Public. This publication contains proprietary information which is subject to change without notice and is supplied ‘as is’, without any warranty of any kind.
## Contents

### Chapter 1: About This Book
- 1.1: Typographical Conventions .................................................. 7
  - 1.1.1: Italic Text ........................................................................ 8
  - 1.1.2: Bold Text .......................................................................... 8
  - 1.1.3: Courier Text ...................................................................... 8
- 1.2: UNPREDICTABLE and UNDEFINED ........................................ 8
  - 1.2.1: UNPREDICTABLE ............................................................... 8
  - 1.2.2: UNDEFINED ..................................................................... 9
  - 1.2.3: UNSTABLE ........................................................................ 9
- 1.3: Special Symbols in Pseudocode Notation ..................................... 9
- 1.4: Notation for Register Field Accessibility .................................... 12
- 1.5: For More Information ............................................................. 14

### Chapter 2: Guide to the Instruction Set
- 2.1: Understanding the Instruction Fields ......................................... 15
  - 2.1.1: Instruction Fields ............................................................... 16
  - 2.1.2: Instruction Descriptive Name and Mnemonic ....................... 17
  - 2.1.3: Format Field ...................................................................... 17
  - 2.1.4: Purpose Field .................................................................... 18
  - 2.1.5: Description Field ............................................................ 18
  - 2.1.6: Restrictions Field ............................................................ 18
  - 2.1.7: Availability and Compatibility Fields ................................. 19
  - 2.1.8: Operation Field ............................................................... 19
  - 2.1.9: Exceptions Field ............................................................... 20
  - 2.1.10: Programming Notes and Implementation Notes Fields ......... 20
- 2.2: Operation Section Notation and Functions ............................... 20
  - 2.2.1: Instruction Execution Ordering ......................................... 21
  - 2.2.2: Pseudocode Functions ....................................................... 21
    - 2.2.2.1: Coprocessor General Register Access Functions ............ 21
    - 2.2.2.2: Memory Operation Functions ........................................ 22
    - 2.2.2.3: Floating Point Functions .............................................. 26
    - 2.2.2.4: Pseudocode Functions Related to Sign and Zero Extension 29
    - 2.2.2.5: Miscellaneous Functions .............................................. 30
- 2.3: Op and Function Subfield Notation ............................................ 32
- 2.4: FPU Instructions ..................................................................... 32

### Chapter 3: The MIPS® DSP Application Specific Extension to the MIPS® Architecture
- 3.1: Base Architecture Requirements ............................................. 33
- 3.2: Software Detection of the Module ........................................... 33
- 3.3: Compliance and Subsetting .................................................... 33
- 3.4: Introduction to the MIPS® DSP Module .................................... 34
- 3.5: DSP Applications and their Requirements ................................ 34
- 3.6: Fixed-Point Data Types .......................................................... 35
- 3.7: Saturating Math ..................................................................... 36
- 3.8: Conventions Used in the Instruction Mnemonics ....................... 37
- 3.9: Effect of Endian-ness on Register SIMD Data .......................... 38
- 3.10: Additional Register State for the DSP Module ....................... 39
### 3.11: Software Detection of the DSP Module ................................................................. 41
### 3.12: Exception Table for the DSP Module ................................................................. 42
### 3.13: DSP Module Instructions that Read and Write the DSPControl Register .......... 42
### 3.14: Arithmetic Exceptions ....................................................................................... 43

### Chapter 4: MIPS® DSP Module Instruction Summary .................................................. 45
#### 4.1: The MIPS® DSP Module Instruction Summary ..................................................... 45

### Chapter 5: Instruction Encoding .................................................................................. 61
#### 5.1: Instruction Bit Encoding ....................................................................................... 61

### Chapter 6: The MIPS® DSP Module Instruction Set ..................................................... 69
#### 6.1: Compliance and Subsetting .................................................................................. 69
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSQ_S.PH</td>
<td>70</td>
</tr>
<tr>
<td>ABSQ_S.QB</td>
<td>71</td>
</tr>
<tr>
<td>ABSQ_S.W</td>
<td>72</td>
</tr>
<tr>
<td>ADDQ_[S].PH</td>
<td>73</td>
</tr>
<tr>
<td>ADDQ_S.W</td>
<td>75</td>
</tr>
<tr>
<td>ADDQH_[R].PH</td>
<td>76</td>
</tr>
<tr>
<td>ADDQH_[R].W</td>
<td>78</td>
</tr>
<tr>
<td>ADDSC</td>
<td>79</td>
</tr>
<tr>
<td>ADDU_[S].PH</td>
<td>80</td>
</tr>
<tr>
<td>ADDU_[S].QB</td>
<td>81</td>
</tr>
<tr>
<td>ADDWC</td>
<td>83</td>
</tr>
<tr>
<td>ADDUH_[R].QB</td>
<td>84</td>
</tr>
<tr>
<td>APPEND</td>
<td>86</td>
</tr>
<tr>
<td>BALIGN</td>
<td>87</td>
</tr>
<tr>
<td>BITREV</td>
<td>88</td>
</tr>
<tr>
<td>BPOSGE32</td>
<td>89</td>
</tr>
<tr>
<td>BPOSGE32C</td>
<td>90</td>
</tr>
<tr>
<td>CMP.cond.PH</td>
<td>91</td>
</tr>
<tr>
<td>CMPGDU kond.QB</td>
<td>92</td>
</tr>
<tr>
<td>CMPGU kond.QB</td>
<td>94</td>
</tr>
<tr>
<td>CMPU kond.QB</td>
<td>96</td>
</tr>
<tr>
<td>DPA.W.PH</td>
<td>98</td>
</tr>
<tr>
<td>DPAQ_S.W.PH</td>
<td>99</td>
</tr>
<tr>
<td>DPAQ_S.A.L.W</td>
<td>100</td>
</tr>
<tr>
<td>DPAQX_S.W.PH</td>
<td>102</td>
</tr>
<tr>
<td>DPAQX_S.A.W.PH</td>
<td>103</td>
</tr>
<tr>
<td>DPAU_H.QBL</td>
<td>105</td>
</tr>
<tr>
<td>DPAU_H.QBR</td>
<td>106</td>
</tr>
<tr>
<td>DPAX_W.PH</td>
<td>107</td>
</tr>
<tr>
<td>DPS_W.PH</td>
<td>108</td>
</tr>
<tr>
<td>DPSQ_S.W.PH</td>
<td>109</td>
</tr>
<tr>
<td>DPSQ_S.A.L.W</td>
<td>110</td>
</tr>
<tr>
<td>DPSQX_S.W.PH</td>
<td>111</td>
</tr>
<tr>
<td>DPSQX_S.A.W.PH</td>
<td>112</td>
</tr>
<tr>
<td>DPSU_H.QBL</td>
<td>114</td>
</tr>
<tr>
<td>DPSU_H.QBR</td>
<td>115</td>
</tr>
<tr>
<td>DPSX_W.PH</td>
<td>116</td>
</tr>
<tr>
<td>EXTP</td>
<td>117</td>
</tr>
<tr>
<td>EXTPDP</td>
<td>118</td>
</tr>
<tr>
<td>Instruction</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------</td>
<td>------</td>
</tr>
<tr>
<td>PRECRQ_RS.PH.W</td>
<td>185</td>
</tr>
<tr>
<td>PREPEND</td>
<td>186</td>
</tr>
<tr>
<td>RADDU.W.QB</td>
<td>187</td>
</tr>
<tr>
<td>RDDSP</td>
<td>188</td>
</tr>
<tr>
<td>REPL.PH</td>
<td>190</td>
</tr>
<tr>
<td>REPL.QB</td>
<td>191</td>
</tr>
<tr>
<td>REPLV.PH</td>
<td>192</td>
</tr>
<tr>
<td>REPLV.QB</td>
<td>193</td>
</tr>
<tr>
<td>SHILO</td>
<td>194</td>
</tr>
<tr>
<td>SHILOV</td>
<td>195</td>
</tr>
<tr>
<td>SHLL[_S].PH</td>
<td>196</td>
</tr>
<tr>
<td>SHLL.QB</td>
<td>198</td>
</tr>
<tr>
<td>SHLLV[_S].PH</td>
<td>199</td>
</tr>
<tr>
<td>SHLLV.QB</td>
<td>200</td>
</tr>
<tr>
<td>SHLLV_S.W</td>
<td>201</td>
</tr>
<tr>
<td>SHLL_S.W</td>
<td>202</td>
</tr>
<tr>
<td>SHRA[ R].QB</td>
<td>203</td>
</tr>
<tr>
<td>SHRA[ R].PH</td>
<td>205</td>
</tr>
<tr>
<td>SHRAV[ R].PH</td>
<td>207</td>
</tr>
<tr>
<td>SHRAV[ R].QB</td>
<td>208</td>
</tr>
<tr>
<td>SHRAV_R.W</td>
<td>210</td>
</tr>
<tr>
<td>SHRA_R.W</td>
<td>211</td>
</tr>
<tr>
<td>SHR_LPH</td>
<td>212</td>
</tr>
<tr>
<td>SHR_L.QB</td>
<td>213</td>
</tr>
<tr>
<td>SHR_LV.PH</td>
<td>214</td>
</tr>
<tr>
<td>SHR_LV.QB</td>
<td>215</td>
</tr>
<tr>
<td>SUBQ[ S].PH</td>
<td>216</td>
</tr>
<tr>
<td>SUBQ_S.W</td>
<td>218</td>
</tr>
<tr>
<td>SUBQH[ R].PH</td>
<td>219</td>
</tr>
<tr>
<td>SUBQH[ R].W</td>
<td>221</td>
</tr>
<tr>
<td>SUBU[ S].PH</td>
<td>222</td>
</tr>
<tr>
<td>SUBU[ S].QB</td>
<td>224</td>
</tr>
<tr>
<td>SUBUH[ R].QB</td>
<td>226</td>
</tr>
<tr>
<td>WRDSP</td>
<td>227</td>
</tr>
</tbody>
</table>

**Appendix A: Endian-Agnostic Reference to Register Elements** ................................................. 229
A.1: Using Endian-Agnostic Instruction Names .............................................................................. 229
A.2: Mapping Endian-Agnostic Instruction Names to DSP Module Instructions .............................. 229

**Appendix B: Revision History** .................................................................................................. 233
Chapter 1

About This Book

The MIPS® DSP Module for MIPS32™ Architecture comes as part of a multi-volume set.

- Volume I-A describes conventions used throughout the document set, and provides an introduction to the MIPS32® Architecture
- Volume I-B describes conventions used throughout the document set, and provides an introduction to the micro-MIPS™ Architecture
- Volume II-A provides detailed descriptions of each instruction in the MIPS32® instruction set
- Volume II-B provides detailed descriptions of each instruction in the microMIPS32™ instruction set
- Volume III describes the MIPS32® and microMIPS32™ Privileged Resource Architecture which defines and governs the behavior of the privileged resources included in a MIPS® processor implementation
- Volume IV-a describes the MIPS16e™ Application-Specific Extension to the MIPS32® Architecture. Beginning with Release 3 of the Architecture, microMIPS is the preferred solution for smaller code size. Release 6 removes MIPS16e: MIPS16e cannot be implemented with Release 6.
- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS64® Architecture and microMIPS64™. With Release 5 of the Architecture, MDMX is deprecated. MDMX and MSA can not be implemented at the same time. Release 6 removes MDMX: MDMX cannot be implemented with Release 6.
- Volume IV-c describes the MIPS-3D® Application-Specific Extension to the MIPS® Architecture. Release 6 removes MIPS-3D: MIPS-3D cannot be implemented with Release 6.
- Volume IV-d describes the SmartMIPS® Application-Specific Extension to the MIPS32® Architecture and the microMIPS32™ Architecture. Release 6 removes SmartMIPS: SmartMIPS cannot be implemented with Release 6, neither MIPS32 Release 6 nor MIPS64 Release 6.
- Volume IV-e describes the MIPS® DSP Module to the MIPS® Architecture.
- Volume IV-f describes the MIPS® MT Module to the MIPS® Architecture
- Volume IV-h describes the MIPS® MCU Application-Specific Extension to the MIPS® Architecture
- Volume IV-i describes the MIPS® Virtualization Module to the MIPS® Architecture
- Volume IV-j describes the MIPS® SIMD Architecture Module to the MIPS® Architecture

1.1 Typographical Conventions

This section describes the use of italic, bold and courier fonts in this book.
1.1.1 Italic Text

- is used for emphasis
- is used for bits, fields, and registers that are important from a software perspective (for instance, address bits used by software, and programmable fields and registers), and various floating point instruction formats, such as $S$ and $D$
- is used for the memory access types, such as cached and uncached

1.1.2 Bold Text

- represents a term that is being defined
- is used for bits and fields that are important from a hardware perspective (for instance, register bits, which are not programmable but accessible only to hardware)
- is used for ranges of numbers; the range is indicated by an ellipsis. For instance, 5..1 indicates numbers 5 through 1
- is used to emphasize UNPREDICTABLE and UNDEFINED behavior, as defined below.

1.1.3 Courier Text

Courier fixed-width font is used for text that is displayed on the screen, and for examples of code and instruction pseudocode.

1.2 UNPREDICTABLE and UNDEFINED

The terms UNPREDICTABLE and UNDEFINED are used throughout this book to describe the behavior of the processor in certain cases. UNDEFINED behavior or operations can occur only as the result of executing instructions in a privileged mode (i.e., in Kernel Mode or Debug Mode, or with the CP0 usable bit set in the Status register). Unprivileged software can never cause UNDEFINED behavior or operations. Conversely, both privileged and unprivileged software can cause UNPREDICTABLE results or operations.

1.2.1 UNPREDICTABLE

UNPREDICTABLE results may vary from processor implementation to implementation, instruction to instruction, or as a function of time on the same implementation or instruction. Software can never depend on results that are UNPREDICTABLE. UNPREDICTABLE operations may cause a result to be generated or not. If a result is generated, it is UNPREDICTABLE. UNPREDICTABLE operations may cause arbitrary exceptions.

UNPREDICTABLE results or operations have several implementation restrictions:

- Implementations of operations generating UNPREDICTABLE results must not depend on any data source (memory or internal state) which is inaccessible in the current processor mode
- UNPREDICTABLE operations must not read, write, or modify the contents of memory or internal state which is inaccessible in the current processor mode. For example, UNPREDICTABLE operations executed in user mode must not access memory or internal state that is only accessible in Kernel Mode or Debug Mode or in another process
1.3 Special Symbols in Pseudocode Notation

- **UNPREDICTABLE** operations must not halt or hang the processor.

1.2.2 UNDEFINED

UNDEFINED operations or behavior may vary from processor implementation to implementation, instruction to instruction, or as a function of time on the same implementation or instruction. UNDEFINED operations or behavior may vary from nothing to creating an environment in which execution can no longer continue. UNDEFINED operations or behavior may cause data loss.

UNDEFINED operations or behavior has one implementation restriction:

- UNDEFINED operations or behavior must not cause the processor to hang (that is, enter a state from which there is no exit other than powering down the processor). The assertion of any of the reset signals must restore the processor to an operational state.

1.2.3 UNSTABLE

UNSTABLE results or values may vary as a function of time on the same implementation or instruction. Unlike UNPREDICTABLE values, software may depend on the fact that a sampling of an UNSTABLE value results in a legal transient value that was correct at some point in time prior to the sampling.

UNSTABLE values have one implementation restriction:

- Implementations of operations generating UNSTABLE results must not depend on any data source (memory or internal state) which is inaccessible in the current processor mode.

1.3 Special Symbols in Pseudocode Notation

In this book, algorithmic descriptions of an operation are described using a high-level language pseudocode resembling Pascal. Special symbols used in the pseudocode notation are listed in Table 1.1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>:=, ...</td>
<td>Tests for equality and inequality</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>xy</td>
<td>A y-bit string formed by y copies of the single-bit value x</td>
</tr>
<tr>
<td>b#n</td>
<td>A constant value n in base b. For instance 10#100 represents the decimal value 100, 2#100 represents the binary value 100 (decimal 4), and 16#100 represents the hexadecimal value 100 (decimal 256). If the &quot;b#&quot; prefix is omitted, the default base is 10.</td>
</tr>
<tr>
<td>0bn</td>
<td>A constant value n in base 2. For instance 0b100 represents the binary value 100 (decimal 4).</td>
</tr>
<tr>
<td>0xn</td>
<td>A constant value n in base 16. For instance 0x100 represents the hexadecimal value 100 (decimal 256).</td>
</tr>
<tr>
<td>xy z</td>
<td>Selection of bits y through z of bit string x. Little-endian bit notation (rightmost bit is 0) is used. If y is less than z, this expression is an empty (zero length) bit string.</td>
</tr>
<tr>
<td>x.bit[y]</td>
<td>Bit y of bitstring x. Alternative to the traditional MIPS notation x_y.</td>
</tr>
<tr>
<td>x.bits[y..z]</td>
<td>Selection of bits y through z of bit string x. Alternative to the traditional MIPS notation x_y z.</td>
</tr>
</tbody>
</table>
### Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.byte[y]</td>
<td>Byte y of bitstring x. Equivalent to the traditional MIPS notation x_8*y+7 8*y.</td>
</tr>
<tr>
<td>x.bytes[y..z]</td>
<td>Selection of bytes y through z of bit string x. Alternative to the traditional MIPS notation x_8*y+7 8*z</td>
</tr>
<tr>
<td>x halfword[y] x.word[i] x.doubleword[i]</td>
<td>Similar extraction of particular bitfields (used in e.g., MSA packed SIMD vectors).</td>
</tr>
<tr>
<td>x.bit31, x.byte0, etc.</td>
<td>Examples of abbreviated form of x.bit[y], etc. notation, when y is a constant.</td>
</tr>
<tr>
<td>x fieldy</td>
<td>Selection of a named subfield of bitstring x, typically a register or instruction encoding. More formally described as “Field y of register x”. For example, FIR.D = “the D bit of the Coprocessor 1 Floating-point Implementation Register (FIR)”.</td>
</tr>
<tr>
<td>+, −</td>
<td>2’s complement or floating point arithmetic: addition, subtraction</td>
</tr>
<tr>
<td>*, ∞</td>
<td>2’s complement or floating point multiplication (both used for either)</td>
</tr>
<tr>
<td>div</td>
<td>2’s complement integer division</td>
</tr>
<tr>
<td>mod</td>
<td>2’s complement modulo</td>
</tr>
<tr>
<td>/</td>
<td>Floating point division</td>
</tr>
<tr>
<td>&lt;</td>
<td>2’s complement less-than comparison</td>
</tr>
<tr>
<td>&gt;</td>
<td>2’s complement greater-than comparison</td>
</tr>
<tr>
<td>≤</td>
<td>2’s complement less-than or equal comparison</td>
</tr>
<tr>
<td>≥</td>
<td>2’s complement greater-than or equal comparison</td>
</tr>
<tr>
<td>nor</td>
<td>Bitwise logical NOR</td>
</tr>
<tr>
<td>xor</td>
<td>Bitwise logical XOR</td>
</tr>
<tr>
<td>and</td>
<td>Bitwise logical AND</td>
</tr>
<tr>
<td>or</td>
<td>Bitwise logical OR</td>
</tr>
<tr>
<td>not</td>
<td>Bitwise inversion</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical (non-Bitwise) AND</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Logical Shift left (shift in zeros at right-hand-side)</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Logical Shift right (shift in zeros at left-hand-side)</td>
</tr>
<tr>
<td>GPRLEN</td>
<td>The length in bits (32 or 64) of the CPU general-purpose registers</td>
</tr>
<tr>
<td>GPR[s]</td>
<td>CPU general-purpose register x. The content of GPR[0] is always zero. In Release 2 of the Architecture, GPR[x] is a short-hand notation for SGPR[SRSCtlCSS, x].</td>
</tr>
<tr>
<td>SGPR[s,x]</td>
<td>In Release 2 of the Architecture and subsequent releases, multiple copies of the CPU general-purpose registers may be implemented. SGPR[s,x] refers to GPR set s, register x.</td>
</tr>
<tr>
<td>FPR[x]</td>
<td>Floating Point operand register x</td>
</tr>
<tr>
<td>FCC[CC]</td>
<td>Floating Point condition code CC. FCC[0] has the same value as COC[1]. Release 6 removes the floating point condition codes.</td>
</tr>
<tr>
<td>FPR[x]</td>
<td>Floating Point (Coprocessor unit 1), general register x</td>
</tr>
<tr>
<td>CPR[z,x,s]</td>
<td>Coprocessor unit z, general register x, select s</td>
</tr>
<tr>
<td>CP2CPR[x]</td>
<td>Coprocessor unit 2, general register x</td>
</tr>
<tr>
<td>CCR[z,x]</td>
<td>Coprocessor unit z, control register x</td>
</tr>
<tr>
<td>CP2CCR[x]</td>
<td>Coprocessor unit 2, control register x</td>
</tr>
</tbody>
</table>
## 1.3 Special Symbols in Pseudocode Notation

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>COC</strong>&lt;sub&gt;z&lt;/sub&gt;</td>
<td>Coprocessor unit z condition signal</td>
</tr>
<tr>
<td><strong>Xlat</strong>&lt;sub&gt;x&lt;/sub&gt;</td>
<td>Translation of the MIPS16e GPR number x into the corresponding 32-bit GPR number</td>
</tr>
<tr>
<td><strong>BigEndianMem</strong></td>
<td>Endian mode as configured at chip reset (0 (\rightarrow) Little-Endian, 1 (\rightarrow) Big-Endian). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory pseudocode function descriptions) and the endianness of Kernel and Supervisor mode execution.</td>
</tr>
<tr>
<td><strong>BigEndianCPU</strong></td>
<td>The endianness for load and store instructions (0 (\rightarrow) Little-Endian, 1 (\rightarrow) Big-Endian). In User mode, this endianness may be switched by setting the RE bit in the Status register. Thus, BigEndianCPU may be computed as (BigEndianMem XOR ReverseEndian).</td>
</tr>
<tr>
<td><strong>ReverseEndian</strong></td>
<td>Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is implemented by setting the RE bit of the Status register. Thus, ReverseEndian may be computed as (SR&lt;sub&gt;RE&lt;/sub&gt; and User mode).</td>
</tr>
<tr>
<td><strong>LLbit</strong></td>
<td>Bit of virtual state used to specify operation for instructions that provide atomic read-modify-write. LLbit is set when a linked load occurs and is tested by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions.</td>
</tr>
<tr>
<td><strong>I</strong>, <strong>I+n</strong>, <strong>I-n</strong></td>
<td>This occurs as a prefix to Operation description lines and functions as a label. It indicates the instruction time during which the pseudocode appears to “execute.” 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. Sometimes effects of an instruction appear to occur either earlier or later — that is, during the instruction time of another instruction. When this happens, the instruction operation is written in sections labeled with the instruction time, relative to the current instruction 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 has the portion of the instruction operation description that writes the result register in a section labeled <strong>I+1</strong>. The effect of pseudocode statements for the current instruction labeled <strong>I+1</strong> appears to occur “at the same time” as the effect of pseudocode statements labeled I for the following instruction. Within one pseudocode sequence, the effects of the statements take place in order. However, between sequences of statements for different instructions that occur “at the same time,” there is no defined order. Programs must not depend on a particular order of evaluation between such sections.</td>
</tr>
<tr>
<td><strong>PC</strong></td>
<td>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 an instruction time by any pseudocode statement, it is automatically incremented by either 2 (in the case of a 16-bit MIPS16e instruction) or 4 before the next instruction time. A taken branch assigns the target address to the <strong>PC</strong> during the instruction time of the instruction in the branch delay slot. In the MIPS Architecture, the <strong>PC</strong> value is only visible indirectly, such as when the processor stores the restart address into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception. Release 6 adds PC-relative address computation and load instructions. The PC value contains a full 32-bit address, all of which are significant during a memory reference.</td>
</tr>
</tbody>
</table>
1.4 Notation for Register Field Accessibility

In this document, the read/write properties of register fields use the notations shown in Table 1.1.

Table 1.2 Read/Write Register Field Notation

<table>
<thead>
<tr>
<th>Read/Write Notation</th>
<th>Hardware Interpretation</th>
<th>Software Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W</td>
<td>A field in which all bits are readable and writable by software and, potentially, by hardware. Hardware updates of this field are visible by software read. Software updates of this field are visible by hardware read. If the Reset State of this field is “Undefined”, either software or hardware must initialize the value before the first read will return a predictable value. This should not be confused with the formal definition of UNDEFINED behavior.</td>
<td></td>
</tr>
</tbody>
</table>
### 1.4 Notation for Register Field Accessibility

#### Table 1.2 Read/Write Register Field Notation (Continued)

<table>
<thead>
<tr>
<th>Read/Write Notation</th>
<th>Hardware Interpretation</th>
<th>Software Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>A field which is either static or is updated only by hardware. If the Reset State of this field is either “0”, “Preset”, or “Externally Set”, hardware initializes this field to zero or to the appropriate state, respectively, on powerup. The term “Preset” is used to suggest that the processor establishes the appropriate state, whereas the term “Externally Set” is used to suggest that the state is established via an external source (e.g., personality pins or initialization bit stream). These terms are suggestions only, and are not intended to act as a requirement on the implementation. If the Reset State of this field is “Undefined”, hardware updates this field only under those conditions specified in the description of the field.</td>
<td>A field to which the value written by software is ignored by hardware. Software may write any value to this field without affecting hardware behavior. Software reads of this field return the last value updated by hardware. If the Reset State of this field is “Undefined”, software reads of this field result in an UNPREDICTABLE value except after a hardware update done under the conditions specified in the description of the field.</td>
</tr>
<tr>
<td>R0</td>
<td>R0 = reserved, read as zero, ignore writes by software. Hardware ignores software writes to an R0 field. Neither the occurrence of such writes, nor the values written, affects hardware behavior. Hardware always returns 0 to software reads of R0 fields. The Reset State of an R0 field must always be 0. If software performs an mtc0 instruction which writes a non-zero value to an R0 field, the write to the R0 field will be ignored, but permitted writes to other fields in the register will not be affected.</td>
<td>Architectural Compatibility: R0 fields are reserved, and may be used for not-yet-defined purposes in future revisions of the architecture. When writing an R0 field, current software should only write either all 0s, or, preferably, write back the same value that was read from the field. Current software should not assume that the value read from R0 fields is zero, because this may not be true on future hardware. Future revisions of the architecture may redefine an R0 field, but must do so in such a way that software which is unaware of the new definition and either writes zeros or writes back the value it has read from the field will continue to work correctly. Writing back the same value that was read is guaranteed to have no unexpected effects on current or future hardware behavior. (Except for non-atomicity of such read-writes.) Writing zeros to an R0 field may not be preferred because in the future this may interfere with the operation of other software which has been updated for the new field definition.</td>
</tr>
</tbody>
</table>
### Table 1.2 Read/Write Register Field Notation (Continued)

<table>
<thead>
<tr>
<th>Read/Write Notation</th>
<th>Hardware Interpretation</th>
<th>Software Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Release 6</strong></td>
<td>Release 6 legacy “0” behaves like R0 - read as zero, nonzero writes ignored. Legacy “0” should not be defined for any new control register fields; R0 should be used instead.</td>
<td>Only zero should be written, or, value read from register.</td>
</tr>
<tr>
<td>HW returns 0 when read.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HW ignores writes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>pre-Release 6</strong></td>
<td>pre-Release 6 legacy “0” - read as zero, nonzero writes UNDEFINED</td>
<td>A field to which the value written by software must be zero. Software writes of non-zero values to this field may result in UNDEFINED behavior of the hardware. Software reads of this field return zero as long as all previous software writes are zero. If the Reset State of this field is “Undefined”, software must write this field with zero before it is guaranteed to read as zero.</td>
</tr>
<tr>
<td>A field which hardware does not update, and for which hardware can assume a zero value.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| R/W0                | Like R/W, except that writes of non-zero to a R/W0 field are ignored. E.g. Status.NMI |                         |
| Hardware may set or clear an R/W0 bit. |                         | Software can only clear an R/W0 bit. |
| Hardware ignores software writes of nonzero to an R/W0 field. Neither the occurrence of such writes, nor the values written, affects hardware behavior. |                         | Software writes 0 to an R/W0 field to clear the field. |
| Software writes of 0 to an R/W0 field may have an effect. |                         | Software writes nonzero to an R/W0 bit in order to guarantee that the bit is not affected by the write. |
| Hardware may return 0 or nonzero to software reads of an R/W0 bit. |                         |                         |
| If software performs an mtc0 instruction which writes a non-zero value to an R/W0 field, the write to the R/W0 field will be ignored, but permitted writes to other fields in the register will not be affected. |                         |                         |

1.5 For More Information

MIPS processor manuals and additional information about MIPS products can be found at http://www.mips.com.
Guide to the Instruction Set

This chapter provides a detailed guide to understanding the instruction descriptions, which are listed in alphabetical order in the tables at the beginning of the next chapter.

2.1 Understanding the Instruction Fields

Figure 2.1 shows an example instruction. Following the figure are descriptions of the fields listed below:

- “Instruction Fields” on page 16
- “Instruction Descriptive Name and Mnemonic” on page 17
- “Format Field” on page 17
- “Purpose Field” on page 18
- “Description Field” on page 18
- “Restrictions Field” on page 18
- “Operation Field” on page 19
- “Exceptions Field” on page 20
- “Programming Notes and Implementation Notes Fields” on page 20
### Figure 2.1 Example of Instruction Description

<table>
<thead>
<tr>
<th>Instruction Mnemonic and Descriptive Name</th>
<th>Example Instruction Name</th>
<th>EXAMPLE</th>
</tr>
</thead>
</table>

<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>0</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>000000</td>
<td>EXAMPLE</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Architecture level at which instruction was defined/**

**Assembler format(s) for each definition**
- **Short**
- **Symbolic**

**Full description of instruction**

**Restrictions on instruction and**

**High-level language description of instruction operation**

```
/* This section describes the operation of an instruction in */
/* a high-level pseudo-language. It is precise in ways that */
/* the Description section is not, but is also missing */
/* information that is hard to express in pseudocode. */
temp = GPR[rs] exampleop GPR[rt]
GPR[rd] = temp
```

**Exceptions:**

A list of exceptions taken by the instruction

**Programming Notes:**

Information useful to programmers, but not necessary to describe the operation of the instruction

**Implementation Notes:**

Like *Programming Notes*, except for processor implementors

---

### 2.1.1 Instruction Fields
2.1 Understanding the Instruction Fields

Fields encoding the instruction word are shown in register form at the top of the instruction description. The following rules are followed:

- The values of constant fields and the *opcode* names are listed in uppercase (SPECIAL and ADD in Figure 2.2). Constant values in a field are shown in binary below the symbolic or hexadecimal value.

- All variable fields are listed with the lowercase names used in the instruction description (*rs*, *rt*, and *rd* in Figure 2.2).

- Fields that contain zeros but are not named are unused fields that are required to be zero (bits 10:6 in Figure 2.2). If such fields are set to non-zero values, the operation of the processor is **UNPREDICTABLE**.

**Figure 2.2 Example of Instruction Fields**

<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>0</td>
<td>0</td>
<td>ADD</td>
<td>0</td>
<td>0</td>
<td>100000</td>
<td></td>
</tr>
</tbody>
</table>

2.1.2 Instruction Descriptive Name and Mnemonic

The instruction descriptive name and mnemonic are printed as page headings for each instruction, as shown in Figure 2.3.

**Figure 2.3 Example of Instruction Descriptive Name and Mnemonic**

Add Word

ADD

2.1.3 Format Field

The assembler formats for the instruction and the architecture level at which the instruction was originally defined are given in the *Format* field. If the instruction definition was later extended, the architecture levels at which it was extended and the assembler formats for the extended definition are shown in their order of extension (for an example, see C.cond fmt). The MIPS architecture levels are inclusive; higher architecture levels include all instructions in previous levels. Extensions to instructions are backwards compatible. The original assembler formats are valid for the extended architecture.

**Figure 2.4 Example of Instruction Format**

<table>
<thead>
<tr>
<th>Format:</th>
<th>MIPS32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD fd, rs, rt</td>
<td>MIPS32</td>
</tr>
</tbody>
</table>

The assembler format is shown with literal parts of the assembler instruction printed in uppercase characters. The variable parts, the operands, are shown as the lowercase names of the appropriate fields.

The architectural level at which the instruction was first defined, for example “MIPS32” is shown at the right side of the page. Instructions introduced at different times by different ISA family members, are indicated by markings such as “MIPS64, MIPS32 Release 2”. Instructions removed by particular architecture release are indicated in the Availability section.
There can be more than one assembler format for each architecture level. Floating point operations on formatted data show an assembly format with the actual assembler mnemonic for each valid value of the \texttt{fmt} field. For example, the ADD \texttt{fmt} instruction lists both ADD.S and ADD.D.

The assembler format lines sometimes include parenthetical comments to help explain variations in the formats (once again, see C.cond.fmt). These comments are not a part of the assembler format.

### 2.1.4 Purpose Field

The \textit{Purpose} field gives a short description of the use of the instruction.

**Figure 2.5 Example of Instruction Purpose**

<table>
<thead>
<tr>
<th>Purpose: Add Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>To add 32-bit integers. If an overflow occurs, then trap.</td>
</tr>
</tbody>
</table>

### 2.1.5 Description Field

If a one-line symbolic description of the instruction is feasible, it appears immediately to the right of the \textit{Description} heading. The main purpose is to show how fields in the instruction are used in the arithmetic or logical operation.

**Figure 2.6 Example of Instruction Description**

<table>
<thead>
<tr>
<th>Description: GPR[rd] ( \diamond ) GPR[rs] + GPR[rt]</th>
</tr>
</thead>
<tbody>
<tr>
<td>The 32-bit word value in GPR ( rt ) is added to the 32-bit value in GPR ( rs ) to produce a 32-bit result.</td>
</tr>
</tbody>
</table>

- If the addition results in 32-bit 2’s complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is placed into GPR \( rd \).

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 \textit{Operation} section.

This section uses acronyms for register descriptions. “GPR \( rt \)” is CPU general-purpose register specified by the instruction field \( rt \). “FPR \( fs \)” is the floating point operand register specified by the instruction field \( fs \). “CP1 register \( fd \)” is the coprocessor 1 general register specified by the instruction field \( fd \). “FCSR” is the floating point \textit{Control / Status} register.

### 2.1.6 Restrictions Field

The \textit{Restrictions} field documents any possible restrictions that may affect the instruction. Most restrictions fall into one of the following six categories:

- Valid values for instruction fields (for example, see floating point ADD.fmt)
- ALIGNMENT requirements for memory addresses (for example, see LW)
- Valid values of operands (for example, see )
2.1 Understanding the Instruction Fields

- Valid operand formats (for example, see floating point ADD.fmt)
- Order of instructions necessary to guarantee correct execution. These ordering constraints avoid pipeline hazards for which some processors do not have hardware interlocks (for example, see MUL).
- Valid memory access types (for example, see LL/SC)

Figure 2.7 Example of Instruction Restrictions

<table>
<thead>
<tr>
<th>Restrictions:</th>
</tr>
</thead>
</table>

2.1.7 Availability and Compatibility Fields

The Availability and Compatibility sections are not provided for all instructions. These sections list considerations relevant to whether and how an implementation may implement some instructions, when software may use such instructions, and how software can determine if an instruction or feature is present. Such considerations include:

- Some instructions are not present on all architecture releases. Sometimes the implementation is required to signal a Reserved Instruction exception, but sometimes executing such an instruction encoding is architecturally defined to give UNPREDICTABLE results.
- Some instructions are available for implementations of a particular architecture release, but may be provided only if an optional feature is implemented. Control register bits typically allow software to determine if the feature is present.
- Some instructions may not behave the same way on all implementations. Typically this involves behavior that was UNPREDICTABLE in some implementations, but which is made architectural and guaranteed consistent so that software can rely on it in subsequent architecture releases.
- Some instructions are prohibited for certain architecture releases and/or optional feature combinations.
- Some instructions may be removed for certain architecture releases. Implementations may then be required to signal a Reserved Instruction exception for the removed instruction encoding; but sometimes the instruction encoding is reused for other instructions.

All of these considerations may apply to the same instruction. If such considerations applicable to an instruction are simple, the architecture level in which an instruction was defined or redefined in the Format field, and/or the Restrictions section, may be sufficient; but if the set of such considerations applicable to an instruction is complicated, the Availability and Compatibility sections may be provided.

2.1.8 Operation Field

The Operation field describes the operation of the instruction as pseudocode in a high-level language notation resembling Pascal. This formal description complements the Description section; it is not complete in itself because many of the restrictions are either difficult to include in the pseudocode or are omitted for legibility.
Figure 2.8 Example of Instruction Operation

Operation:

\[
\text{temp} \leftarrow (\text{GPR}[rs]_{31} | \text{GPR}[rs]_{31..0}) + (\text{GPR}[rt]_{31} | \text{GPR}[rt]_{31..0}) \\
\text{if temp}_{32} \neq \text{temp}_{31} \text{ then} \\
\quad \text{SignalException(IntegerOverflow)} \\
\text{else} \\
\quad \text{GPR}[rd] \leftarrow \text{temp} \\
\text{endif}
\]

See 2.2 “Operation Section Notation and Functions” on page 20 for more information on the formal notation used here.

2.1.9 Exceptions Field

The Exceptions field lists the exceptions that can be caused by Operation of the instruction. It omits exceptions that can be caused by the instruction fetch, for instance, TLB Refill, and also omits exceptions that can be caused by asynchronous external events such as an Interrupt. Although a Bus Error exception may be caused by the operation of a load or store instruction, this section does not list Bus Error for load and store instructions because the relationship between load and store instructions and external error indications, like Bus Error, are dependent upon the implementation.

Figure 2.9 Example of Instruction Exception

Exceptions:

Integer Overflow

An instruction may cause implementation-dependent exceptions that are not present in the Exceptions section.

2.1.10 Programming Notes and Implementation Notes Fields

The Notes sections contain material that is useful for programmers and implementors, respectively, but that is not necessary to describe the instruction and does not belong in the description sections.

Figure 2.10 Example of Instruction Programming Notes

Programming Notes:

ADDU performs the same arithmetic operation but does not trap on overflow.

2.2 Operation Section Notation and Functions

In an instruction description, the Operation section uses a high-level language notation to describe the operation performed by each instruction. Special symbols used in the pseudocode are described in the previous chapter. Specific pseudocode functions are described below.

This section presents information about the following topics:

• “Instruction Execution Ordering” on page 21
2.2 Operation Section Notation and Functions

- “Pseudocode Functions” on page 21

2.2.1 Instruction Execution Ordering

Each of the high-level language statements in the Operations section are executed sequentially (except as constrained by conditional and loop constructs).

2.2.2 Pseudocode Functions

There are several functions used in the pseudocode descriptions. These are used either to make the pseudocode more readable, to abstract implementation-specific behavior, or both. These functions are defined in this section, and include the following:

- “Coprocessor General Register Access Functions” on page 21
- “Memory Operation Functions” on page 23
- “Floating Point Functions” on page 26
- “Miscellaneous Functions” on page 30

2.2.2.1 Coprocessor General Register Access Functions

Defined coprocessors, except for CP0, have instructions to exchange words and doublewords 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. This behavior is abstracted into the functions described in this section.

2.2.2.1.1 COP_LW

The COP_LW function defines 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.

Figure 2.11 COP_LW Pseudocode Function

COP_LW (z, rt, memword)
  z: The coprocessor unit number
  rt: Coprocessor general register specifier
  memword: A 32-bit word value supplied to the coprocessor

  /* Coprocessor-dependent action */

  endfunction COP_LW

2.2.2.1.2 COP_LD

The COP_LD function defines 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.

Figure 2.12 COP_LD Pseudocode Function

COP_LD (z, rt, memdouble)
2.2.2.1.3 COP_SW

The COP_SW function 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 the low-order word in coprocessor general register \( rt \).

**Figure 2.13 COP_SW Pseudocode Function**

\[
dataword \leftarrow \text{COP_SW}(z, rt)
\]

\( z \): The coprocessor unit number
\( rt \): Coprocessor general register specifier
\( dataword \): 32-bit word value

/* Coprocessor-dependent action */

dunction COP_SW

2.2.2.1.4 COP_SD

The COP_SD function 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 low-order doubleword in coprocessor general register \( rt \).

**Figure 2.14 COP_SD Pseudocode Function**

\[
datadouble \leftarrow \text{COP_SD}(z, rt)
\]

\( z \): The coprocessor unit number
\( rt \): Coprocessor general register specifier
\( datadouble \): 64-bit doubleword value

/* Coprocessor-dependent action */

dunction COP_SD

2.2.2.1.5 CoprocessorOperation

The CoprocessorOperation function performs the specified Coprocessor operation.

**Figure 2.15 CoprocessorOperation Pseudocode Function**

\[
\text{CoprocessorOperation}(z, \text{cop}_\text{fun})
\]

/* \( z \): Coprocessor unit number */
/* \( \text{cop}_\text{fun} \): Coprocessor function from function field of instruction */
/* Transmit the \( \text{cop}_\text{fun} \) value to coprocessor \( z \) */

dunction CoprocessorOperation
2.2.2.2 Memory Operation Functions

Regardless of byte ordering (big- or little-endian), the address of a halfword, word, or doubleword is the smallest byte address of the bytes that form the object. For big-endian ordering this is the most-significant byte; for a little-endian ordering this is the least-significant byte.

In the Operation pseudocode for load and store operations, the following functions summarize the handling of virtual addresses and the access of physical memory. The size of the data item to be loaded or stored is passed in the Access-Length field. The valid constant names and values are shown in Table 2.1. The bytes within the addressed unit of memory (word for 32-bit processors or doubleword for 64-bit processors) that are used can be determined directly from the AccessLength and the two or three low-order bits of the address.

2.2.2.2.1 Misaligned Support

MIPS processors originally required all memory accesses to be naturally aligned. MSA (the MIPS SIMD Architecture) supported misaligned memory accesses for its 128 bit packed SIMD vector loads and stores, from its introduction in MIPS Release 5. Release 6 requires systems to provide support for misaligned memory accesses for all ordinary memory reference instructions: the system must provide a mechanism to complete a misaligned memory reference for this instruction, ranging from full execution in hardware to trap-and-emulate.

The pseudocode function MisalignedSupport encapsulates the version number check to determine if misalignment is supported for an ordinary memory access.

**Figure 2.16 MisalignedSupport Pseudocode Function**

```
predicate ← MisalignedSupport ()
return Config.AR = 2 // Architecture Revision 2 corresponds to MIPS Release 6.
end function
```

See Appendix B, “Misaligned Memory Accesses” on page 511 for a more detailed discussion of misalignment, including pseudocode functions for the actual misaligned memory access.

2.2.2.2.2 AddressTranslation

The AddressTranslation function translates a virtual address to a physical address and its cacheability and coherency attribute, 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 cacheability and coherency attribute \((CCA)\) 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 or fixed mapping MMU determines 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.

**Figure 2.17 AddressTranslation Pseudocode Function**

```
(pAddr, CCA) ← AddressTranslation (vAddr, IorD, LorS)
/* pAddr: physical address */
/* CCA: Cacheability&Coherency Attribute, the method used to access caches */
/* and memory and resolve the reference */
/* vAddr: virtual address */
/* IorD: Indicates whether access is for INSTRUCTION or DATA */
/* LorS: Indicates whether access is for LOAD or STORE */
```
2.2.2.2.3 LoadMemory

The LoadMemory function loads a value from memory.

This action uses cache and main memory as specified in both the Cacheability and Coherency Attribute (CCA) and the access (IorD) to find the contents of AccessLength memory bytes, starting at physical location pAddr. The data is returned in a fixed-width naturally aligned memory element (MemElem). The low-order 2 (or 3) bits of the address and the AccessLength indicate which of the bytes within MemElem need to be passed to the processor. If the memory access type of the reference is uncached, only the referenced bytes are read from memory and marked as valid within the memory element. If the access type is cached but 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, this block is the entire memory element.

**Figure 2.18 LoadMemory Pseudocode Function**

```
MemElem ← LoadMemory (CCA, AccessLength, pAddr, vAddr, IorD)
```

- **MemElem**: Data is returned in a fixed width with a natural alignment. The width is the same size as the CPU general-purpose register, respectively.
- **CCA**: Cacheability&CoherencyAttribute-method used to access caches and memory and resolve the reference.
- **AccessLength**: Length, in bytes, of access.
- **pAddr**: physical address.
- **vAddr**: virtual address.
- **IorD**: Indicates whether access is for Instructions or Data.

**endfunction LoadMemory**

2.2.2.2.4 StoreMemory

The StoreMemory function stores 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 Cacheability and Coherency Attribute (CCA). The MemElem contains the data for an aligned, fixed-width memory element (a word for 32-bit processors, a doubleword for 64-bit processors), though only the bytes that are actually stored to memory need be valid. The low-order two (or three) bits of pAddr and the AccessLength field indicate which of the bytes within the MemElem data should be stored; only these bytes in memory will actually be changed.

**Figure 2.19 StoreMemory Pseudocode Function**

```
StoreMemory (CCA, AccessLength, MemElem, pAddr, vAddr)
```

- **CCA**: Cacheability&Coherency Attribute, 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.
2.2 Operation Section Notation and Functions

/* The width is the same size as the CPU general */
/* purpose register, either 4 or 8 bytes, */
/* aligned on a 4- or 8-byte boundary. For a */
/* partial-memory-element store, only the bytes that will be*/
/* stored must be valid. */
/* pAddr: physical address */
/* vAddr: virtual address */
endfunction StoreMemory

2.2.2.2.5 Prefetch

The Prefetch function prefetches 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.

Figure 2.20 Prefetch Pseudocode Function

Prefetch (CCA, pAddr, vAddr, DATA, hint)

/* CCA: Cacheability&Coherency Attribute, 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 */
endfunction Prefetch

Table 2.1 lists the data access lengths and their labels for loads and stores.

Table 2.1 AccessLength Specifications for Loads/Stores

<table>
<thead>
<tr>
<th>AccessLength Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>

2.2.2.2.6 SyncOperation

The SyncOperation function orders loads and stores to synchronize shared memory.

This action makes the effects of the synchronizable loads and stores indicated by stype occur in the same order for all processors.
2.2.2.3 Floating Point Functions

The pseudocode shown below specifies how the unformatted contents loaded or moved to CP1 registers are interpreted to form a formatted value. If an FPR contains a value in some format, rather than unformatted contents from a load (uninterpreted), it is valid to interpret the value in that format (but not to interpret it in a different format).

2.2.2.3.1 ValueFPR

The ValueFPR function returns a formatted value from the floating point registers.

Figure 2.22 ValueFPR Pseudocode Function

```c
value ← ValueFPR(fpr, fmt)

/* value: The formatted value from the FPR */
/* fpr: The FPR number */
/* fmt: The format of the data, one of: */
/* S, D, W, L, PS, */
/* OB, QH, */
/* UNINTERPRETED_WORD, */
/* UNINTERPRETED_DOUBLEWORD */
/* The UNINTERPRETED values are used to indicate that the datatype */
/* is not known as, for example, in SWC1 and SDC1 */

case fmt of
    S, W, UNINTERPRETED_WORD:
        valueFPR ← FPR[fpr]
    D, UNINTERPRETED_DOUBLEWORD:
        if (FP32RegistersMode = 0)
            if (fpr3 ≠ 0) then
                valueFPR ← UNPREDICTABLE
            else
                valueFPR ← FPR[fpr+1]31..0 || FPR[fpr]31..0
            endif
        else
            valueFPR ← FPR[fpr]
        endif
    L, PS:
        if (FP32RegistersMode = 0) then
            valueFPR ← UNPREDICTABLE
        else
            valueFPR ← FPR[fpr]
        endif
```

2.2.2.3.2 ValueFPR Pseudocode Function
2.2 Operation Section Notation and Functions

DEFAULT:
valueFPR \leftarrow \text{UNPREDICTABLE}
endcase
endfunction ValueFPR

The pseudocode shown below specifies the way a binary encoding representing a formatted value is stored into CP1 registers by a computational or move operation. This binary representation is visible to store or move-from instructions. Once an FPR receives a value from the StoreFPR(), it is not valid to interpret the value with ValueFPR() in a different format.

2.2.2.3.2 StoreFPR

Figure 2.23 StoreFPR Pseudocode Function

StoreFPR (fpr, fmt, value)

/* fpr: The FPR number */
/* fmt: The format of the data, one of: */
/* S, D, W, L, PS, */
/* OB, QH, */
/* UNINTERPRETED_WORD, */
/* UNINTERPRETED_DOUBLEWORD */
/* value: The formatted value to be stored into the FPR */

/* The UNINTERPRETED values are used to indicate that the datatype */
/* is not known as, for example, in LWC1 and LDC1 */

begin function StoreFPR

case fmt of
S, W, UNINTERPRETED_WORD:
FPR[fpr] \leftarrow value

D, UNINTERPRETED_DOUBLEWORD:
if (FP32RegistersMode = 0)
if (fpr \neq 0) then
UNPREDICTABLE
else
FPR[fpr] \leftarrow \text{UNPREDICTABLE}^{32} \| \text{value}_{31..0}
FPR[fpr+1] \leftarrow \text{UNPREDICTABLE}^{32} \| \text{value}_{63..32}
endif
else
FPR[fpr] \leftarrow value
endif

L, PS:
if (FP32RegistersMode = 0) then
UNPREDICTABLE
else
FPR[fpr] \leftarrow value
endif

endcase
endfunction StoreFPR
2.2.2.3 CheckFPException

The pseudocode shown below checks for an enabled floating point exception and conditionally signals the exception.

Figure 2.24 CheckFPException Pseudocode Function

```c
CheckFPException()

/* A floating point exception is signaled if the E bit of the Cause field is a 1 */
/* (Unimplemented Operations have no enable) or if any bit in the Cause field */
/* and the corresponding bit in the Enable field are both 1 */

if ( (FCSR[17] = 1) or
     ((FCSR[16..12] and FCSR[11..7]) ≠ 0)) ) then
    SignalException(FloatingPointException)
endif
endfunction CheckFPException
```

2.2.2.3.4 FPConditionCode

The FPConditionCode function returns the value of a specific floating point condition code.

Figure 2.25 FPConditionCode Pseudocode Function

```c
tf ← FPConditionCode(cc)

/* tf: The value of the specified condition code */
/* cc: The Condition code number in the range 0..7 */

if cc = 0 then
    FPConditionCode ← FCSR[23]
else
    FPConditionCode ← FCSR[24+cc]
endif
endfunction FPConditionCode
```

2.2.2.3.5 SetFPConditionCode

The SetFPConditionCode function writes a new value to a specific floating point condition code.

Figure 2.26 SetFPConditionCode Pseudocode Function

```c
SetFPConditionCode(cc, tf)

if cc = 0 then
    FCSR ← FCSR[31..24] || tf || FCSR[22..0]
else
    FCSR ← FCSR[31..25+cc] || tf || FCSR[23+cc..0]
endif
endfunction SetFPConditionCode
```
2.2 Operation Section Notation and Functions

2.2.2.4 Pseudocode Functions Related to Sign and Zero Extension

2.2.2.4.1 Sign extension and zero extension in pseudocode

Much pseudocode uses a generic function `sign_extend` without specifying from what bit position the extension is done, when the intention is obvious. E.g. `sign_extend(immediate16)` or `sign_extend(disp9)`.

