Unpublished rights (if any) reserved under the copyright laws of the United States of America and other countries.

This document contains information that is proprietary to MIPS Tech, LLC, a Wave Computing company ("MIPS") and MIPS’ affiliates as applicable. Any copying, reproducing, modifying or use of this information (in whole or in part) that is not expressly permitted in writing by MIPS or MIPS’ affiliates as applicable or an authorized third party is strictly prohibited. At a minimum, this information is protected under unfair competition and copyright laws. Violations thereof may result in criminal penalties and fines. Any document provided in source format (i.e., in a modifiable form such as in FrameMaker or Microsoft Word format) is subject to use and distribution restrictions that are independent of and supplemental to any and all confidentiality restrictions. UNDER NO CIRCUMSTANCES MAY A DOCUMENT PROVIDED IN SOURCE FORMAT BE DISTRIBUTED TO A THIRD PARTY IN SOURCE FORMAT WITHOUT THE EXPRESS WRITTEN PERMISSION OF MIPS (AND MIPS’ AFFILIATES AS APPLICABLE) reserve the right to change the information contained in this document to improve function, design or otherwise.

MIPS and MIPS’ affiliates do not assume any liability arising out of the application or use of this information, or of any error or omission in such information. Any warranties, whether express, statutory, implied or otherwise, including but not limited to the implied warranties of merchantability or fitness for a particular purpose, are excluded. Except as expressly provided in any written license agreement from MIPS or an authorized third party, the furnishing of this document does not give recipient any license to any intellectual property rights, including any patent rights, that cover the information in this document.

The information contained in this document shall not be exported, reexported, transferred, or released, directly or indirectly, in violation of the law of any country or international law, regulation, treaty, Executive Order, statute, amendments or supplements thereto. Should a conflict arise regarding the export, reexport, transfer, or release of the information contained in this document, the laws of the United States of America shall be the governing law.

The information contained in this document constitutes one or more of the following: commercial computer software, commercial computer software documentation or other commercial items. If the user of this information, or any related documentation of any kind, including related technical data or manuals, is an agency, department, or other entity of the United States government ("Government"), the use, duplication, reproduction, release, modification, disclosure, or transfer of this information, or any related documentation of any kind, is restricted in accordance with Federal Acquisition Regulation 12.212 for civilian agencies and Defense Federal Acquisition Regulation Supplement 227.7202 for military agencies. The use of this information by the Government is further restricted in accordance with the terms of the license agreement(s) and/or applicable contract terms and conditions covering this information from MIPS Technologies or an authorized third party.

MIPS, MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MiPsr3, MIPS32, MIPS64, microMIPS32, microMIPS64, MIPS-3D, MIPS16, MIPS16e, MIPS-Based, MIPSim, MIPSpro, MIPS-VERIFIED, Aptiv logo, microAptiv logo, interAptiv logo, microMIPS logo, MIPS Technologies logo, MIPS-VERIFIED logo, proAptiv logo, 4K, 4Kc, 4Km, 4Kp, 4KE, 4KEc, 4KEm, 4KEp, 4KS, 4KSc, 4Ksd, M4K, M14K, 5K, 5Kc, 5Kc, 5Kf, 24K, 24Kc, 24Kf, 24KF, 24KE, 24KEc, 24KEm, 24KEp, 34K, 34Kc, 34Kf, 34KF, 74K, 74Kc, 74KF, 1004K, 1004Kc, 1004KF, 1074K, 1074KF, R3000, R4000, R5000, Aptiv, ASMACRO, Atlas, "At the core of the user experience.", BusBridge, Bus Navigator, CLAM, CoreExtend, CoreFPGA, CoreLV, EC, FPGA View, FS2, FS2 FIRST SILICON SOLUTIONS logo, FS2 NAVIGATOR, HyperDebug, HyperJTAG, IASim, iFlowtrace, interAptiv, JALGO, Logic Navigator, Malta, MDMX, MED, MGB, microAptiv, microMIPS, Navigator, OCI, PDtrace, the Pipeline, proAptiv, Pro Series, SEAD-3, SmartMIPS, SOC-it, and YAMON are trademarks or registered trademarks of MIPS and MIPS’ affiliates as applicable in the United States and other countries.

All other trademarks referred to herein are the property of their respective owners.
Chapter 1: About This Book .................................................................................................................. 14
  1.1: Typographical Conventions ........................................................................................................ 14

Chapter 2: Guide to the Instruction Set .................................................................................................. 20
  2.1: Understanding the Instruction Fields .......................................................................................... 20
  2.1.1: Instruction Fields ......................................................................................................................... 21
  2.1.2: Instruction Descriptive Name and Mnemonic ........................................................................... 22
  2.1.3: Format Field ............................................................................................................................... 22
  2.1.4: Purpose Field ............................................................................................................................. 23
  2.1.5: Description Field ......................................................................................................................... 23
  2.1.6: Restrictions Field ........................................................................................................................ 23
  2.1.7: Operation Field ............................................................................................................................ 24
  2.1.8: Exceptions Field .......................................................................................................................... 24
  2.1.9: Programming Notes and Implementation Notes Fields ............................................................. 25
  2.2: Operation Section Notation and Functions ................................................................................. 25
  2.2.1: Instruction Execution Ordering ................................................................................................. 25
  2.2.2: Pseudocode Functions ............................................................................................................... 25
  2.3: Op and Function Subfield Notation ............................................................................................... 34
  2.4: FPU Instructions ............................................................................................................................ 34

Chapter 3: The MIPS32® Instruction Set ............................................................................................... 36
  3.1: Compliance and Subsetting ........................................................................................................... 36
  3.2: Alphabetical List of Instructions ................................................................................................... 37
    ABS.fmt ........................................................................................................................................... 46
    ADD ............................................................................................................................................... 47
    ADD.fmt ........................................................................................................................................ 48
    ADDI .............................................................................................................................................. 49
    ADDIU ......................................................................................................................................... 50
    ADDU .............................................................................................................................................. 51
    ALNV.PS ........................................................................................................................--------------- 52
    AND .............................................................................................................................................. 54
    ANDI ............................................................................................................................................. 55
    B ................................................................................................................................................. 56
    BAL ................................................................................................................................................ 57
    BC1F ............................................................................................................................................ 58
    BC1FL .......................................................................................................................................... 60
    BC1T ............................................................................................................................................. 62
    BC1TL ........................................................................................................................................ 64
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC2F</td>
<td>66</td>
</tr>
<tr>
<td>BC2FL</td>
<td>67</td>
</tr>
<tr>
<td>BC2T</td>
<td>68</td>
</tr>
<tr>
<td>BC2TL</td>
<td>69</td>
</tr>
<tr>
<td>BEQ</td>
<td>70</td>
</tr>
<tr>
<td>BEQL</td>
<td>71</td>
</tr>
<tr>
<td>BGEZ</td>
<td>72</td>
</tr>
<tr>
<td>BGEZALL</td>
<td>73</td>
</tr>
<tr>
<td>BGEZL</td>
<td>74</td>
</tr>
<tr>
<td>BG TZ</td>
<td>75</td>
</tr>
<tr>
<td>BGTZL</td>
<td>76</td>
</tr>
<tr>
<td>BLEZ</td>
<td>77</td>
</tr>
<tr>
<td>BLEZL</td>
<td>78</td>
</tr>
<tr>
<td>BLTZ</td>
<td>79</td>
</tr>
<tr>
<td>BLTZALL</td>
<td>80</td>
</tr>
<tr>
<td>BLTZL</td>
<td>81</td>
</tr>
<tr>
<td>BNE</td>
<td>82</td>
</tr>
<tr>
<td>BNEL</td>
<td>83</td>
</tr>
<tr>
<td>BREAK</td>
<td>84</td>
</tr>
<tr>
<td>C.cond.fmt</td>
<td>85</td>
</tr>
<tr>
<td>CACHE</td>
<td>86</td>
</tr>
<tr>
<td>CACHEE</td>
<td>87</td>
</tr>
<tr>
<td>CEIL.L.fmt</td>
<td>88</td>
</tr>
<tr>
<td>CEIL.W.fmt</td>
<td>89</td>
</tr>
<tr>
<td>CFC1</td>
<td>90</td>
</tr>
<tr>
<td>CFC2</td>
<td>91</td>
</tr>
<tr>
<td>CLO</td>
<td>92</td>
</tr>
<tr>
<td>CLZ</td>
<td>93</td>
</tr>
<tr>
<td>COP2</td>
<td>94</td>
</tr>
<tr>
<td>CTC1</td>
<td>95</td>
</tr>
<tr>
<td>CTC2</td>
<td>96</td>
</tr>
<tr>
<td>CVT.D.fmt</td>
<td>97</td>
</tr>
<tr>
<td>CVT.L.fmt</td>
<td>98</td>
</tr>
<tr>
<td>CVT.PS.S</td>
<td>99</td>
</tr>
<tr>
<td>CVT.S.fmt</td>
<td>100</td>
</tr>
<tr>
<td>CVT.S.PL</td>
<td>101</td>
</tr>
<tr>
<td>CVT.S.PU</td>
<td>102</td>
</tr>
<tr>
<td>CVT.W.fmt</td>
<td>103</td>
</tr>
<tr>
<td>DERET</td>
<td>104</td>
</tr>
<tr>
<td>DI</td>
<td>105</td>
</tr>
<tr>
<td>DIV</td>
<td>106</td>
</tr>
<tr>
<td>DIV.fmt</td>
<td>107</td>
</tr>
<tr>
<td>DIVU</td>
<td>108</td>
</tr>
<tr>
<td>EHB</td>
<td>109</td>
</tr>
<tr>
<td>EI</td>
<td>110</td>
</tr>
<tr>
<td>ERET</td>
<td>111</td>
</tr>
<tr>
<td>ERETNC</td>
<td>112</td>
</tr>
<tr>
<td>EXT</td>
<td>113</td>
</tr>
<tr>
<td>FLOOR.L.fmt</td>
<td>114</td>
</tr>
<tr>
<td>FLOOR.W.fmt</td>
<td>115</td>
</tr>
<tr>
<td>INS</td>
<td>116</td>
</tr>
<tr>
<td>INS</td>
<td>117</td>
</tr>
<tr>
<td>INS</td>
<td>118</td>
</tr>
<tr>
<td>INS</td>
<td>119</td>
</tr>
<tr>
<td>INS</td>
<td>120</td>
</tr>
<tr>
<td>INS</td>
<td>121</td>
</tr>
<tr>
<td>INS</td>
<td>122</td>
</tr>
<tr>
<td>INS</td>
<td>123</td>
</tr>
<tr>
<td>INS</td>
<td>124</td>
</tr>
<tr>
<td>INS</td>
<td>125</td>
</tr>
<tr>
<td>INS</td>
<td>126</td>
</tr>
<tr>
<td>INS</td>
<td>127</td>
</tr>
<tr>
<td>INS</td>
<td>128</td>
</tr>
<tr>
<td>INS</td>
<td>129</td>
</tr>
<tr>
<td>INS</td>
<td>130</td>
</tr>
<tr>
<td>INS</td>
<td>131</td>
</tr>
<tr>
<td>INS</td>
<td>132</td>
</tr>
<tr>
<td>INS</td>
<td>133</td>
</tr>
<tr>
<td>INS</td>
<td>134</td>
</tr>
<tr>
<td>INS</td>
<td>135</td>
</tr>
<tr>
<td>INS</td>
<td>136</td>
</tr>
<tr>
<td>INS</td>
<td>137</td>
</tr>
<tr>
<td>INS</td>
<td>138</td>
</tr>
<tr>
<td>INS</td>
<td>139</td>
</tr>
<tr>
<td>INS</td>
<td>140</td>
</tr>
<tr>
<td>Instruction</td>
<td>Page</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>J</td>
<td>142</td>
</tr>
<tr>
<td>JAL</td>
<td>143</td>
</tr>
<tr>
<td>JALR</td>
<td>144</td>
</tr>
<tr>
<td>JALR.HB</td>
<td>146</td>
</tr>
<tr>
<td>JALX</td>
<td>150</td>
</tr>
<tr>
<td>JR</td>
<td>151</td>
</tr>
<tr>
<td>JR.HB</td>
<td>153</td>
</tr>
<tr>
<td>LB</td>
<td>155</td>
</tr>
<tr>
<td>LBE</td>
<td>156</td>
</tr>
<tr>
<td>LBU</td>
<td>157</td>
</tr>
<tr>
<td>LBUE</td>
<td>158</td>
</tr>
<tr>
<td>LDC1</td>
<td>159</td>
</tr>
<tr>
<td>LDC2</td>
<td>160</td>
</tr>
<tr>
<td>LDXC1</td>
<td>161</td>
</tr>
<tr>
<td>LH</td>
<td>162</td>
</tr>
<tr>
<td>LHE</td>
<td>164</td>
</tr>
<tr>
<td>LHU</td>
<td>165</td>
</tr>
<tr>
<td>LHUE</td>
<td>166</td>
</tr>
<tr>
<td>LL</td>
<td>167</td>
</tr>
<tr>
<td>LLE</td>
<td>168</td>
</tr>
<tr>
<td>LUI</td>
<td>170</td>
</tr>
<tr>
<td>LUXC1</td>
<td>171</td>
</tr>
<tr>
<td>LW</td>
<td>172</td>
</tr>
<tr>
<td>LWC1</td>
<td>173</td>
</tr>
<tr>
<td>LWC2</td>
<td>174</td>
</tr>
<tr>
<td>LWE</td>
<td>176</td>
</tr>
<tr>
<td>LWL</td>
<td>177</td>
</tr>
<tr>
<td>LWLE</td>
<td>180</td>
</tr>
<tr>
<td>LWR</td>
<td>182</td>
</tr>
<tr>
<td>LWRE</td>
<td>184</td>
</tr>
<tr>
<td>LWXC1</td>
<td>186</td>
</tr>
<tr>
<td>MADD</td>
<td>187</td>
</tr>
<tr>
<td>MADDU</td>
<td>188</td>
</tr>
<tr>
<td>MFC0</td>
<td>190</td>
</tr>
<tr>
<td>MFC1</td>
<td>191</td>
</tr>
<tr>
<td>MFC2</td>
<td>192</td>
</tr>
<tr>
<td>MFHC0</td>
<td>193</td>
</tr>
<tr>
<td>MFHC1</td>
<td>195</td>
</tr>
<tr>
<td>MFHC2</td>
<td>196</td>
</tr>
<tr>
<td>MFHI</td>
<td>197</td>
</tr>
<tr>
<td>MFLO</td>
<td>198</td>
</tr>
<tr>
<td>MOV.fmt</td>
<td>199</td>
</tr>
<tr>
<td>MOVF</td>
<td>200</td>
</tr>
<tr>
<td>MOVF.fmt</td>
<td>201</td>
</tr>
<tr>
<td>MOVN</td>
<td>202</td>
</tr>
<tr>
<td>MOVN.fmt</td>
<td>203</td>
</tr>
<tr>
<td>MOVT</td>
<td>204</td>
</tr>
<tr>
<td>MOVT.fmt</td>
<td>205</td>
</tr>
<tr>
<td>MOVZ</td>
<td>206</td>
</tr>
<tr>
<td>MOVZ.fmt</td>
<td>207</td>
</tr>
<tr>
<td>MSUB</td>
<td>208</td>
</tr>
<tr>
<td>MSUB.fmt</td>
<td>209</td>
</tr>
<tr>
<td>MSUB.fmt</td>
<td>210</td>
</tr>
<tr>
<td>Instruction</td>
<td>Page</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>MSUBU</td>
<td>211</td>
</tr>
<tr>
<td>MTC0</td>
<td>212</td>
</tr>
<tr>
<td>MTC1</td>
<td>213</td>
</tr>
<tr>
<td>MTC2</td>
<td>215</td>
</tr>
<tr>
<td>MTHC0</td>
<td>216</td>
</tr>
<tr>
<td>MTHC1</td>
<td>217</td>
</tr>
<tr>
<td>MTHC2</td>
<td>218</td>
</tr>
<tr>
<td>MTHI</td>
<td>219</td>
</tr>
<tr>
<td>MTLO</td>
<td>220</td>
</tr>
<tr>
<td>MUL</td>
<td>221</td>
</tr>
<tr>
<td>MUL.fmt</td>
<td>222</td>
</tr>
<tr>
<td>MULT</td>
<td>223</td>
</tr>
<tr>
<td>MULTU</td>
<td>224</td>
</tr>
<tr>
<td>NEG.fmt</td>
<td>225</td>
</tr>
<tr>
<td>NMADD.fmt</td>
<td>226</td>
</tr>
<tr>
<td>NMSUB.fmt</td>
<td>227</td>
</tr>
<tr>
<td>NOP</td>
<td>229</td>
</tr>
<tr>
<td>NOR</td>
<td>230</td>
</tr>
<tr>
<td>OR</td>
<td>231</td>
</tr>
<tr>
<td>ORI</td>
<td>232</td>
</tr>
<tr>
<td>PAUSE</td>
<td>234</td>
</tr>
<tr>
<td>PLL.PS</td>
<td>236</td>
</tr>
<tr>
<td>PLU.PS</td>
<td>237</td>
</tr>
<tr>
<td>PREF</td>
<td>238</td>
</tr>
<tr>
<td>PREFE</td>
<td>242</td>
</tr>
<tr>
<td>PREFIX</td>
<td>245</td>
</tr>
<tr>
<td>PUL.PS</td>
<td>246</td>
</tr>
<tr>
<td>PUU.PS</td>
<td>247</td>
</tr>
<tr>
<td>RDHWR</td>
<td>248</td>
</tr>
<tr>
<td>RDPGPR</td>
<td>250</td>
</tr>
<tr>
<td>RECIP.fmt</td>
<td>251</td>
</tr>
<tr>
<td>ROTR</td>
<td>252</td>
</tr>
<tr>
<td>ROTRV</td>
<td>253</td>
</tr>
<tr>
<td>ROUND.L.fmt</td>
<td>254</td>
</tr>
<tr>
<td>ROUND.W.fmt</td>
<td>255</td>
</tr>
<tr>
<td>RSQRT.fmt</td>
<td>256</td>
</tr>
<tr>
<td>SB</td>
<td>257</td>
</tr>
<tr>
<td>SBE</td>
<td>258</td>
</tr>
<tr>
<td>SC</td>
<td>259</td>
</tr>
<tr>
<td>SCE</td>
<td>262</td>
</tr>
<tr>
<td>SDBBP</td>
<td>265</td>
</tr>
<tr>
<td>SDC1</td>
<td>266</td>
</tr>
<tr>
<td>SDC2</td>
<td>267</td>
</tr>
<tr>
<td>SDXC1</td>
<td>268</td>
</tr>
<tr>
<td>SEB</td>
<td>269</td>
</tr>
<tr>
<td>SEH</td>
<td>270</td>
</tr>
<tr>
<td>SH</td>
<td>271</td>
</tr>
<tr>
<td>SHE</td>
<td>272</td>
</tr>
<tr>
<td>SLL</td>
<td>273</td>
</tr>
<tr>
<td>SLLV</td>
<td>274</td>
</tr>
<tr>
<td>SLT</td>
<td>275</td>
</tr>
<tr>
<td>SLTI</td>
<td>276</td>
</tr>
<tr>
<td>SLTIU</td>
<td>277</td>
</tr>
<tr>
<td>Instruction</td>
<td>Page</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>XOR</td>
<td>340</td>
</tr>
<tr>
<td>WRPGPR</td>
<td>338</td>
</tr>
<tr>
<td>WAIT</td>
<td>337</td>
</tr>
<tr>
<td>TRUNC.W.fmt</td>
<td>336</td>
</tr>
<tr>
<td>TRUNC.L.fmt</td>
<td>335</td>
</tr>
<tr>
<td>SWR</td>
<td>334</td>
</tr>
<tr>
<td>SWC1</td>
<td>333</td>
</tr>
<tr>
<td>SWC2</td>
<td>332</td>
</tr>
<tr>
<td>SWE</td>
<td>331</td>
</tr>
<tr>
<td>SWL</td>
<td>330</td>
</tr>
<tr>
<td>SWLE</td>
<td>329</td>
</tr>
<tr>
<td>SWR</td>
<td>328</td>
</tr>
<tr>
<td>SWRE</td>
<td>327</td>
</tr>
<tr>
<td>SWXC1</td>
<td>326</td>
</tr>
<tr>
<td>SYNC</td>
<td>325</td>
</tr>
<tr>
<td>SYNCI</td>
<td>324</td>
</tr>
<tr>
<td>SYSCALL</td>
<td>323</td>
</tr>
<tr>
<td>TEQ</td>
<td>322</td>
</tr>
<tr>
<td>TEQI</td>
<td>321</td>
</tr>
<tr>
<td>TGE</td>
<td>320</td>
</tr>
<tr>
<td>TGEI</td>
<td>319</td>
</tr>
<tr>
<td>TGEIU</td>
<td>318</td>
</tr>
<tr>
<td>TGEU</td>
<td>317</td>
</tr>
<tr>
<td>TLBINV</td>
<td>316</td>
</tr>
<tr>
<td>TLBINVF</td>
<td>315</td>
</tr>
<tr>
<td>TLPB</td>
<td>314</td>
</tr>
<tr>
<td>TLBR</td>
<td>313</td>
</tr>
<tr>
<td>TLBWI</td>
<td>312</td>
</tr>
<tr>
<td>TLBWR</td>
<td>311</td>
</tr>
<tr>
<td>TLT</td>
<td>310</td>
</tr>
<tr>
<td>TLTI</td>
<td>309</td>
</tr>
<tr>
<td>TLTIU</td>
<td>308</td>
</tr>
<tr>
<td>TLTU</td>
<td>307</td>
</tr>
<tr>
<td>TNE</td>
<td>306</td>
</tr>
<tr>
<td>TNEI</td>
<td>305</td>
</tr>
<tr>
<td>TRUNC.L.fmt</td>
<td>304</td>
</tr>
<tr>
<td>TRUNC.W.fmt</td>
<td>303</td>
</tr>
<tr>
<td>WAIT</td>
<td>302</td>
</tr>
<tr>
<td>WRPGPR</td>
<td>301</td>
</tr>
<tr>
<td>WSBH</td>
<td>300</td>
</tr>
<tr>
<td>XOR</td>
<td>299</td>
</tr>
<tr>
<td>XORI</td>
<td>298</td>
</tr>
</tbody>
</table>

**Appendix A: Instruction Bit Encodings**

A.1: Instruction Encodings and Instruction Classes .................................................. 342
A.2: Instruction Bit Encoding Tables ........................................................................ 342
A.3: Floating Point Unit Instruction Format Encodings ............................................ 350
Figures

Figure 2.1: Example of Instruction Description ............................................................................................................. 21
Figure 2.2: Example of Instruction Fields ..................................................................................................................... 22
Figure 2.3: Example of Instruction Descriptive Name and Mnemonic ............................................................................... 22
Figure 2.4: Example of Instruction Format .................................................................................................................... 22
Figure 2.5: Example of Instruction Purpose .................................................................................................................. 23
Figure 2.6: Example of Instruction Description ............................................................................................................. 23
Figure 2.7: Example of Instruction Restrictions .......................................................................................................... 24
Figure 2.8: Example of Instruction Operation ............................................................................................................... 24
Figure 2.9: Example of Instruction Exception ................................................................................................................. 24
Figure 2.10: Example of Instruction Programming Notes .............................................................................................. 25
Figure 2.11: COP_LW Pseudocode Function ................................................................................................................ 26
Figure 2.12: COP_LD Pseudocode Function ................................................................................................................ 26
Figure 2.13: COP_SW Pseudocode Function ................................................................................................................ 26
Figure 2.14: COP_SD Pseudocode Function ................................................................................................................ 27
Figure 2.15: CoprocessorOperation Pseudocode Function ............................................................................................. 27
Figure 2.16: AddressTranslation Pseudocode Function .............................................................................................. 27
Figure 2.17: LoadMemory Pseudocode Function .......................................................................................................... 28
Figure 2.18: StoreMemory Pseudocode Function ......................................................................................................... 28
Figure 2.19: Prefetch Pseudocode Function ................................................................................................................ 29
Figure 2.20: SyncOperation Pseudocode Function ....................................................................................................... 30
Figure 2.21: ValueFPR Pseudocode Function ............................................................................................................... 30
Figure 2.22: StoreFPR Pseudocode Function ............................................................................................................... 31
Figure 2.23: CheckFPException Pseudocode Function ................................................................................................. 32
Figure 2.24: FPCconditionCode Pseudocode Function ................................................................................................. 32
Figure 2.25: SetFPconditionCode Pseudocode Function ............................................................................................. 32
Figure 2.26: SignalException Pseudocode Function .................................................................................................... 33
Figure 2.27: SignalDebugBreakpointException Pseudocode Function ........................................................................ 33
Figure 2.28: SignalDebugModeBreakpointException Pseudocode Function ................................................................. 33
Figure 2.29: NullifyCurrentInstruction PseudoCode Function .......................................................................................... 34
Figure 2.30: JumpDelaySlot Pseudocode Function ...................................................................................................... 34
Figure 2.31: PolyMult Pseudocode Function ................................................................................................................ 34
Figure 3.1: Example of an ALNV.PS Operation ................................................................................................................ 52
Figure 3.2: Usage of Address Fields to Select Index and Way ........................................................................................ 94
Figure 3.1: Usage of Address Fields to Select Index and Way ........................................................................................ 100
Figure 3.2: Operation of the EXT Instruction ................................................................................................................ 136
Figure 3.3: Operation of the INS Instruction ................................................................................................................ 140
Figure 3.4: Unaligned Word Load Using LWL and LWR ............................................................................................. 177
Figure 3.5: Bytes Loaded by LWL Instruction ................................................................................................................ 178
Figure 3.6: Unaligned Word Load Using LWLE and LWRE .......................................................................................... 180
Figure 3.7: Bytes Loaded by LWLE Instruction .............................................................................................................. 181
Figure 3.8: Unaligned Word Load Using LWL and LWR ............................................................................................... 182
Figure 3.9: Bytes Loaded by LWR Instruction ................................................................................................................. 183
Figure 3.10: Unaligned Word Load Using LWLE and LWRE ........................................................................................ 184
Figure 3.11: Bytes Loaded by LWRE Instruction ............................................................................................................ 185
Figure 4.12: Unaligned Word Store Using SWL and SWR ............................................................................................ 293
Figure 4.13: Bytes Stored by an SWL Instruction .......................................................................................................... 294
Figure 4.14: Unaligned Word Store Using SWLE and SWRE ..................................................................................... 296
Figure 4.15: Bytes Stored by an SWLE Instruction................................................................. 297
Figure 4.16: Unaligned Word Store Using SWR and SWL .................................................. 298
Figure 4.17: Bytes Stored by SWR Instruction................................................................. 299
Figure 4.18: Unaligned Word Store Using SWRE and SWLE ........................................... 300
Figure 4.19: Bytes Stored by SWRE Instruction............................................................... 301
Figure A.1: Sample Bit Encoding Table ........................................................................ 343
Figure B.1: LoadPossiblyMisaligned / StorePossiblyMisaligned pseudocode ................ 361
Figure B.2: LoadAligned / StoreAligned pseudocode ...................................................... 361
Figure B.3: LoadRawMemory Pseudocode Function ..................................................... 362
Figure B.4: StoreRawMemory Pseudocode Function .................................................... 362
Figure B.5: Byteswapping pseudocode functions ............................................................ 362
Figure B.6: LoadMisaligned most general pseudocode .................................................... 363
Figure B.7: Byte-by-byte pseudocode for LoadMisaligned / StoreMisaligned ............... 364
Figure B.8: LoadTYPEVector / StoreTYPEVector used by MSA specification ............... 365
Figure B.9: Pseudocode for LoadVector ................................................................. 366
Figure B.10: Pseudocode for StoreVector ......................................................................... 366
Tables

Table 1.1: Symbols Used in Instruction Operation Statements ...................................................................................... 16
Table 2.1: AccessLength Specifications for Loads/Stores ............................................................................................... 29
Table 3.1: CPU Arithmetic Instructions ....................................................................................................................... 37
Table 3.2: CPU Branch and Jump Instructions .............................................................................................................. 38
Table 3.3: CPU Instruction Control Instructions ........................................................................................................ 38
Table 3.4: CPU Load, Store, and Memory Control Instructions ..................................................................................... 38
Table 3.5: CPU Logical Instructions ......................................................................................................................... 39
Table 3.6: CPU Insert/Extract Instructions ................................................................................................................ 40
Table 3.7: CPU Move Instructions ............................................................................................................................... 40
Table 3.8: CPU Shift Instructions ................................................................................................................................ 40
Table 3.9: CPU Trap Instructions .................................................................................................................................. 41
Table 3.10: Obsolete CPU Branch Instructions ........................................................................................................... 41
Table 3.11: FPU Arithmetic Instructions ...................................................................................................................... 41
Table 3.12: FPU Branch Instructions ......................................................................................................................... 42
Table 3.13: FPU Compare Instructions ....................................................................................................................... 42
Table 3.14: FPU Convert Instructions ........................................................................................................................ 42
Table 3.15: FPU Load, Store, and Memory Control Instructions ................................................................................... 43
Table 3.16: FPU Move Instructions ................................................................................................................................ 43
Table 3.17: Obsolete FPU Branch Instructions ............................................................................................................. 44
Table 3.18: Coprocessor Branch Instructions ............................................................................................................... 44
Table 3.19: Coprocessor Execute Instructions ............................................................................................................. 44
Table 3.20: Coprocessor Load and Store Instructions .................................................................................................. 44
Table 3.21: Coprocessor Move Instructions ................................................................................................................ 44
Table 3.22: Obsolete Coprocessor Branch Instructions ............................................................................................... 45
Table 3.23: Privileged Instructions ................................................................................................................................ 45
Table 3.24: EJTAG Instructions ........................................................................................................................................ 45
Table 3.25: FPU Comparisons Without Special Operand Exceptions ........................................................................... 91
Table 3.26: FPU Comparisons With Special Operand Exceptions for QNaNs ................................................................. 92
Table 3.27: Usage of Effective Address ....................................................................................................................... 94
Table 3.28: Encoding of Bits[17:16] of CACHE Instruction ............................................................................................. 95
Table 3.29: Encoding of Bits[20:18] of the CACHE Instruction ........................................................................................ 96
Table 3.30: Usage of Effective Address .................................................................................................................... 100
Table 3.31: Encoding of Bits[17:16] of CACHEE Instruction ......................................................................................... 101
Table 3.32: Encoding of Bits[20:18] of the CACHEE Instruction .................................................................................... 102
Table 4.4: Values of hint Field for PREF Instruction ................................................................................................ 238
Table 4.5: Values of hint Field for PREFER Instruction ............................................................................................ 243
Table 4.6: RDHWR Register Numbers ..................................................................................................................... 248
Table 4.7: Encodings of the Bits[10:6] of the SYNC instruction; the SType Field ......................................................... 305
Table A.1: Symbols Used in the Instruction Encoding Tables .................................................................................. 343
Table A.2: MIPS32 Encoding of the Opcode Field .................................................................................................... 344
Table A.3: MIPS32 SPECIAL Opcode Encoding of Function Field .................................................................................. 345
Table A.4: MIPS32 REGIMM Encoding of rt Field ....................................................................................................... 345
Table A.5: MIPS32 SPECIAL2 Encoding of Function Field ........................................................................................ 345
Table A.6: MIPS32 SPECIAL3 Encoding of Function Field for Release 2 of the Architecture ................................................ 346
Table A.7: MIPS32 MOVCI Encoding of tt Bit ............................................................................................................... 346
Table A.8: MIPS32 SRL Encoding of Shift/Rotate .................................................................................................... 346
Table A.9: MIPS32 SRLV Encoding of Shift/Rotate .................................................................................................... 346
Table A.10: MIPS32 BSHFL Encoding of sa Field ................................................................. 347
Table A.11: MIPS32 COP0 Encoding of rs Field ............................................................... 347
Table A.12: MIPS32 COP0 Encoding of Function Field When rs = CO ............................... 347
Table A.13: MIPS32 COP1 Encoding of rs Field ............................................................... 348
Table A.14: MIPS32 COP1 Encoding of Function Field When rs = S ............................... 348
Table A.15: MIPS32 COP1 Encoding of Function Field When rs = D ............................... 348
Table A.16: MIPS32 COP1 Encoding of Function Field When rs = W or L ...................... 349
Table A.17: MIPS32 COP1 Encoding of Function Field When rs = PS ............................. 349
Table A.18: MIPS32 COP1 Encoding of tf Bit When rs = S, D, or PS, Function = MOVCF .... 349
Table A.19: MIPS32 COP2 Encoding of rs Field ............................................................. 350
Table A.20: MIPS32 COP1X Encoding of Function Field ............................................... 350
Table A.21: Floating Point Unit Instruction Format Encodings ........................................ 350
Chapter 1

About This Book

The MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set 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 microMIPS32™ 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.

• Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS64® Architecture and microMIPS64™. It is not applicable to the MIPS32® document set nor the microMIPS32™ document set. With Release 5 of the Architecture, MDMX is deprecated. MDMX and MSA can not be implemented at the same time.

• Volume IV-c describes the MIPS-3D® Application-Specific Extension to the MIPS® Architecture

• Volume IV-d describes the SmartMIPS® Application-Specific Extension to the MIPS32® Architecture and the microMIPS32™ Architecture.

• 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, 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, D, and PS
- 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 as pseudocode in a high-level language notation resembling Pascal. Special symbols used in the pseudocode notation are listed in Table 1.1.

#### Table 1.1 Symbols Used in Instruction Operation Statements

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>←</td>
<td>Assignment</td>
</tr>
<tr>
<td>=, ≠</td>
<td>Tests for equality and inequality</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>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>+, −</td>
<td>2’s complement or floating point arithmetic: addition, subtraction</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>*, ×</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[x]</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].</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>
<tr>
<td>COC[z]</td>
<td>Coprocessor unit z condition signal</td>
</tr>
<tr>
<td>Xlat[x]</td>
<td>Translation of the MIPS16e GPR number x into the corresponding 32-bit GPR number</td>
</tr>
</tbody>
</table>

*BigEndianMem* | Endian mode as configured at chip reset (0 → Little-Endian, 1 → 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. |

*BigEndianCPU* | The endianness for load and store instructions (0 → Little-Endian, 1 → 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). |

*ReverseEndian* | 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 (SRRE and User mode). |
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>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>
</tbody>
</table>
| **I**, **I+n**, **I-n** | 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 I+1.

The effect of pseudocode statements for the current instruction labelled I+1 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. |
| **PC** | The Program Counter value. During the instruction time of an instruction, this is the address of the instruction word. The address of the instruction that occurs during the next instruction time is determined by assigning a value to PC during an instruction time. If no value is assigned to PC during 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 PC during the instruction time of the instruction in the branch delay slot.

In the MIPS Architecture, the PC 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. The PC value contains a full 32-bit address all of which are significant during a memory reference. |
| **ISA Mode** | In processors that implement the MIPS16e Application Specific Extension or the microMIPS base architectures, the ISA Mode is a single-bit register that determines in which mode the processor is executing, as follows:

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The processor is executing 32-bit MIPS instructions</td>
</tr>
<tr>
<td>1</td>
<td>The processor is executing MIPS16e or microMIPS instructions</td>
</tr>
</tbody>
</table>

In the MIPS Architecture, the ISA Mode value is only visible indirectly, such as when the processor stores a combined value of the upper bits of PC and the ISA Mode into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception. |
| **PABITS** | The number of physical address bits implemented is represented by the symbol PABITS. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{36}$ bytes. |
### FP32RegistersMode

Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). In MIPS32 Release 1, the FPU has 32 32-bit FPRs in which 64-bit data types are stored in even-odd pairs of FPRs. In MIPS64, (and optionally in MIPS32 Release 2 and MIPSr3) the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR.

In MIPS32 Release 1 implementations, FP32RegistersMode is always a 0. MIPS64 implementations have a compatibility mode in which the processor references the FPRs as if it were a MIPS32 implementation. In such a case FP32RegisterMode is computed from the FR bit in the Status register. If this bit is a 0, the processor operates as if it had 32 32-bit FPRs. If this bit is a 1, the processor operates with 32 64-bit FPRs. The value of FP32RegisterMode is computed from the FR bit in the Status register.

### InstructionInBranchDelaySlot

Indicates whether the instruction at the Program Counter address was executed in the delay slot of a branch or jump. This condition reflects the dynamic state of the instruction, not the static state. That is, the value is false if a branch or jump occurs to an instruction whose PC immediately follows a branch or jump, but which is not executed in the delay slot of a branch or jump.

### SignalException(exception, argument)

Causes an exception to be signaled, using the exception parameter as the type of exception and the argument parameter as an exception-specific argument. Control does not return from this pseudocode function—the exception is signaled at the point of the call.

### 1.4 For More Information

Various MIPS RISC processor manuals and additional information about MIPS products can be found at the MIPS URL: [http://www.mips.com](http://www.mips.com)

For comments or questions on the MIPS32® Architecture or this document, send Email to support@mips.com.
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 21
- “Instruction Descriptive Name and Mnemonic” on page 22
- “Format Field” on page 22
- “Purpose Field” on page 23
- “Description Field” on page 23
- “Restrictions Field” on page 23
- “Operation Field” on page 24
- “Exceptions Field” on page 24
- “Programming Notes and Implementation Notes Fields” on page 25
Figure 2.1 Example of Instruction Description

Example Instruction Name | EXAMPLE
---|---

<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>EXAMPLE</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**Assembler format(s) for each definition**

**Short description**

**Symbolic description**

**Full description of instruction operation**

**Restrictions on instruction and operands**

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

**Exceptions that instruction can cause**

**Notes for programmers**

**Notes for implementors**

**Format:** EXAMPLE fd,rs,rt

**Purpose:** Example Instruction Name

To execute an EXAMPLE op.

**Description:** GPR[rd] ← GPR[rs] exampleop GPR[rt]

This section describes the operation of the instruction in text, tables, and illustrations. It includes information that would be difficult to encode in the Operation section.

**Restrictions:**

This section lists any restrictions for the instruction. This can include values of the instruction encoding fields such as register specifiers, operand values, operand formats, address alignment, instruction scheduling hazards, and type of memory access for addressed locations.

**Operation:**

```c
/* 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 25</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>6 5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>ADD</td>
</tr>
<tr>
<td>0000000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</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**

| Format: | ADD fd, rs, rt | MIPS32 |

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.

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 fmt field. For example, the ADD 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 *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 *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] ← 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>
<tr>
<td>• If the addition results in 32-bit 2’s complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.</td>
</tr>
<tr>
<td>• If the addition does not overflow, the 32-bit result is placed into GPR rd.</td>
</tr>
</tbody>
</table>

The body of the section is a description of the operation of the instruction in text, tables, and figures. This description complements the high-level language description in the *Operation* section.

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 Control / Status register.

### 2.1.6 Restrictions Field

The *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 ALNV.PS)
- Valid operand formats (for example, see floating point ADD fmt)
2.1 Understanding the Instruction Fields

- 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>
<tbody>
<tr>
<td>None</td>
</tr>
</tbody>
</table>

2.1.7 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:
    temp ← (GPR[rs]₃₁ || GPR[rs]₃₁..0) + (GPR[rt]₃₁ || GPR[rt]₃₁..0)
    if temp₃₂ ≠ temp₃₁ then
        SignalException(IntegerOverflow)
    else
        GPR[rd] ← temp
    endif
```

See 2.2 “Operation Section Notation and Functions” on page 25 for more information on the formal notation used here.

2.1.8 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

<table>
<thead>
<tr>
<th>Exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Overflow</td>
</tr>
</tbody>
</table>

An instruction may cause implementation-dependent exceptions that are not present in the *Exceptions* section.
2.1.9 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**

<table>
<thead>
<tr>
<th>Programming Notes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDU performs the same arithmetic operation but does not trap on overflow.</td>
</tr>
</tbody>
</table>

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 25
- “Pseudocode Functions” on page 25

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 25
- “Memory Operation Functions” on page 27
- “Floating Point Functions” on page 30
- “Miscellaneous Functions” on page 33

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 Operation Section Notation and Functions

**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**

```plaintext
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
```

**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**

```plaintext
COP_LD (z, rt, memdouble)
    z: The coprocessor unit number
    rt: Coprocessor general register specifier
    memdouble: A 64-bit doubleword value supplied to the coprocessor

    /* Coprocessor-dependent action */

    endfunction COP_LD
```

**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**

```plaintext
dataword ← COP_SW (z, rt)
    z: The coprocessor unit number
    rt: Coprocessor general register specifier
    dataword: 32-bit word value

    /* Coprocessor-dependent action */

    endfunction COP_SW
```

**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

```plaintext
datadouble ← COP_SD (z, rt)
z: The coprocessor unit number
rt: Coprocessor general register specifier
datadouble: 64-bit doubleword value

/* Coprocessor-dependent action */
endfunction COP_SD
```

**CoprocessorOperation**

The CoprocessorOperation function performs the specified Coprocessor operation.

Figure 2.15 CoprocessorOperation Pseudocode Function

```plaintext
CoprocessorOperation (z, cop_fun)

/* z: Coprocessor unit number */
/* cop_fun: Coprocessor function from function field of instruction */

/* Transmit the cop_fun value to coprocessor z */
endfunction 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 *AccessLength* 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.

**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 2.16 AddressTranslation Pseudocode Function

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

/* pAddr: physical address */
/* CCA: Cacheability&Coherency Attribute, the method used to access caches */
```
2.2 Operation Section Notation and Functions

Endfunction AddressTranslation

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 2.17 LoadMemory Pseudocode Function

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

endfunction LoadMemory

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 2.18 StoreMemory Pseudocode Function

StoreMemory (CCA, AccessLength, MemElem, pAddr, vAddr)
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 2.19 Prefetch Pseudocode Function**

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

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>

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 \textit{stype} occur in the same order for all processors.

**Figure 2.20 SyncOperation Pseudocode Function**

```plaintext
SyncOperation(stype)

/* stype: Type of load/store ordering to perform. */
/* Perform implementation-dependent operation to complete the */
/* required synchronization operation */
endfunction SyncOperation
```

### 2.2.2.3 Floating Point Functions

The pseudocode shown in 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).

**ValueFPR**

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

**Figure 2.21 ValueFPR Pseudocode Function**

```plaintext
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 (fpr0 ≠ 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
```

MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set, Revision 5.04
else
    valueFPR ← FPR[fpr]
endif

DEFAULT:
    valueFPR ← UNPREDICTABLE
endcase
endfunction ValueFPR

The pseudocode shown below specifies the way a binary encoding representing a formatted value is stored into CPI 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.

**StoreFPR**

Figure 2.22 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 formattted 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 */

case fmt of
    S, W, UNINTERPRETED_WORD:
        FPR[fpr] ← value
    D, UNINTERPRETED_DOUBLEWORD:
        if (FP32RegistersMode = 0)
            if (fpr0 ≠ 0) then
                UNPREDICTABLE
            else
                FPR[fpr] ← UNPREDICTABLE
                FPR[fpr+1] ← UNPREDICTABLE
                value31..0
                value63..32
            endif
        else
            FPR[fpr] ← value
        endif
    L, PS:
        if (FP32RegistersMode = 0) then
            UNPREDICTABLE
        else
            FPR[fpr] ← value
        endif
endcase
endfunction StoreFPR

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

**CheckFPException**

**Figure 2.23 CheckFPException Pseudocode Function**

```
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 ( (FCSR17 = 1) or
    ((FCSR16..12 and FCSR11..7) ≠ 0)) then
    SignalException(FloatingPointException)
endif
endfunction CheckFPException
```

**FPConditionCode**

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

**Figure 2.24 FPConditionCode Pseudocode Function**

```
fp ← 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 ← FCSR23
else
    FPConditionCode ← FCSR24+cc
endif
endfunction FPConditionCode
```

**SetFPConditionCode**

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

**Figure 2.25 SetFPConditionCode Pseudocode Function**

```
SetFPConditionCode(cc, tf)
if cc = 0 then
    FCSR ← FCSR31..24 || tf || FCSR22..0
else
    FCSR ← FCSR31..25+cc || tf || FCSR23+cc..0
endif
endfunction SetFPConditionCode
```
2.2.2.4 Miscellaneous Functions

This section lists miscellaneous functions not covered in previous sections.

**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 2.26 SignalException Pseudocode Function**

```plaintext
SignalException(Exception, argument)

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

**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 2.27 SignalDebugBreakpointException Pseudocode Function**

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

**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 2.28 SignalDebugModeBreakpointException Pseudocode Function**

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

**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.
2.3 Op and Function Subfield Notation

Figure 2.29 NullifyCurrentInstruction PseudoCode Function

```c
NullifyCurrentInstruction()
endfunction NullifyCurrentInstruction
```

**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 2.30 JumpDelaySlot Pseudocode Function

```c
JumpDelaySlot(vAddr)
   /* vAddr: Virtual address */
endfunction JumpDelaySlot
```

**PolyMult**

The PolyMult function multiplies two binary polynomial coefficients.

Figure 2.31 PolyMult Pseudocode Function

```c
PolyMult(x, y)
   temp ← 0
   for i in 0 .. 31
      if xi = 1 then
         temp ← temp xor (y(31-i)∥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 34 for a description of the op and function subfields.
Chapter 3

The MIPS32® Instruction Set

3.1 Compliance and Subsetting

To be compliant with the MIPS32 Architecture, designs must implement a set of required features, as described in this document set. To allow flexibility in implementations, the MIPS32 Architecture does provide subsetting rules. An implementation that follows these rules is compliant with the MIPS32 Architecture as long as it adheres strictly to the rules, and fully implements the remaining instructions. Supersetting of the MIPS32 Architecture is only allowed by adding functions to the SPECIAL2 major opcode, by adding control for co-processors via the COP2, LWC2, SWC2, LDC2, and/or SDC2, or via the addition of approved Application Specific Extensions.

Note: The use of COP3 as a customizable coprocessor has been removed in the Release 2 of the MIPS32 architecture. The use of the COP3 is now reserved for the future extension of the architecture. Implementations using Release 1 of the MIPS32 architecture are strongly discouraged from using the COP3 opcode for a user-available coprocessor as doing so will limit the potential for an upgrade path to a 64-bit floating point unit.

The instruction set subsetting rules are as follows:

- All non-privileged (does not need access to Coprocessor 0) CPU (non-FPU) instructions must be implemented - no subsetting is allowed (unless described in this list).

- The FPU and related support instructions, including the MOVF and MOVT CPU instructions, may be omitted. Software may determine if an FPU is implemented by checking the state of the FP bit in the Config1 CP0 register. If the FPU is implemented, it must include S, D, and W formats, operate instructions, and all supporting instructions. Software may determine which FPU data types are implemented by checking the appropriate bit in the FIR CP1 register. The following allowable FPU subsets are compliant with the MIPS32 architecture:
  - No FPU
  - FPU with S, D, and W formats and all supporting instructions

- Coprocessor 2 is optional and may be omitted. Software may determine if Coprocessor 2 is implemented by checking the state of the C2 bit in the Config1 CP0 register. If Coprocessor 2 is implemented, the Coprocessor 2 interface instructions (BC2, CFC2, COP2, CTC2, LDC2, LWC2, MFC2, MTC2, SDC2, and SWC2) may be omitted on an instruction-by-instruction basis.

- The standard TLB-based memory management unit may be replaced with a simpler MMU (e.g., a Fixed Mapping MMU). If this is done, the rest of the interface to the Privileged Resource Architecture must be preserved. If a TLB-based MMU is not implemented, the TLB related instructions can be subsetted out. Software may determine the type of the MMU by checking the MT field in the Config CP0 register.

- Instruction, CP0 Register, and CP1 Control Register fields that are marked “Reserved” or shown as “0” in the description of that field are reserved for future use by the architecture and are not available to implementations. Implementations may only use those fields that are explicitly reserved for implementation dependent use.
• Supported Modules and ASEs are optional and may be subsetted out. If most cases, software may determine if a supported Module/ASE is implemented by checking the appropriate bit in the Config1 or Config3 or Config4 CP0 register. If they are implemented, they must implement the entire ISA applicable to the component, or implement subsets that are approved by the ASE specifications.

• EJTAG is optional and may be subsetted out. If it is implemented, it must implement only those subsets that are approved by the EJTAG specification. If EJTAG is not implemented, the EJTAG instructions (SDBBP and DERET) can be subsetted out.

• The JALX instruction is only implemented when there are other instruction sets are available on the device (microMIPS or MIPS16e).

• EVA load/store (LWE, LHE, LHUE, LBE, LBUE, SWE, SHE, SBE) instructions are optional.

• If any instruction is subsetted out based on the rules above, an attempt to execute that instruction must cause the appropriate exception (typically Reserved Instruction or Coprocessor Unusable).

3.2 Alphabatical List of Instructions

Table 3.1 through Table 3.24 provide a list of instructions grouped by category. Individual instruction descriptions follow the tables, arranged in alphabetical order.

**Table 3.1 CPU Arithmetic Instructions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add Word</td>
</tr>
<tr>
<td>ADDI</td>
<td>Add Immediate Word</td>
</tr>
<tr>
<td>ADDIU</td>
<td>Add Immediate Unsigned Word</td>
</tr>
<tr>
<td>ADDU</td>
<td>Add Unsigned Word</td>
</tr>
<tr>
<td>CLO</td>
<td>Count Leading Ones in Word</td>
</tr>
<tr>
<td>CLZ</td>
<td>Count Leading Zeros in Word</td>
</tr>
<tr>
<td>DIV</td>
<td>Divide Word</td>
</tr>
<tr>
<td>DIVU</td>
<td>Divide Unsigned Word</td>
</tr>
<tr>
<td>MADD</td>
<td>Multiply and Add Word to Hi, Lo</td>
</tr>
<tr>
<td>MADDU</td>
<td>Multiply and Add Unsigned Word to Hi, Lo</td>
</tr>
<tr>
<td>MSUB</td>
<td>Multiply and Subtract Word to Hi, Lo</td>
</tr>
<tr>
<td>MSUBU</td>
<td>Multiply and Subtract Unsigned Word to Hi, Lo</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply Word to GPR</td>
</tr>
<tr>
<td>MULT</td>
<td>Multiply Word</td>
</tr>
<tr>
<td>MULTU</td>
<td>Multiply Unsigned Word</td>
</tr>
<tr>
<td>SEB</td>
<td>Sign-Extend Byte</td>
</tr>
<tr>
<td>SEH</td>
<td>Sign-Extend Halfword</td>
</tr>
<tr>
<td>SLT</td>
<td>Set on Less Than</td>
</tr>
<tr>
<td>SLTI</td>
<td>Set on Less Than Immediate</td>
</tr>
<tr>
<td>SLTIU</td>
<td>Set on Less Than Immediate Unsigned</td>
</tr>
<tr>
<td>SLTU</td>
<td>Set on Less Than Unsigned</td>
</tr>
</tbody>
</table>

Release 2 & subsequent
### Table 3.1 CPU Arithmetic Instructions (Continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB</td>
<td>Subtract Word</td>
</tr>
<tr>
<td>SUBU</td>
<td>Subtract Unsigned Word</td>
</tr>
</tbody>
</table>

### Table 3.2 CPU Branch and Jump Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Unconditional Branch</td>
</tr>
<tr>
<td>BAL</td>
<td>Branch and Link</td>
</tr>
<tr>
<td>BEQ</td>
<td>Branch on Equal</td>
</tr>
<tr>
<td>BGEZ</td>
<td>Branch on Greater Than or Equal to Zero</td>
</tr>
<tr>
<td>BGEZAL</td>
<td>Branch on Greater Than or Equal to Zero and Link</td>
</tr>
<tr>
<td>BGTZ</td>
<td>Branch on Greater Than Zero</td>
</tr>
<tr>
<td>BLEZ</td>
<td>Branch on Less Than or Equal to Zero</td>
</tr>
<tr>
<td>BLTZ</td>
<td>Branch on Less Than Zero</td>
</tr>
<tr>
<td>BLTZAL</td>
<td>Branch on Less Than Zero and Link</td>
</tr>
<tr>
<td>BNE</td>
<td>Branch on Not Equal</td>
</tr>
<tr>
<td>J</td>
<td>Jump</td>
</tr>
<tr>
<td>JAL</td>
<td>Jump and Link</td>
</tr>
<tr>
<td>JALR</td>
<td>Jump and Link Register</td>
</tr>
<tr>
<td>JALR.HB</td>
<td>Jump and Link Register with Hazard Barrier</td>
</tr>
<tr>
<td>JALX</td>
<td>Jump and Link Exchange</td>
</tr>
<tr>
<td>JR</td>
<td>Jump Register</td>
</tr>
<tr>
<td>JR.HB</td>
<td>Jump Register with Hazard Barrier</td>
</tr>
</tbody>
</table>

### Table 3.3 CPU Instruction Control Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>EHB</td>
<td>Execution Hazard Barrier</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
</tr>
<tr>
<td>PAUSE</td>
<td>Wait for LLBit to Clear</td>
</tr>
<tr>
<td>SSNOP</td>
<td>Superscalar No Operation</td>
</tr>
</tbody>
</table>

### Table 3.4 CPU Load, Store, and Memory Control Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>Load Byte</td>
</tr>
<tr>
<td>LBE</td>
<td>Load Byte EVA</td>
</tr>
</tbody>
</table>
### Table 3.4 CPU Load, Store, and Memory Control Instructions (Continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBU</td>
<td>Load Byte Unsigned</td>
<td></td>
</tr>
<tr>
<td>LBUE</td>
<td>Load Byte Unsigned EVA</td>
<td></td>
</tr>
<tr>
<td>LH</td>
<td>Load Halfword</td>
<td></td>
</tr>
<tr>
<td>LHE</td>
<td>Load Halfword EVA</td>
<td></td>
</tr>
<tr>
<td>LHU</td>
<td>Load Halfword Unsigned</td>
<td></td>
</tr>
<tr>
<td>LHUE</td>
<td>Load Halfword Unsigned EVA</td>
<td></td>
</tr>
<tr>
<td>LL</td>
<td>Load Linked Word</td>
<td></td>
</tr>
<tr>
<td>LLE</td>
<td>Load Linked Word-EVA</td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td>Load Word</td>
<td></td>
</tr>
<tr>
<td>LWE</td>
<td>Load Word EVA</td>
<td></td>
</tr>
<tr>
<td>LWL</td>
<td>Load Word Left</td>
<td></td>
</tr>
<tr>
<td>LWLE</td>
<td>Load Word Left EVA</td>
<td></td>
</tr>
<tr>
<td>LWR</td>
<td>Load Word Right</td>
<td></td>
</tr>
<tr>
<td>LWRE</td>
<td>Load Word Right EVA</td>
<td></td>
</tr>
<tr>
<td>PREF</td>
<td>Prefetch</td>
<td></td>
</tr>
<tr>
<td>PREFE</td>
<td>Prefetch-EVA</td>
<td></td>
</tr>
<tr>
<td>SB</td>
<td>Store Byte</td>
<td></td>
</tr>
<tr>
<td>SBE</td>
<td>Store Byte EVA</td>
<td></td>
</tr>
<tr>
<td>SC</td>
<td>Store Conditional Word</td>
<td></td>
</tr>
<tr>
<td>SCE</td>
<td>Store Conditional Word EVA</td>
<td></td>
</tr>
<tr>
<td>SH</td>
<td>Store Halfword</td>
<td></td>
</tr>
<tr>
<td>SHE</td>
<td>Store Halfword EVA</td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>Store Word</td>
<td></td>
</tr>
<tr>
<td>SWE</td>
<td>Store Word EVA</td>
<td></td>
</tr>
<tr>
<td>SWL</td>
<td>Store Word Left</td>
<td></td>
</tr>
<tr>
<td>SWLE</td>
<td>Store Word Left EVA</td>
<td></td>
</tr>
<tr>
<td>SWR</td>
<td>Store Word Right</td>
<td></td>
</tr>
<tr>
<td>SWRE</td>
<td>Store Word Right EVA</td>
<td></td>
</tr>
<tr>
<td>SYNC</td>
<td>Synchronize Shared Memory</td>
<td></td>
</tr>
<tr>
<td>SYNCI</td>
<td>Synchronize Caches to Make Instruction Writes Effective</td>
<td>Release 2 &amp; subsequent</td>
</tr>
</tbody>
</table>

### Table 3.5 CPU Logical Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>And</td>
</tr>
<tr>
<td>ANDI</td>
<td>And Immediate</td>
</tr>
<tr>
<td>LUI</td>
<td>Load Upper Immediate</td>
</tr>
</tbody>
</table>
### Table 3.5 CPU Logical Instructions (Continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOR</td>
<td>Not Or</td>
</tr>
<tr>
<td>OR</td>
<td>Or</td>
</tr>
<tr>
<td>ORI</td>
<td>Or Immediate</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive Or</td>
</tr>
<tr>
<td>XORI</td>
<td>Exclusive Or Immediate</td>
</tr>
</tbody>
</table>

### Table 3.6 CPU Insert/Extract Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXT</td>
<td>Extract Bit Field</td>
<td>Release 2 &amp; subsequent</td>
</tr>
<tr>
<td>INS</td>
<td>Insert Bit Field</td>
<td>Release 2 &amp; subsequent</td>
</tr>
<tr>
<td>WSBH</td>
<td>Word Swap Bytes Within Halfwords</td>
<td>Release 2 &amp; subsequent</td>
</tr>
</tbody>
</table>

### Table 3.7 CPU Move Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFHI</td>
<td>Move From HI Register</td>
</tr>
<tr>
<td>MFLO</td>
<td>Move From LO Register</td>
</tr>
<tr>
<td>MOVF</td>
<td>Move Conditional on Floating Point False</td>
</tr>
<tr>
<td>MOVN</td>
<td>Move Conditional on Not Zero</td>
</tr>
<tr>
<td>MOVN</td>
<td>Move Conditional on Not Zero</td>
</tr>
<tr>
<td>MOVN</td>
<td>Move Conditional on Floating Point True</td>
</tr>
<tr>
<td>MOVT</td>
<td>Move Conditional on Floating Point True</td>
</tr>
<tr>
<td>MOVT</td>
<td>Move Conditional on Zero</td>
</tr>
<tr>
<td>MTHI</td>
<td>Move To HI Register</td>
</tr>
<tr>
<td>MTLO</td>
<td>Move To LO Register</td>
</tr>
<tr>
<td>RDHWR</td>
<td>Read Hardware Register</td>
</tr>
</tbody>
</table>

### Table 3.8 CPU Shift Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROTR</td>
<td>Rotate Word Right</td>
<td>Release 2 &amp; subsequent</td>
</tr>
<tr>
<td>ROTRV</td>
<td>Rotate Word Right Variable</td>
<td>Release 2 &amp; subsequent</td>
</tr>
<tr>
<td>SLL</td>
<td>Shift Word Left Logical</td>
<td></td>
</tr>
<tr>
<td>SLLV</td>
<td>Shift Word Left Logical Variable</td>
<td></td>
</tr>
<tr>
<td>SRA</td>
<td>Shift Word Right Arithmetic</td>
<td></td>
</tr>
<tr>
<td>SRAV</td>
<td>Shift Word Right Arithmetic Variable</td>
<td></td>
</tr>
<tr>
<td>SRL</td>
<td>Shift Word Right Logical</td>
<td></td>
</tr>
<tr>
<td>SRLV</td>
<td>Shift Word Right Logical Variable</td>
<td></td>
</tr>
</tbody>
</table>
### Table 3.9 CPU Trap Instructions

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

### Table 3.10 Obsolete¹ CPU Branch Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQL</td>
<td>Branch on Equal Likely</td>
</tr>
<tr>
<td>BGEZALL</td>
<td>Branch on Greater Than or Equal to Zero and Link Likely</td>
</tr>
<tr>
<td>BGEZL</td>
<td>Branch on Greater Than or Equal to Zero Likely</td>
</tr>
<tr>
<td>BGZL</td>
<td>Branch on Greater Than Zero Likely</td>
</tr>
<tr>
<td>BLEZL</td>
<td>Branch on Less Than or Equal to Zero Likely</td>
</tr>
<tr>
<td>BLTZALL</td>
<td>Branch on Less Than Zero and Link Likely</td>
</tr>
<tr>
<td>BLTZL</td>
<td>Branch on Less Than Zero Likely</td>
</tr>
<tr>
<td>BNEL</td>
<td>Branch on Not Equal Likely</td>
</tr>
</tbody>
</table>

¹. Software is strongly encouraged to avoid use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS32 architecture.

