresponding PageMask register, the lower bits of PFN may not be used for address translation and lower bits of VPN2 in EntryHi register which is masked by the content of PageMask register are forced to zeros during a TLB write. This does not affect TLB address translation, however, a TLB read may not retrieve what was originally written.

**Operation:**

\[
\text{TLB}[\text{Random5..0}] \leftarrow \text{PageMask} \middle| (\text{EntryHi31..13} \middle| (\text{EntryLo0} \text{ and } \text{EntryLo1}) \middle| \text{EntryHi11..0}) \text{ and } (\text{not PageMask}) \middle| \text{EntryLo031..1} \middle| 0 \middle| \text{EntryLo131..1} \middle| 0
\]

**Programming Notes:**

The TLBWR instruction MUST be executed from either unmapped space or global mapped space (mapped space with a TLB entry which has the G bit set).

The TLBWR instruction MUST be followed by a ERET or a SYNC.P instruction to insure TLB update.

**Exceptions:**

Coprocessor Unusable exception
## C.2 COP0 Instruction Encoding

<table>
<thead>
<tr>
<th>OpCode</th>
<th>bits 28..26</th>
<th>Instructions encoded by OpCode field (COP0, CACHE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31..29</td>
<td>000</td>
<td>SPECIAL REGIMM J JAL BEQ BNE BLEZ BGTZ</td>
</tr>
<tr>
<td></td>
<td>001</td>
<td>ADDI ADDIU SLTI SLTIU ANDI ORI XORI LUI</td>
</tr>
<tr>
<td></td>
<td>010</td>
<td>COP0 δ COP1 * * BEQL BNEL BLEZL BGTZL</td>
</tr>
<tr>
<td></td>
<td>011</td>
<td>DADDI DADDIU LDL LDR MMI * LO SQ</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>LB LH LWL LW LBU LHU LWR LWU</td>
</tr>
<tr>
<td></td>
<td>101</td>
<td>SB SH SWL SW SDL SDR SWR CACHE</td>
</tr>
<tr>
<td>31..26</td>
<td>110</td>
<td>η LWC1 η PREF η LDC1 η LD</td>
</tr>
<tr>
<td></td>
<td>111</td>
<td>η SWC1 η * η SDC1 η SD</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OpCode = COP0</th>
<th>rs</th>
<th>bits 23..21</th>
<th>Instructions encoded by rs field when OpCode field = COP0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31..26</td>
<td>21</td>
<td>000 001 010 011 100 101 110 111</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MF0 * * * MT0 * * *</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BC0 δ * * * * * *</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C0 δ * * * * * *</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OpCode = COP0</th>
<th>rs = MF0 or MT0</th>
<th>rd = Debug*</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>31..26</td>
<td>21 12 16 11 10 3 2 0</td>
<td>MF0 MFPC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MTPC MTBPC</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OpCode = COP0</th>
<th>rs = MF0 or MT0</th>
<th>rd = Perf*</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>31..26</td>
<td>21 12 16 11 10 1 0</td>
<td>MF0 MFPC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MTPC MTBPC</td>
<td></td>
</tr>
</tbody>
</table>

* Debug and Perf are the CP0 register names.
Debug = 11000 (24), Perf = 11001 (25)
### Appendix C COP0 System Control Coprocessor Instruction Set Details

#### 31 26 25 21 20 16 0

<table>
<thead>
<tr>
<th>OpCode = COP0</th>
<th>rs = BC0</th>
<th>rt</th>
</tr>
</thead>
</table>

#### rt bits 18..16

Instructions encoded by rt field when OpCode field = COP0 & rs field = BC0

<table>
<thead>
<tr>
<th>bits</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>BC0F</td>
<td>BC0T</td>
<td>BC0FL</td>
<td>BC0TL</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>01</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>10</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>11</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

#### function bits 2..0

Instructions encoded by function field when OpCode field = COP0 & rs field = C0

<table>
<thead>
<tr>
<th>bits</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>TLBR</td>
<td>TLBWI</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>TLBWR</td>
<td>φ</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>TLBP</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>ERET</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>EI</td>
<td>DI</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td></td>
</tr>
</tbody>
</table>

* This OpCode is reserved for future use. An attempt to execute it causes a Reserved Instruction exception.

φ This OpCode is reserved for future use. An attempt to execute it produces an undefined result. The result may be a Reserved Instruction exception but this is not guaranteed.

δ This OpCode indicates an instruction class. The instruction word must be further decoded by examining additional tables that show the values for another instruction field.

η This OpCode is reserved for one of the following instructions which are currently not supported: DMULT, DMULTU, DDIV, DDIVU, LL, LLD, SC, SCD, LWC2, SWC2. An attempt to execute it causes a Reserved Instruction exception.
D. COP1 (FPU) Instruction Set Details

This appendix provides a detailed description of each of the COP1 coprocessor instructions. COP1 is implemented as a floating point unit (FPU).

The instruction descriptions provide:

• a bit by bit field definition of the instruction word signifying that instruction
• a verbal description of the operation performed by the instruction
• pseudo-code identifying the entire sphere of influence of the instruction in terms of operand dependency and the state (s) of the processor changed.

Omission of any/all states is taken to mean that the same have not changed by the act of execution of the instruction under description.
D.1 Conventions Used in This Chapter

D.1.1 Instruction Description Notation and Functions

The Operation sections of the instruction descriptions use a high-level language notation, or pseudocode, to describe the instruction’s operations. Symbols, functions, and structures used in the Operation sections are described here.

The notation FPR as used here refers to the 32 floating-point registers FPR0 through FPR31 of the FPU.

D.1.2 Pseudocode Language Statement Execution

Each of the high-level language statements in an operation description is executed in sequential order (as modified by conditional and loop constructs).

D.1.3 Pseudocode Symbols

Special symbols used in the notation are described in Appendix A.

D.2 Definitions for Pseudocode Functions Used in Operation Descriptions

A variety of functions are used in the pseudocode descriptions to make the pseudocode more readable and also to abstract implementation-specific behavior. These functions are defined in Appendix A; in addition, certain COP1 FPU-specific functions are described in the following section. The following pseudocode notation is used in functions in the descriptions of floating-point operations:

<table>
<thead>
<tr>
<th>Pseudocode Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>StoreFPR (fpr, value)</td>
<td>FPR[fpr] ← value</td>
</tr>
<tr>
<td>ConvertFmt (value, fmt1, fmt2)</td>
<td>The value in the format fmt1 is converted to a value in the format fmt2.</td>
</tr>
<tr>
<td>Negate (value)</td>
<td>The value is negated by changing the sign bit value.</td>
</tr>
<tr>
<td>Sign-extend (Value)</td>
<td>A sign-extended 32-bit value has bits 63..31 of equal value</td>
</tr>
</tbody>
</table>
D.3 Instruction Descriptions

Descriptions of FPU Instructions follow.
ABS.fmt

Floating Point Absolute Value

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>fmt</td>
<td>0</td>
<td>00000</td>
<td>fs</td>
<td>fd</td>
<td>ABS</td>
<td>000101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format: ABS.S fd, fs
ABS.D fd, fs

Purpose: To compute the absolute value of an FP value.