However, sometimes it is necessary to specify the bit position. For example, `sign_extend(temp31..0)` or the more complicated `(offset_{15})^{\text{GPRLEN}}-(16+2) \ || \ offset \ || \ 0^2`.

The explicit notation `sign_extend.nbits(val)` or `sign_extend(val,nbits)` is suggested as a simplification. They say to sign extend as if an nbits-sized signed integer. The width to be sign extended to is usually apparent by context, and is usually GPRLEN, 32 or 64 bits. The previous examples then become.

\[
\text{sign_extend}(\text{temp31..0}) = \text{sign_extend.32}(\text{temp})
\]

and

\[
(\text{offset}_{15})^{\text{GPRLEN}}-(16+2) \ || \ offset \ || \ 0^2 = \text{sign_extend.16}(\text{offset}) \ll 2
\]

Note that `sign_extend.N(value)` extends from bit position N-1, if the bits are numbered 0..N-1 as is typical.

The explicit notations `sign_extend.nbits(val)` or `sign_extend(val,nbits)` is used as a simplification. These notations say to sign extend as if an nbits-sized signed integer. The width to be sign extended to is usually apparent by context, and is usually GPRLEN, 32 or 64 bits.

Floating point may use notations such as `zero_extend.fmt` corresponding to the format of the FPU instruction. E.g. `zero_extend.S` and `zero_extend.D` are equivalent to `zero_extend.32` and `zero_extend.64`.

Existing pseudocode may use any of these, or other, notations. TBD: rewrite pseudocode.

2.2.2.4.2 memory_address

The pseudocode function `memory_address` performs mode-dependent address space wrapping for compatibility between MIPS32 and MIPS64. It is applied to all memory references. It may be specified explicitly in some places, particularly for new memory reference instructions, but it is also declared to apply implicitly to all memory refer-
ences as defined below. In addition, certain instructions that are used to calculate effective memory addresses but which are not themselves memory accesses specify memory_address explicitly in their pseudocode.

**Figure 2.28 memory_address Pseudocode Function**

```plaintext
function memory_address(ea)
    return ea
end function
```

On a 32-bit CPU, memory_address returns its 32-bit effective address argument unaffected.

In addition to the use of memory_address for all memory references (including load and store instructions, LL/SC), Release 6 extends this behavior to control transfers (branch and call instructions), and to the PC-relative address calculation instructions (ADDIUPC, AUIPC, ALUIPC). In newer instructions the function is explicit in the pseudocode.

Implicit address space wrapping for all instruction fetches is described by the following pseudocode fragment which should be considered part of instruction fetch:

**Figure 2.29 Instruction Fetch Implicit memory_address Wrapping**

```plaintext
PC ← memory_address( PC )
(instruction_data, length) ← instruction_fetch( PC )
/* decode and execute instruction */
```

Implicit address space wrapping for all data memory accesses is described by the following pseudocode, which is inserted at the top of the AddressTranslation pseudocode function:

**Figure 2.30 AddressTranslation implicit memory_address Wrapping**

```plaintext
(pAddr, CCA) ← AddressTranslation (vAddr, IorD, LorS)
vAddr ← memory_address(vAddr)
```

In addition to its use in instruction pseudocode,

### 2.2.2.5 Miscellaneous Functions

This section lists miscellaneous functions not covered in previous sections.

#### 2.2.2.5.1 SignalException

The SignalException function signals an exception condition.

This action results in an exception that aborts the instruction. The instruction operation pseudocode never sees a return from this function call.

**Figure 2.31 SignalException Pseudocode Function**

```plaintext
SignalException(Exception, argument)
/* Exception: The exception condition that exists. */
/* argument: A exception-dependent argument, if any */
endfunction SignalException
```
2.2.2.5.2 SignalDebugBreakpointException

The SignalDebugBreakpointException function signals a condition that causes entry into Debug Mode from non-Debug Mode.

This action results in an exception that aborts the instruction. The instruction operation pseudocode never sees a return from this function call.

**Figure 2.32 SignalDebugBreakpointException Pseudocode Function**

```plaintext
SignalDebugBreakpointException()
endfunction SignalDebugBreakpointException
```

2.2.2.5.3 SignalDebugModeBreakpointException

The SignalDebugModeBreakpointException function signals a condition that causes entry into Debug Mode from Debug Mode (i.e., an exception generated while already running in Debug Mode).

This action results in an exception that aborts the instruction. The instruction operation pseudocode never sees a return from this function call.

**Figure 2.33 SignalDebugModeBreakpointException Pseudocode Function**

```plaintext
SignalDebugModeBreakpointException()
endfunction SignalDebugModeBreakpointException
```

2.2.2.5.4 NullifyCurrentInstruction

The NullifyCurrentInstruction function nullifies the current instruction.

The instruction is aborted, inhibiting not only the functional effect of the instruction, but also inhibiting all exceptions detected during fetch, decode, or execution of the instruction in question. For branch-likely instructions, nullification kills the instruction in the delay slot of the branch likely instruction.

**Figure 2.34 NullifyCurrentInstruction PseudoCode Function**

```plaintext
NullifyCurrentInstruction()
endfunction NullifyCurrentInstruction
```

2.2.2.5.5 JumpDelaySlot

The JumpDelaySlot function is used in the pseudocode for the PC-relative instructions in the MIPS16e ASE. The function returns TRUE if the instruction at `vAddr` is executed in a jump delay slot. A jump delay slot always immediately follows a JR, JAL, JALR, or JALX instruction.

**Figure 2.35 JumpDelaySlot Pseudocode Function**

```plaintext
JumpDelaySlot(vAddr)
/* vAddr:Virtual address */
endfunction JumpDelaySlot
2.2.2.5.6 PolyMult

The PolyMult function multiplies two binary polynomial coefficients.

Figure 2.36 PolyMult Pseudocode Function

```
PolyMult(x, y)
    temp ← 0
    for i in 0 .. 31
        if xi = 1 then
            temp ← temp xor (y(31-i)..0 || 0i)
        endif
    endfor
    PolyMult ← temp
endfunction PolyMult
```

2.3 Op and Function Subfield Notation

In some instructions, the instruction subfields op and function can have constant 5- or 6-bit values. When reference is made to these instructions, uppercase mnemonics are used. For instance, in the floating point ADD instruction, op=COP1 and function=ADD. In other cases, a single field has both fixed and variable subfields, so the name contains both upper- and lowercase characters.

2.4 FPU Instructions

In the detailed description of each FPU instruction, all variable subfields in an instruction format (such as fs, ft, immediate, and so on) are shown in lowercase. The instruction name (such as ADD, SUB, and so on) is shown in uppercase.

For the sake of clarity, an alias is sometimes used for a variable subfield in the formats of specific instructions. For example, rs=base in the format for load and store instructions. Such an alias is always lowercase since it refers to a variable subfield.

Bit encodings for mnemonics are given in Volume I, in the chapters describing the CPU, FPU, MDMX, and MIPS16e instructions.

See “Op and Function Subfield Notation” on page 32 for a description of the op and function subfields.
The MIPS® DSP Application Specific Extension to the MIPS® Architecture

3.1 Base Architecture Requirements

The MIPS DSP Module requires the following base architecture support:

- MIPS32 Release 2 or MIPS64 Release 2 Architecture: The MIPS DSP Module requires a compliant implementation of the MIPS32 Release 2 or MIPS64 Release 2 Architecture.

The MIPS DSP Module Rev2 requires the following base architecture support:

- MIPS DSP Module
- MIPS32 Release 2 or MIPS64 Release 2 Architecture

3.2 Software Detection of the Module

Software may determine if the MIPS DSP Module is implemented by checking the state of the DSPP (DSP Present) bit, which is bit 10 in the Config3 CP0 register.

Software may determine if the MIPS DSP Module Rev2 is implemented by checking the state of the DSP2P (DSP Rev2 Present) bit, which is bit 11 in the Config3 CP0 register. Compliant MIPS DSP Module Rev2 implementations must set both DSPP and DSP2P bits.

An implementation supports MIPS DSP Module Rev3 if CP0 Config3_DSPP=1 and Config3_DSP2P=1 and ConfigAR>=2.

The DSPP and DSP2P bits are fixed by the hardware implementation and are read-only for software.

3.3 Compliance and Subsetting

There are no instruction subsets of the MIPS DSP Module—all DSP Module instructions and state must be implemented.

There are no instruction subsets of the MIPS DSP Module Rev2 — all DSP Module and DSP Module Rev2 instructions and state must be implemented.
3.4 Introduction to the MIPS® DSP Module

This document contains a complete specification of the MIPS® DSP Module to the MIPS® architecture. Statements about MIPS DSP Module include MIPS DSP Module Rev2, except where noted. The table entries in Chapter 4, “MIPS® DSP Module Instruction Summary” on page 50 contain notations which flag the Rev2 instructions; this information is also available in the per instruction pages. The extensions comprises new integer instructions and new state that includes new HI-LO accumulator pairs and a DSPControl register. The MIPS DSP Module can be included in either a MIPS32 or MIPS64 architecture implementation. The Module has been designed to benefit a wide range of DSP, multimedia, and DSP-like algorithms. The performance increase from these extensions can be used to integrate DSP-like functionality into MIPS cores used in a SOC (System on Chip), potentially reducing overall system cost. The Module includes many of the typical features found in other integer-based DSP extensions, for example, support for operations on fractional data types and register SIMD (Single Instruction Multiple Data) operations such as add, subtract, multiply, shift, etc. In addition, the extensions includes some key features that efficiently address specific problems often encountered in DSP applications. These include, for example, support for complex multiplication, variable bit insertion and extraction, and the implementation and use of virtual circular buffers.

This chapter contains a basic overview of the principles behind DSP application processing and the data types and structures needed to efficiently process such applications. Chapter 4, “MIPS® DSP Module Instruction Summary” on page 50, contains a list of all the instructions in the MIPS DSP Module arranged by function type. Chapter 5, “Instruction Encoding” on page 70, describes the position of the new instructions in the MIPS instruction opcode map. The rest of the specification contains a complete list of all the instructions that comprise the MIPS DSP Module, and serves as a quick reference guide to all the instructions. Finally, various Appendix chapters describe how to implement and use the DSP Module instructions in some common algorithms and inner loops.

3.5 DSP Applications and their Requirements

The MIPS DSP Module has been designed specifically to improve the performance of a set of DSP and DSP-like applications. Table 3.1 shows these application areas sorted by the size of the data operands typically preferred by that application for internal computations. For example, raw audio data is usually signed 16-bit, but 32-bit internal calculations are often necessary for high quality audio. (Typically, an internal precision of about 28 bits may be all that is required which can be achieved using a fractional data type of the appropriate width.) There is some cross-over in some cases, which are not explicitly listed here. For example, some hand-held consumer devices may use lower precision internal arithmetic for audio processing, that is, 16-bit internal data formats may be sufficient for the quality required for hand-held devices.

<table>
<thead>
<tr>
<th>In/Out Data Size</th>
<th>Internal Data Size</th>
<th>Applications</th>
</tr>
</thead>
</table>
| 8 bits           | 8/16 bits          | • Printer image processing.  
                      • Still JPEG processing.  
                      • Moving video processing |
| 16 bits          | 16 bits            | • Voice Processing. For example, G.723.1, G.729, G.726, echo cancellation, noise cancellation, channel equalization, etc.  
                      • Soft modem processing. For example V.92.  
                      • General DSP processing. For example, filters, correlation, convolution, etc. |
| 16/24 bits       | 32 bits            | • Audio decoding and encoding. For example, MP3, AAC, SRS TruSurround, Dolby Digital Decoder, Pro Logic II, etc. |
3.6 Fixed-Point Data Types

Typical implementations of DSP algorithms use fractional fixed-point arithmetic, for reasons of size, cost, and power efficiency. Unlike floating-point arithmetic, fractional fixed-point arithmetic assumes that the position of the decimal point is fixed with respect to the bits representing the fractional value in the operand. To understand this type of arithmetic further, please consult DSP textbooks or other references that are easily available on the internet.

Fractional fixed-point data types are often referred to using Q format notation. The general form for this notation is \( \text{Q}_m.n \), where \( Q \) designates that the data is in fractional fixed-point format, \( m \) is the number of bits used to designate the two's complement integer portion of the number, and \( n \) is the number of bits used to designate the two's complement fractional part of the number. Because the two's complement number is signed, the number of bits required to express a number is \( m+n+1 \), where the additional bit is required to denote the sign. In typical usage, it is very common for \( m \) to be zero. That is, only fractional bits are represented. In this case, a Q notation of the form \( \text{Q}_0.n \) is abbreviated to \( \text{Q}_n \).

For example, a 32-bit word can be used to represent data in Q31 format, which implies one (left-most) sign bit followed by the binary point and then 31 bits representing the fractional data value. The interpretation of the 32 bits of the Q31 representation is shown in Table 3.2. Negative values are represented using the two-complement of the equivalent positive value. This format can represent numbers in the range of \(-1.0\) to \(+0.999999999...\). Similarly a 16-bit halfword can be used to represent data in Q15 format, which implies one sign bit followed by 15 fractional bits that represent a value between \(-1.0\) and \(+0.9999...\).

Table 3.2 The Value of a Fixed-Point Q31 Number

|---------|------|------|------|------|------|------|------|------|------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|

Table 3.3 shows the limits of the Q15 and the Q31 representations. Note that the value \(-1.0\) can be represented exactly, but the value \(+1.0\) cannot. For practical purposes, \(0x7FFFFFFF\) is used to represent \(1.0\) inexactly. Thus, the multiplication of two values where both are \(-1\) will result in an overflow since there is no representation for \(+1\) in fixed-point format. Saturating instructions must check for this case and prevent the overflow by clamping the result to the maximal representable value. Instructions in the MIPS DSP Module that operate on fractional data types include a “Q” in the instruction mnemonic; the assumed size of the instruction operands is detailed in the instruction description.

Table 3.3 The Limits of Q15 and Q31 Representations

<table>
<thead>
<tr>
<th>Fixed-Point Representation</th>
<th>Definition</th>
<th>Hexadecimal Representation</th>
<th>Decimal Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q15 minimum</td>
<td>(-2^{15}/2^{15})</td>
<td>0x8000</td>
<td>(-1.0)</td>
</tr>
<tr>
<td>Q15 maximum</td>
<td>((2^{15}-1)/2^{15})</td>
<td>0x7FFF</td>
<td>0.999969482421875</td>
</tr>
<tr>
<td>Q31 minimum</td>
<td>(-2^{31}/2^{31})</td>
<td>0x80000000</td>
<td>(-1.0)</td>
</tr>
<tr>
<td>Q31 maximum</td>
<td>((2^{31}-1)/2^{31})</td>
<td>0x7FFFFFFF</td>
<td>0.9999999995343387126922607421875</td>
</tr>
</tbody>
</table>

Given a fixed-point representation, we can compute the corresponding decimal value by using bit weights per position as shown in Figure 3.1 for a hypothetical Q7 format number representation with 8 total bits.

DSP applications often, but not always, prefer to saturate the result after an arithmetic operation that causes an overflow or underflow. For operations on signed values, saturation clamps the result to the smallest negative or largest
positive value in the case of underflow and overflow, respectively. For operations on unsigned values, saturation clamps the result to either zero or the maximum positive value.

**Figure 3.1 Computing the Value of a Fixed-Point (Q7) Number**

<table>
<thead>
<tr>
<th>bit weights</th>
<th>-2^0</th>
<th>2^-1</th>
<th>2^-2</th>
<th>2^-3</th>
<th>2^-4</th>
<th>2^-5</th>
<th>2^-6</th>
<th>2^-7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example binary value</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>decimal value is</td>
<td>2^-1 + 2^-2 + 2^-5</td>
<td>= 0.5 + 0.25 + 0.03125</td>
<td>= 0.78125</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Example binary value | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| decimal value is | 2^-2 + 2^-3 | = 0.25 + 0.125 | = 0.375 |

| Example binary value | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| maximum positive value | decimal value is | 2^-1 + 2^-2 + 2^-3 + 2^-4 + 2^-5 + 2^-6 + 2^-7 | = 0.5 + 0.25 + 0.125 + 0.0625 + 0.03125 + 0.015625 + 0.0078125 | = 0.99218 |

| Example binary value | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| maximum negative value | decimal value is | -2^0 + 2^-2 + 2^-4 | = -1.0 + 0.25 + 0.0625 | = -0.6875 |

| Example binary value | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

### 3.7 Saturating Math

Many of the MIPS DSP Module arithmetic instructions provide optional saturation of the results, as detailed in each instructions description.

Saturation of fixed-point addition, subtraction, or shift operations that result in an underflow or overflow requires clamping the result value to the closest available fixed-point value representable in the given number of result bits. For operations on unsigned values, underflow is clamped to zero, and overflow to the largest positive fixed-point value. For operations on signed values, underflow is clamped to the minimum negative fixed-point value and overflow to the maximum positive value.
Saturation of fractional fixed-point multiplication operations clamps the result to the maximum representable
fixed-point value when both input multiplicands are equal to the minimum negative value of -1.0, which is independ-
ent of the Q format used.

3.8 Conventions Used in the Instruction Mnemonics

MIPS DSP Module instructions with a Q in the mnemonic assume the input operands to be in fractional fixed-point
format. Multiplication instructions that operate on fractional fixed-point data will not produce correct results when
used with integer fixed-point data. However, addition and subtraction instructions will work correctly with either
fractional fixed-point or signed integer fixed-point data.

Instructions that use unsigned data are indicated with the letter U. This letter appears after the letter Q for fractional in
the instruction mnemonic. For example, the ADDQU instruction performs an unsigned addition of fractional data. In
the MIPS base instruction set, the overflow trap distinguishes signed and unsigned arithmetic instructions. In the
MIPS DSP Module, the results of saturation distinguish signed and unsigned arithmetic instructions.

Some instructions provide optional rounding up, saturation, or rounding up and saturation of the result(s). These
instructions use one of the modifiers _RS, _R, _S, or _SA in their mnemonic. For example, MULQ_RS is a multiply
instruction (MUL) where the result is the same size as the input operands (indicated by the absence of E for expanded
result in the mnemonic) that assumes fractional (Q) input data operands, and where the result is rounded up and satu-
rated (_RS) before writing the result in the destination register. (For fractional multiplication, saturation clamps the
result to the maximum positive representable value if both multiplicands are equal to -1.0.) Several multiply-accumulate (dot product) instructions use a variant of the saturation flag, _SA, indicating that the accumulated value is satu-
rated in addition to the regular fractional multiplication saturation check.

The MIPS DSP Module instructions provide support for single-instruction, multiple data (SIMD) operations where a
single instruction can invoke multiple operation on multiple data operands. As noted previously, DSP applications
typically use data types that are 8, 16, or 32 bits wide. In the MIPS32 architecture a general-purpose register (GPR) is
32 bits wide, and in the MIPS64 architecture, 64 bits wide. Thus, each GPR can be used to hold one or more operands
of each size. For example, a 64-bit GPR can store eight 8-bit operands, a 32-bit GPR can store two 16-bit operands,
and so on. A GPR containing multiple data operands is referred to as a vector.

MIPS implementations of the MIPS DSP Module support three basic formats for data operands: 32 bit, 16 bit, and 8
bit. The latter format is motivated by the fact that video applications typically operate on 8-bit data. The instruction
mnemonics indicate the supported data types as follows:

- W = “Word”, 1 × 32-bit
- PH = “Paired Halfword”, 2 × 16-bit. See Figure 3.2.
- QB = “Quad Byte”, 4 × 8-bit. See Figure 3.3.

Figure 3.2 A Paired-Half (PH) Representation in a GPR for the MIPS32 Architecture

<table>
<thead>
<tr>
<th>16</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>halfword</td>
<td>halfword</td>
</tr>
<tr>
<td>32 bits</td>
<td></td>
</tr>
</tbody>
</table>
Figure 3.3 A Quad-Byte (QB) Representation in a GPR for the MIPS32 Architecture

For example, `MULQ_RS.PH rd, rs, rt` refers to the multiply instruction (MUL) that multiplies two vector elements of type fractional (Q) 16 bit (Halfword) data (PH) with rounding and saturation (RS). Each source register supplies two data elements and the two results are written into the destination register in the corresponding vector position as shown in Figure 3.4.

When an instruction shows two format types, then the first is the output size and the second is the input size. For example, `PRECRQ.PH.W` is the (fractional) precision reduction instruction that creates a PH output format and uses W format as input from the two source registers. When the instruction only shows one format then this implies the same source and destination format.

Figure 3.4 Operation of `MULQ_RS.PH rd, rs, rt`

3.9 Effect of Endian-ness on Register SIMD Data

The order of data in memory and therefore in the register has a direct impact on the algorithm being executed. To reduce the effort required by the programmer and the development tools to take endian-ness into account, many of the instructions operate on pre-defined bits of a given register. The assembler can be used to map the endian-agnostic names to the actual instructions based on the endian-ness of the processor during the compilation and assembling of the instructions.

When a SIMD vector is loaded into a register or stored back to memory from a register, the endian-ness of the processor and memory has an impact on the view of the data. For example, consider a vector of eight byte values aligned in memory on a 64-bit boundary and loaded into a 64-bit register using the load double instruction: the order of the eight byte values within the register depends on the processor endian-ness. In a big-endian processor, the byte value stored
at the lowest memory address is loaded into the left-most (most-significant) 8 bits of the 64-bit register. In a little-endian processor, the same byte value is loaded into the right-most (least-significant) 8 bits of the register.

In general, if the byte elements are numbered 0-7 according to their order in memory, in a big-endian configuration, element 0 is at the most-significant end and element 7 is at the least-significant end. In a little-endian configuration, the order is reversed. This effect applies to all the sizes of data when they are in SIMD format.

To avoid dealing with the endian-ness issue directly, the instructions in the DSP Module simply refer to the left and right elements of the register when it is required to specify a subset of the elements. This issue can quite easily be dealt with in the assembler or user code using suitably defined mnemonics that use the appropriate instruction for a given endian-ness of the processor. A description of how to do this is specified in Appendix 7.

### 3.10 Additional Register State for the DSP Module

The MIPS DSP Module adds four new registers. The operating system is required to recognize the presence of the MIPS DSP Module and to include these additional registers in context save and restore operations.

- Three additional HI-LO registers to create a total of four accumulator registers. Many common DSP computations involve accumulation, e.g., convolution. MIPS DSP Module instructions that target the accumulators use two bits to specify the destination accumulator, with the zero value referring to the original accumulator of the MIPS architecture.

Release 6 of the MIPS Architecture moves the accumulators into the DSP Module for use as a DSP resource exclusively.

- A new control register, `$DSPControl$`, is used to hold extra state bits needed for efficient support of the new instructions. Figure 3.5 illustrates the bits in this register. Table 3.4 describes the use of the various bits and the instructions that refer to the fields. Table 3.5 lists the instructions that affect the `$DSPControl$` register `ouflag` field.

#### Figure 3.5 MIPS® DSP Module Control Register (DSPControl) Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ccond</code></td>
<td>27:24</td>
</tr>
<tr>
<td><code>ouflag</code></td>
<td>23:16</td>
</tr>
</tbody>
</table>

#### Table 3.4 MIPS® DSP Module Control Register (DSPControl) Field Descriptions

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
<th>Read / Write</th>
<th>Reset State</th>
<th>Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ccond</code></td>
<td>27:24</td>
<td>Condition code bits set by vector comparison instructions and used as source selectors by PICK instructions. The vector element size determines the number of bits set by a comparison (1, 2, or 4); bits not set are UNPREDICTABLE after the comparison.</td>
<td>R/W</td>
<td>0</td>
<td>Required</td>
</tr>
<tr>
<td><code>ouflag</code></td>
<td>23:16</td>
<td>Overflow/underflow indication bits set when the result(s) of specific instructions (listed in Table 3.5) caused, or, if optional saturation has been used, would have caused overflow or underflow.</td>
<td>R/W</td>
<td>0</td>
<td>Required</td>
</tr>
</tbody>
</table>
The bits of the overflow flag (ouflag) field in the DSPControl register are set by a number of instructions. These bits are sticky and can be reset only by an explicit write to these bits in the register (using the WRDSP instruction). The table below shows which bits can be set by which instructions and under what conditions.

### Table 3.4 MIPS® DSP Module Control Register (DSPControl) Field Descriptions

<table>
<thead>
<tr>
<th>Fields</th>
<th>Description</th>
<th>Read / Write</th>
<th>Reset State</th>
<th>Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI</td>
<td>Extract Fail Indicator. This bit is set to 1 when one of the extraction instructions (EXTP, EXTPV, EXTPDP, or EXTPDP) fails. Failure occurs when there are insufficient bits to extract, i.e., when the value of the pos field in the DSPControl register is less than the size argument specified in the instruction. This bit is not sticky—the bit is set or reset after each extraction operation.</td>
<td>R/W</td>
<td>0</td>
<td>Required</td>
</tr>
<tr>
<td>c</td>
<td>Carry bit set and used by a special add instruction used to implement a 64-bit addition across two GPRs in a MIPS32 implementation. Instruction ADDSC sets the bit and instruction ADDWC uses this bit.</td>
<td>R/W</td>
<td>0</td>
<td>Required</td>
</tr>
<tr>
<td>scount</td>
<td>This field is used by the INSV instruction to specify the size of the bit field to be inserted.</td>
<td>R/W</td>
<td>0</td>
<td>Required</td>
</tr>
<tr>
<td>pos</td>
<td>This field is used by the variable insert instruction INSV to specify the position to insert bits. It is also used to indicate the extract position for the EXTP, EXTPV, EXTPDP, and EXTPDV instructions. The decrement pos (DP) variants of these instructions decrement the value of the pos field by the amount size+1 after the extraction completes successfully. The MTHLIP instruction increments the value of pos by 32 after copying the value of LO to HI.</td>
<td>R/W</td>
<td>0</td>
<td>Required</td>
</tr>
</tbody>
</table>

The bits of the overflow flag (ouflag) field in the DSPControl register are set by a number of instructions. These bits are sticky and can be reset only by an explicit write to these bits in the register (using the WRDSP instruction). The table below shows which bits can be set by which instructions and under what conditions.

### Table 3.5 Instructions that set the ouflag bits in DSPControl

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Instructions That Set This Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Instructions that set this bit when the destination is accumulator (HI-LO pair) zero and an operation overflow or underflow occurs: DPAQ_S, DPAQ_SA, DPSQ_S, DPSQ_SA, MAQ_S, MAQ_SA, and MULSAQ_S, DPAQX_S, DPAQX_SA, DPSQX_S, DPSQX_SA.</td>
</tr>
<tr>
<td>17</td>
<td>Instructions as above, when the destination is accumulator (HI-LO pair) one.</td>
</tr>
<tr>
<td>18</td>
<td>Instructions as above, when the destination is accumulator (HI-LO pair) two.</td>
</tr>
<tr>
<td>19</td>
<td>Instructions as above, when the destination is accumulator (HI-LO pair) three.</td>
</tr>
<tr>
<td>20</td>
<td>Instructions that on an overflow/underflow will set this bit are: ABSQ_S, ADD, ADD_S, ADDQ, ADDQ_S, ADDU, ADDU_S, ADDWC, SUB, SUB_S, SUBQ, SUBQ_S, SUBU, and SUBU_S.</td>
</tr>
</tbody>
</table>
3.11 Software Detection of the DSP Module

Bit 10 in the config3 CP0 register, “DSP Present” (DSPP), is used to indicate the presence of the MIPS DSP Module, and bit 11, “DSP Rev2 Present,” (DSP2P), the presence of the MIPS DSP Module Rev2, as shown in Figure 3.6. Valid MIPS DSP Module Rev2 implementations set both DSPP and DSP2P bits: the condition of DSP2P set and DSPP unset is invalid. Software may read the DSPP, DSP2P bits of the config3 CP0 register to check whether this processor has implemented the MIPS DSP Module and MIPS DSP Module Rev2.

Release 6 of the MIPS Architecture moves the accumulators into the DSP Module for use as a DSP resource exclusively, and introduces the compact branch BPOSGE32C, for which DSP Module Rev3 is required. An implementation supports Rev3 if CP0 Config3DSPP = 1 and Config3DSP2P = 1 and ConfigAR >= 2.

Any attempt to execute MIPS DSP Module instructions must cause a Reserved Instruction Exception if DSPP, and DSP2P are not indicating the presence of the appropriate MIPS DSP Module implementation. The DSPP and DSP2P bits are fixed by the hardware implementation and are read-only for software.

### Table 3.5 Instructions that set the ouflag bits in DSPControl

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Instructions That Set This Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>Instructions that on an overflow/underflow will set this bit are: MUL, MUL_S, MULEQ_S, MULEU_S, MULQ_RS, and MULQ_S.</td>
</tr>
<tr>
<td>22</td>
<td>Instructions that on an overflow/underflow will set this bit are: PRECRQ_RS, PRECRQU_RS, SHLL, SHLL_S, SHLLV, and SHLLV_S.</td>
</tr>
<tr>
<td>23</td>
<td>Instructions that on an overflow/underflow will set this bit are: EXTR, EXTR_S, EXTR_RS, EXTRV, EXTRV_RS</td>
</tr>
</tbody>
</table>

### Figure 3.6 Config3 Register Format

The “DSP Module Enable” (DSPEn) bit—the MX bit, bit 24 in the CP0 Status register as shown in Figure 3.7—is used to enable access to the extra instructions defined by the MIPS DSP Module as well as enabling four modified move instructions (MTLO/HI and MFLO/HI) that provide access to the three additional accumulators ac1, ac2, and ac3. Executing a MIPS DSP Module instruction or one of the four modified move instructions when DSPEn is set to zero causes a DSP State Disabled Exception and results in exception code 26 in the CP0 Cause register. This allows the OS to do lazy context-switching. Table 3.6 shows the Cause Register exception code fields.

### Figure 3.7 CP0 Status Register Format

### Table 3.6 Cause Register ExcCode Field

<table>
<thead>
<tr>
<th>Exception Code Value</th>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>16#1a</td>
<td></td>
<td>DSPDis</td>
<td>DSP Module State Disabled Exception</td>
</tr>
</tbody>
</table>
3.12 Exception Table for the DSP Module

Table 3.7 shows the exceptions caused when a MIPS DSP Module or MIPS DSP Module Rev2 instruction, MTLO/HI or MFLO/HI, or any other instruction such as an CorExtend instruction attempts to access the new DSP Module state, that is, \( ac1 \), \( ac2 \), or \( ac3 \), or the \( DSPControl \) register, and all other possible exceptions that relate to the DSP Module.

### Table 3.7 Exception Table for the DSP Module

<table>
<thead>
<tr>
<th>( Config_{DSP2P} )</th>
<th>( Config_{DSPP} )</th>
<th>( Status_{MX} )</th>
<th>Exception for DSP Module Rev2 Instructions</th>
<th>Exception for DSP Module Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>( x )</td>
<td>Reserved Instruction</td>
<td>Reserved Instruction</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Reserved Instruction</td>
<td>DSP Module State Disabled</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Reserved Instruction</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>DSP Module State Disabled</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>DSP Module State Disabled</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

3.13 DSP Module Instructions that Read and Write the DSPControl Register

Many MIPS DSP Module instructions read and write the \( DSPControl \) register, some explicitly and some implicitly. Like other register resource in the architecture, it is the responsibility of the hardware implementation to ensure that appropriate execution dependency barriers are inserted and the pipeline stalled for read-after-write dependencies and
other data dependencies that may occur. Table 3.8 lists the MIPS DSP Module instructions that can read and write the \textsl{DSPControl} register and the bits or fields in the register that they read or write.

### Table 3.8 Instructions that Read/Write Fields in DSPControl

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Read/Write</th>
<th>DSPControl Field (Bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRDSP</td>
<td>W</td>
<td>All (31:0)</td>
</tr>
<tr>
<td>EXTPDP, EXTPDPV, MTHLP</td>
<td>W</td>
<td>pos (5:0)</td>
</tr>
<tr>
<td>ADDSC</td>
<td>W</td>
<td>c (13)</td>
</tr>
<tr>
<td>EXTP, EXTPV, EXTPDP, EXTPDPV</td>
<td>W</td>
<td>EFI (14)</td>
</tr>
<tr>
<td>See Table 3.5</td>
<td>W</td>
<td>ouflag (23:16)</td>
</tr>
<tr>
<td>CMP, CMPU, and CMPGDU variants</td>
<td>W</td>
<td>ccond (27:24)</td>
</tr>
<tr>
<td>RDDSP</td>
<td>R</td>
<td>All (31:0)</td>
</tr>
<tr>
<td>BPOSGE32, BPOSGE32C, EXTP, EXTPV, EXTPDP, EXTPDPV, INSV</td>
<td>R</td>
<td>pos (5:0)</td>
</tr>
<tr>
<td>INSV</td>
<td>R</td>
<td>scount (12:7)</td>
</tr>
<tr>
<td>ADDWC</td>
<td>R</td>
<td>c (13)</td>
</tr>
<tr>
<td>PICK variants</td>
<td>R</td>
<td>ccond (27:24)</td>
</tr>
</tbody>
</table>

### 3.14 Arithmetic Exceptions

Under no circumstances do any of the MIPS DSP Module instructions cause an arithmetic exception. Other exceptions are possible, for example, the indexed load instruction can cause an address exception. The specific exceptions caused by the different instructions are listed in the per-instruction description pages.
Chapter 4

MIPS® DSP Module Instruction Summary

4.1 The MIPS® DSP Module Instruction Summary

The tables in this chapter list all the instructions in the DSP Module. For operation details about each instruction, refer to the per-page descriptions. In each table, the column entitled “Writes GPR / ac / DSPControl”, indicates the explicit write performed by each instruction. This column indicates the writing of a field in the DSPControl register other than the ouflag field (which is written by a large number of instructions as a side-effect).

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDQ.PH rd,rs,rt</td>
<td>Pair Q15</td>
<td>Pair Q15</td>
<td>GPR</td>
<td>VoIP SoftM</td>
<td>Element-wise addition of two vectors of Q15 fractional values, with optional saturation.</td>
</tr>
<tr>
<td>ADDQ_S.PH rd,rs,rt</td>
<td>Q31</td>
<td>Q31</td>
<td>GPR</td>
<td>Audio</td>
<td>Add two Q31 fractional values with saturation.</td>
</tr>
<tr>
<td>ADDU.QB rd,rs,rt</td>
<td>Quad Unsigned Byte</td>
<td>Quad Unsigned Byte</td>
<td>GPR</td>
<td>Video</td>
<td>Element-wise addition of unsigned byte values, with optional unsigned saturation.</td>
</tr>
<tr>
<td>ADDU_QB rd,rs,rt</td>
<td>Quad Unsigned Byte</td>
<td>Quad Unsigned Byte</td>
<td>GPR</td>
<td>Video</td>
<td>Element-wise addition of vectors of four unsigned byte values, halving each result by right-shifting by one bit position. Results may be optionally rounded up in the least-significant bit.</td>
</tr>
<tr>
<td>ADDUH.QB rd,rs,rt</td>
<td>Quad Unsigned Byte</td>
<td>Quad Unsigned Byte</td>
<td>GPR</td>
<td>Video</td>
<td>Element-wise addition of vectors of two unsigned halfword values, with optional saturation on overflow.</td>
</tr>
<tr>
<td>ADDUH_RB rd,rs,rt</td>
<td>Signed Halfword</td>
<td>Signed Halfword</td>
<td>GPR</td>
<td>Misc</td>
<td>Element-wise addition of vectors of two signed halfword values, halving each result with right-shifting by one bit position. Results may be optionally rounded up in the least-significant bit.</td>
</tr>
<tr>
<td>ADDQUH.W rd,rs,rt</td>
<td>Signed Word</td>
<td>Signed Word</td>
<td>GPR</td>
<td>Misc</td>
<td>Add two signed word values, halving the result with right-shifting by one bit position. Result may be optionally rounded up in the least-significant bit.</td>
</tr>
<tr>
<td>SUBQ.PH rd,rs,rt</td>
<td>Pair Q15</td>
<td>Pair Q15</td>
<td>GPR</td>
<td>VoIP</td>
<td>Element-wise subtraction of two vectors of Q15 fractional values, with optional saturation.</td>
</tr>
<tr>
<td>SUBQ_S.PH rd,rs,rt</td>
<td>Q31</td>
<td>Q31</td>
<td>GPR</td>
<td>Audio</td>
<td>Subtraction with Q31 fractional values, with saturation.</td>
</tr>
</tbody>
</table>
### Table 4.1 List of Instructions in MIPS® DSP Module in Arithmetic Sub-class (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBU.QB rd,rs,rt SUBU.S.QB rd,rs,rt</td>
<td>Quad Unsigned Byte</td>
<td>Quad Unsigned Byte</td>
<td>GPR</td>
<td>Video</td>
<td>Element-wise subtraction of unsigned byte values, with optional unsigned saturation.</td>
</tr>
<tr>
<td>SUBUH.QB rd,rs,rt SUBUH_R.QB rd,rs,rt MIPSDSP-R2 Only</td>
<td>Quad Unsigned Byte</td>
<td>Quad Unsigned Byte</td>
<td>GPR</td>
<td>Video</td>
<td>Element-wise subtraction of unsigned byte values, shifting the results right one bit position (halving). The results may be optionally rounded up by adding 1 to each result at the most-significant discarded bit position before shifting.</td>
</tr>
<tr>
<td>SUBUH.PH rd,rs,rt SUBUH_S.PH rd,rs,rt</td>
<td>Pair Unsigned Halfword</td>
<td>Pair Unsigned Halfword</td>
<td>GPR</td>
<td>Video</td>
<td>Element-wise subtraction of vectors of two unsigned halfword values, with optional saturation on overflow.</td>
</tr>
<tr>
<td>SUBQH.PH rd,rs,rt SUBQH_R.PH rd,rs,rt MIPSDSP-R2 Only</td>
<td>Pair Signed Halfword</td>
<td>Pair Signed Halfword</td>
<td>GPR</td>
<td>Misc</td>
<td>Element-wise subtraction of vectors of two signed halfword values, halving each result with right-shifting by one bit position. Results may be optionally rounded up in the least-significant bit.</td>
</tr>
<tr>
<td>SUBQH.W rd,rs,rt SUBQH_R.W rd,rs,rt MIPSDSP-R2 Only</td>
<td>Signed Word</td>
<td>Signed Word</td>
<td>GPR</td>
<td>Misc</td>
<td>Subtract two signed word values, halving the result with right-shifting by one bit position. Result may be optionally rounded up in the least-significant bit.</td>
</tr>
<tr>
<td>ADDSC rd,rs,rt</td>
<td>Signed Word</td>
<td>Signed Word</td>
<td>GPR &amp; DSPControl</td>
<td>Audio</td>
<td>Add two signed words and set the carry bit in the DSPControl register.</td>
</tr>
<tr>
<td>ADDWC rd,rs,rt</td>
<td>Signed Word</td>
<td>Signed Word</td>
<td>GPR</td>
<td>Audio</td>
<td>Add two signed words with the carry bit from the DSPControl register.</td>
</tr>
<tr>
<td>MODSUB rd,rs,rt</td>
<td>Signed Word</td>
<td>Signed Word</td>
<td>GPR</td>
<td>Misc</td>
<td>Modulo addressing support: update a byte index into a circular buffer by subtracting a specified decrement (in bytes) from the index, resetting the index to a specified value if the subtraction results in underflow.</td>
</tr>
<tr>
<td>RADDU.W.QB rd,rs</td>
<td>Quad Unsigned Byte</td>
<td>Unsigned Word</td>
<td>GPR</td>
<td>Misc</td>
<td>Reduce (add together) the 4 unsigned byte values in rs, zero-extending the sum to 32 bits before writing to the destination register. For example, if all 4 input values are 0x80 (decimal 128), then the result in rd is 0x200 (decimal 512).</td>
</tr>
<tr>
<td>ABSQ_S.QB rd,rt MIPSDSP-R2 Only</td>
<td>Quad Q7</td>
<td>Quad Q7</td>
<td>GPR</td>
<td>Misc</td>
<td>Find the absolute value of each of four Q7 fractional byte elements in the source register, saturating values of -1.0 to the maximum positive Q7 fractional value.</td>
</tr>
<tr>
<td>ABSQ_S.PH rd,rt</td>
<td>Pair Q15</td>
<td>Pair Q15</td>
<td>GPR</td>
<td>Misc</td>
<td>Find the absolute value of each of two Q15 fractional halfword elements in the source register, saturating values of -1.0 to the maximum positive Q15 fractional value.</td>
</tr>
<tr>
<td>Instruction Mnemonics</td>
<td>Input Data Type</td>
<td>Output Data Type</td>
<td>Writes GPR / ac / DSPControl</td>
<td>App</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------</td>
<td>-----------------</td>
<td>------------------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ABSQ_S.W rd,rt</td>
<td>Q31</td>
<td>Q31</td>
<td>GPR</td>
<td>Misc</td>
<td>Find the absolute value of the Q31 fractional element in the source register, saturating the value -1.0 to the maximum positive Q31 fractional value.</td>
</tr>
<tr>
<td>PRECR.QB.PH rd,rs,rt</td>
<td>Two Pair Integer Halfwords</td>
<td>Four Integer Bytes</td>
<td>GPR</td>
<td>Misc</td>
<td>Reduce the precision of four signed integer halfword input values by discarding the eight most-significant bits from each to create four signed integer byte output values. The two halfword values from register rs are used to create the two left-most byte results, allowing an endian-agnostic implementation.</td>
</tr>
<tr>
<td>PRECRQ.QB.PH rd,rs,rt</td>
<td>2 Pair Q15</td>
<td>Quad Byte</td>
<td>GPR</td>
<td>Misc</td>
<td>Reduce the precision of four Q15 fractional input values by truncation to create four Q7 fractional output values. The two Q15 values from register rs are written to the two left-most byte results, allowing an endian-agnostic implementation.</td>
</tr>
<tr>
<td>PRECR_SRA.PH.W rt,rs,sa</td>
<td>Two Integer Words</td>
<td>Pair Integer Halfword</td>
<td>GPR</td>
<td>Misc</td>
<td>Reduce the precision of two integer word values to create a pair of integer halfword values. Each word value is first shifted right arithmetically by sa bit positions, and optionally rounded up by adding 1 at the most-significant discard bit position. The 16 least-significant bits of each word are then written to the corresponding halfword elements of destination register rt.</td>
</tr>
<tr>
<td>PRECRQ.PH.W rd,rs,rt</td>
<td>2 Q31</td>
<td>Pair half-word</td>
<td>GPR</td>
<td>Misc</td>
<td>Reduce the precision of two Q31 fractional input values by truncation to create two Q15 fractional output values. The Q15 value obtained from register rs creates the left-most result, allowing an endian-agnostic implementation. Results may be optionally rounded up and saturated before being written to the destination.</td>
</tr>
<tr>
<td>PRECRQU_S.QB.PH rd,rs,rt</td>
<td>2 Pair Q15</td>
<td>Quad Unsigned Byte</td>
<td>GPR</td>
<td>Misc</td>
<td>Reduce the precision of four Q15 fractional values by saturating and truncating to create four unsigned byte values.</td>
</tr>
<tr>
<td>PRECEQ.W.PHL rd,rt</td>
<td>Q15</td>
<td>Q31</td>
<td>GPR</td>
<td>Misc</td>
<td>Expand the precision of a Q15 fractional value to create a Q31 fractional value by adding 16 least-significant bits to the input value.</td>
</tr>
<tr>
<td>PRECEQU.PH.QBL rd,rt</td>
<td>Unsigned Byte</td>
<td>Q15</td>
<td>GPR</td>
<td>Video</td>
<td>Expand the precision of two unsigned byte values by prepending a sign bit and adding seven least-significant bits to each to create two Q15 fractional values.</td>
</tr>
</tbody>
</table>
### Table 4.1 List of Instructions in MIPS® DSP Module in Arithmetic Sub-class (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRECEU.PH.QBL rd,rt</td>
<td>Unsigned Byte</td>
<td>Unsigned halfword</td>
<td>GPR</td>
<td>Video</td>
<td>Expand the precision of two unsigned byte values by adding eight least-significant bits to each to create two unsigned halfword values.</td>
</tr>
<tr>
<td>PRECEU.PH.QBR rd,rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRECEU.PH.QBLA rd,rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRECEU.PH.QBRA rd,rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 4.2 List of Instructions in MIPS® DSP Module in GPR-Based Shift Sub-class

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHLL.QB rd, rt, sa</td>
<td>Quad Unsigned Byte</td>
<td>Quad Unsigned Byte</td>
<td>GPR</td>
<td>Misc</td>
<td>Element-wise left shift of eight signed bytes. Zeros are inserted into the bits emptied by the shift. The shift amount is specified by the three least-significant bits of sa or rs.</td>
</tr>
<tr>
<td>SHLLV.QB rd, rt, rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHLL.PH rd, rt, sa</td>
<td>Pair Signed halfword</td>
<td>Pair Signed halfword</td>
<td>GPR</td>
<td>Misc</td>
<td>Element-wise left shift of two signed half-words, with optional saturation on overflow. Zeros are inserted into the bits emptied by the shift. The shift amount is specified by the four least-significant bits of sa or rs. Use the MIPS32 instructions SLL or SLLV for non-saturating shift operations.</td>
</tr>
<tr>
<td>SHLL_S.PH rd, rt, sa</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHLL_S.PH rd, rt, rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHLL_S.W rd, rt, sa</td>
<td>Signed Word</td>
<td>Signed Word</td>
<td>GPR</td>
<td>Misc</td>
<td>Left shift of a signed word, with saturation on overflow. Zeros are inserted into the bits emptied by the shift. The shift amount is specified by the five least-significant bits of sa or rs. Use the MIPS32 instructions SLL or SLLV for non-saturating shift operations.</td>
</tr>
<tr>
<td>SHLLS.W rd, rt, rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHRL.QB rd, rt, sa</td>
<td>Quad Unsigned Byte</td>
<td>Quad Unsigned Byte</td>
<td>GPR</td>
<td>Video</td>
<td>Element-wise logical right shift of four byte values. Zeros are inserted into the bits emptied by the shift. The shift amount is specified by the three least-significant bits of sa or rs.</td>
</tr>
<tr>
<td>SHRLV.QB rd, rt, rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHRL.PH rd, rt, sa</td>
<td>Pair Half-words</td>
<td>Pair Half-words</td>
<td>GPR</td>
<td>Video</td>
<td>Element-wise logical right shift of two half-word values. Zeros are inserted into the bits emptied by the shift. The shift amount is specified by the four least-significant bits of rs or the sa argument.</td>
</tr>
<tr>
<td>SHRLV.PH rd, rt, rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPSDSP-R2 Only</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHRA.QB rd,rt,sa</td>
<td>Quad Byte</td>
<td>Quad Byte</td>
<td>GPR</td>
<td>Misc</td>
<td>Element-wise arithmetic (sign preserving) right shift of four byte values. Optional rounding may be performed, adding 1 at the most-significant discard bit position. The shift amount is specified by the three least-significant bits of rs or by the argument sa.</td>
</tr>
<tr>
<td>SHRA_R.QB rd,rt,sa</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHRAV.QB rd,rt,rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHRAV_R.QB rd,rt,rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPSDSP-R2 Only</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Table 4.2 List of Instructions in MIPS® DSP Module in GPR-Based Shift Sub-class (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHRA.PH rd, rt, sa</td>
<td>Pair Signed halfword</td>
<td>Pair Signed halfword</td>
<td>GPR</td>
<td>Misc</td>
<td>Element-wise arithmetic (sign preserving) right shift of two halfword values. Optionally, rounding may be performed, adding 1 at the most-significant discard bit position. The shift amount is specified by the four least-significant bits of rs or by the argument sa.</td>
</tr>
<tr>
<td>SHRAV.PH rd, rt, rs</td>
<td>Pair Signed halfword</td>
<td>Pair Signed halfword</td>
<td>GPR</td>
<td>Misc</td>
<td>Element-wise arithmetic (sign preserving) right shift of two halfword values. Optionally, rounding may be performed, adding 1 at the most-significant discard bit position. The shift amount is specified by the four least-significant bits of rs or by the argument sa.</td>
</tr>
<tr>
<td>SHRA_R.PH rd, rt, sa</td>
<td>Signed Word</td>
<td>Signed Word</td>
<td>GPR</td>
<td>Video</td>
<td>Arithmetic (sign preserving) right shift of a word value. Optionally, rounding may be performed, adding 1 at the most-significant discard bit position. The shift amount is specified by the five least-significant bits of rs or the argument sa.</td>
</tr>
<tr>
<td>SHRAV_R.PH rd, rt, rs</td>
<td>Signed Word</td>
<td>Signed Word</td>
<td>GPR</td>
<td>Video</td>
<td>Arithmetic (sign preserving) right shift of a word value. Optionally, rounding may be performed, adding 1 at the most-significant discard bit position. The shift amount is specified by the five least-significant bits of rs or the argument sa.</td>
</tr>
</tbody>
</table>

## Table 4.3 List of Instructions in MIPS® DSP Module in Multiply Sub-class

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULEU_S.PH.QBL rd,rs,rt</td>
<td>Pair Unsigned Byte, Pair Unsigned Halfword</td>
<td>Pair Unsigned Fullword</td>
<td>GPR</td>
<td>Still Image</td>
<td>Element-wise multiplication of two unsigned byte values from register rs with two unsigned halfword values from register rt. Each 24-bit product is truncated to 16 bits, with saturation if the product exceeds 0xFFFF, and written to the corresponding element in the destination register.</td>
</tr>
<tr>
<td>MULEU_S.PH.QBR rd,rs,rt</td>
<td>Pair Unsigned Byte, Pair Unsigned Halfword</td>
<td>Pair Unsigned Fullword</td>
<td>GPR</td>
<td>Still Image</td>
<td>Element-wise multiplication of two unsigned byte values from register rs with two unsigned halfword values from register rt. Each 24-bit product is truncated to 16 bits, with saturation if the product exceeds 0xFFFF, and written to the corresponding element in the destination register.</td>
</tr>
<tr>
<td>MULQ_RS.PH rd,rs,rt</td>
<td>Pair Q15</td>
<td>Pair Q15</td>
<td>GPR</td>
<td>Misc</td>
<td>Element-wise multiplication of two Q15 fractional values to create two Q15 fractional results, with rounding and saturation. After multiplication, each 32-bit product is rounded up by adding 0x00008000, then truncated to create a Q15 fractional value that is written to the destination register. If both multiplicands are -1.0, the result is saturated to the maximum positive Q15 fractional value. To stay compliant with the base architecture, this instruction leaves the base HI-LO pair UNPREDICTABLE after the operation. The other DSP Module accumulators ac1-ac3 are untouched.</td>
</tr>
</tbody>
</table>
### Table 4.3 List of Instructions in MIPS® DSP Module in Multiply Sub-class (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULEQ_S.W.PHL rd,rs,rt</td>
<td>Pair Q15</td>
<td>Q31</td>
<td>GPR</td>
<td>VoIP</td>
<td>Multiplication of two Q15 fractional values, shifting the product left by 1 bit to create a Q31 fractional result. If both multiplicands are -1.0 the result is saturated to the maximum positive Q31 value. To stay compliant with the base architecture, this instruction leaves the base HI-LO pair UNPREDICTABLE after the operation. The other DSP Module accumulators ac1-ac3 must be untouched.</td>
</tr>
<tr>
<td>DPAU.H.QBL DPAU.H.QBR</td>
<td>Pair Bytes</td>
<td>Halfword</td>
<td>Acc</td>
<td>Image</td>
<td>Dot-product accumulation. Two pairs of corresponding unsigned byte elements from source registers rt and rs are separately multiplied, and the two 16-bit products are then summed together. The summed products are then added to the accumulator.</td>
</tr>
<tr>
<td>DPSU.H.QBL DPSU.H.QBR</td>
<td>Pair Bytes</td>
<td>Halfword</td>
<td>Acc</td>
<td>Image</td>
<td>Dot-product subtraction. Two pairs of corresponding unsigned byte elements from source registers rt and rs are separately multiplied, and the two 16-bit products are then summed together. The summed products are then subtracted from the accumulator.</td>
</tr>
<tr>
<td>DPA.W.PH ac,rs,rt MIPSDSP-R2 Only</td>
<td>Pair Signed Halfword</td>
<td>Pair Signed Halfword</td>
<td>ac</td>
<td>VoIP / SoftM</td>
<td>Dot-product accumulation. The two pairs of corresponding signed integer halfword values from source registers rt and rs are separately multiplied to create two separate integer word products. The products are then summed and accumulated into the specified accumulator.</td>
</tr>
<tr>
<td>DPAX.W.PH ac,rs,rt MIPSDSP-R2 Only</td>
<td>Pair Signed Halfword</td>
<td>Double-word</td>
<td>ac</td>
<td>VoIP</td>
<td>Dot-product with crossed operands and accumulation. The two crossed pairs of signed integer halfword values from source registers rt and rs are separately multiplied to create two separate integer word products. The products are then summed and accumulated into the specified accumulator.</td>
</tr>
</tbody>
</table>
### Table 4.3 List of Instructions in MIPS® DSP Module in Multiply Sub-class (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPAQ_S.W.PH ac,rs,rt</td>
<td>Pair Q15</td>
<td>Q32.31</td>
<td>ac</td>
<td>VoIP / SoftM</td>
<td>Dot-product accumulation. Two pairs of corresponding Q15 fractional values from source registers ( rt ) and ( rs ) are separately multiplied and left-shifted 1 bit to create two Q31 fractional products. For each product, if both multipli- cands are equal to -1.0 the product is clamped to the maximum positive Q31 fractional value. The products are then summed, and the sum is then sign extended to the width of the accumu- lator and accumulated into the specified accumu- lator. This instruction may be used to compute the imaginary component of a 16-bit complex multiplication operation after first swapping the operands to place them in the correct order.</td>
</tr>
<tr>
<td>DPAQX_S.W.PH ac,rs,rt</td>
<td>Pair Signed Halfword</td>
<td>Q32.31</td>
<td>ac</td>
<td>VoIP</td>
<td>Dot-product with saturating fractional multi- plication and using crossed operands, with a final accumulation. The two crossed pairs of signed fractional halfword values from source registers ( rt ) and ( rs ) are separately multiplied to create two separate fractional word products. The products are then summed and accumulated into the specified accumulator.</td>
</tr>
<tr>
<td>DPAQX_SA.W.PH ac,rs,rt</td>
<td>Pair Signed Halfword</td>
<td>Q32.31</td>
<td>ac</td>
<td>VoIP</td>
<td>Dot-product with saturating fractional multi- plication and using crossed operands, with a final saturating accumulation. The two crossed pairs of signed fractional halfword values from source registers ( rt ) and ( rs ) are separately multiplied to create two separate fractional word products. The products are then summed and accumulated with saturation into the specified accumulator.</td>
</tr>
<tr>
<td>DPS.W.PH ac,rs,rt</td>
<td>Pair Signed Halfword</td>
<td>Double- word</td>
<td>ac</td>
<td>VoIP / SoftM</td>
<td>Dot-product subtraction. The two pairs of cor- responding signed integer halfword values from source registers ( rt ) and ( rs ) are separately multiplied to create two separate integer word products. The products are then summed and subtracted from the specified accumulator.</td>
</tr>
<tr>
<td>DPSX.W.PH ac,rs,rt</td>
<td>Pair Signed Halfword</td>
<td>Q32.31</td>
<td>ac</td>
<td>VoIP</td>
<td>Dot-product with crossed operands and sub- traction. The two crossed pairs of signed integer halfword values from source registers ( rt ) and ( rs ) are separately multiplied to create two separate integer word products. The products are then summed and subtracted into the specified accumulator.</td>
</tr>
</tbody>
</table>
## Table 4.3 List of Instructions in MIPS® DSP Module in Multiply Sub-class (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPSQ_S.W.PH ac,rs,rt</td>
<td>Pair Q15</td>
<td>Q32.31</td>
<td>ac</td>
<td>VoIP / SoftM</td>
<td>Dot-product subtraction. Two pairs of corresponding Q15 fractional values from source registers rt and rs are separately multiplied and left-shifted 1 bit to create two Q31 fractional products. For each product, if both multiplicands are equal to -1.0 the product is clamped to the maximum positive Q31 fractional value. The products are then summed, and the sum is then sign extended to the width of the accumulator and subtracted from the specified accumulator. This instruction may be used to compute the imaginary component of a 16-bit complex multiplication operation after first swapping the operands to place them in the correct order.</td>
</tr>
<tr>
<td>DPSQX_S.W.PH ac,rs,rt</td>
<td>Pair Signed Halfword</td>
<td>Q32.31</td>
<td>ac</td>
<td>VoIP</td>
<td>Dot-product with saturating fractional multiplication and using crossed operands, with a final subtraction. The two crossed pairs of signed fractional halfword values from source registers rt and rs are separately multiplied to create two separate fractional word products. The products are then summed and subtracted from the specified accumulator.</td>
</tr>
<tr>
<td>DPSQX_SA.W.PH ac,rs,rt</td>
<td>Pair Signed Halfword</td>
<td>Q32.31</td>
<td>ac</td>
<td>VoIP</td>
<td>Dot-product with saturating fractional multiplication and using crossed operands, with a final saturating subtraction. The two crossed pairs of signed fractional halfword values from source registers rt and rs are separately multiplied to create two separate fractional word products. The products are then summed and subtracted with saturation into the specified accumulator.</td>
</tr>
<tr>
<td>MULSAQ_S.W.PH ac,rs,rt</td>
<td>Pair Q15</td>
<td>Q32.31</td>
<td>ac</td>
<td>SoftM</td>
<td>Complex multiplication step. Performs element-wise fractional multiplication of the two Q15 fractional values from registers rt and rs, subtracting one product from the other to create a Q31 fractional result that is added to accumulator ac. The intermediate products are saturated to the maximum positive Q31 fractional value if both multiplicands are equal to -1.0.</td>
</tr>
<tr>
<td>Instruction Mnemonics</td>
<td>Input Data Type</td>
<td>Output Data Type</td>
<td>Writes GPR / ac / DSPControl</td>
<td>App</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>-----------------</td>
<td>------------------</td>
<td>------------------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DPAQ_SA.L.W ac,rs,rt</td>
<td>Q31</td>
<td>Q63</td>
<td>ac</td>
<td>Audio</td>
<td>Fractional multiplication of two Q31 fractional values to produce a Q63 fractional product. If both multiplicands are -1.0 the product is saturated to the maximum positive Q63 fractional value. The product is then added to accumulator ac. If the addition results in overflow or underflow, the accumulator is saturated to the maximum positive or minimum negative value.</td>
</tr>
<tr>
<td>DPSQ_SA.L.W ac,rs,rt</td>
<td>Q31</td>
<td>Q63</td>
<td>ac</td>
<td>Audio</td>
<td>Fractional multiplication of two Q31 fractional values to produce a Q63 fractional product. If both multiplicands are -1.0 the product is saturated to the maximum positive Q63 fractional value. The product is then subtracted from accumulator ac. If the addition results in overflow or underflow, the accumulator is saturated to the maximum positive or minimum negative value.</td>
</tr>
<tr>
<td>MAQ_S.W.PHL ac,rs,rt</td>
<td>Q15</td>
<td>Q32,31</td>
<td>ac</td>
<td>SoftM</td>
<td>Fractional multiply-accumulate. The product of two Q15 fractional values is sign extended to the width of the accumulator and added to accumulator ac. The intermediate product is saturated to the maximum positive Q31 fractional value if both multiplicands are equal to -1.0.</td>
</tr>
<tr>
<td>MAQ_S.W.PHR ac,rs,rt</td>
<td>Q15</td>
<td>Q31</td>
<td>ac</td>
<td>speech</td>
<td>Fractional multiply-accumulate with saturation after accumulation. The product of two Q15 fractional values is sign extended to the width of the accumulator and added to accumulator ac. The intermediate product is saturated to the maximum positive Q31 fractional value if both multiplicands are equal to -1.0. If the accumulation results in overflow or underflow, the accumulator value is saturated to the maximum positive or minimum negative Q31 fractional value.</td>
</tr>
<tr>
<td>MUL.PH rd,rs,rt</td>
<td>Pair Signed Halfword</td>
<td>Pair Signed Halfword</td>
<td>GPR</td>
<td>speech</td>
<td>Element-wise multiplication of two vectors of signed integer halfwords, writing the 16 least-significant bits of each 32-bit product to the corresponding element of the destination register. Optional saturation clamps each 16-bit result to the maximum positive or minimum negative value if the product cannot be accurately represented in 16 bits.</td>
</tr>
</tbody>
</table>
### Table 4.3 List of Instructions in MIPS® DSP Module in Multiply Sub-class (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULQ_S.PH rd,rs,rt</td>
<td>Pair Q15</td>
<td>Pair Q15</td>
<td>GPR</td>
<td>speech</td>
<td>Element-wise multiplication of two vectors of Q15 fractional halfwords, writing the 16 most-significant bits of each Q31-format product to the corresponding element of the destination register. Each result is saturated to the maximum positive Q15 value if both multiplicands were equal to -1.0 (0x8000 hexadecimal).</td>
</tr>
<tr>
<td>MULQ_S.W rd,rs,rt</td>
<td>Q31</td>
<td>Q31</td>
<td>GPR</td>
<td>speech</td>
<td>Fractional multiplication of two Q31 format words to create a Q63 format result that is truncated by discarding the 32 least-significant bits before being written to the destination register. The result is saturated to the maximum positive Q31 value if both multiplicands were equal to -1.0 (0x80000000 hexadecimal).</td>
</tr>
<tr>
<td>MULQ_RS.W rd,rs,rt</td>
<td>Q31</td>
<td>Q31</td>
<td>GPR</td>
<td>speech</td>
<td>Multiplication of two Q31 fractional words to create a Q63-format intermediate product that is rounded up by adding a 1 at bit position 31. The 32 most-significant bits of the rounded result are then written to the destination register. If both multiplicands were equal to -1.0 (0x80000000 hexadecimal), rounding is not performed and the result is clamped to the maximum positive Q31 value before being written to the destination.</td>
</tr>
<tr>
<td>MULSA.W.PH ac,rs,rt</td>
<td>Pair Signed Halfword</td>
<td>Double-word</td>
<td>ac</td>
<td>speech</td>
<td>Element-wise multiplication of two vectors of signed integer halfwords to create two 32-bit word intermediate results. The right intermediate result is subtracted from the left intermediate result, and the resulting sum is accumulated into the specified accumulator.</td>
</tr>
<tr>
<td>MADD ac,rs,rt</td>
<td>Word</td>
<td>Double-word</td>
<td>ac</td>
<td>Misc</td>
<td>Allows these instructions to target accumulators ac1, ac2, and ac3 (in addition to the original ac0 destination).</td>
</tr>
</tbody>
</table>
4.1 The MIPS® DSP Module Instruction Summary

Table 4.4 List of Instructions in MIPS® DSP Module in Bit/Manipulation Sub-class

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITREV rd,rt</td>
<td>Unsigned Word</td>
<td>Unsigned Word</td>
<td>GPR</td>
<td>Audio / FFT</td>
<td>Reverse the order of the 16 least-significant bits of register rt, writing the result to register rd. The most-significant bits are set to zero.</td>
</tr>
<tr>
<td>NISV rt,rs</td>
<td>Unsigned Word</td>
<td>Unsigned Word</td>
<td>GPR</td>
<td>Misc</td>
<td>Like the Release 2 INS instruction, except that the 5 bits for pos and size values are obtained from the DSPControl register. size = scount[14:10], and pos = pos[20:16].</td>
</tr>
<tr>
<td>REPL.QB rd,imm REPLV.QB rd,rt</td>
<td>Byte</td>
<td>Quad Byte</td>
<td>GPR</td>
<td>Video / Misc</td>
<td>Replicate a signed byte value into the four byte elements of register rd. The byte value is given by the 8 least-significant bits of the specified 10-bit immediate constant or by the 8 least-significant bits of register rt.</td>
</tr>
<tr>
<td>CMPU.EQ.QB rs,rt</td>
<td>Quad Unsigned Byte</td>
<td>Quad Unsigned Byte</td>
<td>DSPControl</td>
<td>Video</td>
<td>Element-wise unsigned comparison of the four unsigned byte elements of rs and rt, recording the boolean comparison results to the four right-most bits in the ccond field of the DSPControl register.</td>
</tr>
<tr>
<td>CMPGU.EQ.QB rd,rs,rt</td>
<td>Quad Unsigned Byte</td>
<td>Quad Unsigned Byte</td>
<td>DSPControl</td>
<td>Video</td>
<td>Element-wise unsigned comparison of the four right-most unsigned byte elements of rs and rt, recording the boolean comparison results to the four right-most bits of register rd and to the four right-most bits in the ccond field of the DSPControl register.</td>
</tr>
<tr>
<td>CMPGU.LE.QB rd,rs,rt</td>
<td>Quad Unsigned Byte</td>
<td>Quad Unsigned Byte</td>
<td>GPR</td>
<td>Video</td>
<td>Element-wise unsigned comparison of the four right-most unsigned byte elements of rs and rt, recording the boolean comparison results to the four least-significant bits of register rd.</td>
</tr>
</tbody>
</table>

Table 4.5 List of Instructions in MIPS® DSP Module in Compare-Pick Sub-class

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPU.LT.QB rs,rt</td>
<td>Quad Unsigned Byte</td>
<td>Quad Unsigned Byte</td>
<td>DSPControl</td>
<td>Video</td>
<td>Element-wise unsigned comparison of the four unsigned byte elements of rs and rt, recording the boolean comparison results to the four right-most bits in the ccond field of the DSPControl register.</td>
</tr>
<tr>
<td>CMPGU.LT.QB rd,rs,rt</td>
<td>Quad Unsigned Byte</td>
<td>Quad Unsigned Byte</td>
<td>GPR</td>
<td>Video</td>
<td>Element-wise unsigned comparison of the four right-most unsigned byte elements of rs and rt, recording the boolean comparison results to the four right-most bits in the ccond field of the DSPControl register.</td>
</tr>
<tr>
<td>CMPGU.LE.QB rd,rs,rt</td>
<td>Quad Unsigned Byte</td>
<td>Quad Unused Byte</td>
<td>GPR</td>
<td>Video</td>
<td>Element-wise unsigned comparison of the four right-most unsigned byte elements of rs and rt, recording the boolean comparison results to the four least-significant bits of register rd.</td>
</tr>
</tbody>
</table>

MIPS® DSP Module for MIPS32™ Architecture, Revision 3.01
### Table 4.5 List of Instructions in MIPS® DSP Module in Compare-Pick Sub-class (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP.EQ.PH rs,rt</td>
<td>Pair Signed halfword</td>
<td>Pair Signed halfword</td>
<td>DSPControl</td>
<td>Misc</td>
<td>Element-wise signed comparison of the two halfword elements of rs and rt, recording the boolean comparison results to the two right-most bits in the ccond field of the DSPControl register.</td>
</tr>
<tr>
<td>CMP.LT.PH rs,rt</td>
<td>Quad Unsigned Byte</td>
<td>Quad Unsigned Byte</td>
<td>GPR</td>
<td>Video</td>
<td>Element-wise selection of unsigned bytes from the four bytes of registers rs and rt into the corresponding elements of register rd, based on the value of the four right-most bits of the ccond field in the DSPControl register. If the corresponding ccond bit is 1, the byte value is copied from register rs, otherwise it is copied from rt.</td>
</tr>
<tr>
<td>CMP.LE.PH rs,rt</td>
<td>Pair Signed halfword</td>
<td>Pair Signed halfword</td>
<td>GPR</td>
<td>Misc</td>
<td>Element-wise selection of signed halfwords from the two halfwords in registers rs and rt into the corresponding elements of register rd, based on the value of the two right-most bits of the ccond field in the DSPControl register. If the corresponding ccond bit is 1, the halfword value is copied from register rs, otherwise it is copied from rt.</td>
</tr>
<tr>
<td>PICK.QB rd,rs,rt</td>
<td>Two Words</td>
<td>Word</td>
<td>GPR</td>
<td>Misc</td>
<td>Shifts the 32-bit word in register rt left by sa bits, inserting the sa least-significant bits from register rs into the bit positions emptied by the shift. The 32-bit result is then written to register rt.</td>
</tr>
<tr>
<td>PICK.PH rd,rs,rt</td>
<td>Two Words</td>
<td>Word</td>
<td>GPR</td>
<td>Misc</td>
<td>Shifts the 32-bit word in register rt right by sa bits, inserting the sa least-significant bits from register rs into the bit positions emptied by the shift. The 32-bit result is then written to register rt.</td>
</tr>
<tr>
<td>APPEND rt,rs,sa</td>
<td>Two Words</td>
<td>Word</td>
<td>GPR</td>
<td>Misc</td>
<td>Packs bp bytes from register rt and (4-bp) bytes from register rs into a 32-bit word and writes it to register rt.</td>
</tr>
<tr>
<td>PACKRL.PH rd,rs,rt</td>
<td>Pair Signed Halfwords</td>
<td>Pair Signed Halfword</td>
<td>GPR</td>
<td>Misc</td>
<td>Pack two halfwords taken from registers rs and rt into destination register rd.</td>
</tr>
<tr>
<td>Instruction Mnemonics</td>
<td>Input Data Type</td>
<td>Output Data Type</td>
<td>Writes GPR / ac / DSPControl</td>
<td>App</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------</td>
<td>-----------------</td>
<td>-----------------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>EXTR.W rt,ac,shift</td>
<td>Q63</td>
<td>Q31</td>
<td>GPR</td>
<td>Misc</td>
<td>Extract a Q31 fractional value from the 32 least-significant bits of 64-bit accumulator ac. The accumulator value may be shifted right logically by shift bits prior to the extraction, and the extracted value may be optionally rounded or rounded and saturated before being written to register rt. The shift argument value ranges from 0 to 31. The optional rounding step adds 1 at the most-significant bit position discarded by the shift. The optional saturation clamps the extracted value to the maximum positive Q31 value if the rounding step results in overflow.</td>
</tr>
<tr>
<td>EXTR_R.W rt,ac,shift</td>
<td>Q63</td>
<td>Q31</td>
<td>GPR</td>
<td>Misc</td>
<td>Extract a Q31 fractional value from the 32 least-significant bits of 64-bit accumulator ac. The accumulator value may be shifted right logically by shift bits prior to the extraction, and the extracted value may be optionally rounded or rounded and saturated before being written to register rt. The shift argument value ranges from 0 to 31. The optional rounding step adds 1 at the most-significant bit position discarded by the shift. The optional saturation clamps the extracted value to the maximum positive Q31 value if the rounding step results in overflow.</td>
</tr>
<tr>
<td>EXTRV_S.H rt,ac,rs</td>
<td>Q63</td>
<td>Q15</td>
<td>GPR</td>
<td>Misc</td>
<td>Extract a Q15 fractional value from the 16 least-significant bits of 64-bit accumulator ac. The accumulator value may be shifted right logically by shift bits prior to the extraction, and the extracted value is saturated before being written to register rt. The shift argument value ranges from 0 to 31. The saturation clamps the extracted value to the maximum positive or minimum negative Q15 value if the shifted accumulator value cannot be represented accurately as a Q15 format value.</td>
</tr>
<tr>
<td>EXTRV_S.H rt,ac,rs</td>
<td>Q63</td>
<td>Q15</td>
<td>GPR</td>
<td>Misc</td>
<td>Extract a Q15 fractional value from the 16 least-significant bits of 64-bit accumulator ac. The accumulator value may be shifted right logically by shift bits prior to the extraction, and the extracted value is saturated before being written to register rt. The shift argument ranges from 0 to 31 and is given by the five least-significant bits of register rs. The saturation clamps the extracted value to the maximum positive or minimum negative Q15 value if the shifted accumulator value cannot be represented accurately as a Q15 format value.</td>
</tr>
</tbody>
</table>
### MIPS® DSP Module Instruction Summary

#### Table 4.6 List of Instructions in MIPS® DSP Module in Accumulator and DSPControl Access Sub-class

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTRV.W rt,ac,rs</td>
<td>Q63</td>
<td>Q31</td>
<td>GPR</td>
<td>Misc</td>
<td>Extract a Q31 fractional value from the 32 least-significant bits of 64-bit accumulator ac. The accumulator value may be shifted right logically by shift bits prior to the extraction, and the extracted value may be optionally rounded or rounded and saturated before being written to register rt. The shift argument value is provided by the five least-significant bits of rs and ranges from 0 to 31. The optional rounding step adds 1 at the most-significant bit position discarded by the shift. The optional saturation clamps the extracted value to the maximum positive Q31 value if the rounding step results in overflow.</td>
</tr>
<tr>
<td>EXTRV_R.W rt,ac,rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXTRV_RS.W rt,ac,rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXTP rt,ac,size</td>
<td>Unsigned DWord</td>
<td>Unsigned Word</td>
<td>GPR / DSPControl</td>
<td>Audio / Video</td>
<td>Extract a set of size+1 contiguous bits from accumulator ac, right-justifying and sign-extending the result to 32 bits before writing the result to register rt. The position of the left-most bit to extract is given by the value of the pos field in the DSPControl register (see Appendix A for details). The number of bits (less one) to extract is provided either by the size immediate operand or by the five least-significant bits of rs. The EXTPDP and EXTPDPV instructions also decrement the pos field by size+1 to facilitate sequential bit field extraction operations.</td>
</tr>
<tr>
<td>EXTPV rt,ac,rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXTPDP rt,ac,size</td>
<td>Unsigned Word</td>
<td>Unsized Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXTPDPV rt,ac,rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHILO ac,shift</td>
<td>Unsigned DWord</td>
<td>Unsigned DWord</td>
<td>ac</td>
<td>Misc</td>
<td>Shift accumulator ac left or right by the specified number of bits, writing the shifted value back to the accumulator. The signed shift argument is specified either by the immediate operand shift or by the six least-significant bits of register rs. A negative shift argument results in a right shift of up to 32 bits, and a positive shift argument results in a left shift of up to 31 bits.</td>
</tr>
<tr>
<td>SHILOV ac,rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MTHLIP rs, ac</td>
<td>Unsigned Word</td>
<td>Unsigned Word</td>
<td>ac / DSPControl</td>
<td>Audio / Video</td>
<td>Copy the LO register of the specified accumulator to the HI register, copy rs to LO, and increment the pos field in DSPControl by 32.</td>
</tr>
<tr>
<td>MFHI/MFLO/MTHI/MTLO</td>
<td>Unsigned Word</td>
<td>Unsigned Word</td>
<td>GPR/ac</td>
<td>Misc</td>
<td>Copy an unsigned word to or from the specified accumulator HI or LO register to the specified GPR.</td>
</tr>
</tbody>
</table>
4.1 The MIPS® DSP Module Instruction Summary

Table 4.6 List of Instructions in MIPS® DSP Module in Accumulator and DSPControl Access Sub-class

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRDSP rt,mask</td>
<td>Unsigned Word</td>
<td>Unsigned Word</td>
<td>DSPControl</td>
<td>Misc</td>
<td>Overwrite specific fields in the DSPControl register using the corresponding bits from the specified GPR. Bits in the mask argument correspond to specific fields in DSPControl; a value of 1 causes the corresponding DSPControl field to be overwritten using the corresponding bits in rt, otherwise the field is unchanged.</td>
</tr>
<tr>
<td>RDDSP rt,mask</td>
<td>Unsigned Word</td>
<td>Unsigned Word</td>
<td>GPR</td>
<td>Misc</td>
<td>Copy the values of specific fields in the DSPControl register to the specified GPR. Bits in the mask argument correspond to specific fields in DSPControl; a value of 1 causes the corresponding DSPControl field to be copied to the corresponding bits in rt, otherwise the bits in rt are unchanged.</td>
</tr>
</tbody>
</table>

Table 4.7 List of Instructions in MIPS® DSP Module in Indexed-Load Sub-class

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBUX rd,index(base)</td>
<td>-</td>
<td>Unsigned byte</td>
<td>GPR</td>
<td>Misc</td>
<td>Index byte load from address base+(index). Loads the byte in the low-order bits of the destination register and zero-extends the result.</td>
</tr>
<tr>
<td>LHX rd,index(base)</td>
<td>-</td>
<td>Signed halfword</td>
<td>GPR</td>
<td>Misc</td>
<td>Index halfword load from address base+(index). Loads the halfword in the low-order bits of the register and sign-extends the result.</td>
</tr>
<tr>
<td>LWX rd, index(base)</td>
<td>-</td>
<td>Signed Word</td>
<td>GPR</td>
<td>Misc</td>
<td>Indexed word load from address base+(index).</td>
</tr>
</tbody>
</table>

Table 4.8 List of Instructions in MIPS® DSP Module in Branch Sub-class

<table>
<thead>
<tr>
<th>Instruction Mnemonics</th>
<th>Input Data Type</th>
<th>Output Data Type</th>
<th>Writes GPR / ac / DSPControl</th>
<th>App</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPOSGE32 offset</td>
<td>-</td>
<td>-</td>
<td>Audio / Video</td>
<td></td>
<td>Branch if the pos value is greater than or equal to integer 32.</td>
</tr>
<tr>
<td>BPOSGE32C offset</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 5

Instruction Encoding

5.1 Instruction Bit Encoding

This chapter describes the bit encoding tables used for the MIPS DSP Module. **Table 5.1** describes the meaning of the symbols used in the tables. These tables only list the instruction encoding for the MIPS DSP Module instructions. See Volumes I and II of this multi-volume set for a full encoding of all instructions.

**Figure 5.1** shows a sample encoding table and the instruction **opcode** field this table encodes. Bits 31..29 of the **opcode** field are listed in the left-most columns of the table. Bits 28..26 of the **opcode** field are listed along the top-most rows of the table. Both decimal and binary values are given, with the first three bits designating the row, and the last three bits designating the column.

An instruction’s encoding is found at the intersection of a row (bits 31..29) and column (bits 28..26) value. For instance, the **opcode** value for the instruction labelled EX1 is 33 (decimal, row and column), or 011011 (binary). Similarly, the **opcode** value for EX2 is 64 (decimal), or 110100 (binary).

**Figure 5.1 Sample Bit Encoding Table**

<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>14</th>
<th>13</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>opcode</strong></td>
<td><strong>rs</strong></td>
<td><strong>rt</strong></td>
<td><strong>immediate</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Binary encoding of opcode (28..26)**

**Decimal encoding of opcode (28..26)**

**Decimal encoding of opcode (31..29)**

<table>
<thead>
<tr>
<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>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>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<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>
</tr>
<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>
</tr>
<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>
</tr>
</tbody>
</table>

**Binary encoding of opcode (31..29)**

**EX1**

**EX2**
Table 5.1 Symbols Used in the Instruction Encoding Tables

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Operation or field codes marked with this symbol are reserved for future use. Executing such an instruction must cause a Reserved Instruction Exception.</td>
</tr>
<tr>
<td>δ</td>
<td>(Also italic field name.) Operation or field codes marked with this symbol denotes a field class. The instruction word must be further decoded by examining additional tables that show values for another instruction field.</td>
</tr>
<tr>
<td>β</td>
<td>Operation or field codes marked with this symbol represent a valid encoding for a higher-order MIPS ISA level. Executing such an instruction must cause a Reserved Instruction Exception.</td>
</tr>
<tr>
<td>θ</td>
<td>Operation or field codes marked with this symbol are available to licensed MIPS partners. To avoid multiple conflicting instruction definitions, MIPS Technologies will assist the partner in selecting appropriate encoding if requested by the partner. The partner is not required to consult with MIPS Technologies when one of these encoding is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction Exception (SPECIAL2 encoding or coprocessor instruction encoding for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encoding for a coprocessor to which access is not allowed).</td>
</tr>
<tr>
<td>σ</td>
<td>Field codes marked with this symbol represent an EJTAG support instruction and implementation of this encoding is optional for each implementation. If the encoding is not implemented, executing such an instruction must cause a Reserved Instruction Exception. If the encoding is implemented, it must match the instruction encoding as shown in the table.</td>
</tr>
<tr>
<td>ε</td>
<td>Operation or field codes marked with this symbol are reserved for MIPS Modules/Application Specific Extensions. If the Module/ASE is not implemented, executing such an instruction must cause a Reserved Instruction Exception.</td>
</tr>
<tr>
<td>φ</td>
<td>Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS ISA. Software should avoid using these operation or field codes.</td>
</tr>
<tr>
<td>⊕</td>
<td>Operation or field codes marked with this symbol are valid for Release 2 implementations of the architecture. Executing such an instruction in a Release 1 implementation must cause a Reserved Instruction Exception.</td>
</tr>
</tbody>
</table>

Table 5.2 MIPS®DSP Module Encoding of Opcode Field

<table>
<thead>
<tr>
<th>opcode bits 28..26</th>
<th>bits 31..29</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></td>
<td>REGIMM δ</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>012</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SPECIAL3 00</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The instructions in the MIPS DSP Module are encoded in the SPECIAL3 space under the opcode map as shown in Table 5.2 and Table 5.3. The sub-encoding for individual instructions defined by the MIPS DSP Module are shown in the following tables in this chapter.

Table 5.3 MIPS® SPECIAL3 Encoding of Function Field for DSP Module Instructions

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>001</td>
<td>LX δ INSV * * *</td>
</tr>
<tr>
<td>010</td>
<td>ADDU.QB δ CMPU.EQ.QB δ ABSQ.S.PH δ SHLL.QB δ</td>
</tr>
</tbody>
</table>
| 011      | ADDUH.QB δ * * * * * * *
| 100      | * * * * * * * *
| 101      | * * * * * * * *
| 110      | DPA.W.PH δ APPEND δ * * * * * * *
| 111      | EXTR.W δ * * * * * * *

1. Release 2 of the Architecture added the SPECIAL3 opcode. Implementations of Release 1 of the Architecture signaled a Reserved Instruction Exception for this opcode and all function field values shown above.
2. The empty slots in this table are used by Release 2 instructions not shown here, refer to Volume II of this multi-volume specification for these instructions.

Table 5.4 MIPS® REGIMM Encoding of rt Field

<table>
<thead>
<tr>
<th>rt</th>
<th>bits 18..16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>bits 20..19</td>
<td>000 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td>0 00</td>
<td>* * * *</td>
</tr>
<tr>
<td>1 01</td>
<td>* * * *</td>
</tr>
<tr>
<td>2 10</td>
<td>* * * *</td>
</tr>
<tr>
<td>3 11</td>
<td>* * * * BPOSGE32</td>
</tr>
</tbody>
</table>

Each MIPS DSP Module instruction sub-class in SPECIAL3 that needs further decoding, is done via the op field as shown in Figure 5.2.

Figure 5.2 SPECIAL3 Encoding of ADDU.QB/CMPU.EQ.QB Instruction Sub-classes

<table>
<thead>
<tr>
<th>31 26 25 21 16 11 06 5 4 0</th>
<th>6 5 5 5 5 5 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL3 011111</td>
<td>rs rt rd op ADDU.QB 01 0000</td>
</tr>
</tbody>
</table>
Table 5.5 MIPS® ADDU.QB Encoding of op Field\(^1\)

<table>
<thead>
<tr>
<th>op</th>
<th>bits 8..6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
</tr>
<tr>
<td>00 ADDU.QB</td>
<td>SUBU.QB</td>
</tr>
<tr>
<td>10 ADDU.PH</td>
<td>SUBU.PH</td>
</tr>
<tr>
<td>11 ADDQ.PH</td>
<td>ADDU.PH</td>
</tr>
</tbody>
</table>

1. The op field is decoded to identify the final instructions. Entries in this table with no mnemonic are reserved for future use by MIPS Technologies and may or may not cause a Reserved Instruction exception.

Table 5.6 MIPS® CMPU.EQ.QB Encoding of op Field

<table>
<thead>
<tr>
<th>op</th>
<th>bits 8..6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
</tr>
<tr>
<td>00 CMPU.EQ.QB</td>
<td>CMPU.LT.QB</td>
</tr>
<tr>
<td>01 CMP.EQ.PH</td>
<td>CMP.LT.PH</td>
</tr>
<tr>
<td>10 CMP.QB</td>
<td>CMP.GD.QB</td>
</tr>
<tr>
<td>11 CMP.GD.QB</td>
<td>CMP.GD.LE.QB</td>
</tr>
</tbody>
</table>

Figure 5.3 SPECIAL3 Encoding of ABSQ.S.PH Instruction Sub-class without Immediate Field

<table>
<thead>
<tr>
<th>SPECIAL3</th>
<th>011111</th>
<th>0</th>
<th>rt</th>
<th>rd</th>
<th>ABSQ.S.PH 01001</th>
<th>ABSQ.S.PH 01 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>04</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 5.4 SPECIAL3 Encoding of ABSQ.S.PH Instruction Sub-class with Immediate Field

<table>
<thead>
<tr>
<th>SPECIAL3</th>
<th>011111</th>
<th>immediate</th>
<th>rd</th>
<th>REPL.PH 01010</th>
<th>ABSQ.S.PH 01 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>04</td>
<td>6</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
5.1 Instruction Bit Encoding

For the LX sub-class of instructions, the format to interpret the op field is similar to the instructions above, with the exception that the rs and rt fields are named to be the base and index fields respectively for the indexed load operation. The instruction format is shown in Figure 5.6.

Figure 5.5 SPECIAL3 Encoding of SHLL.QB Instruction Sub-class

Table 5.7 MIPS® ABSQ_S.PH Encoding of op Field

<table>
<thead>
<tr>
<th>op</th>
<th>bits 8..6</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 10..9</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>0 00</td>
<td>+</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>

Table 5.8 MIPS® SHLL.QB Encoding of op Field

<table>
<thead>
<tr>
<th>op</th>
<th>bits 8..6</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 10..9</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>0 00</td>
<td>SHLL.QB</td>
</tr>
<tr>
<td>1 01</td>
<td>SHLL.PH</td>
</tr>
<tr>
<td>2 10</td>
<td>+</td>
</tr>
<tr>
<td>3 11</td>
<td>SHRL.PH</td>
</tr>
</tbody>
</table>
The sub-class of DPA.W.PH instruction target one of the accumulators for the destination. These instructions use the lower bits of the rd field of the opcode to specify the accumulator number which can range from 0 to 3. This format is shown in Figure 5.7.

### Table 5.9 MIPS® LX Encoding of op Field

<table>
<thead>
<tr>
<th>bits 8..6</th>
<th>bits 10..9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
</tbody>
</table>

### Table 5.10 MIPS® DPA.W.PH Encoding of op Field

<table>
<thead>
<tr>
<th>bits 8..6</th>
<th>bits 10..9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
</tbody>
</table>

The EXTR.W sub-class is an assortment that has three types of instructions:

1. In the first one, the destination is a GPR and this is specified by the rt field in the opcode, as shown in Figure 5.8. The source is an accumulator and this comes from the right-most 2 bits of the rd field, again, as shown in the figure. When a second source must be specified, then the rs field is used. The second value could be a 5-bit immediate or a variable from a GPR. The first and the second rows of Table 5.11 show this type of instruction.

2. The RDDSP and WRDSP instructions specify one immediate 6 bit mask field and a GPR that holds both the position and size values, as seen in Figure 5.9.

3. The MTHLIP instruction copies the LO part of the specified accumulator to the HI, the GPR contents to LO. In this case, the source rs field is used and the destination is specified by ac, which is both a source and destination, as shown in Figure 5.10. The SHILO and SHILOV instructions which shift the HI-LO pair and leave the result in the HI-LO register pair is a variant that does not use the source rs register. The shift amount can be specified as an immediate value or in the rs register as a variable value.
Finally, the opcode change for the MFHI and MTLO instructions requires the specification of the accumulator number. For the MTHI and MTLO instructions, the change will use bits 11 and 12 of the opcode to specify the accumulator, where the value of 0 provides backwards compatibility and refers to the original Hi-Lo pair. For the MFHI and MFLO instructions, the change will use bits 21 and 22 to encode the accumulator, and zero is the original pair as before.

**Table 5.11 MIPS® EXTR.W Encoding of op Field**

<table>
<thead>
<tr>
<th>op</th>
<th>bits 8..6</th>
<th>bits 10..9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 1 2 3 4 5 6 7</td>
<td>0 0 0 00 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td>0 0</td>
<td>EXTR.W</td>
<td>EXTRV.W</td>
</tr>
<tr>
<td>1 1</td>
<td>EXTPDP</td>
<td>EXTPDPV</td>
</tr>
<tr>
<td>2 1</td>
<td>RDDS</td>
<td>WRDS</td>
</tr>
<tr>
<td>3 1</td>
<td>SHILO</td>
<td>SHILOV</td>
</tr>
</tbody>
</table>

**Figure 5.11 SPECIAL3 Encoding of ADDUH.QB Instruction Sub-classes**

<table>
<thead>
<tr>
<th>SPECIAL3</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>op</th>
<th>ADDUH.QB</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
### Table 5.12 MIPS® ADDUH.QB Encoding of op Field

<table>
<thead>
<tr>
<th>op</th>
<th>bits 10..9</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>ADDUH.QB</td>
<td>SUBUH.QB</td>
<td>ADDUH_R.QB</td>
<td>SUBUH_R.QB</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>ADDQH.PH</td>
<td>SUBQH.PH</td>
<td>ADDQH_R.PH</td>
<td>SUBQH_R.PH</td>
<td>MUL.PH</td>
<td>*</td>
<td>*</td>
<td>MUL_S.PH</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>ADDQH.W</td>
<td>SUBQH.W</td>
<td>ADDQH_R.W</td>
<td>SUBQH_R.W</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>MULQ_S.W</td>
<td>MULQ_RS.W</td>
<td></td>
</tr>
</tbody>
</table>

1. The op field is decoded to identify the final instructions. Entries in this table with no mnemonic are reserved for future use by MIPS Technologies and may or may not cause a Reserved Instruction exception.

### Table 5.13 MIPS® APPEND Encoding of op Field

<table>
<thead>
<tr>
<th>op</th>
<th>bits 10..9</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>APPEND</td>
<td>PREPEND</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>BALIGN</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</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>
</tr>
</tbody>
</table>

### Figure 5.12 SPECIAL3 Encoding of APPEND Instruction Sub-class

<table>
<thead>
<tr>
<th>SPECIAL3</th>
<th>rs</th>
<th>rt</th>
<th>sa/bp</th>
<th>op</th>
<th>APPEND</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

### Table 5.13 MIPS® APPEND Encoding of op Field
Chapter 6

The MIPS® DSP Module Instruction Set

6.1 Compliance and Subsetting

There are no instruction subsets allowed for the MIPS DSP Module—all instructions must be implemented with all data format types as shown. Instructions are listed in alphabetical order, with a secondary sort on data type format from narrowest to widest, i.e., quad byte, paired halfword, and word.
ABSQ_S.PH  Find Absolute Value of Two Fractional Halfwords

Format: ABSQ_S.PH  rd, rt

Purpose: Find Absolute Value of Two Fractional Halfwords

Find the absolute value of each of a pair of Q15 fractional halfword values with 16-bit saturation.

Description: rd  sat16(abs(rt31..16)) || sat16(abs(rt15..0))

For each value in the pair of Q15 fractional halfword values in register rt, the absolute value is found and written to the corresponding Q15 halfword in register rd. If either input value is the minimum Q15 value (-1.0 in decimal, 0x8000 in hexadecimal), the corresponding result is saturated to 0x7FFF.

This instruction sets bit 20 in the DSPControl register in the ouflag field if either input value was saturated.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