### Table 3.11 FPU Arithmetic Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS fmt</td>
<td>Floating Point Absolute Value</td>
</tr>
<tr>
<td>ADD.fmt</td>
<td>Floating Point Add</td>
</tr>
<tr>
<td>DIV fmt</td>
<td>Floating Point Divide</td>
</tr>
<tr>
<td>MADD.fmt</td>
<td>Floating Point Multiply Add</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MADD.fmt</td>
<td>Release 2 &amp; subsequent</td>
</tr>
</tbody>
</table>
### Table 3.11 FPU Arithmetic Instructions (Continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Release 2 &amp; subsequent</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSUB fmt</td>
<td>Floating Point Multiply Subtract</td>
<td></td>
</tr>
<tr>
<td>MUL fmt</td>
<td>Floating Point Multiply</td>
<td></td>
</tr>
<tr>
<td>NEG.fmt</td>
<td>Floating Point Negate</td>
<td></td>
</tr>
<tr>
<td>NMADD fmt</td>
<td>Floating Point Negative Multiply Add</td>
<td></td>
</tr>
<tr>
<td>NMSUB fmt</td>
<td>Floating Point Negative Multiply Subtract</td>
<td></td>
</tr>
<tr>
<td>RECIP fmt</td>
<td>Reciprocal Approximation</td>
<td></td>
</tr>
<tr>
<td>RSQRT fmt</td>
<td>Reciprocal Square Root Approximation</td>
<td></td>
</tr>
<tr>
<td>SQRT fmt</td>
<td>Floating Point Square Root</td>
<td></td>
</tr>
<tr>
<td>SUB fmt</td>
<td>Floating Point Subtract</td>
<td></td>
</tr>
</tbody>
</table>

### Table 3.12 FPU Branch Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC1F</td>
<td>Branch on FP False</td>
</tr>
<tr>
<td>BC1T</td>
<td>Branch on FP True</td>
</tr>
</tbody>
</table>

### Table 3.13 FPU Compare Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.cond fmt</td>
<td>Floating Point Compare</td>
</tr>
</tbody>
</table>

### Table 3.14 FPU Convert Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>64-bit FPU Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALNV.PS</td>
<td>Floating Point Align Variable</td>
<td></td>
</tr>
<tr>
<td>CEIL.L fmt</td>
<td>Floating Point Ceiling Convert to Long Fixed Point</td>
<td></td>
</tr>
<tr>
<td>CEIL.W fmt</td>
<td>Floating Point Ceiling Convert to Word Fixed Point</td>
<td></td>
</tr>
<tr>
<td>CVT.D fmt</td>
<td>Floating Point Convert to Double Floating Point</td>
<td></td>
</tr>
<tr>
<td>CVT.L fmt</td>
<td>Floating Point Convert to Long Fixed Point</td>
<td></td>
</tr>
<tr>
<td>CVT.PS,S</td>
<td>Floating Point Convert Pair to Paired Single</td>
<td></td>
</tr>
<tr>
<td>CVT.S.PL</td>
<td>Floating Point Convert Pair Lower to Single Floating Point</td>
<td></td>
</tr>
<tr>
<td>CVT.S.PU</td>
<td>Floating Point Convert Pair Upper to Single Floating Point</td>
<td></td>
</tr>
<tr>
<td>CVT.S fmt</td>
<td>Floating Point Convert to Single Floating Point</td>
<td></td>
</tr>
<tr>
<td>CVT.W.fmt</td>
<td>Floating Point Convert to Word Fixed Point</td>
<td></td>
</tr>
<tr>
<td>FLOOR.L.fmt</td>
<td>Floating Point Floor Convert to Long Fixed Point</td>
<td></td>
</tr>
<tr>
<td>FLOOR.W.fmt</td>
<td>Floating Point Floor Convert to Word Fixed Point</td>
<td></td>
</tr>
<tr>
<td>PLL.PS</td>
<td>Pair Lower Lower</td>
<td></td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Instruction</td>
<td>Notes</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>PLU.PS</td>
<td>Pair Lower Upper</td>
<td>64-bit FPU Only</td>
</tr>
<tr>
<td>PUL.PS</td>
<td>Pair Upper Lower</td>
<td>64-bit FPU Only</td>
</tr>
<tr>
<td>PUU.PS</td>
<td>Pair Upper Upper</td>
<td>64-bit FPU Only</td>
</tr>
<tr>
<td>ROUND.L.fmt</td>
<td>Floating Point Round to Long Fixed Point</td>
<td>64-bit FPU Only</td>
</tr>
<tr>
<td>ROUND.W.fmt</td>
<td>Floating Point Round to Word Fixed Point</td>
<td></td>
</tr>
<tr>
<td>TRUNC.L.fmt</td>
<td>Floating Point Truncate to Long Fixed Point</td>
<td>64-bit FPU Only</td>
</tr>
<tr>
<td>TRUNC.W.fmt</td>
<td>Floating Point Truncate to Word Fixed Point</td>
<td></td>
</tr>
</tbody>
</table>

### Table 3.15 FPU Load, Store, and Memory Control Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDC1</td>
<td>Load Doubleword to Floating Point</td>
<td></td>
</tr>
<tr>
<td>LDXC1</td>
<td>Load Doubleword Indexed to Floating Point</td>
<td>Release 2 &amp; subsequent</td>
</tr>
<tr>
<td>LUXC1</td>
<td>Load Doubleword Indexed Unaligned to Floating Point</td>
<td>Release 2 &amp; subsequent</td>
</tr>
<tr>
<td>LWC1</td>
<td>Load Word to Floating Point</td>
<td></td>
</tr>
<tr>
<td>LWXC1</td>
<td>Load Word Indexed to Floating Point</td>
<td>Release 2 &amp; subsequent</td>
</tr>
<tr>
<td>PREFIX</td>
<td>Prefetch Indexed</td>
<td>Release 2 &amp; subsequent</td>
</tr>
<tr>
<td>SDC1</td>
<td>Store Doubleword from Floating Point</td>
<td></td>
</tr>
<tr>
<td>SDXC1</td>
<td>Store Doubleword Indexed from Floating Point</td>
<td>Release 2 &amp; subsequent</td>
</tr>
<tr>
<td>SUXC1</td>
<td>Store Doubleword Indexed Unaligned from Floating Point</td>
<td>Release 2 &amp; subsequent</td>
</tr>
<tr>
<td>SWC1</td>
<td>Store Word from Floating Point</td>
<td></td>
</tr>
<tr>
<td>SWXC1</td>
<td>Store Word Indexed from Floating Point</td>
<td>Release 2 &amp; subsequent</td>
</tr>
</tbody>
</table>

### Table 3.16 FPU Move Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFC1</td>
<td>Move Control Word from Floating Point</td>
<td></td>
</tr>
<tr>
<td>CTC1</td>
<td>Move Control Word to Floating Point</td>
<td></td>
</tr>
<tr>
<td>MFC1</td>
<td>Move Word from Floating Point</td>
<td></td>
</tr>
<tr>
<td>MFHC1</td>
<td>Move Word from High Half of Floating Point Register</td>
<td>Release 2 &amp; subsequent</td>
</tr>
<tr>
<td>MOV.fmt</td>
<td>Floating Point Move</td>
<td></td>
</tr>
<tr>
<td>MOVF.fmt</td>
<td>Floating Point Move Conditional on Floating Point False</td>
<td></td>
</tr>
<tr>
<td>MOVN.fmt</td>
<td>Floating Point Move Conditional on Not Zero</td>
<td></td>
</tr>
<tr>
<td>MOT.fmt</td>
<td>Floating Point Move Conditional on Floating Point True</td>
<td></td>
</tr>
<tr>
<td>MOVZ.fmt</td>
<td>Floating Point Move Conditional on Zero</td>
<td></td>
</tr>
<tr>
<td>MTC1</td>
<td>Move Word to Floating Point</td>
<td></td>
</tr>
<tr>
<td>MTHC1</td>
<td>Move Word to High Half of Floating Point Register</td>
<td>Release 2 &amp; subsequent</td>
</tr>
</tbody>
</table>
### 3.2 Alphabetical List of Instructions

#### Table 3.17 Obsolete FPU Branch Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC1FL</td>
<td>Branch on FP False Likely</td>
</tr>
<tr>
<td>BC1TL</td>
<td>Branch on FP True Likely</td>
</tr>
</tbody>
</table>

1. Software is strongly encouraged to avoid use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS32 architecture.

#### Table 3.18 Coprocessor Branch Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC2F</td>
<td>Branch on COP2 False</td>
</tr>
<tr>
<td>BC2T</td>
<td>Branch on COP2 True</td>
</tr>
</tbody>
</table>

#### Table 3.19 Coprocessor Execute Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP2</td>
<td>Coprocessor Operation to Coprocessor 2</td>
</tr>
</tbody>
</table>

#### Table 3.20 Coprocessor Load and Store Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDC2</td>
<td>Load Doubleword to Coprocessor 2</td>
</tr>
<tr>
<td>LWC2</td>
<td>Load Word to Coprocessor 2</td>
</tr>
<tr>
<td>SDC2</td>
<td>Store Doubleword from Coprocessor 2</td>
</tr>
<tr>
<td>SWC2</td>
<td>Store Word from Coprocessor 2</td>
</tr>
</tbody>
</table>

#### Table 3.21 Coprocessor Move Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFC2</td>
<td>Move Control Word from Coprocessor 2</td>
</tr>
<tr>
<td>CTC2</td>
<td>Move Control Word to Coprocessor 2</td>
</tr>
<tr>
<td>MFC2</td>
<td>Move Word from Coprocessor 2</td>
</tr>
<tr>
<td>MFHC2</td>
<td>Move Word from High Half of Coprocessor 2 Register Release 2 &amp; subsequent</td>
</tr>
<tr>
<td>MTC2</td>
<td>Move Word to Coprocessor 2</td>
</tr>
<tr>
<td>MTHC2</td>
<td>Move Word to High Half of Coprocessor 2 Register Release 2 &amp; subsequent</td>
</tr>
</tbody>
</table>
Table 3.22 Obsolete\(^1\) Coprocessor Branch Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC2FL</td>
<td>Branch on COP2 False Likely</td>
</tr>
<tr>
<td>BC2TL</td>
<td>Branch on COP2 True Likely</td>
</tr>
</tbody>
</table>

1. Software is strongly encouraged to avoid use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS32 architecture.

Table 3.23 Privileged Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHE</td>
<td>Perform Cache Operation</td>
<td></td>
</tr>
<tr>
<td>CACHEE</td>
<td>Perform Cache Operation EVA</td>
<td>3.03 &amp; subsequent</td>
</tr>
<tr>
<td>DI</td>
<td>Disable Interrupts</td>
<td>2 &amp; subsequent</td>
</tr>
<tr>
<td>EI</td>
<td>Enable Interrupts</td>
<td>2 &amp; subsequent</td>
</tr>
<tr>
<td>ERET</td>
<td>Exception Return</td>
<td></td>
</tr>
<tr>
<td>MFC0</td>
<td>Move from Coprocessor 0</td>
<td></td>
</tr>
<tr>
<td>MTC0</td>
<td>Move to Coprocessor 0</td>
<td></td>
</tr>
<tr>
<td>RDPGPR</td>
<td>Read GPR from Previous Shadow Set</td>
<td>2 &amp; subsequent</td>
</tr>
<tr>
<td>TLBP</td>
<td>Probe TLB for Matching Entry</td>
<td></td>
</tr>
<tr>
<td>TLBR</td>
<td>Read Indexed TLB Entry</td>
<td></td>
</tr>
<tr>
<td>TLBW1</td>
<td>Write Indexed TLB Entry</td>
<td></td>
</tr>
<tr>
<td>TLBWR</td>
<td>Write Random TLB Entry</td>
<td></td>
</tr>
<tr>
<td>WAIT</td>
<td>Enter Standby Mode</td>
<td></td>
</tr>
<tr>
<td>WRPGPR</td>
<td>Write GPR to Previous Shadow Set</td>
<td>2 &amp; subsequent</td>
</tr>
</tbody>
</table>

Table 3.24 EJTAG Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>DERET</td>
<td>Debug Exception Return</td>
</tr>
<tr>
<td>SDBBP</td>
<td>Software Debug Breakpoint</td>
</tr>
</tbody>
</table>
Floating Point Absolute Value

**ABS.fmt**

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>ABS</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>000101</td>
</tr>
</tbody>
</table>

**Format:**

ABS.fmt

ABS.S fd, fs  
ABS.D fd, fs  
ABS.PS fd, fs

MIPS32

MIPS32

MIPS64, MIPS32 Release 2

**Purpose:** Floating Point Absolute Value

**Description:** FPR[fd] ← abs(FPR[fs])

The absolute value of the value in FPR fs is placed in FPR fd. The operand and result are values in format fmt.

ABS.PS takes the absolute value of the two values in FPR fs independently, and ORs together any generated exceptions.

Cause bits are ORed into the Flag bits if no exception is taken.

If FIRHas2008=0 or FCSRABS2008=0 then this operation is arithmetic. For this case, any NaN operand signals invalid operation.

If FCSRABS2008=1 then this operation is non-arithmetic. For this case, both regular floating point numbers and NaN values are treated alike, only the sign bit is affected by this instruction. No IEEE exception can be generated for this case.

**Restrictions:**

The fields fs and fd must specify FPRs valid for operands of type fmt. If they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

The result of ABS.PS is UNPREDICTABLE if the processor is executing in the FR=0 32-bit FPU register model; i.e. it is predictable if executing on a 64-bit FPU in the FR=1 mode, but not with FR=0, and not on a 32-bit FPU.

**Operation:**

\[
\text{StoreFPR}(fd, fmt, \text{AbsoluteValue}(\text{ValueFPR}(fs, fmt)))
\]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation
Add Word

Format: \texttt{ADD rd, rs, rt}

Purpose: Add Word
To add 32-bit integers. If an overflow occurs, then trap.

Description: \texttt{GPR[rd] \leftarrow GPR[rs] + GPR[rt]}
The 32-bit word value in GPR \texttt{rt} is added to the 32-bit value in GPR \texttt{rs} to produce a 32-bit result.

- 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 \texttt{rd}.

Restrictions:
None

Operation:
\begin{verbatim}
\texttt{temp ~- (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)}
\texttt{if temp32 \neq temp31 then}
\texttt{SignalException(IntegerOverflow)}
\texttt{else}
\texttt{GPR[rd] \leftarrow temp}
\texttt{endif}
\end{verbatim}

Exceptions:
Integer Overflow

Programming Notes:
ADDU performs the same arithmetic operation but does not trap on overflow.
Floating Point Add

ADD.fmt

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>ADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>000000</td>
</tr>
</tbody>
</table>

Format:
- ADD.fmt
  - ADD.S fd, fs, ft  
  - ADD.D fd, fs, ft  
  - ADD.PS fd, fs, ft

MIPS32
MIPS32
MIPS64, MIPS32 Release 2

Purpose: Floating Point Add
To add floating point values

Description:
FPR[fd] ← FPR[fs] + FPR[ft]
The value in FPR ft is added to the value in FPR fs. The result is calculated to infinite precision, rounded by using to the current rounding mode in FCSR, and placed into FPR fd. The operands and result are values in format fmt.
ADD.PS adds the upper and lower halves of FPR fs and FPR ft independently, and ORs together any generated exceptions.

Cause bits are ORed into the Flag bits if no exception is taken.

Restrictions:
The fields fs, ft, and fd must specify FPRs valid for operands of type fmt. If they are not valid, the result is UNPREDICTABLE.
The operands must be values in format fmt; if they are not, the result is UNPREDICTABLE and the value of the operand FPRs becomes UNPREDICTABLE.
The result of ADD.PS is UNPREDICTABLE if the processor is executing in the FR=0 32-bit FPU register model; i.e. it is predictable if executing on a 64-bit FPU in the FR=1 mode, but not with FR=0, and not on a 32-bit FPU.

Operation:

StoreFPR(fd, fmt, ValueFPR(fs, fmt) +fmt ValueFPR(ft, fmt))

Exceptions:
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:
Unimplemented Operation, Invalid Operation, Inexact, Overflow, Underflow
Add Immediate Word

Format: \texttt{ADDI \textit{rt}, \textit{rs}, \textit{immediate}}

Purpose: Add Immediate Word
To add a constant to a 32-bit integer. If overflow occurs, then trap.

Description: \texttt{GPR[rt] \leftarrow GPR[rs] + \textit{immediate}}
The 16-bit signed \textit{immediate} is added to the 32-bit value in \textit{GPR rs} to produce a 32-bit result.

- 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 \textit{GPR rt}.

Restrictions: None

Operation:
\begin{verbatim}
  temp \leftarrow (GPR[rs]_{31}|GPR[rs]_{31..0}) + \text{sign\_extend(\textit{immediate})}
  \text{if } temp_{32} \neq temp_{31} \text{ then}
    \text{SignalException(IntegerOverflow)}
  \text{else}
    \text{GPR[rt] \leftarrow temp}
\end{verbatim}

Exceptions: Integer Overflow

Programming Notes:
ADDIU performs the same arithmetic operation but does not trap on overflow.
### Add Immediate Unsigned Word

**Format:** ADDIU rt, rs, immediate

**Purpose:** Add Immediate Unsigned Word
To add a constant to a 32-bit integer

**Description:** $\text{GPR}_r \leftarrow \text{GPR}_s + \text{immediate}$
The 16-bit signed `immediate` is added to the 32-bit value in GPR $s$ and the 32-bit arithmetic result is placed into GPR $t$.
No Integer Overflow exception occurs under any circumstances.

**Restrictions:**
None

**Operation:**
\[
\text{temp} \leftarrow \text{GPR}_s + \text{sign} \_\text{extend} (\text{immediate}) \\
\text{GPR}_t \leftarrow \text{temp}
\]

**Exceptions:**
None

**Programming Notes:**
The term "unsigned" in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDIU</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
</table>
Add Unsigned Word

Add Unsigned Word

Purpose: Add Unsigned Word
To add 32-bit integers

Description: \( GPR[rd] \leftarrow GPR[rs] + GPR[rt] \)
The 32-bit word value in \( GPR_{rt} \) is added to the 32-bit value in \( GPR_{rs} \) and the 32-bit arithmetic result is placed into \( GPR_{rd} \).
No Integer Overflow exception occurs under any circumstances.

Restrictions:
None

Operation:
\[
\begin{align*}
\text{temp} & \leftarrow GPR[rs] + GPR[rt] \\
GPR[rd] & \leftarrow \text{temp}
\end{align*}
\]

Exceptions:
None

Programming Notes:
The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.
Floating Point Align Variable

<table>
<thead>
<tr>
<th>COP1X</th>
<th>rs</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>ALNV.PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>010011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>011110</td>
</tr>
</tbody>
</table>

**Format:** ALNV.PS fd, fs, ft, rs

**MIPS64, MIPS32 Release 2**

**Purpose:** Floating Point Align Variable

To align a misaligned pair of paired single values

**Description:**

\[ FPR[fd] \leftarrow \text{ByteAlign}(GPR[rs]_{2..0}, FPR[fs], FPR[ft]) \]

FPR fs is concatenated with FPR ft and this value is funnel-shifted by GPR rs_{2..0} bytes, and written into FPR fd. If GPR rs_{2..0} is 0, FPR fd receives FPR fs. If GPR rs_{2..0} is 4, the operation depends on the current endianness.

Figure 3-1 illustrates the following example: for a big-endian operation and a byte alignment of 4, the upper half of FPR fd receives the lower half of the paired single value in fs, and the lower half of FPR fd receives the upper half of the paired single value in FPR ft.

**Figure 3.1 Example of an ALNV.PS Operation**

![Figure 3.1](image)

The move is non arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

- The fields fs, ft, and fd must specify FPRs valid for operands of type PS. If they are not valid, the result is UNPREDICTABLE.
- If GPR rs_{1..0} are non-zero, the results are UNPREDICTABLE.

The result of this instruction is UNPREDICTABLE if the processor is executing in the FR=0 32-bit FPU register model; i.e. it is predictable if executing on a 64-bit FPU in the FR=1 mode, but not with FR=0, and not on a 32-bit FPU.

**Operation:**

- if GPR[rs]_{2..0} = 0 then
  - StoreFPR(fd, PS, ValueFPR(fs, PS))
- else if GPR[rs]_{2..0} \neq 4 then
  - UNPREDICTABLE
- else if BigEndianCPU then
  - StoreFPR(fd, PS, ValueFPR(fs, PS)_{31..0} || ValueFPR(ft, PS)_{63..32})

MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set, Revision 5.04
else
  StoreFPR(fd, PS, ValueFPR(ft, PS)_31..0 || ValueFPR(fs, PS)_63..32)
endif

Exceptions:
Coprocessor Unusable, Reserved Instruction

Programming Notes:
ALNV.PS is designed to be used with LUXC1 to load 8 bytes of data from any 4-byte boundary. For example:

/* Copy T2 bytes (a multiple of 16) of data T0 to T1, T0 unaligned, T1 aligned. */
  LUXC1 F0, 0(T0) /* set up by reading 1st src dw */
  LI T3, 0 /* index into src and dst arrays */
  ADDIU T4, T0, 8 /* base for odd dw loads */
  ADDIU T5, T1, -8/* base for odd dw stores */
LOOP:
  LUXC1 F1, T3(T4)
  ALNV.PS F2, F0, F1, T0/* switch F0, F1 for little-endian */
  SDC1 F2, T3(T1)
  ADDIU T3, T3, 16
  LUXC1 F0, T3(T0)
  ALNV.PS F2, F1, F0, T0/* switch F1, F0 for little-endian */
  BNE T3, T2, LOOP
  SDC1 F2, T3(T5)
DONE:

ALNV.PS is also useful with SUXC1 to store paired-single results in a vector loop to a possibly misaligned address:

/* T1[i] = T0[i] + F8, T0 aligned, T1 unaligned. */
  CVT.PS.S F8, F8, F8/* make addend paired-single */
/* Loop header computes 1st pair into F0, stores high half if T1 */
/* misaligned */
LOOP:
  LDC1 F2, T3(T4)/* get T0[i+2]/T0[i+3] */
  ADD.PS F1, F2, F8/* compute T1[i+2]/T1[i+3] */
  ALNV.PS F3, F0, F1, T1/* align to dst memory */
  SUXC1 F3, T3(T1)/* store to T1[i+0]/T1[i+1] */
  ADDIU T3, 16 /* i = i + 4 */
  LDC1 F2, T3(T0)/* get T0[i+0]/T0[i+1] */
  ADD.PS F0, F2, F8/* compute T1[i+0]/T1[i+1] */
  ALNV.PS F3, F1, F0, T1/* align to dst memory */
  BNE T3, T2, LOOP
  SUXC1 F3, T3(T5)/* store to T1[i+2]/T1[i+3] */
/* Loop trailer stores all or half of F0, depending on T1 alignment */
**Purpose:** And
To do a bitwise logical AND

**Description:** \(GPR[rd] \leftarrow GPR[rs] \text{ AND } GPR[rt]\)

The contents of GPR \(rs\) are combined with the contents of GPR \(rt\) in a bitwise logical AND operation. The result is placed into GPR \(rd\).

**Restrictions:**
None

**Operation:**
\[GPR[rd] \leftarrow GPR[rs] \text{ and } GPR[rt]\]

**Exceptions:**
None
**And Immediate**

- **Format:** ANDI rt, rs, immediate
- **Purpose:** And Immediate
  - To do a bitwise logical AND with a constant
- **Description:** GPR[rt] ← GPR[rs] AND immediate
  - The 16-bit immediate is zero-extended to the left and combined with the contents of GPR rs in a bitwise logical AND operation. The result is placed into GPR rt.
- **Restrictions:**
  - None
- **Operation:**
  - GPR[rt] ← GPR[rs] and zero_extend(immediate)
- **Exceptions:**
  - None
Unconditional Branch

Purpose: Unconditional Branch
To do an unconditional branch

Description: branch
B offset is the assembly idiom used to denote an unconditional branch. The actual instruction is interpreted by the hardware as BEQ r0, r0, offset.

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

Restrictions:
Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

I: \( \text{target\_offset} \leftarrow \text{sign\_extend}(\text{offset} \mid 02) \)
I+1: \( \text{PC} \leftarrow \text{PC} + \text{target\_offset} \)

Exceptions:
None

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 this range.
Branch and Link

**Format:** BAL offset

**Assembly Idiom**

**Purpose:** Branch and Link

To do an unconditional PC-relative procedure call

**Description:** `procedure_call`

BAL offset is the assembly idiom used to denote an unconditional branch. The actual instruction is interpreted by the hardware as BGEZAL r0, offset.

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

An 18-bit signed offset (the 16-bit `offset` field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

GPR 31 must not be used for the source register `rs`, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is **UNPREDICTABLE**. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

**Operation:**

\[

target\_offset \leftarrow \text{sign\_extend}(\text{offset} \; || \; 0^2) \\
\text{GPR}[31] \leftarrow \text{PC} + 8 \\
\text{I+1:} \quad \text{PC} \leftarrow \text{PC} + \text{target\_offset}
\]

**Exceptions:**

None

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch range is ±128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.
Branch on FP False

**Format:**

BC1F offset (cc = 0 implied)  
MIPS32  
BC1F cc, offset  
MIPS32

**Purpose:** Branch on FP False

To test an FP condition code and do a PC-relative conditional branch

**Description:** if FPConditionCode(cc) = 0 then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the FP condition code bit cc is false (0), the program branches to the effective target address after the instruction in the delay slot is executed. An FP condition code is set by the FP compare instruction, C.cond.fmt.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

This operation specification is for the general Branch On Condition operation with the tf (true/false) and nd (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for tf and nd.

I:  
condition ← FPConditionCode(cc) = 0  
target_offset ← (offset15)GPRLEN−(16+2) || offset || 02

I+1:  
if condition then  
    PC ← PC + target_offset  
endif

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation

**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 this range

**Historical Information:**

The MIPS I architecture defines a single floating point condition code, implemented as the coprocessor 1 condition signal (Cp1Cond) and the C bit in the FP Control/Status register. MIPS I, II, and III architectures must have the CC field set to 0, which is implied by the first format in the “Format” section.

The MIPS IV and MIPS32 architectures add seven more Condition Code bits to the original condition code 0. FP compare and conditional branch instructions specify the Condition Code bit to set or test. Both assembler formats are valid for MIPS IV and MIPS32.

In the MIPS I, II, and III architectures there must be at least one instruction between the compare instruction that sets
the condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.
**Branch on FP False Likely**

**BC1FL**

**Format:**
- BC1FL offset (cc = 0 implied)
- BC1FL cc, offset

**Purpose:** Branch on FP False Likely

To test an FP condition code and make a PC-relative conditional branch; execute the instruction in the delay slot only if the branch is taken.

**Description:** if FPConditionCode(cc) = 0 then branch_likely

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the FP Condition Code bit cc is false (0), the program branches to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

An FP condition code is set by the FP compare instruction, C.cond fmt.

**Restrictions:**

Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

This operation specification is for the general Branch On Condition operation with the tf (true/false) and nd (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for tf and nd.

- **I:**
  - condition ← FPConditionCode(cc) = 0
  - target_offset ← (offset_{15})_{GPRLEN-(16+2)} || offset || 0^2

- **I+1:**
  - if condition then
  - PC ← PC + target_offset
  - else
  - NullifyCurrentInstruction()
  - endif

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation

**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 this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is
encouraged to use the BC1F instruction instead.

**Historical Information:**

The MIPS I architecture defines a single floating point condition code, implemented as the coprocessor 1 condition signal (Cp1Cond) and the C bit in the FP Control/Status register. MIPS I, II, and III architectures must have the CC field set to 0, which is implied by the first format in the “Format” section.

The MIPS IV and MIPS32 architectures add seven more Condition Code bits to the original condition code 0. FP compare and conditional branch instructions specify the Condition Code bit to set or test. Both assembler formats are valid for MIPS IV and MIPS32.