Description: fd ← absolute (fs)

The absolute value of the value in FPR fs is placed in FPR fd. The operand and result are values in format fmt.

This operation is arithmetic; a NaN operand signals invalid operation.

Restrictions:

The field fs and fd must specify FPRs valid for operands of type fmt; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

\[ \text{StoreFPR (fd, fmt, AbsoluteValue (ValueFPR (fs, fmt)))} \]

Exceptions:

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Unimplemented Operation
  - Invalid Operation
### ADD.fmt

**Floating Point Add**

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>ADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000000</td>
</tr>
</tbody>
</table>

#### Format:

ADD.S  fd, fs, ft  
ADD.D  fd, fs, ft

#### Purpose:
To add FP values.

#### Description:

fd ← fs + ft

The value in FPR ft is added to the value in FPR fs. The result is calculated to infinite precision, rounded according to the current rounding mode in FCR31, and placed into FPR fd. The operands and result are values in format fmt.

#### Restrictions:

The field fs, ft and fd must specify FPRs valid for operands of type fmt; see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

#### Operation:

StoreFPR (fd, fmt, ValueFPR (fs, fmt) + ValueFPR (ft, fmt))

#### Exceptions:

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Unimplemented Operation
  - Invalid Operation
  - Inexact
  - Overflow
  - Underflow
BC1F  
Branch on FP False

Format:  BC1F  offset

Purpose:  To test an FP condition code and do a PC-relative conditional branch.

Description:  if (C = 0) then branch where C is FCR3123

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the result of the last floating point compare is false, branch to the effective target address after the instruction in the delay slot is executed.

An FP condition code is set by the FP compare instruction, C.cond.fmt.

Operation:

I:  condition ← (FCR3123 = 0)
    target_offset ← (offset15)↑GPRLEN-(16+2) || offset || 0²
I+1: if condition then
    PC ← PC + target
    endif

Exceptions:

Coprocessor Unusable
Reserved Instruction

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.
BC1T  
Branch on FP True  
BC1T

<table>
<thead>
<tr>
<th>COP1</th>
<th>BC1</th>
<th>BC1T</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>01000</td>
<td>00001</td>
<td></td>
</tr>
</tbody>
</table>

6  5  5  16  0

MIPS I

Format:  BC1T  offset
Purpose:  To test an FP condition code and do a PC-relative conditional branch.
Description:  if (C = 1) then branch where C is FCR3123.
An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of
the instruction following the branch (not the branch itself), in the branch delay slot, to
form a PC-relative effective target address.
If the result of the last floating point compare is true, branch to the effective target
address after the instruction in the delay slot is executed.
An FP condition code is set by the FP compare instruction, C.cond.fmt.
Operation:
I:   condition ← (FCR3123 = 1)
target ← (offset16)GPRLEN-(16+2) || offset || 02
I+1: if condition then
    PC ← PC + target
endif
Exceptions:
Coprocessor Unusable
Reserved Instruction
Programming Notes:
With the 18-bit signed instruction offset, the conditional branch range is ± 128KB. Use
jump (J) or jump register (JR) instructions to branch to more distant addresses.
C.cond.fmt  Floating Point Compare  C.cond.fmt

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>ft</th>
<th>fs</th>
<th>0</th>
<th>FC</th>
<th>cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td></td>
<td></td>
<td></td>
<td>00000</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format:  
C.cond.S  fs, ft  
C.cond.D  fs, ft

Purpose:  
To compare FP values and record the Boolean result in a condition code.

Description:  
C ← fs  compare_cond  ft

The value in FPR fs is compared to the value in FPR ft; the values are in format fmt. The comparison is exact and neither overflows nor underflows. If the comparison specified by cond 2..1 is true for the operand values, then the result is true, otherwise it is false. If no exception is taken, the result is written into condition code C; true is 1 and false is 0.

If cond3 is set and at least one of the values is a NaN, an Invalid Operation condition is raised; the result depends on the FP exception model currently active.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written and an Invalid Operation exception is taken immediately. Otherwise, the Boolean result is written into condition code C.

There are four mutually exclusive ordering relations for comparing floating-point values; one relation is always true and the others are false. The familiar relations are greater than, less than, and equal. In addition, the IEEE floating-point standard defines the relation unordered which is true when at least one operand value is NaN; NaN compares unordered with everything, including itself. Comparisons ignore the sign of zero, so +0 equals -0.

The comparison condition is a logical predicate, or equation, of the ordering relations such as “less than or equal”, “equal”, “not less than”, or “unordered or equal”. Compare distinguishes sixteen comparison predicates. The Boolean result of the instruction is obtained by substituting the Boolean value of each ordering relation for the two FP values into equation. If the equal relation is true, for example, then all four example predicates above would yield a true result. If the unordered relation is true then only the final predicate, “unordered or equal” would yield a true result.

Logical negation of a compare result allows eight distinct comparisons to test for sixteen predicates as shown in Table D-1. Each mnemonic tests for both a predicate and its logical negation. For each mnemonic, compare tests the truth of the first predicate. When the first predicate is true, the result is true as shown in the “if predicate is true” column (note that the False predicate is never true and False/True do not follow the normal pattern). When the first predicate is true, the second predicate must be false, and vice versa. The truth of the second predicate is the logical negation of the instruction result. After a compare instruction, test for the truth of the first predicate with the Branch on FP True (BC1T) instruction and the truth of the second with Branch on FP False (BC1F).
Table D-1. FPU Comparisons Without Special Operand Exceptions

<table>
<thead>
<tr>
<th>Instr</th>
<th>Comparison Predicate</th>
<th>relation values</th>
<th>If predicate is true</th>
<th>Inv Op excep if Q NaN</th>
<th>cond field</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond Mnemonic</td>
<td>name of predicate and logically negated predicate (abbreviation)</td>
<td>&gt;</td>
<td>&lt;</td>
<td>=</td>
<td>?</td>
</tr>
<tr>
<td>F</td>
<td>False [this predicate is always False, it never has a True result]</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>UN</td>
<td>Unordered Ordered (OR)</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>EQ</td>
<td>Equal Not Equal (NEQ)</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>UEQ</td>
<td>Unordered or Equal Ordered or Greater than or Less than (OGL)</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>OLT</td>
<td>Ordered or Less Than Unordered or Greater than or Equal (UGE)</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>ULT</td>
<td>Unordered or Less Than Ordered or Greater than or Equal (OGE)</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>OLE</td>
<td>Ordered or Less than or Equal Unordered or Greater Than (UGT)</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>ULE</td>
<td>Unordered or Less than or Equal Ordered or Greater Than (OGT)</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

key: “?” = unordered, “>” = greater than, “<” = less than, “=” is equal, “T” = True, “F” = False
There is another set of eight compare operations, distinguished by a \texttt{cond3} value of 1, testing the same sixteen conditions. For these additional comparisons, if at least one of the operands is a NaN, including Quiet NaN, then an Invalid Operation condition is raised. If the Invalid Operation condition is enabled in the FCR31, then an Invalid Operation exception occurs.