```plaintext
tempB15..0    satAbs16( GPR[rt]31..16 )
tempA15..0    satAbs16( GPR[rt]15..0 )
GPR[rd]..0    tempB15..0  ||  tempA15..0

function satAbs16( a15..0 )
    if ( a15..0 = 0x8000 ) then
        DSPControlouflag:20    1
        temp15..0    0x7FFF
    else
        if ( a15 = 1 ) then
            temp15..0    -a15..0
        else
            temp15..0    a15..0
        endif
    endif
    return temp15..0
endfunction satAbs16
```

Exceptions:

Reserved Instruction, DSP Disabled
AbsQ_S.QB

Find Absolute Value of Four Fractional Byte Values

**Format:**
```
ABSQ_S.QB rd, rt
```

**Purpose:** Find Absolute Value of Four Fractional Byte Values

Find the absolute value of four fractional byte vector elements with saturation.

**Description:**
```
rd ← sat8(abs(rt31..24)) || sat8(abs(rt23..16)) || sat8(abs(rt15..8)) || sat8(abs(rt7..0))
```

For each value in the four Q7 fractional byte elements in register rt, the absolute value is found and written to the corresponding byte in register rd. If either input value is the minimum Q7 value (-1.0 in decimal, 0x80 in hexadecimal), the corresponding result is saturated to 0x7F.

This instruction sets bit 20 in ouflag field of the DSPControl register if any input value was saturated.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**
```
tempD7..0 ← abs8( GPR[rt]31..24 )
tempC7..0 ← abs8( GPR[rt]23..16 )
tempB7..0 ← abs8( GPR[rt]15..8 )
tempA7..0 ← abs8( GPR[rt]7..0 )
GPR[rd]..0 ← tempD7..0 || tempC7..0 || tempB7..0 || tempA7..0
```

```
function abs8( a7..0 )
  if ( a7..0 = 0x80 ) then
    DSPControl.ouflag:20 ← 1
    temp7..0 ← 0x7F
  else
    if ( a7 = 1 ) then
      temp7..0 ← -a7..0
    else
      temp7..0 ← a7..0
    endif
  endif
  return temp7..0
endfunction abs8
```

**Exceptions:**

Reserved Instruction, DSP Disabled
ABSQ_S.W  Find Absolute Value of Fractional Word

Format:  ABSQ_S.W  rd, rt

Purpose: Find Absolute Value of Fractional Word
Find the absolute value of a fractional Q31 value with 32-bit saturation.

Description: rd \leftarrow \text{sat32}(\text{abs}(rt_{31..0}))
The absolute value of the Q31 fractional value in register \( rt \) is found and written to destination register \( rd \). If the input value is the minimum Q31 value (-1.0 in decimal, 0x80000000 in hexadecimal), the result is saturated to 0x7FFFFFFF before being sign-extended and written to register \( rd \).
This instruction sets bit 20 in the \text{DSPControl} register in the \text{ouflag} field if the input value was saturated.

Restrictions:
No data-dependent exceptions are possible.
The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:
\[
\text{temp}_{31..0} \leftarrow \text{satAbs32}(\text{GPR}[rt]_{31..0})
\]
\[
\text{GPR}[rd]_{..0} \leftarrow \text{temp}_{31..0}
\]

function satAbs32( a_{31..0} )
if ( a_{31..0} = 0x80000000 ) then
\[
\text{DSPControlouflag:20} \leftarrow 1
\]
\[
\text{temp}_{31..0} \leftarrow 0x7FFFFFFF
\]
else
if ( a_{31} = 1 ) then
\[
\text{temp}_{31..0} \leftarrow -a_{31..0}
\]
else
\[
\text{temp}_{31..0} \leftarrow a_{31..0}
\]
endif
endif
return \text{temp}_{31..0}
endfunction satAbs32

Exceptions:
Reserved Instruction, DSP Disabled
ADDQ[_S].PH     Add Fractional Halfword Vectors

Format:
ADDQ[_S].PH     ADDQ.PH rd, rs, rt
ADDQ_S.PH rd, rs, rt

Purpose:
Add Fractional Halfword Vectors

Element-wise addition of two vectors of Q15 fractional values to produce a vector of Q15 fractional results, with optional saturation.

Description:
rd ← sat16(rs31..16 + rt31..16) || sat16(rs15..0 + rt15..0)

Each of the two fractional halfword elements in register rt are added to the corresponding fractional halfword elements in register rs.

For the non-saturating version of the instruction, the result of each addition is written into the corresponding element in register rd. If the addition results in overflow or underflow, the result modulo 2 is written to the corresponding element in register rd.

For the saturating version of the instruction, signed saturating arithmetic is performed, where an overflow is clamped to the largest representable value (0x7FFF hexadecimal) and an underflow to the smallest representable value (0x8000 hexadecimal) before being written to the destination register rd.

For each instruction, if either of the individual additions result in underflow, overflow, or saturation, a 1 is written to bit 20 in the DSPControl register in the ouflag field.

Restrictions:
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:
ADDQ.PH:
    tempB15..0 ← add16( GPR[rs]31..16 , GPR[rt]31..16 )
    tempA15..0 ← add16( GPR[rs]15..0 , GPR[rt]15..0 )
    GPR[rd]..0 ← tempB15..0 || tempA15..0

ADDQ_S.PH:
    tempB15..0 ← satAdd16( GPR[rs]31..16 , GPR[rt]31..16 )
    tempA15..0 ← satAdd16( GPR[rs]15..0 , GPR[rt]15..0 )
    GPR[rd]..0 ← tempB15..0 || tempA15..0

function add16( a15..0, b15..0 )
    temp16..0 ← ( a15 || a15..0 ) + ( b15 || b15..0 )
    if ( temp16 ≠ temp15 ) then
        DSPControlouflag:20 ← 1
    endif
    return temp15..0
endfunction add16
function satAdd16( a_{15..0}, b_{15..0} )
    temp_{16..0} \leftarrow ( a_{15} \ || \ a_{15..0} ) + ( b_{15} \ || \ b_{15..0} )
    if ( temp_{16} \neq temp_{15} ) then
        if ( temp_{16} = 0 ) then
            temp_{15..0} \leftarrow 0x7FFF
        else
            temp_{15..0} \leftarrow 0x8000
        endif
        DSPControlcuflag:20 \leftarrow 1
    endif
    return temp_{15..0}
endfunction satAdd16

Exceptions:
Reserved Instruction, DSP Disabled
ADDQ_S.W

Add Fractional Words

**Format:**  ADDQ_S.W rd, rs, rt

**Purpose:** Add Fractional Words

Addition of two Q31 fractional values to produce a Q31 fractional result, with saturation.

**Description:** rd $\leftarrow$ sat32(rs$^{31..0} + rt^{31..0}$)

The Q31 fractional word in register rt is added to the corresponding fractional word in register rs. The result is then written to the destination register rd.

Signed saturating arithmetic is used, where an overflow is clamped to the largest representable value (0x7FFFFFFF hexadecimal) and an underflow to the smallest representable value (0x80000000 hexadecimal) before being sign-extended and written to the destination register rd.

If the addition results in underflow, overflow, or saturation, a 1 is written to bit 20 in the DSPControl register within the ouflag field.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