In the MIPS II and III architectures, there must be at least one instruction between the compare instruction that sets a condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.
Branch on FP True

**Format:**

```
BC1T offset (cc = 0 implied)  
BC1T cc, offset
```

**Purpose:**

Branch on FP True

To test an FP condition code and do a PC-relative conditional branch

**Description:**

```
if FPConditionCode(cc) = 1 then branch
```

An 18-bit signed offset (the 16-bit `offset` field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the FP condition code bit `cc` is true (1), the program branches to the effective target address after the instruction in the delay slot is executed. An FP condition code is set by the FP compare instruction, C.cond.fmt.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

This operation specification is for the general Branch On Condition operation with the `tf` (true/false) and `nd` (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for `tf` and `nd`.

```
I: condition ← FPConditionCode(cc) = 1  
target_offset ← (offset15)GPRLEN-(16+2) || offset || 02
I+1: if condition then  
      PC ← PC + target_offset  
      endif
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation

**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 this range.

**Historical Information:**

The MIPS I architecture defines a single floating point condition code, implemented as the coprocessor 1 condition signal (`Cp1Cond`) and the `C` bit in the FP Control/Status register. MIPS I, II, and III architectures must have the `CC` field set to 0, which is implied by the first format in the “Format” section.

The MIPS IV and MIPS32 architectures add seven more `Condition Code` bits to the original condition code 0. FP compare and conditional branch instructions specify the `Condition Code` bit to set or test. Both assembler formats are valid for MIPS IV and MIPS32.

In the MIPS I, II, and III architectures there must be at least one instruction between the compare instruction that sets
the condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.
Branch on FP True Likely

**Purpose:** Branch on FP True Likely

To test an FP condition code and do a PC-relative conditional branch; execute the instruction in the delay slot only if the branch is taken.

**Description:**

if FPConditionCode(cc) = 1 then branch_likely

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the FP Condition Code bit cc is true (1), the program branches to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

An FP condition code is set by the FP compare instruction, C.cond fmt.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

This operation specification is for the general Branch On Condition operation with the tf (true/false) and nd (nullify delay slot) fields as variables. The individual instructions BC1F, BC1FL, BC1T, and BC1TL have specific values for tf and nd.

\[
\begin{align*}
\text{I:} & \quad \text{condition} \leftarrow \text{FPConditionCode}(cc) = 1 \\
& \quad \text{target_offset} \leftarrow (\text{offset15})^{\text{GPRLEN}-(16+2)} || \text{offset} || 0^2 \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \quad \text{PC} \leftarrow \text{PC} + \text{target_offset} \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{NullifyCurrentInstruction()} \\
& \quad \text{endif}
\end{align*}
\]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation

**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 this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch
will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BC1T instruction instead.

**Historical Information:**

The MIPS I architecture defines a single floating point condition code, implemented as the coprocessor 1 condition signal \((Cp1Cond)\) and the \(C\) bit in the FP Control/Status register. MIPS I, II, and III architectures must have the \(CC\) field set to 0, which is implied by the first format in the “Format” section.

The MIPS IV and MIPS32 architectures add seven more Condition Code bits to the original condition code 0. FP compare and conditional branch instructions specify the Condition Code bit to set or test. Both assembler formats are valid for MIPS IV and MIPS32.

In the MIPS II and III architectures, there must be at least one instruction between the compare instruction that sets a condition code and the branch instruction that tests it. Hardware does not detect a violation of this restriction.
Branch on COP2 False

**Format:**
- BC2F offset (cc = 0 implied)
- BC2F cc, offset

**Purpose:** Branch on COP2 False
To test a COP2 condition code and do a PC-relative conditional branch

**Description:**
if COP2Condition(cc) = 0 then branch
An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the COP2 condition specified by cc is false (0), the program branches to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**
Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**
This operation specification is for the general Branch On Condition operation with the tf (true/false) and nd (nullify delay slot) fields as variables. The individual instructions BC2F, BC2FL, BC2T, and BC2TL have specific values for tf and nd.

```
I:    condition ← COP2Condition(cc) = 0
      target_offset ← (offset15)GPRLEN-(16+2) || offset || 0²
I+1:  if condition then
      PC ← PC + target_offset
      endif
```

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**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 this range.
Branch on COP2 False Likely

**Purpose:** Branch on COP2 False Likely

To test a COP2 condition code and make a PC-relative conditional branch; execute the instruction in the delay slot only if the branch is taken.

**Description:**

\[
\text{if COP2Condition}(cc) = 0 \text{ then branch\_likely}
\]

An 18-bit signed offset (the 16-bit \textit{offset} field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the COP2 condition specified by \textit{cc} is false (0), the program branches to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

This operation specification is for the general Branch On Condition operation with the \textit{tf} (true/false) and \textit{nd} (nullify delay slot) fields as variables. The individual instructions BC2F, BC2FL, BC2T, and BC2TL have specific values for \textit{tf} and \textit{nd}.

\[
\begin{align*}
\text{I:} & \quad \text{condition} \leftarrow \text{COP2Condition}(cc) = 0 \\
\text{target\_offset} & \leftarrow (\text{offset}_{15})_{\text{GPRLEN)-(16+2)}} \left\lfloor | \text{offset} \right\rfloor + 0^2 \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \quad \text{PC} \leftarrow \text{PC} + \text{target\_offset} \\
& \quad \quad \text{else} \\
& \quad \quad \text{NullifyCurrentInstruction}() \\
& \quad \text{endif}
\end{align*}
\]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**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 this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BC2F instruction instead.
Branch on COP2 True

**Format:**  
BC2T offset (cc = 0 implied)  
BC2T cc, offset

**Purpose:** Branch on COP2 True  
To test a COP2 condition code and do a PC-relative conditional branch

**Description:** if COP2Condition(cc) = 1 then branch  
An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the COP2 condition specified by cc is true (1), the program branches to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**  
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**  
This operation specification is for the general Branch On Condition operation with the *tf* (true/false) and *nd* (nullify delay slot) fields as variables. The individual instructions BC2F, BC2FL, BC2T, and BC2TL have specific values for *tf* and *nd*.

I:  
condition ← COP2Condition(cc) = 1  
target_offset ← (offset_{15})^{GPRLEN-(16+2)} || offset || 0^2

I+1:  
if condition then  
    PC ← PC + target_offset  
endif

**Exceptions:**  
Coprocessor Unusable, Reserved Instruction

**Programming Notes:**  
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytesj. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.
Branch on COP2 True Likely

**Purpose:** Branch on COP2 True Likely

To test a COP2 condition code and do a PC-relative conditional branch; execute the instruction in the delay slot only if the branch is taken.

**Description:**

\[
\text{if COP2Condition(cc) = 1 then branch\_likely}
\]

An 18-bit signed offset (the 16-bit `offset` field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If the COP2 condition specified by `cc` is true (1), the program branches to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

This operation specification is for the general Branch On Condition operation with the `tf` (true/false) and `nd` (nullify delay slot) fields as variables. The individual instructions BC2F, BC2FL, BC2T, and BC2TL have specific values for `tf` and `nd`.

\[
\begin{align*}
\text{I:} & \quad \text{condition} \leftarrow \text{COP2Condition}(cc) = 1 \\
& \quad \text{target\_offset} \leftarrow (\text{offset}_{15})_{GPRLEN-16-2} || \text{offset} \; || \; 0^2 \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \text{PC} \leftarrow \text{PC} + \text{target\_offset} \\
& \quad \text{else} \\
& \quad \text{NullifyCurrentInstruction()} \\
& \quad \text{endif}
\end{align*}
\]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**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 this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BC2T instruction instead.
### Branch on Equal

**BEQ**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td>rs</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** BEQ rs, rt, offset

**MIPS32**

**Purpose:** Branch on Equal

To compare GPRs then do a PC-relative conditional branch

**Description:** if GPR[rs] = GPR[rt] then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs and GPR rt are equal, branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

- **I:** target_offset ← sign_extend(offset || 0²)
  condition ← (GPR[rs] = GPR[rt])

- **I+1:** if condition then
  PC ← PC + target_offset
  endif

**Exceptions:**

None

**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 this range.

BEQ r0, r0 offset, expressed as B offset, is the assembly idiom used to denote an unconditional branch.
Branch on Equal Likely

**Format:** BEQL rs, rt, offset

**Purpose:** Branch on Equal Likely
To compare GPRs then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

**Description:**
if GPR[rs] = GPR[rt] then branch
likely
An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.
If the contents of GPR rs and GPR rt are equal, branch to the target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

\[
\begin{align*}
&I: \quad \text{target\_offset} \leftarrow \text{sign\_extend}(\text{offset} || 0^2) \\
&\quad \text{condition} \leftarrow (\text{GPR}[rs] = \text{GPR}[rt]) \\
&I+1: \quad \text{if condition then} \\
&\quad \quad \text{PC} \leftarrow \text{PC} + \text{target\_offset} \\
&\quad \quad \text{else} \\
&\quad \quad \quad \text{NullifyCurrentInstruction()} \\
&\quad \text{endif}
\end{align*}
\]

**Exceptions:**
None

**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 this range.
Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.
Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BEQ instruction instead.

**Historical Information:**
In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
Branch on Greater Than or Equal to Zero

**Format:** BGEZ rs, offset

**Purpose:** Branch on Greater Than or Equal to Zero
To test a GPR then do a PC-relative conditional branch

**Description:** if GPR[rs] ≥ 0 then branch
An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

I: target\_offset ← sign\_extend(offset || 0^2)
condition ← GPR[rs] ≥ 0\^GPRLEN

I+1: if condition then
    PC ← PC + target\_offset
endif

**Exceptions:**
None

**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 this range.
Branch on Greater Than or Equal to Zero and Link  

**Format:**  
\[
\text{BGEZAL } \text{rs, offset}
\]

**MIPS32**

**Purpose:** Branch on Greater Than or Equal to Zero and Link

To test a GPR then do a PC-relative conditional procedure call

**Description:** if \( \text{GPR[rs]} \geq 0 \) then procedure_call

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

An 18-bit signed offset (the 16-bit \( \text{offset} \) field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR \( \text{rs} \) are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

GPR 31 must not be used for the source register \( \text{rs} \), because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is **UNPREDICTABLE**. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

**Operation:**

\[
\begin{align*}
\text{I:} & \quad \text{target_offset} \leftarrow \text{sign_extend}(\text{offset} || 0^2) \\
& \quad \text{condition} \leftarrow \text{GPR[rs]} \geq 0^{\text{GPRLEN}} \\
& \quad \text{GPR[31]} \leftarrow \text{PC} + 8 \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \quad \text{PC} \leftarrow \text{PC} + \text{target_offset} \\
& \quad \quad \text{endif}
\end{align*}
\]

**Exceptions:**

None

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch range is \( \pm 128 \text{ KBytes} \). Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.

BGEZAL \( r0, \) offset, expressed as BAL offset, is the assembly idiom used to denote a PC-relative branch and link. BAL is used in a manner similar to JAL, but provides PC-relative addressing and a more limited target PC range.
Branch on Greater Than or Equal to Zero and Link Likely

**IBGEZALL**

**Format:**  
`IBGEZALL rs, offset`

**MIPS32**

**Purpose:**  
Branch on Greater Than or Equal to Zero and Link Likely

To test a GPR then do a PC-relative conditional procedure call; execute the delay slot only if the branch is taken.

**Description:**  
if GPR[rs] ≥ 0 then procedure_call_likely

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**

GPR 31 must not be used for the source register rs, because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is **UNPREDICTABLE**. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:   target_offset ← sign_extend(offset || 02)
     condition ← GPR[rs] ≥ 0^GPRLEN
     GPR[31] ← PC + 8

I+1: if condition then
       PC ← PC + target_offset
    else
       NullifyCurrentInstruction()
    endif
```

**Exceptions:**

None

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch range is ±128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BGEZAL instruction instead.
**Branch on Greater Than or Equal to Zero and Link Likely**

**BGEZALL**

**Historical Information:**

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
Branch on Greater Than or Equal to Zero Likely

**Format:**  
BGEZL rs, offset

**Purpose:**  
Branch on Greater Than or Equal to Zero Likely  
To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

**Description:**  
if GPR[rs] ≥ 0 then branch likely  
An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.  
If the contents of GPR rs are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**  
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**  
I:  
target_offset ← sign_extend(offset || 0^2)  
condition ← GPR[rs] ≥ 0
I+1:  
if condition then  
PC ← PC + target_offset  
else  
NullifyCurrentInstruction()  
endif

**Exceptions:**  
None

**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 this range.  
Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.  
Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BGEZ instruction instead.

**Historical Information:**  
In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
### Branch on Greater Than Zero (BGTZ)

**Format:**  
BGTZ rs, offset

**Purpose:** Branch on Greater Than Zero  
To test a GPR then do a PC-relative conditional branch

**Description:**  
if GPR[rs] > 0 then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are greater than zero (sign bit is 0 but value not zero), branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**  
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

\[
\begin{align*}
\text{I:} & \quad \text{target\_offset} \leftarrow \text{sign\_extend}(\text{offset} \| 0^2) \\
& \quad \text{condition} \leftarrow \text{GPR[rs]} > 0^{\text{GPRLEN}} \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \quad \text{PC} \leftarrow \text{PC} + \text{target\_offset} \\
& \quad \quad \text{endif}
\end{align*}
\]

**Exceptions:**  
None

**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 this range.

---

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGTZ</td>
<td>rs</td>
<td>0</td>
<td>00000</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

31 26 25 21 20 16 15 0

BGTZ 000111 rs 0000 offset

MIPS32
Branch on Greater Than Zero Likely

**Format:** BGTZL rs, offset

**Purpose:** Branch on Greater Than Zero Likely

To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

**Description:** if GPR[rs] > 0 then branch_likely

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are greater than zero (sign bit is 0 but value not zero), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**
Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

\[
\begin{align*}
\text{I:} & \quad \text{target\_offset} \leftarrow \text{sign\_extend}(\text{offset} || 0^2) \\
& \quad \text{condition} \leftarrow \text{GPR}[rs] > 0^{\text{GPRLEN}} \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \quad \text{PC} \leftarrow \text{PC} + \text{target\_offset} \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{NullifyCurrentInstruction()} \\
& \quad \quad \text{endif}
\end{align*}
\]

**Exceptions:**
None

**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 this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BGTZ instruction instead.

**Historical Information:**
In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
Branch on Less Than or Equal to Zero

**Format:** \( \text{BLEZ \, rs, \, offset} \)

**Purpose:** Branch on Less Than or Equal to Zero
To test a GPR then do a PC-relative conditional branch

**Description:** if \( \text{GPR[rs]} \leq 0 \) then branch
An 18-bit signed offset (the 16-bit \( \text{offset} \) field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.
If the contents of GPR \( \text{rs} \) are less than or equal to zero (sign bit is 1 or value is zero), branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**
Processor operation is \textbf{UNPREDICTABLE} if a branch, jump, ERET, DERET, or \textit{WAIT} instruction is placed in the delay slot of a branch or jump.

**Operation:**
\[
\begin{align*}
\text{I:} & \quad \text{target_offset} \leftarrow \text{sign_extend(offset \, || \, 0^2)} \\
& \quad \text{condition} \leftarrow \text{GPR[rs]} \leq \text{0^GPRLEN} \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \quad \quad \text{PC} \leftarrow \text{PC} + \text{target_offset} \\
& \quad \quad \quad \text{endif}
\end{align*}
\]

**Exceptions:**
None

**Programming Notes:**
With the 18-bit signed instruction offset, the conditional branch range is \( \pm 128 \text{ KBytes} \). Use jump (\( \text{J} \)) or jump register (\( \text{JR} \)) instructions to branch to addresses outside this range.
Branch on Less Than or Equal to Zero Likely

Format: BLEZL rs, offset

Purpose: Branch on Less Than or Equal to Zero Likely
To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if GPR[rs] ≤ 0 then branch_likely
An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.
If the contents of GPR rs are less than or equal to zero (sign bit is 1 or value is zero), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:
Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:
I:  target_offset ← sign_extend(offset || 02)
    condition ← GPR[rs] ≤ 0GPRLEN
I+1: if condition then
     PC ← PC + target_offset
     else
     NullifyCurrentInstruction()
     endif

Exceptions:
None

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 this range.
Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.
Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BLEZ instruction instead.

Historical Information:
In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
### Branch on Less Than Zero (BLTZ)

**Format:** `BLTZ rs, offset`

**Purpose:** Branch on Less Than Zero

To test a GPR then do a PC-relative conditional branch

**Description:**

If the contents of GPR `rs` are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

1. `target_offset ← sign_extend(offset || 02)`
2. `condition ← GPR[rs] < 0^GPRLEN`
3. If `condition` then
   1. `PC ← PC + target_offset`
   2. **endif**

**Exceptions:**

None

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch range is ±128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.
Branch on Less Than Zero and Link

**Format:** BLTZAL rs, offset

**MIPS32**

**Purpose:** Branch on Less Than Zero and Link
To test a GPR then do a PC-relative conditional procedure call

**Description:** if GPR[rs] < 0 then procedure_call
Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**
GPR 31 must not be used for the source register rs, because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is UNPREDICTABLE. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

| I: | target_offset ← sign_extend(offset || 02) | condition ← GPR[rs] < 0GPRLEN | GPR[31] ← PC + 8 |
| I+1: | if condition then | PC ← PC + target_offset | endif |

**Exceptions:**
None

**Programming Notes:**
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.
Branch on Less Than Zero and Link Likely

**Format:** \texttt{BLTZALL rs, offset}

**Purpose:** Branch on Less Than Zero and Link Likely

To test a GPR then do a PC-relative conditional procedure call; execute the delay slot only if the branch is taken.

**Description:** if \( \text{GPR}[rs] < 0 \) then \text{procedure_call_likely}

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR \( rs \) are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**

GPR 31 must not be used for the source register \( rs \), because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is \text{UNPREDICTABLE}. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

Processor operation is \text{UNPREDICTABLE} if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

\[
\begin{align*}
\text{I:} & \quad \text{target\_offset} \leftarrow \text{sign\_extend}(\text{offset} || 0^2) \\
& \quad \text{condition} \leftarrow \text{GPR}[rs] < 0^{\text{GPR\_LEN}} \\
& \quad \text{GPR}[31] \leftarrow \text{PC} + 8 \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \quad \text{PC} \leftarrow \text{PC} + \text{target\_offset} \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{NullifyCurrentInstruction()} \\
& \quad \quad \text{endif}
\end{align*}
\]

**Exceptions:**

None

**Programming Notes:**

With the 18-bit signed instruction offset, the conditional branch range is \( \pm 128 \text{ KBytes} \). Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BLTZAL instruction instead.
Historical Information:

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
Branch on Less Than Zero Likely

Format: \texttt{BLTZL rs, offset}

\noindent \textbf{Purpose:} Branch on Less Than Zero Likely

To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

\noindent \textbf{Description:} if GPR[rs] < 0 then branch\_likely

An 18-bit signed offset (the 16-bit \textit{offset} field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

\noindent \textbf{Restrictions:}

Processor operation is \textbf{UNPREDICTABLE} if a branch, jump, ERET, DERET, or \texttt{WAIT} instruction is placed in the delay slot of a branch or jump.

\noindent \textbf{Operation:}

\begin{verbatim}
I:    target_offset ← sign_extend(offset || 02)
      condition ← GPR[rs] < 0^GPRLEN
I+1:  if condition then
       PC ← PC + target_offset
      else
       NullifyCurrentInstruction()
      endif
\end{verbatim}

\noindent \textbf{Exceptions:}

None

\noindent \textbf{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 this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98\% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BLTZ instruction instead.

\noindent \textbf{Historical Information:}

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
Branch on Not Equal

**Format:**  BNE rs, rt, offset

**Purpose:** Branch on Not Equal
To compare GPRs then do a PC-relative conditional branch

**Description:**  \( \text{if GPR}[rs] \neq \text{GPR}[rt] \) then branch
An 18-bit signed offset (the 16-bit \( \text{offset} \) field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.
If the contents of GPR \( rs \) and GPR \( rt \) are not equal, branch to the effective target address after the instruction in the delay slot is executed.

**Restrictions:**
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

\[
\begin{aligned}
\text{I:} & \quad \text{target\_offset} \leftarrow \text{sign\_extend}(\text{offset} || 0^2) \\
& \quad \text{condition} \leftarrow (\text{GPR}[rs] \neq \text{GPR}[rt]) \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \text{PC} \leftarrow \text{PC} + \text{target\_offset} \\
& \quad \text{endif}
\end{aligned}
\]

**Exceptions:**
None

**Programming Notes:**
With the 18-bit signed instruction offset, the conditional branch range is \( \pm 128 \text{ KBytes} \). Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.
Branch on Not Equal Likely

**Format:**  
BNEL rs, rt, offset

**Purpose:** Branch on Not Equal Likely  
To compare GPRs then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

**Description:** if GPR[rs] ≠ GPR[rt] then branch_likely

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs and GPR rt are not equal, branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

**Restrictions:**
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

```
I:  target_offset ← sign_extend(offset || 02)
    condition ← (GPR[rs] ≠ GPR[rt])

I+1: if condition then
     PC ← PC + target_offset
    else
     NullifyCurrentInstruction()
    endif
```

**Exceptions:**
None

**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 this range.

Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

Some implementations always predict the branch will be taken, so there is a significant penalty if the branch is not taken. Software should only use this instruction when there is a very high probability (98% or more) that the branch will be taken. If the branch is not likely to be taken or if the probability of a taken branch is unknown, software is encouraged to use the BNE instruction instead.

**Historical Information:**

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
Breakpoint

**Format:** `BREAK`  
**MIPS32**

**Purpose:** Breakpoint  
To cause a Breakpoint exception

**Description:**  
A breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler. The `code` field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

**Restrictions:**  
None

**Operation:**  
`SignalException(Breakpoint)`

**Exceptions:**  
Breakpoint
Floating Point Compare

C.cond.fmt

Format:

<table>
<thead>
<tr>
<th>Format:</th>
<th>C.cond.fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.cond.S fs, ft (cc = 0 implied)</td>
<td>MIPS32</td>
</tr>
<tr>
<td>C.cond.D fs, ft (cc = 0 implied)</td>
<td>MIPS32</td>
</tr>
<tr>
<td>C.cond.PS fs, ft (cc = 0 implied)</td>
<td>MIPS64, MIPS32 Release 2</td>
</tr>
<tr>
<td>C.cond.S cc, fs, ft</td>
<td>MIPS64, MIPS32 Release 2</td>
</tr>
<tr>
<td>C.cond.D cc, fs, ft</td>
<td>MIPS32</td>
</tr>
<tr>
<td>C.cond.PS cc, fs, ft</td>
<td>MIPS32</td>
</tr>
</tbody>
</table>

Purpose: Floating Point Compare

To compare FP values and record the Boolean result in a condition code

Description: FPConditionCode(cc) ← FPR[fs] compare_cond FPR[ft]

The value in FPR fs is compared to the value in FPR ft; the values are in format fmt. The comparison is exact and neither overflows nor underflows.

If the comparison specified by the cond field of the instruction is true for the operand values, the result is true; otherwise, the result is false. If no exception is taken, the result is written into condition code CC; true is 1 and false is 0.

In the cond field of the instruction: cond2..1 specify the nature of the comparison (equals, less than, and so on); cond0 specifies whether the comparison is ordered or unordered, i.e. false or true if any operand is a NaN; cond3 indicates whether the instruction should signal an exception on QNaN inputs, or not (see Table 3.26).

c.cond.PS compares the upper and lower halves of FPR fs and FPR ft independently and writes the results into condition codes CC +1 and CC respectively. The CC number must be even. If the number is not even the operation of the instruction is UNPREDICTABLE.

If one of the values is an SNaN, or cond3 is set and at least one of the values is a QNaN, an Invalid Operation condition is raised and the Invalid Operation flag is set in the FCSR. If the Invalid Operation Enable bit is set in the FCSR, no result is written and an Invalid Operation exception is taken immediately. Otherwise, the Boolean result is written into condition code CC.

There are four mutually exclusive ordering relations for comparing floating point values; one relation is always true and the others are false. The familiar relations are greater than, less than, and equal. In addition, the IEEE floating point standard defines the relation unordered, which is true when at least one operand value is NaN; NaN compares unordered with everything, including itself. Comparisons ignore the sign of zero, so +0 equals -0.

The comparison condition is a logical predicate, or equation, of the ordering relations such as less than or equal, equal, not less than, or unordered or equal. Compare distinguishes among the 16 comparison predicates. The Boolean result of the instruction is obtained by substituting the Boolean value of each ordering relation for the two FP values in the equation. If the equal relation is true, for example, then all four example predicates above yield a true result. If the unordered relation is true then only the final predicate, unordered or equal, yields a true result.

Logical negation of a compare result allows eight distinct comparisons to test for the 16 predicates as shown in Table 3.25. Each mnemonic tests for both a predicate and its logical negation. For each mnemonic, compare tests the truth of the first predicate. When the first predicate is true, the result is true as shown in the “If Predicate Is True” column, and the second predicate must be false, and vice versa. (Note that the False predicate is never true and False/True do not follow the normal pattern.)

The truth of the second predicate is the logical negation of the instruction result. After a compare instruction, test for the truth of the first predicate can be made with the Branch on FP True (BC1T) instruction and the truth of the second
can be made with Branch on FP False (BC1F).

Table 3.26 shows another set of eight compare operations, distinguished by a $cond_3$ value of 1 and testing the same 16 conditions. For these additional comparisons, if at least one of the operands is a NaN, including Quiet NaN, then an Invalid Operation condition is raised. If the Invalid Operation condition is enabled in the FCSR, an Invalid Operation exception occurs.
<table>
<thead>
<tr>
<th>Cond Mnemonic</th>
<th>Name of Predicate and Logically Negated Predicate (Abbreviation)</th>
<th>Comparison Predicate</th>
<th>Relation Values</th>
<th>Inv Op Excp. if QNaN?</th>
<th>Condition Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>False [this predicate is always False]</td>
<td></td>
<td>F</td>
<td>No</td>
<td>0 0</td>
</tr>
<tr>
<td></td>
<td>True (T)</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UN</td>
<td>Unordered</td>
<td></td>
<td>F</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Ordered (OR)</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EQ</td>
<td>Equal</td>
<td></td>
<td>F</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Not Equal (NEQ)</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UEQ</td>
<td>Unordered or Equal</td>
<td></td>
<td>F</td>
<td>F</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Ordered or Greater Than or Less Than (OGL)</td>
<td></td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>OLT</td>
<td>Ordered or Less Than</td>
<td></td>
<td>F</td>
<td>F</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Unordered or Greater Than or Equal (UGE)</td>
<td></td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>ULT</td>
<td>Unordered or Less Than</td>
<td></td>
<td>F</td>
<td>F</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Ordered or Greater Than or Equal (OGE)</td>
<td></td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>OLE</td>
<td>Ordered or Less Than or Equal</td>
<td></td>
<td>F</td>
<td>T</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Unordered or Greater Than (UGT)</td>
<td></td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>ULE</td>
<td>Unordered or Less Than or Equal</td>
<td></td>
<td>F</td>
<td>T</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Ordered or Greater Than (OGT)</td>
<td></td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

Key: ? = unordered, > = greater than, < = less than, = is equal, T = True, F = False
Table 3.26 FPU Comparisons With Special Operand Exceptions for QNaNs

<table>
<thead>
<tr>
<th>Cond Mnemonic</th>
<th>Name of Predicate and Logically Negated Predicate (Abbreviation)</th>
<th>Relation Values</th>
<th>Comparison CC Result</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>Signaling False [this predicate always False]</td>
<td>F F F F</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Signaling True (ST)</td>
<td>T T T T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NGLE</td>
<td>Not Greater Than or Less Than or Equal</td>
<td>F F F T</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Greater Than or Less Than (GLE)</td>
<td>T T T F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEQ</td>
<td>Signaling Equal</td>
<td>F F T F</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Signaling Not Equal (SNE)</td>
<td>T T F T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NGL</td>
<td>Not Greater Than or Less Than</td>
<td>F F T T</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Greater Than or Less Than (GL)</td>
<td>T T F F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LT</td>
<td>Less Than</td>
<td>F T F F</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Not Less Than (NLT)</td>
<td>T F T T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NGE</td>
<td>Not Greater Than or Equal</td>
<td>F T F T</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Greater Than or Equal (GE)</td>
<td>T F T F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LE</td>
<td>Less Than</td>
<td>F T T F</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Not Less Than or Equal (NLE)</td>
<td>T F T F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NGT</td>
<td>Not Greater Than</td>
<td>F T T T</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Greater Than (GT)</td>
<td>T F T F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Key: ? = unordered, > = greater than, < = less than, = is equal, T = True, F = False

Restrictions:
The fields fs and ft must specify FPRs valid for operands of type fmt; if they are not valid, the result is UNPREDICTABLE.

<table>
<thead>
<tr>
<th>Operation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>if SNaN(ValueFPR(fs, fmt)) or SNaN(ValueFPR(ft, fmt)) or QNaN(ValueFPR(fs, fmt)) or QNaN(ValueFPR(ft, fmt)) then</td>
</tr>
<tr>
<td>less ← false</td>
</tr>
<tr>
<td>equal ← false</td>
</tr>
<tr>
<td>unordered ← true</td>
</tr>
<tr>
<td>endif</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>less ← ValueFPR(fs, fmt) &lt;fmt ValueFPR(ft, fmt)</td>
</tr>
<tr>
<td>equal ← ValueFPR(fs, fmt) =fmt ValueFPR(ft, fmt)</td>
</tr>
</tbody>
</table>

The operands must be values in format fmt; if they are not, the result is UNPREDICTABLE and the value of the operand FPRs becomes UNPREDICTABLE.

The result of C.cond.PS is UNPREDICTABLE if the processor is executing in the FR=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the FR=1 mode, but not with FR=0, and not on a 32-bit FPU.

The result of C.cond.PS is UNPREDICTABLE if the condition code number is odd.
unordered ← false
endif
condition ← (cond₂ and less) or (cond₁ and equal)
or (cond₀ and unordered)
SetFPConditionCode(cc, condition)

For c.cond.PS, the pseudo code above is repeated for both halves of the operand registers, treating each half as an independent single-precision value. Exceptions on the two halves are logically ORed and reported together. The results of the lower half comparison are written to condition code CC; the results of the upper half comparison are written to condition code CC+1.

Exceptions:
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:
Unimplemented Operation, Invalid Operation

Programming Notes:
FP computational instructions, including compare, that receive an operand value of Signaling NaN raise the Invalid Operation condition. Comparisons that raise the Invalid Operation condition for Quiet NaNs in addition to SNaNs permit a simpler programming model if NaNs are errors. Using these compares, programs do not need explicit code to check for QNaNs causing the unordered relation. Instead, they take an exception and allow the exception handling system to deal with the error when it occurs. For example, consider a comparison in which we want to know if two numbers are equal, but for which unordered would be an error.

```assembly
# comparisons using explicit tests for QNaN
  c.eq.d $f2,$f4    # check for equal
  nop
  bclt  L2          # it is equal
  c.un.d $f2,$f4    # it is not equal,
                   # but might be unordered
  bclt  ERROR       # unordered goes off to an error handler
  # not-equal-case code here
...
# equal-case code here
L2:
  # --------------------------------------------------------------
  # comparison using comparisons that signal QNaN
  c.seq.d $f2,$f4   # check for equal
  nop
  bclt  L2          # it is equal
  nop
  # it is not unordered here
  ....
  # not-equal-case code here
  ...
  # equal-case code here
```
Perform Cache Operation

Format: CACHE op, offset(base)

Purpose: Perform Cache Operation
To perform the cache operation specified by op.