### Table D-2 FPU Comparisons With Special Operand Exceptions for QNaNs

<table>
<thead>
<tr>
<th>Instr</th>
<th>Comparison Predicate</th>
<th>Comparison CC Result</th>
<th>Instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mne-</td>
<td>name of predicate and logically negated</td>
<td></td>
<td></td>
</tr>
<tr>
<td>monic</td>
<td>predicate (abbreviation)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SF</td>
<td>Signaling False</td>
<td>F F F F</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Signaling True (ST)</td>
<td>T T T T</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[this predicate always False]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NGLE</td>
<td>Not Greater than or Less than or Equal</td>
<td>F F F F</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Greater than or Less than or Equal (GLE)</td>
<td>T T T T</td>
<td></td>
</tr>
<tr>
<td>SEQ</td>
<td>Signaling Equal</td>
<td>F F T F</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Signaling Not Equal (SNE)</td>
<td>T T F T</td>
<td></td>
</tr>
<tr>
<td>NGL</td>
<td>Not Greater than or Less than</td>
<td>F F T T</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Greater than or Less than (GL)</td>
<td>T T F F</td>
<td></td>
</tr>
<tr>
<td>LT</td>
<td>Less Than</td>
<td>F T F T</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Not Less Than (NLT)</td>
<td>T F F T</td>
<td></td>
</tr>
<tr>
<td>NGE</td>
<td>Not Greater than or Equal</td>
<td>F T F T</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Greater than or Equal (GE)</td>
<td>T F F F</td>
<td></td>
</tr>
<tr>
<td>LE</td>
<td>Less than or Equal</td>
<td>F T T F</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Not Less than or Equal (NLE)</td>
<td>T F T T</td>
<td></td>
</tr>
<tr>
<td>NGT</td>
<td>Not Greater Than</td>
<td>F T T T</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Greater Than (GT)</td>
<td>T F F F</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>key: “?” = unordered, “&gt;” = greater than, “&lt;” = less than, “=” is equal, “T” = True, “F” = False</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Restrictions:

The field \(fs\) and \(ft\) must specify FPRs valid for operands of type \(fmt\); see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

### Operation:

if NaN (Value FPR (fs, fmt)) or NaN (ValueFPR (ft, fmt)) then
  less ← false
  equal ← false
  unordered ← true
  if t then
    SignalException (InvalidOperation)
  endif
else
  less ← ValueFPR (fs, fmt) < ValueFPR (ft, fmt)
  equal ← ValueFPR (fs, fmt) = ValueFPR (ft, fmt)
  unordered ← false
endif
condition ← (cond2 and less) or (cond1 and equal) or (cond0 and unordered)
C ← condition
Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
    Unimplemented Operation
    Invalid Operation

Programming Notes:

FP computational instructions, including compare, that receive an operand value of Signaling NaN, will raise the Invalid Operation condition. The comparisons that raise the Invalid Operation condition for Quiet NaNs in addition to SNaNs, permit a simpler programming model if NaNs are errors. Using these compares, programs do not need explicit code to check for QNaNs causing the unordered relation. Instead, they take an exception and allow the exception handling system to deal with the error when it occurs.

For example, consider a comparison in which we want to know if two numbers are equal, but for which unordered would be an error.

```assembly
# comparisons using explicit tests for QNaN
  c.eq.d $f2,$f4 # check for equal
  nop
  bc1t L2 # it is equal
  c.un.d $f2,$f4 # it is not equal, but might be unordered
  bc1t ERROR# unordered goes off to an error handler
# not-equal-case code here
...
# equal-case code here
L2:

# --------------------------------------
# comparison using comparisons that signal QNaN
  c.seq.d $f2,$f4 # check for equal
  nop
  bc1t L2 # it is equal
  nop
  nop
# it is not unordered here...
# not-equal-case code here
...
#equal-case code here
L2:
```
### CEIL.L.fmt  Floating-Point Ceiling Convert to Long Fixed-Point

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>CEIL.L</th>
<th>001010</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**MIPS III**

**Format:**
- CEIL.L.S fd, fs
- CEIL.L.D fd, fs

**Purpose:** To convert an FP value to 64-bit fixed-point, rounding up.

**Description:**
\[ fd \leftarrow \text{convert\_and\_round}\ (fs) \]

The value in FPR \(fs\) in format \(fmt\), is converted to a value in 64-bit long fixed-point format rounding toward \(+\infty\) (rounding mode 2). The result is placed in FPR \(fd\).

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{63}\) to \(2^{63} - 1\), the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to \(fd\) and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{63} – 1\), is written to \(fd\).

**Restrictions:**

The fields \(fs\) and \(fd\) must specify valid FPRs; \(fs\) for type \(fmt\) and \(fd\) for long fixed-point; see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

**Operation:**

\[ \text{StoreFPR}\ (fd, L, \text{ConvertFmt}\ (\text{ValueFPR}\ (fs, fmt), fmt, L)) \]

**Exceptions:**

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Invalid Operation
  - Unimplemented Operation
  - Inexact
  - Overflow
CEIL.W.fmt Floating-Point Ceiling Convert to Word Fixed-Point

### Format:

- CEIL.W.S fd, fs
- CEIL.W.D fd, fs

### Purpose:
To convert an FP value to 32-bit fixed-point, rounding up.

### Description:

\[ fd \leftarrow \text{convert\_and\_round} (fs) \]

The value in FPR \( fs \) in format \( fmt \), is converted to a value in 32-bit word fixed-point format rounding toward \(+\infty\) (rounding mode 2). The result is placed in FPR \( fd \).

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{31}\) to \(2^{31}-1\), the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to \( fd \) and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{31}-1\), is written to \( fd \).

### Restrictions:

The fields \( fs \) and \( fd \) must specify valid FPRs; \( fs \) for type \( fmt \) and \( fd \) for word fixed-point; see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

### Operation:

\[ \text{StoreFPR} (fd, W, \text{ConvertFmt} (\text{ValueFPR} (fs, fmt), fmt, W)) \]

### Exceptions:

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Invalid Operation
  - Unimplemented Operation
  - Inexact
  - Overflow
Appendix D  COP1 (FPU) Instruction Set Details

CFC1  Move Control Word from Floating Point  CFC1

<table>
<thead>
<tr>
<th>COP1</th>
<th>CFC1</th>
<th>rt</th>
<th>fs</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00010</td>
<td>6</td>
<td>5</td>
<td>11</td>
</tr>
</tbody>
</table>

MIPS I

Format:  CFC1  rt, fs
Purpose:  To copy a word from an FPU control register to a GPR.
Description:  rt ← FP_Control[fs]

Copy the 32-bit word from FP (coprocessor 1) control register fs into GPR rt, sign-extending it if the GPR is 64 bits.