```
temp31..0 $\leftarrow$ satAdd32( GPR[rs]$^{31..0}$, GPR[rt]$^{31..0}$ )
GPR[rd]$^{..0}$ $\leftarrow$ temp31..0

function satAdd32( a$^{31..0}$, b$^{31..0}$ )
  temp32..0 $\leftarrow$ ( a$^{31} || a^{31..0}$ ) + ( b$^{31} || b^{31..0}$ )
  if ( temp32 $\neq$ temp31 ) then
    if ( temp32 = 0 ) then
      temp31..0 $\leftarrow$ 0x7FFFFFFF
    else
      temp31..0 $\leftarrow$ 0x80000000
    endif
  endif
  DSPControl[ouflag:20] $\leftarrow$ 1
endfunction
```

**Exceptions:**

Reserved Instruction, DSP Disabled
ADDQH\([\_R].PH\) Add Fractional Halfword Vectors And Shift Right to Halve Results

<table>
<thead>
<tr>
<th>SPECIAL3 011111</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>ADDQH.PH 01000</th>
<th>ADDUH.QB 011000</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL3 011111</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>ADDQH_R.PH 01010</td>
<td>ADDUH.QB 011000</td>
</tr>
</tbody>
</table>

**Format:**
ADDQH\([\_R].PH\)
ADDQH.PH \(rd, rs, rt\)
ADDQH_R.PH \(rd, rs, rt\)

**Purpose:** Add Fractional Halfword Vectors And Shift Right to Halve Results

Element-wise fractional addition of halfword vectors, with a right shift by one bit to halve each result, with optional rounding.

**Description:** \(rd \leftarrow \text{round}((rs_{31..16} + rt_{31..16}) \gg 1) \mid \text{round}((rs_{15..0} + rt_{15..0}) \gg 1)\)

Each element from the two halfword values in register \(rs\) is added to the corresponding halfword element in register \(rt\) to create an interim 17-bit result.

In the non-rounding instruction variant, each interim result is then shifted right by one bit before being written to the corresponding halfword element of destination register \(rd\).

In the rounding version of the instruction, a value of 1 is added at the least-significant bit position of each interim result; the interim result is then right-shifted by one bit and written to the destination register.

This instruction does not modify the DSPControl register.

**Restrictions:**
No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

ADDQH.PH

\[
\begin{align*}
\text{tempB}_{15..0} & \leftarrow \text{rightShift1AddQ16}(\ GPR[rs]_{31..16}, \ GPR[rt]_{31..16}) \\
\text{tempA}_{15..0} & \leftarrow \text{rightShift1AddQ16}(\ GPR[rs]_{15..0}, \ GPR[rt]_{15..0}) \\
\text{GPR}[rd]..0 & \leftarrow \text{tempB}_{15..0} \mid \text{tempA}_{15..0}
\end{align*}
\]

ADDQH_R.PH

\[
\begin{align*}
\text{tempB}_{15..0} & \leftarrow \text{roundRightShift1AddQ16}(\ GPR[rs]_{31..16}, \ GPR[rt]_{31..16}) \\
\text{tempA}_{15..0} & \leftarrow \text{roundRightShift1AddQ16}(\ GPR[rs]_{15..0}, \ GPR[rt]_{15..0}) \\
\text{GPR}[rd]..0 & \leftarrow \text{tempB}_{15..0} \mid \text{tempA}_{15..0}
\end{align*}
\]

function rightShift1AddQ16( a_{15..0} , b_{15..0} )

\[
\text{temp}_{16..0} \leftarrow ((a_{15} || a_{15..0}) + (b_{15} || b_{15..0}))
\]

return temp_{16..1}
endfunction rightShift1AddQ16

function roundRightShift1AddQ16( a_{15..0} , b_{15..0} )

\[
\text{temp}_{16..0} \leftarrow ((a_{15} || a_{15..0}) + (b_{15} || b_{15..0}))
\]

\[
\text{temp}_{16..0} \leftarrow \text{temp}_{16..0} + 1
\]

return temp_{16..1}
endfunction roundRightShift1AddQ16
Exceptions:

Reserved Instruction, DSP Disabled
ADDQH[R].W  Add Fractional Words And Shift Right to Halve Results

Purpose: Add Fractional Words And Shift Right to Halve Results
Fractional addition of word vectors, with a right shift by one bit to halve the result, with optional rounding.

Description: rd ← round(\((rs\_{31..0} + rt\_{31..0}) >> 1\))
The word in register \(rs\) is added to the word in register \(rt\) to create an interim 33-bit result.
In the non-rounding instruction variant, the interim result is then shifted right by one bit before being written to the destination register \(rd\).
In the rounding version of the instruction, a value of 1 is added at the least-significant bit position of the interim result; the interim result is then right-shifted by one bit and written to the destination register.
This instruction does not modify the \(DSPControl\) register.

Restrictions:
No data-dependent exceptions are possible.
The operands must be a value in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:
ADDQH.W
\[
\text{temp}_{31..0} \leftarrow \text{rightShift1AddQ32}( \text{GPR}[rs\_{31..0}], \text{GPR}[rt\_{31..0}] )
\]
\[
\text{GPR}[rd\_{31..0}] \leftarrow \text{temp}_{31..0}
\]
ADDQH_R.W
\[
\text{temp}_{31..0} \leftarrow \text{roundRightShift1AddQ32}( \text{GPR}[rs\_{31..0}], \text{GPR}[rt\_{31..0}] )
\]
\[
\text{GPR}[rd\_{31..0}] \leftarrow \text{temp}_{31..0}
\]

function rightShift1AddQ32( a\_{31..0}, b\_{31..0} )
\[
\text{temp}_{32..1} \leftarrow ((a\_{31} \mid a\_{31..0}) + (b\_{31} \mid b\_{31..0}))
\]
return temp\_{32..1}
endfunction rightShift1AddQ32

function roundRightShift1AddQ32( a\_{31..0}, b\_{31..0} )
\[
\text{temp}_{32..0} \leftarrow ((a\_{31} \mid a\_{31..0}) + (b\_{31} \mid b\_{31..0}))
\]
\[
\text{temp}_{32..0} \leftarrow \text{temp}_{32..0} + 1
\]
return temp\_{32..1}
endfunction roundRightShift1AddQ32

Exceptions:
Reserved Instruction, DSP Disabled
ADDSC Add Signed Word and Set Carry Bit

MIPS® DSP Module for MIPS32™ Architecture, Revision 3.01

Format: ADDSC rd, rs, rt

Purpose: Add Signed Word and Set Carry Bit
Add two signed 32-bit values and set the carry bit in the DSPControl register if the addition generates a carry-out bit.

Description: DSPControl[c],rd ← rs + rt
The 32-bit signed value in register rt is added to the 32-bit signed value in register rs. The result is then written into register rd. The carry bit result out of the addition operation is written to bit 13 (the c field) of the DSPControl register.
This instruction does not modify the ouflag field in the DSPControl register.

Restrictions:
No data-dependent exceptions are possible.

Operation:
\[
\text{temp}_{31..0} \leftarrow (0 \| \text{GPR[rs]}_{31..0}) + (0 \| \text{GPR[rt]}_{31..0}) \\
\text{DSPControl}_{c:13} \leftarrow \text{temp}_{32} \\
\text{GPR[rd]}_{..0} \leftarrow \text{temp}_{31..0}
\]

Exceptions:
Reserved Instruction, DSP Disabled

Programming Notes:
Note that this is really two’s complement (modulo) arithmetic on the two integer values, where the overflow is preserved in architectural state. The ADDWC instruction can be used to do an add using this carry bit. These instructions are provided in the MIPS32 ISA to support 64-bit addition and subtraction using two pairs of 32-bit GPRs to hold each 64-bit value. In the MIPS64 ISA, 64-bit addition and subtraction can be performed directly, without requiring the use of these instructions.
ADDU[_S].PH

Unsigned Add Integer Halfwords

**Format:**

ADDU.PH  
ADDU.PH  rd, rs, rt
ADDU.S.PH  rd, rs, rt

**Purpose:**

Unsigned Add Integer Halfwords

Add two pairs of unsigned integer halfwords, with optional saturation.

**Description:**

\[
\text{rd} \leftarrow \text{sat16}(\text{rs}_{31..16} + \text{rt}_{31..16}) \ || \ \text{sat16}(\text{rs}_{15..0} + \text{rt}_{15..0})
\]

The two unsigned integer halfword elements in register \(rt\) are added to the corresponding unsigned integer halfword elements in register \(rs\).

For the non-saturating version of the instruction, the result modulo 65,536 is written into the corresponding element in register \(rd\).

For the saturating version of the instruction, the addition is performed using unsigned saturating arithmetic. Results that overflow are clamped to the largest representable value (65,535 decimal, 0xFFFF hexadecimal) before being written to the destination register \(rd\).

For either instruction, if any of the individual additions result in overflow or saturation, a 1 is written to bit 20 in the DSPControl register within the ouflag field.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

**ADDU.PH**

\[
\begin{align*}
\text{tempB}_{15..0} & \leftarrow \text{addU16} ( \text{GPR}[\text{rs}]_{31..16} , \text{GPR}[\text{rt}]_{31..16} ) \\
\text{tempA}_{15..0} & \leftarrow \text{addU16} ( \text{GPR}[\text{rs}]_{15..0} , \text{GPR}[\text{rt}]_{15..0} ) \\
\text{GPR}[\text{rd}]_{..0} & \leftarrow \text{tempB}_{15..0} || \text{tempA}_{15..0}
\end{align*}
\]

**ADDU_S.PH**

\[
\begin{align*}
\text{tempB}_{15..0} & \leftarrow \text{satAddU16} ( \text{GPR}[\text{rs}]_{31..16} , \text{GPR}[\text{rt}]_{31..16} ) \\
\text{tempA}_{15..0} & \leftarrow \text{satAddU16} ( \text{GPR}[\text{rs}]_{15..0} , \text{GPR}[\text{rt}]_{15..0} ) \\
\text{GPR}[\text{rd}]_{..0} & \leftarrow \text{tempB}_{15..0} || \text{tempA}_{15..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
### ADDU[S].QB

**Unsigned Add Quad Byte Vectors**

**Format:**

```
ADDU[S].QB
ADDU.QB rd, rs, rt
ADDU.S.QB rd, rs, rt
```

**Purpose:**

Unsigned Add Quad Byte Vectors

Element-wise addition of two vectors of unsigned byte values to produce a vector of unsigned byte results, with optional saturation.

**Description:**

```
rd ← sat8(rs31..24 + rt31..24) || sat8(rs23..16 + rt23..16) || sat8(rs15..8 + rt15..8) || sat8(rs7..0 + rt7..0)
```

The four byte elements in register `rt` are added to the corresponding byte elements in register `rs`.

For the non-saturating version of the instruction, the result modulo 256 is written into the corresponding element in register `rd`.

For the saturating version of the instruction, the addition is performed using unsigned saturating arithmetic. Results that overflow are clamped to the largest representable value (255 decimal, 0xFF hexadecimal) before being written to the destination register `rd`.

For either instruction, if any of the individual additions result in overflow or saturation, a 1 is written to bit 20 in the `DSPControl` register within the `ouflag` field.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

**ADDU.QB:**

```
tempD7..0 ← addU8( GPR[rs]31..24 , GPR[rt]31..24 )
tempC7..0 ← addU8( GPR[rs]23..16 , GPR[rt]23..16 )
tempB7..0 ← addU8( GPR[rs]15..8 , GPR[rt]15..8 )
tempA7..0 ← addU8( GPR[rs]7..0 , GPR[rt]7..0 )
GPR[rd]..0 ← tempD7..0 || tempC7..0 || tempB7..0 || tempA7..0
```

**ADDU_S.QB:**

```
tempD7..0 ← satAddU8( GPR[rs]31..24 , GPR[rt]31..24 )
tempC7..0 ← satAddU8( GPR[rs]23..16 , GPR[rt]23..16 )
tempB7..0 ← satAddU8( GPR[rs]15..8 , GPR[rt]15..8 )
tempA7..0 ← satAddU8( GPR[rs]7..0 , GPR[rt]7..0 )
GPR[rd]..0 ← tempD7..0 || tempC7..0 || tempB7..0 || tempA7..0
```

**Function addU8:**

```
function addU8( a7..0 , b7..0 )
temp8..0 ← ( 0 || a7..0 ) + ( 0 || b7..0 )
if ( temp8 = 1 ) then
    DSPControl.ouflag:20 ← 1
```
function satAddU8( a7..0, b7..0 )
  temp8..0 ← ( 0 || a7..0 ) + ( 0 || b7..0 )
  if ( temp8 = 1 ) then
    temp7..0 ← 0xFF
    DSPControloutflag:20 ← 1
  endif
  return temp7..0
endfunction satAddU8

Exceptions:

Reserved Instruction, DSP Disabled
ADDWC Add Word with Carry Bit

MIPS® DSP Module for MIPS32™ Architecture, Revision 3.01

Format:

\[ \text{ADDWC rm, rs, rt} \]

Purpose:
Add two signed 32-bit values with the carry bit in the DSPControl register.

Description:
\[ rd \leftarrow rs + rt + \text{DSPControl}_c:13 \]

The 32-bit value in register \( rt \) is added to the 32-bit value in register \( rs \) and the carry bit in the DSPControl register. The result is then written to destination register \( rd \).

If the addition results in either overflow or underflow, this instruction writes a 1 to bit 20 in the ouflag field of the DSPControl register.

Restrictions:
No data-dependent exceptions are possible.

Operation:

\[
\begin{align*}
t_{31..0} & \leftarrow ( \text{GPR}[rs]_{31} || \text{GPR}[rs]_{31..0} ) + ( \text{GPR}[rt]_{31} || \text{GPR}[rt]_{31..0} ) + ( 0_{32} || \text{DSPControl}_c:13 ) \\
\text{if } ( \text{temp}_{32} \neq \text{temp}_{31} ) \text{ then} & \\
& \text{DSPControl}_{ouflag:20} \leftarrow 1 \\
\text{endif} \\
\text{GPR}[rd]_{31..0} & \leftarrow \text{temp}_{31..0}
\end{align*}
\]

Exceptions:
Reserved Instruction, DSP Disabled
**ADDUH[_R].QB**  
Unsigned Add Vector Quad-Bytes And Right Shift to Halve Results

<table>
<thead>
<tr>
<th>SPECIAL3 011111</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>ADDUH 00000</th>
<th>ADDUH.QB 011000</th>
<th>ADDUH.R.QB 00010</th>
<th>ADDUH.QB 011000</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 26 25 21 20 16 15 11 10 6 5 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  
ADDUH[_R].QB  
ADDUH.QB rd, rs, rt  
ADDUH_R.QB rd, rs, rt

**Purpose:**  
Unsigned Add Vector Quad-Bytes And Right Shift to Halve Results

Element-wise unsigned addition of unsigned byte vectors, with right shift by one bit to halve each result, with optional rounding.

**Description**  
\[ \text{rd} \leftarrow \text{round}((\text{rs}_{31..24} + \text{rt}_{31..24}) \gg 1) \mid\mid \text{round}((\text{rs}_{23..16} + \text{rt}_{23..16}) \gg 1) \mid\mid \text{round}((\text{rs}_{15..8} + \text{rt}_{15..8}) \gg 1) \mid\mid \text{round}((\text{rs}_{7..0} + \text{rt}_{7..0}) \gg 1) \]

Each element from the four unsigned byte values in register rs is added to the corresponding unsigned byte element in register rt to create an unsigned interim result.

In the non-rounding instruction variant, each interim result is then shifted right by one bit before being written to the corresponding unsigned byte element of destination register rd.

In the rounding version of the instruction, a value of 1 is added at the least-significant bit position of each interim result before being right-shifted by one bit and written to the destination register.

This instruction does not modify the DSPControl register.

**Restrictions:**
No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

ADDUH.QB  
\[ \begin{align*}
\text{tempD}_{7..0} & \leftarrow \text{rightShift1AddU8}( \text{GPR[rs]}_{31..24}, \text{GPR[rt]}_{31..24} ) \\
\text{tempC}_{7..0} & \leftarrow \text{rightShift1AddU8}( \text{GPR[rs]}_{23..16}, \text{GPR[rt]}_{23..16} ) \\
\text{tempB}_{7..0} & \leftarrow \text{rightShift1AddU8}( \text{GPR[rs]}_{15..8}, \text{GPR[rt]}_{15..8} ) \\
\text{tempA}_{7..0} & \leftarrow \text{rightShift1AddU8}( \text{GPR[rs]}_{7..0}, \text{GPR[rt]}_{7..0} ) \\
\text{GPR[rd]}_{7..0} & \leftarrow \text{tempD}_{7..0} \mid\mid \text{tempC}_{7..0} \mid\mid \text{tempB}_{7..0} \mid\mid \text{tempA}_{7..0}
\end{align*} \]

ADDUH_R.QB  
\[ \begin{align*}
\text{tempD}_{7..0} & \leftarrow \text{roundRightShift1AddU8}( \text{GPR[rs]}_{31..24}, \text{GPR[rt]}_{31..24} ) \\
\text{tempC}_{7..0} & \leftarrow \text{roundRightShift1AddU8}( \text{GPR[rs]}_{23..16}, \text{GPR[rt]}_{23..16} ) \\
\text{tempB}_{7..0} & \leftarrow \text{roundRightShift1AddU8}( \text{GPR[rs]}_{15..8}, \text{GPR[rt]}_{15..8} ) \\
\text{tempA}_{7..0} & \leftarrow \text{roundRightShift1AddU8}( \text{GPR[rs]}_{7..0}, \text{GPR[rt]}_{7..0} ) \\
\text{GPR[rd]}_{7..0} & \leftarrow \text{tempD}_{7..0} \mid\mid \text{tempC}_{7..0} \mid\mid \text{tempB}_{7..0} \mid\mid \text{tempA}_{7..0}
\end{align*} \]

```plaintext```
function rightShift1AddU8( a7..0 , b7..0 )
    temp8..0 \leftarrow ( ( 0 \mid a7..0 ) + ( 0 \mid b7..0 ) )
    return temp8..1
endfunction rightShift1AddU8
```

MIPS® DSP Module for MIPS32™ Architecture, Revision 3.01
function roundRightShift1AddU8( a[7..0], b[7..0] )
    temp[8..0] ← (( 0 || a[7..0] ) + ( 0 || b[7..0] ))
    temp[8..0] ← temp[8..0] + 1
    return temp[8..1]
endfunction roundRightShift1AddU8

Exceptions:
Reserved Instruction, DSP Disabled
APPEND Left Shift and Append Bits to the LSB

<table>
<thead>
<tr>
<th>SPECIAL3</th>
<th>rs</th>
<th>rt</th>
<th>sa</th>
<th>APPEND 00000</th>
<th>APPEND 110001</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Format: **APPEND** rt, rs, sa

**Purpose:** Left Shift and Append Bits to the LSB

Shift a general-purpose register left, inserting bits from the another GPR into the bit positions emptied by the shift.

**Description:**

\[
rt \leftarrow (rt_{31..0} \ll sa_{4..0}) \mid \mid rs_{sa-1..0}
\]

The 32-bit value in register \( rt \) is left-shifted by the specified shift amount \( sa \), and \( sa \) bits from the least-significant positions of the \( rs \) register are inserted into the bit positions in \( rt \) emptied by the shift. The 32-bit shifted value is written to destination register \( rt \).

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

\[
\begin{align*}
\text{if } (sa_{4..0} = 0) \text{ then} & \\
& \text{temp}_{31..0} \leftarrow \text{GPR[rt]}_{31..0} \\
\text{else} & \\
& \text{temp}_{31..0} \leftarrow (\text{GPR[rt]}_{31-sa..0} \mid \mid \text{GPR[rs]}_{sa-1..0}) \\
\text{endif} & \\
\text{GPR[rt]}_{..0} = \text{temp}_{31..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
BALIGN

**Byte Align Contents from Two Registers**

**Format:**
BALIGN rt, rs, bp

**Purpose:**
Byte Align Contents from Two Registers
Create a word result by combining a specified number of bytes from each of two source registers.

**Description:**
rt ← (rt << 8*bp) || (rs >> 8*(4-bp))
The 32-bit word in register rt is left-shifted as a 32-bit value by bp byte positions, and the right-most word in register rs is right-shifted as a 32-bit value by (4-bp) byte positions. The shifted values are then or-ed together to create a 32-bit result that is written to destination register rt.

The argument bp is provided by the instruction, and is interpreted as an unsigned two-bit integer taking values between zero and three.

**Restrictions:**
No data-dependent exceptions are possible.

**Operation:**
if (bp1..0 = 0) or (bp1..0 = 2) then
  GPR[rt]..0 ← UNPREDICTABLE
else
  temp31..0 ← ( GPR[rt]31..0 << (8*bp1..0) ) || ( GPR[rs]31..0 >> (8*(4-bp1..0)) )
  GPR[rt]..0 = temp31..0
endif

**Exceptions:**
Reserved Instruction, DSP Disabled
**BITREV**

Bit-Reverse Halfword

**Format:** BITREV rd, rt

**Purpose:** Bit-Reverse Halfword

To reverse the order of the bits of the least-significant halfword in the specified register.

**Description:**

\[ \text{rd} \leftarrow \text{rt}_{0..15} \]

The right-most halfword value in register \( rt \) is bit-reversed into the right-most halfword position in the destination register \( rd \). The most-significant bits of the destination register are zero-filled.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

\[
\begin{align*}
temp_{15..0} & \leftarrow \text{GPR}[rt]_{0..15} \\
\text{GPR}[rd]_{..0} & \leftarrow 0 || temp_{15..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
BPOSGE32 Branch on Greater Than or Equal To Value 32 in DSPControl Pos Field

Format:  BPOSGE32 offset

Purpose:  Branch on Greater Than or Equal To Value 32 in DSPControl Pos Field
Perform a PC-relative branch if the value of the pos field in the DSPControl register is greater than or equal to 32.

Description:  if (DSPControl.pos:..0 >= 32) then goto PC+offset
First, the offset argument is left-shifted by two bits to form an 18-bit signed integer value. This value is added to the address of the instruction immediately following the branch to form a target branch address. Then, if the value of the pos field of the DSPControl register is greater than or equal to 32, the branch is taken and execution begins from the target address after the instruction in the branch delay slot has been executed.

Restrictions:
Pre-Release 6: Processor operation is UNPREDICTABLE if a control transfer instruction (CTI), specifically a branch, jump, NAL (Release 6), ERET, ERETNC (Release 5), DERET, WAIT, or PAUSE (Release 2) instruction is placed in the delay slot of a branch or jump.

Release 6: If a control transfer instruction (CTI) is executed in the delay slot of a branch or jump, Release 6 implementations are required to signal a Reserved Instruction Exception.

Availability:
None.

Operation:
I:  
\[ \text{se\_offset}\text{GPRLEN..0} \leftarrow (\text{offset}\text{GPRLEN-18} || \text{offset}\text{15..0} || 0^2 \text{branch\_condition} ) \]

I+1:  
\[ \text{PCGPRLEN..0} \leftarrow \text{PCGPRLEN..0} + \text{se\_offset}\text{GPRLEN..0} \]

Exceptions:
Reserved Instruction, DSP Disabled

Programming Notes:
With the 18-bit signed instruction offset, the conditional branch range is ±128 Kbytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside of this range.
Branch on Greater Than or Equal To Value 32 in DSPControl Pos Field

Perform a PC-relative branch if the value of the pos field in the DSPControl register is greater than or equal to 32.

Description:

if (DSPControl\_pos:..0 \(\geq\) 32) then goto PC+\(se\_offset\)

First, the \(se\_offset\) argument is left-shifted by two bits to form an 18-bit signed integer value. This value is added to the address of the instruction immediately following the branch to form a target branch address. Then, if the value of the pos field of the DSPControl register is greater than or equal to 32, the branch is taken and execution begins from the target address.

Restrictions:

If a control transfer instruction (CTI) is executed in the forbidden slot of a branch or jump, Release 6 implementations are required to signal a Reserved Instruction Exception. A CTI is considered to be one of the following instructions: branch, jump, NAL (Release 6), ERET, ER\(\text{ETN}\)C (Release 5), DER\(\text{ET}\), WAIT, or PAUSE (Release 2). An instruction is in the forbidden slot if it is the instruction following the branch.

Availability:

This instruction is introduced by and required as of Revision 3 of the DSP Module.

Operation:

\[
\begin{align*}
I: & \quad se\_offset\_GPR\_LEN..0 \leftarrow ( offset\_15 )^{GPR\_LEN-18} \ || \ offset\_15..0 \ || \ 0^2 \\
branch\_condition & \quad \leftarrow ( DSPControl\_pos:..0 \geq 32 \ ? \ 1 \ : \ 0 ) \\
I+1: & \quad \text{if ( branch\_condition = 1 ) then} \\
& \quad PC\_GPR\_LEN..0 \leftarrow PC\_GPR\_LEN..0 + se\_offset\_GPR\_LEN..0 \\
& \quad \text{endif}
\end{align*}
\]

Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is \(\pm 128\) Kbytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside of this range.
**CMP.cond.PH**

**Compare Vectors of Signed Integer Halfword Values**

**Format:**

<table>
<thead>
<tr>
<th>SPECIAL3</th>
<th>011111</th>
<th>rs</th>
<th>rt</th>
<th>0</th>
<th>CMPEQ.PH</th>
<th>01000</th>
<th>CMPU.EQ.QB</th>
<th>010001</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs</td>
<td>rt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Purpose:** Compare Vectors of Signed Integer Halfword Values

Perform an element-wise comparison of two vectors of two signed integer halfwords, recording the results of the comparison in condition code bits.

**Description:**

\[
\text{DSPControl}_{\text{cccond}:25..24} \leftarrow (\text{rs}_{31..16} \text{ cond } \text{rt}_{31..16}) \mid \mid (\text{rs}_{15..0} \text{ cond } \text{rt}_{15..0})
\]

The two signed integer halfword elements in register \( rs \) are compared with the corresponding signed integer halfword element in register \( rt \). The two 1-bit boolean comparison results are written to bits 24 and 25 of the \( \text{DSPControl} \) register’s -bit condition code field. The values of the remaining condition code bits (bits 26 through of the \( \text{DSPControl} \) register) are **UNPREDICTABLE**.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

**CMP.EQ.PH**

\[
\begin{align*}
\text{ccB} & \leftarrow \text{GPR}[\text{rs}]_{31..16} \text{ EQ GPR}[\text{rt}]_{31..16} \\
\text{ccA} & \leftarrow \text{GPR}[\text{rs}]_{15..0} \text{ EQ GPR}[\text{rt}]_{15..0} \\
\text{DSPControl}_{\text{cccond}:25..24} & \leftarrow \text{ccB} \mid \mid \text{ccA} \\
\text{DSPControl}_{\text{cccond}:26} & \leftarrow \text{UNPREDICTABLE}
\end{align*}
\]

**CMP.LT.PH**

\[
\begin{align*}
\text{ccB} & \leftarrow \text{GPR}[\text{rs}]_{31..16} \text{ LT GPR}[\text{rt}]_{31..16} \\
\text{ccA} & \leftarrow \text{GPR}[\text{rs}]_{15..0} \text{ LT GPR}[\text{rt}]_{15..0} \\
\text{DSPControl}_{\text{cccond}:25..24} & \leftarrow \text{ccB} \mid \mid \text{ccA} \\
\text{DSPControl}_{\text{cccond}:26} & \leftarrow \text{UNPREDICTABLE}
\end{align*}
\]

**CMP.LE.PH**

\[
\begin{align*}
\text{ccB} & \leftarrow \text{GPR}[\text{rs}]_{31..16} \text{ LE GPR}[\text{rt}]_{31..16} \\
\text{ccA} & \leftarrow \text{GPR}[\text{rs}]_{15..0} \text{ LE GPR}[\text{rt}]_{15..0} \\
\text{DSPControl}_{\text{cccond}:25..24} & \leftarrow \text{ccB} \mid \mid \text{ccA} \\
\text{DSPControl}_{\text{cccond}:26} & \leftarrow \text{UNPREDICTABLE}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
**CMPGDU.cond.QB**  
**Compare Unsigned Vector of Four Bytes and Write Result to GPR and DSPControl**

<table>
<thead>
<tr>
<th>SPECIAL3</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>CMPGDU.EQ.QB</th>
<th>11000</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111</td>
<td></td>
<td></td>
<td></td>
<td>CMPGDU.EQ.QB</td>
<td>010001</td>
</tr>
<tr>
<td>SPECIAL3</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>CMPGDU.LT.QB</td>
<td>11001</td>
</tr>
<tr>
<td>011111</td>
<td></td>
<td></td>
<td></td>
<td>CMPGDU.EQ.QB</td>
<td>010001</td>
</tr>
<tr>
<td>SPECIAL3</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>CMPGDU.LE.QB</td>
<td>11010</td>
</tr>
<tr>
<td>011111</td>
<td></td>
<td></td>
<td></td>
<td>CMPGDU.EQ.QB</td>
<td>010001</td>
</tr>
</tbody>
</table>

**Format:**  
CMPGDU.cond.QB  
CMPGDU.EQ.QB  
rd, rs, rt  
MIPS DSP-R2  
CMPGDU.LT.QB  
rd, rs, rt  
MIPS DSP-R2  
CMPGDU.LE.QB  
rd, rs, rt  
MIPS DSP-R2

**Purpose:** Compare Unsigned Vector of Four Bytes and Write Result to GPR and DSPControl  
Compare two vectors of four unsigned bytes each, recording the comparison results in condition code bits that are written to both the specified destination GPR and the condition code bits in the DSPControl register.

**Description:**  
DSPControl[cccond]27..24 ← (rs31..24 cond rt31..24) || (rs23..16 cond rt23..16) || (rs15..8 cond rt15..8) || (rs7..0 cond rt7..0);  
rd ← 0(GPRLEN-4) || DSPControl[cccond]27..24

Each of the unsigned byte elements in register rs are compared with the corresponding unsigned byte elements in register rt. The four 1-bit boolean comparison results are written to the four least-significant bits of destination register rd and to bits 24 through 27 of the DSPControl register’s -bit condition code field. The remaining bits in destination register rd are set to zero.

**Restrictions:**  
No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

**CMPGDU.EQ.QB**

ccD ← GPR[rs]31..24 EQ GPR[rt]31..24
ccC ← GPR[rs]23..16 EQ GPR[rt]23..16
ccB ← GPR[rs]15..8 EQ GPR[rt]15..8
ccA ← GPR[rs]7..0 EQ GPR[rt]7..0
DSPControlcc:27..24 ← ccD || ccC || ccB || ccA
GPR[rd]..0 ← 0(GPRLEN-4) || ccD || ccC || ccB || ccA

**CMPGDU.LT.QB**

ccD ← GPR[rs]31..24 LT GPR[rt]31..24
ccC ← GPR[rs]23..16 LT GPR[rt]23..16
ccB ← GPR[rs]15..8 LT GPR[rt]15..8
ccA ← GPR[rs]7..0 LT GPR[rt]7..0
DSPControlcc:27..24 ← ccD || ccC || ccB || ccA
GPR[rd]..0 ← 0(GPRLEN-4) || ccD || ccC || ccB || ccA

**CMPGDU.LE.QB**

ccD ← GPR[rs]31..24 LE GPR[rt]31..24
ccC ← GPR[rs]23..16 LE GPR[rt]23..16
CMPGDU.cond.QB  Compare Unsigned Vector of Four Bytes and Write Result to GPR and DSPControl

\[
\begin{align*}
\text{ccB} & \leftarrow \text{GPR}\{rs\}_{15..8} \text{ LE } \text{GPR}\{rt\}_{15..8} \\
\text{ccA} & \leftarrow \text{GPR}\{rs\}_{7..0} \text{ LE } \text{GPR}\{rt\}_{7..0} \\
\text{DSPControl}_{cc:27..24} & \leftarrow \text{ccD} \mid \text{ccC} \mid \text{ccB} \mid \text{ccA} \\
\text{GPR}\{rd\}_{..0} & \leftarrow 0^{(\text{GPRLEN}-4)} \mid \text{ccD} \mid \text{ccC} \mid \text{ccB} \mid \text{ccA}
\end{align*}
\]

Exceptions:
Reserved Instruction, DSP Disabled
**CMPGU.cond.QB**  
**Compare Vectors of Unsigned Byte Values and Write Results to a GPR**

- **Format:**
  - `CMPGU.cond.QB`  
    - `CMPGU.EQ.QB` rd, rs, rt  
    - `CMPGU.LT.QB` rd, rs, rt  
    - `CMPGU.LE.QB` rd, rs, rt  

- **Purpose:** Compare Vectors of Unsigned Byte Values and Write Results to a GPR

Perform an element-wise comparison of two vectors of unsigned bytes, recording the results of the comparison in condition code bits that are written to the specified GPR.

- **Description:**
  - `rd ← (rs\_31..24 \ cond \ rt\_31..24) \ || \ (rs\_23..16 \ cond \ rt\_23..16) \ || \ (rs\_15..8 \ cond \ rt\_15..8) \ || \ (rs\_7..0 \ cond \ rt\_7..0)`

Each of the unsigned byte elements in register `rs` are compared with the corresponding unsigned byte elements in register `rt`. The four 1-bit boolean comparison results are written to the four least-significant bits of destination register `rd`. The remaining bits in `rd` are set to zero.

- **Restrictions:**
  - No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

- **Operation:**
  - `CMPGU.EQ.QB`
    - `ccD ← GPR[rs]_{31..24} EQ GPR[rt]_{31..24}`
    - `ccC ← GPR[rs]_{23..16} EQ GPR[rt]_{23..16}`
    - `ccB ← GPR[rs]_{15..8} EQ GPR[rt]_{15..8}`
    - `ccA ← GPR[rs]_{7..0} EQ GPR[rt]_{7..0}`
    - `GPR[rd]..0 ← 0(GPRLEN-4) || ccD || ccC || ccB || ccA`

  - `CMPGU.LT.QB`
    - `ccD ← GPR[rs]_{31..24} LT GPR[rt]_{31..24}`
    - `ccC ← GPR[rs]_{23..16} LT GPR[rt]_{23..16}`
    - `ccB ← GPR[rs]_{15..8} LT GPR[rt]_{15..8}`
    - `ccA ← GPR[rs]_{7..0} LT GPR[rt]_{7..0}`
    - `GPR[rd]..0 ← 0(GPRLEN-4) || ccD || ccC || ccB || ccA`

  - `CMPGU.LE.QB`
    - `ccD ← GPR[rs]_{31..24} LE GPR[rt]_{31..24}`
    - `ccC ← GPR[rs]_{23..16} LE GPR[rt]_{23..16}`
    - `ccB ← GPR[rs]_{15..8} LE GPR[rt]_{15..8}`
    - `ccA ← GPR[rs]_{7..0} LE GPR[rt]_{7..0}`
    - `GPR[rd]..0 ← 0(GPRLEN-4) || ccD || ccC || ccB || ccA`
Exceptions:

Reserved Instruction, DSP Disabled
CMPU.cond.QB

CMPU.cond.QB

Perform an element-wise comparison of two vectors of unsigned bytes, recording the results of the comparison in condition code bits.

Description:

Each of the unsigned byte elements in register \( rs \) are compared with the corresponding unsigned byte elements in register \( rt \). The four 1-bit boolean comparison results are written to bits 24 through 27 of the DSPControl register's 32-bit condition code field.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

\[
\begin{align*}
\text{CMPU.EQ.QB} & \quad rs, \ rt \\
\text{CMPU.LT.QB} & \quad rs, \ rt \\
\text{CMPU.LE.QB} & \quad rs, \ rt
\end{align*}
\]
Exceptions:
Reserved Instruction, DSP Disabled
**DPA.W.PH**

**Dot Product with Accumulate on Vector Integer Halfword Elements**

**Format:**

DPA.W.PH ac, rs, rt

**Purpose:**

Dot Product with Accumulate on Vector Integer Halfword Elements

Generate the dot-product of two integer halfword vector elements using full-size intermediate products and then accumulate into the specified accumulator register.

**Description:**

\[
ac \leftarrow ac + ((rs_{31..16} \times rt_{31..16}) + (rs_{15..0} \times rt_{15..0}))
\]

Each of the two halfword integer values from register \(rt\) is multiplied with the corresponding halfword element from register \(rs\) to create two integer word results. These two products are summed to generate a dot-product result, which is then accumulated into the specified 64-bit HI/LO accumulator, creating a 64-bit integer result.

The value of \(ac\) selects an accumulator numbered from 0 to 3. When \(ac=0\), this refers to the original HI/LO register pair of the architecture.

This instruction does not set any bits of the ouflag field in the DSPControl register.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the result is UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

\[
\begin{align*}
tempB_{31..0} & \leftarrow (GPR[rs]_{31..16} \times GPR[rt]_{31..16}) \\
tempA_{31..0} & \leftarrow (GPR[rs]_{15..0} \times GPR[rt]_{15..0}) \\
dotp_{32..0} & \leftarrow (tempB_{31} || tempB_{31..0}) + (tempA_{31} || tempA_{31..0}) \\
acc_{63..0} & \leftarrow (HI[ac]_{31..0} || LO[ac]_{31..0}) + (dotp_{32})^{31} || dotp_{32..0}) \\
& \quad + (HI[ac]..0 || LO[ac]..0) \leftarrow acc_{63..32} || acc_{31..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
DPAQ_S.W.PH Dot Product with Accumulation on Fractional Halfword Elements

Format:  DPAQ_S.W.PH ac, rs, rt

MIPS DSP

Purpose:  Dot Product with Accumulation on Fractional Halfword Elements

Element-wise multiplication of two vectors of fractional halfword elements and accumulation of the accumulated 32-bit intermediate products into the specified 64-bit accumulator register, with saturation.

Description:

\[
ac \leftarrow ac + (\text{sat32}(rs_{31..16} \times rt_{31..16}) + \text{sat32}(rs_{15..0} \times rt_{15..0}))
\]

Each of the two Q15 fractional word values from registers rt and rs are multiplied together, and the results left-shifted by one bit position to generate two Q31 fractional format intermediate products. If both multiplicands for either of the multiplications are equal to -1.0 (0x8000 hexadecimal), the resulting intermediate product is saturated to the maximum positive Q31 fractional value (0x7FFFFFFF hexadecimal).

The two intermediate products are then sign-extended and summed to generate a 64-bit, Q32.31 fractional format dot-product result that is accumulated into the specified 64-bit HI/LO accumulator to produce a final Q32.31 fractional result.

The value of ac can range from 0 to 3; a value of 0 refers to the original HI/LO register pair of the architecture.

If saturation occurs as a result of a halfword multiplication, a 1 is written to one of bits 16 through 19 of the DSPControl register, within the ouflag field. The value of ac determines which of these bits is set: bit 16 corresponds to ac0, bit 17 to ac1, bit 18 to ac2, and bit 19 to ac3.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the result is UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

\[
tempB_{31..0} \leftarrow \text{multiplyQ15Q15}(ac, \text{GPR}[rs]_{31..16}, \text{GPR}[rt]_{31..16})
\]
\[
tempA_{31..0} \leftarrow \text{multiplyQ15Q15}(ac, \text{GPR}[rs]_{15..0}, \text{GPR}[rt]_{15..0})
\]
\[
dotp_{63..0} \leftarrow (\text{tempB}_{31})^{32} || \text{tempB}_{31..0} + (\text{tempA}_{31})^{32} || \text{tempA}_{31..0}
\]
\[
tempC_{63..0} \leftarrow (\text{HI}[ac]_{31..0} || \text{LO}[ac]_{31..0}) + \text{dotp}_{63..0}
\]
\[
(\text{HI}[ac]_{.0} || \text{LO}[ac]_{.0}) \leftarrow \text{tempC}_{63..32} || \text{tempC}_{31..0}
\]

function multiplyQ15Q15( acc_{15..0}, a_{15..0}, b_{15..0} )

\[
\text{if} \ ( a_{15..0} = 0x8000 \text{ and } b_{15..0} = 0x8000 ) \text{ then}
\]
\[
\text{temp}_{31..0} \leftarrow 0x7FFFFFFF
\]
\[
\text{DSPControl}._{ouflag:16 + acc} \leftarrow 1
\]

else
\[
\text{temp}_{31..0} \leftarrow (a_{15..0} \times b_{15..0}) << 1
\]
endif

return temp_{31..0}
endfunction multiplyQ15Q15

Exceptions:

Reserved Instruction, DSP Disabled
**Format:** DPAQ_SA.L.W ac, rs, rt

**Purpose:** Dot Product with Accumulate on Fractional Word Element

Multiplication of two fractional word elements, accumulating the product to the specified 64-bit accumulator register, with saturation.

**Description:**
\[
ac \leftarrow \text{s}\text{at64} (ac + \text{s}\text{at32}(rs_{31..0} \times rt_{31..0}))
\]

The two right-most Q31 fractional word values from registers \(rt\) and \(rs\) are multiplied together and the result left-shifted by one bit position to generate a 64-bit, Q63 fractional format intermediate product. If both multiplicands are equal to -1.0 (0x80000000 hexadecimal), the intermediate product is saturated to the maximum positive Q63 fractional value (0x7FFFFFFFFFFFFFFF hexadecimal).

The intermediate product is then added to the specified 64-bit HI/LO accumulator, creating a Q63 fractional result. If the accumulation results in overflow or underflow, the accumulator is saturated to either the maximum positive or minimum negative Q63 fractional value (0x8000000000000000 hexadecimal), respectively.

The value of \(ac\) can range from 0 to 3; a value of 0 refers to the original HI/LO register pair of the architecture.

If saturation occurs, a 1 is written to one of bits 16 through 19 of the DSPControl register, within the ouflag field. The value of \(ac\) determines which of these bits is set: bit 16 corresponds to \(ac0\), bit 17 to \(ac1\), bit 18 to \(ac2\), and bit 19 to \(ac3\).

**Restrictions:**
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the result is UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

```plaintext
dotp63..0 ← multiplyQ31Q31( ac, GPR[rs]31..0, GPR[rt]31..0 )
temp64..0 ← HI[ac]31 || HI[ac]31..0 || LO[ac]31..0
temp64..0 ← temp64..0 + dotp63..0
if ( temp64 ≠ temp63 ) then
    if ( temp64 = 1 ) then
        temp63..0 ← 0x8000000000000000
    else
        temp63..0 ← 0x7FFFFFFFFFFFFFFF
    endif
endif
DSPControl.ouflag:16+ac ← 1
endif
( HI[ac]..0 || LO[ac]..0 ) ← temp63..32 || temp31..0

function multiplyQ31Q31( acc1..0, a31..0, b31..0 )
    if (( a31..0 = 0x80000000 ) and ( b31..0 = 0x80000000 )) then
        temp63..0 ← 0x7FFFFFFFFFFFFFFF
        DSPControl.ouflag:16+acc ← 1
    else
        temp63..0 ← ( a31..0 * b31..0 ) << 1
    endif
    return temp63..0
```

### MIPS® DSP Module for MIPS32™ Architecture, Revision 3.01
endfunction multiplyQ31Q31

Exceptions:
Reserved Instruction, DSP Disabled
**DPAQX_S.W.PH Cross Dot Product with Accumulation on Fractional Halfword Elements**

**Format:**  
```
DPAQX_S.W.PH ac, rs, rt
```

**MIPSDSP-R2**

**Purpose:** Cross Dot Product with Accumulation on Fractional Halfword Elements  
Element-wise cross multiplication of two vectors of fractional halfword elements and accumulation of the 32-bit intermediate products into the specified 64-bit accumulator register, with saturation.

**Description:**  
```
ac ← ac + (sat32(rs31..16 * rt15..0) + sat32(rs15..0 * rt31..16))
```

The left Q15 fractional word value from registers \( rt \) is multiplied with the right halfword element from register \( rs \) and the result left-shifted by one bit position to generate a Q31 fractional format intermediate product. Similarly, the right Q15 fractional word value from registers \( rt \) is multiplied with the left halfword element from register \( rs \) and the result left-shifted by one bit position to generate a Q31 fractional format intermediate product. If both multiplicands for either of the multiplications are equal to -1.0 (0x8000 hexadecimal), the resulting intermediate product is saturated to the maximum positive Q31 fractional value (0x7FFFFFFF hexadecimal).

The two intermediate products are then sign-extended and summed to generate a 64-bit, Q32.31 fractional format dot-product result that is accumulated into the specified 64-bit \( HI/LO \) accumulator to produce a final Q32.31 fractional result.

The value of \( ac \) can range from 0 to 3; a value of 0 refers to the original \( HI/LO \) register pair of the architecture.

If saturation occurs as a result of a halfword multiplication, a 1 is written to one of bits 16 through 19 of the \( DSPControl \) register, within the ouflag field. The value of \( ac \) determines which of these bits is set: bit 16 corresponds to \( ac0 \), bit 17 to \( ac1 \), bit 18 to \( ac2 \), and bit 19 to \( ac3 \).

**Restrictions:**  
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the result is **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

```
tempB31..0 ← multiplyQ15Q15( ac, GPR[rs]31..16, GPR[rt]15..0 )
tempA31..0 ← multiplyQ15Q15( ac, GPR[rs]15..0, GPR[rt]31..16 )
dotp63..0 ← ( (tempB31)32 || tempB31..0 ) + ( (tempA31)32 || tempA31..0 )
tempC63..0 ← ( HI[ac]31..0 || LO[ac]31..0 ) + dotp63..0
( HI[ac]..0 || LO[ac]..0 ) ← tempC63..32 || tempC31..0
```

```
function multiplyQ15Q15( acc1..0, a15..0, b15..0 )
if ( a15..0 = 0x8000 ) and ( b15..0 = 0x8000 ) then
  temp31..0 ← 0x7FFFFFFF
  DSPControlouflag:16+acc ← 1
else
  temp31..0 ← ( a15..0 * b15..0 ) << 1
endif
return temp31..0
endfunction multiplyQ15Q15
```

**Exceptions:**

Reserved Instruction, DSP Disabled
Cross Dot Product with Accumulation on Fractional Halfword Elements

Format:  DPAQX_SA.W.PH ac, rs, rt

Purpose:  Cross Dot Product with Accumulation on Fractional Halfword Elements

Element-wise cross multiplication of two vectors of fractional halfword elements and accumulation of the 32-bit intermediate products into the specified 64-bit accumulator register, with saturation of the accumulator.

Description:  ac ← sat32(ac + (sat32(rs31..16 * rt15..0) + sat32(rs15..0 * rt31..16)))

The left Q15 fractional word value from registers rt is multiplied with the right halfword element from register rs and the result left-shifted by one bit position to generate a Q31 fractional format intermediate product. Similarly, the right Q15 fractional word value from registers rt is multiplied with the left halfword element from register rs and the result left-shifted by one bit position to generate a Q31 fractional format intermediate product. If both multiplicands for either of the multiplications are equal to -1.0 (0x8000 hexadecimal), the resulting intermediate product is saturated to the maximum positive Q31 fractional value (0x7FFFFFFF hexadecimal).

The two intermediate products are then sign-extended and summed to generate a 64-bit, Q32.31 fractional format dot-product result that is accumulated into the specified 64-bit HI/LO accumulator to produce a Q32.31 fractional result. If this result is larger than or equal to +1.0, or smaller than -1.0, it is saturated to the Q31 range.

The value of ac can range from 0 to 3; a value of 0 refers to the original HI/LO register pair of the architecture.

If saturation occurs as a result of halfword multiplication or accumulation, a 1 is written to one of bits 16 through 19 of the DSPControl register, within the ouflag field. The value of ac determines which of these bits is set: bit 16 corresponds to ac0, bit 17 to ac1, bit 18 to ac2, and bit 19 to ac3.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the result is UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