Description:
The 16-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used in one of the following ways based on the operation to be performed and the type of cache as described in the following table.

Table 3.27 Usage of Effective Address

<table>
<thead>
<tr>
<th>Operation Requires an</th>
<th>Type of Cache</th>
<th>Usage of Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Virtual</td>
<td>The effective address is used to address the cache. An address translation may or may not be performed on the effective address (with the possibility that a TLB Refill or TLB Invalid exception might occur)</td>
</tr>
<tr>
<td>Address</td>
<td>Physical</td>
<td>The effective address is translated by the MMU to a physical address. The physical address is then used to address the cache</td>
</tr>
</tbody>
</table>
| Index                 | N/A           | The effective address is translated by the MMU to a physical address. It is implementation dependent whether the effective address or the translated physical address is used to index the cache. As such, an unmapped address (such as within kseg0) should always be used for cache operations that require an index. See the Programming Notes section below. Assuming that the total cache size in bytes is CS, the associativity is A, and the number of bytes per tag is BPT, the following calculations give the fields of the address which specify the way and the index:

\[
\begin{align*}
\text{OffsetBit} & \leftarrow \log_2(\text{BPT}) \\
\text{IndexBit} & \leftarrow \log_2(\text{CS} / A) \\
\text{WayBit} & \leftarrow \text{IndexBit} + \text{Ceiling}(\log_2(A)) \\
\text{Way} & \leftarrow \text{Addr}_{\text{WayBit-1..IndexBit}} \\
\text{Index} & \leftarrow \text{Addr}_{\text{IndexBit-1..offsetBit}}
\end{align*}
\]

For a direct-mapped cache, the Way calculation is ignored and the Index value fully specifies the cache tag. This is shown symbolically in the figure below.

Figure 3.2 Usage of Address Fields to Select Index and Way

A TLB Refill and TLB Invalid (both with cause code equal TLBL) exception can occur on any operation. For index
operations (where the address is used to index the cache but need not match the cache tag) software should use unmapped addresses to avoid TLB exceptions. This instruction never causes TLB Modified exceptions nor TLB Refill exceptions with a cause code of TLBS. This instruction never causes Execute-Inhibit nor Read-Inhibit exceptions.

The effective address may be an arbitrarily-aligned by address. The CACHE instruction never causes an Address Error Exception due to an non-aligned address.

A Cache Error exception may occur as a by-product of some operations performed by this instruction. For example, if a Writeback operation detects a cache or bus error during the processing of the operation, that error is reported via a Cache Error exception. Similarly, a Bus Error Exception may occur if a bus operation invoked by this instruction is terminated in an error. However, cache error exceptions must not be triggered by an Index Load Tag or Index Store tag operation, as these operations are used for initialization and diagnostic purposes.

An Address Error Exception (with cause code equal AdEL) may occur if the effective address references a portion of the kernel address space which would normally result in such an exception. It is implementation dependent whether such an exception does occur.

It is implementation dependent whether a data watch is triggered by a cache instruction whose address matches the Watch register address match conditions.

The CACHE instruction and the memory transactions which are sourced by the CACHE instruction, such as cache refill or cache writeback, obey the ordering and completion rules of the SYNC instruction.

Bits [17:16] of the instruction specify the cache on which to perform the operation, as follows:

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>I</td>
<td>Primary Instruction</td>
</tr>
<tr>
<td>0b01</td>
<td>D</td>
<td>Primary Data or Unified Primary</td>
</tr>
<tr>
<td>0b10</td>
<td>T</td>
<td>Tertiary</td>
</tr>
<tr>
<td>0b11</td>
<td>S</td>
<td>Secondary</td>
</tr>
</tbody>
</table>

Bits [20:18] of the instruction specify the operation to perform. To provide software with a consistent base of cache operations, certain encodings must be supported on all processors. The remaining encodings are recommended for implementations which implement multiple level of caches and where the hardware maintains the smaller cache as a proper subset of a larger cache (every address which is resident in the smaller cache is also resident in the larger cache; also known as the inclusion property), it is recommended that the CACHE instructions which operate on the larger, outer-level cache; should first operate on the smaller, inner-level cache. For example, a Hit_Writeback _Invalidate operation targeting the Secondary cache, should first operate on the primary data cache first. If the CACHE instruction implementation does not follow this policy then any software which flushes the caches must mimic this behavior. That is, the software sequences must first operate on the inner cache then operate on the outer cache. The software must place a SYNC instruction after the CACHE instruction whenever there are possible writebacks from the inner cache to ensure that the writeback data is resident in the outer cache before operating on the outer cache. If neither the CACHE instruction implementation nor the software cache flush sequence follow this policy, then the inclusion property of the caches can be broken, which might be a condition that the cache management hardware cannot properly deal with.

For implementations which implement multiple level of caches without the inclusion property, the use of a SYNC instruction after the CACHE instruction is still needed whenever writeback data has to be resident in the next level of memory hierarchy.
For multiprocessor implementations that maintain coherent caches, some of the Hit type of CACHE instruction operations may optionally affect all coherent caches within the implementation. If the effective address uses a coherent Cache Coherency Attribute (CCA), then the operation is *globalized*, meaning it is broadcast to all of the coherent caches within the system. If the effective address does not use one of the coherent CCAs, there is no broadcast of the operation. If multiple levels of caches are to be affected by one CACHE instruction, all of the affected cache levels must be processed in the same manner - either all affected cache levels use the globalized behavior or all affected cache levels use the non-globalized behavior.

### Table 3.29 Encoding of Bits [20:18] of the CACHE Instruction

<table>
<thead>
<tr>
<th>Code</th>
<th>Caches</th>
<th>Name</th>
<th>Effective Address Operand Type</th>
<th>Operation</th>
<th>Compliance Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000</td>
<td>I</td>
<td>Index Invalidate</td>
<td>Index</td>
<td>Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire instruction cache by stepping through all valid indices.</td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Index Writeback Invalidate / Index Invalidate</td>
<td>Index</td>
<td>For a write-back cache: If the state of the cache block at the specified index is valid and dirty, write the block back to the memory address specified by the cache tag. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid.</td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>S, T</td>
<td>Index Writeback Invalidate / Index Invalidate</td>
<td>Index</td>
<td>For a write-through cache: Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire data cache by stepping through all valid indices. Note that Index Store Tag should be used to initialize the cache at power up. Required if S, T cache is implemented</td>
<td></td>
</tr>
<tr>
<td>0b001</td>
<td>All</td>
<td>Index Load Tag</td>
<td>Index</td>
<td>Read the tag for the cache block at the specified index into the TagLo and TagHi Coprocessor 0 registers. If the DataLo and DataHi registers are implemented, also read the data corresponding to the byte index into the DataLo and DataHi registers. This operation must not cause a Cache Error Exception. The granularity and alignment of the data read into the DataLo and DataHi registers is implementation-dependent, but is typically the result of an aligned access to the cache, ignoring the appropriate low-order bits of the byte index.</td>
<td>Recommended</td>
</tr>
</tbody>
</table>
### Table 3.29 Encoding of Bits [20:18] of the CACHE Instruction (Continued)

<table>
<thead>
<tr>
<th>Code</th>
<th>Caches</th>
<th>Name</th>
<th>Operation</th>
<th>Compliance Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b010</td>
<td>All</td>
<td>Index Store Tag</td>
<td>Write the tag for the cache block at the specified index from the TagLo and TagHi Coprocessor 0 registers. This operation must not cause a Cache Error Exception. This required encoding may be used by software to initialize the entire instruction or data caches by stepping through all valid indices. Doing so requires that the TagLo and TagHi registers associated with the cache be initialized first.</td>
<td>Required</td>
</tr>
<tr>
<td>0b011</td>
<td>All</td>
<td>Implementation Dependent</td>
<td>Available for implementation-dependent operation.</td>
<td>Optional</td>
</tr>
<tr>
<td>0b100</td>
<td>I, D</td>
<td>Hit Invalidate</td>
<td>If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the instruction cache by stepping through the address range by the line size of the cache. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.</td>
<td>Required (Instruction Cache Encoding Only), Recommended otherwise</td>
</tr>
<tr>
<td></td>
<td>S, T</td>
<td>Hit Invalidate</td>
<td>Address</td>
<td>Optional, if Hit_Invalidate_D is implemented, the S and T variants are recommended.</td>
</tr>
<tr>
<td>0b101</td>
<td>I</td>
<td>Fill</td>
<td>Fill the cache from the specified address.</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Hit Writeback Invalidate / Hit Invalidate</td>
<td>For a write-back cache: If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After that operation is completed, set the state of the cache block to invalid. For a write-through cache: If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the data cache by stepping through the address range by the line size of the cache. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.</td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>S, T</td>
<td>Hit Writeback Invalidate / Hit Invalidate</td>
<td>Address</td>
<td>Required if S, T cache is implemented</td>
</tr>
</tbody>
</table>
### Perform Cache Operation

The operation of this instruction is **UNDEFINED** for any operation/cache combination that is not implemented.

---

**Table 3.29 Encoding of Bits [20:18] of the CACHE Instruction (Continued)**

<table>
<thead>
<tr>
<th>Code</th>
<th>Caches</th>
<th>Name</th>
<th>Effective Address Operand Type</th>
<th>Operation</th>
<th>Compliance Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b110</td>
<td>D</td>
<td>Hit Writeback</td>
<td>Address</td>
<td>If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After the operation is completed, leave the state of the line valid, but clear the dirty state. For a write-through cache, this operation may be treated as a nop. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td>S, T</td>
<td>Hit Writeback</td>
<td>Address</td>
<td></td>
<td>Optional, if Hit_Writeback_D is implemented, the S and T variants are recommended.</td>
</tr>
<tr>
<td>0b111</td>
<td>I, D</td>
<td>Fetch and Lock</td>
<td>Address</td>
<td>If the cache does not contain the specified address, fill it from memory, performing a writeback if required, and set the state to valid and locked. If the cache already contains the specified address, set the state to locked. In set-associative or fully-associative caches, the way selected on a fill from memory is implementation dependent. The lock state may be cleared by executing an Index Invalidate, Index Writeback Invalidate, Hit Invalidate, or Hit Writeback Invalidate operation to the locked line, or via an Index Store Tag operation to the line that clears the lock bit. Note that clearing the lock state via Index Store Tag is dependent on the implementation-dependent cache tag and cache line organization, and that Index and Index Writeback Invalidate operations are dependent on cache line organization. Only Hit and Hit Writeback Invalidate operations are generally portable across implementations. It is implementation dependent whether a locked line is displaced as the result of an external invalidate or intervention that hits on the locked line. Software must not depend on the locked line remaining in the cache if an external invalidate or intervention would invalidate the line if it were not locked. It is implementation dependent whether a Fetch and Lock operation affects more than one line. For example, more than one line around the referenced address may be fetched and locked. It is recommended that only the single line containing the referenced address be affected.</td>
<td>Recommended</td>
</tr>
</tbody>
</table>
The operation of this instruction is **UNDEFINED** if the operation requires an address, and that address is uncacheable.

The operation of the instruction is **UNPREDICTABLE** if the cache line that contains the CACHE instruction is the target of an invalidate or a writeback invalidate.

If this instruction is used to lock all ways of a cache at a specific cache index, the behavior of that cache to subsequent cache misses to that cache index is **UNDEFINED**.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Any use of this instruction that can cause cacheline writebacks should be followed by a subsequent SYNC instruction to avoid hazards where the writeback data is not yet visible at the next level of the memory hierarchy.

**Operation:**

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{GPR}\[\text{base}\] + \text{sign}_{-}\text{extend}(\text{offset}) \\
(p\text{Addr}, \text{uncached}) & \leftarrow \text{AddressTranslation}(\text{vAddr}, \text{DataReadReference}) \\
\text{CacheOp}(\text{op}, \text{vAddr}, \text{pAddr}) &
\end{align*}
\]

**Exceptions:**

- TLB Refill Exception.
- TLB Invalid Exception
- Coprocessor Unusable Exception
- Address Error Exception
- Cache Error Exception
- Bus Error Exception

**Programming Notes:**

For cache operations that require an index, it is implementation dependent whether the effective address or the translated physical address is used as the cache index. Therefore, the index value should always be converted to an unmapped address (such as an kseg0 address - by ORing the index with 0x80000000 before being used by the cache instruction). For example, the following code sequence performs a data cache Index Store Tag operation using the index passed in GPR a0:

```assembly
li   a1, 0x80000000 /* Base of kseg0 segment */
or  a0, a0, al     /* Convert index to kseg0 address */
cache DCIndexStTag, 0(al) /* Perform the index store tag operation */
```
Perform Cache Operation EVA

```
  31  26  25  21  20  16  15  7  6  5  0
SPECIAL3 011111
  6  5  5  9  1  8
base op offset 0
```

**Format:** \texttt{CACHEE op, offset(base)} \quad \text{MIPS32}

**Purpose:** Perform Cache Operation EVA

To perform the cache operation specified by \texttt{op} using a user mode virtual address while in kernel mode.

**Description:**

The 9 bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used in one of the following ways based on the operation to be performed and the type of cache as described in the following table.

### Table 3.1 Usage of Effective Address

<table>
<thead>
<tr>
<th>Operation Requires an</th>
<th>Type of Cache</th>
<th>Usage of Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Virtual</td>
<td>The effective address is used to address the cache. An address translation may or may not be performed on the effective address (with the possibility that a TLB Refill or TLB Invalid exception might occur)</td>
</tr>
<tr>
<td>Address</td>
<td>Physical</td>
<td>The effective address is translated by the MMU to a physical address. The physical address is then used to address the cache</td>
</tr>
<tr>
<td>Index</td>
<td>N/A</td>
<td>The effective address is translated by the MMU to a physical address. It is implementation dependent whether the effective address or the translated physical address is used to index the cache. As such, a kseg0 address should always be used for cache operations that require an index. See the Programming Notes section below.</td>
</tr>
</tbody>
</table>

Assuming that the total cache size in bytes is CS, the associativity is A, and the number of bytes per tag is BPT, the following calculations give the fields of the address which specify the way and the index:

\[
\begin{align*}
\text{OffsetBit} & \leftarrow \log_2 (\text{BPT}) \\
\text{IndexBit} & \leftarrow \log_2 (\text{CS} / \text{A}) \\
\text{WayBit} & \leftarrow \text{IndexBit} + \text{Ceiling}(\log_2 (\text{A})) \\
\text{Way} & \leftarrow \text{Addr}_{\text{WayBit} - 1..\text{IndexBit}} \\
\text{Index} & \leftarrow \text{Addr}_{\text{IndexBit} - 1..\text{OffsetBit}}
\end{align*}
\]

For a direct-mapped cache, the Way calculation is ignored and the Index value fully specifies the cache tag. This is shown symbolically in the figure below.

**Figure 3.1 Usage of Address Fields to Select Index and Way**

```
  Unused    Way    Index    OffsetBit
              \mid \leftarrow \text{WayBit} \\
\mid \leftarrow \text{IndexBit} \\
\mid \leftarrow \text{OffsetBit}
```

A TLB Refill and TLB Invalid (both with cause code equal TLBL) exception can occur on any operation. For index
operations (where the address is used to index the cache but need not match the cache tag) software should use unmapped addresses to avoid TLB exceptions. This instruction never causes TLB Modified exceptions nor TLB Refill exceptions with a cause code of TLBS. This instruction never causes Execute-Inhibit nor Read-Inhibit exceptions.

The effective address may be an arbitrarily-aligned by address. The CACHEE instruction never causes an Address Error Exception due to a non-aligned address.

A Cache Error exception may occur as a by-product of some operations performed by this instruction. For example, if a Writeback operation detects a cache or bus error during the processing of the operation, that error is reported via a Cache Error exception. Similarly, a Bus Error Exception may occur if a bus operation invoked by this instruction is terminated in an error. However, cache error exceptions must not be triggered by an Index Load Tag or Index Store tag operation, as these operations are used for initialization and diagnostic purposes.

An Address Error Exception (with cause code equal AdEL) may occur if the effective address references a portion of the kernel address space which would normally result in such an exception. It is implementation dependent whether such an exception does occur.

It is implementation dependent whether a data watch is triggered by a cache instruction whose address matches the Watch register address match conditions.

The CACHEE instruction and the memory transactions which are sourced by the CACHEE instruction, such as cache refill or cache writeback, obey the ordering and completion rules of the SYNC instruction.

Bits [17:16] of the instruction specify the cache on which to perform the operation, as follows:

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>I</td>
<td>Primary Instruction</td>
</tr>
<tr>
<td>0b01</td>
<td>D</td>
<td>Primary Data or Unified Primary</td>
</tr>
<tr>
<td>0b10</td>
<td>T</td>
<td>Tertiary</td>
</tr>
<tr>
<td>0b11</td>
<td>S</td>
<td>Secondary</td>
</tr>
</tbody>
</table>

Bits [20:18] of the instruction specify the operation to perform. To provide software with a consistent base of cache operations, certain encodings must be supported on all processors. The remaining encodings are recommended.

For implementations which implement multiple level of caches and where the hardware maintains the smaller cache as a proper subset of a larger cache (every address which is resident in the smaller cache is also resident in the larger cache; also known as the inclusion property), it is recommended that the CACHEE instructions which operate on the larger, outer-level cache; should first operate on the smaller, inner-level cache. For example, a Hit_Writeback _Invalidate operation targeting the Secondary cache, should first operate on the primary data cache first. If the CACHEE instruction implementation does not follow this policy then any software which flushes the caches must mimic this behavior. That is, the software sequences must first operate on the inner cache then operate on the outer cache. The software must place a SYNC instruction after the CACHEE instruction whenever there are possible writebacks from the inner cache to ensure that the writeback data is resident in the outer cache before operating on the outer cache. If neither the CACHEE instruction implementation nor the software cache flush sequence follow this policy, then the inclusion property of the caches can be broken, which might be a condition that the cache management hardware cannot properly deal with.

For implementations which implement multiple level of caches without the inclusion property, the use of a SYNC instruction after the CACHEE instruction is still needed whenever writeback data has to be resident in the next level of memory hierarchy.
For multiprocessor implementations that maintain coherent caches, some of the Hit type of CACHEE instruction operations may optionally affect all coherent caches within the implementation. If the effective address uses a coherent Cache Coherency Attribute (CCA), then the operation is globalized, meaning it is broadcast to all of the coherent caches within the system. If the effective address does not use one of the coherent CCAs, there is no broadcast of the operation. If multiple levels of caches are to be affected by one CACHEE instruction, all of the affected cache levels must be processed in the same manner - either all affected cache levels use the globalized behavior or all affected cache levels use the non-globalized behavior.

The CACHEE instruction functions in exactly the same fashion as the CACHE instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the Config5EVA field being set to one.

### Table 3.3 Encoding of Bits [20:18] of the CACHEE Instruction

<table>
<thead>
<tr>
<th>Code</th>
<th>Caches</th>
<th>Name</th>
<th>Effective Address Operand Type</th>
<th>Operation</th>
<th>Compliance Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000</td>
<td>I</td>
<td>Index Invalidate</td>
<td>Index</td>
<td>Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire instruction cache by stepping through all valid indices.</td>
<td>Required</td>
</tr>
<tr>
<td>D</td>
<td>Index Writeback Invalidate / Index Invalidate</td>
<td>Index</td>
<td>For a write-back cache: If the state of the cache block at the specified index is valid and dirty, write the block back to the memory address specified by the cache tag. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid.</td>
<td>Required</td>
<td></td>
</tr>
<tr>
<td>S, T</td>
<td>Index Writeback Invalidate / Index Invalidate</td>
<td>Index</td>
<td>For a write-through cache: Set the state of the cache block at the specified index to invalid. This required encoding may be used by software to invalidate the entire data cache by stepping through all valid indices. Note that Index Store Tag should be used to initialize the cache at power up.</td>
<td>Required if S, T cache is implemented</td>
<td></td>
</tr>
</tbody>
</table>
Table 3.3 Encoding of Bits [20:18] of the CACHEE Instruction (Continued)

<table>
<thead>
<tr>
<th>Code</th>
<th>Caches</th>
<th>Name</th>
<th>Effective Address Operand Type</th>
<th>Operation</th>
<th>Compliance Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b001</td>
<td>All</td>
<td>Index Load Tag</td>
<td>Index</td>
<td>Read the tag for the cache block at the specified index into the TagLo and TagHi Coprocessor 0 registers. If the DataLo and DataHi registers are implemented, also read the data corresponding to the byte index into the DataLo and DataHi registers. This operation must not cause a Cache Error Exception. The granularity and alignment of the data read into the DataLo and DataHi registers is implementation-dependent, but is typically the result of an aligned access to the cache, ignoring the appropriate low-order bits of the byte index.</td>
<td>Recommended</td>
</tr>
<tr>
<td>0b010</td>
<td>All</td>
<td>Index Store Tag</td>
<td>Index</td>
<td>Write the tag for the cache block at the specified index from the TagLo and TagHi Coprocessor 0 registers. This operation must not cause a Cache Error Exception. This required encoding may be used by software to initialize the entire instruction or data caches by stepping through all valid indices. Doing so requires that the TagLo and TagHi registers associated with the cache be initialized first.</td>
<td>Required</td>
</tr>
<tr>
<td>0b011</td>
<td>All</td>
<td>Implementation Dependent</td>
<td>Unspecified</td>
<td>Available for implementation-dependent operation.</td>
<td>Optional</td>
</tr>
<tr>
<td>0b100</td>
<td>I, D</td>
<td>Hit Invalidate</td>
<td>Address</td>
<td>If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the instruction cache by stepping through the address range by the line size of the cache.</td>
<td>Required (Instruction Cache Encoding Only), Recommended otherwise</td>
</tr>
<tr>
<td></td>
<td>S, T</td>
<td>Hit Invalidate</td>
<td>Address</td>
<td>In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.</td>
<td>Optional, if Hit_Invalidate_D is implemented, the S and T variants are recommended.</td>
</tr>
</tbody>
</table>
Table 3.3 Encoding of Bits [20:18] of the CACHEE Instruction (Continued)

<table>
<thead>
<tr>
<th>Code</th>
<th>Caches</th>
<th>Name</th>
<th>Effective Address Operand Type</th>
<th>Operation</th>
<th>Compliance Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b101</td>
<td>I</td>
<td>Fill</td>
<td>Address</td>
<td>Fill the cache from the specified address.</td>
<td>Recommended</td>
</tr>
<tr>
<td>D</td>
<td>Hit Writeback Invalidate / Hit Invalidate</td>
<td>Address</td>
<td>For a write-back cache: If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid. For a write-through cache: If the cache block contains the specified address, set the state of the cache block to invalid. This required encoding may be used by software to invalidate a range of addresses from the data cache by stepping through the address range by the line size of the cache. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.</td>
<td>Required</td>
<td></td>
</tr>
<tr>
<td>S, T</td>
<td>Hit Writeback Invalidate / Hit Invalidate</td>
<td>Address</td>
<td>If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After the operation is completed, leave the state of the line valid, but clear the dirty state. For a write-through cache, this operation may be treated as a nop. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.</td>
<td>Required if S, T cache is implemented</td>
<td></td>
</tr>
<tr>
<td>0b110</td>
<td>D</td>
<td>Hit Writeback</td>
<td>Address</td>
<td>If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After the operation is completed, leave the state of the line valid, but clear the dirty state. For a write-through cache, this operation may be treated as a nop. In multiprocessor implementations with coherent caches, the operation may optionally be broadcast to all coherent caches within the system.</td>
<td>Recommended</td>
</tr>
<tr>
<td>S, T</td>
<td>Hit Writeback</td>
<td>Address</td>
<td>Optional, if Hit_Writeback_D is implemented, the S and T variants are recommended.</td>
<td>Recommended</td>
<td></td>
</tr>
</tbody>
</table>

MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set, Revision 5.04
Restrictions:

The operation of this instruction is **UNDEFINED** for any operation-cache combination that is not implemented.

The operation of this instruction is **UNDEFINED** if the operation requires an address, and that address is uncacheable.

The operation of the instruction is **UNPREDICTABLE** if the cache line that contains the CACHEE instruction is the target of an invalidate or a writeback invalidate.

If this instruction is used to lock all ways of a cache at a specific cache index, the behavior of that cache to subsequent cache misses to that cache index is **UNDEFINED**.

Any use of this instruction that can cause cacheline writebacks should be followed by a subsequent SYNC instruction to avoid hazards where the writeback data is not yet visible at the next level of the memory hierarchy.

Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

---

**Table 3.3 Encoding of Bits [20:18] of the CACHEE Instruction (Continued)**

<table>
<thead>
<tr>
<th>Code</th>
<th>Caches</th>
<th>Name</th>
<th>Effective Address Operand Type</th>
<th>Operation</th>
<th>Compliance Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b111</td>
<td>I, D</td>
<td>Fetch and Lock</td>
<td>Address</td>
<td>If the cache does not contain the specified address, fill it from memory, performing a writeback if required, and set the state to valid and locked. If the cache already contains the specified address, set the state to locked. In set-associative or fully-associative caches, the way selected on a fill from memory is implementation dependent. The lock state may be cleared by executing an Index Invalidate, Index Writeback Invalidate, Hit Invalidate, or Hit Writeback Invalidate operation to the locked line, or via an Index Store Tag operation to the line that clears the lock bit. Note that clearing the lock state via Index Store Tag is dependent on the implementation-dependent cache tag and cache line organization, and that Index and Index Writeback Invalidate operations are dependent on cache line organization. Only Hit and Hit Writeback Invalidate operations are generally portable across implementations. It is implementation dependent whether a locked line is displaced as the result of an external invalidate or intervention that hits on the locked line. Software must not depend on the locked line remaining in the cache if an external invalidate or intervention would invalidate the line if it were not locked. It is implementation dependent whether a Fetch and Lock operation affects more than one line. For example, more than one line around the referenced address may be fetched and locked. It is recommended that only the single line containing the referenced address be affected.</td>
<td>Recommended</td>
</tr>
</tbody>
</table>
**Operation:**

\[
\text{vAddr} \leftarrow \text{GPR[base]} + \text{sign}\_\text{extend}\text{(offset)} \\
(p\text{Addr, uncached}) \leftarrow \text{AddressTranslation(vAddr, DataReadReference)} \\
\text{CacheOp(op, vAddr, p\text{Addr})}
\]

**Exceptions:**
- TLB Refill Exception.
- TLB Invalid Exception
- Coprocessor Unusable Exception
- Reserved Instruction
- Address Error Exception
- Cache Error Exception
- Bus Error Exception

**Programming Notes:**

For cache operations that require an index, it is implementation dependent whether the effective address or the translated physical address is used as the cache index. Therefore, the index value should always be converted to a kseg0 address by ORing the index with 0x80000000 before being used by the cache instruction. For example, the following code sequence performs a data cache Index Store Tag operation using the index passed in GPR a0:

```assembly
li a1, 0x80000000 /* Base of kseg0 segment */
or a0, a0, a1 /* Convert index to kseg0 address */
cache DCIndexStTag, 0(a1) /* Perform the index store tag operation */
```
Fixed Point Ceiling Convert to Long Fixed Point

**Purpose:** Fixed Point Ceiling Convert to Long Fixed Point
To convert an FP value to 64-bit fixed point, rounding up

**Description:** FPR[fd] ← convert_and_round(FPR[fs])
The value in FPR fs, in format fmt, is converted to a value in 64-bit long fixed point format and rounding toward +∞ (rounding mode 2). The result is placed in FPR fd.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^63 to 2^63-1, the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the FCSR. If the Invalid Operation Enable bit is set in the FCSR, no result is written to fd and an Invalid Operation exception is taken immediately. Otherwise, the default result, 2^63–1, is written to fd.

**Restrictions:**
The fields fs and fd must specify valid FPRs; fs for type fmt and fd for long fixed point; if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

The result of this instruction is UNPREDICTABLE if the processor is executing in the FR=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the FR=1 mode, but not with FR=0, and not on a 32-bit FPU.

**Operation:**

\[
\text{StoreFPR}(fd, L, \text{ConvertFmt(Val}\text{ueFPR}(fs, fmt), fmt, L))
\]

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Invalid Operation, Unimplemented Operation, Inexact
Floating Point Ceiling Convert to Word Fixed Point

Format:

CEIL.W.fmt

CEIL.W.S   fd, fs  

CEIL.W.D   fd, fs  

Purpose: Floating Point Ceiling Convert to Word Fixed Point

To convert an FP value to 32-bit fixed point, rounding up

Description:

FPR[fd] ← convert_and_round(FPR[fs])

The value in FPR fs, in format fmt, is converted to a value in 32-bit word fixed point format and rounding toward +∞ (rounding mode 2). The result is placed in FPR fd.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^31 to 2^31-1, the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the FCSR. If the Invalid Operation Enable bit is set in the FCSR, no result is written to fd and an Invalid Operation exception is taken immediately. Otherwise, the default result, 2^31–1, is written to fd.

Restrictions:

The fields fs and fd must specify valid FPRs; fs for type fmt and fd for word fixed point; if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

Operation:

\[
\text{StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))}
\]

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact
Move Control Word From Floating Point

CFC1

**Format:**

CFC1 rt, fs

**Purpose:** Move Control Word From Floating Point

To copy a word from an FPU control register to a GPR

**Description:**

GPR[rt] ← FP_Control[fs]

Copy the 32-bit word from FP (coprocessor 1) control register fs into GPR rt.

The definition of this instruction has been extended in Release 5 to support user mode read of StatusFr under the control of Config5UFR. This required feature is meant to facilitate transition from FR=0 to FR=1 floating-point register modes in order to obsolete FR=0 mode.