Restrictions:

There are only a couple control registers defined for the floating point unit. The result is not defined if fs specifies a register that does not exist.

Operation:

GPR[rt] ← sign_extend (FCR[fs])

Exceptions:

Coprocessor Unusable
CTC1  Move Control Word to Floating Point

Format: CTC1 rt, fs

Purpose: To copy a word from a GPR to an FPU control register.

Description: FP_Control[fs] ← rt

Copy the low word from GPR rt into FP (coprocessor 1) control register fs.

Writing to control register 31, the Floating-Point Control and Status Register or FCR31, causes the appropriate exception if any cause bit and its corresponding enable bit are both set. The register will be written before the exception occurs.

Restrictions:

There are only a couple control registers defined for the floating point unit. The result is not defined if fs specifies a register that does not exist.

Operation:

\[
\begin{align*}
temp & \leftarrow \text{GPR}[rt]_{31:0} \\
\text{FCR}[fs] & \leftarrow temp
\end{align*}
\]

Exceptions:

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Invalid Operation
  - Unimplemented Operation
  - Inexact
  - Overflow
  - Underflow
  - Division by Zero
CVT.D.fmt  Floating-Point Convert to Double Floating Point  CVT.D.fmt

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>CVT.D</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td>100001</td>
</tr>
</tbody>
</table>

MIPS I, III

Format:
- CVT.D.S fd, fs
- CVT.D.W fd, fs
- CVT.D.L fd, fs

Purpose: To convert an FP or fixed-point value to double FP.

Description: fd ← convert_and_round (fs)

The value in FPR fs in format fmt is converted to a value in double floating-point format rounded according to the current rounding mode in FCR31. The result is placed in FPR fd.

If fmt is S or W, then the operation is always exact.

Restrictions:

The field fs and fd must specify valid FPRs; fs for type fmt and fd for double floating point; see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, D, ConvertFmt (ValueFPR (fs, fmt), fmt, D))

Exceptions:

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Invalid Operation
  - Unimplemented Operation
  - Inexact

Note:

Overflow and Underflow exceptions never occur because double precision data format can represent any value in other data types.
**CVT.L.fmt**  
Floating-Point Convert to Long Fixed-Point

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>00000</th>
<th>fs</th>
<th>fd</th>
<th>CVT.L.fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>100101</td>
</tr>
</tbody>
</table>

**MIPS III**

**Format:**
- CVT.L.S fd, fs
- CVT.L.D fd, fs

**Purpose:** To convert an FP value to a 64-bit fixed-point.

**Description:** $fd \leftarrow \text{convert\_and\_round}(fs)$

Convert the value in format fmt in FPR fs to long fixed-point format, round according to the current rounding mode in FCR31, and place the result in FPR fd.

When the source value is Infinity, NaN, or rounds to an integer outside the range $-2^{63}$ to $2^{63} - 1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to $fd$ and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63} - 1$, is written to $fd$.

**Restrictions:**

The field $fs$ and $fd$ must specify valid FPRs; $fs$ for type fmt and $fd$ for long floating point; see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

**Operation:**

$\text{StoreFPR}(fd, L, \text{ConvertFmt}(\text{ValueFPR}(fs, fmt), fmt, L))$

**Exceptions:**

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Invalid Operation
  - Unimplemented Operation
  - Inexact
  - Overflow
Appendix D  COP1 (FPU) Instruction Set Details

CVT.S.fmt  Floating-Point Convert to Single Floating-Point  CVT.S.fmt

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>CVT.S</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td>100000</td>
</tr>
</tbody>
</table>

MIPS I, III

Format:  CVT.S.D fd, fs  
CVT.S.W fd, fs  
CVT.S.L fd, fs

Purpose:  To convert an FP or fixed-point value to single FP.

Description:  fd ← convert_and_round (fs)

The value in FPR fs in format fmt is converted to a value in single floating-point format rounded according to the current rounding mode in FCR31. The result is placed in FPR fd.

Restrictions:

The field fs and fd must specify valid FPRs; fs for type fmt and fd for single floating point; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, S, ConvertFmt (ValueFPR (fs, fmt), fmt, S))

Exceptions:

Coprocessor Unusable  
Reserved Instruction  
Floating-Point  
   Invalid Operation  
   Unimplemented Operation  
   Inexact  
   Overflow  
   Underflow

D-18
**CVT.W.fmt** Floating-Point Convert to Word Fixed-Point

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>fs</th>
<th>fd</th>
<th>CVT.W.fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:**

- CVT.W.S fd, fs
- CVT.W.D fd, fs

**Purpose:**

To convert an FP value to a 32-bit fixed-point.

**Description:**

\[fd \leftarrow \text{convert\_and\_round}(fs)\]

The value in FPR \(fs\) in format \(fmt\) is converted to a value in 32-bit word fixed-point format rounded according to the current rounding mode in FCR31. The result is placed in FPR \(fd\).

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{31}\) to \(2^{31} - 1\), the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to \(fd\) and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{31} - 1\), is written to \(fd\).

**Restrictions:**

The field \(fs\) and \(fd\) must specify valid FPRs; \(fs\) for type \(fmt\) and \(fd\) for word fixed point; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

**Operation:**

\[\text{StoreFPR}(fd, W, \text{ConvertFmt}(\text{ValueFPR}(fs, fmt), fmt, W))\]

**Exceptions:**

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Invalid Operation
  - Unimplemented Operation
  - Inexact
  - Overflow
DIV.fmt Floating Point Divide DIV.fmt

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>DIV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100001</td>
<td>000011ft</td>
<td>fs</td>
<td>fd</td>
<td>000011</td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format:  
DIV.S  fd, fs, ft  
DIV.D  fd, fs, ft

Purpose:  
To divide FP values.

Description:  
fd ← fs / ft

The value in FPR fs is divided by the value in FPR ft. The result is calculated to infinite precision, rounded according to the current rounding mode in FCR31, and placed into FPR fd. The operands and result are values in format fmt.

Restrictions:

The field fs, ft and fd must specify FPRs valid for operands of type fmt; see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, fmt, ValueFPR (fs, fmt) / ValueFPR (ft, fmt))

Exceptions:

Coprocessor Unusable  
Reserved Instruction  
Floating-Point  
Inexact  
Unimplemented Operation  
Division-by-zero  
Invalid Operation  
Overflow  
Underflow
Doubleword Move From Floating-Point

Format: \text{DMFC1} \ rt, \ fs

Purpose: To copy a doubleword from an FPR to a GPR.


description: \rt \leftarrow \fs

The doubleword contents of FPR \( \fs \) are placed into GPR \( \rt \).