```
tempB31..0 ← multiplyQ15Q15( ac, GPR[rs]31..16, GPR[rt]15..0 )
tempA31..0 ← multiplyQ15Q15( ac, GPR[rs]15..0, GPR[rt]31..16 )
dotp63..0 ← ( (tempB31)32 || tempB31..0 ) + ( (tempA31)32 || tempA31..0 )
tempC63..0 ← ( HI[ac]31..0 || LO[ac]31..0 ) + dotp63..0
if ( tempC63 = 0 ) and ( tempC62..31 ≠ 0 ) then
    tempC63..0 = 032 || 0x7FFFFFFF
    DSPControl.ouflag:16+acc ← 1
endif
if ( tempC63 = 1 ) and ( tempC62..31 ≠ 132 ) then
    tempC63..0 = 132 || 0x80000000
    DSPControl.ouflag:16+acc ← 1
endif
( HI[ac]..0 || LO[ac]..0 ) ← tempC63..32 || tempC31..0

function multiplyQ15Q15( acc1..0, a15..0, b15..0 )
    if ( a15..0 = 0x8000 ) and ( b15..0 = 0x8000 ) then
        temp31..0 ← 0x7FFFFFFF
        DSPControl.ouflag:16+acc ← 1
```
else
    temp31..0 ← ( a15..0 * b15..0 ) << 1
endif
return temp31..0
endfunction multiplyQ15Q15

Exceptions:
Reserved Instruction, DSP Disabled
Format: \texttt{DPAU.H.QBL ac, rs, rt} \\

**Purpose:** Dot Product with Accumulate on Vector Unsigned Byte Elements \\
Element-wise multiplication of the two left-most elements of the four elements of each of two vectors of unsigned bytes, accumulating the sum of the products into the specified 64-bit accumulator register.

**Description:** \\
\[
\text{ac} \leftarrow \text{ac} + \text{zero\_extend}( (\text{rs31..24} \times \text{rt31..24}) + (\text{rs23..16} \times \text{rt23..16}) )
\]

The two left-most elements of the four unsigned byte elements of each of registers \( rt \) and \( rs \) are multiplied together using unsigned arithmetic to generate two 16-bit unsigned intermediate products. The intermediate products are then zero-extended to 64 bits and accumulated into the specified 64-bit HI/LO accumulator.

The value of \( ac \) can range from 0 to 3; a value of 0 refers to the original HI/LO register pair of the architecture.

This instruction does not set any bits in the ouflag field in the \texttt{DSPControl} register.

**Restrictions:** \\
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the result is \texttt{UNPREDICTABLE} and the values of the operand vectors become \texttt{UNPREDICTABLE}.

**Operation:** \\
\[
\begin{align*}
\text{tempB15..0} & \leftarrow \text{multiplyU8U8}( \text{GPR[rs]31..24}, \text{GPR[rt]31..24} ) \\
\text{tempA15..0} & \leftarrow \text{multiplyU8U8}( \text{GPR[rs]23..16}, \text{GPR[rt]23..16} ) \\
\text{dotp63..0} & \leftarrow ( 0^{48} || \text{tempB15..0} ) + ( 0^{48} || \text{tempA15..0} ) \\
\text{tempC63..0} & \leftarrow ( \text{HI[ac]31..0} || \text{LO[ac]31..0} ) + \text{dotp63..0} \\
& \quad ( \text{HI[ac]..0} || \text{LO[ac]..0} ) \leftarrow \text{tempC63..32} || \text{tempC31..0} \\
\end{align*}
\]

function \texttt{multiplyU8U8}( a7..0, b7..0 ) \\
\[
\begin{align*}
\text{temp17..0} & \leftarrow ( 0 || a7..0 ) \times ( 0 || b7..0 ) \\
\text{return temp15..0}
\end{align*}
\]
endfunction \texttt{multiplyU8U8}

**Exceptions:** \\
Reserved Instruction, DSP Disabled
DPAU.H.QBR  Dot Product with Accumulate on Vector Unsigned Byte Elements

**Format:**  
DPAU.H.QBR ac, rs, rt

**Purpose:**  
Dot Product with Accumulate on Vector Unsigned Byte Elements

Element-wise multiplication of the two right-most elements of the four elements of each of two vectors of unsigned bytes, accumulating the sum of the products into the specified 64-bit accumulator register.

**Description:**  
\[ ac \leftarrow ac + \text{zero}_{-}\text{extend}((rs_{15..8} \times rt_{15..8}) + (rs_{7..0} \times rt_{7..0})) \]

The two right-most elements of the four unsigned byte elements of each of registers \( rt \) and \( rs \) are multiplied together using unsigned arithmetic to generate two 16-bit unsigned intermediate products. The intermediate products are then zero-extended to 64 bits and accumulated into the specified 64-bit \( HI/LO \) accumulator.

The value of \( ac \) can range from 0 to 3; a value of 0 refers to the original \( HI/LO \) register pair of the architecture.

This instruction does not set any bits in the ouflag field in the DSPControl register.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the result is UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

\[
\begin{align*}
\text{tempB}_{15..0} & \leftarrow \text{multiplyU8U8}( \text{GPR}[rs]_{15..8}, \text{GPR}[rs]_{15..8} ) \\
\text{tempA}_{15..0} & \leftarrow \text{multiplyU8U8}( \text{GPR}[rs]_{7..0}, \text{GPR}[rs]_{7..0} ) \\
\text{dotp}_{63..0} & \leftarrow ( 0^{48} || \text{tempB}_{15..0} ) + ( 0^{48} || \text{tempA}_{15..0} ) \\
\text{tempC}_{63..0} & \leftarrow ( \text{HI}[ac]_{31..0} || \text{LO}[ac]_{31..0} ) + \text{dotp}_{63..0} \\
& \quad \text{( HI}[ac]_{31..0} || \text{LO}[ac]_{31..0} ) \leftarrow \text{tempC}_{63..32} || \text{tempC}_{31..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
Format: DPAX.W.PH ac, rs, rt

Purpose: Cross Dot Product with Accumulate on Vector Integer Halfword Elements

Generate the cross dot-product of two integer halfword vector elements using full-size intermediate products and then accumulate into the specified accumulator register.

Description: \( \text{ac} \leftarrow \text{ac} + ((\text{rs}_{31..16} \times \text{rt}_{15..0}) + (\text{rs}_{15..0} \times \text{rt}_{31..16})) \)

The left halfword integer value from register \( \text{rt} \) is multiplied with the right halfword element from register \( \text{rs} \) to create an integer word result. Similarly, the right halfword integer value from register \( \text{rt} \) is multiplied with the left halfword element from register \( \text{rs} \) to create the second integer word result. These two products are summed to generate the dot-product result, which is then accumulated into the specified 64-bit HI/LO accumulator, creating a 64-bit integer result.

The value of \( \text{ac} \) selects an accumulator numbered from 0 to 3. When \( \text{ac}=0 \), this refers to the original HI/LO register pair of the architecture.

This instruction will not set any bits of the outflag field in the DSPControl register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the result is UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

\[
\begin{align*}
\text{tempB}_{31..0} & \leftarrow (\text{GPR}[\text{rs}]_{31..16} \times \text{GPR}[\text{rt}]_{15..0}) \\
\text{tempA}_{31..0} & \leftarrow (\text{GPR}[\text{rs}]_{15..0} \times \text{GPR}[\text{rt}]_{31..16}) \\
\text{dotp}_{32..0} & \leftarrow ((\text{tempB}_{31} \| \text{tempB}_{31..0}) + (\text{tempA}_{31} \| \text{tempA}_{31..0})) \\
\text{acc}_{63..0} & \leftarrow (\text{HI}[\text{ac}]_{31..0} \| \text{LO}[\text{ac}]_{31..0}) + (\text{dotp}_{32})_{31} \| \text{dotp}_{32..0}) \\
(\text{HI}[\text{ac}]_{..0} \| \text{LO}[\text{ac}]_{..0}) & \leftarrow \text{acc}_{63..32} \text{acc}_{31..0}
\end{align*}
\]

Exceptions:

Reserved Instruction, DSP Disabled
DPS.W.PH Dot Product with Subtract on Vector Integer Half-Word Elements

**Purpose:** Dot Product with Subtract on Vector Integer Half-Word Elements

Generate the dot-product of two integer halfword vector elements using full-size intermediate products and then subtract from the specified accumulator register.

**Description:**

\[ ac \leftarrow ac - ((rs_{31..16} \times rt_{31..16}) + (rs_{15..0} \times rt_{15..0})) \]

Each of the two halfword integer values from register \( rt \) is multiplied with the corresponding halfword element from register \( rs \) to create two integer word results. These two products are summed to generate the dot-product result, which is then subtracted from the specified 64-bit \( HI/LO \) accumulator, creating a 64-bit integer result.

The value of \( ac \) selects an accumulator numbered from 0 to 3. When \( ac=0 \), this refers to the original \( HI/LO \) register pair of the architecture.

This instruction will not set any bits of the ouflag field in the \( DSPControl \) register.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the result is UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

\[
\begin{align*}
\text{tempB}_{31..0} & \leftarrow (\text{GPR}[rs]_{31..16} \times \text{GPR}[rt]_{31..16}) \\
\text{tempA}_{31..0} & \leftarrow (\text{GPR}[rs]_{15..0} \times \text{GPR}[rt]_{15..0}) \\
\text{dotp}_{32..0} & \leftarrow (\text{tempB}_{31}) || \text{tempB}_{31..0} + (\text{tempA}_{31}) || \text{tempA}_{31..0} \\
\text{acc}_{63..0} & \leftarrow (\text{HI}[ac]_{31..0} || \text{LO}[ac]_{31..0}) - (\text{dotp}_{32})_{31} || \text{dotp}_{32..0} \\
\text{acc}_{63..32} || \text{acc}_{31..0} & \leftarrow \text{acc}_{63..32} || \text{acc}_{31..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
DPSQ_S.W.PH Dot Product with Subtraction on Fractional Halfword Elements

**Format:** DPSQ_S.W.PH ac, rs, rt

**Purpose:** Dot Product with Subtraction on Fractional Halfword Elements

Element-wise multiplication of two vectors of fractional halfword elements and subtraction of the accumulated 32-bit intermediate products from the specified 64-bit accumulator register, with saturation.

**Description:**

\[ ac \leftarrow ac - \left( \text{sat32}(rs_{31..16} \times rt_{31..16}) + \text{sat32}(rs_{15..0} \times rt_{15..0}) \right) \]

Each of the two Q15 fractional word values from registers \( rt \) and \( rs \) are multiplied together, and the results left-shifted by one bit position to generate two Q31 fractional format intermediate products. If both multiplicands for either of the multiplications are equal to -1.0 (0x8000 hexadecimal), the resulting intermediate product is saturated to the maximum positive Q31 fractional value (0x7FFFFFFF hexadecimal).

The two intermediate products are then sign-extended and summed to generate a 64-bit, Q32.31 fractional format dot-product result that is subtracted from the specified 64-bit \( HI/LO \) accumulator to produce a final Q32.31 fractional result.

The value of \( ac \) can range from 0 to 3; a value of 0 refers to the original \( HI/LO \) register pair of the architecture.

If saturation occurs as a result of a halfword multiplication, a 1 is written to one of bits 16 through 19 of the \( DSPControl \) register, within the ouflag field. The value of \( ac \) determines which of these bits is set: bit 16 corresponds to \( ac0 \), bit 17 to \( ac1 \), bit 18 to \( ac2 \), and bit 19 to \( ac3 \).

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the result is UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

\[
\begin{align*}
\text{tempB}_{31..0} & \leftarrow \text{multiplyQ15Q15}(\ ac, \ GPR[rs]_{31..16}, \ GPR[rt]_{31..16} ) \\
\text{tempA}_{31..0} & \leftarrow \text{multiplyQ15Q15}(\ ac, \ GPR[rs]_{15..0}, \ GPR[rt]_{15..0} ) \\
\text{dotp}_{63..0} & \leftarrow \left( \text{tempB}_{31}32 || \text{tempB}_{31..0} \right) + \left( \text{tempA}_{31}32 || \text{tempA}_{31..0} \right) \\
\text{tempC}_{63..0} & \leftarrow \left( \text{HI}[ac]_{31..0} || \text{LO}[ac]_{31..0} \right) - \text{dotp}_{63..0} \\
& \left( \text{HI}[ac]_{31..0} || \text{LO}[ac]_{31..0} \right) \leftarrow \text{tempC}_{63..32} || \text{tempC}_{31..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
DPSQ_SA.L.W

Dot Product with Subtraction on Fractional Word Element

Format:  DPSQ_SA.L.W ac, rs, rt

Purpose:  Dot Product with Subtraction on Fractional Word Element

Multiplication of two fractional word elements, subtracting the accumulated product from the specified 64-bit accumulator register, with saturation.

Description:  \( ac \leftarrow \text{sat64}(ac - \text{sat32}(rs_{31..0} \times rt_{31..0})) \)

The two right-most Q31 fractional word values from registers \( rt \) and \( rs \) are multiplied together and the result left-shifted by one bit position to generate a 64-bit Q63 fractional format intermediate product. If both multiplicands are equal to -1.0 (0x80000000 hexadecimal), the intermediate product is saturated to the maximum positive Q63 fractional value (0x7FFFFFFFFFFFFFFF hexadecimal).

The intermediate product is then subtracted from the specified 64-bit HI/LO accumulator, creating a Q63 fractional result. If the accumulation results in overflow or underflow, the accumulator is saturated to either the maximum positive or minimum negative Q63 fractional value (0x8000000000000000 hexadecimal), respectively.

The value of \( ac \) can range from 0 to 3; a value of 0 refers to the original HI/LO register pair of the architecture.

If saturation occurs, a 1 is written to one of bits 16 through 19 of the DSPControl register, within the ouflag field. The value of \( ac \) determines which of these bits is set: bit 16 corresponds to \( ac0 \), bit 17 to \( ac1 \), bit 18 to \( ac2 \), and bit 19 to \( ac3 \).

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the result is UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

\[
\begin{align*}
\text{dotp}_{63..0} & \leftarrow \text{multiplyQ31Q31}( ac, \ GPR[rs]_{31..0}, \ GPR[rt]_{31..0} ) \\
\text{temp}_{64..0} & \leftarrow \text{HI}[ac]_{31} || \text{HI}[ac]_{31..0} || \text{LO}[ac]_{31..0} \\
\text{temp}_{64..0} & \leftarrow \text{temp} - \text{dotp}_{63..0} \\
\text{if} \ ( \text{temp}_{64} \neq \text{temp}_{63} ) \ & \text{then} \\
& \quad \text{if} \ ( \text{temp}_{64} = 1 ) \ & \text{then} \\
& \quad \quad \text{temp}_{63..0} \leftarrow 0x8000000000000000 \\
& \quad \text{else} \\
& \quad \quad \text{temp}_{63..0} \leftarrow 0x7FFFFFFFFFFFFFFF \\
& \text{endif} \\
& \quad \text{DSPControl}_{ouflag; 16+ac} \leftarrow 1 \\
& \text{endif} \\
( \text{HI}[ac]_{..0} || \text{LO}[ac]_{..0} ) & \leftarrow \text{temp}_{63..32} || \text{temp}_{31..0}
\end{align*}
\]

Exceptions:

Reserved Instruction, DSP Disabled
DPSQX_S.W.PH  Cross Dot Product with Subtraction on Fractional Halfword Elements

Format:  DPSQX_S.W.PH ac, rs, rt  

Purpose:  Cross Dot Product with Subtraction on Fractional Halfword Elements

Element-wise cross multiplication of two vectors of fractional halfword elements and subtraction of the accumulated 32-bit intermediate products from the specified 64-bit accumulator register, with saturation.

Description:  ac ← ac - (sat32(rs31..16 * rt15..0) + sat32(rs15..0 * rt31..16))

The left Q15 fractional word value from registers rt is multiplied with the right halfword element from register rs and the result left-shifted by one bit position to generate a Q31 fractional format intermediate product. Similarly, the right Q15 fractional word value from registers rt is multiplied with the left halfword element from register rs and the result left-shifted by one bit position to generate a Q31 fractional format intermediate product. If both multiplicands for either of the multiplications are equal to -1.0 (0x8000 hexadecimal), the resulting intermediate product is saturated to the maximum positive Q31 fractional value (0x7FFFFFFF hexadecimal).

The two intermediate products are then sign-extended and summed to generate a 64-bit, Q32.31 fractional format dot-product result that is subtracted from the specified 64-bit HI/LO accumulator to produce a final Q32.31 fractional result.

The value of ac can range from 0 to 3; a value of 0 refers to the original HI/LO register pair of the architecture.

If saturation occurs as a result of a halfword multiplication, a 1 is written to one of bits 16 through 19 of the DSPControl register, within the ouflag field. The value of ac determines which of these bits is set: bit 16 corresponds to ac0, bit 17 to ac1, bit 18 to ac2, and bit 19 to ac3.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the result is UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

```
function multiplyQ15Q15( acc1..0, a15..0, b15..0 )
  if ( a15..0 = 0x8000 ) and ( b15..0 = 0x8000 ) then
    temp31..0 ← 0x7FFFFFFF
  DSPControl.ouflag:16+acc ← 1
  else
    temp31..0 ← ( a15..0 * b15..0 ) << 1
  endif
  return temp31..0
endfunction multiplyQ15Q15
```

Exceptions:

Reserved Instruction, DSP Disabled
### Purpose:
Cross Dot Product with Subtraction on Fractional Halfword Elements

Element-wise cross multiplication of two vectors of fractional halfword elements and subtraction of the accumulated 32-bit intermediate products from the specified 64-bit accumulator register, with saturation of the accumulator.

### Description:
\[
ac \leftarrow \text{sat32}(ac - \left(\text{sat}32(rs_{31..16} \times rt_{15..0}) + \text{sat}32(rs_{15..0} \times rt_{31..16})\right))
\]

The left Q15 fractional word value from registers \(rt\) is multiplied with the right halfword element from register \(rs\) and the result left-shifted by one bit position to generate a Q31 fractional format intermediate product. Similarly, the right Q15 fractional word value from registers \(rt\) is multiplied with the left halfword element from register \(rs\) and the result left-shifted by one bit position to generate a Q31 fractional format intermediate product. If both multiplicands for either of the multiplications are equal to -1.0 (0x8000 hexadecimal), the resulting intermediate product is saturated to the maximum positive Q31 fractional value (0x7FFFFFFF hexadecimal).

The two intermediate products are then sign-extended and summed to generate a 64-bit, Q32.31 fractional format dot-product result that is subtracted from the specified 64-bit HI/LO accumulator to produce a Q32.31 fractional result. If this result is larger than or equal to +1.0, or smaller than -1.0, it is saturated to the Q31 range.

The value of \(ac\) can range from 0 to 3; a value of 0 refers to the original HI/LO register pair of the architecture.

If saturation occurs as a result of halfword multiplication or accumulation, a 1 is written to one of bits 16 through 19 of the DSPControl register, within the ouflag field. The value of \(ac\) determines which of these bits is set: bit 16 corresponds to \(ac0\), bit 17 to \(ac1\), bit 18 to \(ac2\), and bit 19 to \(ac3\).

### Restrictions:
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the result is \textit{UNPREDICTABLE} and the values of the operand vectors become \textit{UNPREDICTABLE}.

### Operation:

```plaintext
tempB31..0 \leftarrow \text{multiplyQ15Q15}( ac, GPR[rs]_{31..16}, GPR[rt]_{15..0} )
tempA31..0 \leftarrow \text{multiplyQ15Q15}( ac, GPR[rs]_{15..0}, GPR[rt]_{31..16} )
dotp63..0 \leftarrow \left( (tempB31)_32 \| tempB31..0 \right) + \left( (tempA31)_32 \| tempA31..0 \right)
tempC63..0 \leftarrow \left( HI[ac]_{31..0} \| LO[ac]_{31..0} \right) - dotp63..0
if ( tempC63 = 0 ) and ( tempC62..31 \neq 0 ) then
  tempC63..0 = 0_32 \| 0x7FFFFFFF
  DSPControl\_ouflag:16,acc \leftarrow 1
endif
if ( tempC63 = 1 ) and ( tempC62..31 \neq 1_32 ) then
  tempC63..0 = 1_32 \| 0x80000000
  DSPControl\_ouflag:16,acc \leftarrow 1
endif
( HI[ac]..0 \| LO[ac]..0 ) \leftarrow tempC63..32 \| tempC31..0

function multiplyQ15Q15( acc_{..0}, a_{15..0}, b_{15..0} )
  if ( a_{15..0} = 0x8000 ) and ( b_{15..0} = 0x8000 ) then
    temp31..0 \leftarrow 0x7FFFFFFF
    DSPControl\_ouflag:16,acc \leftarrow 1
  endif
```
else
    temp31..0 ← ( a15..0 * b15..0 ) << 1
endif
return temp31..0
endfunction multiplyQ15Q15

**Exceptions:**
Reserved Instruction, DSP Disabled
DPSU.H.QBL Dot Product with Subtraction on Vector Unsigned Byte Elements

Format: DPSU.H.QBL ac, rs, rt

Purpose: Dot Product with Subtraction on Vector Unsigned Byte Elements

Element-wise multiplication of two left-most elements from the four elements of each of two vectors of unsigned bytes, subtracting the sum of the products from the specified 64-bit accumulator register.

Description: ac ← ac - zero_extend((rs31..24 * rt31..24) + (rs23..16 * rt23..16))

The two left-most elements of the four unsigned byte elements of each of registers rt and rs are multiplied together using unsigned arithmetic to generate two 16-bit unsigned intermediate products. The intermediate products are then zero-extended to 64 bits and subtracted from the specified 64-bit HI/LO accumulator. The result of the subtraction is written back to the specified 64-bit HI/LO accumulator.

The value of ac can range from 0 to 3; a value of 0 refers to the original HI/LO register pair of the architecture.

This instruction does not set any bits in the ouflag field in the DSPControl register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the result is UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

```
tempB15..0 ← multiplyU8U8( GPR[rs]31..24, GPR[rt]31..24 )
tempA15..0 ← multiplyU8U8( GPR[rs]23..16, GPR[rt]23..16 )
dotp63..0 ← ( 0^8 || tempB15..0 ) + ( 0^8 || tempA15..0 )
tempC63..0 ← ( HI[ac]31..0 || LO[ac]31..0 ) - dotp63..0
   ( HI[ac]..0 || LO[ac]..0 ) ← tempC63..32 || tempC31..0
```

Exceptions:

Reserved Instruction, DSP Disabled
DPSU.H.QBR  Dot Product with Subtraction on Vector Unsigned Byte Elements

Format:  DPSU.H.QBR ac, rs, rt

Purpose:  Dot Product with Subtraction on Vector Unsigned Byte Elements

Element-wise multiplication of the two right-most elements of the four elements of each of two vectors of unsigned bytes, subtracting the sum of the products from the specified 64-bit accumulator register.

Description:  \( ac \leftarrow ac - \text{zero\_extend}((rs_{15..8} \times rt_{15..8}) + (rs_{7..0} \times rt_{7..0})) \)

The two right-most elements of the four unsigned byte elements of each of registers \( rt \) and \( rs \) are multiplied together using unsigned arithmetic to generate two 16-bit unsigned intermediate products. The intermediate products are then zero-extended to 64 bits and subtracted from the specified 64-bit \( HI/LO \) accumulator.

The value of \( ac \) can range from 0 to 3; a value of 0 refers to the original \( HI/LO \) register pair of the architecture.

This instruction does not set any bits in the \( ouflag \) field in the \( DSPControl \) register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the result is \text{UNPREDICTABLE} and the values of the operand vectors become \text{UNPREDICTABLE}.

Operation:

\[
\begin{align*}
\text{temp}_{B15..0} & \leftarrow \text{multiplyU8U8}( \text{GPR}[rs]_{15..8}, \text{GPR}[rt]_{15..8} ) \\
\text{temp}_{A15..0} & \leftarrow \text{multiplyU8U8}( \text{GPR}[rs]_{7..0}, \text{GPR}[rt]_{7..0} ) \\
\text{dotp}_{63..0} & \leftarrow ( \text{048} || \text{temp}_{B15..0} ) + ( \text{048} || \text{temp}_{A15..0} ) \\
\text{temp}_{C63..0} & \leftarrow ( \text{HI}[ac]_{31..0} || \text{LO}[ac]_{31..0} ) - \text{dotp}_{63..0} \\
( \text{HI}[ac]_{..0} || \text{LO}[ac]_{..0} ) & \leftarrow \text{temp}_{C63..32} || \text{temp}_{C31..0}
\end{align*}
\]

Exceptions:

Reserved Instruction, DSP Disabled
**Format:** DPSX.W.PH ac, rs, rt  

**Purpose:** Cross Dot Product with Subtract on Vector Integer Halfword Elements  

Generate the cross dot-product of two integer halfword vector elements using full-size intermediate products and then subtract from the specified accumulator register.

**Description:**  
\[ ac \leftarrow ac - ((rs_{31..16} \times rt_{15..0}) + (rs_{15..0} \times rt_{31..16})) \]  
The left halfword integer value from register \( rt \) is multiplied with the right halfword element from register \( rs \) to create an integer word result. Similarly, the right halfword integer value from register \( rt \) is multiplied with the left halfword element from register \( rs \) to create the second integer word result. These two products are summed to generate the dot-product result, which is then subtracted from the specified 64-bit HI/LO accumulator, creating a 64-bit integer result.

The value of \( ac \) selects an accumulator numbered from 0 to 3. When \( ac=0 \), this refers to the original HI/LO register pair of the architecture.

This instruction will not set any bits of the outflag field in the DSPControl register.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the result is UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

\[
\begin{align*}
tempB_{31..0} & \leftarrow (GPR[rs]_{31..16} \times GPR[rt]_{15..0}) \\
tempA_{31..0} & \leftarrow (GPR[rs]_{15..0} \times GPR[rt]_{31..16}) \\
dotp_{32..0} & \leftarrow (tempB_{31}) || tempB_{31..0} + (tempA_{31}) || tempA_{31..0} \\
acc_{63..0} & \leftarrow (HI[ac]_{31..0} || LO[ac]_{31..0}) - (dotp_{32})_{31} || dotp_{32..0} \\
& \quad (HI[ac]..0 || LO[ac]..0) \leftarrow acc_{63..32} acc_{31..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
**EXTP**

**Extract Fixed Bitfield From Arbitrary Position in Accumulator to GPR**

**Format:** EXTP rt, ac, size

**Purpose:** Extract Fixed Bitfield From Arbitrary Position in Accumulator to GPR

Extract size+1 contiguous bits from a 64-bit accumulator from a position specified in the DSPControl register, writing the bits to a GPR with zero-extension.

**Description:**

rt ← zero_extend(acpos..pos-size)

A set of size+1 contiguous bits are extracted from an arbitrary position in accumulator ac, zero-extended to bits, and then written to register rt.

The bit position, start_pos, of the first bit of the contiguous set to extract is specified by the pos field in bits 0 through 5 of the DSPControl register. The last bit in the set is start_pos - size, where size is specified in the instruction.

The value of ac can range from 0 to 3. When ac=0, this refers to the original HI/LO register pair of the architecture. After the execution of this instruction, accumulator ac remains unmodified.

If start_pos - (size + 1) ≥ -1, the extraction is valid, otherwise the extraction is invalid and is said to have failed. The value of the destination register is UNPREDICTABLE when the extraction is invalid. Upon an invalid extraction this instruction writes a 1 to bit 14, the Extract Failed Indicator (EFI) bit of the DSPControl register, and 0 otherwise.

The values of bits 0 to in the pos field of the DSPControl register are unchanged by this instruction.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

```
start_pos5..0 ← DSPControlpos5..0
if ( start_pos - (size+1) >= -1 ) then
    temp_size..0 ← ( HI[ac]31..0 || LO[ac]31..0 )start_pos..start_pos-size
    temp31..0 ← 0(32-(size+1)) || temp_size..0
    GPR[rt]..0 ← temp31..0
    DSPControlEFI:14 ← 0
else
    DSPControlEFI:14 ← 1
    GPR[rt] ← UNPREDICTABLE
endif
```

**Exceptions:**

Reserved Instruction, DSP Disabled
EXTPDP  Extract Fixed Bitfield From Arbitrary Position in Accumulator to GPR and Decrement Pos

Format:  EXTPDP rt, ac, size

Purpose:  Extract Fixed Bitfield From Arbitrary Position in Accumulator to GPR and Decrement Pos

Extract size+1 contiguous bits from a 64-bit accumulator from a position specified in the DSPControl register, writing the bits to a GPR with zero-extension and modifying the extraction position.

Description:  rt ← zero_extend(acpos..pos-size) ; DSPControlpos:..0 -= (size+1)

A set of size+1 contiguous bits are extracted from an arbitrary position in accumulator ac, zero-extended to size bits, then written to register rt.

The bit position, start_pos, of the first bit of the contiguous set to extract is specified by the pos field in bits 0 through 5 of the DSPControl register. The position of the last bit in the extracted set is start_pos - size, where the size argument is specified in the instruction.

The value of ac can range from 0 to 3. When ac=0, this refers to the original HI/LO register pair of the architecture. After the execution of this instruction, accumulator ac remains unmodified.

If start_pos-(size+1) ≥ -1, the extraction is valid and the value of the pos field in the DSPControl register is decremented by size+1. Otherwise, the extraction is invalid and is said to have failed. The value of the destination register is UNPREDICTABLE when the extraction is invalid, and the value of the pos field in the DSPControl register (bits 0 through ) is not modified.

Upon an invalid extraction this instruction writes a 1 to bit 14, the Extract Failed Indicator (EFI) bit of the DSPControl register, and 0 otherwise.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

```
start_pos5..0 ← DSPControlpos5..0
if ( start_pos - (size+1) >= -1 ) then
    tempsize..0 ← ( HI[ac]31..0 || LO[ac]31..0 )start_pos..start_pos-size
    GPR[rt] ← 0(GPRLEN-(size+1)) || tempsize..0
    DSPControlpos:..0 ← DSPControlpos:..0 - (size + 1)
    DSPControlEFI:14 ← 0
else
    DSPControlEFI:14 ← 1
    GPR[rt] ← UNPREDICTABLE
endif
```

Exceptions:

Reserved Instruction, DSP Disabled

MIPS® DSP Module for MIPS32™ Architecture, Revision 3.01
EXTPDPV Extract Variable Bitfield From Arbitrary Position in Accumulator to GPR and Decrement Pos

Format: EXTPDPV rt, ac, rs

Purpose: Extract Variable Bitfield From Arbitrary Position in Accumulator to GPR and Decrement Pos

Extract a fixed number of contiguous bits from a 64-bit accumulator from a position specified in the DSPControl register, writing the bits to a GPR with zero-extension and modifying the extraction position.

Description: rt ← zero_extend(acpos..pos-GPR[rs][4:0]) ; DSPControlpos...0 -= (GPR[rs]4..0+1)

A fixed number of contiguous bits are extracted from an arbitrary position in accumulator ac, zero-extended to bits, then written to destination register rt. The number of bits extracted is size+1, where size is specified by the five least-significant bits in register rs, interpreted as a five-bit unsigned integer. The remaining bits in register rs are ignored.

The bit position, start_pos, of the first bit of the contiguous set to extract is specified by the pos field in bits 0 through 5 of the DSPControl register. The position of the last bit in the extracted set is start_pos-size.

The value of ac can range from 0 to 3. When ac=0, this refers to the original HI/LO register pair of the architecture. After the execution of this instruction, accumulator ac remains unmodified.

If start_pos-(size+1) ≥ -1, the extraction is valid and the value of the pos field in the DSPControl register is decremented by size+1. Otherwise, the extraction is invalid and is said to have failed. The value of the destination register is UNPREDICTABLE when the extraction is invalid, and the value of the pos field in the DSPControl register (bits 0 through ) is not modified.

Upon an invalid extraction this instruction writes a 1 to bit 14, the Extract Failed Indicator (EFI) bit of the DSPControl register, and 0 otherwise.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

\[
\begin{align*}
\text{start} \_ \text{pos}_{5..0} & \leftarrow \text{DSPControl}_{pos_{5..0}} \\
\text{size}_{4..0} & \leftarrow \text{GPR[rs]}_{4..0} \\
\text{if} & \ ( \text{start} \_ \text{pos} - (\text{size}+1) \geq -1 ) \ \text{then} \\
& \quad \text{temp} \_ \text{size}_{0} \leftarrow ( \text{HI[ac]}_{31..0} || \text{LO[ac]}_{31..0} )_{start \_ \text{pos}.. start \_ \text{pos} - \text{size}} \\
& \quad \text{GPR[rt]} \leftarrow 0_{(\text{GPRLEN}-(\text{size}+1))} || \text{temp} \_ \text{size}_{0} \\
& \quad \text{DSPControl}_{pos_{5..0}} \leftarrow \text{DSPControl}_{pos_{5..0}} - (\text{size} + 1) \\
& \quad \text{DSPControl}_{\text{EFI}:14} \leftarrow 0 \\
\text{else} & \\
& \quad \text{DSPControl}_{\text{EFI}:14} \leftarrow 1 \\
& \quad \text{GPR[rt]} \leftarrow \text{UNPREDICTABLE}
\end{align*}
\]

Exceptions:

Reserved Instruction, DSP Disabled
EXTPV Extract Variable Bitfield From Arbitrary Position in Accumulator to GPR

**Format:**  
EXTPV rt, ac, rs

**Purpose:** Extract Variable Bitfield From Arbitrary Position in Accumulator to GPR

Extract a variable number of contiguous bits from a 64-bit accumulator from a position specified in the DSPControl register, writing the bits to a GPR with zero-extension.

**Description:**  
rt ← zero_extend(acpos..pos-rs[4:0])

A variable number of contiguous bits are extracted from an arbitrary position in accumulator ac, zero-extended to bits, then written to register rt. The number of bits extracted is size+1, where size is specified by the five least-significant bits in register rs, interpreted as a five-bit unsigned integer. The remaining bits in register rs are ignored.

The position of the first bit of the contiguous set to extract, start_pos, is specified by the pos field in bits 0 through 5 of the DSPControl register. The position of the last bit in the contiguous set is start_pos - size.

The value of ac can range from 0 to 3. When ac=0, this refers to the original HI/LO register pair of the architecture. After the execution of this instruction, accumulator ac remains unmodified.

An extraction is valid if start_pos - (size + 1) ≥ -1; otherwise, the extraction is invalid and is said to have failed. The value of the destination register is UNPREDICTABLE when the extraction is invalid. Upon an invalid extraction this instruction writes a 1 to bit 14, the Extract Failed Indicator (EFI) bit of the DSPControl register, and 0 otherwise.

The values of bits 0 to 5 in the pos field of the DSPControl register are unchanged by this instruction.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

```
start_pos5..0 ← DSPControlpos5..0
size4..0 ← GPR[rs]4..0
if ( start_pos - (size+1) >= -1 ) then
  tempsize..0 ← ( HI[ac]31..0 || LO[ac]31..0 )start_pos..start_pos-size
  GPR[rt] ← 0(GPRLEN-(size+1)) || tempsize..0
  DSPControlEFI:14 ← 0
else
  DSPControlEFI:14 ← 1
  GPR[rt] ← UNPREDICTABLE
endif
```

**Exceptions:**

Reserved Instruction, DSP Disabled
**EXTR[ RS].W**

*Extract Word Value With Right Shift From Accumulator to GPR*

<table>
<thead>
<tr>
<th>SPECIAL3</th>
<th>shift</th>
<th>rt</th>
<th>ac</th>
<th>EXTR.W</th>
<th>EXTR.W</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111</td>
<td></td>
<td>000</td>
<td>ac</td>
<td>00000</td>
<td>111000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000</td>
<td>ac</td>
<td>00100</td>
<td>111000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000</td>
<td>ac</td>
<td>00110</td>
<td>111000</td>
</tr>
</tbody>
</table>

**Format:**

- `EXTR[ RS].W`
- `EXTR.W  rt, ac, shift`
- `EXTR_R.W rt, ac, shift`
- `EXTR_RS.W rt, ac, shift`

**Purpose:**

Extract a word value from a 64-bit accumulator to a GPR with right shift, and with optional rounding or rounding and saturation.

**Description:**

\[ rt \leftarrow \text{sat32}(\text{round}(ac \gg shift)) \]

The value in accumulator `ac` is shifted right by `shift` bits with sign extension (arithmetic shift right). The 32 least-significant bits of the shifted value are then written to the destination register `rt`.

The rounding variant of the instruction adds a 1 at the most-significant discarded bit position. The 32 least-significant bits of the rounded result are then written to the destination register.

The rounding and saturating variant of the instruction adds a 1 at the most-significant discarded bit position. If the rounding operation results in an overflow, the shifted value is clamped to the maximum positive Q31 fractional value (0x7FFFFFFF hexadecimal). The rounded and saturated result is then written to the destination register.

The value of `ac` can range from 0 to 3. When `ac=0`, this refers to the original HI/LO register pair of the architecture. After the execution of this instruction, `ac` remains unmodified.

For all variants of the instruction, including EXTR.W, bit 23 of the `DSPControl` register is set to 1 if either of the rounded or non-rounded calculation results in overflow or saturation.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

```plaintext
EXTR_W temp64..0 ← _shiftShortAccRightArithmetic( ac, shift )
if (( temp64..32 ≠ 0 ) and ( temp64..32 ≠ 0x1FFFFFFF ) ) then
    DSPControlouflag:23 ← 1
endif
GPR[rt]..0 ← temp32..1
temp64..0 ← temp + 1
if (( temp64..32 ≠ 0 ) and ( temp64..32 ≠ 0x1FFFFFFF ) ) then
    DSPControlouflag:23 ← 1
endif
EXTR_R.W
```
temp64..0 ← _shiftShortAccRightArithmetic( ac, shift )
if ((temp64..32 ≠ 0) and (temp64..32 ≠ 0x1FFFFFFFF)) then
    DSPControloutflag:23 ← 1
endif

temp64..0 ← temp + 1
if ((temp64..32 ≠ 0) and (temp64..32 ≠ 0x1FFFFFFFF)) then
    DSPControloutflag:23 ← 1
endif

GPR[rt]..0 ← temp32..1

EXTR_RS.W

temp64..0 ← _shiftShortAccRightArithmetic( ac, shift )
if ((temp64..32 ≠ 0) and (temp64..32 ≠ 0x1FFFFFFFF)) then
    DSPControloutflag:23 ← 1
endif

temp64..0 ← temp + 1
if ((temp64..32 ≠ 0) and (temp64..32 ≠ 0x1FFFFFFFF)) then
    if (temp64 = 0) then
        temp32..1 ← 0x7FFFFFFF
    else
        temp32..1 ← 0x80000000
    endif
    DSPControloutflag:23 ← 1
endif

GPR[rt]..0 ← temp32..1

function _shiftShortAccRightArithmetic( ac1..0, shift4..0 )
if (shift4..0 = 0) then
    temp64..0 ← (HI[ac]31..0 || LO[ac]31..0 || 0)
else
    temp64..0 ← (HI[ac]31shift || HI[ac]31..0 || LO[ac]31..shift-1)
endif
return temp64..0
endfunction _shiftShortAccRightArithmetic

Exceptions:
Reserved Instruction, DSP Disabled
EXTR_S.H

Extract Halfword Value From Accumulator to GPR With Right Shift and Saturate

Format: EXTR_S.H rt, ac, shift

Purpose: Extract Halfword Value From Accumulator to GPR With Right Shift and Saturate

Extract a halfword value from a 64-bit accumulator to a GPR with right shift and saturation.

Description:

The value in the 64-bit accumulator $ac$ is shifted right by $shift$ bits with sign extension (arithmetic shift right). The 64-bit value is then saturated to 16-bits, sign extended to 16 bits, and written to the destination register $rt$. The shift argument is provided in the instruction.

The value of $ac$ can range from 0 to 3. When $ac=0$, this refers to the original HI/LO register pair of the architecture. After the execution of this instruction, $ac$ remains unmodified.

This instruction sets bit 23 of the $DSPControl$ register in the $ouflag$ field if the operation results in saturation.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

```plaintext
temp63..0 ← shiftShortAccRightArithmetic( ac, shift )
if ( temp63..0 > 0x0000000000007FFF ) then
    temp31..0 ← 0x00007FFF
    DSPControl$ouflag:23 ← 1
else if ( temp63..0 < 0xFFFFFFFFFFFF8000 ) then
    temp31..0 ← 0xFFFF8000
    DSPControl$ouflag:23 ← 1
endif
GPR[rt]..0 ← temp31..0

function shiftShortAccRightArithmetic( ac1..0, shift4..0 )
    sign ← HI[ac]31
    if ( shift = 0 ) then
        temp63..0 ← HI[ac]31..0 || LO[ac]31..0
    else
        temp63..0 ← sign shift || (( HI[ac]31..0 || LO[ac]31..0 ) >> shift )
    endif
    if ( sign ≠ temp31 ) then
        DSPControl$ouflag:23 ← 1
    endif
    return temp63..0
endfunction shiftShortAccRightArithmetic
```

Exceptions:

Reserved Instruction, DSP Disabled
**EXTRV[ RS].W**

**Extract Word Value With Variable Right Shift From Accumulator to GPR**

<table>
<thead>
<tr>
<th>SPECIAL3</th>
<th>rs</th>
<th>rt</th>
<th>0</th>
<th>ac</th>
<th>EXTRV.W</th>
<th>EXTR.W</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111</td>
<td>rs</td>
<td>rt</td>
<td>0</td>
<td>ac</td>
<td>000001</td>
<td>111000</td>
</tr>
<tr>
<td>011111</td>
<td>rs</td>
<td>rt</td>
<td>0</td>
<td>ac</td>
<td>001001</td>
<td>111000</td>
</tr>
<tr>
<td>011111</td>
<td>rs</td>
<td>rt</td>
<td>0</td>
<td>ac</td>
<td>001111</td>
<td>111000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31 26 25 21 20 16 15 13 12 11 10 6 5 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL3</td>
</tr>
<tr>
<td>011111</td>
</tr>
<tr>
<td>011111</td>
</tr>
<tr>
<td>011111</td>
</tr>
</tbody>
</table>

**Format:**
- EXTRV[ RS].W
- EXTRV.W rt, ac, rs
- EXTRV_R.W rt, ac, rs
- EXTRV_RS.W rt, ac, rs

**Purpose:** Extract Word Value With Variable Right Shift From Accumulator to GPR

Extract a word value from a 64-bit accumulator to a GPR with variable right shift, and with optional rounding or rounding and saturation.

**Description:**
- rt $\leftarrow$ sat32(round(ac $>>$ rs5..0))

The value in accumulator ac is shifted right by shift bits with sign extension (arithmetic shift right). The lower 32 bits of the shifted value are then written to the destination register rt. The number of bits to shift is given by the five least-significant bits of register rs; the remaining bits of rs are ignored.

The rounding variant of the instruction adds a 1 at the most-significant discarded bit position. The 32 least-significant bits of the rounded result are then written to the destination register.

The rounding and saturating variant of the instruction adds a 1 at the most-significant discarded bit position. If the rounding operation results in an overflow, the shifted value is clamped to the maximum positive Q31 fractional value (0x7FFFFFFF hexadecimal). The rounded and saturated result is then written to the destination register.

The value of ac can range from 0 to 3. When ac=0, this refers to the original HI/LO register pair of the architecture. After the execution of this instruction, ac remains unmodified.

For all variants of the instruction, including EXTRV.W, bit 23 of the DSPControl register is set to 1 if either of the rounded or non-rounded calculation results in overflow or saturation.

**Restrictions:**
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