**Restrictions:**

There are a few control registers defined for the floating point unit. The result is UNPREDICTABLE if fs specifies a register that does not exist.

In particular, the result is UNPREDICTABLE if fs specifies the UNFR write-only control. R5.03 implementations are required to produce a Reserved Instruction Exception; software must assume it is UNPREDICTABLE.

**Operation:**

if fs = 0 then
  temp ← FIR
elseif fs = 1 and FIRQ then /* read UFR (CP1 Register 1) */
  if Config5UFR
    temp ← StatusFr
  else
    signalException(RI)
  endif
/* note: fs=4 UNFR not supported for reading - UFR suffices */
elseif fs = 25 then /* FCCR */
  temp ← 024 || FCSR41..25 || FCSR23
elseif fs = 26 then /* FEXR */
  temp ← 014 || FCSR17..12 || 05 || FCSR6..2 || 02
elseif fs = 28 then /* FENR */
  temp ← 020 || FCSR11.7 || 04 || FCSR24 || FCSR1..0
elseif fs = 31 then /* FCSR */
  temp ← FCSR
else
  temp ← UNPREDICTABLE
endif
GPR[rt] ← temp

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Historical Information:**

For the MIPS I, II and III architectures, the contents of GPR rt are UNPREDICTABLE for the instruction immedi-
ate following CFC1.

MIPS V and MIPS32 introduced the three control registers that access portions of FCSR. These registers were not available in MIPS I, II, III, or IV.

MIPS32r5 introduced the UFR and UNFR register aliases that allow user level access to $Status_{FR}$. 
Move Control Word From Coprocessor 2

Format:  

CFC2 rt, Impl

The syntax shown above is an example using CFC1 as a model. The specific syntax is implementation dependent.

Purpose:  Move Control Word From Coprocessor 2

To copy a word from a Coprocessor 2 control register to a GPR

Description:  

GPR[rt] ← CP2CCR[Impl]

Copy the 32-bit word from the Coprocessor 2 control register denoted by the Impl field. The interpretation of the Impl field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

Restrictions:  

The result is UNPREDICTABLE if Impl specifies a register that does not exist.

Operation:  

temp ← CP2CCR[Impl]
GPR[rt] ← temp

Exceptions:  

Coprocessor Unusable, Reserved Instruction
Count Leading Ones in Word

**Format:** CLO rd, rs

**Purpose:** Count Leading Ones in Word

To count the number of leading ones in a word.

**Description:**

GPR[rd] ← count_leading_ones GPR[rs]

Bits 31..0 of GPR rs are scanned from most significant to least significant bit. The number of leading ones is counted and the result is written to GPR rd. If all of bits 31..0 were set in GPR rs, the result written to GPR rd is 32.

**Restrictions:**

To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in both the rt and rd fields of the instruction. The operation of the instruction is **UNPREDICTABLE** if the rt and rd fields of the instruction contain different values.

**Operation:**

```plaintext
temp ← 32
for i in 31 .. 0
    if GPR[rs]i = 0 then
        temp ← 31 - i
        break
    endif
endfor
GPR[rd] ← temp
```

**Exceptions:**

None
Count Leading Zeros in Word

**Format:** CLZ rd, rs

**Purpose:** Count Leading Zeros in Word

Count the number of leading zeros in a word.

**Description:**

\[
\text{temp} \leftarrow 32 \\
\text{for } i \text{ in } 31 \ldots 0 \\
\quad \text{if } \text{GPR}[rs]_i = 1 \text{ then} \\
\quad \quad \text{temp} \leftarrow 31 - i \\
\quad \quad \text{break} \\
\quad \text{endif} \\
\text{endfor} \\
\text{GPR}[rd] \leftarrow \text{temp}
\]

**Restrictions:**

To be compliant with the MIPS32 and MIPS64 Architecture, software must place the same GPR number in both the rt and rd fields of the instruction. The operation of the instruction is **UNPREDICTABLE** if the rt and rd fields of the instruction contain different values.

**Exceptions:**

None
Coprocessor Operation to Coprocessor 2

**Format:**  
```  
<table>
<thead>
<tr>
<th>COP2 func</th>
<th>MIPS32</th>
</tr>
</thead>
<tbody>
<tr>
<td>010010 CO</td>
<td></td>
</tr>
<tr>
<td>1 cofun</td>
<td></td>
</tr>
</tbody>
</table>
```

**Purpose:**  
Coprocessor Operation to Coprocessor 2  
To perform an operation to Coprocessor 2

**Description:**  
CoprocessorOperation(2, cofun)  
An implementation-dependent operation is performed to Coprocessor 2, with the cofun value passed as an argument. The operation may specify and reference internal coprocessor registers, and may change the state of the coprocessor conditions, but does not modify state within the processor. Details of coprocessor operation and internal state are described in the documentation for each Coprocessor 2 implementation.

**Restrictions:**

**Operation:**  
CoprocessorOperation(2, cofun)

**Exceptions:**  
Coprocessor Unusable  
Reserved Instruction
Move Control Word to Floating Point

**Format:**
CTC1 rt, fs

**Purpose:**
Move Control Word to Floating Point
To copy a word from a GPR to an FPU control register

**Description:**
FP_Control[fs] ← GPR[rt]

Copy the low word from GPR rt into the FP (coprocessor 1) control register indicated by fs.

Writing to the floating point Control/Status register, the FCSR, causes the appropriate exception if any Cause bit and its corresponding Enable bit are both set. The register is written before the exception occurs. Writing to FEXR to set a cause bit whose enable bit is already set, or writing to FENR to set an enable bit whose cause bit is already set causes the appropriate exception. The register is written before the exception occurs and whose EPC register contains the address of the CTC1 instruction.

The definition of this instruction has been extended in Release 5 to support user mode set and clear of StatusFR under the control of Config5UFR. This required feature is meant to facilitate transition from FR=0 to FR=1 floating-point register modes in order to obsolete FR=0 mode.

**Restrictions:**
There are a few control registers defined for the floating point unit. The result is UNPREDICTABLE if fs specifies a register that does not exist.

Furthermore, the result is UNPREDICTABLE if fd specifies the UFR or UNFR aliases, with fs anything other than 00000, GPR[0]. R5.03 implementations are required to produce a Reserved Instruction Exception; software must assume it is UNPREDICTABLE.

**Operation:**

```plaintext
temp ← GPR[rt]31..0
if fs = 1 and rt = 0 and FIRUFRP then /* clear UFR (CP1 Register 1) */
  if Config5UFR
    StatusFR ← 0
  else
    signalException(RI)
  endif
elseif fs = 4 and rt = 0 and FIRUFRP then /* clear UNFR (CP1 Register 4) */
  if Config5UFR
    StatusFR ← 1
  else
    signalException(RI)
  endif
elseif fs = 25 then /* FCCR */
  if temp31..8 ≠ 024 then
    UNPREDICTABLE
  else
    FCSR ← temp7..1 || FCSR24 || temp0 || FCSR22..0
  endif
```

![CTC1 Format](image-url)
elseif fs = 26 then /* FEXR */
    if temp31..18 ≠ 0 or temp11..7 ≠ 0 or temp2..0 ≠ 0 then
        UNPREDICTABLE
    else
        FCSR ← FCSR31..18 || temp17..12 || FCSR11..7 ||
        temp6..2 || FCSR1..0
        endif
    elseif fs = 28 then /* FENR */
    if temp31..12 ≠ 0 or temp6..3 ≠ 0 then
        UNPREDICTABLE
    else
        FCSR ← FCSR31..25 || temp2 || FCSR23..12 || temp11..7
        || FCSR6..2 || temp1..0
        endif
    elseif fs = 31 then /* FCSR */
    if (FCSRImpl field is not implemented) and(temp22..18 ≠ 0) then
        UNPREDICTABLE
    elseif (FCSRImpl field is implemented) and temp20..18 ≠ 0 then
        UNPREDICTABLE
    else
        FCSR ← temp
        endif
    else
        UNPREDICTABLE
    endif
CheckFPException()

Exceptions:
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:
Unimplemented Operation, Invalid Operation, Division-by-zero, Inexact, Overflow, Underflow

Historical Information:
For the MIPS I, II and III architectures, the contents of floating point control register fs are UNPREDICTABLE for the instruction immediately following CTC1.
MIPS V and MIPS32 introduced the three control registers that access portions of FCSR. These registers were not available in MIPS I, II, III, or IV.
MIPS32r5 introduced the UFR and UNFR register aliases that allow user level access to Status_FSR.
Move Control Word to Coprocessor 2

Format:  
CTC2 rt, Impl

The syntax shown above is an example using CTC1 as a model. The specific syntax is implementation dependent.

Purpose: Move Control Word to Coprocessor 2
To copy a word from a GPR to a Coprocessor 2 control register

Description:  
CP2CCR[Impl] ← GPR[rt]
Copy the low word from GPR rt into the Coprocessor 2 control register denoted by the Impl field. The interpretation of the Impl field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

Restrictions:
The result is UNPREDICTABLE if rd specifies a register that does not exist.

Operation:
    temp ← GPR[rt]
    CP2CCR[Impl] ← temp

Exceptions:
Coprocessor Unusable, Reserved Instruction
Floating Point Convert to Double Floating Point

**Format:**

- CVT.D.S fd, fs
- CVT.D.W fd, fs
- CVT.D.L fd, fs

**Purpose:**
Floating Point Convert to Double Floating Point

To convert an FP or fixed point value to double FP

**Description:**

\[ \text{FPR}[fd] \leftarrow \text{convert_and_round} \left( \text{FPR}[fs] \right) \]

The value in FPR \( fs \), in format \( fmt \), is converted to a value in double floating point format and rounded according to the current rounding mode in \( FCSR \). The result is placed in FPR \( fd \). If \( fmt \) is S or W, then the operation is always exact.

**Restrictions:**

The fields \( fs \) and \( fd \) must specify valid FPRs—\( fs \) for type \( fmt \) and \( fd \) for double floating point—if they are not valid, the result is \text{UNPREDICTABLE}.

The operand must be a value in format \( fmt \); if it is not, the result is \text{UNPREDICTABLE} and the value of the operand FPR becomes \text{UNPREDICTABLE}.

For CVT.D.L, the result of this instruction is \text{UNPREDICTABLE} if the processor is executing in the \( FR=0 \) 32-bit FPU register model; i.e. it is the \( FR=0 \) 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the \( FR=1 \) mode, but not with \( FR=0 \), and not on a 32-bit FPU.

**Operation:**

\[ \text{StoreFPR} \left( fd, D, \text{ConvertFmt} \left( \text{ValueFPR} \left( fs, fmt \right) \right), fmt, D \) \]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Unimplemented Operation, Inexact
Floating Point Convert to Long Fixed Point

Format:

- CVT.L.fmt
- CVT.L.S fd, fs
- CVT.L.D fd, fs

MIPS64, MIPS32 Release 2

MIPS64, MIPS32 Release 2

Purpose: Floating Point Convert to Long Fixed Point

To convert an FP value to a 64-bit fixed point

Description:

FPR[fd] ← convert_and_round(FPR[fs])

Convert the value in format fmt in FPR fs to long fixed point format and round according to the current rounding mode in FCSR. The result is placed in FPR fd.

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{63}\) to \(2^{63}-1\), the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the FCSR. If the Invalid Operation Enable bit is set in the FCSR, no result is written to fd and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{63}-1\), is written to fd.

Restrictions:

The fields fs and fd must specify valid FPRs—fs for type fmt and fd for long fixed point—if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

The result of this instruction is UNPREDICTABLE if the processor is executing in the \(FR=0\) 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the \(FR=1\) mode, but not with \(FR=0\), and not on a 32-bit FPU.

Operation:

StoreFPR (fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact,
Floating Point Convert Pair to Paired Single

Format: \texttt{CVT.PS.S \texttt{fd, fs, ft}}

Purpose: Floating Point Convert Pair to Paired Single

To convert two FP values to a paired single value

Description: \[ \text{FPR[fd]} \leftarrow \text{FPR[fs]_{31..0}} \parallel \text{FPR[ft]_{31..0}} \]

The single-precision values in FPR/\(fs\) and \(ft\) are written into FPR/\(fd\) as a paired-single value. The value in FPR/\(fs\) is written into the upper half, and the value in FPR/\(ft\) is written into the lower half.

CVT.PS.S is similar to PLL.PS, except that it expects operands of format \(S\) instead of \(PS\).

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields \(fs\) and \(ft\) must specify FPRs valid for operands of type \(S\); if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format \(S\); if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

The result of this instruction is UNPREDICTABLE if the processor is executing in the \(FR=0\) 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the \(FR=1\) mode, but not with \(FR=0\), and not on a 32-bit FPU.

Operation:

\[ \text{StoreFPR(fd, S, ValueFPR(fs,S) \parallel ValueFPR(ft,S))} \]

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation
Floating Point Convert to Single Floating Point

To convert an FP or fixed point value to single FP

**Description:**
\[ \text{FPR}[fd] \leftarrow \text{convert_and_round} (\text{FPR}[fs]) \]

The value in FPR \( fs \), in format \( fmt \), is converted to a value in single floating point format and rounded according to the current rounding mode in \( FCSR \). The result is placed in FPR \( fd \).

**Restrictions:**
The fields \( fs \) and \( fd \) must specify valid FPRs—\( fs \) for type \( fmt \) and \( fd \) for single floating point. If they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format \( fmt \); if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

For CVT.S.L, the result of this instruction is **UNPREDICTABLE** if the processor is executing in the \( FR=0 \) 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the \( FR=1 \) mode, but not with \( FR=0 \), and not on a 32-bit FPU.

**Operation:**
\[ \text{StoreFPR}(fd, S, \text{ConvertFmt(ValueFPR}(fs, fmt)), fmt, S) \]

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Invalid Operation, Unimplemented Operation, Inexact, Overflow, Underflow
Floating Point Convert Pair Lower to Single Floating Point

**Format:** \( \text{CVT.S.PL } \text{fd, } \text{fs} \)

**Purpose:**
Floating Point Convert Pair Lower to Single Floating Point
To convert one half of a paired single FP value to single FP

**Description:**
\[ FPR[fd] \leftarrow FPR[fs]_{31..0} \]
The lower paired single value in FPR \( fs \), in format \( PS \), is converted to a value in single floating point format. The result is placed in FPR \( fd \). This instruction can be used to isolate the lower half of a paired single value.
The operation is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**
The fields \( fs \) and \( fd \) must specify valid FPRs—\( fs \) for type \( PS \) and \( fd \) for single floating point. If they are not valid, the result is UNPREDICTABLE.
The operand must be a value in format \( PS \); if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.
The result of CVT.S.PL is UNPREDICTABLE if the processor is executing in the \( FR=0 \) 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the \( FR=1 \) mode, but not with \( FR=0 \), and not on a 32-bit FPU.

**Operation:**
\[ \text{StoreFPR (fd, S, ConvertFmt(ValueFPR(fs, PS), PL, S))} \]

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Floating Point Convert Pair Upper to Single Floating Point

**Format:** \( \text{CVT}.S.PU \ fd, \ fs \)

**Purpose:** Floating Point Convert Pair Upper to Single Floating Point

To convert one half of a paired single FP value to single FP

**Description:**
\[
\text{FPR}[fd] \leftarrow \text{FPR}[fs]_{63..32}
\]

The upper paired single value in FPR \( fs \), in format \( PS \), is converted to a value in single floating point format. The result is placed in FPR \( fd \). This instruction can be used to isolate the upper half of a paired single value.

The operation is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

The fields \( fs \) and \( fd \) must specify valid FPRs—\( fs \) for type \( PS \) and \( fd \) for single floating point. If they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format \( PS \); if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of CVT.S.PU is **UNPREDICTABLE** if the processor is executing the \( FR=0 \) 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the \( FR=1 \) mode, but not with \( FR=0 \), and not on a 32-bit FPU

**Operation:**

\[
\text{StoreFPR} \ (fd, \ S, \ \text{ConvertFmt} (\text{ValueFPR}(fs, \ PS), \ PU, \ S))
\]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Floating Point Convert to Word Fixed Point

CVT.W.fmt

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>fmt</td>
<td>0</td>
<td>00000</td>
<td>fs</td>
<td>fd</td>
<td>CVT.W</td>
<td>100100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**
- CVT.W.fmt
- CVT.W.S fd, fs
- CVT.W.D fd, fs

**Purpose:** Floating Point Convert to Word Fixed Point

To convert an FP value to 32-bit fixed point

**Description:**

\[
\text{FPR}[fd] \leftarrow \text{convert_and_round}(\text{FPR}[fs])
\]

The value in FPR \(fs\), in format \(fmt\), is converted to a value in 32-bit word fixed point format and rounded according to the current rounding mode in FCSR. The result is placed in FPR \(fd\).

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{31}\) to \(2^{31}-1\), the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the FCSR. If the Invalid Operation Enable bit is set in the FCSR, no result is written to \(fd\) and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{31}-1\), is written to \(fd\).

**Restrictions:**

- The fields \(fs\) and \(fd\) must specify valid FPRs—\(fs\) for type \(fmt\) and \(fd\) for word fixed point—if they are not valid, the result is UNPREDICTABLE.
- The operand must be a value in format \(fmt\); if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

**Operation:**

\[
\text{StoreFPR}(fd, W, \text{ConvertFmt}(\text{ValueFPR}(fs, fmt), fmt, W))
\]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Unimplemented Operation, Inexact
Debug Exception Return

**DERET**

### Format:
DERET

### Purpose:
Debug Exception Return

To return from a debug exception.

### Description:
DERET clears execution and instruction hazards, returns from Debug Mode and resumes non-debug execution at the instruction whose address is contained in the DEPC register. DERET does not execute the next instruction (i.e. it has no delay slot).

### Restrictions:
A DERET placed between an LL and SC instruction does not cause the SC to fail.

If the DEPC register with the return address for the DERET was modified by an MTC0 or a DMTC0 instruction, a CP0 hazard exists that must be removed via software insertion of the appropriate number of SSNOP instructions (for implementations of Release 1 of the Architecture) or by an EHB, or other execution hazard clearing instruction (for implementations of Release 2 of the Architecture).

DERET implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes (for Release 2 implementations, refer to the SYNCl instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier are seen starting with the instruction fetch and decode of the instruction at the PC to which the DERET returns.

This instruction is legal only if the processor is executing in Debug Mode. The operation of the processor is **UNDEFINED** if a DERET is executed in the delay slot of a branch or jump instruction.

### Operation:

```
DebugDM ← 0
DebugIEXI ← 0
if IsMIPS16Implemented() | (Config3ISA > 0) then
    PC ← DEPC31..1 || 0
    ISAMode ← DEPC0
else
    PC ← DEPC
endif
ClearHazards()
```

### Exceptions:

Coprocessor Unusable Exception
Reserved Instruction Exception
Disable Interrupts

**Purpose:** Disable Interrupts

To return the previous value of the *Status* register and disable interrupts. If DI is specified without an argument, GPR r0 is implied, which discards the previous value of the *Status* register.

**Description:**

\[
\text{GPR}[rt] \leftarrow \text{Status}; \text{Status}_{IE} \leftarrow 0
\]

The current value of the *Status* register is loaded into general register rt. The Interrupt Enable (IE) bit in the *Status* register is then cleared.

**Restrictions:**

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

**Operation:**

This operation specification is for the general interrupt enable/disable operation, with the sc field as a variable. The individual instructions DI and EI have a specific value for the sc field.

\[
\begin{align*}
\text{data} & \leftarrow \text{Status} \\
\text{GPR}[rt] & \leftarrow \text{data} \\
\text{Status}_{IE} & \leftarrow 0
\end{align*}
\]

**Exceptions:**

Coprocessor Unusable
Reserved Instruction (Release 1 implementations)

**Programming Notes:**

The effects of this instruction are identical to those accomplished by the sequence of reading *Status* into a GPR, clearing the IE bit, and writing the result back to *Status*. Unlike the multiple instruction sequence, however, the DI instruction cannot be aborted in the middle by an interrupt or exception.

This instruction creates an execution hazard between the change to the *Status* register and the point where the change to the interrupt enable takes effect. This hazard is cleared by the EHB, JALR.HB, JR.HB, or ERET instructions. Software must not assume that a fixed latency will clear the execution hazard.
Format:  DIV rs, rt

Purpose:  Divide Word
To divide a 32-bit signed integers

Description:  (HI, LO) ← GPR[rs] / GPR[rt]
The 32-bit word value in GPR rs is divided by the 32-bit value in GPR rt, treating both operands as signed values. The 32-bit quotient is placed into special register LO and the 32-bit remainder is placed into special register HI.

No arithmetic exception occurs under any circumstances.

Restrictions:
If the divisor in GPR rt is zero, the arithmetic result value is UNPREDICTABLE.

Operation:
q ← GPR[rs] 31..0 div GPR[rt] 31..0
LO ← q
r ← GPR[rs] 31..0 mod GPR[rt] 31..0
HI ← r

Exceptions:
None

Programming Notes:
No arithmetic exception occurs under any circumstances. If divide-by-zero or overflow conditions are detected and some action taken, then the divide instruction is typically followed by additional instructions to check for a zero divisor and/or for overflow. If the divide is asynchronous then the zero-divisor check can execute in parallel with the divide. The action taken on either divide-by-zero or overflow is either a convention within the program itself, or more typically within the system software; one possibility is to take a BREAK exception with a code field value to signal the problem to the system software.

As an example, the C programming language in a UNIX® environment expects division by zero to either terminate the program or execute a program-specification signal handler. C does not expect overflow to cause any exceptional condition. If the C compiler uses a divide instruction, it also emits code to test for a zero divisor and execute a BREAK instruction to inform the operating system if a zero is detected.

By default, most compilers for the MIPS architecture will emit additional instructions to check for the divide-by-zero and overflow cases when this instruction is used. In many compilers, the assembler mnemonic “DIV r0, rs, rt” can be used to prevent these additional test instructions to be emitted.

In some processors the integer divide 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 divide so that other instructions can execute in parallel.

Historical Perspective:
In MIPS I through MIPS III, if either of the two instructions preceding the divide is an MFHI or MFLO, the result of
the MFHI or MFLO is **UNPREDICTABLE**. Reads of the $HI$ or $LO$ special register must be separated from subsequent instructions that write to them by two or more instructions. This restriction was removed in MIPS IV and MIPS32 and all subsequent levels of the architecture.
Floating Point Divide

**Format:**

DIV.fmt

\[
\begin{array}{cccccc}
\text{COP1} & \text{fmt} & \text{ft} & \text{fs} & \text{fd} & \text{DIV} \\
010001 & 6 & 5 & 5 & 5 & 000011 \\
\end{array}
\]

**Purpose:** Floating Point Divide

To divide FP values

**Description:**

\[ \text{FPR}[fd] \leftarrow \text{FPR}[fs] / \text{FPR}[ft] \]

The value in FPR \( fs \) is divided by the value in FPR \( ft \). The result is calculated to infinite precision, rounded according to the current rounding mode in \( FCSR \), and placed into FPR \( fd \). The operands and result are values in format \( fmt \).

**Restrictions:**

The fields \( fs, ft, \) and \( fd \) must specify FPRs valid for operands of type \( fmt \); if they are not valid, the result is UNPREDICTABLE.

The operands must be values in format \( fmt \); if they are not, the result is UNPREDICTABLE and the value of the operand FPRs becomes UNPREDICTABLE.

**Operation:**

\[ \text{StoreFPR} (fd, fmt, \text{ValueFPR}(fs, fmt) / \text{ValueFPR}(ft, fmt)) \]

**Exceptions:**

Coprocessor Usable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Invalid Operation, Unimplemented Operation, Division-by-zero, Overflow, Underflow
**Divide Unsigned Word**

**MIPS32**

**Format:** DIVU rs, rt

**Purpose:** Divide Unsigned Word

To divide a 32-bit unsigned integers

**Description:** (HI, LO) ← GPR[rs] / GPR[rt]

The 32-bit word value in GPR rs is divided by the 32-bit value in GPR rt, treating both operands as unsigned values. The 32-bit quotient is placed into special register LO and the 32-bit remainder is placed into special register HI.

No arithmetic exception occurs under any circumstances.

**Restrictions:**

If the divisor in GPR rt is zero, the arithmetic result value is UNPREDICTABLE.

**Operation:**

\[
\begin{align*}
q & \leftarrow (0 || \text{GPR}[rs]_{31..0}) \text{ div } (0 || \text{GPR}[rt]_{31..0}) \\
r & \leftarrow (0 || \text{GPR}[rs]_{31..0}) \text{ mod } (0 || \text{GPR}[rt]_{31..0}) \\
\text{LO} & \leftarrow \text{sign extend}(q_{31..0}) \\
\text{HI} & \leftarrow \text{sign extend}(r_{31..0})
\end{align*}
\]

**Exceptions:**

None

**Programming Notes:**

See “Programming Notes” for the DIV instruction.

**Historical Perspective:**

In MIPS I through MIPS III, if either of the two instructions preceding the divide is an MFHI or MFLO, the result of the MFHI or MFLO is UNPREDICTABLE. Reads of the HI or LO special register must be separated from subsequent instructions that write to them by two or more instructions. This restriction was removed in MIPS IV and MIPS32 and all subsequent levels of the architecture.
Execution Hazard Barrier

**Format:** EHB

**MIPS32 Release 2**

**Purpose:** Execution Hazard Barrier

To stop instruction execution until all execution hazards have been cleared.

**Description:**

EHB is the assembly idiom used to denote execution hazard barrier. The actual instruction is interpreted by the hardware as SLL r0, r0, 3.

This instruction alters the instruction issue behavior on a pipelined processor by stopping execution until all execution hazards have been cleared. Other than those that might be created as a consequence of setting StatusCU0, there are no execution hazards visible to an unprivileged program running in User Mode. All execution hazards created by previous instructions are cleared for instructions executed immediately following the EHB, even if the EHB is executed in the delay slot of a branch or jump. The EHB instruction does not clear instruction hazards—such hazards are cleared by the JALR.HB, JR.HB, and ERET instructions.

**Restrictions:**

None

**Operation:**

ClearExecutionHazards()

**Exceptions:**

None

**Programming Notes:**

In MIPS32 Release 2 implementations, this instruction resolves all execution hazards. On a superscalar processor, EHB alters the instruction issue behavior in a manner identical to SSNOP. For backward compatibility with Release 1 implementations, the last of a sequence of SSNOPs can be replaced by an EHB. In Release 1 implementations, the EHB will be treated as an SSNOP, thereby preserving the semantics of the sequence. In Release 2 implementations, replacing the final SSNOP with an EHB should have no performance effect because a properly sized sequence of SSNOPs will have already cleared the hazard. As EHB becomes the standard in MIPS implementations, the previous SSNOPs can be removed, leaving only the EHB.
Enable Interrupts

**EI**

**Format:**

<table>
<thead>
<tr>
<th>Format</th>
<th>MIPS32 Release 2</th>
<th>MIPS32 Release 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>EI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EI rt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Purpose:**
Enable Interrupts

To return the previous value of the *Status* register and enable interrupts. If EI is specified without an argument, GPR r0 is implied, which discards the previous value of the *Status* register.

**Description:**

GPR[rt] ← Status; StatusIE ← 1

The current value of the *Status* register is loaded into general register *rt*. The Interrupt Enable (*IE*) bit in the *Status* register is then set.

**Restrictions:**

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

**Operation:**

This operation specification is for the general interrupt enable/disable operation, with the *sc* field as a variable. The individual instructions DI and EI have a specific value for the *sc* field.

```
data ← Status
GPR[rt] ← data
StatusIE ← 1
```

**Exceptions:**

Coprocessor Unusable
Reserved Instruction (Release 1 implementations)

**Programming Notes:**

The effects of this instruction are identical to those accomplished by the sequence of reading *Status* into a GPR, setting the *IE* bit, and writing the result back to *Status*. Unlike the multiple instruction sequence, however, the EI instruction cannot be aborted in the middle by an interrupt or exception.

This instruction creates an execution hazard between the change to the Status register and the point where the change to the interrupt enable takes effect. This hazard is cleared by the EHB, JALR.HB, JR.HB, or ERET instructions. Software must not assume that a fixed latency will clear the execution hazard.
**Exception Return**

<table>
<thead>
<tr>
<th>Format</th>
<th>ERET</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS32</td>
<td></td>
</tr>
</tbody>
</table>

**Purpose:** Exception Return

To return from interrupt, exception, or error trap.

**Description:**

ERET clears execution and instruction hazards, conditionally restores $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$ in a Release 2 implementation, and returns to the interrupted instruction at the completion of interrupt, exception, or error processing. ERET does not execute the next instruction (i.e., it has no delay slot).

**Restrictions:**

The operation of the processor is UNDEFINED if an ERET is executed in the delay slot of a branch or jump instruction.

An ERET placed between an LL and SC instruction will always cause the SC to fail.

ERET implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes (for Release 2 implementations, refer to the SYNCI instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier are seen starting with the instruction fetch and decode of the instruction at the PC to which the ERET returns.

In a Release 2 implementation, ERET does not restore $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$ if $Status_{BEV} = 1$, or if $Status_{ERL} = 1$ because any exception that sets $Status_{ERL}$ to 1 (Reset, Soft Reset, NMI, or cache error) does not save $SRSCtl_{CSS}$ in $SRSCtl_{PSS}$. If software sets $Status_{ERL}$ to 1, it must be aware of the operation of an ERET that may be subsequently executed.

**Operation:**

```plaintext
if Status_{ERL} = 1 then
    temp ← ErrorEPC
    Status_{ERL} ← 0
else
    temp ← EPC
    Status_{EXL} ← 0
    if (ArchitectureRevision ≥ 2) and (SRSCtl_{HSS} > 0) and (Status_{BEV} = 0) then
        SRSCtl_{CSS} ← SRSCtl_{PSS}
    endif
endif
if IsMIPS16Implemented() | (Config3ISA > 0) then
    PC ← temp_{31..1} || 0
    ISAMode ← temp_0
else
    PC ← temp
endif
LLbit ← 0
ClearHazards()
```

**Exceptions:**

- Coprocessor Unusable Exception
**Exception Return No Clear**

**Purpose:** Exception Return No Clear

To return from interrupt, exception, or error trap without clearing the LL.bit.