If the coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR \( \fs \) is held in an even/odd register pair. The low word is taken from the even register \( \fs \) and the high word is from \( \fs+1 \).

Restrictions:

If \( \fs \) does not specify an FPR that can contain a doubleword, the result is undefined; see Floating Point Registers on page 10-2.

Operation:

\begin{verbatim}
if SizeFGR() = 64 then /* 64-bit wide FGRs */
data \leftarrow \text{FGR}[\fs]
elseif fs0 = 0 then /* valid specifier, 32-bit wide FGRs */
data \leftarrow \text{FGR}[\fs+1] \| \text{FGR}[\fs]
else /* undefined for odd 32-bit FGRs */
UnDefinedResult()
endif
GPR[rt] \leftarrow data
\end{verbatim}

Exceptions:

Reserved Instruction
Coprocessor Unusable
DMTC1

Doubleword Move To Floating-Point

<table>
<thead>
<tr>
<th>COP1</th>
<th>DMTC1</th>
<th>rt</th>
<th>fs</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00101</td>
<td></td>
<td></td>
<td>00000000</td>
</tr>
</tbody>
</table>

MIPS III

Format: DMTC1 rt, fs

Purpose: To copy a doubleword from a GPR to an FPR.

Description: fs ← rt

The doubleword contents of GPR rt are placed into FPR fs.

If the coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR fs is held in an even/odd register pair. The low word is placed in the even register fs and the high word is placed in fs+1.

Restrictions:

If fs does not specify an FPR that can contain a doubleword, the result is undefined; see Floating Point Registers on page 10-2.

Operation:

```plaintext
data ← GPR[rt]
if SizeFGR() = 64 then /* 64-bit wide FGRs */
    FGR[fs] ← data
elseif fs0 = 0 then /* valid specifier, 32-bit wide FGRs */
    FGR[fs+1] ← data63..32
    FGR[fs] ← data31..0
else /* undefined result for odd 32-bit FGRs */
    UndefinedResult()
endif
```

Exceptions:

Reserved Instruction
Coprocessor Unusable
FLOOR.L.fmt
Floating-Point Floor Convert to Long
Fixed-Point

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>FLOOR.L.fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td>001011</td>
</tr>
</tbody>
</table>

MIPS III

Format:
FLOOR.L.S fd, fs
FLOOR.L.D fd, fs

Purpose:
To convert an FP value to a 64-bit fixed-point, rounding down.

Description:
fd ← convert_and_round (fs)

The value in FPR fs in format fmt, is converted to a value in 64-bit long fixed-point format rounding toward $-\infty$ (rounding mode 3). The result is placed in FPR fd.

When the source value is Infinity, NaN, or rounds to an integer outside the range $-2^{63}$ to $2^{63} - 1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to fd and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63} - 1$, is written to fd.

Restrictions:
The field fs and fd must specify valid FPRs; fs for type fmt and fd for long fixed point; see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

Operation:
StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt, L))

Exceptions:
Coprocessor Unusable
Reserved Instruction
Floating-Point
  Invalid Operation
  Unimplemented Operation
  Inexact
  Overflow

148x708

FLOOR.W.fmt Floating-Point Floor Convert to Word

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>fs</th>
<th>fd</th>
<th>FLOOR.W.fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>00000</td>
<td>001111</td>
<td>001111</td>
</tr>
</tbody>
</table>

MIPS II

Format: FLOOR.W.S fd, fs
        FLOOR.W.D fd, fs

Purpose: To convert an FP value to a 32-bit fixed-point, rounding down.

Description: \( fd \leftarrow \text{convert\_and\_round} \ (fs) \)

The value in FPR \( fs \) in format \( fmt \), is converted to a value in 32-bit word fixed-point format rounding toward \(-\infty\) (rounding mode 3). The result is placed in FPR \( fd \).

When the source value is Infinity, NaN, or rounds to an integer outside the range \( -2^{31} \) to \( 2^{31} -1 \), the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to \( fd \) and an Invalid Operation exception is taken immediately. Otherwise, the default result, \( 2^{31} -1 \), is written to \( fd \).

Restrictions: The field \( fs \) and \( fd \) must specify valid FPRs; \( fs \) for type \( fmt \) and \( fd \) for word fixed point; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation: StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W))

Exceptions:
- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Invalid Operation
  - Unimplemented Operation
  - Inexact
  - Overflow
### LDC1

**Load Doubleword to Floating-Point**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDC1</td>
<td>base</td>
<td>ft</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110101</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS II**

**Format:**

LDC1 ft, offset (base)

**Purpose:**

To load a doubleword from memory to an FPR.

**Description:**

\[ ft \leftarrow \text{memory}\left[\text{base}+\text{offset}\right] \]

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in FPR ft. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

If coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR ft is held in an even/odd register pair. The low word is placed in the even register ft and the high word is placed in ft+1.

**Restrictions:**

If ft does not specify an FPR that can contain a doubleword, the result is undefined; see Floating-Point Resisters on page 10-2.

An Address Error exception occurs if EffectiveAddress2..0 \(\neq 0\) (not doubleword-aligned).

**Operation:**

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend (offset)} + \text{GPR[base]} \\
\text{if vAddr2..0} & \neq 0^3 \text{then SignalException (AddressError)} \text{endif} \\
(\text{pAddr, uncached}) & \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \\
\text{data} & \leftarrow \text{LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)} \\
\text{if SizeFGR()} & = 64 \text{ then} \quad /* \text{64-bit wide FGRs} */ \\
\text{FGR[ft]} & \leftarrow \text{data} \\
\text{elseif ft0} & = 0 \text{ then} \quad /* \text{valid specifier, 32-bit wide FGRs} */ \\
\text{FGR[ft+1]} & \leftarrow \text{data31..32} \\
\text{FGR[ft]} & \leftarrow \text{data63..32} \\
\text{else} & \quad /* \text{undefined result for odd 32-bit FGRs} */ \\
\text{UndefinedResult()} & \\
\text{endif}
\end{align*}
\]

**Exceptions:**

- Coprocessor Unusable
- TLB Refill
- TLB Invalid
- Address Error
Appendix D  COP1 (FPU) Instruction Set Details

LWC1  Load Word to Floating Point

<table>
<thead>
<tr>
<th>ft</th>
<th>base</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>110001</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

MIPS I

Format:  LWC1  ft, offset (base)

Purpose:  To load a word from memory to an FPR.

Description:  ft ← memory[base+offset]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed into the low word of coprocessor 1 general register ft. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

If coprocessor 1 general registers are 64-bits wide, bits 63..32 of register ft become undefined. See Floating Point Register on page 10-2.

Restrictions:

An Address Error exception occurs if EffectiveAddress1..0 ≠ 0 (not word-aligned).