```c
EXTRV.W
    temp64..0 $\leftarrow$ _shiftShortAccRightArithmetic( ac, GPR[rs]4..0 )
    if (( temp64..32 $\neq$ 0 ) and ( temp64..32 $\neq$ 0x1FFFFFFF )) then
        DSPControl.outflag:23 $\leftarrow$ 1
    endif
    GPR[rt]..0 $\leftarrow$ temp32..1
    temp64..0 $\leftarrow$ temp + 1
    if (( temp64..32 $\neq$ 0 ) and ( temp64..32 $\neq$ 0x1FFFFFFF )) then
        DSPControl.outflag:23 $\leftarrow$ 1
    endif
```
EXTRV[RS].W  
Extract Word Value With Variable Right Shift From Accumulator to GPR

EXTRV[RS].W

\[
\begin{align*}
temp_{64..0} & \leftarrow \text{shiftShortAccRightArithmetic}(\ ac, \ GPR[rs]_{4..0}) \\
& \quad \text{if } ((temp_{64..32} \neq 0) \text{ and } (temp_{64..32} \neq 0x1FFFFFFFF)) \text{ then} \\
& \quad \quad \text{DSPControloutflag}_{23} \leftarrow 1 \\
& \quad \text{endif} \\
temp_{64..0} & \leftarrow temp + 1 \\
& \quad \text{if } ((temp_{64..32} \neq 0) \text{ and } (temp_{64..32} \neq 0x1FFFFFFFF)) \text{ then} \\
& \quad \quad \text{DSPControloutflag}_{23} \leftarrow 1 \\
& \quad \text{endif} \\
GPR[rt]_{..0} & \leftarrow temp_{32..1}
\end{align*}
\]

EXTRV_RS.W

\[
\begin{align*}
temp_{64..0} & \leftarrow \text{shiftShortAccRightArithmetic}(\ ac, \ GPR[rs]_{4..0}) \\
& \quad \text{if } ((temp_{64..32} \neq 0) \text{ and } (temp_{64..32} \neq 0x1FFFFFFFF)) \text{ then} \\
& \quad \quad \text{DSPControloutflag}_{23} \leftarrow 1 \\
& \quad \text{endif} \\
temp_{64..0} & \leftarrow temp + 1 \\
& \quad \text{if } ((temp_{64..32} \neq 0) \text{ and } (temp_{64..32} \neq 0x1FFFFFFFF)) \text{ then} \\
& \quad \quad \text{if } (temp_{64} = 0) \text{ then} \\
& \quad \quad \quad temp_{32..1} \leftarrow 0x7FFFFFFFF \\
& \quad \quad \text{else} \\
& \quad \quad \quad temp_{32..1} \leftarrow 0x80000000 \\
& \quad \quad \text{endif} \\
& \quad \quad \text{DSPControloutflag}_{23} \leftarrow 1 \\
& \quad \text{endif} \\
GPR[rt]_{..0} & \leftarrow temp_{32..1}
\end{align*}
\]

Exceptions:

Reserved Instruction, DSP Disabled
EXTRV_S.H  Extract Halfword Value Variable From Accumulator to GPR With Right Shift and Saturate

Format:  EXTRV_S.H rt, ac, rs

Purpose: Extract Halfword Value Variable From Accumulator to GPR With Right Shift and Saturate

Extract a halfword value from a 64-bit accumulator to a GPR with right shift and saturation.

Description: rt \leftarrow \text{saturated}((ac >> rs4\ldots0))

The value in the 64-bit accumulator \(ac\) is shifted right by \(shift\) bits with sign extension (arithmetic shift right). The 64-bit value is then saturated to 16-bits and sign-extended to 31 bits before being written to the destination register \(rt\). The five least-significant bits of register \(rs\) provide the shift argument, interpreted as a five-bit unsigned integer; the remaining bits in \(rs\) are ignored.

The value of \(ac\) can range from 0 to 3. When \(ac=0\), this refers to the original HI/LO register pair of the architecture. After the execution of this instruction, \(ac\) remains unmodified.

This instruction sets bit 23 of the DSPControl register in the ouflag field if the operation results in saturation.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

\begin{verbatim}
shift4\ldots0 \leftarrow \text{GPR}[rs]4\ldots0

\text{temp31\ldots0} \leftarrow \text{shiftShortAccRightArithmetic}(\ ac, \ shift) 

\text{if ( temp63\ldots0 > 0x0000000000007FFF ) then}
\quad \text{temp31\ldots0} \leftarrow 0x00007FFF
\quad \text{DSPControl}_{23} \leftarrow 1
\quad \text{endif}
\text{if ( temp63\ldots0 < 0xFFFFFFFFFFFF8000 ) then}
\quad \text{temp31\ldots0} \leftarrow 0xFFFFFFFF8000
\quad \text{DSPControl}_{23} \leftarrow 1
\quad \text{endif}

GPR[rt] \ldots0 \leftarrow \text{temp31\ldots0}
\end{verbatim}

Exceptions:

Reserved Instruction, DSP Disabled
Insert Bit Field Variable

Format: INSV rt, rs

Purpose: Insert Bit Field Variable

To merge a right-justified bit field from register rs into a specified field in register rt.

Description: \( rt \leftarrow \text{InsertFieldVar}(rt, rs, \text{Scount}, \text{Pos}) \)

The DSPControl register provides the size value from the Scount field, and the pos value from the pos field. The rightmost size bits from register rs are merged into the value from register rt starting at bit position pos. The result is put back in register rt. These pos and size values are converted by the instruction into the fields msb (the most significant bit of the field), and lsb (least significant bit of the field), as follows:

\[
\begin{align*}
\text{pos} & \leftarrow \text{DSPControl}_{5..0} \\
\text{size} & \leftarrow \text{DSPControl}_{12..7} \\
\text{msb} & \leftarrow \text{pos} + \text{size} - 1 \\
\text{lsb} & \leftarrow \text{pos}
\end{align*}
\]

The values of pos and size must satisfy all of the following relations, or the instruction results in UNPREDICTABLE results:

\[
\begin{align*}
0 \leq \text{pos} & < 32 \\
0 < \text{size} & \leq 32 \\
0 < \text{pos} + \text{size} & \leq 32
\end{align*}
\]

Figure 6.1 shows the symbolic operation of the instruction.

Restrictions:

The operation is UNPREDICTABLE if lsb > msb.
INSV  Insert Bit Field Variable

Operation:

\[
\text{if (lsb} > \text{msb) then} \\
\quad \text{UNPREDICTABLE} \\
\text{endif} \\
\text{GPR[rt]}_{\ldots0} \leftarrow \text{GPR[rt]}_{31\ldots\text{msb}+1} \parallel \text{GPR[rs]}_{\text{msb} \ldots \text{lsb} \ldots 0} \parallel \text{GPR[rt]}_{\text{lsb} \ldots 1\ldots0}
\]

Exceptions:

Reserved Instruction, DSP Disabled
LBUX Load Unsigned Byte Indexed

Format: \texttt{LBUX \textit{rd}, index(base)}

Purpose: Load Unsigned Byte Indexed

To load a byte from memory as an unsigned value, using indexed addressing.

Description: \( \text{rd} \leftarrow \text{memory[base+index]} \)

The contents of GPR index is added to the contents of GPR base to form an effective address. The contents of the 8-bit byte at the memory location specified by the aligned effective address are fetched, zero-extended to the GPR register length and placed in GPR rd.

Restrictions: None.

Operation:

\begin{align*}
\text{vAddr}_{31..0} & \leftarrow \text{GPR[index]}_{31..0} + \text{GPR[base]}_{31..0} \\
( \text{pAddr, CCA} ) & \leftarrow \text{AddressTranslation( vAddr, DATA, LOAD )} \\
\text{pAddr} & \leftarrow \text{pAddr}_{PSIZE-1..2} \ || \ ( \text{pAddr}_{1..0} \ xor \ \text{ReverseEndian}^2 ) \\
\text{memword}_{GPRLEN..0} & \leftarrow \text{LoadMemory ( CCA, BYTE, pAddr, vAddr, DATA )} \\
\text{GPR[rd]}_{..0} & \leftarrow \text{zero\_extend( memword}_{7..0} )
\end{align*}

Exceptions:

Reserved Instruction, DSP Disabled, TLB Refill, TLB Invalid, Bus Error, Address Error, Watch
**Format:**  LHX rd, index(base)

**Purpose:**  Load Halfword Indexed

To load a halfword value from memory as a signed value, using indexed addressing.

**Description:**  \( rd \leftarrow \text{memory}[\text{base} + \text{index}] \)

The contents of GPR \( \text{index} \) is added to the contents of GPR \( \text{base} \) to form an effective address. The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, sign-extended to the length of the destination GPR, and placed in GPR \( rd \).

**Restrictions:**

The effective address must be naturally-aligned. If the least-significant bit of the effective address is non-zero, an Address Error exception occurs.

**Operation:**

\[
\text{vAddr}_{31..0} \leftarrow \text{GPR}[\text{index}]_{31..0} + \text{GPR}[\text{base}]_{31..0}
\]

if ( \text{vAddr}_0 \neq 0 ) then
  SignalException( AddressError )
endif

( \text{pAddr}, \text{CCA} ) \leftarrow \text{AddressTranslation}( \text{vAddr}, \text{DATA}, \text{LOAD} )

\text{halfword}_{\text{GPRLEN}..0} \leftarrow \text{LoadMemory}( \text{CCA}, \text{HALFWORD}, \text{pAddr}, \text{vAddr}, \text{DATA} )

\text{GPR}[rd]_{..0} \leftarrow \text{sign\_extend}( \text{halfword}_{15..0} )

**Exceptions:**

Reserved Instruction, DSP Disabled, TLB Refill, TLB Invalid, Bus Error, Address Error, Watch
Format: LWX rd, index(base)

Purpose: Load Word Indexed

To load a word value from memory as a signed value, using indexed addressing.

Description: \( rd \leftarrow \text{memory}[\text{base}+\text{index}] \)

The contents of GPR \( \text{index} \) is added to the contents of GPR \( \text{base} \) to form an effective address. The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed in GPR \( rd \).

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:

\[
\begin{align*}
\text{vAddr}_{31..0} & \leftarrow \text{GPR[\text{index}]}_{31..0} + \text{GPR[\text{base}]}_{31..0} \\
\text{if} \ ( \text{vAddr}_{1..0} \neq 0^2 ) \text{ then} & \\
& \quad \text{SignalException( AddressError )} \\
\text{endif} \\
( \text{pAddr, CCA} ) & \leftarrow \text{AddressTranslation( vAddr, DATA, LOAD )} \\
\text{memword}_{\text{GPRLEN}..0} & \leftarrow \text{LoadMemory( CCA, WORD, pAddr, vAddr, DATA )} \\
\text{GPR[rd]}_{..0} & \leftarrow \text{memword}_{31..0}
\end{align*}
\]

Exceptions:

Reserved Instruction, DSP Disabled, TLB Refill, TLB Invalid, Bus Error, Address Error, Watch
**MADD**

### Multiply Word and Add to Accumulator

**Format:** \( \text{MADD ac, rs, rt} \)

**MIPS32 pre-Release 6, MIPSDSP**

**Purpose:** Multiply Word and Add to Accumulator

To multiply two 32-bit integer words and add the 64-bit result to the specified accumulator.

**Description:** \( (\text{HI}[ac] || \text{LO}[ac]) \leftarrow (\text{HI}[ac] || \text{LO}[ac]) + (rs_{31\ldots0} \times rt_{31\ldots0}) \)

The 32-bit signed integer word in register \( rs \) is multiplied by the corresponding 32-bit signed integer word in register \( rt \) to produce a 64-bit result. The 64-bit product is added to the specified 64-bit accumulator.

These special registers \( \text{HI} \) and \( \text{LO} \) are specified by the value of \( ac \). When \( ac=0 \), this refers to the original \( \text{HI}/\text{LO} \) register pair of the architecture.

In Release 6 of the MIPS Architecture, accumulators are eliminated from MIPS32.

No arithmetic exception occurs under any circumstances.

**Restrictions:**

This instruction does not provide the capability of writing directly to a target GPR.

**Operation:**

\[
\begin{align*}
\text{temp}_{63\ldots0} & \leftarrow ((\text{GPR}[rs]_{31})^{32} || \text{GPR}[rs]_{31\ldots0}) \times ((\text{GPR}[rt]_{31})^{32} || \text{GPR}[rt]_{31\ldots0}) \\
\text{acc}_{63\ldots0} & \leftarrow (\text{HI}[ac]_{31\ldots0} || \text{LO}[ac]_{31\ldots0}) + \text{temp}_{63\ldots0} \\
(\text{HI}[ac]_{31\ldots0} || \text{LO}[ac]_{31\ldots0}) & \leftarrow \text{acc}_{63\ldots32} || \text{acc}_{31\ldots0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled

**Programming Notes:**

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read \( \text{LO} \) or \( \text{HI} \) before the results are written interlocks 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.

Where the size of the operands are known, software should place the shorter operand in register \( rt \). This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
MADDU Multiply Unsigned Word and Add to Accumulator

**Format:** MADDU ac, rs, rt

**Purpose:** Multiply Unsigned Word and Add to Accumulator

To multiply two 32-bit unsigned integer words and add the 64-bit result to the specified accumulator.

**Description:**

\[(HI[ac]||LO[ac]) \leftarrow (HI[ac]||LO[ac]) + (rs_{31..0} \times rt_{31..0})\]

The 32-bit unsigned integer word in register \(rs\) is multiplied by the corresponding 32-bit unsigned integer word in register \(rt\) to produce a 64-bit result. The 64-bit product is added to the specified 64-bit accumulator.

These special registers \(HI\) and \(LO\) are specified by the value of \(ac\). When \(ac=0\), this refers to the original \(HI/LO\) register pair of the architecture.

In Release 6 of the MIPS Architecture, accumulators are eliminated from MIPS32.

No arithmetic exception occurs under any circumstances.

**Restrictions:**

This instruction does not provide the capability of writing directly to a target GPR.

**Operation:**

\[
\begin{align*}
temp_{64..0} & \leftarrow (0^{32} || GPR[rs]_{31..0}) \times (0^{32} || GPR[rt]_{31..0}) \\
acc_{63..0} & \leftarrow (HI[ac]_{31..0} || LO[ac]_{31..0}) + temp_{63..0} \\
& (HI[ac]_{..0} || LO[ac]_{..0}) \leftarrow acc_{63..32} || acc_{31..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled

**Programming Notes:**

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read \(LO\) or \(HI\) before the results are written interlocks 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.

Where the size of the operands are known, software should place the shorter operand in register \(rt\). This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
### Multiply with Accumulate Single Vector Fractional Halfword Element

**Format:**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL3</td>
<td>rs, rt, ac</td>
</tr>
<tr>
<td>MAQ_S.W.PHL</td>
<td>ac, rs, rt</td>
</tr>
<tr>
<td>MAQ_SA.W.PHL</td>
<td>ac, rs, rt</td>
</tr>
</tbody>
</table>

**Purpose:** Multiply with Accumulate Single Vector Fractional Halfword Element

To multiply one pair of elements from two vectors of fractional halfword values using full-sized intermediate products and accumulate the result into the specified 64-bit accumulator, with optional saturating accumulation.

**Description:**

\[
ac \leftarrow \text{sat32}(ac + \text{sat32}(rs_{31..16} \times rt_{31..16}))
\]

The left-most Q15 fractional halfword values from the paired halfword vectors in each of registers \(rt\) and \(rs\) are multiplied together, and the product left-shifted by one bit position to generate a Q31 fractional format intermediate result. If both multiplicands are equal to \(-1.0\) in Q15 fractional format (0x8000 hexadecimal), the intermediate result is saturated to the maximum positive Q31 fractional value (0x7FFFFFFF hexadecimal). The intermediate result is then sign-extended and accumulated into accumulator \(ac\) to generate a 64-bit Q32.31 fractional format result.

In the saturating accumulation variant of this instruction, if the accumulation of the intermediate product with the accumulator results in a value that cannot be represented as a Q31 fractional format value, the accumulator is saturated to either the maximum positive Q31 fractional format value (0x7FFFFFFF hexadecimal) or the minimum negative Q31 fractional format value (0x80000000), sign-extended to 64 bits.

The value of \(ac\) can range from 0 to 3; a value of 0 refers to the original \(HI/LO\) register pair of the architecture.

If overflow or saturation occurs, a 1 is written to one of bits 16 through 19 of the \(DSPControl\) register, within the \(ouflag\) field. The value of \(ac\) determines which of these bits is set: bit 16 corresponds to \(ac0\), bit 17 to \(ac1\), bit 18 to \(ac2\), and bit 19 to \(ac3\).

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the result is UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

**MAQ_S.W.PHL**

\[
\begin{align*}
tempA_{31..0} & \leftarrow \text{multiplyQ15Q15}(ac, GPR[rs]_{31..16}, GPR[rt]_{31..16}) \\
tempB_{63..0} & \leftarrow (HI[ac]_{31..0} || LO[ac]_{31..0}) + (tempA_{31} || tempA_{31..0}) \\
(HI[ac]_{..0} || LO[ac]_{..0}) & \leftarrow tempB_{63..32} || tempB_{31..0}
\end{align*}
\]

**MAQ_SA.W.PHL**

\[
\begin{align*}
tempA_{31..0} & \leftarrow \text{multiplyQ15Q15}(ac, GPR[rs]_{31..16}, GPR[rt]_{31..16}) \\
tempA_{31..0} & \leftarrow \text{sat32AccumulateQ31}(ac, temp) \\
tempB_{63..0} & \leftarrow (tempA_{31})_{32} || tempA_{31..0} \\
(HI[ac]_{..0} || LO[ac]_{..0}) & \leftarrow tempB_{63..32} || tempB_{31..0}
\end{align*}
\]

function \(\text{sat32AccumulateQ31}(acc_{1..0}, a_{31..0})\)

\[
\begin{align*}
signA & \leftarrow a_{31} \\
temp_{.0} & \leftarrow HI[acc]_{.0} || LO[acc]_{.0}
\end{align*}
\]
temp ..0 ← temp + ( (signA) || a31..0 )
if ( temp32 ≠ temp31 ) then
    if ( temp32 = 0 ) then
        temp31..0 ← 0x80000000
    else
        temp31..0 ← 0x7FFFFFFF
    endif
endif
DSPControlouflag:16+acc ← 1
endfunction sat32AccumulateQ31

Exceptions:
Reserved Instruction, DSP Disabled

Programming Notes:
The MAQ_SA version of the instruction is useful for compliance with some ITU speech processing codecs that require a 32-bit saturation after every multiply-accumulate operation.
### MAQ_S[A].W.PHR

Multiply with Accumulate Single Vector Fractional Halfword Element

#### Format:

<table>
<thead>
<tr>
<th>SPECIAL3</th>
<th>rs</th>
<th>rt</th>
<th>0</th>
<th>ac</th>
<th>MAQ_S.W.PHR</th>
<th>DPA.W.PH</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111</td>
<td></td>
<td></td>
<td>0</td>
<td>000</td>
<td>101110</td>
<td>110000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPECIAL3</th>
<th>rs</th>
<th>rt</th>
<th>0</th>
<th>ac</th>
<th>MAQ_SA.W.PHR</th>
<th>DPA.W.PH</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111</td>
<td></td>
<td></td>
<td>0</td>
<td>000</td>
<td>10010</td>
<td>110000</td>
</tr>
</tbody>
</table>

#### Purpose:

Multiply with Accumulate Single Vector Fractional Halfword Element

To multiply one pair of elements from two vectors of fractional halfword values using full-sized intermediate products and accumulate the result into the specified 64-bit accumulator, with optional saturating accumulation.

#### Description:

\[ ac \leftarrow \text{sat32}(ac + \text{sat32}(rs_{15..0} \times rt_{15..0})) \]

The right-most Q15 fractional halfword values from each of the registers \( rt \) and \( rs \) are multiplied together and the product left-shifted by one bit position to generate a Q31 fractional format intermediate result. If both multiplicands are equal to -1.0 in Q15 fractional format (0x8000 hexadecimal), the intermediate result is saturated to the maximum positive Q31 fractional value (0x7FFFFFFF hexadecimal). The intermediate result is then sign-extended and accumulated into accumulator \( ac \) to generate a 64-bit Q32.31 fractional format result.

In the saturating accumulation variant of this instruction, if the accumulation of the intermediate product with the accumulator results in a value that cannot be represented as a Q31 fractional format value, the accumulator is saturated to either the maximum positive Q31 fractional format value (0x7FFFFFFF hexadecimal) or the minimum negative Q31 fractional format value (0x80000000), sign-extended to 64 bits.

The value of \( ac \) can range from 0 to 3; a value of 0 refers to the original HI/LO register pair of the architecture.

If overflow or saturation occurs, a 1 is written to one of bits 16 through 19 of the DSPControl register, within the ouflag field. The value of \( ac \) determines which of these bits is set: bit 16 corresponds to \( ac0 \), bit 17 to \( ac1 \), bit 18 to \( ac2 \), and bit 19 to \( ac3 \).

#### Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the result is **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

#### Operation:

- **MAQ_S.W.PHR**
  
  \[
  \begin{align*}
  \text{tempA}_{31..0} & \leftarrow \text{multiplyQ15Q15}( \ ac, \ GPR[rs]_{15..0}, \ GPR[rt]_{15..0} ) \\
  \text{tempB}_{63..32} & \leftarrow ( \text{HI}[ac]_{31..0} \ || \ \text{LO}[ac]_{31..0} ) + ( (\text{tempA}_{31})^{32} \ || \ \text{tempA}_{31..0} ) \\
  ( \text{HI}[ac]..0 \ || \ \text{LO}[ac]..0 ) & \leftarrow \text{tempB}_{63..32} \ || \ \text{tempB}_{31..0}
  \end{align*}
  \]

- **MAQ_SA.W.PHR**
  
  \[
  \begin{align*}
  \text{tempA}_{31..0} & \leftarrow \text{multiplyQ15Q15}( \ ac, \ GPR[rs]_{15..0}, \ GPR[rt]_{15..0} ) \\
  \text{tempA}_{31..0} & \leftarrow \text{sat32AccumulateQ31}( \ ac, \ \text{temp} ) \\
  \text{tempB}_{63..0} & \leftarrow (\text{tempA}_{31})^{32} \ || \ \text{tempA}_{31..0} \\
  ( \text{HI}[ac]..0 \ || \ \text{LO}[ac]..0 ) & \leftarrow \text{tempB}_{63..32} \ || \ \text{tempB}_{31..0}
  \end{align*}
  \]

#### Exceptions:

Reserved Instruction, DSP Disabled
Programming Notes:

The MAQ_SA version of the instruction is useful for compliance with some ITU speech processing codecs that require a 32-bit saturation after every multiply-accumulate operation.
**MFHI**

**Move from HI register**

**Format:**  
MFHI rd, ac  

**Purpose:**  Move from HI register  
To copy the special purpose HI register to a GPR.

**Description:**  
rd \( \leftarrow \) HI[ac]  
The HI part of accumulator ac is copied to the general-purpose register rd. The HI part of the accumulator is defined to be bits through of the DSP Module accumulator register.

The value of ac can range from 0 to 3. When ac=0, this refers to the original HI/LO register pair of the architecture. In Release 6 of the MIPS Architecture, accumulators are eliminated from MIPS32.

**Restrictions:**  
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**  
GPR[rd]..0 \( \leftarrow \) HI[ac]..0

**Exceptions:**  
Reserved Instruction, DSP Disabled
MFLO Move from LO register

Purpose: Move from LO register
To copy the special purpose LO register to a GPR.

Description: \( \text{rd} \leftarrow \text{LO}[ac] \)
The LO part of accumulator \( ac \) is copied to the general-purpose register \( rd \). The LO part of the accumulator is defined to be bits 0 through \( \text{ac} \) of the DSP Module accumulator register.
The value of \( ac \) can range from 0 to 3. When \( ac=0 \), this refers to the original \( HI/LO \) register pair of the architecture.
In Release 6 of the MIPS Architecture, accumulators are eliminated from MIPS32.

Restrictions:
No data-dependent exceptions are possible.
The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:
\[ \text{GPR}[rd]_{..0} \leftarrow \text{LO}[ac]_{..0} \]

Exceptions:
Reserved Instruction, DSP Disabled
The 32-bit value in register \( rs \) is compared to the value zero. If it is zero, then the index value has reached the bottom of the buffer and must be rolled back around to the top of the buffer. The index value of the top element of the buffer is obtained from bits 8 through 23 in register \( rt \); this value is zero-extended to 0 bits and written to destination register \( rd \).

If the value of register \( rs \) is not zero, then it is simply decremented by the size of the elements in the buffer. The size of the elements, in bytes, is specified by bits 0 through 7 of register \( rt \), interpreted as an unsigned integer.

This instruction does not modify the ouflag field in the DSPControl register.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

\[
\begin{align*}
\text{decr}_{7..0} &\leftarrow \text{GPR}[rt]_{7..0} \\
\text{lastindex}_{15..0} &\leftarrow \text{GPR}[rt]_{23..8} \\
\text{if} \ ( \text{GPR}[rs]_{31..0} = 0x00000000) \ &\text{then} \\
&\quad \text{GPR}[rd]_{.0} \leftarrow \text{GPR}[rs]_{.0} \& \text{GPRLEN-16} \ | | \text{lastindex}_{15..0} \\
\text{else} \\
&\quad \text{GPR}[rd]_{.0} \leftarrow \text{GPR}[rs]_{.0} \& \text{decr}_{7..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
Multiply Word and Subtract from Accumulator

**Format:** MSUB ac, rs, rt

**Purpose:** Multiply Word and Subtract from Accumulator

To multiply two 32-bit integer words and subtract the 64-bit result from the specified accumulator.

**Description:**

\[
(HI[ac] | LO[ac]) \leftarrow (HI[ac] | LO[ac]) - (rs_{31..0} \times rt_{31..0})
\]

The 32-bit signed integer word in register \( rs \) is multiplied by the corresponding 32-bit signed integer word in register \( rt \) to produce a 64-bit result. The 64-bit product is subtracted from the specified 64-bit accumulator.

These special registers \( HI \) and \( LO \) are specified by the value of \( ac \). When \( ac=0 \), this refers to the original \( HI/LO \) register pair of the architecture.

In Release 6 of the MIPS Architecture, accumulators are eliminated from MIPS32.

No arithmetic exception occurs under any circumstances.

**Restrictions:**

This instruction does not provide the capability of writing directly to a target GPR.

**Operation:**

\[
\begin{align*}
temp_{63..0} & \leftarrow ((GPR[rs]_{31})^{32} || GPR[rs]_{31..0}) \times ((GPR[rt]_{31})^{32} || GPR[rt]_{31..0}) \\
acc_{63..0} & \leftarrow (HI[ac]_{31..0} || LO[ac]_{31..0}) - temp_{63..0} \\
& (HI[ac]_{..0} || LO[ac]_{..0}) \leftarrow acc_{63..32} || acc_{31..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled

**Programming Notes:**

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read \( LO \) or \( HI \) before the results are written interlocks 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.

Where the size of the operands are known, software should place the shorter operand in register \( rt \). This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
MSUBU Multiply Unsigned Word and Add to Accumulator

**Format:**  MSUBU ac, rs, rt

**Purpose:** Multiply Unsigned Word and Add to Accumulator

To multiply two 32-bit unsigned integer words and subtract the 64-bit result from the specified accumulator.

**Description:**  
\[(\text{HI}[ac] || \text{LO}[ac]) \leftarrow (\text{HI}[ac] || \text{LO}[ac]) - (rs_{31..0} * rt_{31..0})\]

The 32-bit unsigned integer word in register \(rs\) is multiplied by the corresponding 32-bit unsigned integer word in register \(rt\) to produce a 64-bit result. The 64-bit product is subtracted from the specified 64-bit accumulator.

These special registers \(HI\) and \(LO\) are specified by the value of \(ac\). When \(ac=0\), this refers to the original \(HI/LO\) register pair of the architecture.

In Release 6 of the MIPS Architecture, accumulators are eliminated from MIPS32. No arithmetic exception occurs under any circumstances.

**Restrictions:**
This instruction does not provide the capability of writing directly to a target GPR.

**Operation:**
\[
\begin{align*}
\text{temp}_{64..0} & \leftarrow (0^{32} || \text{GPR}[rs]_{31..0}) * (0^{32} || \text{GPR}[rt]_{31..0}) \\
\text{acc}_{63..0} & \leftarrow (\text{HI}[ac]_{31..0} || \text{LO}[ac]_{31..0}) - \text{temp}_{63..0} \\
(\text{HI}[ac]_{..0} || \text{LO}[ac]_{..0}) & \leftarrow \text{acc}_{63..32} || \text{acc}_{31..0}
\end{align*}
\]

**Exceptions:**
Reserved Instruction, DSP Disabled

**Programming Notes:**
In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read \(LO\) or \(HI\) before the results are written interlocks 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.

Where the size of the operands are known, software should place the shorter operand in register \(rt\). This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
**Format:**  MTHI rs, ac

**Purpose:**  Move to HI register

To copy a GPR to the special purpose HI part of the specified accumulator register.

**Description:**  HI[ac] ← GPR[rs]

The source register rs is copied to the HI part of accumulator ac. The HI part of the accumulator is defined to be bits 0 to 8 of the DSP Module accumulator register.

The value of ac can range from 0 to 3. When ac = 0, this refers to the original HI/LO register pair of the architecture.

In Release 6 of the MIPS Architecture, accumulators are eliminated from MIPS32.

**Restrictions:**

A computed result written to the HI/LO pair by DIV, DIVU, DDIV, DDIVU, DMULT, DMULTU, MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either HI or LO. Note that this restriction applies to the original HI/LO accumulator pair, and does not apply to the new accumulators, ac1, ac2, and ac3.

If an MTHI instruction is executed following one of these arithmetic instructions, but before an MFLO or MFHI instruction, the contents of LO are UNPREDICTABLE. The following example shows this illegal situation:

```
MULT r2,r4  # start operation that will eventually write to HI,LO
...         # code not containing mfhi or mflo
MTHI r6
...         # code not containing mflo
MFLO r3     # this mflo would get an UNPREDICTABLE value
```

**Operation:**

```c
HI[ac]..0 ← GPR[rs]..0
```

**Exceptions:**

Reserved Instruction, DSP Disabled
**MTHLIP**

**Copy LO to HI and a GPR to LO and Increment Pos by 32**

**Format:** MTHLIP rs, ac

**Purpose:** Copy LO to HI and a GPR to LO and Increment Pos by 32

Copy the LO part of an accumulator to the HI part, copy a GPR to LO, and increment the pos field in the DSPControl register by 32.

**Description:**
```
ac ← LO[ac] 31..0 || GPR[rs] 31..0 ; DSPControlpos:..0 += 32
```

The 32 least-significant bits of the specified accumulator are copied to the most-significant bits of the same accumulator. Then the 32 least-significant bits of register rs are copied to the least-significant bits of the accumulator. The instruction then increments the value of bits 0 through of the DSPControl register (the pos field) by 32.

The result of this instruction is **UNPREDICTABLE** if the value of the pos field before the execution of the instruction is greater than 32.

The value of ac can range from 0 to 3. When ac=0, this refers to the original HI/LO register pair of the architecture.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**
```
tempA..0 ← GPR[rs] 31..0
tempB..0 ← LO[ac] 31..0
( HI[ac]..0 || LO[ac]..0 ) ← tempB..0 || tempA..0
oldpos..0 ← DSPControlpos:..0
if ( oldpos..0 > 32 ) then
   DSPControlpos:..0 ← UNPREDICTABLE
else
   DSPControlpos:..0 ← oldpos..0 + 32
endif
```

**Exceptions:**

Reserved Instruction, DSP Disabled
MTLO Move to LO register

**Format:** MTLO rs, ac

**Purpose:** Move to LO register

To copy a GPR to the special purpose LO part of the specified accumulator register.

**Description:**

\[
\text{LO}[ac] \leftarrow \text{GPR}[rs]
\]

The source register \( rs \) is copied to the LO part of accumulator \( ac \). The LO part of the accumulator is defined to be bits 0 to 8 of the DSP Module accumulator register.

The value of \( ac \) can range from 0 to 3. When \( ac=0 \), this refers to the original HI/LO register pair of the architecture. In Release 6 of the MIPS Architecture, accumulators are eliminated from MIPS32.

**Restrictions:**

A computed result written to the HI/LO pair by DIV, DIVU, DDIV, DDIVU, DMULT, DMULTU, MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either HI or LO. Note that this restriction only applies to the original HI/LO accumulator pair, and does not apply to the new accumulators, \( ac1, ac2, \) and \( ac3 \).

If an MTHI instruction is executed following one of these arithmetic instructions, but before an MFLO or MFHI instruction, the contents of LO are UNPREDICTABLE. The following example shows this illegal situation:

```
MULT r2,r4  # start operation that will eventually write to HI,LO
...        # code not containing mfhi or mflo
MTHI r6    # code not containing mflo
MFLO r3    # this mflo would get an UNPREDICTABLE value
```

**Operation:**

\[
\text{LO}[ac]..0 \leftarrow \text{GPR}[rs]..0
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
**MUL[ S].PH**  
Multiply Vector Integer HalfWords to Same Size Products

**Format:**

\[
\begin{array}{cccc}
\text{SPECIAL3} & \text{rs} & \text{rt} & \text{rd} \\
011111 & & & \\
\text{SPECIAL3} & \text{rs} & \text{rt} & \text{rd} \\
011111 & & & \\
\end{array}
\]

MUL.PH  
rd, rs, rt  
MIPSDSP-R2

MUL_S.PH  
rd, rs, rt  
MIPSDSP-R2

**Purpose:** Multiply Vector Integer HalfWords to Same Size Products

Multiply two vector halfword values.

**Description:**

\[
\text{rd} \leftarrow (\text{rs}_{31..16} \times \text{rt}_{31..16}) \text{ || } (\text{rs}_{15..0} \times \text{rt}_{15..0})
\]

Each of the two integer halfword elements in register \(rs\) is multiplied by the corresponding integer halfword element in register \(rt\) to create a 32-bit signed integer intermediate result.

In the non-saturation version of the instruction, the 16 least-significant bits of each 32-bit intermediate result are written to the corresponding vector element in destination register \(rd\).

In the saturating version of the instruction, intermediate results that cannot be represented in 16 bits are clipped to either the maximum positive 16-bit value (0x7FFF hexadecimal) or the minimum negative 16-bit value (0x8000 hexadecimal), depending on the sign of the intermediate result. The saturated results are then written to the destination register.

To stay compliant with the base architecture, this instruction leaves the base \(HI/LO\) pair (accumulator \(ac0\)) \text{UNPREDICTABLE} after the operation completes. The other DSP Module accumulators, \(ac1\), \(ac2\), and \(ac3\), are unchanged.

In the saturating instruction variant, if either multiplication results in an overflow or underflow, the instruction writes a 1 to bit 21 in the \text{ouflag} field in the \text{DSPControl} register.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are \text{UNPREDICTABLE} and the values of the operand vectors become \text{UNPREDICTABLE}.

**Operation:**

\[
\begin{align*}
\text{MUL.PH} \\
\text{tempB}_{31..0} & \leftarrow \text{MultiplyI16I16}( \text{GPR[rs]}_{31..16}, \text{GPR[rt]}_{31..16} ) \\
\text{tempA}_{31..0} & \leftarrow \text{MultiplyI16I16}( \text{GPR[rs]}_{15..0}, \text{GPR[rt]}_{15..0} ) \\
\text{GPR[rd]}_{15..0} & \leftarrow \text{tempB}_{15..0} \text{ || } \text{tempA}_{15..0} \\
\text{HI}_{..0} & \leftarrow \text{UNPREDICTABLE} \\
\text{LO}_{..0} & \leftarrow \text{UNPREDICTABLE} \\
\text{MUL_S.PH} \\
\text{tempB}_{31..0} & \leftarrow \text{sat16MultiplyI16I16}( \text{GPR[rs]}_{31..16}, \text{GPR[rt]}_{31..16} ) \\
\text{tempA}_{31..0} & \leftarrow \text{sat16MultiplyI16I16}( \text{GPR[rs]}_{15..0}, \text{GPR[rt]}_{15..0} ) \\
\text{GPR[rd]}_{15..0} & \leftarrow \text{tempB}_{15..0} \text{ || } \text{tempA}_{15..0} \\
\text{HI}_{..0} & \leftarrow \text{UNPREDICTABLE} \\
\text{LO}_{..0} & \leftarrow \text{UNPREDICTABLE}
\end{align*}
\]

function \text{MultiplyI16I16}( a_{15..0}, b_{15..0} )
temp31..0 ← a15..0 * b15..0
if ( temp31..0 > 0x7FFF ) or ( temp31..0 < 0xFFFF8000 ) then
    DSPControlOutflag:21 ← 1
endif
return temp15..0
endfunction MultiplyI16I16

function satMultiplyI16I16( a15..0, b15..0 )
temp31..0 ← a15..0 * b15..0
if ( temp31..0 > 0x7FFF ) then
    temp31..0 ← 0x00007FFF
    DSPControlOutflag:21 ← 1
else
    if ( temp31..0 < 0xFFFF8000 ) then
        temp31..0 ← 0xFFFF8000
        DSPControlOutflag:21 ← 1
    endif
endif
return temp15..0
endfunction satMultiplyI16I16

Exceptions:
Reserved Instruction, DSP Disabled

Programming Notes:
The base architecture states that upon the after a GPR-targeting multiply instruction such as MUL, the contents of HI and LO are UNPREDICTABLE. To stay compliant with the base architecture, this multiply instruction states the same requirement. But this requirement does not apply to the new accumulators ac1-ac3 and hence a programmer must save the value in ac0 (which is the same as HI and LO) across a GPR-targeting multiply instruction, it needed, while the values in ac1-ac3 do not need to be saved.
MULEQ_S.W.PHL Multiply Vector Fractional Left Halfwords to Expanded Width Products

Format: MULEQ_S.W.PHL rd, rs, rt

Purpose: Multiply Vector Fractional Left Halfwords to Expanded Width Products

Multiply two Q15 fractional halfword values to produce a Q31 fractional word result, with saturation.

Description: rd \leftarrow \text{sat32} \left( rs_{31..16} \times rt_{31..16} \right)

The left-most Q15 fractional halfword value from the paired halfword vector in register rs is multiplied by the corresponding Q15 fractional halfword value from register rt. The result is left-shifted one bit position to create a Q31 format result and written into the destination register rd. If both input values are -1.0 in Q15 format (0x8000 in hexadecimal) the result is clamped to the maximum positive Q31 fractional value (0x7FFFFFFF in hexadecimal) before being written to the destination register.

To stay compliant with the base architecture, this instruction leaves the base HI/LO pair (accumulator ac0) UNPREDICTABLE after the operation completes. The other DSP Module accumulators, ac1, ac2, and ac3 are unmodified.

If the result is saturated, this instruction writes a 1 to bit 21 in the ouflag field of the DSPControl register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

\[
\begin{align*}
temp_{31..0} & \leftarrow \text{multiplyQ15Q15ouflag21} ( GPR[rs]_{31..16}, GPR[rt]_{31..16} ) \\
\text{GPR}[rd]_{..0} & \leftarrow temp_{31..0} \\
\text{HI}[0]_{..0} & \leftarrow \text{UNPREDICTABLE} \\
\text{LO}[0]_{..0} & \leftarrow \text{UNPREDICTABLE} \\

\text{function multiplyQ15Q15ouflag21} ( a_{15..0}, b_{15..0} ) \\
& \quad \text{if} ( a_{15..0} = 0x8000 ) \text{ and } ( b_{15..0} = 0x8000 ) \text{ then} \\
& \quad \quad temp_{31..0} \leftarrow 0x7FFFFFFF \\
& \quad \quad \text{DSPControlouflag:21} \leftarrow 1 \\
& \quad \text{else} \\
& \quad \quad temp_{31..0} \leftarrow ( a_{15..0} \times b_{15..0} ) \ll 1 \\
& \quad \quad \text{return temp}_{31..0} \\
\text{endfunction multiplyQ15Q15ouflag21}
\end{align*}
\]

Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

The base architecture states that after a GPR-targeting multiply instruction such as MUL, the contents of registers HI and LO are UNPREDICTABLE. To maintain compliance with the base architecture this multiply instruction, MULEQ_S.W.PHL, has the same requirement. Software must save and restore the ac0 register if the previous value in the ac0 register is needed following the MULEQ_S.W.PHL instruction.

Note that the requirement on HI and LO does not apply to the new accumulator registers ac1, ac2, and ac3; as a result
the values in these accumulators need not be saved.
MULEQ_S.W.PHR Multiply Vector Fractional Right Halfwords to Expanded Width Products

**Format:**

MULEQ_S.W.PHR rd, rs, rt

**Purpose:**
Multiply Vector Fractional Right Halfwords to Expanded Width Products

Multiply two Q15 fractional halfword values to produce a Q31 fractional word result, with saturation.

**Description:**

rd \(\leftarrow\) sat32\(rs_{15..0} \times rt_{15..0}\)

The right-most Q15 fractional halfword value from register \(rs\) is multiplied by the corresponding Q15 fractional halfword value from register \(rt\). The result is left-shifted one bit position to create a Q31 format result and written into the destination register \(rd\). If both input values are -1.0 in Q15 format (0x8000 in hexadecimal) the result is clamped to the maximum positive Q31 fractional value (0x7FFFFFFF in hexadecimal) before being written to the destination register.

To stay compliant with the base architecture, this instruction leaves the base HI/LO pair (accumulator \(ac0\)) UNPREDICTABLE after the operation completes. The other DSP Module accumulators, \(ac1\), \(ac2\), and \(ac3\) are unmodified.

If the result is saturated, this instruction writes a 1 to bit 21 in the ouflag field of the DSPControl register.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

\[
\text{temp}_{31..0} \leftarrow \text{multiplyQ15Q15ouflag21}(\ GPR[rs]_{15..0}, \ GPR[rt]_{15..0} )
\]

\[
\text{GPR}[rd]_{..0} \leftarrow \text{temp}_{31..0}
\]

\[
\text{HI}[0]_{..0} \leftarrow \text{UNPREDICTABLE}
\]

\[
\text{LO}[0]_{..0} \leftarrow \text{UNPREDICTABLE}
\]

\[
\text{function multiplyQ15Q15ouflag21}(\ a_{15..0}, \ b_{15..0} )
\]

\[
\text{if ( } a_{15..0} = 0x8000 ) \text{ and ( } b_{15..0} = 0x8000 ) \text{ then}
\]

\[
\text{temp}_{31..0} \leftarrow 0x7FFFFFFF
\]

\[
\text{DSPControl.ouflag:21} \leftarrow 1
\]

\[
\text{else}
\]

\[
\text{temp}_{31..0} \leftarrow ( a_{15..0} \times b_{15..0} ) << 1
\]

\[
\text{endif}
\]

\[
\text{return temp}_{31..0}
\]

endfunction multiplyQ15Q15ouflag21

**Exceptions:**

Reserved Instruction, DSP Disabled

**Programming Notes:**

The base architecture states that after a GPR-targeting multiply instruction such as MUL, the contents of registers HI and LO are UNPREDICTABLE. To maintain compliance with the base architecture this multiply instruction, MULEQ_S.W.PHR, has the same requirement. Software must save and restore the \(ac0\) register if the previous value in the \(ac0\) register is needed following the MULEQ_S.W.PHR instruction.
Note that the requirement on $HI$ and $LO$ does not apply to the new accumulator registers $ac1$, $ac2$, and $ac3$; as a result the values in these accumulators need not be saved.
MULEU_S.PH.QBL Multiply Unsigned Vector Left Bytes by Halfwords to Halfword Products

Format: MULEU_S.PH.QBL rd, rs, rt

Purpose: Multiply Unsigned Vector Left Bytes by Halfwords to Halfword Products

Multiply two left-most unsigned byte vector elements in a byte vector by two unsigned halfword vector elements to produce two unsigned halfword results, with saturation.

Description: rd ← sat16(rs31..24 * rt31..16) || sat16(rs23..16 * rt15..0)

The two left-most unsigned byte elements in four-element byte vector in register rs are multiplied as unsigned integer values with the four corresponding unsigned halfword elements from register rt. The eight most-significant bits of each 24-bit result are discarded, and the remaining 16 least-significant bits are written to the corresponding elements in halfword vector register rd. The instruction saturates the result to the maximum positive value (0xFFFF hexadecimal) if any of the discarded bits from each intermediate result are non-zero.

To stay compliant with the base architecture, this instruction leaves the base HI/LO pair (accumulator ac0) UNPREDICTABLE after the operation completes. The other DSP Module accumulators, ac1, ac2, and ac3 are unmodified.

If either result is saturated this instruction writes a 1 to bit 21 in the DSPControl register in the ouflag field.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

```
tempB15..0 ← multiplyU8U16( GPR[rs]31..24, GPR[rt]31..16 )
tempA15..0 ← multiplyU8U16( GPR[rs]23..16, GPR[rt]15..0 )
GPR[rd]..0 ← tempB15..0 || tempA15..0
HI[0]..0 ← UNPREDICTABLE
LO[0]..0 ← UNPREDICTABLE
```

```function multiplyU8U16( a7..0, b15..0 )
temp25..0 ← (0 || a) * (0 || b)
if ( temp25..16 > 0x00 ) then
    temp25..0 ← 010 || 0xFFFF
    DSPControl.ouflag.21 ← 1
endif
return temp15..0
endfunction multiplyU8U16```

Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

The base architecture states that after a GPR-targeting multiply instruction such as MUL, the contents of registers HI and LO are UNPREDICTABLE. To maintain compliance with the base architecture this multiply instruction, MULEU_S.PH.QBL, has the same requirement. Software must save and restore the ac0 register if the previous value in the ac0 register is needed following the MULEU_S.PH.QBL instruction.
Note that the requirement on HI and LO does not apply to the new accumulator registers ac1, ac2, and ac3; as a result the values in these accumulators need not be saved.
Multiply Unsigned Vector Right Bytes with halfwords to Half Word Products

Format: MULEU_S.PH.QBR rd, rs, rt

Purpose: Multiply Unsigned Vector Right Bytes with halfwords to Half Word Products

Element-wise multiplication of unsigned byte elements with corresponding unsigned halfword elements, with saturation.

Description: rd ← sat16(rs15..8 * rt31..16) || sat16(rs7..0 * rt15..0)

The two right-most unsigned byte elements in four-element byte vector in register rs are multiplied as unsigned integer values with the corresponding right-most 16-bit unsigned values from register rt. Each result is clipped to preserve the 16 least-significant bits and written back into the respective halfword element positions in the destination register rd. The instruction saturates the result to the maximum positive value (0xFFFF hexadecimal) if any of the clipped bits are non-zero.

To stay compliant with the base architecture, this instruction leaves the base HI/LO pair (accumulator ac0) UNPREDICTABLE after the operation completes. The other DSP Module accumulators, ac1, ac2, and ac3 must be unmodified.

This instruction writes a 1 to bit 21 in the ouflag field in the DSPControl register if either multiplication results in saturation.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

```
tempB15..0 ← multiplyU8U16( GPR[rs]15..8, GPR[rt]31..16 )
tempA15..0 ← multiplyU8U16( GPR[rs]7..0, GPR[rt]15..0 )
GPR[rd] ← tempB15..0 || tempA15..0
HI[0]..0 ← UNPREDICTABLE
LO[0]..0 ← UNPREDICTABLE
```

Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

The base architecture states that after a GPR-targeting multiply instruction such as MUL, the contents of registers HI and LO are UNPREDICTABLE. To maintain compliance with the base architecture this multiply instruction, MULEU_S.PH.QBR, has the same requirement. Software must save and restore the ac0 register if the previous value in the ac0 register is needed following the MULEU_S.PH.QBR instruction.

Note that the requirement on HI and LO does not apply to the new accumulator registers ac1, ac2, and ac3; as a result the values in these accumulators need not be saved.
Multiply Vector Fractional Halfwords to Fractional Halfword Products

**Format:**  
MULQ_RS.PH rd, rs, rt

**Purpose:** Multiply Vector Fractional Halfwords to Fractional Halfword Products

Multiply Q15 fractional halfword vector elements with rounding and saturation to produce two Q15 fractional halfword results.

**Description:**  
\[ \text{rd} \leftarrow \text{rndQ15}(rs_{31..16} \times rt_{31..16}) || \text{rndQ15}(rs_{15..0} \times rt_{15..0}) \]

The two Q15 fractional halfword elements from register rs are separately multiplied by the corresponding Q15 fractional halfword elements from register rt to produce 32-bit intermediate results. Each intermediate result is left-shifted by one bit position to produce a Q31 fractional value, then rounded by adding 0x00008000 hexadecimal. The rounded intermediate result is then truncated to a Q15 fractional value and written to the corresponding position in destination register rd.

If the two input values to either multiplication are both -1.0 (0x8000 in hexadecimal), the final halfword result is saturated to the maximum positive Q15 value (0x7FFF in hexadecimal) and rounding and truncation are not performed.

To stay compliant with the base architecture, this instruction leaves the base HI/LO pair (accumulator ac0) UNPREDICTABLE after the operation completes. The other DSP Module accumulators, ac1, ac2, and ac3 must be unmodified.

If either result is saturated this instruction writes a 1 to bit 21 in the DSPControl register in the ouflag field.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

```
tempB_{15..0} \leftarrow \text{rndQ15MultiplyQ15Q15}( GPR[rs]_{31..16}, GPR[rt]_{31..16} )
tempA_{15..0} \leftarrow \text{rndQ15MultiplyQ15Q15}( GPR[rs]_{15..0}, GPR[rt]_{15..0} )
GPR[rd]_{..0} \leftarrow tempB_{15..0} || tempA_{15..0}
HI[0]_{..0} \leftarrow \text{UNPREDICTABLE}
LO[0]_{..0} \leftarrow \text{UNPREDICTABLE}
```

function rndQ15MultiplyQ15Q15( a_{15..0}, b_{15..0} )

```
if ( a_{15..0} = 0x8000 ) and ( b_{15..0} = 0x8000 ) then
    temp_{31..0} \leftarrow 0x7FFF0000
    DSPControl_ouflag:21 \leftarrow 1
else
    temp_{31..0} \leftarrow ( a_{15..0} \times b_{15..0} ) << 1
    temp_{31..0} \leftarrow temp_{31..0} + 0x00000800
endif
return temp_{31..16}
endfunction rndQ15MultiplyQ15Q15
```

**Exceptions:**

Reserved Instruction, DSP Disabled
Programming Notes:

The base architecture states that after a GPR-targeting multiply instruction such as MUL, the contents of registers $HI$ and $LO$ are UNPREDICTABLE. To maintain compliance with the base architecture, this multiply instruction, MULQ_RS.PH, has the same requirement. Software must save and restore the $ac0$ register if the previous value in the $ac0$ register is needed following the MULQ_RS.PH instruction.

Note that the requirement on $HI$ and $LO$ does not apply to the new accumulator registers $ac1$, $ac2$, and $ac3$; as a result, the values in these accumulators need not be saved.
MULQ_RS.W Multiply Fractional Words to Same Size Product with Saturation and Rounding

**Format:**  MULQ_RS.W rd, rs, rt

**Purpose:** Multiply Fractional Words to Same Size Product with Saturation and Rounding

Multiply fractional Q31 word values, with saturation and rounding.

**Description:**  

\[ rd \leftarrow \text{round(sat32}(rs_{31..0} \times rt_{31..0})) \]

The Q31 fractional format words in registers `rs` and `rt` are multiplied together and the product shifted left by one bit position to create a 64-bit fractional format intermediate result. The intermediate result is rounded up by adding a 1 at bit position 31, and then truncated by discarding the 32 least-significant bits to create a 32-bit fractional format result. The result is then written to destination register `rd`.

If both input multiplicands are equal to -1 (0x80000000 hexadecimal), rounding is not performed and the maximum positive Q31 fractional format value (0x7FFFFFFF hexadecimal) is written to the destination register.

To stay compliant with the base architecture, this instruction leaves the base HI/LO pair (accumulator `ac0`) UNPREDICTABLE after the operation completes. The other DSP Module accumulators, `ac1`, `ac2`, and `ac3`, are unchanged.

This instruction, on an overflow or underflow of the operation, writes a 1 to bit 21 in the `DSPControl` register in the `ouflag` field.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

```plaintext
if ( GPR[rs]_{31..0} = 0x80000000 ) and ( GPR[rt]_{31..0} = 0x80000000 ) then
    temp_{63..0} \leftarrow 0x7FFFFFFFF00000000
    DSPControl.ouflag:21 \leftarrow 1
else
    temp_{63..0} \leftarrow ( GPR[rs]_{31..0} \times GPR[rt]_{31..0} ) \ll 1
    temp_{63..0} \leftarrow temp_{63..0} + ( 0^{32} || 0x80000000 )
endif
GPR[rd]_{..0} \leftarrow temp_{63..32}
HI[0]_{..0} \leftarrow UNPREDICTABLE
LO[0]_{..0} \leftarrow UNPREDICTABLE
```

**Exceptions:**

Reserved Instruction, DSP Disabled

**Programming Notes:**

The base architecture states that after a GPR-targeting multiply instruction such as MUL, the contents of registers HI and LO are UNPREDICTABLE. To maintain compliance with the base architecture, this multiply instruction, MULQ_RS.W, has the same requirement. Software must save and restore the `ac0` register if the previous value in the `ac0` register is needed following the MULQ_RS.W instruction.

Note that the requirement on HI and LO does not apply to the new accumulator registers `ac1`, `ac2`, and `ac3`; as a result, the values in these accumulators need not be saved.
Multiply two vector fractional Q15 values to create a Q15 result, with saturation.

The two vector fractional Q15 values in register `rs` are multiplied with the corresponding elements in register `rt` to produce two 32-bit products. Each product is left-shifted by one bit position to create a Q31 fractional word intermediate result. The two 32-bit intermediate results are then each truncated by discarding the 16 least-significant bits of each result, and the resulting Q15 fractional format halfwords are then written to the corresponding positions in destination register `rd`. For each halfword result, if both input multiplicands are equal to -1 (0x8000 hexadecimal), the final halfword result is saturated to the maximum positive Q15 value (0x7FFF hexadecimal).

To stay compliant with the base architecture, this instruction leaves the base `HI/LO` pair (accumulator `ac0`) UNPREDICTABLE after the operation completes. The other DSP Module accumulators, `ac1`, `ac2`, and `ac3`, must be untouched.

This instruction, on an overflow or underflow of any one of the two vector operation, writes bit 21 in the `ouflag` field in the `DSPControl` register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

```
function sat16MultiplyQ15Q15( a15..0, b15..0 )
    if ( a15..0 = 0x8000 ) and ( b15..0 = 0x8000 ) then
        temp31..0 ← 0x7FFF0000
        DSPControl.ouflag:21 ← 1
    else
        temp31..0 ← ( a15..0 * b15..0 )
        temp31..0 ← ( temp30..0 || 0 )
    endif
    return temp31..16
endfunction sat16MultiplyQ15Q15
```
Programming Notes:

The base architecture states that after a GPR-targeting multiply instruction such as MUL, the contents of registers HI and LO are UNPREDICTABLE. To maintain compliance with the base architecture, this multiply instruction, MULQ_S.PH, has the same requirement. Software must save and restore the ac0 register if the previous value in the ac0 register is needed following the MULQ_S.PH instruction.

Note that the requirement on HI and LO does not apply to the new accumulator registers ac1, ac2, and ac3; as a result, the values in these accumulators need not be saved.
Multiply Fractional Words to Same Size Product with Saturation

**Format:**  
\[
\text{MULQ\_S\_W}\ \text{rd},\ \text{rs},\ \text{rt}
\]

**Purpose:**  
Multiply two Q31 fractional format word values to create a fractional Q31 result, with saturation.

**Description:**  
\[
\text{rd} \leftarrow \text{sat32}(\text{rs}_{31..0} \times \text{rt}_{31..0})
\]

The Q31 fractional format words in registers \(\text{rs}\) and \(\text{rt}\) are multiplied together to create a 64-bit fractional format intermediate result. The intermediate result is left-shifted by one bit position, and then truncated by discarding the 32 least-significant bits to create a Q31 fractional format result. This result is then written to destination register \(\text{rd}\).

If both input multiplicands are equal to -1 (0x80000000 hexadecimal), the product is clipped to the maximum positive Q31 fractional format value (0x7FFFFFFF hexadecimal), and written to the destination register.

To stay compliant with the base architecture, this instruction leaves the base \(HI/LO\) pair (accumulator \(ac0\)) UNPREDICTABLE after the operation completes. The other DSP Module accumulators, \(ac1\), \(ac2\), and \(ac3\), are unchanged.

This instruction, on an overflow or underflow of the operation, writes a 1 to bit 21 in the \(DSPControl\) register in the \(ouflag\) field.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

\[
\begin{align*}
\text{if} \ ( \ GPR[\text{rs}]_{31..0} = 0x80000000 \ ) \ \text{and} \ ( \ GPR[\text{rt}]_{31..0} = 0x80000000 \ ) \ \text{then} \\
\text{temp}_{63..0} \leftarrow 0x7FFFFFFFFFF00000000 \\
\text{DSPControl}_{\text{ouflag}}.21 \leftarrow 1 \\
\text{else} \\
\text{temp}_{63..0} \leftarrow ( GPR[\text{rs}]_{31..0} \times GPR[\text{rt}]_{31..0} ) << 1 \\
\text{endif} \\
GPR[\text{rd}]_{..0} \leftarrow \text{temp}_{63..32} \\
\text{HI}[0]_{..0} \leftarrow \text{UNPREDICTABLE} \\
\text{LO}[0]_{..0} \leftarrow \text{UNPREDICTABLE}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled

**Programming Notes:**

The base architecture states that after a GPR-targeting multiply instruction such as MUL, the contents of registers \(HI\) and \(LO\) are UNPREDICTABLE. To maintain compliance with the base architecture, this multiply instruction, MULQ\_S\_W, has the same requirement. Software must save and restore the \(ac0\) register if the previous value in the \(ac0\) register is needed following the MULQ\_S\_W instruction.

Note that the requirement on \(HI\) and \(LO\) does not apply to the new accumulator registers \(ac1\), \(ac2\), and \(ac3\); as a result, the values in these accumulators need not be saved.
MULSA.W.PH Multiply and Subtract Vector Integer Halfword Elements and Accumulate

Format: MULSA.W.PH ac, rs, rt

Purpose: Multiply and Subtract Vector Integer Halfword Elements and Accumulate

To multiply and subtract two integer vector elements using full-size intermediate products, accumulating the result into the specified accumulator.

Description:

\[ \text{ac} \leftarrow \text{ac} + \left( (\text{rs}_{31..16} \times \text{rt}_{31..16}) - (\text{rs}_{15..0} \times \text{rt}_{15..0}) \right) \]

Each of the two halfword integer elements from register \( rt \) are multiplied by the corresponding elements in \( rs \) to create two word results. The right-most result is subtracted from the left-most result to generate the intermediate result, which is then added to the specified 64-bit accumulator.

The value of \( ac \) selects an accumulator numbered from 0 to 3. When \( ac=0 \), this refers to the original \( HI/LO \) register pair of the architecture.

This instruction does not set any bits of the \( outflag \) field in the \( DSPControl \) register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the result is \textbf{UNPREDICTABLE} and the values of the operand vectors become \textbf{UNPREDICTABLE}.

Operation:

\[
\begin{align*}
\text{tempB}_{31..0} & \leftarrow (\text{GPR}[\text{rs}]_{31..16} \times \text{GPR}[\text{rt}]_{31..16}) \\
\text{tempA}_{31..0} & \leftarrow (\text{GPR}[\text{rs}]_{15..0} \times \text{GPR}[\text{rt}]_{15..0}) \\
\text{dotp}_{32..0} & \leftarrow (\text{tempB}_{31} || \text{tempB}_{31..0}) - (\text{tempA}_{31} || \text{tempA}_{31..0}) \\
\text{acc}_{63..0} & \leftarrow (\text{HI}[\text{ac}]_{31..0} || \text{LO}[\text{ac}]_{31..0}) + (\text{dotp}_{32} || \text{dotp}_{32..0}) \\
& (\text{HI}[\text{ac}]_{..0} || \text{LO}[\text{ac}]_{..0}) \leftarrow \text{acc}_{63..32} || \text{acc}_{31..0}
\end{align*}
\]

Exceptions:

Reserved Instruction, DSP Disabled
MULSAQ_S.W.PH  Multiply And Subtract Vector Fractional Halfwords And Accumulate

Format:  MULSAQ_S.W.PH ac, rs, rt

Purpose:  Multiply And Subtract Vector Fractional Halfwords And Accumulate

Multiply and subtract two Q15 fractional halfword vector elements using full-size intermediate products, accumulating the result from the specified accumulator, with saturation.

Description:  \( ac \leftarrow ac + (\text{sat32}(rs_{31..16} \times rt_{31..16}) - \text{sat32}(rs_{15..0} \times rt_{15..0})) \)

The two corresponding Q15 fractional values from registers \( rt \) and \( rs \) are multiplied together and left-shifted by 1 bit to generate two Q31 fractional format intermediate products. If the input multiplicands to either of the multiplications are both -1.0 (0x8000 hexadecimal), the intermediate result is saturated to 0x7FFFFFFF hexadecimal.

The two intermediate products (named left and right) are summed with alternating sign to create a sum-of-products, i.e., the sign of the right product is negated before summation. The sum-of-products is then sign-extended to 64 bits and accumulated into the specified 64-bit accumulator, producing a Q32.31 result.

The value of \( ac \) can range from 0 to 3; a value of 0 refers to the original HI/LO register pair of the architecture.

If saturation occurs, a 1 is written to one of bits 16 through 19 of the DSPControl register, within the ouflag field. The value of \( ac \) determines which of these bits is set: bit 16 corresponds to \( ac0 \), bit 17 to \( ac1 \), bit 18 to \( ac2 \), and bit 19 to \( ac3 \).

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

\[
\begin{align*}
\text{tempB}_{31..0} & \leftarrow \text{multiplyQ15Q15}( ac, rs_{31..16}, rt_{31..16} ) \\
\text{tempA}_{31..0} & \leftarrow \text{multiplyQ15Q15}( ac, rs_{15..0}, rt_{15..0} ) \\
\text{dotp}_{63..0} & \leftarrow (\text{tempB}_{31})^{32} || \text{tempB}_{31..0} - (\text{tempA}_{31})^{32} || \text{tempA}_{31..0} \\
\text{tempC}_{63..0} & \leftarrow (\text{HI}[ac]_{31..0} || \text{LO}[ac]_{31..0}) + \text{dotp}_{63..0} \\
(\text{HI}[ac]_{..0} || \text{LO}[ac]_{..0}) & \leftarrow \text{tempC}_{63..32} || \text{tempC}_{31..0}
\end{align*}
\]

Exceptions:

Reserved Instruction, DSP Disabled
MULT Multiply Word

MIPS® DSP Module for MIPS32™ Architecture, Revision 3.01

Format: MULT ac, rs, rt

MIPS32 pre-Release 6, MIPSDSP

Purpose: Multiply Word

To multiply two 32-bit signed integers, writing the 64-bit result to the specified accumulator.

Description: ac ← rs31..0 * rt31..0

The 32-bit signed integer value in register rt is multiplied by the corresponding 32-bit signed integer value in register rs, to produce a 64-bit result that is written to the specified accumulator register.

The value of ac selects an accumulator numbered from 0 to 3. When ac=0, this refers to the original HI/LO register pair of the architecture.

In Release 6 of the MIPS Architecture, accumulators are eliminated from MIPS32.

No arithmetic exception occurs under any circumstances.

Restrictions:

Operation:

\[
t_{63..0} \leftarrow ((GPR[rs]_{31})^{32} \ || \ GPR[rs]_{31..0}) \times ((GPR[rt]_{31})^{32} \ || \ GPR[rt]_{31..0})
\]

\[
(\text{HI}[ac]_{..0} \ || \ \text{LO}[ac]_{..0}) \leftarrow t_{63..32} \ || \ t_{31..0}
\]

Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read LO or HI before the results are written interlocks 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.

Where the size of the operands are known, software should place the shorter operand in register rt. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
MULTU Multiply Unsigned Word

164 MIPS® DSP Module for MIPS32™ Architecture, Revision 3.01

Format: MULTU ac, rs, rt

MIPS32 pre-Release 6, MIPSDSP

Purpose: Multiply Unsigned Word

To multiply 32-bit unsigned integers, writing the 64-bit result to the specified accumulator.

Description: ac ← rs31..0 * rt31..0

The 32-bit unsigned integer value in register rt is multiplied by the corresponding 32-bit unsigned integer value in register rs, to produce a 64-bit unsigned result that is written to the specified accumulator register.

The value of ac selects an accumulator numbered from 0 to 3. When ac=0, this refers to the original HI/LO register pair of the architecture.

In Release 6 of the MIPS Architecture, accumulators are eliminated from MIPS32.

No arithmetic exception occurs under any circumstances.