**Description:**

ERETNC clears execution and instruction hazards, conditionally restores $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$ when implemented, and returns to the interrupted instruction at the completion of interrupt, exception, or error processing. ERETNC does not execute the next instruction (i.e., it has no delay slot).

ERETNC is identical to ERET except that an ERETNC will not clear the LL.bit that is set by execution of an LL instruction, and thus when placed between an LL and SC sequence, will never cause the SC to fail.

An ERET should continue to be used by default in interrupt and exception processing handlers: the handler may have accessed a synchronizable block of memory common to code that is atomically accessing the memory, and where the code caused the exception or was interrupted. Similarly, a process context-swap must also continue to use an ERET in order to avoid a possible false success on execution of SC in the restored context.

Multiprocessor systems with non-coherent cores (i.e., without hardware coherence snooping) should also continue to use ERET, since it is the responsibility of software to maintain data coherence in the system.

An ERETNC is useful in cases where interrupt/exception handlers and kernel code involved in a process context-swap can guarantee no interference in accessing synchronizable memory across different contexts. ERETNC can also be used in an OS-level debugger to single-step through code for debug purposes, avoiding the false clearing of the LL.bit and thus failure of an LL and SC sequence in single-stepped code.

Software can detect the presence of ERETNC by reading $Config_{5LLB}$.

**Restrictions:**

The operation of the processor is UNDEFINED if an ERETNC is executed in the delay slot of a branch or jump instruction.

ERETNC implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes. (For Release 2 implementations, refer to the SYNCI instruction for additional information on resolving instruction hazards created by writing the instruction stream.) The effects of this barrier are seen starting with the instruction fetch and decode of the instruction in the PC to which the ERETNC returns.

**Operation:**

```
if StatusERL = 1 then
  temp ← ErrorEPC
  StatusERL ← 0
else
  temp ← EPC
  StatusEXL ← 0
  if (ArchitectureRevision ≥ 2) and (SRSCtl_{HSS} > 0) and (StatusBEV = 0) then
    SRSCtl_{CSS} ← SRSCtl_{PSS}
  endif
endif
if IsMIPS16Implemented() | (Config3ISA > 0) then
```
PC ← temp₃₁..₁ || 0
ISAMode ← temp₀
else
  PC ← temp
endif
ClearHazards()

Exceptions:
  Coprocessor Unusable Exception
Extract Bit Field

**Format:**  EXT rt, rs, pos, size

**Purpose:** Extract Bit Field

To extract a bit field from GPR rs and store it right-justified into GPR rt.

**Description:** GPR[rt] ← ExtractField(GPR[rs], msbd, lsb)

The bit field starting at bit pos and extending for size bits is extracted from GPR rs and stored zero-extended and right-justified in GPR rt. The assembly language arguments pos and size are converted by the assembler to the instruction fields msbd (the most significant bit of the destination field in GPR rt), in instruction bits 15..11, and lsb (least significant bit of the source field in GPR rs), in instruction bits 10..6, as follows:

\[
\begin{align*}
msbd & \leftarrow size - 1 \\
lsb & \leftarrow pos
\end{align*}
\]

The values of pos and size must satisfy all of the following relations:

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

*Figure 3.2* shows the symbolic operation of the instruction.

**Restrictions:**

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

The operation is **UNPREDICTABLE** if lsb+msbd > 31.

**Operation:**

\[
\begin{align*}
\text{if } (lsb + msbd) > 31 & \text{ then} \\
\text{UNPREDICTABLE} & \\
\text{endif} \\
temp & \leftarrow 0^{32-(msbd+1)} || GPR[rs]_{msbd+lsb..lsb}
\end{align*}
\]
GPR[rt] ← temp

Exceptions:
Reserved Instruction
Floating Point Floor Convert to Long Fixed Point

**Purpose:** Floating Point Floor Convert to Long Fixed Point

To convert an FP value to 64-bit fixed point, rounding down

**Description:**  
\[ \text{FPR}[fd] \leftarrow \text{convert_and_round}(\text{FPR}[fs]) \]

The value in FPR \(fs\), in format \(fmt\), is converted to a value in 64-bit long fixed point format and rounded toward \(-\infty\) (rounding mode 3). The result is placed in FPR \(fd\).

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{63}\) to \(2^{63}-1\), the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the FCSR. If the Invalid Operation Enable bit is set in the FCSR, no result is written to \(fd\) and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{63}-1\), is written to \(fd\).

**Restrictions:**

The fields \(fs\) and \(fd\) must specify valid FPRs—\(fs\) for type \(fmt\) and \(fd\) for long fixed point—if they are not valid, the result is **UNPREDICTABLE**.

The operand must be a value in format \(fmt\); if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in the \(FR=0\) 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the \(FR=1\) mode, but not with \(FR=0\), and not on a 32-bit FPU.

**Operation:**

\[ \text{StoreFPR}(fd, L, \text{ConvertFmt}(\text{ValueFPR}(fs, fmt), fmt, L)) \]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Unimplemented Operation, Inexact
Floating Point Floor Convert to Word Fixed Point

FLOOR.W.fmt

Format:  
FLOOR.W.fmt
FLOOR.W.S fd, fs  
FLOOR.W.D fd, fs

Purpose: Floating Point Floor Convert to Word Fixed Point

To convert a FP value to 32-bit fixed point, rounding down

Description:
FPR[fd] ← convert_and_round(FPR[fs])

The value in FPR fs, in format fmt, is converted to a value in 32-bit word fixed point format and rounded toward −∞ (rounding mode 3). The result is placed in FPR fd.

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{31}\) to \(2^{31} - 1\), the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the FCSR. If the Invalid Operation Enable bit is set in the FCSR, no result is written to fd and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{31} - 1\), is written to fd.

Restrictions:

The fields fs and fd must specify valid FPRs—fs for type fmt and fd for word fixed point—if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

Operation:

StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))

Exceptions:

Coprocessor Usable, Reserved Instruction

Floating Point Exceptions:

Invalid Operation, Unimplemented Operation, Inexact
Insert Bit Field

**Format:** INS rt, rs, pos, size

**Purpose:** Insert Bit Field

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

**Description:**

\[
\text{GPR}[rt] \leftarrow \text{InsertField}(\text{GPR}[rt], \text{GPR}[rs], \text{msb}, \text{lsb})
\]

The right-most size bits from GPR rs are merged into the value from GPR rt starting at bit position pos. The result is placed back in GPR rt. The assembly language arguments pos and size are converted by the assembler to the instruction fields msb (the most significant bit of the field), in instruction bits 15..11, and lsb (least significant bit of the field), in instruction bits 10..6, as follows:

\[
\begin{align*}
\text{msb} & \leftarrow \text{pos} + \text{size} - 1 \\
\text{lsb} & \leftarrow \text{pos}
\end{align*}
\]

The values of pos and size must satisfy all of the following relations:

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

Figure 3-10 shows the symbolic operation of the instruction.

**Restrictions:**

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

---

**Figure 3.3 Operation of the INS Instruction**

![Figure showing the operation of the INS instruction](image-url)
The operation is **UNPREDICTABLE** if $\text{lsb} > \text{msb}$.

**Operation:**

```
if lsb > msb) then
    UNPREDICTABLE
endif
GPR[rt] ← GPR[rt]_{31..msb+1} || GPR[rs]_{msb..lsb} || GPR[rt]_{lsb-1..0}
```

**Exceptions:**

Reserved Instruction
Jump

Format:  J target

Purpose: Jump

To branch within the current 256 MB-aligned region

Description:

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256 MB-aligned region. The low 28 bits of the target address is the instr_index field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

Restrictions:

Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

\[
\begin{align*}
\text{I:} & \\
\text{I+1:} & \quad \text{PC} \leftarrow \text{PC}_{\text{OPPLEN-1..28}} | | \text{instr_index} | | 0^2
\end{align*}
\]

Exceptions:
None

Programming Notes:

Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256MB region aligned on a 256MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

This definition creates the following boundary case: When the jump instruction is in the last word of a 256MB region, it can branch only to the following 256MB region containing the branch delay slot.
Jump and Link

**Format:** JAL target

**Purpose:** Jump and Link

To execute a procedure call within the current 256MB-aligned region

**Description:**
Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, at which location execution continues after a procedure call.

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256MB-aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

**Restrictions:**
Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

\[
\begin{align*}
I & : \quad \text{GPR}[31] \leftarrow \text{PC} + 8 \\
I+1 & : \quad \text{PC} \leftarrow \text{PC}_{\text{GPRLEN} - 1..28} || \text{instr}_\text{index} || 0^2
\end{align*}
\]

**Exceptions:**
None

**Programming Notes:**
Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256MB region aligned on a 256MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

This definition creates the following boundary case: When the branch instruction is in the last word of a 256MB region, it can branch only to the following 256MB region containing the branch delay slot.
Format: \texttt{JALR rs (rd = 31 implied)}
\texttt{JALR rd, rs}

Purpose: Jump and Link Register
To execute a procedure call to an instruction address in a register

Description: \texttt{GPR[rd] \leftarrow \text{return\_addr}, PC \leftarrow GPR[rs]}
Place the return address link in GPR \textit{rd}. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

\textit{For processors that do not implement the MIPS16e ASE nor microMIPS32/64 ISA:}

- Jump to the effective target address in GPR \textit{rs}. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

\textit{For processors that do implement the MIPS16e ASE or microMIPS32/64 ISA:}

- Jump to the effective target address in GPR \textit{rs}. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself. Set the \textit{ISA Mode} bit to the value in GPR \textit{rs} bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

In release 1 of the architecture, the only defined hint field value is 0, which sets default handling of JALR. In Release 2 of the architecture, bit 10 of the hint field is used to encode a hazard barrier. See the JALR.HB instruction description for additional information.

Restrictions:
Register specifiers \textit{rs} and \textit{rd} must not be equal, because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is \textit{UNPREDICTABLE}. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

If only one instruction set is implemented, then the effective target address must obey the alignment rules of the instruction set. If multiple instruction sets are implemented, the effective target address must obey the alignment rules of the intended instruction set of the target address as specified by the bit 0 or GPR \textit{rs}.

For processors that do not implement the microMIPS32/64 ISA, the effective target address in GPR \textit{rs} must be naturally-aligned. For processors that do not implement the MIPS16e ASE nor microMIPS32/64 ISA, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction.

For processors that do implement the MIPS16e ASE or microMIPS32/64 ISA, if target ISAMode bit is 0 (GPR \textit{rs} bit 0) is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

Processor operation is \textit{UNPREDICTABLE} if a branch, jump, ERET, DERET, or \texttt{WAIT} instruction is placed in the delay slot of a branch or jump.

Operation:
\begin{verbatim}
I: temp ← GPR[rs]
    GPR[rd] ← PC + 8
\end{verbatim}
I+1: if Config1_CA = 0 then
    PC ← temp
else
    PC ← temp\_GPRLEN-1..1 || 0
    ISAMode ← temp₀
endif

Exceptions:
None

Programming Notes:
This branch-and-link instruction that can select a register for the return link; other link instructions use GPR 31. The default register for GPR \( rd \), if omitted in the assembly language instruction, is GPR 31.
Jump and Link Register with Hazard Barrier

**Format:**
- \texttt{JALR.HB }\texttt{rs} (rd = 31 implied)
- \texttt{JALR.HB rd, rs}

**Purpose:** Jump and Link Register with Hazard Barrier

To execute a procedure call to an instruction address in a register and clear all execution and instruction hazards

**Description:**
- \texttt{GPR[rd]} ← \texttt{return_addr}, \texttt{PC ← GPR[rs]}, clear execution and instruction hazards

Place the return address link in GPR \texttt{rd}. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

*For processors that do not implement the MIPS16 ASE nor microMIPS32/64 ISA:*

- Jump to the effective target address in GPR \texttt{rs}. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

*For processors that do implement the MIPS16 ASE or microMIPS32/64 ISA:*

- Jump to the effective target address in GPR \texttt{rs}. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself. Set the \textit{ISA Mode} bit to the value in GPR \texttt{rs} bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

\texttt{JALR.HB} implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes (for Release 2 implementations, refer to the SYNCI instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier are seen starting with the instruction fetch and decode of the instruction at the PC to which the \texttt{JALR.HB} instruction jumps. An equivalent barrier is also implemented by the \texttt{ERET} instruction, but that instruction is only available if access to Coprocessor 0 is enabled, whereas \texttt{JALR.HB} is legal in all operating modes.

This instruction clears both execution and instruction hazards. Refer to the \texttt{EHB} instruction description for the method of clearing execution hazards alone.

\texttt{JALR.HB} uses bit 10 of the instruction (the upper bit of the hint field) to denote the hazard barrier operation.

**Restrictions:**

Register specifiers \texttt{rs} and \texttt{rd} must not be equal, because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is \textit{UNPREDICTABLE}. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

If only one instruction set is implemented, then the effective target address must obey the alignment rules of the instruction set. If multiple instruction sets are implemented, the effective target address must obey the alignment rules of the intended instruction set of the target address as specified by the bit 0 or GPR \texttt{rs}.

For processors that do not implement the microMIPS32/64 ISA, the effective target address in GPR \texttt{rs} must be naturally-aligned. For processors that do not implement the MIPS16 ASE nor microMIPS32/64 ISA, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched.
as an instruction.

For processors that do implement the MIPS16 ASE or microMIPS32/64 ISA, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

After modifying an instruction stream mapping or writing to the instruction stream, execution of those instructions has **UNPREDICTABLE** behavior until the instruction hazard has been cleared with JALR.HB, JR.HB, ERET, or DERET. Further, the operation is **UNPREDICTABLE** if the mapping of the current instruction stream is modified.

JALR.HB does not clear hazards created by any instruction that is executed in the delay slot of the JALR.HB. Only hazards created by instructions executed before the JALR.HB are cleared by the JALR.HB.

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

\[
\begin{align*}
I: & \quad \text{temp} \leftarrow \text{GPR}[rs] \\
& \quad \text{GPR}[rd] \leftarrow \text{PC} + 8 \\
I+1: & \quad \text{if Config1CA} = 0 \text{ then} \\
& \quad \text{PC} \leftarrow \text{temp} \\
& \quad \text{else} \\
& \quad \text{PC} \leftarrow \text{tempGPRLEN-1..1} \mathbin{|} 0 \\
& \quad \text{ISAMode} \leftarrow \text{temp}_0 \\
& \quad \text{endif} \\
& \quad \text{ClearHazards}()
\end{align*}
\]

**Exceptions:**

None

**Programming Notes:**

This branch-and-link instruction can select a register for the return link; other link instructions use GPR 31. The default register for GPR \( rd \), if omitted in the assembly language instruction, is GPR 31.

This instruction implements the final step in clearing execution and instruction hazards before execution continues. A hazard is created when a Coprocessor 0 or TLB write affects execution or the mapping of the instruction stream, or after a write to the instruction stream. When such a situation exists, software must explicitly indicate to hardware that the hazard should be cleared. Execution hazards alone can be cleared with the EHB instruction. Instruction hazards can only be cleared with a JR.HB, JALR.HB, or ERET instruction. These instructions cause hardware to clear the hazard before the instruction at the target of the jump is fetched. Note that because these instructions are encoded as jumps, the process of clearing an instruction hazard can often be included as part of a call (JALR) or return (JR) sequence, by simply replacing the original instructions with the HB equivalent.

**Example:** Clearing hazards due to an ASID change

```
/*
 * Code used to modify ASID and call a routine with the new
 * mapping established.
 *
 * a0 = New ASID to establish
 * al = Address of the routine to call
 */

mfc0 v0, C0_EntryHi    /* Read current ASID */
li v1, ~M_EntryHiASID /* Get negative mask for field */
and v0, v0, v1        /* Clear out current ASID value */
or v0, v0, a0          /* OR in new ASID value */
mtc0 v0, C0_EntryHi    /* Rewrite EntryHi with new ASID */
jalr.hb al            /* Call routine, clearing the hazard */
```
nop
Jump and Link Exchange

**Format:**  JALX target

**Purpose:** Jump and Link Exchange

To execute a procedure call within the current 256 MB-aligned region and change the *ISA Mode* from MIPS32 to microMIPS32 or MIPS16e.

**Description:**

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, at which location execution continues after a procedure call. The value stored in GPR 31 bit 0 reflects the current value of the *ISA Mode* bit.

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256 MB-aligned region. The low 28 bits of the target address is the `instr_index` field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address, toggling the *ISA Mode* bit. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

**Restrictions:**

This instruction only supports 32-bit aligned branch target addresses.

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

If the microMIPS base architecture is not implemented and the MIPS16e ASE is not implemented, a Reserved Instruction Exception is initiated.

**Operation:**

\[
\begin{align*}
I & : & \text{GPR}[31] & \leftarrow \text{PC} + 8 \\
I+1 & : & \text{PC} & \leftarrow \text{PC}_{\text{GPRLEN}-1..28} || \text{instr_index} || 0^2 \\
& & \text{ISAMode} & \leftarrow (\text{not ISAMode})
\end{align*}
\]

**Exceptions:**

None

**Programming Notes:**

Forming the branch target address by concatenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

This definition creates the following boundary case: When the branch instruction is in the last word of a 256 MB region, it can branch only to the following 256 MB region containing the branch delay slot.
**Jump Register (JR)**

**Format:** \texttt{JR rs}  

**Purpose:** Jump Register  
To execute a branch to an instruction address in a register

**Description:**  
\[ \text{PC} \leftarrow \text{GPR}[rs] \]  
Jump to the effective target address in GPR \( rs \). Execute the instruction following the jump, in the branch delay slot, before jumping.

For processors that implement the MIPS16e ASE or microMIPS32/64 ISA, set the \textit{ISA Mode} bit to the value in GPR \( rs \) bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

**Restrictions:**  
If only one instruction set is implemented, then the effective target address must obey the alignment rules of the instruction set. If multiple instruction sets are implemented, the effective target address must obey the alignment rules of the intended instruction set of the target address as specified by the bit 0 or GPR \( rs \).

For processors that do not implement the microMIPS ISA, the effective target address in GPR \( rs \) must be naturally-aligned. For processors that do not implement the MIPS16e ASE or microMIPS ISA, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction.

For processors that do implement the MIPS16e ASE or microMIPS ISA, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

In release 1 of the architecture, the only defined hint field value is 0, which sets default handling of JR. In Release 2 of the architecture, bit 10 of the hint field is used to encode an instruction hazard barrier. See the JR.HB instruction description for additional information.

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

\[
\begin{align*}
I & : \text{temp} \leftarrow \text{GPR}[rs] \\
I+1: & \text{if Config1CA = 0 then} \\
& \quad \text{PC} \leftarrow \text{temp} \\
& \quad \text{else} \\
& \quad \quad \text{PC} \leftarrow \text{temp_{GPRLEN-1..1} | | 0} \\
& \quad \quad \text{ISAMode} \leftarrow \text{temp_0} \\
& \quad \text{endif}
\end{align*}
\]

**Exceptions:**  
None

**Programming Notes:**  
Software should use the value 31 for the \( rs \) field of the instruction word on return from a JAL, JALR, or BGEZAL, and should use a value other than 31 for remaining uses of JR.
Jump Register with Hazard Barrier

**Purpose:** Jump Register with Hazard Barrier

To execute a branch to an instruction address in a register and clear all execution and instruction hazards.

**Description:** $PC \leftarrow GPR[rs]$, clear execution and instruction hazards

Jump to the effective target address in GPR $rs$. Execute the instruction following the jump, in the branch delay slot, before jumping.

JR.HB implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes (for Release 2 implementations, refer to the SYNCI instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier are seen starting with the instruction fetch and decode of the instruction at the PC to which the JR.HB instruction jumps. An equivalent barrier is also implemented by the ERET instruction, but that instruction is only available if access to Coprocessor 0 is enabled, whereas JR.HB is legal in all operating modes.

This instruction clears both execution and instruction hazards. Refer to the EHB instruction description for the method of clearing execution hazards alone.

JR.HB uses bit 10 of the instruction (the upper bit of the hint field) to denote the hazard barrier operation.

For processors that implement the MIPS16e ASE or microMIPS32/64 ISA, set the ISA Mode bit to the value in GPR $rs$ bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one.

**Restrictions:**

If only one instruction set is implemented, then the effective target address must obey the alignment rules of the instruction set. If multiple instruction sets are implemented, the effective target address must obey the alignment rules of the intended instruction set of the target address as specified by the bit 0 or GPR $rs$.

For processors that do not implement the microMIPS ISA, the effective target address in GPR $rs$ must be naturally-aligned. For processors that do not implement the MIPS16 ASE or microMIPS ISA, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction.

For processors that do implement the MIPS16 ASE or microMIPS ISA, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

After modifying an instruction stream mapping or writing to the instruction stream, execution of those instructions has UNPREDICTABLE behavior until the hazard has been cleared with JALR.HB, JR.HB, ERET, or DERET. Further, the operation is UNPREDICTABLE if the mapping of the current instruction stream is modified.

JR.HB does not clear hazards created by any instruction that is executed in the delay slot of the JR.HB. Only hazards created by instructions executed before the JR.HB are cleared by the JR.HB.

Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**

$I$: \( temp \leftarrow GPR[rs] \)
Jump Register with Hazard Barrier

**I+1:** if Config1_CA = 0 then
\[
\text{PC} \leftarrow \text{temp}
\]
exthelse\[
\text{PC} \leftarrow \text{temp}_{\text{CSRLEN-1..1}} || 0
\]
\[
\text{ISAMode} \leftarrow \text{temp}_{0}
\]
endif
ClearHazards()

**Exceptions:**
None

**Programming Notes:**
This instruction implements the final step in clearing execution and instruction hazards before execution continues. A hazard is created when a Coprocessor 0 or TLB write affects execution or the mapping of the instruction stream, or after a write to the instruction stream. When such a situation exists, software must explicitly indicate to hardware that the hazard should be cleared. Execution hazards alone can be cleared with the EHB instruction. Instruction hazards can only be cleared with a JR.HB, JALR.HB, or ERET instruction. These instructions cause hardware to clear the hazard before the instruction at the target of the jump is fetched. Note that because these instructions are encoded as jumps, the process of clearing an instruction hazard can often be included as part of a call (JALR) or return (JR) sequence, by simply replacing the original instructions with the HB equivalent.

**Example:** Clearing hazards due to an ASID change

```c
/*
 * Routine called to modify ASID and return with the new
 * mapping established.
 *
 * a0 = New ASID to establish
 */
```
```c
mfc0 v0, C0_EntryHi /* Read current ASID */
l1 v1, ~M_EntryHiASID /* Get negative mask for field */
and v0, v0, v1 /* Clear out current ASID value */
or v0, v0, a0 /* OR in new ASID value */
mtc0 v0, C0_EntryHi /* Rewrite EntryHi with new ASID */
jr.hb ra /* Return, clearing the hazard */
nop
```

**Example:** Making a write to the instruction stream visible

```c
/*
 * Routine called after new instructions are written to
 * make them visible and return with the hazards cleared.
 */
```
```c
{Synchronize the caches - see the SYNCI and CACHE instructions}
sync /* Force memory synchronization */
jr.hb ra /* Return, clearing the hazard */
nop
```

**Example:** Clearing instruction hazards in-line

```c
la AT, 10f
jr.hb AT /* Jump to next instruction, clearing */
nop /* hazards */
```

10:
Load Byte

Format: \texttt{LB rt, offset(base)}

Purpose: Load Byte

To load a byte from memory as a signed value

Description: \texttt{GPR[rt] \leftarrow memory[GPR[base] + offset]}

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, sign-extended, and placed in GPR \texttt{rt}. The 16-bit signed \texttt{offset} is added to the contents of GPR \texttt{base} to form the effective address.

Restrictions:

None

Operation:

\begin{align*}
\text{vAddr} & \leftarrow \text{signExtend}(\text{offset}) + \text{GPR[base]} \\
(\text{pAddr}, \text{CCA}) & \leftarrow \text{AddressTranslation(vAddr, DATA, LOAD)} \\
\text{pAddr} & \leftarrow \text{pAddr}_{\text{PSIZE}-1..2} \parallel (\text{pAddr}_{1..0} \oplus \text{ReverseEndian}^2) \\
\text{memword} & \leftarrow \text{LoadMemory(CCA, BYTE, pAddr, vAddr, DATA)} \\
\text{byte} & \leftarrow \text{vAddr}_{1..0} \oplus \text{BigEndianCPU}^2 \\
\text{GPR[rt]} & \leftarrow \text{signExtend(\text{memword}_{7\text{..}8}\text{byte}..8\text{byte})}
\end{align*}

Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch
Load Byte EVA

Purpose: Load Byte EVA

To load a byte as a signed value from user mode virtual address space when executing in kernel mode.

Description: $GPR[rt] \leftarrow \text{memory}[GPR[base] + \text{offset}]$

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, sign-extended, and placed in GPR $rt$. The 9-bit signed $\text{offset}$ is added to the contents of GPR $base$ to form the effective address.

The LBE instruction functions in exactly the same fashion as the LB instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode and executing in kernel mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $Config5_{EVA}$ field being set to one.

Restrictions:

Only usable when access to Coprocessor0 is enabled and accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Operation:

$vAddr \leftarrow \text{sign\_extend}(\text{offset}) + GPR[base]$

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

$pAddr \leftarrow pAddr_{\text{PSIZE}-1..2} || (pAddr_{1..0} \text{xor ReverseEndian}^2)$

$\text{memword} \leftarrow \text{LoadMemory} (CCA, \text{BYTE}, pAddr, vAddr, \text{DATA})$

$\text{byte} \leftarrow vAddr_{1..0} \text{xor BigEndianCPU}^2$

$GPR[rt] \leftarrow \text{sign\_extend}(\text{memword}_{7+8*\text{byte}..8*\text{byte}})$

Exceptions:

TLB Refill
TLB Invalid
Bus Error
Address Error
Watch
Reserved Instruction
Coprocessor Unusable
### Load Byte Unsigned (LBU)

#### Format:

\[
\text{LBU } rt, \text{ offset}(\text{base})
\]

#### Purpose:
Load Byte Unsigned

To load a byte from memory as an unsigned value.

#### Description:

\[
\text{GPR}[rt] \leftarrow \text{memory}[\text{GPR}[\text{base}] + \text{offset}]
\]

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, zero-extended, and placed in GPR \( rt \). The 16-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form the effective address.

#### Restrictions:
None

#### Operation:

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
(\text{pAddr}, \text{CCA}) & \leftarrow \text{AddressTranslation}(\text{vAddr}, \text{DATA}, \text{LOAD}) \\
\text{pAddr} & \leftarrow \text{pAddr}_{\text{PSIZE}-1..2} \ | \ (\text{pAddr}_{1..0} \ \text{xor} \ \text{ReverseEndian})^2 \\
\text{memword} & \leftarrow \text{LoadMemory}(\text{CCA}, \text{BYTE}, \text{pAddr}, \text{vAddr}, \text{DATA}) \\
\text{byte} & \leftarrow \text{vAddr}_{1..0} \ \text{xor} \ \text{BigEndianCPU}^2 \\
\text{GPR}[rt] & \leftarrow \text{zero\_extend}(\text{memword}_{7+8\times\text{byte}..8\times\text{byte}})
\end{align*}
\]

#### Exceptions:

TLB Refill, TLB Invalid, Address Error, Watch
Load Byte Unsigned EVA

**Format:**  
LBUE rt, offset(base)  

**Purpose:** Load Byte Unsigned EVA  
To load a byte as an unsigned value from user mode virtual address space when executing in kernel mode.

**Description:**  
GPR[rt] ← memory[GPR[base] + offset]  
The contents of the 8-bit byte at the memory location specified by the effective address are fetched, zero-extended, and placed in GPR rt. The 9-bit signed offset is added to the contents of GPR base to form the effective address.

The LBUE instruction functions in exactly the same fashion as the LBU instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the Config5EVA field being set to one.

**Restrictions:**  
Only usable when access to Coprocessor0 is enabled and accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

**Operation:**  
\[
\begin{align*}
&v\text{Addr} \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR[base]} \\
&(p\text{Addr}, \text{CCA}) \leftarrow \text{Address\_Translation} \left(v\text{Addr}, \text{DATA, LOAD}\right) \\
&p\text{Addr} \leftarrow p\text{Addr}_{\text{P\text{SIZE}}-1..2} \ || \ (p\text{Addr}_{1..0} \ xor \ \text{Reverse\_Endian}_{2}) \\
&\text{memword} \leftarrow \text{LoadMemory} \left(\text{CCA, BYTE, pAddr, vAddr, DATA}\right) \\
&\text{byte} \leftarrow v\text{Addr}_{1..0} \ xor \ \text{Big\_Endian}_{2} \\
&\text{GPR}[rt] \leftarrow \text{zero\_extend}(\text{memword}_{7+8\times \text{byte}..8\times \text{byte}})
\end{align*}
\]

**Exceptions:**  
TLB Refill  
TLB Invalid  
Bus Error  
Address Error  
Watch  
Reserved Instruction  
Coprocessor Unusable
Load Doubleword to Floating Point

**Format:** LDC1 ft, offset(base)  

**Purpose:** Load Doubleword to Floating Point  
To load a doubleword from memory to an FPR  

**Description:** FPR[ft] ← memory[GPR[base] + offset]  
The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in FPR ft. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

**Restrictions:**  
An Address Error exception occurs if EffectiveAddress2..0 ≠ 0 (not doubleword-aligned).

**Operation:**  
```
vAddr ← sign_extend(offset) + GPR[base]  
if vAddr2..0 ≠ 03 then  
    SignalException(AddressError)  
endif  
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)  
paddr ← paddr xor ((BigEndianCPU xor ReverseEndian) || 02)  
memlsw ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)  
paddr ← paddr xor 0b100  
memmsw ← LoadMemory(CCA, WORD, pAddr, vAddr+4, DATA)  
memdoubleword ← memmsw || memlsw  
StoreFPR(ft, UNINTERPRETED_DOUBLEWORD, memdoubleword)  
```

**Exceptions:**  
Coprocessor Usable, Reserved Instruction, TLB Refill, TLB Invalid, Address Error, Watch
Load Doubleword to Coprocessor 2

**Format:** \( \text{LDC2} \ rt, \ \text{offset(base)} \)

**Purpose:** Load Doubleword to Coprocessor 2

To load a doubleword from memory to a Coprocessor 2 register

**Description:** \( \text{CPR}[2, rt, 0] \leftarrow \text{memory[GPR[base] + offset]} \)

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in Coprocessor 2 register \( rt \). The 16-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form the effective address.