Operation: 32-bit Processors

1. /* "mem" is aligned 64-bits from memory. Pick out correct bytes. */
   vAddr ← sign_extend (offset) + GPR[base]
   if vAddr1..0 ≠ 0 then SignalException (AddressError) endif
   (pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
   mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
   i + 1: FGR[ft] ← mem

Operation: 64-bit Processors

/* "mem" is aligned 64-bits from memory. Pick out correct bytes. */
   vAddr ← sign_extend (offset) + GPR[base]
   if vAddr1..0 ≠ 0 then SignalException (AddressError) endif
   (pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
   pAddr ← pAddr PSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 02))
   mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
   bytesel ← vAddr2..0 xor (BigEndianCPU || 02)
   if SizeFGR() = 64 then /* 64-bit wide FGRs */
      FGR[ft] ← undefined 32 || mem31+8*bytesel..8*bytesel
   else /* 32-bit wide FGRs */
      FGR[ft] ← mem31+8*bytesel..8*bytesel
   endif

Exceptions:

Coprocessor unusable
TLB Refill
TLB Invalid
Address Error
**MFC1**

**Move Word from Floating Point**

<table>
<thead>
<tr>
<th>COP1</th>
<th>MFC1</th>
<th>rt</th>
<th>fs</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>0000000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:** MFC1 rt, fs

**Purpose:** To copy a word from an FPU (COP1) general register to a GPR.

**Description:**

rt ← fs

The low word from FPR fs is placed into the low word of GPR rt. If GPR rt is 64 bits wide, then the value is sign extended. See Floating Point Resisters on page 10-2.

**Restrictions:**

None

**Operation:**

GPR[rt] ← sign_extend (FPR[fs][31:0])

**Exceptions:**

Coprocessor Unusable
MOV.fmt
Floating Point Move

<table>
<thead>
<tr>
<th></th>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>MOV</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>010001</td>
<td></td>
<td>00000</td>
<td></td>
<td></td>
<td>000110</td>
</tr>
<tr>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

MIPS I

Format:
- MOV.S fd, fs
- MOV.D fd, fs

Purpose:
To move an FP value between FPRs.

Description:
- fd ← fs
  
The value in FPR fs is placed into FPR fd. The source and destination are values in format fmt.

  The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:
The field fs and fd must specify FPRs valid for operands of type fmt; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:
- StoreFPR (fd, fmt, ValueFPR (fs, fmt))

Exceptions:
- Coprocessor Usable
- Reserved Instruction
- Floating-Point
- Unimplemented Operation
MTC1  Move Word to Floating Point  MTC1

<table>
<thead>
<tr>
<th>COP1</th>
<th>MTC1</th>
<th>rt</th>
<th>fs</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00100</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

### MIPS I

**Format:**  
MTC1 rt, fs

**Purpose:**  
To copy a word from a GPR to an FPU (COP1) general register.

**Description:**  
fs ← rt

The low word in GPR rt is placed into the low word of floating-point (coprocessor 1) general register fs. If coprocessor 1 general registers are 64-bits wide, bits 63..32 of register fs become undefined. See Floating-Point Registers on page 10-2.

**Operation:**  

data ← GPR[rt]31..0  
if SizeFGR() = 64 then  
    FGR[fs] ← undefined32|| data  
else  
    FGR[fs] ← data  
endif

**Exceptions:**  
Coprocessor Unusable
**MUL.fmt**

**Floating Point Multiply**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>fmt</td>
<td>ft</td>
<td>fs</td>
<td>fd</td>
<td>MUL</td>
<td>000010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010001</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS I**

**Format:**

- MUL.S \( fd, fs, ft \)
- MUL.D \( fd, fs, ft \)

**Purpose:** To multiply FP values.

**Description:** \( fd \leftarrow fs \times ft \)

The value in FPR \( fs \) is multiplied by the value in FPR \( ft \). The result is calculated to infinite precision, rounded according to the current rounding mode in FCR31, and placed into FPR \( fd \). The operands and result are value in format \( fmt \).

**Restrictions:**

The field \( fs, ft \) and \( fd \) must specify FPRs valid for operands of type \( fmt \); see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

**Operation:**

\[
\text{StoreFPR} \ (fd, \ fmt, \ \text{ValueFPR} \ (fs, \ fmt) \ \ast \ \text{ValueFPR} \ (ft, \ fmt))
\]

**Exceptions:**

- Coprocessor Usable
- Reserved Instruction
- Floating-Point
  - Inexact
  - Unimplemented Operation
- Invalid Operation
- Overflow
- Underflow
### NEGF.fmt

#### Floating Point Negate

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>fs</th>
<th>fd</th>
<th>NEGF</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>00000</td>
<td>fd</td>
<td>000111</td>
</tr>
</tbody>
</table>

#### MIPS I

**Format:**

- NEGF.S   fd, fs
- NEGF.D   fd, fs

**Purpose:** To negate a floating-point value.

**Description:**

\[ fd \leftarrow -(fs) \]

The value in FPR \( fs \) is negated and placed into FPR \( fd \). The value is negated by changing the sign bit value. The operand and result are values in format \( fmt \).

This operation is arithmetic; a NaN operand signals invalid operation.

**Restrictions:**

The field \( fs \) and \( fd \) must specify FPRs valid for operands of type \( fmt \); see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

**Operation:**

\[ \text{StoreFPR} \ (fd, \ fmt, \ \text{Negate} \ (\text{ValueFPR} \ (fs, \ fmt))) \]

**Exceptions:**

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Unimplemented Operation
  - Invalid Operation
ROUND.L.fmt Floating Point Round to Long Fixed-Point ROUND.L.fmt

| COP1 010001 | fmt 000000 | fs 000000 | fd 000000 | ROUND.L 001000 |
| 6 | 5 | 5 | 5 | 6 |

MIPS III

Format: ROUND.L.S fd, fs

ROUND.L.D fd, fs

Purpose: To convert an FP value to 64-bit fixed-point, round to nearest.

Description: fd ← convert_and_round (fs)

The value in FPR fs in format fmt, is converted to a value in 64-bit long fixed-point format rounding to nearest/even (rounding mode 0). The result is placed in FPR fd.

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{63}\) to \(2^{63} - 1\), the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to fd and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{63} - 1\), is written to fd.

Restrictions:

The field fs and fd must specify valid FPRs; fs for type fmt and fd for long fixed point; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt,L))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
Inexact
Unimplemented Operation
Overflow
Invalid Operation
ROUND.W.fmt Floating Point Round to Word Fixed-Point ROUND.W.fmt

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>00000</th>
<th>fs</th>
<th>fd</th>
<th>ROUND.W 001100</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

MIPS II

Format: ROUND.W.S fd, fs
ROUND.W.D fd, fs

Purpose: To convert an FP value to 32-bit fixed-point, round to nearest.

Description: fd ← convert_and_round (fs)

The value in FPR fs in format fmt, is converted to a value in 32-bit word fixed-point format rounding to nearest/even (rounding mode 0). The result is placed in FPR fd.