Restrictions:

Operation:

\[
t_{64..0} \leftarrow (0^{32} \mid | \text{GPR}[rs]_{31..0}) * (0^{32} \mid | \text{GPR}[rt]_{31..0})
\]

\[
(\text{HI}[ac]_{..0} \mid | \text{LO}[ac]_{..0}) \leftarrow t_{63..32} \mid | t_{31..0}
\]

Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

In some processors the integer multiply operation may proceed asynchronously and allow other CPU instructions to execute before it is complete. An attempt to read LO or HI before the results are written interlocks 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.

Where the size of the operands are known, software should place the shorter operand in register rt. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
PACKRL.PH Pack a Vector of Halfwords from Vector Halfword Sources

**Format:** PACKRL.PH rd, rs, rt

**Purpose:** Pack a Vector of Halfwords from Vector Halfword Sources

Pick two elements for a halfword vector using the right halfword and left halfword respectively from the two source registers.

**Description:** rd $\leftarrow$ rs$_{15..0}$ $||$ rt$_{31..16}$

The right halfword element from register rs and the left halfword register rt are packed into the two halfword positions of the destination register rd.

**Restrictions:**
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

\[
\begin{align*}
\text{tempB}_{15..0} & \leftarrow \text{GPR}[rs]_{15..0} \\
\text{tempA}_{15..0} & \leftarrow \text{GPR}[rt]_{31..16} \\
\text{GPR}[rd]_{..0} & \leftarrow \text{tempB}_{15..0} || \text{tempA}_{15..0}
\end{align*}
\]

**Exceptions:**
Reserved Instruction, DSP Disabled
PICK.PH  Pick a Vector of Halfword Values Based on Condition Code Bits

Format:  PICK.PH rd, rs, rt

Purpose:  Pick a Vector of Halfword Values Based on Condition Code Bits

Select two halfword elements from either of two source registers based on condition code bits, writing the selected elements to the destination register.

Description:  rd ← pick(cc25,rs31..16,rt31..16) || pick(cc24,rs15..0,rt15..0)

The two right-most condition code bits in the DSPControl register are used to select halfword values from the corresponding element of either source register rs or source register rt. If the value of the corresponding condition code bit is 1, then the halfword value is selected from register rs; otherwise, it is selected from rt. The selected halfwords are written to the destination register rd.

Restrictions:

No data-dependent exceptions are possible.

The operands must be in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

\[
\begin{align*}
\text{tempB}_{15..0} & \leftarrow \begin{cases} 
\text{DSPControl}_{ccod:25} = 1 \, ? \, \text{GPR}[rs]_{31..16} : \text{GPR}[rt]_{31..16} 
\end{cases} \\
\text{tempA}_{15..0} & \leftarrow \begin{cases} 
\text{DSPControl}_{ccod:24} = 1 \, ? \, \text{GPR}[rs]_{15..0} : \text{GPR}[rt]_{15..0} 
\end{cases} \\
\text{GPR}[rd]_{..0} & \leftarrow \text{tempB}_{15..0} || \text{tempA}_{15..0}
\end{align*}
\]

Exceptions:

Reserved Instruction, DSP Disabled
PICK.QB Pick a Vector of Byte Values Based on Condition Code Bits

**Format:** PICK.QB rd, rs, rt

**Purpose:** Pick a Vector of Byte Values Based on Condition Code Bits

Select four byte elements from either of two source registers based on condition code bits, writing the selected elements to the destination register.

**Description:**

\[
\text{rd} \leftarrow \text{pick}(cc_{27}, rs_{31..24}, rt_{31..24}) \ || \ \text{pick}(cc_{26}, rs_{23..16}, rt_{23..16}) \ || \\
\text{pick}(cc_{25}, rs_{15..8}, rt_{15..8}) \ || \ \text{pick}(cc_{24}, rs_{7..0}, rt_{7..0})
\]

Four condition code bits in the **DSPControl** register are used to select byte values from the corresponding byte element of either source register *rs* or source register *rt*. If the value of the corresponding condition code bit is 1, then the byte value is selected from register *rs*; otherwise, it is selected from *rt*. The selected bytes are written to the destination register *rd*.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

\[
\begin{align*}
\text{tempD}_{7..0} & \leftarrow ( \text{DSPControl}_{ccond:27} = 1 \ ? \ \text{GPR}[rs]_{31..24} : \ \text{GPR}[rt]_{31..24} ) \\
\text{tempC}_{7..0} & \leftarrow ( \text{DSPControl}_{ccond:26} = 1 \ ? \ \text{GPR}[rs]_{23..16} : \ \text{GPR}[rt]_{23..16} ) \\
\text{tempB}_{7..0} & \leftarrow ( \text{DSPControl}_{ccond:25} = 1 \ ? \ \text{GPR}[rs]_{15..8} : \ \text{GPR}[rt]_{15..8} ) \\
\text{tempA}_{7..0} & \leftarrow ( \text{DSPControl}_{ccond:24} = 1 \ ? \ \text{GPR}[rs]_{7..0} : \ \text{GPR}[rt]_{7..0} ) \\
\text{GPR}[rd]_{7..0} & \leftarrow \text{tempD}_{7..0} \ || \ \text{tempC}_{7..0} \ || \ \text{tempB}_{7..0} \ || \ \text{tempA}_{7..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
**PRECEQ.W.PHL**

**Precision Expand Fractional Halfword to Fractional Word Value**

**Format:**  
PRECEQ.W.PHL rd, rt  

**Purpose:**  
Precision Expand Fractional Halfword to Fractional Word Value

Expand the precision of a Q15 fractional value taken from the left element of a paired halfword vector to create a Q31 fractional word value.

**Description:**  
\[ \text{rd} \leftarrow \text{expand_prec} (\text{rt}_{31..16}) \]

The left Q15 fractional halfword value from the register rt is expanded to a Q31 fractional value and written to destination register rd. The precision expansion is achieved by appending 16 least-significant zero bits to the original halfword value to generate the 32-bit fractional value.

**Restrictions:**
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**
\[
\begin{align*}
\text{temp}_{31..0} & \leftarrow \text{GPR}[\text{rt}]_{31..16} || 0^{16} \\
\text{GPR}[\text{rd}]_{..0} & \leftarrow \text{temp}_{31..0}
\end{align*}
\]

**Exceptions:**
Reserved Instruction, DSP Disabled
**PRECEQ.W.PHR**

**Precision Expand Fractional Halfword to Fractional Word Value**

**Format:**  
PRECEQ.W.PHR  \( \text{rd, rt} \)

**Purpose:** Precision Expand Fractional Halfword to Fractional Word Value  
Expand the precision of a Q15 fractional value taken from the right element of a paired halfword vector to create a Q31 fractional word value.

**Description:**  
\( \text{rd} \leftarrow \text{expand_prec} (\text{rt}_{15..0}) \)  
The right Q15 fractional halfword value from the \( \text{rt} \) register is expanded to a Q31 fractional value and written to destination register \( \text{rd} \). The precision expansion is achieved by appending 16 least-significant zero bits to the original halfword value to generate the 32-bit fractional value.

**Restrictions:**  
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**  
\[
\text{temp}_{31..0} \leftarrow \text{GPR}[\text{rt}]_{15..0} \mid 0^{16} \\
\text{GPR}[\text{rd}]_{..0} \leftarrow \text{temp}_{31..0}
\]

**Exceptions:**  
Reserved Instruction, DSP Disabled
**Format:**  
PRECEQU.PH.QBL rd, rt  

**Purpose:**  
Precision Expand two Unsigned Bytes to Fractional Halfword Values  

Expand the precision of two unsigned byte values taken from the two left-most elements of a quad byte vector to create two Q15 fractional halfword values.

**Description:**  
rd ← expand_prec(rt31..24) || expand_prec(rt23..16)  
The two left-most unsigned integer byte values from the four byte elements in register rt are expanded to create two Q15 fractional values that are then written to destination register rd. The precision expansion is achieved by prepending a single zero bit (for positive sign) to the original byte value and appending seven least-significant zeros to generate each 16-bit fractional value.

**Restrictions:**  
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

\[
\begin{align*}
\text{tempB}_{15..0} &\leftarrow 0^1 || \text{GPR}[rt]_{31..24} || 0^7 \\
\text{tempA}_{15..0} &\leftarrow 0^1 || \text{GPR}[rt]_{23..16} || 0^7 \\
\text{GPR}[rd]_{15..0} &\leftarrow \text{tempB}_{15..0} || \text{tempA}_{15..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
PRECEQU.PH.QBLA  

**Precision Expand two Unsigned Bytes to Fractional Halfword Values**

### Format:

PRECEQU.PH.QBLA rd, rt  

### Purpose:

Precision Expand two Unsigned Bytes to Fractional Halfword Values

Expand the precision of two unsigned byte values taken from the two left-alternate aligned elements of a quad byte vector to create two Q15 fractional halfword values.

### Description:

\[
\text{rd} \leftarrow \text{expand_prec}(rt_{31..24}) \ || \ \text{expand_prec}(rt_{15..8})
\]

The two left-alternate aligned unsigned integer byte values from the four byte elements in register rt are expanded to create two Q15 fractional values that are then written to destination register rd. The precision expansion is achieved by pre-pending a single zero bit (for positive sign) to the original byte value and appending seven least-significant zeros to generate each 16-bit fractional value.

### Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

### Operation:

\[
\begin{align*}
\text{tempB}_{15..0} & \leftarrow 0^7 || \text{GPR}[rt]_{31..24} || 0^7 \\
\text{tempA}_{15..0} & \leftarrow 0^7 || \text{GPR}[rt]_{15..8} || 0^7 \\
\text{GPR}[rd]_{15..0} & \leftarrow \text{tempB}_{15..0} || \text{tempA}_{15..0}
\end{align*}
\]

### Exceptions:

Reserved Instruction, DSP Disabled
PRECEQU.PH.QBR Precision Expand two Unsigned Bytes to Fractional Halfword Values

Format: PRECEQU.PH.QBR rd, rt

Purpose: Precision Expand two Unsigned Bytes to Fractional Halfword Values

Expand the precision of two unsigned byte values taken from the two right-most elements of a quad byte vector to create two Q15 fractional halfword values.

Description: rd ← expand_prec(rt15..8) || expand_prec(rt7..0)

The two right-most unsigned integer byte values from the four byte elements in register rt are expanded to create two Q15 fractional values that are then written to destination register rd. The precision expansion is achieved by prepending a single zero bit (for positive sign) to the original byte value and appending seven least-significant zeros to generate each 16-bit fractional value.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

\[
\begin{align*}
\text{tempB}_{15..0} & \leftarrow 0^1 || \text{GPR}[rt]_{15..8} || 0^7 \\
\text{tempA}_{15..0} & \leftarrow 0^1 || \text{GPR}[rt]_{7..0} || 0^7 \\
\text{GPR}[rd]_{15..0} & \leftarrow \text{tempB}_{15..0} || \text{tempA}_{15..0}
\end{align*}
\]

Exceptions:

Reserved Instruction, DSP Disabled
**PRECEQU.PH.QBRA**

**Precision Expand two Unsigned Bytes to Fractional Halfword Values**

**Format:**  
PRECEQU.PH.QBRA rd, rt

**Purpose:** Precision Expand two Unsigned Bytes to Fractional Halfword Values

Expand the precision of two unsigned byte values taken from the two right-alternate aligned elements of a quad byte vector to create two Q15 fractional halfword values.

**Description:**  
rd = expand_prec(rt23..16) || expand_prec(rt7..0)

The two right-alternate aligned unsigned integer byte values from the four byte elements in register rt are expanded to create two Q15 fractional values that are then written to destination register rd. The precision expansion is achieved by pre-pending a single zero bit (for positive sign) to the original byte value and appending seven least-significant zeros to generate each 16-bit fractional value.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

```
tempB15..0 ← 01 || GPR[rt]23..16 || 07
tempA15..0 ← 01 || GPR[rt]7..0 || 07
GPR[rd]..0 ← tempB15..0 || tempA15..0
```

**Exceptions:**

Reserved Instruction, DSP Disabled
Precision Expand Two Unsigned Bytes to Unsigned Halfword Values

**Format:**  
PRECEU.PH.QBL rd, rt

**Purpose:** Precision Expand Two Unsigned Bytes to Unsigned Halfword Values

Expand the precision of two unsigned byte values taken from the two left-most elements of a quad byte vector to create two unsigned halfword values.

**Description:**  
rd $\leftarrow$ expand_prec8u16(rt$^{31..24}$) $||$ expand_prec8u16(rt$^{23..16}$)

The two left-most unsigned integer byte values from the four byte elements in register rt are expanded to create two unsigned halfword values that are then written to destination register rd. The precision expansion is achieved by prepending eight most-significant zeros to each original value to generate each 16 bit unsigned value.

**Restrictions:**
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

- `tempB_{15..0} \leftarrow 0^8 || GPR[rt]_{31..24}`
- `tempA_{15..0} \leftarrow 0^8 || GPR[rt]_{23..16}`
- `GPR[rd]_{15..0} \leftarrow tempB_{15..0} || tempA_{15..0}`

**Exceptions:**

Reserved Instruction, DSP Disabled
**Format:** PRECEU.PH.QBLA rd, rt

**Purpose:** Precision Expand Two Unsigned Bytes to Unsigned Halfword Values

Expand the precision of two unsigned integer byte values taken from the two left-alternate aligned positions of a quad byte vector to create four unsigned halfword values.

**Description:**

\[
\text{rd} \leftarrow \text{expand\_prec8u16}(rt_{31..24}) \ | \ | \text{expand\_prec8u16}(rt_{15..8})
\]

The two left-alternate aligned unsigned integer byte values from the four right-most byte elements in register \(rt\) are each expanded to unsigned halfword values and written to destination register \(rd\). The precision expansion is achieved by pre-pending eight most-significant zero bits to the original byte value to generate each 16 bit unsigned halfword value.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

\[
\begin{align*}
temp_{15..0} &\leftarrow 0^8 \ | \ | GPR[rt]_{31..24} \\
temp_{15..0} &\leftarrow 0^8 \ | \ | GPR[rt]_{15..8} \\
GPR[rd]_{15..0} &\leftarrow temp_{15..0} \ | \ | temp_{15..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
**Format:** PRECEU.PH.QBR rd, rt

**Purpose:** Precision Expand two Unsigned Bytes to Unsigned Halfword Values

Expand the precision of two unsigned integer byte values taken from the two right-most elements of a quad byte vector to create two unsigned halfword values.

**Description:**

\[
\text{rd} \leftarrow \text{expand_prec8u16}(rt_{15..8}) \ || \ \text{expand_prec8u16}(rt_{7..0})
\]

The two right-most unsigned integer byte values from the four byte elements in register \(rt\) are expanded to create two unsigned halfword values that are then written to destination register \(rd\). The precision expansion is achieved by prepending eight most-significant zero bits to each original value to generate each 16 bit halfword value.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are \text{UNPREDICTABLE} and the values of the operand vectors become \text{UNPREDICTABLE}.

**Operation:**

\[
\begin{align*}
\text{tempB}_{15..0} & \leftarrow 0^8 \ || \ \text{GPR}[rt]_{15..8} \\
\text{tempA}_{15..0} & \leftarrow 0^8 \ || \ \text{GPR}[rt]_{7..0} \\
\text{GPR}[rd]_{..0} & \leftarrow \text{tempB}_{15..0} \ || \ \text{tempA}_{15..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
Precision Expand Two Unsigned Bytes to Unsigned Halfword Values

**Format:** PRECEU.PH.QBRA rd, rt

**Purpose:** Precision Expand Two Unsigned Bytes to Unsigned Halfword Values

Expand the precision of two unsigned byte values taken from the two right-alternate aligned positions of a quad byte vector to create two unsigned halfword values.

**Description:**

```plaintext
drd \leftarrow \text{expand\_prec8u16(rt_{23..16})} \ || \ \text{expand\_prec8u16(rt_{7..0})}
```

The two right-alternate aligned unsigned integer byte values from the four byte elements in register `rt` are each expanded to unsigned halfword values and written to destination register `rd`. The precision expansion is achieved by pre-pending eight most-significant zero bits to the original byte value to generate each 16 bit unsigned halfword value.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

```plaintext
\text{tempB}_{15..0} \leftarrow 0^8 \ || \ \text{GPR[rt]}_{23..16} \\
\text{tempA}_{15..0} \leftarrow 0^8 \ || \ \text{GPR[rt]}_{7..0} \\
\text{GPR[rd]}_{15..0} \leftarrow \text{tempB}_{15..0} \ || \ \text{tempA}_{15..0}
```

**Exceptions:**

Reserved Instruction, DSP Disabled
PRECR.QB.PH  Precision Reduce Four Integer Halfwords to Four Bytes

**Format:**  PRECR.QB.PH  rd, rs, rt

**Purpose:** Precision Reduce Four Integer Halfwords to Four Bytes

Reduce the precision of four integer halfwords to four byte values.

**Description:**

\[
rd \leftarrow rs_{23..16} || rs_{7..0} || rt_{23..16} || rt_{7..0}
\]

The 8 least-significant bits from each of the two integer halfword values in registers \(rs\) and \(rt\) are taken to produce four byte-sized results that are written to the four byte elements in destination register \(rd\). The two bytes values obtained from \(rs\) are written to the two left-most destination byte elements, and the two bytes obtained from \(rt\) are written to the two right-most destination byte elements.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

\[
\begin{align*}
tempD_{7..0} &\leftarrow GPR[rs]_{23..16} \\
tempC_{7..0} &\leftarrow GPR[rs]_{7..0} \\
tempB_{7..0} &\leftarrow GPR[rt]_{23..16} \\
tempA_{7..0} &\leftarrow GPR[rt]_{7..0} \\
GPR[rd]_{7..0} &\leftarrow tempD_{7..0} || tempC_{7..0} || tempB_{7..0} || tempA_{7..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
**Purpose:** Precision Reduce Two Integer Words to Halfwords after a Right Shift

Do an arithmetic right shift of two integer words with optional rounding, and then reduce the precision to halfwords.

**Description:**

\[ rt \leftarrow (\text{round}(rt>>\text{shift}))_{15..0} \mid | \mid (\text{round}(rs>>\text{shift}))_{15..0} \]

The two words in registers \( rs \) and \( rt \) are right shifted arithmetically by the specified shift amount \( sa \) to create interim results. The 16 least-significant bits of each interim result are then written to the corresponding elements of destination register \( rt \).

In the rounding version of the instruction, a value of 1 is added at the most-significant discarded bit position after the shift is performed. The 16 least-significant bits of each interim result are then written to the corresponding elements of destination register \( rt \).

The shift amount \( sa \) is interpreted as a five-bit unsigned integer taking values between 0 and 31.

This instruction does not write any bits of the ouflag field in the DSPControl register.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

### PRECR_SRA.PH.W

```plaintext
if (sa_{4..0} = 0) then
    tempB_{15..0} \leftarrow \text{GPR}[rt]_{15..0}
    tempA_{15..0} \leftarrow \text{GPR}[rs]_{15..0}
else
    tempB_{15..0} \leftarrow (\text{GPR}[rt]_{31})^{sa} | | \text{GPR}[rt]_{31..sa} 
    tempA_{15..0} \leftarrow (\text{GPR}[rs]_{31})^{sa} | | \text{GPR}[rs]_{31..sa}
endif
GPR[rt]_{..0} \leftarrow tempB_{15..0} | | tempA_{15..0}
```

### PRECR_SRA_R.PH.W

```plaintext
if (sa_{4..0} = 0) then
    tempB_{16..0} \leftarrow (\text{GPR}[rt]_{15..0} | | 0)
    tempA_{16..0} \leftarrow (\text{GPR}[rs]_{15..0} | | 0)
else
    tempB_{32..0} \leftarrow (\text{GPR}[rt]_{31})^{sa} | | \text{GPR}[rt]_{31..sa-1} + 1 
    tempA_{32..0} \leftarrow (\text{GPR}[rs]_{31})^{sa} | | \text{GPR}[rs]_{31..sa-1} + 1
endif
GPR[rt]_{..0} \leftarrow tempB_{16..1} | | tempA_{16..1}
```
Exceptions:
Reserved Instruction, DSP Disabled
Precision Reduce Fractional Words to Fractional Halfwords

**Format:** PRECRQ.PH.W rd, rs, rt

**Purpose:** Precision Reduce Fractional Words to Fractional Halfwords
Reduce the precision of two fractional words to produce two fractional halfword values.

**Description:**
\[ rd \leftarrow rt_{31..16} \| rs_{31..16} \]
The 16 most-significant bits from each of the Q31 fractional word values in registers \( rs \) and \( rt \) are written to destination register \( rd \), creating a vector of two Q15 fractional values. The fractional word from the \( rs \) register is used to create the left-most Q15 fractional value in \( rd \), and the fractional word from the \( rt \) register is used to create the right-most Q15 fractional value.

**Restrictions:**
No data-dependent exceptions are possible.
The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**
\[
\begin{align*}
tempB_{15..0} & \leftarrow GPR[rs]_{31..16} \\
tempA_{15..0} & \leftarrow GPR[rt]_{31..16} \\
GPR[rd]_{15..0} & \leftarrow tempB_{15..0} \| tempA_{15..0}
\end{align*}
\]

**Exceptions:**
Reserved Instruction, DSP Disabled
Format:  PRECRQ.QB.PH rd, rs, rt

Purpose:  Precision Reduce Four Fractional Halfwords to Four Bytes
Reduce the precision of four fractional halfwords to four byte values.

Description:  rd ← rs31..24 || rs15..8 || rt31..24 || rt15..8
The Q15 fractional values in registers rs and rt are truncated by dropping the eight least significant bits from each value to produce four fractional byte values. The four fractional byte values are written to the four byte elements of destination register rd. The two values obtained from register rt are placed in the two right-most byte positions in the destination register, and the two values obtained from register rs are placed in the two remaining byte positions.

Restrictions:
No data-dependent exceptions are possible.
The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:
\[
\begin{align*}
tempD_{7..0} & \leftarrow \text{GPR}[rs]_{31..24} \\
tempC_{7..0} & \leftarrow \text{GPR}[rs]_{15..8} \\
tempB_{7..0} & \leftarrow \text{GPR}[rt]_{31..24} \\
tempA_{7..0} & \leftarrow \text{GPR}[rt]_{15..8} \\
\text{GPR}[rd]_{0..7} & \leftarrow tempD_{7..0} || tempC_{7..0} || tempB_{7..0} || tempA_{7..0}
\end{align*}
\]

Exceptions:
Reserved Instruction, DSP Disabled
**PRECRQU_S.QB.PH**

**Precision Reduce Fractional Halfwords to Unsigned Bytes With Saturation**

**Format:**

```plaintext
PRECRQU_S.QB.PH rd, rs, rt
```

**Purpose:**

Precision Reduce Fractional Halfwords to Unsigned Bytes With Saturation

Reduce the precision of four fractional halfwords with saturation to produce four unsigned byte values, with saturation.

**Description:**

```plaintext
rd ← sat(reduce_prec(rs31..16)) || sat(reduce_prec(rs15..0)) ||
      sat(reduce_prec(rt31..16)) || sat(reduce_prec(rt15..0))
```

The four Q15 fractional halfwords from registers `rs` and `rt` are used to create four unsigned byte values that are written to corresponding elements of destination register `rd`. The two halfwords from the `rs` register and the two halfwords from the `rt` register are used to create the four unsigned byte values.

Each unsigned byte value is created from the Q15 fractional halfword input value after examining the sign and magnitude of the halfword. If the sign of the halfword value is positive and the value is greater than 0x7F80 hexadecimal, the result is clamped to the maximum positive 8-bit value (255 decimal, 0xFF hexadecimal). If the sign of the halfword value is negative, the result is clamped to the minimum positive 8-bit value (0 decimal, 0x00 hexadecimal). Otherwise, the sign bit is discarded from the input and the result is taken from the eight most-significant bits that remain.

If clamping was needed to produce any of the unsigned output values, bit 22 of the `DSPControl` register is set to 1.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

```plaintext
function sat8ReducePrecision( a15..0 )
    sign ← a15
    mag14..0 ← a14..0
    if ( sign = 0 ) then
        if ( mag14..0 > 0x7F80 ) then
            temp7..0 ← 0xFF
            DSPControl.outflag:22 ← 1
        else
            temp7..0 ← mag14..7
        endif
    else
        temp7..0 ← 0x00
        DSPControl.outflag:22 ← 1
    endif
endfunction
```

```plaintext
tempD7..0 ← sat8ReducePrecision( GPR[rs]31..16 )
tempC7..0 ← sat8ReducePrecision( GPR[rs]15..0 )
tempB7..0 ← sat8ReducePrecision( GPR[rt]31..16 )
tempA7..0 ← sat8ReducePrecision( GPR[rt]15..0 )
GPR[rd]..0 ← tempD7..0 || tempC7..0 || tempB7..0 || tempA7..0
```

MIPS® DSP Module for MIPS32™ Architecture, Revision 3.01  183
endfunction sat8ReducePrecision

Exceptions:
Reserved Instruction, DSP Disabled
**Precision Reduce Fractional Words to Halfwords With Rounding and Saturation**

**Format:** PRECRQ_RS.PH.W rd, rs, rt

**Purpose:** Precision Reduce Fractional Words to Halfwords With Rounding and Saturation

Reduce the precision of two fractional words to produce two fractional halfword values, with rounding and saturation.

**Description:**

\[
rd \leftarrow \text{truncQ15SatRound}(rs_{31..0}) \mid \text{truncQ15SatRound}(rt_{31..0})
\]

The two Q31 fractional word values in each of registers rs and rt are used to create two Q15 fractional halfword values that are written to the two halfword elements in destination register rd. The fractional word from the rs register is used to create the left-most Q15 fractional halfword result in rd, and the fractional word from the rt register is used to create the right-most halfword value.

Each input Q31 fractional value is rounded and saturated before being truncated to create the Q15 fractional halfword result. First, the value 0x00008000 is added to the input Q31 value to round even, creating an interim rounded result. If this addition causes overflow, the interim rounded result is saturated to the maximum Q31 value (0x7FFFFFFF hexadecimal). Then, the 16 least-significant bits of the interim rounded and saturated result are discarded and the 16 most-significant bits are written to the destination register in the appropriate position.

If either of the rounding operations results in overflow and saturation, a 1 is written to bit 22 in the DSPControl register within the ouflag field.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

\[
\begin{align*}
\text{tempB}_{15..0} & \leftarrow \text{trunc16Sat16Round}( \text{GPR}[rs]_{31..0} ) \\
\text{tempA}_{15..0} & \leftarrow \text{trunc16Sat16Round}( \text{GPR}[rt]_{31..0} ) \\
\text{GPR}[rd]_{..0} & \leftarrow \text{tempB}_{15..0} \mid \text{tempA}_{15..0} \\
\text{DSPControl}_{ouflag:22} & \leftarrow 1 \\
\end{align*}
\]

function trunc16Sat16Round( a_{31..0} )

\[
\begin{align*}
\text{temp}_{32..0} & \leftarrow ( a_{31} \mid a_{31..0} ) + 0x0000080000 \\
\text{if ( temp}_{32} \neq \text{temp}_{31} ) \text{ then} \\
\text{temp}_{32..0} & \leftarrow 0 \mid 0x7FFFFFFF \\
\text{DSPControl}_{ouflag:22} & \leftarrow 1 \\
\text{endif} \\
\text{return temp}_{31..16}
\end{align*}
\]

endfunction trunc16Sat16Round

**Exceptions:**

Reserved Instruction, DSP Disabled
**PREPEND**

**Right Shift and Prepend Bits to the MSB**

---

<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>SPECIAL3</td>
<td></td>
<td>rs</td>
<td>rt</td>
<td>sa</td>
<td>PREPEND</td>
<td>APPEND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>00001</td>
<td>110001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** PREPEND rt, rs, sa

**Purpose:** Right Shift and Prepend Bits to the MSB

Logically right-shift the first source register, replacing the bits emptied by the shift with bits from the source register.

**Description:** rt ← rs_{sa-1..0} || (rt >> sa)

The word value in register rt is logically right-shifted by the specified shift amount sa, and sa bits from the least-significant positions of register rs are written into the sa most-significant bits emptied by the shift. The result is then written to destination register rt.

**Restrictions:**
No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

```plaintext
if ( sa_{4..0} = 0 ) then
    temp_{31..0} ← GPR[rt]_{31..0}
else
    temp_{31..0} ← ( GPR[rs]_{sa-1..0} || GPR[rt]_{31..sa} )
endif
GPR[rt]_{..0} = temp_{31..0}
```

**Exceptions:**
Reserved Instruction, DSP Disabled
RADDU.W.QB

Unsigned Reduction Add Vector Quad Bytes

Format: RADDU.W.QB rd, rs

Purpose: Unsigned Reduction Add Vector Quad Bytes

Reduction add of four unsigned byte values in a vector register to produce an unsigned word result.

Description: rd ← zero_extension(rs31..24 + rs23..16 + rs15..8 + rs7..0)

The unsigned byte elements in register rs are added together as unsigned 8-bit values, and the result is zero extended to a word and written to register rd.

Restrictions:

No data-dependent exceptions are possible.

The operands must be in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

\[
\text{temp}_{9..0} \leftarrow (0^2 || \text{GPR}[rs]_{31..24}) + (0^2 || \text{GPR}[rs]_{23..16}) + (0^2 || \text{GPR}[rs]_{15..8}) + (0^2 || \text{GPR}[rs]_{7..0})
\]

\[
\text{GPR}[rd]_{11..0} \leftarrow 0^{(\text{GPRLEN}-10)} || \text{temp}_{9..0}
\]

Exceptions:

Reserved Instruction, DSP Disabled
**RDDSP**

**Purpose:** Read DSPControl Register Fields to a GPR

To copy selected fields from the special-purpose *DSPControl* register to the specified GPR.

**Description:** $rd \leftarrow \text{select}(mask, \text{DSPControl})$

Selected fields in the special register *DSPControl* are copied into the corresponding bits of destination register $rd$. Each of bits 0 through 5 of the *mask* operand corresponds to a specific field in the *DSPControl* register. A mask bit value of 1 indicates that the bits from the corresponding field in *DSPControl* will be copied into the same bit positions in register $rd$, and a mask bit value of 0 indicates that the corresponding bit positions in $rd$ will be set to zero. Bits 6 through 9 of the *mask* operand are ignored.

The table below shows the correspondence between the bits in the *mask* operand and the fields in the *DSPControl* register; mask bit 0 is the least-significant bit in *mask*.

For example, to copy only the bits from the scount field in *DSPControl*, the value of the *mask* operand used will be 2 decimal (0x02 hexadecimal). After execution of the instruction, bits 7 through 12 of register $rd$ will have the value of bits 7 through 12 from the scount field in *DSPControl*. The remaining bits in register $rd$ will be set to zero.

The one-operand version of the instruction provides a convenient assembly idiom that allows the programmer to read all fields in the *DSPControl* register into the destination register, i.e., it is equivalent to specifying a *mask* value of 31 decimal (0x1F hexadecimal).

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

```plaintext
    temp..0  \leftarrow  0
    if ( mask_0 = 1 ) then
        temp..0  \leftarrow  \text{DSPControl}_{pos:...0}
    endif
    if ( mask_1 = 1 ) then
        temp_{12..7}  \leftarrow  \text{DSPControl}_{scount:12..7}
    endif
    if ( mask_2 = 1 ) then
        temp_{13}  \leftarrow  \text{DSPControl}_{c:13}
    endif
    if ( mask_3 = 1 ) then
        temp_{23..16}  \leftarrow  \text{DSPControl}_{ouflag:23..16}
    endif
    if ( mask_4 = 1 ) then
        temp_{24}  \leftarrow  \text{DSPControl}_{ccond:24}
    endif
```
if ( mask₅ = 1 ) then
    temp₁₄ ← DSPControlₑ₁₄
endif

GPR[rd]₀₀ ← temp₀₀

Exceptions:
Reserved Instruction, DSP Disabled
Replicate Immediate Integer into all Vector Element Positions

**Format:** REPL.PH rd, immediate

**Purpose:** Replicate Immediate Integer into all Vector Element Positions

Replicate a sign-extended, 10-bit signed immediate integer value into the two halfwords in a halfword vector.

**Description:** rd ← sign_extend(immediate) || sign_extend(immediate)

The specified 10-bit signed immediate integer value is sign-extended to 16 bits and replicated into the two halfword positions in destination register rd.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

\[
\text{temp}_{15..0} \leftarrow (\text{immediate}_9)^6 || \text{immediate}_9..0 \\
\text{GPR}[\text{rd}]_{..0} \leftarrow \text{temp}_{15..0} || \text{temp}_{15..0}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
**REPL.QB**

Replicate Immediate Integer into all Vector Element Positions

**Format:**
```plaintext
REPL.QB rd, immediate
```

**Purpose:** Replicate Immediate Integer into all Vector Element Positions

Replicate a immediate byte into all elements of a quad byte vector.

**Description:**
```plaintext
rd ← immediate || immediate || immediate || immediate
```

The specified 8-bit signed immediate value is replicated into the four byte elements of destination register `rd`.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**
```plaintext
temp7..0 ← immediate7..0
GPR[rd]..0 ← temp7..0 || temp7..0 || temp7..0 || temp7..0
```

**Exceptions:**

Reserved Instruction, DSP Disabled
REPLV.PH  Replicate a Halfword into all Vector Element Positions

Format: REPLV.PH  rd, rt

Purpose: Replicate a Halfword into all Vector Element Positions
Replicate a variable halfword into the elements of a halfword vector.

Description: rd ← rt15..0 || rt15..0)
The halfword value in register rt is replicated into the two halfword elements of destination register rd.

Restrictions:
No data-dependent exceptions are possible.
The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:
    temp15..0 ← GPR[rt]15..0
    GPR[rd]15..0 ← temp15..0 || temp15..0

Exceptions:
Reserved Instruction, DSP Disabled
**REPLV.QB**

Replicate Byte into all Vector Element Positions

**Format:** REPLV.QB rd, rt

**Purpose:** Replicate Byte into all Vector Element Positions
Replicate a variable byte into all elements of a quad byte vector.

**Description:**
rd \(\leftarrow rt_{7..0} || rt_{7..0} || rt_{7..0} || rt_{7..0}\)

The byte value in register \(rt\) is replicated into the four byte elements of destination register \(rd\).

**Restrictions:**
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

\[
\begin{align*}
temp_{7..0} & \leftarrow GPR[rt]_{7..0} \\
GPR[rd]_{7..0} & \leftarrow temp_{7..0} || temp_{7..0} || temp_{7..0} || temp_{7..0}
\end{align*}
\]

** Exceptions:**
Reserved Instruction, DSP Disabled

<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>SPECIAL3</td>
<td>011111</td>
<td>0</td>
<td>00000</td>
<td>rt</td>
<td>rd</td>
<td>REPLV.QB</td>
<td>00011</td>
<td>ABSQ_S.PH</td>
<td>010010</td>
<td>MIPS DSP</td>
<td></td>
</tr>
</tbody>
</table>
SHILO  Shift an Accumulator Value Leaving the Result in the Same Accumulator

Format:  SHILO ac, shift

Purpose: Shift an Accumulator Value Leaving the Result in the Same Accumulator
Shift the HI/LO paired value in a 64-bit accumulator either left or right, leaving the result in the same accumulator.

Description: ac ← (shift >= 0) ? (ac >> shift) : (ac << -shift)
The HI/LO register pair is treated as a single 64-bit accumulator that is shifted logically by shift bits, with the result of the shift written back to the source accumulator. The shift argument is a six-bit signed integer value: a positive argument results in a right shift of up to 31 bits, and a negative argument results in a left shift of up to 32 bits.
The value of ac can range from 0 to 3. When ac=0, this refers to the original HI/LO register pair of the architecture.

Restrictions:
No data-dependent exceptions are possible.
The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

sign ← shift5
shift5..0 ← ( sign = 0 ? shift5..0 : -shift5..0 )
if ( shift5..0 = 0 ) then
    temp63..0 ← (HI[ac]31..0 || LO[ac]31..0)
else
    if (sign = 0) then
        temp63..0 ← 0shift || (( HI[ac]31..0 || LO[ac]31..0 ) >> shift )
    else
        temp63..0 ← (( HI[ac]31..0 || LO[ac]31..0 ) << shift ) || 0shift
    endif
endif
( HI[ac]..0 || LO[ac]..0 ) ← temp63..32 || temp31..0

Exceptions:
Reserved Instruction, DSP Disabled
**Format:** SHILOV ac, rs

**Purpose:** Variable Shift of Accumulator Value Leaving the Result in the Same Accumulator

Shift the HI/LO paired value in an accumulator either left or right by the amount specified in a GPR, leaving the result in the same accumulator.

**Description:**

\[
ac \leftarrow (GPR[rs]_{6..0} \geq 0) \ ? \ (ac >> GPR[rs]_{6..0}) : (ac << -GPR[rs]_{6..0})
\]

The HI/LO register pair is treated as a single 64-bit accumulator that is shifted logically by \(shift\) bits, with the result of the shift written back to the source accumulator. The \(shift\) argument is provided by the six least-significant bits of register \(rs\); the remaining bits of \(rs\) are ignored. The \(shift\) argument is interpreted as a six-bit signed integer: a positive argument results in a right shift of up to 31 bits, and a negative argument results in a left shift of up to 32 bits.

The value of \(ac\) can range from 0 to 3. When \(ac=0\), this refers to the original HI/LO register pair of the architecture.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

\[
\begin{align*}
\text{sign} & \leftarrow GPR[rs]_5 \\
\mathrm{shift}_{5..0} & \leftarrow (\text{sign} = 0 \ ? \ GPR[rs]_{5..0} : -GPR[rs]_{5..0}) \\
\text{if (} \mathrm{shift}_{5..0} = 0 \text{) then} & \\
& \quad \text{temp}_{63..0} \leftarrow (\text{HI}[ac]_{31..0} \ || \ \text{LO}[ac]_{31..0}) \\
\text{else} & \\
& \quad \text{if (} \text{sign} = 0 \text{) then} \\
& \quad & \quad \text{temp}_{63..0} \leftarrow 0^{\text{shift}} \ || \ ((\text{HI}[ac]_{31..0} \ || \ \text{LO}[ac]_{31..0}) >> \text{shift}) \\
& \quad & \quad \text{else} \\
& \quad & \quad \quad \text{temp}_{63..0} \leftarrow ((\text{HI}[ac]_{31..0} \ || \ \text{LO}[ac]_{31..0}) << \text{shift}) \ || 0^{\text{shift}} \\
& \quad & \quad \text{endif} \\
& \quad \text{endif} \\
& \quad (\text{HI}[ac]_{31..0} \ || \ \text{LO}[ac]_{31..0}) \leftarrow \text{temp}_{63..32} \ || \ \text{temp}_{31..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
SHLL_[S].PH  Shift Left Logical Vector Pair Halfwords

Format:

SHLL_[S].PH
SHLL.PH  rd, rt, sa
SHLL_S.PH  rd, rt, sa
MIPSDSP

Purpose: Shift Left Logical Vector Pair Halfwords
Element-wise shift of two independent halfwords in a vector data type by a fixed number of bits, with optional saturation.

Description: rd ← sat16( rt31..16 << sa ) || ( rt15..0 << sa )
The two halfword values in register rt are each independently shifted left, inserting zeros into the least-significant bit positions emptied by the shift. In the saturating version of the instruction, if the shift results in an overflow the intermediate result is saturated to either the maximum positive or the minimum negative 16-bit value, depending on the sign of the original unshifted value. The two independent results are then written to the corresponding halfword elements of destination register rd.

This instruction writes a 1 to bit 22 in the DSPControl register in the ouflag field if any of the left shift operations results in an overflow or saturation.

Restrictions:
No data-dependent exceptions are possible.
The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

SHLL.PH

tempB15..0 ← shift16Left( GPR[rt]31..16, sa )
tempA15..0 ← shift16Left( GPR[rt]15..0, sa )
GPR[rd]..0 ← tempB15..0 || tempA15..0

SHLL_S.PH

tempB15..0 ← sat16ShiftLeft( GPR[rt]31..16, sa )
tempA15..0 ← sat16ShiftLeft( GPR[rt]15..0, sa )
GPR[rd]..0 ← tempB15..0 || tempA15..0

function shift16Left( a15..0, s3..0 )
if ( s3..0 = 0 ) then
    temp15..0 ← a15..0
else
    sign ← a15
    temp15..0 ← ( a15..-s || 0^s )
    discard15..0 ← ( sign |16-s || a14..14-(s-1) )
    if ( ( discard15..0 ≠ 0xFFFF ) and ( discard15..0 ≠ 0xPPPP ) ) then
        DSPControl.ouflag:22 ← 1
    endif
endif
endif
return temp15..0
endfunction shift16Left

function sat16ShiftLeft( a15..0, s3..0 )
    if ( s3..0 = 0 ) then
        temp15..0 ← a15..0
    else
        sign ← a15
        temp15..0 ← ( a15-s..0 || 0s )
        discard15..0 ← ( sign[16-s] || a14..14-(s-1) )
        if (( discard15..0 ≠ 0x0000 ) and ( discard15..0 ≠ 0xFFFF )) then
            temp15..0 ← ( sign = 0 ? 0x7FFF : 0x8000 )
        DSPControlouflag:22 ← 1
    endif
    return temp15..0
endfunction sat16ShiftLeft

Exceptions:

Reserved Instruction, DSP Disabled
Format: SHLL.QB rd, rt, sa

Purpose: Shift Left Logical Vector Quad Bytes

Element-wise left shift of four independent bytes in a vector data type by a fixed number of bits.

Description: rd  (rt31..24 << sa) || (rt23..16 << sa) || (rt15..8 << sa) || (rt7..0 << sa)

The four byte values in register rt are each independently shifted left by sa bits and the sa least significant bits of each value are set to zero. The four independent results are then written to the corresponding byte elements of destination register rd.

This instruction writes a 1 to bit 22 in the DSPControl register in the ouflag field if any of the left shift operations results in an overflow.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

```plaintext
function shift8Left( a7..0, s2..0 )
    if ( s2..0 = 0 ) then
        temp7..0 ← a7..0
    else
        sign ← a7
        temp7..0 ← ( a7..s..0 || 0s )
        discard7..0 ← ( sign [8-s] || a6..6-(s-1) )
        if ( discard7..0 ≠ 0x00 ) then
            DSPControl[ouflag:22] ← 1
        endif
    endif
    return temp7..0
endfunction shift8Left
```

Exceptions:

Reserved Instruction, DSP Disabled
SHLLV_[S].PH  
Shift Left Logical Variable Vector Pair Halfwords

Format:  
SHLLV[_S].PH  
SHLLV.PH  
rd, rt, rs  
MIPSDSP  
SHLLV_S.PH  
rd, rt, rs  
MIPSDSP

Purpose:  Shift Left Logical Variable Vector Pair Halfwords
Element-wise left shift of the two right-most independent halfwords in a vector data type by a variable number of bits, with optional saturation.

Description:  
rd ← sat16(rt31..16 << rs3..0) || sat16(rt15..0 << rs3..0)

The two halfword values in register rt are each independently shifted left by shift bits, inserting zeros into the least-significant bit positions emptied by the shift. In the saturating version of the instruction, if the shift results in an overflow the intermediate result is saturated to either the maximum positive or the minimum negative 16-bit value, depending on the sign of the original unshifted value. The two independent results are then written to the corresponding halfword elements of destination register rd.

The four least-significant bits of rs provide the shift value, interpreted as a four-bit unsigned integer; the remaining bits of rs are ignored.

This instruction writes a 1 to bit 22 in the DSPControl register in the ouflag field if any of the left shift operations results in an overflow or saturation.

Restrictions:  
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:  
SHLLV.PH  
tempB15..0 ← shift16Left( GPR[rt]31..16, GPR[rs]3..0 )  
tempA15..0 ← shift16Left( GPR[rt]15..0, GPR[rs]3..0 )  
GPR[rd]..0 ← tempB15..0 || tempA15..0

SHLLV_S.PH  
tempB15..0 ← sat16ShiftLeft( GPR[rt]31..16, GPR[rs]3..0 )  
tempA15..0 ← sat16ShiftLeft( GPR[rt]15..0, GPR[rs]3..0 )  
GPR[rd]..0 ← tempB15..0 || tempA15..0

Exceptions:  
Reserved Instruction, DSP Disabled
### SHLLV.QB

**Shifting Left Logical Variable Vector Quad Bytes**

**Format:**

```
SHLLV.QB rd, rt, rs
```

**Purpose:**

Shift Left Logical Variable Vector Quad Bytes

Element-wise left shift of four independent bytes in a vector data type by a variable number of bits.

**Description:**

\[
rd \leftarrow (rt_{31..24} \ll rs_{2..0}) \mid (rt_{23..16} \ll rs_{2..0}) \mid (rt_{15..8} \ll rs_{2..0}) \mid (rt_{7..0} \ll rs_{2..0})
\]

The four byte values in register \( rt \) are each independently shifted left by \( sa \) bits, inserting zeros into the least-significant bit positions emptied by the shift. The four independent results are then written to the corresponding byte elements of destination register \( rd \).

The three least-significant bits of \( rs \) provide the shift value, interpreted as a three-bit unsigned integer; the remaining bits of \( rs \) are ignored.

This instruction writes a 1 to bit 22 in the \textit{DSPControl} register in the \textit{ouflag} field if any of the left shift operations results in an overflow.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are \textit{UNPREDICTABLE} and the values of the operand vectors become \textit{UNPREDICTABLE}.

**Operation:**

\[
\begin{align*}
tempD_{7..0} & \leftarrow \text{shift8Left}( GPR[rt]_{31..24}, GPR[rs]_{2..0} ) \\
tempC_{7..0} & \leftarrow \text{shift8Left}( GPR[rt]_{23..16}, GPR[rs]_{2..0} ) \\
tempB_{7..0} & \leftarrow \text{shift8Left}( GPR[rt]_{15..8}, GPR[rs]_{2..0} ) \\
tempA_{7..0} & \leftarrow \text{shift8Left}( GPR[rt]_{7..0}, GPR[rs]_{2..0} ) \\
gpr[rd]_{7..0} & \leftarrow tempD_{7..0} \mid tempC_{7..0} \mid tempB_{7..0} \mid tempA_{7..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
**Format:**  SHLLV_S.W rd, rt, rs  

**Purpose:**  Shift Left Logical Variable Vector Word  
A left shift of the word in a vector data type by a variable number of bits, with optional saturation.

**Description:**  
\[ rd \leftarrow \text{sat32}(rt_{31..0} \ll rs_{4..0}) \]  
The word element in register \( rt \) is shifted left by \( shift \) bits, inserting zeros into the least-significant bit positions emptied by the shift. If the shift results in an overflow the intermediate result is saturated to either the maximum positive or the minimum negative 32-bit value, depending on the sign of the original unshifted value.

The shifted result is then written to destination register \( rd \).

The five least-significant bits of \( rs \) are used as the shift value, interpreted as a five-bit unsigned integer; the remaining bits of \( rs \) are ignored.

This instruction writes a 1 to bit 22 in the \( DSPControl \) register in the \( ouflag \) field if either of the left shift operations results in an overflow or saturation.

**Restrictions:**  
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**  
\[ \begin{align*} 
\text{temp}_{31..0} & \leftarrow \text{sat32ShiftLeft}( \text{GPR}[rt]_{31..0}, \text{GPR}[rs]_{4..0} ) \\
\text{GPR}[rd]_{..0} & \leftarrow \text{temp}_{31..0} 
\end{align*} \]  

**Exceptions:**  
Reserved Instruction, DSP Disabled
**SHLL_S.W**

**Shift Left Logical Word with Saturation**

**Format:**  
\[ \text{SHLL\_S}\_W \text{ rd, rt, sa} \]

**Purpose:**  
Shift Left Logical Word with Saturation

To execute a left shift of a word with saturation by a fixed number of bits.

**Description:**  
\[ \text{rd} \leftarrow \text{sat32(rt} \ll \text{sa)} \]

The 32-bit word in register \( rt \) is shifted left by \( sa \) bits, with zeros inserted into the bit positions emptied by the shift. If the shift results in a signed overflow, the shifted result is saturated to either the maximum positive (hexadecimal \( 0x7FFFFFFF \)) or minimum negative (hexadecimal \( 0x80000000 \)) 32-bit value, depending on the sign of the original unshifted value. The shifted result is then written to destination register \( rd \).

The instruction’s \( sa \) field specifies the shift value, interpreted as a five-bit unsigned integer.

If the shift operation results in an overflow and saturation, this instruction writes a 1 to bit 22 of the ** DSPControl ** register within the ** ouflag ** field.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

\[
\text{temp}_{31..0} \leftarrow \text{sat32ShiftLeft( GPR[rt]31..0, sa4..0 )}
\]

GPR[rd]..0 \( \leftarrow \text{temp}_{31..0} \)

function sat32ShiftLeft( \( a_{13..0}, s_{4..0} \) )

if \( ( s = 0 ) \) then
  \( \text{temp}_{31..0} \leftarrow a \)
else
  \( \text{sign} \leftarrow a_{31} \)
  \( \text{temp}_{31..0} \leftarrow ( a_{31-s..0} \oplus 0^s ) \)
  \( \text{discard}_{31..0} \leftarrow ( \text{sign}^{132-s} \oplus a_{30..30-(s-1)} ) \)
  if \( ( ( \text{discard}_{31..0} \neq 0x00000000 ) \) and \( ( \text{discard}_{31..0} \neq 0xFFFFFFFF ) \) then
    \( \text{temp}_{31..0} \leftarrow ( \text{sign} = 0 \ ? \ 0x7FFFFFFF : 0x80000000 ) \)
    \( \text{DSPControl}_{ouflag:22} \leftarrow 1 \)
  endif
endif
return \( \text{temp}_{31..0} \)
endfunction sat32ShiftLeft

**Exceptions:**

Reserved Instruction, DSP Disabled

**Programming Notes:**

To do a logical left shift of a word in a register without saturation, use the **SLL** instruction.
Format:  
SHRA[[_R].QB
SHRA.QB  rd, rt, sa
SHRA_R.QB rd, rt, sa
MIPSDSP-R2

Purpose: Shift Right Arithmetic Vector of Four Bytes
To execute an arithmetic right shift on four independent bytes by a fixed number of bits.

Description:  
rt31..24 >> sa || rt23..16 >> sa || rt15..8 >> sa || rt7..0 >> sa
The four byte elements in register rt are each shifted right arithmetically by sa bits, then written to the corresponding vector elements in destination register rd. The sa argument is interpreted as an unsigned three-bit integer taking values from zero to seven.

In the rounding variant of the instruction, a value of 1 is added at the most significant discarded bit position of each result prior to writing the rounded result to the destination register.

Restrictions:
No data-dependent exceptions are possible.
The operands must be a value in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

SHRA.QB
tempD7..0 ← ( GPR[rt]31)sa || GPR[rt]31..24+sa )
tempC7..0 ← ( GPR[rt]23)sa || GPR[rt]23..16+sa )
tempB7..0 ← ( GPR[rt]15)sa || GPR[rt]15..8+sa )
tempA7..0 ← ( GPR[rt]7)sa || GPR[rt]7..sa )
GPR[rd]..0 ← tempD7..0 || tempC7..0 || tempB7..0 || tempA7..0

SHRA_R.QB
if ( sa2..0 = 0 ) then
  tempD7..0 ← GPR[rt]31..24
  tempC7..0 ← GPR[rt]23..16
  tempB7..0 ← GPR[rt]15..8
  tempA7..0 ← GPR[rt]7..0
else
  tempD8..0 ← ( GPR[rt]31)sa || GPR[rt]31..24+sa-1 ) + 1
  tempC8..0 ← ( GPR[rt]23)sa || GPR[rt]23..16+sa-1 ) + 1
  tempB8..0 ← ( GPR[rt]15)sa || GPR[rt]15..8+sa-1 ) + 1
  tempA8..0 ← ( GPR[rt]7)sa || GPR[rt]7..sa-1 ) + 1
endif
GPR[rd]..0 ← tempD8..1 || tempC8..1 || tempB8..1 || tempA8..1
endif
Exceptions:
Reserved Instruction, DSP Disabled
Format:  
SHRA[_R].PH
SHRA.PH  rd, rt, sa
SHRA_R.PH rd, rt, sa

Purpose:  Shift Right Arithmetic Vector Pair Halfwords
Element-wise arithmetic right-shift of two independent halfwords in a vector data type by a fixed number of bits, with optional rounding.

Description:  rd \leftarrow \text{rnd16}(rt_{31..16} \gg sa) \| \text{rnd16}(rt_{15..0} \gg sa)

The two halfword values in register \( rt \) are each independently shifted right by \( sa \) bits, with each value’s original sign bit duplicated into the \( sa \) most-significant bits emptied by the shift.

In the non-rounding variant of this instruction, the two independent results are then written to the corresponding half-word elements of destination register \( rd \).

In the rounding variant of the instruction, a 1 is added at the most-significant discarded bit position before the results are written to destination register \( rd \).

Restrictions:
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

SHRA.PH
\[
\begin{align*}
tempB_{15..0} & \leftarrow \text{shift16RightArithmetic}( \text{GPR}[rt]_{31..16}, sa ) \\
tempA_{15..0} & \leftarrow \text{shift16RightArithmetic}( \text{GPR}[rt]_{15..0}, sa ) \\
\text{GPR}[rd]_{..0} & \leftarrow tempB_{15..0} \| tempA_{15..0}
\end{align*}
\]

SHRA_R.PH
\[
\begin{align*}
tempB_{15..0} & \leftarrow \text{rnd16ShiftRightArithmetic}( \text{GPR}[rt]_{31..16}, sa ) \\
tempA_{15..0} & \leftarrow \text{rnd16ShiftRightArithmetic}( \text{GPR}[rt]_{15..0}, sa ) \\
\text{GPR}[rd]_{..0} & \leftarrow tempB_{15..0} \| tempA_{15..0}
\end{align*}
\]

function \text{shift16RightArithmetic}( a_{15..0}, s_{3..0} )
\[
\begin{align*}
\text{if} \ ( s_{3..0} = 0 ) \ \text{then} \\
\text{temp}_{15..0} & \leftarrow a_{15..0} \\
\text{else} \\
\text{sign} & \leftarrow a_{15} \\
\text{temp}_{15..0} & \leftarrow ( \text{sign} \| a_{15..s} ) \\
\text{endif} \\
\text{return temp}_{15..0}
\end{align*}
\]
endfunction \text{shift16RightArithmetic}

function \text{rnd16ShiftRightArithmetic}( a_{15..0}, s_{3..0} )
\[
\begin{align*}
\text{if} \ ( s_{3..0} = 0 ) \ \text{then}
\end{align*}
\]
temp16..0 ← ( a15..0 || 0 )
else
    sign ← a15
    temp16..0 ← ( sign^s || a15..s-1 )
endif
    temp16..0 ← temp + 1
return temp16..1
endfunction rnd16ShiftRightArithmetic

Exceptions:
Reserved Instruction, DSP Disabled
**SHRAV[ _R].PH**  
Shift Right Arithmetic Variable Vector Pair Halfwords

<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>SPECIAL3</td>
<td>(rs)</td>
<td>(rt)</td>
<td>(rd)</td>
<td>SHRAV.PH</td>
<td>(01011)</td>
<td>SHLL.QB</td>
<td>(010011)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPECIAL3</td>
<td>(rs)</td>
<td>(rt)</td>
<td>(rd)</td>
<td>SHRAV_R.PH</td>
<td>(01111)</td>
<td>SHLL.QB</td>
<td>(010011)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  
SHRAV[ _R].PH  
SHRAV.PH \(rd, rt, rs\)  
SHRAV_R.PH \(rd, rt, rs\)  
MIPSDSP

**Purpose:** Shift Right Arithmetic Variable Vector Pair Halfwords  
Element-wise arithmetic right shift of two independent halfwords in a vector data type by a variable number of bits, with optional rounding.

**Description:**  
\(rd \leftarrow \text{rnd}16(rt_{31..16} \gg rs_{3..0}) \mid \mid \text{rnd}16(rt_{15..0} \gg rs_{3..0})\)

The two halfword values in register \(rt\) are each independently shifted right, with each value’s original sign bit duplicated into the most-significant bits emptied by the shift. In the non-rounding variant of this instruction, the two independent results are then written to the corresponding halfword elements of destination register \(rd\).

In the rounding variant of this instruction, a 1 is added at the most-significant discarded bit position before the results are written to destination register \(rd\).

The shift amount \(sa\) is given by the four least-significant bits of register \(rs\); the remaining bits of \(rs\) are ignored.

**Restrictions:**  
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

**SHRAV.PH**  
\[
\begin{align*}
tempB_{15..0} & \leftarrow \text{shift}16\text{RightArithmetic}(GPR[rt]_{31..16}, GPR[rs]_{3..0}) \\
tempA_{15..0} & \leftarrow \text{shift}16\text{RightArithmetic}(GPR[rt]_{15..0}, GPR[rs]_{3..0}) \\
GPR[rd]_{..0} & \leftarrow tempB_{15..0} \mid \mid tempA_{15..0}
\end{align*}
\]

**SHRAV_R.PH**  
\[
\begin{align*}
tempB_{15..0} & \leftarrow \text{rnd}16\text{ShiftRightArithmetic}(GPR[rt]_{31..16}, GPR[rs]_{3..0}) \\
tempA_{15..0} & \leftarrow \text{rnd}16\text{ShiftRightArithmetic}(GPR[rt]_{15..0}, GPR[rs]_{3..0}) \\
GPR[rd]_{..0} & \leftarrow tempB_{15..0} \mid \mid tempA_{15..0}
\end{align*}
\]

**Exceptions:**  
Reserved Instruction, DSP Disabled
**Purpose:** Shift Right Arithmetic Variable Vector of Four Bytes

To execute an arithmetic right shift on four independent bytes by a variable number of bits.

**Description:**

r

\[
\text{rd} \leftarrow \text{round}(rt_{31..24} \gg rs_{2..0}) || \text{round}(rt_{23..16} \gg rs_{2..0}) || \text{round}(rt_{15..8} \gg rs_{2..0}) || \text{round}(rt_{7..0} \gg rs_{2..0})
\]

The four byte elements in register \( rt \) are each shifted right arithmetically by \( sa \) bits, then written to the corresponding byte elements in destination register \( rd \). The \( sa \) argument is provided by the three least-significant bits of register \( rs \), interpreted as an unsigned three-bit integer taking values from zero to seven. The remaining bits of \( rs \) are ignored.

In the rounding variant of the instruction, a value of 1 is added at the most significant discarded bit position of each result prior to writing the rounded result to the destination register.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

### SHRAV.QB

\[
\begin{align*}
\text{sa}_{2..0} & \leftarrow \text{GPR}[rs]_{2..0} \\
& \text{if } ( \text{sa}_{2..0} = 0 ) \text{ then} \\
& \quad \text{tempD}_{7..0} \leftarrow \text{GPR}[rt]_{31..24} \\
& \quad \text{tempC}_{7..0} \leftarrow \text{GPR}[rt]_{23..16} \\
& \quad \text{tempB}_{7..0} \leftarrow \text{GPR}[rt]_{15..8} \\
& \quad \text{tempA}_{7..0} \leftarrow \text{GPR}[rt]_{7..0} \\
& \text{else} \\
& \quad \text{tempD}_{7..0} \leftarrow ( \text{GPR}[rt]_{31} )_{sa} || \text{GPR}[rt]_{31..24+sa} \\
& \quad \text{tempC}_{7..0} \leftarrow ( \text{GPR}[rt]_{23} )_{sa} || \text{GPR}[rt]_{23..16+sa} \\
& \quad \text{tempB}_{7..0} \leftarrow ( \text{GPR}[rt]_{15} )_{sa} || \text{GPR}[rt]_{15..8+sa} \\
& \quad \text{tempA}_{7..0} \leftarrow ( \text{GPR}[rt]_{7} )_{sa} || \text{GPR}[rt]_{7..sa} \\
& \text{endif} \\
\text{GPR}[rd]_{7..0} \leftarrow \text{tempD}_{7..0} || \text{tempC}_{7..0} || \text{tempB}_{7..0} || \text{tempA}_{7..0}
\end{align*}
\]

### SHRAV_R.QB

\[
\begin{align*}
\text{sa}_{2..0} & \leftarrow \text{GPR}[rs]_{2..0} \\
& \text{if } ( \text{sa}_{2..0} = 0 ) \text{ then} \\
& \quad \text{tempD}_{8..0} \leftarrow ( \text{GPR}[rt]_{31..24} || 0 ) \\
& \quad \text{tempC}_{8..0} \leftarrow ( \text{GPR}[rt]_{23..16} || 0 ) \\
& \quad \text{tempB}_{8..0} \leftarrow ( \text{GPR}[rt]_{15..8} || 0 ) \\
& \quad \text{tempA}_{8..0} \leftarrow ( \text{GPR}[rt]_{7..0} || 0 ) \\
& \text{else}
\end{align*}
\]
tempD[8..0] ← (GPR[rt]31)sa || (GPR[rt]31..24+sa-1) + 1
tempC[8..0] ← (GPR[rt]23)sa || (GPR[rt]23..16+sa-1) + 1
tempB[8..0] ← (GPR[rt]15)sa || (GPR[rt]15..8+sa-1) + 1
tempA[8..0] ← (GPR[rt]7)sa || (GPR[rt]7..sa-1) + 1
endif
GPR[rd]..0 ← tempD..1 || tempC..1 || tempB..1 || tempA..1

Exceptions:
Reserved Instruction, DSP Disabled
Format: SHRAV_R.W rd, rt, rs

Purpose: Shift Right Arithmetic Variable Word with Rounding

Arithmetic right shift with rounding of a signed 32-bit word by a variable number of bits.

Description: \( rd \leftarrow \text{rnd32}(rt_{31..0} >> rs_{4..0}) \)

The word value in register \( rt \) is shifted right, with the value’s original sign bit duplicated into the most-significant bits emptied by the shift. A 1 is then added at the most-significant discarded bit position before the result is written to destination register \( rd \).

The shift amount \( sa \) is given by the five least-significant bits of register \( rs \); the remaining bits of \( rs \) are ignored.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

\[
\text{temp}_{31..0} \leftarrow \text{rnd32ShiftRightArithmetic}( \text{GPR}[rt]_{31..0}, \text{GPR}[rs]_{4..0} ) \\
\text{GPR}[rd]_{..0} \leftarrow \text{temp}_{31..0}
\]

Exceptions:

Reserved Instruction, DSP Disabled
Format: SHRA_R.W rd, rt, sa

Purpose: Shift Right Arithmetic Word with Rounding

To execute an arithmetic right shift with rounding on a word by a fixed number of bits.

Description: rd \leftarrow \text{rnd32}(rt_{31:0} \gg sa)

The word in register rt is shifted right by sa bits, and the sign bit is duplicated into the sa bits emptied by the shift. The shifted result is then rounded by adding a 1 bit to the most-significant discarded bit. The rounded result is then written to the destination register rd.

Restrictions: No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

\begin{align*}
\text{temp}_{31..0} & \leftarrow \text{rnd32ShiftRightArithmetic}( \text{GPR}[rt]_{31..0}, sa_{4..0} ) \\
\text{GPR}[rd]_{32..0} & \leftarrow \text{temp}_{32..1}
\end{align*}

function rnd32ShiftRightArithmetic( a_{31..0}, s_{4..0} )

if ( s_{4..0} = 0 ) then
\begin{align*}
\text{temp}_{32..0} & \leftarrow ( a_{31..0} \mid \mid 0 )
\end{align*}
else
\begin{align*}
\text{sign} & \leftarrow a_{31} \\
\text{temp}_{32..0} & \leftarrow ( \text{sign}^{s} \mid \mid a_{31..s-1} )
\end{align*}
endif
\begin{align*}
\text{temp}_{32..0} & \leftarrow \text{temp} + 1 \\
\text{return temp}_{32..1}
\end{align*}
endfunction rnd32ShiftRightArithmetic

Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

To do an arithmetic right shift of a word in a register without rounding, use the SRA instruction.
**SHRL.PH**

**Shift Right Logical Two Halfwords**

**Format:** SHRL.PH rd, rt, sa  
MIPS® DSP Module for MIPS32™ Architecture, Revision 3.01

**Purpose:** Shift Right Logical Two Halfwords  
To execute a right shift of two independent halfwords in a vector data type by a fixed number of bits.

**Description:**

\[
\text{rd} \leftarrow (\text{rt}_{31\ldots16} >> \text{sa}) \ || \ (\text{rt}_{15\ldots0} >> \text{sa})
\]

The two halfwords in register rt are independently logically shifted right, inserting zeros into the bit positions emptied by the shift. The two halfword results are then written to the corresponding halfword elements in destination register rd.

The shift amount is provided by the sa field, which is interpreted as a four bit unsigned integer taking values between 0 and 15.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

\[
\begin{align*}
\text{tempB}_{15\ldots0} & \leftarrow 0^{\text{sa}} \ || \ \text{GPR[rt]}_{31\ldots\text{sa}+16} \\
\text{tempA}_{15\ldots0} & \leftarrow 0^{\text{sa}} \ || \ \text{GPR[rt]}_{15\ldots\text{sa}} \\
\text{GPR[rd]}_{15\ldots0} & \leftarrow \text{tempB}_{15\ldots0} \ || \ \text{tempA}_{15\ldots0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction, DSP Disabled
**Format:** SHRL.QB rd, rt, sa

**Purpose:** Shift Right Logical Vector Quad Bytes

Element-wise logical right shift of four independent bytes in a vector data type by a fixed number of bits.

**Description:**

$$rd \leftarrow rt_{31..24} >> sa \mid \mid rt_{23..16} >> sa \mid \mid rt_{15..8} >> sa \mid \mid rt_{7..0} >> sa$$

The four byte values in register \(rt\) are each independently shifted right by \(sa\) bits and the \(sa\) most-significant bits of each value are set to zero. The four independent results are then written to the corresponding byte elements of destination register \(rd\).

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

```plaintext
function shift8Right( a7..0, s2..0 )
    if ( s2..0 = 0 ) then
        temp7..0 \leftarrow a7..0
    else
        temp7..0 \leftarrow ( 0s || a7..s )
    endif
    return temp7..0
endfunction shift8Right
```

**Exceptions:**

Reserved Instruction, DSP Disabled

**Programming Notes:**

To do a logical left shift of a word in a register without saturation, use the SLL instruction.
**SHRLV.PH**  
Shift Variable Right Logical Pair of Halfwords

**Format:**  
SHRLV.PH rd, rt, rs

**Purpose:**  
Shift Variable Right Logical Pair of Halfwords  
To execute a right shift of two independent halfwords in a vector data type by a variable number of bits.

**Description:**  
\[ rd \leftarrow (rt_{31..16} >> rs_{3..0}) \parallel (rt_{15..0} >> rs_{3..0}) \]

The two halfwords in register \( rt \) are independently logically shifted right, inserting zeros into the bit positions emptied by the shift. The two halfword results are then written to the corresponding halfword elements in destination register \( rd \).

The shift amount is provided by the four least-significant bits of register \( rs \), which is interpreted as a four bit unsigned integer taking values between 0 and 15. The remaining bits of \( rs \) are ignored.

**Restrictions:**
No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**
\[
\begin{align*}
  sa_{3..0} & \leftarrow \text{GPR}[rs]_{3..0} \\
  \text{tempB}_{15..0} & \leftarrow 0^{sa} \parallel \text{GPR}[rt]_{31..sa+16} \\
  \text{tempA}_{15..0} & \leftarrow 0^{sa} \parallel \text{GPR}[rt]_{15..sa} \\
  \text{GPR}[rd]_{..0} & \leftarrow \text{tempB}_{15..0} \parallel \text{tempA}_{15..0}
\end{align*}
\]

**Exceptions:**
Reserved Instruction, DSP Disabled
**SHRLV.QB**

**Shift Right Logical Variable Vector Quad Bytes**

<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>SPECIAL3</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>SHRLV.QB</td>
<td>SHLL.QB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>00011</td>
<td>010011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  
`SHRLV.QB rd, rt, rs`  

**Purpose:** Shift Right Logical Variable Vector Quad Bytes  
Element-wise logical right shift of four independent bytes in a vector data type by a variable number of bits.

**Description:**  
`rd ← (rt31..24 >> rs2..0) || (rt23..16 >> rs2..0) || (rt15..8 >> rs2..0) || (rt7..0 >> rs2..0)`  
The four byte values in register `rt` are each independently shifted right, inserting zeros into the most-significant bit positions emptied by the shift. The four independent results are then written to the corresponding byte elements of destination register `rd`.

The three least-significant bits of `rs` provide the shift value, interpreted as an unsigned integer; the remaining bits of `rs` are ignored.

**Restrictions:**  
No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

`tempD7..0 ← shift8Right( GPR[rt]31..24, GPR[rs]2..0 )`
`tempC7..0 ← shift8Right( GPR[rt]23..16, GPR[rs]2..0 )`
`tempB7..0 ← shift8Right( GPR[rt]15..8, GPR[rs]2..0 )`
`tempA7..0 ← shift8Right( GPR[rt]7..0, GPR[rs]2..0 )`
`GPR[rd]7..0 ← tempD7..0 || tempC7..0 || tempB7..0 || tempA7..0`

**Exceptions:**  
Reserved Instruction, DSP Disabled
### SUBQ[._S].PH

**Subtract Fractional Halfword Vector**

Element-wise subtraction of one vector of Q15 fractional halfword values from another to produce a vector of Q15 fractional halfword results, with optional saturation.

**Description:**

\[
rd \leftarrow \text{sat16}(rs_{31..16} - rt_{31..16}) \mid \text{sat16}(rs_{15..0} - rt_{15..0})
\]

The two fractional halfwords in register \(rt\) are subtracted from the corresponding fractional halfword elements in register \(rs\).

For the non-saturating version of this instruction, each result is written to the corresponding element in register \(rd\). In the case of overflow or underflow, the result modulo 2 is written to register \(rd\).

For the saturating version of the instruction, the subtraction is performed using signed saturating arithmetic. If the operation results in an overflow or an underflow, the result is clamped to either the largest representable value (0x7FFF hexadecimal) or the smallest representable value (0x8000 hexadecimal), respectively, before being written to the destination register \(rd\).

For both instructions, if any of the individual subtractions result in underflow, overflow, or saturation, a 1 is written to bit 20 in the \(DSPControl\) register within the \(ouflag\) field.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

**SUBQ.PH:**

\[
\begin{align*}
\text{tempB}_{15..0} & \leftarrow \text{subtract16}(GPR[rs]_{31..16}, GPR[rt]_{31..16}) \\
\text{tempA}_{15..0} & \leftarrow \text{subtract16}(GPR[rs]_{15..0}, GPR[rt]_{15..0}) \\
GPR[rd]_{15..0} & \leftarrow \text{tempB}_{15..0} \mid \text{tempA}_{15..0}
\end{align*}
\]

**SUBQ_S.PH:**

\[
\begin{align*}
\text{tempB}_{15..0} & \leftarrow \text{sat16Subtract}(GPR[rs]_{31..16}, GPR[rt]_{31..16}) \\
\text{tempA}_{15..0} & \leftarrow \text{sat16Subtract}(GPR[rs]_{15..0}, GPR[rt]_{15..0}) \\
GPR[rd]_{15..0} & \leftarrow \text{tempB}_{15..0} \mid \text{tempA}_{15..0}
\end{align*}
\]

function subtract16( \(a_{15..0}\), \(b_{15..0}\) )

\[
\text{temp}_{16..0} \leftarrow (a_{15} \mid a_{15..0}) - (b_{15} \mid b_{15..0})
\]

if (temp_{16} \neq \text{temp}_{15}) then

\[
\text{DSPControl}_{ouflag:20} \leftarrow 1
\]
endif

return temp_{15..0}

endfunction subtract16
function sat16Subtract( a15..0, b15..0 )
    temp16..0 ← ( a15 || a15..0 ) - ( b15 || b15..0 )
    if ( temp16 ≠ temp15 ) then
        if ( temp16 = 0 ) then
            temp ← 0x7FFF
        else
            temp ← 0x8000
        endif
        DSPControlcuflag:20 ← 1
    endif
    return temp15..0
endfunction sat16Subtract

Exceptions:
Reserved Instruction, DSP Disabled
**Format:** SUBQ_S.W rd, rs, rt

**Purpose:** Subtract Fractional Word

One Q31 fractional word is subtracted from another to produce a Q31 fractional result, with saturation.

**Description:**

\[
rd \leftarrow \text{sat32}(rs_{31..0} - rt_{31..0})
\]

The Q31 fractional word in register \(rt\) is subtracted from the corresponding fractional word in register \(rs\), and the 32-bit result is written to destination register \(rd\). The subtraction is performed using signed saturating arithmetic. If the operation results in an overflow or an underflow, the result is clamped to either the largest representable value (0x7FFFFFFF hexadecimal) or the smallest representable value (0x80000000 hexadecimal), respectively, before being sign-extended and written to the destination register \(rd\).

If the subtraction results in underflow, overflow, or saturation, a 1 is written to bit 20 in the **DSPControl** register within the **ouflag** field.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

\[
\begin{align*}
\text{temp}_{31..0} & \leftarrow \text{sat32Subtract}( \ GPR[rs]_{31..0} \ , \ GPR[rt]_{31..0} \ ) \\
GPR[rd]_{31..0} & \leftarrow \text{temp}_{31..0}
\end{align*}
\]

```
function sat32Subtract( a_{31..0}, b_{31..0} )
    temp_{32..0} ← ( a_{31} || a_{31..0} ) - ( b_{31} || b_{31..0} )
    if ( temp_{32} ≠ temp_{31} ) then
        if ( temp_{32} = 0 ) then
            temp_{31..0} ← 0x7FFFFFFFF
        else
            temp_{31..0} ← 0x80000000
        endif
        DSPControl.ouflag:20 ← 1
    endif
    return temp_{31..0}
endfunction sat32Subtract
```

**Exceptions:**

Reserved Instruction, DSP Disabled
SUBQH[ R].PH

Add Fractional Halfword Vectors And Shift Right to Halve Results

Format:

SUBQH[ R].PH
SUBQH.PH rd, rs, rt
SUBQH_R.PH rd, rs, rt

Purpose:

Subtract Fractional Halfword Vectors And Shift Right to Halve Results

Element-wise fractional subtraction of halfword vectors, with a right shift by one bit to halve each result, with optional rounding.

Description:

\[ rd \leftarrow \text{round}((rs_{31..16} - rt_{31..16}) \gg 1) \quad \| \quad \text{round}((rs_{15..0} - rt_{15..0}) \gg 1) \]

Each element from the two halfword values in register rt is subtracted from the corresponding halfword element in register rs to create an interim 17-bit result.

In the non-rounding instruction variant, each interim result is then shifted right by one bit before being written to the corresponding halfword element of destination register rd.

In the rounding version of the instruction, a value of 1 is added at the least-significant bit position of each interim result; the interim result is then right-shifted by one bit and written to the destination register.

This instruction does not modify the DSPControl register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

ADDQH.PH

\[
\begin{align*}
\text{temp}_{B15..0} & \leftarrow \text{rightShift1SubQ16}( \text{GPR}[rs]_{31..16}, \text{GPR}[rt]_{31..16}) \\
\text{temp}_{A15..0} & \leftarrow \text{rightShift1SubQ16}( \text{GPR}[rs]_{15..0}, \text{GPR}[rt]_{15..0}) \\
\text{GPR}[rd]_{..0} & \leftarrow \text{temp}_{B15..0} \| \text{temp}_{A15..0}
\end{align*}
\]

ADDQH_R.PH

\[
\begin{align*}
\text{temp}_{B15..0} & \leftarrow \text{roundRightShift1SubQ16}( \text{GPR}[rs]_{31..16}, \text{GPR}[rt]_{31..16}) \\
\text{temp}_{A15..0} & \leftarrow \text{roundRightShift1SubQ16}( \text{GPR}[rs]_{15..0}, \text{GPR}[rt]_{15..0}) \\
\text{GPR}[rd]_{..0} & \leftarrow \text{temp}_{B15..0} \| \text{temp}_{A15..0}
\end{align*}
\]

function rightShift1SubQ16( a15..0 , b15..0 )

\[
\begin{align*}
\text{temp}_{16..0} & \leftarrow ((a_{15} \| a_{15..0} ) - (b_{15} \| b_{15..0} ))
\text{return temp}_{16..1}
\end{align*}
\]

endfunction rightShift1SubQ16

function roundRightShift1SubQ16( a15..0 , b15..0 )

\[
\begin{align*}
\text{temp}_{16..0} & \leftarrow ((a_{15} \| a_{15..0} ) - (b_{15} \| b_{15..0} ))
\text{temp}_{16..0} & \leftarrow \text{temp}_{16..0} + 1
\text{return temp}_{16..1}
\end{align*}
\]

endfunction roundRightShift1SubQ16
<table>
<thead>
<tr>
<th>SUBQH[ R].PH</th>
<th>Subtract Fractional Halfword Vectors And Shift Right to Halve Results</th>
</tr>
</thead>
</table>

**Exceptions:**
- Reserved Instruction, DSP Disabled
**SUBQH[ R].W**  
**Subtract Fractional Words And Shift Right to Halve Results**

**Format:**

```
SUBQH[ R].W
SUBQH.W  rd, rs, rt
SUBQH_R.W rd, rs, rt
```

**Purpose:** Subtract Fractional Words And Shift Right to Halve Results

Fractional subtraction of word vectors, with a right shift by one bit to halve the result, with optional rounding.

**Description:**

```
rd  \leftarrow \text{round}((rs_{31..0} - rt_{31..0}) \gg 1)
```

The word in register `rt` is subtracted from the word in register `rs` to create an interim 33-bit result.

In the non-rounding instruction variant, the interim result is then shifted right by one bit before being written to the destination register `rd`.

In the rounding version of the instruction, a value of 1 is added at the least-significant bit position of the interim result; the interim result is then right-shifted by one bit and written to the destination register.

This instruction does not modify the `DSPControl` register.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

**Operation:**

```
ADDQH.W
\text{temp}_{31..0} \leftarrow \text{rightShift1SubQ32}( \text{GPR}[rs]_{31..0}, \text{GPR}[rt]_{31..0} )
\text{GPR}[rd]_{..0} \leftarrow \text{temp}_{31..0}

ADDQH_R.W
\text{temp}_{31..0} \leftarrow \text{roundRightShift1SubQ32}( \text{GPR}[rs]_{31..0}, \text{GPR}[rt]_{31..0} )
\text{GPR}[rd]_{..0} \leftarrow \text{temp}_{31..0}
```

**Exceptions:**

Reserved Instruction, DSP Disabled
SUBU[_S].PH Subtract Unsigned Integer Halfwords

**Format:**

SUBU.PH  
rd, rs, rt  

SUBU_S.PH  
rd, rs, rt

**Purpose:** Subtract Unsigned Integer Halfwords

Element-wise subtraction of pairs of unsigned integer halfwords, with optional saturation.

**Description:**

\[ \text{rd} \leftarrow \text{sat16}(\text{rs}_{31..16} - \text{rt}_{31..16}) \ || \ \text{sat16}(\text{rs}_{15..0} - \text{rt}_{15..0}) \]

The two unsigned integer halfwords in register \( rs \) are subtracted from the corresponding unsigned integer halfwords in register \( rt \). The unsaturated results are then written to the corresponding element in destination register \( rd \).

In the saturating version of the instruction, if either subtraction results in an underflow the result is clamped to the minimum unsigned integer halfword value (0x0000 hexadecimal), before being written to the destination register \( rd \).

For both instruction variants, if either subtraction causes an underflow the instruction writes a 1 to bit 20 in the DSPControl register in the ouflag field.

**Restrictions:**

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

**Operation:**

SUBU.PH

\[
\begin{align*}
\text{temp}
\text{B}_{15..0} & \leftarrow \text{subtractU16U16}( \text{GPR}[\text{rt}]_{31..16} , \text{GPR}[\text{rs}]_{31..16} ) \\
\text{temp}
\text{A}_{15..0} & \leftarrow \text{subtractU16U16}( \text{GPR}[\text{rt}]_{15..0} , \text{GPR}[\text{rs}]_{15..0} ) \\
\text{GPR}[\text{rd}]_{..0} & \leftarrow \text{temp}
\text{B}_{15..0} \ || \ \text{temp}
\text{A}_{15..0}
\end{align*}
\]

SUBU_S.PH

\[
\begin{align*}
\text{temp}
\text{B}_{15..0} & \leftarrow \text{satU16SubtractU16U16}( \text{GPR}[\text{rt}]_{31..16} , \text{GPR}[\text{rs}]_{31..16} ) \\
\text{temp}
\text{A}_{15..0} & \leftarrow \text{satU16SubtractU16U16}( \text{GPR}[\text{rt}]_{15..0} , \text{GPR}[\text{rs}]_{15..0} ) \\
\text{GPR}[\text{rd}]_{..0} & \leftarrow \text{temp}
\text{B}_{15..0} \ || \ \text{temp}
\text{A}_{15..0}
\end{align*}
\]

Function `subtractU16U16`

\[
\begin{align*}
\text{temp}
\text{p}_{16..0} & \leftarrow ( 0 \ || \ \text{a}_{15..0} ) - ( 0 \ || \ \text{b}_{15..0} ) \\
\text{if} ( \ \text{temp}
\text{p}_{16} = 1 ) \ \text{then} \\
\text{DSPControl}[\text{ouflag} : 20] & \leftarrow 1 \\
\text{endif} \\
\text{return} \ \text{temp}
\text{p}_{15..0}
\end{align*}
\]

Function `satU16SubtractU16U16`

\[
\begin{align*}
\text{temp}
\text{p}_{16..0} & \leftarrow ( 0 \ || \ \text{a}_{15..0} ) - ( 0 \ || \ \text{b}_{15..0} ) \\
\text{if} ( \ \text{temp}
\text{p}_{16} = 1 ) \ \text{then} \\
\text{temp}
\text{p}_{15..0} & \leftarrow 0x0000 \\
\text{DSPControl}[\text{ouflag} : 20] & \leftarrow 1
\end{align*}
\]

---

222  
MIPS® DSP Module for MIPS32™ Architecture, Revision 3.01
endif
return temp15..0
endfunction satU16SubtractU16U16

Exceptions:
Reserved Instruction, DSP Disabled
SUBU_[S].QB Subtract Unsigned Quad Byte Vector

Format:

SUBU.QB rd, rs, rt  
SUBU_S.QB rd, rs, rt

Purpose: Subtract Unsigned Quad Byte Vector

Element-wise subtraction of one vector of unsigned byte values from another to produce a vector of unsigned byte results, with optional saturation.

Description:

\[ rd \leftarrow \text{sat8}(rs_{31..24} - rt_{31..24}) \parallel \text{sat8}(rs_{23..16} - rt_{23..16}) \parallel \text{sat8}(rs_{15..8} - rt_{15..8}) \parallel \text{sat8}(rs_{7..0} - rt_{7..0}) \]

The four byte elements in \( rt \) are subtracted from the corresponding byte elements in register \( rs \).

For the non-saturating version of the instruction, the result modulo 256 is written into the corresponding position in register \( rd \).

For the saturating version of the instruction the subtraction is performed using unsigned saturating arithmetic. If the subtraction results in underflow, the value is clamped to the smallest representable value (0 decimal, 0x00 hexadecimal) before being written to the destination register \( rd \).

For each instruction, if any of the individual subtractions result in underflow or saturation, a 1 is written to bit 20 in the DSPControl register within the ouflag field.

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

SUBU.QB:

\[
\begin{align*}
tempD_{7..0} & \leftarrow \text{subtractU8}( GPR[rs]_{31..24}, GPR[rt]_{31..24} ) \\
tempC_{7..0} & \leftarrow \text{subtractU8}( GPR[rs]_{23..16}, GPR[rt]_{23..16} ) \\
tempB_{7..0} & \leftarrow \text{subtractU8}( GPR[rs]_{15..8}, GPR[rt]_{15..8} ) \\
tempA_{7..0} & \leftarrow \text{subtractU8}( GPR[rs]_{7..0}, GPR[rt]_{7..0} ) \\
GPR[rd]_{7..0} & \leftarrow tempD_{7..0} \parallel tempC_{7..0} \parallel tempB_{7..0} \parallel tempA_{7..0}
\end{align*}
\]

SUBU_S.QB:

\[
\begin{align*}
tempD_{7..0} & \leftarrow \text{satU8Subtract}( GPR[rs]_{31..24}, GPR[rt]_{31..24} ) \\
tempC_{7..0} & \leftarrow \text{satU8Subtract}( GPR[rs]_{23..16}, GPR[rt]_{23..16} ) \\
tempB_{7..0} & \leftarrow \text{satU8Subtract}( GPR[rs]_{15..8}, GPR[rt]_{15..8} ) \\
tempA_{7..0} & \leftarrow \text{satU8Subtract}( GPR[rs]_{7..0}, GPR[rt]_{7..0} ) \\
GPR[rd]_{7..0} & \leftarrow tempD_{7..0} \parallel tempC_{7..0} \parallel tempB_{7..0} \parallel tempA_{7..0}
\end{align*}
\]

function subtractU8( a_{7..0}, b_{7..0} )

\[
\text{temp}_{8..0} \leftarrow ( 0 \parallel a_{7..0} ) - ( 0 \parallel b_{7..0} )
\]

if ( temp_{8} = 1 ) then

\[
\text{DSPControl}_{ouflag:20} \leftarrow 1
\]
endif
return temp7..0
endfunction subtractU8

function satU8Subtract( a7..0, b7..0 )
  temp8..0 ← ( 0 || a7..0 ) - ( 0 || b7..0 )
  if ( temp8 = 1 ) then
    temp7..0 ← 0x00
    DSPControlouflag:20 ← 1
  endif
  return temp7..0
endfunction satU8Subtract

Exceptions:
 Reserved Instruction, DSP Disabled
**SUBUH[ R].QB**  
**Subtract Unsigned Bytes And Right Shift to Halve Results**

### Format:

<table>
<thead>
<tr>
<th>SPECIAL3</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>SUBUH</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111</td>
<td></td>
<td></td>
<td></td>
<td>00001</td>
</tr>
</tbody>
</table>

**ADDUH.QB**  
**MIPS DSP - R2**

<table>
<thead>
<tr>
<th>SPECIAL3</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>ADDUH.QB</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111</td>
<td></td>
<td></td>
<td></td>
<td>011000</td>
</tr>
</tbody>
</table>

### Purpose:

Subtract Unsigned Bytes And Right Shift to Halve Results

Element-wise subtraction of two vectors of unsigned bytes, with a one-bit right shift to halve results and optional rounding.

### Description:

rd = \text{round}((rs_{31..24} - rt_{31..24}) >> 1) || \text{round}((rs_{23..16} - rt_{23..16}) >> 1) || \text{round}((rs_{15..8} - rt_{15..8}) >> 1) || \text{round}((rs_{7..0} - rt_{7..0}) >> 1)

The four unsigned byte values in register \( rt \) are subtracted from the corresponding unsigned byte values in register \( rs \). Each unsigned result is then halved by shifting right by one bit position. The byte results are then written to the corresponding elements of destination register \( rd \).

In the rounding variant of the instruction, a value of 1 is added to the result of each subtraction at the discarded bit position before the right shift.

The results of this instruction never overflow; no bits of the \( ouflag \) field in the \( DSPControl \) register are written.

### Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

### Operation:

**SUBUH.QB**

\[
\begin{align*}
\text{tempD}_{7..0} & \leftarrow (0 || \text{GPR}[rs]_{31..24}) - (0 || \text{GPR}[rt]_{31..24}) >> 1 \\
\text{tempC}_{7..0} & \leftarrow (0 || \text{GPR}[rs]_{23..16}) - (0 || \text{GPR}[rt]_{23..16}) >> 1 \\
\text{tempB}_{7..0} & \leftarrow (0 || \text{GPR}[rs]_{15..8}) - (0 || \text{GPR}[rt]_{15..8}) >> 1 \\
\text{tempA}_{7..0} & \leftarrow (0 || \text{GPR}[rs]_{7..0}) - (0 || \text{GPR}[rt]_{7..0}) >> 1 \\
\text{GPR}[rd]_{7..0} & \leftarrow \text{tempD}_{7..0} || \text{tempC}_{7..0} || \text{tempB}_{7..0} || \text{tempA}_{7..0}
\end{align*}
\]

**SUBUH.R.QB**

\[
\begin{align*}
\text{tempD}_{7..0} & \leftarrow (0 || \text{GPR}[rs]_{31..24}) - (0 || \text{GPR}[rt]_{31..24}) + 1) >> 1 \\
\text{tempC}_{7..0} & \leftarrow (0 || \text{GPR}[rs]_{23..16}) - (0 || \text{GPR}[rt]_{23..16}) + 1) >> 1 \\
\text{tempB}_{7..0} & \leftarrow (0 || \text{GPR}[rs]_{15..8}) - (0 || \text{GPR}[rt]_{15..8}) + 1) >> 1 \\
\text{tempA}_{7..0} & \leftarrow (0 || \text{GPR}[rs]_{7..0}) - (0 || \text{GPR}[rt]_{7..0}) + 1) >> 1 \\
\text{GPR}[rd]_{7..0} & \leftarrow \text{tempD}_{7..0} || \text{tempC}_{7..0} || \text{tempB}_{7..0} || \text{tempA}_{7..0}
\end{align*}
\]

### Exceptions:

Reserved Instruction, DSP Disabled
WRDSP  Write Fields to DSPControl Register from a GPR

MIPS® DSP Module for MIPS32™ Architecture, Revision 3.01

Format:

WRDSP
WRDSP rs, mask
WRDSP rs

MIPSDSP
Assembly Idiom

Purpose: Write Fields to DSPControl Register from a GPR

To copy selected fields from the specified GPR to the special-purpose DSPControl register.

Description: DSPControl \(\leftarrow\) select(mask, GPR[rs])

Selected fields in the special register DSPControl are overwritten with the corresponding bits from the source GPR rs. Each of bits 0 through 5 of the mask operand corresponds to a specific field in the DSPControl register. A mask bit value of 1 indicates that the field will be overwritten using the bits from the same bit positions in register rs, and a mask bit value of 0 indicates that the corresponding field will be unchanged. Bits 6 through 9 of the mask operand are ignored.

The table below shows the correspondence between the bits in the mask operand and the fields in the DSPControl register; mask bit 0 is the least-significant bit in mask.

For example, to overwrite only the scount field in DSPControl, the value of the mask operand used will be 2 decimal (0x02 hexadecimal). After execution of the instruction, the scount field in DSPControl will have the value of bits 7 through 12 of the specified source register rs and the remaining bits in DSPControl are unmodified.

The one-operand version of the instruction provides a convenient assembly idiom that allows the programmer to write all the allowable fields in the DSPControl register from the source GPR, i.e., it is equivalent to specifying a mask value of 31 decimal (0x1F hexadecimal).

Restrictions:

No data-dependent exceptions are possible.

The operands must be values in the specified format. If they are not, the results are UNPREDICTABLE and the values of the operand vectors become UNPREDICTABLE.

Operation:

\[
\begin{align*}
\text{newbits}_{31..0} & \leftarrow 0^{32} \\
\text{overwrite}_{31..0} & \leftarrow 0xFFFFFFFF \\
\text{if } ( \text{mask}_0 = 1 ) \text{ then} \\
& \quad \text{newbits}_{0} \leftarrow 0 \\
& \quad \text{overwrite}_{0} \leftarrow \text{GPR[rs]}_{0} \\
\text{endif} \\
\text{if } ( \text{mask}_1 = 1 ) \text{ then} \\
& \quad \text{newbits}_{12..7} \leftarrow 0^6 \\
& \quad \text{overwrite}_{12..7} \leftarrow \text{GPR[rs]}_{12..7} \\
\text{endif} \\
\text{if } ( \text{mask}_2 = 1 ) \text{ then} \\
& \quad \text{newbits}_{13} \leftarrow 0 \\
& \quad \text{overwrite}_{13} \leftarrow \text{GPR[rs]}_{13} \\
\text{endif} \\
\text{if } ( \text{mask}_3 = 1 ) \text{ then} \\
& \quad \text{newbits}_{23..16} \leftarrow 0^8 \\
& \quad \text{overwrite}_{23..16} \leftarrow \text{GPR[rs]}_{23..16} \\
\end{align*}
\]
newbits\textsubscript{23..16} \leftarrow \text{GPR}[rs]\textsubscript{23..16}
\text{endif}
\text{if ( mask}_4 \text{ = 1 ) then}
\quad \text{overwrite}_{31..24} \leftarrow 0^8
\quad \text{newbits}_{31..24} \leftarrow \text{GPR}[rs]\textsubscript{31..24}
\text{endif}
\text{if ( mask}_5 \text{ = 1 ) then}
\quad \text{overwrite}_{14} \leftarrow 0
\quad \text{newbits}_{14} \leftarrow \text{GPR}[rs]\textsubscript{14}
\text{endif}
\text{DSPControl} \leftarrow \text{DSPControl and overwrite}_{31..0}
\text{DSPControl} \leftarrow \text{DSPControl or new}_{31..0}

\textbf{Exceptions:}

Reserved Instruction, DSP Disabled
A.1 Using Endian-Agnostic Instruction Names

Certain instructions being proposed in the Module only operate on a subset of the operands in the register. In most cases, this is simply the left (L) or right (R) half of the register. Some instructions refer to the left alternating (LA) or right alternating (RA) elements of the register. But this type of reference does not take the endian-ness of the processor and memory into account. Since the DSP Module instructions do not take the endian-ness into account and simply use the left or right part of the register, this section describes a method by which users can take advantage of user-defined macros to translate the given instruction to the appropriate one for a given processor endian-ness.

An example is given below that uses actual element numbers in the mnemonics to be endian-agnostic.

In the MIPS32 architecture, the following conventions could be used:

- PH0 refers to halfword element 0 (from a pair in the specified register).
- PH1 refers to halfword element 1.
- QB01 refers to byte elements 0 and 1 (from a quad in the specified register).
- QB23 refers to byte elements 2 and 3.
- QB02 refers to (even) byte elements 0 and 2.
- QB13 refers to (odd) byte elements 1 and 3.

The even and odd subsets are mainly used in storing, computing on, and loading complex numbers that have a real and imaginary part. If the real and imaginary parts of a complex number are stored in consecutive memory locations, then computations that involve only the real or only the imaginary parts must first extract these to a different register. This can most effectively be done using the even and odd formats of the relevant operations.

Note that these mnemonics are translated by the assembler to underlying real instructions that operate on absolute element positions in the register based on the endian-ness of the processor.

A.2 Mapping Endian-Agnostic Instruction Names to DSP Module Instructions

To illustrate this process, we will use one instruction as an example. This can be repeated for all the relevant instructions in the Module.

The MULEQ_S instruction multiplies fractional data operands to expanded full-size results in a destination register with optional saturation. Since the result occupies twice the width of the input operands, only half the operands from the source registers are operated on at a time. So the complete instruction mnemonic would be given as
Endianness Reference to Register Elements

MULEQ_S.W.PH0 rd, rs, rt where the second part (after the first dot) indicates the size of the result, and the third part (after the second dot) indicates the element of the source register being used, which in this example is the 0th element. The real instructions that the hardware implements are MULEQ_S.W.PHL and MULEQ_S.W.PHR which operate on the left halfword element and the right halfword element respectively, of the given source registers, as shown in Figure A.1. The user can map the user instruction (with .PH0) to the MULEQ_S.W.PHL real instruction if the processor is big-endian or to the real instruction MULEQ_S.W.PHR if the processor is little-endian.

Figure A.1 The Endian-Independent PHL and PHR Elements in a GPR for the MIPS32 Architecture

Then MULEQ_S.W.PH1 rd, rs, rt instruction can be mapped to MULEQ_S.W.PHR if the processor is big-endian (see Figure A.2), and to MULEQ_S.W.PHL real instruction if the processor is little-endian (see Figure A.3).

Figure A.2 The Big-Endian PH0 and PH1 Elements in a GPR for the MIPS32 Architecture

Figure A.3 The Little-Endian PH0 and PH1 Elements in a GPR for the MIPS32 Architecture

To specify the even and odd type operations, a user instruction (to use odd elements) such as PRECEQ_S.PH.QB02 (which precision expands the values) would be mapped to PRECEQ_S.PH.QBLA or PRECEQ_S.PH.QBRA depending on whether the endianness of the processor was big or little, respectively. (LA stands for left-alternating and RA for right-alternating).
A.2 Mapping Endian-Agnostic Instruction Names to DSP Module Instructions

Figure A.4 The Endian-Independent QBL and QBR Elements in a GPR for the MIPS32 Architecture

Figure A.5 The Endian-Independent QBLA and QBRA Elements in a GPR for the MIPS32 Architecture
Endian-Agnostic Reference to Register Elements
Revision History

Vertical change bars in the left page margin note the location of changes to this document since its last release.

NOTE: Change bars on figure titles are used to denote a potential change in the figure itself.

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>6 July, 2005</td>
<td>Initial revision</td>
</tr>
<tr>
<td>1.10</td>
<td>30 January, 2006</td>
<td>Typographical fixes.</td>
</tr>
<tr>
<td>2.00</td>
<td>12 January, 2007</td>
<td>Added the DSP Module Rev2 instructions to the specification and related material.</td>
</tr>
<tr>
<td>2.10</td>
<td>18 May, 2007</td>
<td>Allow MADD, MADDU, MSUB, MSUBU, MULT, and MULTU that access ac1-ac3 to be in the DSP Module (Revision 1) version. Fix typographical errors.</td>
</tr>
<tr>
<td>2.20</td>
<td>July 15, 2008</td>
<td>• Update copyrights.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Update contact information.</td>
</tr>
<tr>
<td>2.21</td>
<td>January 02, 2009</td>
<td>• EXTR.W, EXTR_R.W, EXTR_RS.W, EXTRV.W, EXTRV_R.W and EXTRV_RS.W all set DSPControl_outflag for overflow/saturation, even for intermediate values.</td>
</tr>
<tr>
<td>2.22</td>
<td>January 06, 2009</td>
<td>• SHRA[_R].* Operation description was incorrectly not using the rounded intermediate values.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• PRECRQ_S* instructions set bit 22 in DSPControl if clamping occurred.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DPAQX_S.W.PH, PDAQX_SA.W.PH, DPSQX_S.W.PH, DPSQX_SA.W.PH were incorrectly marked DSP Module Rev1 instructions, actually Rev2 instructions.</td>
</tr>
<tr>
<td>2.23</td>
<td>June 26, 2009</td>
<td>• MADD, MADDU, MSUB, MSUBU, MULT and MULTU description pages listed these as Rev2 DSPASE, when they were actually included in Rev1.</td>
</tr>
<tr>
<td>2.24</td>
<td>September 03, 2009</td>
<td>• No content change. Rev 2.23 was generated with incorrect script parameters.</td>
</tr>
<tr>
<td>2.25</td>
<td>April 06, 2010</td>
<td>• Title change to match microMIPS32/64 and updated MIPS32/64 base ISA document sets.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• microMIPS mentioned in “About This Book” chapter.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Got rid of blank page.</td>
</tr>
<tr>
<td>2.30</td>
<td>October 20, 2010</td>
<td>• Some clean-up for microMIPS version. Those edits are not visible for MIPS32/64 versions.</td>
</tr>
<tr>
<td>2.31</td>
<td>March 20, 2011</td>
<td>• Reclassification of microMIPS AFP version. No changes for MIPS32/64.</td>
</tr>
<tr>
<td>2.32</td>
<td>March 21, 2011</td>
<td>• Edit for microMIPS. No changes for MIPS32/64.</td>
</tr>
<tr>
<td>2.33</td>
<td>April 23, 2011</td>
<td>• Remove the x fields in the instruction encoding diagrams. Replace them with explicit binary values.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MUL.PH &amp; MUL_S.PH had wrong minor opcode mnemonic string in the instruction description page. Binary value was correct.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• EXTR.W and EXTRV.W pseudocode – comparison checks are for 33bit values not 32bit values.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• PRECR_SRA[_R].PH.W, PRECR_SRA[_R].QH.PW not listed as DSPRev2 in Summary.</td>
</tr>
</tbody>
</table>
### Revision History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Comments</th>
</tr>
</thead>
</table>
| 2.34    | May 6, 2011   | - SPECIAL3 SHLL.QB instruction sub-class opcode changed from SLL.QB to SHLL.QB.  
- SPECIAL3 DPAQ.W.PH instruction sub-class name changed to DPA.W.PH  
- SHRA_R.W with shift amount 0 does not round – changed the pseudocode and created a new function, rnd32ShiftRightArithmetic(), which is shared with SHRAV_R.W.  
- SHRAV_R.W does not operate element-wise and the rounding is not optional – changed the description accordingly.  
- Pseudocode functions shift16Left(), sat16ShiftLeft(), and sat32ShiftLeft() fixed to show the correct discarded bits.  
- Pseudocode function shift8Left() fixed to handle unsigned bytes and to show the correct discarded bits.  
- MULQ_[R]S instructions’ pseudocode fixed to use a 64-bit temporary for the overflow condition.  
- Added a new pseudocode function for MUL.PH to set DSPControl bit 21 in case of overflow.  
- Changed pseudocode function sat16MultiplyQ15Q15() to set DSPControl bit 21 in case of overflow.  
- Added a new pseudocode function for MULEQ_S.W.PHL and MULEQ_S.W.PHR to set DSPControl bit 21 in case of overflow.  
- MODSUB pseudocode changed to use all 32 bits of source register.  
- Resorted some of the instructions in alphabetical order. |
| 2.40    | December 16, 2012 | - DSP ASE -> DSP Module  
- Updated logos on Cover  
- Updated copyright text |
| 2.41    | July 16, 2013  | - New cover page and legal text. |
| 3.00    | November 7, 2014 | - Release 6 new BPOSGE32C instruction. |
| 3.01    | December 15, 2014 | - New BPOSGE32C instruction  
- Modified Section 3.10 to note changes due to Release 6 of MIPS Architecture  
- Modified Section 3.2 and 3.11 to note detection of Rev 3.0.  
- Removed DSP3P references |