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{31}\) to \(2^{31} - 1\), the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to fd and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{31} - 1\), is written to fd.

Restrictions:
The field fs and fd must specify valid FPRs; fs for type fmt and fd for word fixed point; see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

Operation:
StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt,W)

Exceptions:
Coprocessor Unusable
Reserved Instruction
Floating-Point
Inexact
Unimplemented Operation
Overflow
Invalid Operation
SDC1 Store Doubleword to Floating-Point

Format: SDC1 ft, offset (base)

Purpose: To store a doubleword from an FPR to memory.

Description: memory[base+offset] ← ft

The 64-bit doubleword in FPR ft is stored in memory at the location specified by the aligned effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

If coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR ft is held in an even/odd register pair. The low word is taken from the even register ft and the high word is from ft+1.

Restrictions:

If ft does not specify an FPR that can contain a doubleword, the result is undefined; see Floating-Point Resistors on page 10-2.

An Address Error exception occurs if EffectiveAddress2..0 ≠ 0 (not doubleword-aligned).

Operation:

vAddr ← sign_extend (offset) + GPR[base]
if vAddr2..0 ≠ 03 then SignalException (AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
if SizeFGR() = 64 then /* 64-bit wide FGRs */
data ← FGR[ft]
elseif ft0 = 0 then /* valid specifier, 32-bit wide FGRs */
data ← FGR[ft+1] || FGR[ft]
else /* undefined for odd 32-bit FGRs */
    UndefinedResult()
endif
StoreMemory(uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)

Exceptions:

Coprocessor Unusable
TLB Refill
TLB Invalid
TLB Modified
Address Error
_FLOATING POINT SQUARE ROOT

**Format:**

- `SQRT.S fd, fs`
- `SQRT.D fd, fs`

**Purpose:**

To compute the square root of an FP value.

**Description:**

- `fd ← SQRT (fs)`

The square root of the value in FPR `fs` is calculated to infinite precision, rounded according to the current rounding mode in FCR31, and placed into FPR `fd`. The operand and result are values in format `fmt`.

If the value in FPR `fs` corresponds to −0, the result will be −0.

**Restrictions:**

- If the value in FPR `fs` is less than 0, an Invalid Operation condition is raised.

The field `fs` and `fd` must specify FPRs valid for operands of type `fmt`; see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

**Operation:**

- `StoreFPR (fd, fmt, SquareRoot (FPR (fs, fmt)))`

**Exceptions:**

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Inexact
  - Unimplemented Operation
- Invalid Operation
SUB.fmt
Floating Point Subtract
SUB.fmt

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>fmt</td>
<td>ft</td>
<td>fs</td>
<td>fd</td>
<td>SUB</td>
<td>010001</td>
<td>000001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS I

Format: SUB.S fd, fs, ft
SUB.S fd, fs, ft

Purpose: To subtract FP values.

Description: fd ← fs - ft

The value in FPR ft is subtracted from the value in FPR fs. The result is calculated to infinite precision, rounded according to the current rounding mode in FCR31, and placed into FPR fd. The operands and result are value in format fmt.

Restrictions:
The field fs, ft, and fd must specify FPRs valid for operands of type fmt; see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

Operation: StoreFPR (fd, fmt, ValueFPR (fs, fmt) – ValueFPR (ft, fmt))

Exceptions:
- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Inexact
  - Unimplemented Operation
  - Invalid Operation
  - Overflow
  - Underflow
SWC1  
Store Word from Floating Point  

<table>
<thead>
<tr>
<th>SWC1</th>
<th>base</th>
<th>ft</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>111001</td>
<td>26</td>
<td>25</td>
<td>21</td>
</tr>
</tbody>
</table>

SWC1

Format: SWC1 ft, offset (base)

Purpose: To store a word from an FPR to memory.

Description: memory[base+offset] ← ft

The low 32-bit word from FPR ft is stored in memory at the location specified by the aligned effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

Restrictions:

An Address Error exception occurs if EffectiveAddress1..0 ≠ 0 (not word-aligned).

Operation: 32-bit Processors

\[
\text{vAddr} \leftarrow \text{sign\_extend}\ (\text{offset}) + \text{GPR}\[\text{base}]\\
\text{if } \text{vAddr1..0} \neq 0^2 \text{ then SignalException (AddressError) endif}\\
\text{(pAddr, uncached) } \leftarrow \text{AddressTranslation (vAddr, DATA, STORE)}\\
\text{data } \leftarrow \text{FGR}[\text{ft}]\\
\text{StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)}
\]

Operation: 64-bit Processors

\[
\text{vAddr} \leftarrow \text{sign\_extend}\ (\text{offset}) + \text{GPR}\[\text{base}]\\
\text{if } \text{vAddr1..0} \neq 0^2 \text{ then SignalException (AddressError) endif}\\
\text{(pAddr, uncached) } \leftarrow \text{AddressTranslation (vAddr, DATA, STORE)}\\
\text{pAddr} \leftarrow \text{pAddr PSIZE}-1..3 || (\text{pAddr2..0 xor (ReverseEndian || 0^2 )})\\
\text{bytesel } \leftarrow \text{vAddr2..0 xor (BigEndianCPU || 0^2 )}\\
/* the bytes of the word are moved into the correct byte lanes */\\
\text{if } \text{SizeFGR()} = 64 \text{ then } /* 64-bit wide FGRs */\\
\text{data } \leftarrow 0^{32\text{-byte}\text{sel}} || \text{FGR}[\text{ft}]31..0 || 0^8\text{bytesel }/* \text{top or bottom wd of 64-bit data */}\text{else}\text{ */ 32-bit wide FGRs */}\\
\text{data } \leftarrow 0^{32\text{-byte}\text{sel}} || \text{FGR}[\text{ft}] || 0^8\text{bytesel }/* \text{top or bottom wd of 64-bit data */}\\
\text{endif}\\
\text{StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)}
\]

Exceptions:

Coprocessor Unusable
TLB Refill
TLB Invalid
TLB Modified
Address Error
**TRUNC.L.fmt** Floating Point Truncate to Long Fixed-Point

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>TRUNC.L</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td>0010001</td>
</tr>
</tbody>
</table>

**MIPS III**

**Format:**
- TRUNC.L.S fd, fs
- TRUNC.L.D fd, fs

**Purpose:** To convert an FP value to 64-bit fixed-point, rounding toward zero.

**Description:** fd ← convert_and_round (fs)

The value in FPR fs in format fmt, is converted to a value in 64-bit long fixed-point format rounding toward zero (rounding mode 1). The result is placed in FPR fd.

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{63}\) to \(2^{63} - 1\), the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to fd and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{63} - 1\), is written to fd.

**Restrictions:**

The fields fs and fd must specify valid FPRs; fs for type fmt and fd for long fixed-point; see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

**Operation:**

\[
\text{StoreFPR} \ (fd, L, \text{ConvertFmt} \ (\text{ValueFPR} \ (fs, fmt), fmt, L))
\]

**Exceptions:**
- Coprocessor Usable
- Reserved Instruction
- Floating-Point
  - Invalid Operation
  - Unimplemented Operation
  - Inexact
  - Overflow
## TRUNC.W.fmt

### Floating Point Truncate to Word Fixed-Point

<table>
<thead>
<tr>
<th>COP1</th>
<th>ft</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>TRUNC.W</th>
<th>001101</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>001101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### MIPS II

**Format:**
- TRUNC.W.S  fd, fs
- TRUNC.W.D  fd, fs

**Purpose:** To convert an FP value to 32-bit fixed-point, rounding toward zero.

**Description:**
\[ \text{fd} \leftarrow \text{convert_and_round} \left( \text{fs} \right) \]

The value in FPR \( \text{fs} \) in format \( \text{fmt} \), is converted to a value in 32-bit word fixed-point format rounding toward zero (rounding mode 1). The result is placed in FPR \( \text{fd} \).

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{31} \) to \(2^{31} - 1\), the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to \( \text{fd} \) and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{31} - 1\), is written to \( \text{fd} \).

**Restrictions:**

The fields \( \text{fs} \) and \( \text{fd} \) must specify valid FPRs; \( \text{fs} \) for type \( \text{fmt} \) and \( \text{fd} \) for word fixed-point; see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

**Operation:**

\[ \text{StoreFPR} \left( \text{fd}, \text{W}, \text{ConvertFmt} \left( \text{ValueFPR} \left( \text{fs}, \text{fmt} \right), \text{fmt}, \text{W} \right) \right) \]

**Exceptions:**

- Coprocessor Unusable
- Reserved Instruction
- Floating-Point
  - Invalid Operation
  - Unimplemented Operation
  - Inexact
  - Overflow
## D.4 COP1 Instruction Encoding

### OpCode

<table>
<thead>
<tr>
<th>OpCode bits 28..26</th>
<th>Instructions encoded by OpCode field (COP1, LWC1, SWC1, LDC1, SDC1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>000 SPECIAL REGIMM J JAL BEQ BNE BLEZ BGTZ</td>
</tr>
<tr>
<td>1 001</td>
<td>ADDI ADDIU SLTI SLTIU ANDI ORI XORI LUI</td>
</tr>
<tr>
<td>2 010</td>
<td>COP0 COP1 δ * * BEQL BNE BGTZL</td>
</tr>
<tr>
<td>3 011</td>
<td>DADDI DADDIU LDL LDR MMI * LQ SQ</td>
</tr>
<tr>
<td>4 100</td>
<td>LB LH LWL LW LBU LHU LWR LWU</td>
</tr>
<tr>
<td>5 101</td>
<td>SB SH SWL SW SDL SDR SWR CACHE</td>
</tr>
<tr>
<td>6 110</td>
<td>η LWC1 η PREF η LDC1 η LD</td>
</tr>
<tr>
<td>7 111</td>
<td>η SWC1 η * η SDC1 η SD</td>
</tr>
</tbody>
</table>

### rs

<table>
<thead>
<tr>
<th>rs bits 23..21</th>
<th>Instructions encoded by rs field when OpCode field = COP1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>000 MFC1 DMFC1 CFC1 * MTC1 DMTC1 CTC1 *</td>
</tr>
<tr>
<td>1 01</td>
<td>BC1 δ * * * * * * *</td>
</tr>
<tr>
<td>2 10</td>
<td>S δ D δ ϕ ϕ W δ L δ ϕ ϕ</td>
</tr>
<tr>
<td>3 11</td>
<td>ϕ ϕ ϕ ϕ ϕ ϕ ϕ ϕ</td>
</tr>
</tbody>
</table>

### rt

<table>
<thead>
<tr>
<th>rt bits 18..16</th>
<th>Instructions encoded by rt field when OpCode field = COP1 &amp; rs field = BC1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>000 BC1F BC1T * * * * * * *</td>
</tr>
<tr>
<td>1 01</td>
<td>* * * * * * * * * * * * *</td>
</tr>
<tr>
<td>2 10</td>
<td>* * * * * * * * * * * *</td>
</tr>
<tr>
<td>3 11</td>
<td>* * * * * * * * * * * *</td>
</tr>
</tbody>
</table>
### Appendix D  COP1 (FPU) Instruction Set Details

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode =</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COP1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs = S, D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>function</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**function** bits 2..0

Instructions encoded by **function** field
when OpCode field = COP1 & rs field = S, D

<table>
<thead>
<tr>
<th>bits</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>0</td>
<td>ADD</td>
<td>SUB</td>
<td>MUL</td>
<td>DIV</td>
<td>SQRT</td>
<td>ABS</td>
<td>MOV</td>
<td>NEG</td>
</tr>
<tr>
<td>1</td>
<td>ROUND.L</td>
<td>TRUNC.L</td>
<td>CEIL.L</td>
<td>FLOOR.L</td>
<td>ROUND.W</td>
<td>TRUNC.W</td>
<td>CEIL.W</td>
<td>FLOOR.W</td>
</tr>
<tr>
<td>2</td>
<td>ROUND.L</td>
<td>TRUNC.L</td>
<td>CEIL.L</td>
<td>FLOOR.L</td>
<td>ROUND.W</td>
<td>TRUNC.W</td>
<td>CEIL.W</td>
<td>FLOOR.W</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>4</td>
<td>CVT.S</td>
<td>CVT.D</td>
<td>CVT.W</td>
<td>CVT.L</td>
<td>CVT.S</td>
<td>CVT.D</td>
<td>CVT.W</td>
<td>CVT.L</td>
</tr>
<tr>
<td>5</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>6</td>
<td>C.F</td>
<td>C.UN</td>
<td>C.EQ</td>
<td>C.UEQ</td>
<td>C.OLT</td>
<td>C.ULT</td>
<td>C.OLE</td>
<td>C.ULE</td>
</tr>
<tr>
<td>7</td>
<td>C.SF</td>
<td>C.NGLE</td>
<td>C.SEQ</td>
<td>C.NGE</td>
<td>C.LT</td>
<td>C.NGE</td>
<td>C.LE</td>
<td>C.NGT</td>
</tr>
</tbody>
</table>

---

This OpCode is reserved for future use. An attempt to execute it causes a Reserved Instruction exception but this is not guaranteed.

This OpCode is reserved for future use. An attempt to execute it produces an undefined result. The result may be an Unimplemented Operation exception.

This OpCode indicates an instruction class. The instruction word must be further decoded by examining additional tables that show the values for another instruction field.

This OpCode is reserved for one of the following instructions which are currently not supported: DMULT, DMULTU, DDIV, DDIVU, LL, LLD, SC, SCD, LWC2, SWC2. An attempt to execute it causes a Reserved Instruction exception.