**Restrictions:**

An Address Error exception occurs if EffectiveAddress\( _{2,0} \neq 0 \) (not doubleword-aligned).

**Operation:**

\[
\begin{align*}
v\text{Addr} & \leftarrow \text{sign\_extend(\text{offset}) + GPR[base]} \\
\text{if vAddr} \_{2..0} & \neq 0^3 \text{ then SignalException(AddressError) endif} \\
p\text{Addr} & \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \\
p\text{addr} & \leftarrow p\text{addr} \oplus ((\text{BigEndianCPU} \oplus \text{ReverseEndian}) || \ 0^2) \\
\text{memlsw} & \leftarrow \text{LoadMemory(CCA, WORD, pAddr, vAddr, DATA)} \\
p\text{addr} & \leftarrow p\text{addr} \oplus \text{0b100} \\
\text{memmsw} & \leftarrow \text{LoadMemory(CCA, WORD, pAddr, vAddr+4, DATA)} \\
& \leftarrow \text{memlsw} \\
& \leftarrow \text{memmsw}
\end{align*}
\]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, Address Error, Watch
Load Doubleword Indexed to Floating Point

**Format:**
LDXC1 fd, index(base)

**Purpose:** Load Doubleword Indexed to Floating Point
To load a doubleword from memory to an FPR (GPR+GPR addressing)

**Description:**
FPR[fd] ← memory[GPR[base] + GPR[index]]
The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in FPR fd. The contents of GPR index and GPR base are added to form the effective address.

**Restrictions:**
An Address Error exception occurs if EffectiveAddress2..0 ≠ 0 (not doubleword-aligned).

**Compatibility and Availability:**
LDXC1: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required in MIPS32r2 and all subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode (FIRF64=0 or 1, StatusFR=0 or 1).

**Operation:**
```
vAddr ← GPR[base] + GPR[index]
if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
paddr ← paddr xor ((BigEndianCPU xor ReverseEndian) || 02)
memlsw ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
paddr ← paddr xor 0b100
memmsw ← LoadMemory(CCA, WORD, pAddr, vAddr+4, DATA)
memdoubleword ← memmsw || memlsw
StoreFPR(fd, UNINTERPRETED_DOUBLEWORD, memdoubleword)
```

**Exceptions:**
TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Coprocessor Unusable, Watch
Load Halfword

**Format:**  LH rt, offset(base)  

**Purpose:**  Load Halfword  
To load a halfword from memory as a signed value  

**Description:**  GPR[rt] ← memory[GPR[base] + offset]  
The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, sign-extended, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.  

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

**Operation:**  
vAddr ← sign_extend(offset) + GPR[base]  
if vAddr0 ≠ 0 then  
    SignalException(AddressError)  
endif  
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)  
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor (ReverseEndian || 0))  
memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)  
byte ← vAddr1..0 xor (BigEndianCPU || 0)  
GPR[rt] ← sign_extend(memword15+8*byte..8*byte)  

**Exceptions:**  
TLB Refill, TLB Invalid, Bus Error, Address Error, Watch
Load Halfword LH
Load Halfword EVA

**Purpose:** Load Halfword EVA

To load a halfword as a signed value from user mode virtual address space when executing in kernel mode.

**Description:**

\[
\text{GPR}[rt] \leftarrow \text{memory}[\text{GPR}[\text{base}] + \text{offset}]
\]

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, sign-extended, and placed in GPR \( rt \). The 9-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form the effective address.

The LHE instruction functions in exactly the same fashion as the LH instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the \( \text{Config}_{EVA} \) field being set to one.

**Restrictions:**

Only usable when access to Coprocessor0 is enabled and accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

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

**Operation:**

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
(\text{pAddr, CCA}) & \leftarrow \text{AddressTranslation}(\text{vAddr, DATA, LOAD}) \\
\text{pAddr} & \leftarrow \text{pAddr}_{\text{PSIZE}-1..2} \ || \ (\text{pAddr}_{1..0} \text{xor (ReverseEndian || 0)}) \\
\text{memword} & \leftarrow \text{LoadMemory}(\text{CCA, HALFWORD, pAddr, vAddr, DATA}) \\
\text{byte} & \leftarrow \text{vAddr}_{1..0} \text{xor (BigEndianCPU || 0)} \\
\text{GPR}[rt] & \leftarrow \text{sign\_extend}(\text{memword}_{15..8} \text{byte}..8\text{byte})
\end{align*}
\]

**Exceptions:**

- TLB Refill
- TLB Invalid
- Bus Error
- Address Error
- Watch
- Reserved Instruction
- Coprocessor Unusable
Load Halfword Unsigned

**Format:** \( \text{LHU} \ rt, \ offset(\text{base}) \)

**Purpose:** Load Halfword Unsigned
To load a halfword from memory as an unsigned value

**Description:** \( \text{GPR}[rt] \leftarrow \text{memory}[\text{GPR}[\text{base}] + \text{offset}] \)
The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, zero-extended, and placed in GPR \( rt \). The 16-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form the effective address.

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

**Operation:**
\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
\text{if } \text{vAddr}_0 \neq 0 \text{ then} & \\
& \quad \text{SignalException(AddressError)} \\
\text{endif} \\
(\text{pAddr}, \text{CCA}) & \leftarrow \text{AddressTranslation}(\text{vAddr}, \text{DATA}, \text{LOAD}) \\
\text{pAddr} & \leftarrow \text{pAddr}_{\text{PSIZE}-1..2} \ || \ (\text{pAddr}_{1..0} \ xor \ (\text{ReverseEndian} \ || \ 0)) \\
\text{memword} & \leftarrow \text{LoadMemory}(\text{CCA}, \text{HALFWORD}, \text{pAddr}, \text{vAddr}, \text{DATA}) \\
\text{byte} & \leftarrow \text{vAddr}_{1..0} \ xor \ (\text{BigEndianCPU} \ || \ 0) \\
\text{GPR}[rt] & \leftarrow \text{zero\_extend}(\text{memword}_{15+8*\text{byte}..8*\text{byte}})
\end{align*}
\]

**Exceptions:**
TLB Refill, TLB Invalid, Address Error, Watch
Load Halfword Unsigned EVA

**Format:** LHUE rt, offset(base)

**Purpose:** Load Halfword Unsigned EVA

To load a halfword as an unsignedor value from user mode virtual address space when executing in kernel mode.

**Description:**
\[
\text{GPR}[rt] \leftarrow \text{memory}[\text{GPR}[\text{base}] + \text{offset}]
\]

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, zero-extended, and placed in GPR \(rt\). The 9-bit signed \(\text{offset}\) is added to the contents of GPR \(\text{base}\) to form the effective address.

The LHUE instruction functions in exactly the same fashion as the LHU instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the \(\text{Config}_5\_\text{EVA}\) field being set to one.

**Restrictions:**

Only usable when access to Coprocessor0 is enabled and accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

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

**Operation:**

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
(\text{pAddr, CCA}) & \leftarrow \text{AddressTranslation}(\text{vAddr}, \text{DATA}, \text{LOAD}) \\
\text{pAddr} & \leftarrow \text{pAddr}_{\text{PSIZE}-1..2} | | (\text{pAddr}_{1..0} \text{ xor} (\text{ReverseEndian} | | 0)) \\
\text{memword} & \leftarrow \text{LoadMemory}(\text{CCA}, \text{HALFWORD}, \text{pAddr}, \text{vAddr}, \text{DATA}) \\
\text{byte} & \leftarrow \text{vAddr}_{1..0} \text{ xor} (\text{BigEndianCPU} | | 0) \\
\text{GPR}[rt] & \leftarrow \text{zero\_extend}(\text{memword}_{15..8}*\text{byte}..8*\text{byte})
\end{align*}
\]

**Exceptions:**

TLB Refill
TLB Invalid
Bus Error
Address Error
Watch
Reserved Instruction
Coprocessor Unusable
Format: \text{LL } rt, \text{ offset}(base) \\

Purpose: Load Linked Word
To load a word from memory for an atomic read-modify-write

Description: \text{GPR}[rt] \leftarrow \text{memory}[\text{GPR}[base] + \text{offset}]
The LL and SC instructions provide the primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and written into GPR \textit{rt}. The 16-bit signed \textit{offset} is added to the contents of GPR \textit{base} to form an effective address.

This begins a RMW sequence on the current processor. There can be only one active RMW sequence per processor. When an LL is executed it starts an active RMW sequence replacing any other sequence that was active. The RMW sequence is completed by a subsequent SC instruction that either completes the RMW sequence atomically and succeeds, or does not and fails.

Executing LL on one processor does not cause an action that, by itself, causes an SC for the same block to fail on another processor.

An execution of LL does not have to be followed by execution of SC; a program is free to abandon the RMW sequence without attempting a write.

Restrictions:
The addressed location must be synchronizable by all processors and I/O devices sharing the location; if it is not, the result is \textit{UNPREDICTABLE}. Which storage is synchronizable is a function of both CPU and system implementations. See the documentation of the SC instruction for the formal definition.

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

Operation:
\begin{verbatim}
  vAddr ← sign_extend(offset) + GPR[base]
  if vAddr_{1..0} ≠ 0^2 then
    SignalException(AddressError)
  endif
  (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
  memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
  GPR[rt] ← memword
  LLbit ← 1
\end{verbatim}

Exceptions:
TLB Refill, TLB Invalid, Address Error, Watch
Load Linked Word EVA

**Format:** LLE rt, offset(base)

**Purpose:** Load Linked Word EVA

To load a word from a user mode virtual address when executing in kernel mode for an atomic read-modify-write operation.

**Description:**

\[ GPR[rt] \leftarrow \text{memory}[GPR[base] + \text{offset}] \]

The LLE and SCE instructions provide the primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations using user mode virtual addresses while executing in kernel mode.

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and written into GPR \( rt \). The 16-bit signed \( offset \) is added to the contents of GPR \( base \) to form an effective address.

This begins a RMW sequence on the current processor. There can be only one active RMW sequence per processor. When an LLE is executed it starts an active RMW sequence replacing any other sequence that was active. The RMW sequence is completed by a subsequent SCE instruction that either completes the RMW sequence atomically and succeeds, or does not and fails.

Executing LLE on one processor does not cause an action that, by itself, causes an SCE for the same block to fail on another processor.

An execution of LLE does not have to be followed by execution of SCE; a program is free to abandon the RMW sequence without attempting a write.

The LLE instruction functions in exactly the same fashion as the LL instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Segmentation Control for additional information.

Implementation of this instruction is specified by the \( \text{Config5}_{EVA} \) field being set to one.

**Restrictions:**

The addressed location must be synchronizable by all processors and I/O devices sharing the location; if it is not, the result is UNPREDICTABLE. Which storage is synchronizable is a function of both CPU and system implementations. See the documentation of the SCE instruction for the formal definition.

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

**Operation:**

\[
\begin{align*}
v\text{Addr} &\leftarrow \text{sign\_extend}(\text{offset}) + GPR[\text{base}] \\
\text{if } v\text{Addr}_{1...0} &\neq 0^2 \text{ then} \\
\quad &\text{SignalException(AddressError)} \\
\text{endif} \\
(p\text{Addr}, CCA) &\leftarrow \text{AddressTranslation}(v\text{Addr}, \text{DATA}, \text{LOAD}) \\
\text{memword} &\leftarrow \text{LoadMemory}(CCA, \text{WORD}, p\text{Addr}, v\text{Addr}, \text{DATA}) \\
GPR[rt] &\leftarrow \text{memword} \\
\text{LLbit} &\leftarrow 1
\end{align*}
\]
Exceptions:
TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Watch, Coprocessor Unusable

Programming Notes:
Load Upper Immediate

**Format:** \texttt{LUI \textit{rt}, immediate}

**Purpose:** Load Upper Immediate

To load a constant into the upper half of a word

**Description:** \texttt{GPR[rt] \leftarrow immediate \mid \mid 0^{16}}

The 16-bit \textit{immediate} is shifted left 16 bits and concatenated with 16 bits of low-order zeros. The 32-bit result is placed into \texttt{GPR rt}.

**Restrictions:**

None

**Operation:**

\texttt{GPR[rt] \leftarrow immediate \mid \mid 0^{16}}

**Exceptions:**

None
Load Doubleword Indexed Unaligned to Floating Point

**Purpose:** Load Doubleword Indexed Unaligned to Floating Point

To load a doubleword from memory to an FPR (GPR+GPR addressing), ignoring alignment

**Description:**
\[ \text{FPR}[fd] \leftarrow \text{memory}[(\text{GPR}[base] + \text{GPR}[index])_{PSIZE-1..3}] \]

The contents of the 64-bit doubleword at the memory location specified by the effective address are fetched and placed into the low word of FPR \( fd \). The contents of GPR \( index \) and GPR \( base \) are added to form the effective address. The effective address is doubleword-aligned; EffectiveAddress\( _{2..0} \) are ignored.

**Restrictions:**
The result of this instruction is **UNPREDICTABLE** if the processor is executing in the \( FR=0 \) 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the \( FR=1 \) mode, but not with \( FR=0 \), and not on a 32-bit FPU.

**Operation:**
\[
\begin{align*}
\text{vAddr} & \leftarrow (\text{GPR}[base]+\text{GPR}[index])_{63..3} || 0^3 \\
(pAddr, CCA) & \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \\
paddr & \leftarrow paddr \ xor ((\text{BigEndianCPU xor ReverseEndian}) || 0^2) \\
\text{memlsw} & \leftarrow \text{LoadMemory(CCA, WORD, pAddr, vAddr, DATA)} \\
paddr & \leftarrow paddr \ xor 0b100 \\
\text{memmsw} & \leftarrow \text{LoadMemory(CCA, WORD, pAddr, vAddr+4, DATA)} \\
\text{memdoubleword} & \leftarrow \text{memmsw} || \text{memlsw} \\
\text{StoreFPR(ft, UNINTERPRETED_DOUBLEWORD, memdoubleword)} 
\end{align*}
\]

**Exceptions:**
Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, Watch
Load Word

**Format:** \( \text{LW } rt, \text{ offset}(\text{base}) \)

**Purpose:** Load Word

To load a word from memory as a signed value

**Description:** \( \text{GPR}[rt] \leftarrow \text{memory}[\text{GPR}[\text{base}] + \text{offset}] \)

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR \( rt \). The 16-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form the effective address.

**Restrictions:**

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

**Operation:**

\[
\text{vAddr} \leftarrow \text{sign\_extend} (\text{offset}) + \text{GPR}[\text{base}]
\]

\[
\text{if } \text{vAddr}_{1..0} \neq 0\text{ then}
\]

\[
\text{SignalException(AddressError)}
\]

\[
\text{endif}
\]

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

\[
\text{memword} \leftarrow \text{LoadMemory (CCA, WORD, pAddr, vAddr, DATA)}
\]

\[
\text{GPR}[rt] \leftarrow \text{memword}
\]

**Exceptions:**

TLB Refill, TLB Invalid, Bus Error, Address Error, Watch
**Load Word to Floating Point**

**LWC1**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWC1</td>
<td>base</td>
<td>ft</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110001</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** LWC1 ft, offset(base)  

**Purpose:** Load Word to Floating Point  
To load a word from memory to an FPR

**Description:** FPR[ft] ← memory[GPR[base] + offset]  
The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed into the low word of FPR ft. If FPRs are 64 bits wide, bits 63..32 of FPR ft become UNPREDICTABLE. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

**Restrictions:**  
An Address Error exception occurs if EffectiveAddress1..0 ≠ 0 (not word-aligned).

**Operation:**

```
  vAddr ← sign_extend(offset) + GPR[base]
  if vAddr1..0 ≠ 0 then
      SignalException(AddressError)
  endif
  (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
  memword ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
  StoreFPR(ft, UNINTERPRETED_WORD, memword)
```

**Exceptions:**  
TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Coprocessor Unusable, Watch
Format: LWC2 rt, offset(base)

Purpose: Load Word to Coprocessor 2

To load a word from memory to a COP2 register

Description: CPR[2,rt,0] ← memory[GPR[base] + offset]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed into the low word of COP2 (Coprocessor 2) general register rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

Restrictions:
An Address Error exception occurs if EffectiveAddress1:0 ≠ 0 (not word-aligned).

Operation:

\[
\text{vAddr} \leftarrow \text{sign\_extend(offset)} + \text{GPR[base]}
\]

if vAddr12..0 ≠ 0 then
  SignalException(AddressError)
endif

(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)

memword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)

CPR[2,rt,0] ← memword

Exceptions:

TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Coprocessor Unusable, Watch
Format: \texttt{LWE \textit{rt}, \textit{offset}\{\textit{base}\}}

\textbf{Purpose:} Load Word EVA

To load a word from user mode virtual address space when executing in kernel mode.

\textbf{Description:} \texttt{GPR[rt]  \leftarrow \text{memory[GPR[base] + offset]}}

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR \textit{rt}. The 9-bit signed \textit{offset} is added to the contents of GPR \textit{base} to form the effective address.

The LWE instruction functions in exactly the same fashion as the LW instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the \textit{Config5_{EVA}} field being set to one.

\textbf{Restrictions:}

Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

\textbf{Operation:}

\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR[base]} \\
(p\text{Addr}, \text{CCA}) & \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \\
\text{memword} & \leftarrow \text{LoadMemory (CCA, WORD, p\text{Addr}, v\text{Addr}, DATA)} \\
\text{GPR[rt]} & \leftarrow \text{memword}
\end{align*}

\textbf{Exceptions:}

TLB Refill
TLB Invalid
Bus Error
Address Error
Watch
Reserved Instruction
Coprocessor Unusable
Load Word Left

LWL

31 26 25 21 20 16 15 0

LWL 100010

base rt offset

Format: LWL rt, offset(base) MIP32

Purpose: Load Word Left

To load the most-significant part of a word as a signed value from an unaligned memory address.

Description: GPR[rt] ← GPR[rt] MERGE memory[GPR[base] + offset]

The 16-bit signed offset is added to the contents of GPR base to form an effective address (EffAddr). EffAddr is the address of the most-significant of 4 consecutive bytes forming a word (W) in memory starting at an arbitrary byte boundary.

The most-significant 1 to 4 bytes of W is in the aligned word containing the EffAddr. This part of W is loaded into the most-significant (left) part of the word in GPR rt. The remaining least-significant part of the word in GPR rt is unchanged.

The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of W, 2 bytes, is in the aligned word containing the most-significant byte at 2. First, LWL loads these 2 bytes into the left part of the destination register word and leaves the right part of the destination word unchanged. Next, the complementary LWR loads the remainder of the unaligned word.

Figure 3.4 Unaligned Word Load Using LWL and LWR

Word at byte 2 in big-endian memory; each memory byte contains its own address

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>
|  e | f | g | h |   |   | GPR 24 Initial contents
| 2 | 3 | 4 | 5 |

Memory initial contents

After executing LWL $24, 2 ($0)

Then after LWR $24, 5 ($0)

The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address (vAddr_{1,0}), and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded for every combination of offset and byte ordering.
Figure 3.5 Bytes Loaded by LWL Instruction

<table>
<thead>
<tr>
<th>Memory contents and byte offsets</th>
<th>Initial contents of Dest Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 ← big-endian</td>
<td>e f g h</td>
</tr>
<tr>
<td>3 2 1 0 ← little-endian</td>
<td>most — significance — least</td>
</tr>
</tbody>
</table>

Destination register contents after instruction (shaded is unchanged)

<table>
<thead>
<tr>
<th>Big-endian</th>
<th>vAddr1,0</th>
<th>Little-endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>I J K L</td>
<td>0</td>
<td>L f g h</td>
</tr>
<tr>
<td>J K L h</td>
<td>1</td>
<td>K L g h</td>
</tr>
<tr>
<td>K L g h</td>
<td>2</td>
<td>J K L h</td>
</tr>
<tr>
<td>L f g h</td>
<td>3</td>
<td>I J K L</td>
</tr>
</tbody>
</table>

Restrictions:
None

Operation:
\[
vAddr \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}]
\]
\[
\text{pAddr} \leftarrow \text{AddressTranslation}(\text{vAddr}, \text{DATA}, \text{LOAD})
\]
\[
\text{pAddr} \leftarrow \text{pAddr}_{\text{PSIZE}-1..2} \ || \ (\text{pAddr}_{1..0} \ xor \ \text{ReverseEndian}^2)
\]
if BigEndianMem = 0 then
\[
\text{pAddr} \leftarrow \text{pAddr}_{\text{PSIZE}-1..2} \ || \ 0^2
\]
endif
\[
\text{byte} \leftarrow \text{vAddr}_{1..0} \ xor \ \text{BigEndianCPU}^2
\]
\[
\text{memword} \leftarrow \text{LoadMemory(CCA, byte, pAddr, vAddr, DATA)}
\]
\[
\text{temp} \leftarrow \text{memword}_{7+8 \times \text{byte}..0} \ || \ \text{GPR}[\text{rt}]_{23-8 \times \text{byte}..0}
\]
\[
\text{GPR}[\text{rt}] \leftarrow \text{temp}
\]

Exceptions:
None
TLB Refill, TLB Invalid, Bus Error, Address Error, Watch

Programming Notes:
The architecture provides no direct support for treating unaligned words as unsigned values, that is, zeroing bits 63..32 of the destination register when bit 31 is loaded.

Historical Information:
In the MIPS I architecture, the LWL and LWR instructions were exceptions to the load-delay scheduling restriction. A LWL or LWR instruction which was immediately followed by another LWL or LWR instruction, and used the same destination register would correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction. All such restrictions were removed from the architecture in MIPS II.
Load Word Left EVA

<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>7</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>011</td>
<td>111</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td>0</td>
<td>LWLE</td>
<td>011001</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \( \text{LWLE } rt, \text{ offset(base)} \)

**Purpose:** Load Word Left EVA

To load the most-significant part of a word as a signed value from an unaligned user mode virtual address while executing in kernel mode.

**Description:** \( \text{GPR}[rt] \leftarrow \text{GPR}[rt] \text{ MERGE memory}[\text{GPR[base]} + \text{offset}] \)

The 9-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form an effective address (\( \text{EffAddr} \)). \( \text{EffAddr} \) is the address of the most-significant of 4 consecutive bytes forming a word \( W \) in memory starting at an arbitrary byte boundary.

The most-significant 1 to 4 bytes of \( W \) is in the aligned word containing the \( \text{EffAddr} \). This part of \( W \) is loaded into the most-significant (left) part of the word in GPR \( rt \). The remaining least-significant part of the word in GPR \( rt \) is unchanged.

The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2.5 form an unaligned word starting at location 2. A part of \( W \), 2 bytes, is in the aligned word containing the most-significant byte at 2. First, LWLE loads these 2 bytes into the left part of the destination register word and leaves the right part of the destination word unchanged. Next, the complementary LWRE loads the remainder of the unaligned word.

**Figure 3.6 Unaligned Word Load Using LWLE and LWRE**

<table>
<thead>
<tr>
<th>Word at byte 2 in big-endian memory; each memory byte contains its own address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most: 0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>Memory initial contents</td>
</tr>
</tbody>
</table>

- GPR 24 Initial contents
- After executing LWLE \( \$24, 2 (0) \)
- Then after LWRE \( \$24, 5 (0) \)

The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address \( \text{vAddr}_{10} \), and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded for every combination of offset and byte ordering.

The LWLE instruction functions in exactly the same fashion as the LWL instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the \( \text{Config}_{EVA} \) field being set to one.
Restrictions:
Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Operation:
\[
\text{vAddr} \leftarrow \text{sign}_{-}\text{extend}(\text{offset}) + \text{GPR}\{\text{base}\}
\]
\[
(\text{pAddr, CCA}) \leftarrow \text{AddressTranslation} (\text{vAddr, DATA, LOAD})
\]
\[
\text{pAddr} \leftarrow \text{pAddr}_{\text{PSIZE}-1..2} || (\text{pAddr}_{1..0} \text{xor ReverseEndian}^2)
\]
if BigEndianMem = 0 then
\[
\text{pAddr} \leftarrow \text{pAddr}_{\text{PSIZE}-1..2} || 0^2
\]
endif
\[
\text{byte} \leftarrow \text{vAddr}_{1..0} \text{xor BigEndianCPU}^2
\]
\[
\text{memword} \leftarrow \text{LoadMemory} (\text{CCA, byte, pAddr, vAddr, DATA})
\]
\[
\text{temp} \leftarrow \text{memword}_{7+8*\text{byte}..0} || \text{GPR}\{\text{rt}\}_{23-8*\text{byte}..0}
\]
\[
\text{GPR}\{\text{rt}\} \leftarrow \text{temp}
\]

Exceptions:
TLB Refill, TLB Invalid, Bus Error, Address Error, Watch, Reserved Instruction, Coprocessor Unusable

Programming Notes:
The architecture provides no direct support for treating unaligned words as unsigned values, that is, zeroing bits 63..32 of the destination register when bit 31 is loaded.

Historical Information:
In the MIPS I architecture, the LWL and LWR instructions were exceptions to the load-delay scheduling restriction. A LWL or LWR instruction which was immediately followed by another LWL or LWR instruction, and used the same destination register would correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction. All such restrictions were removed from the architecture in MIPS II.
Load Word Right (LWR)

Format: LWR rt, offset(base)

Purpose: Load Word Right
To load the least-significant part of a word from an unaligned memory address as a signed value

Description: GPR[rt] ← GPR[rt] MERGE memory[GPR[base] + offset]

The 16-bit signed offset is added to the contents of GPR base to form an effective address (EffAddr). EffAddr is the address of the least-significant of 4 consecutive bytes forming a word (W) in memory starting at an arbitrary byte boundary.

A part of W, the least-significant 1 to 4 bytes, is in the aligned word containing EffAddr. This part of W is loaded into the least-significant (right) part of the word in GPR rt. The remaining most-significant part of the word in GPR rt is unchanged.

Executing both LWR and LWL, in either order, delivers a sign-extended word value in the destination register.

The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of W, 2 bytes, is in the aligned word containing the least-significant byte at 5. First, LWR loads these 2 bytes into the right part of the destination register. Next, the complementary LWL loads the remainder of the unaligned word.

Figure 3.8 Unaligned Word Load Using LWL and LWR

The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address (vAddr[1:0]), and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded for every combination of offset and byte ordering.
### Restrictions:
None

### Operation:

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
(p\text{Addr}, \text{CCA}) & \leftarrow \text{AddressTranslation}(\text{vAddr}, \text{DATA}, \text{LOAD}) \\
p\text{Addr} & \leftarrow p\text{Addr}_{\text{PSIZE}-1..2} \|| (p\text{Addr}_{1..0} \oplus \text{ReverseEndian}^2) \\
\text{if} \ \text{BigEndianMem} = 0 \ \text{then} \\
p\text{Addr} & \leftarrow p\text{Addr}_{\text{PSIZE}-1..2} \|| 0^2 \\
\text{endif} \\
\text{byte} & \leftarrow \text{vAddr}_{1..0} \oplus \text{BigEndianCPU}^2 \\
\text{memword} & \leftarrow \text{LoadMemory}(\text{CCA}, \text{byte}, p\text{Addr}, \text{vAddr}, \text{DATA}) \\
\text{temp} & \leftarrow \text{memword}_{31..32-8^*\text{byte}} \|| \text{GPR}[rt]_{31-8^*\text{byte}.0} \\
\text{GPR}[rt] & \leftarrow \text{temp}
\end{align*}
\]

### Exceptions:
TLB Refill, TLB Invalid, Bus Error, Address Error, Watch

### Programming Notes:
The architecture provides no direct support for treating unaligned words as unsigned values, that is, zeroing bits 63..32 of the destination register when bit 31 is loaded.

### Historical Information:
In the MIPS I architecture, the LWL and LWR instructions were exceptions to the load-delay scheduling restriction. A LWL or LWR instruction which was immediately followed by another LWL or LWR instruction, and used the same destination register would correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction. All such restrictions were removed from the architecture in MIPS II.
Load Word Right EVA (LWRE)

Format: \texttt{LWRE \text{rt}, offset(base)}

Purpose: Load Word Right EVA

To load the least-significant part of a word from an unaligned user mode virtual memory address as a signed value while executing in kernel mode.

Description: \texttt{GPR[rt] \leftarrow GPR[rt] \text{MERGE} \text{memory}[GPR[base] + \text{offset}]}

The 9-bit signed \texttt{offset} is added to the contents of GPR \texttt{base} to form an effective address \texttt{(EffAddr)}. \texttt{EffAddr} is the address of the least-significant of 4 consecutive bytes forming a word \texttt{(W)} in memory starting at an arbitrary byte boundary.

A part of \texttt{W}, the least-significant 1 to 4 bytes, is in the aligned word containing \texttt{EffAddr}. This part of \texttt{W} is loaded into the least-significant (right) part of the word in GPR \texttt{rt}. The remaining most-significant part of the word in GPR \texttt{rt} is unchanged.

Executing both LWRE and LWLE, in either order, delivers a sign-extended word value in the destination register.

The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2.5 form an unaligned word starting at location 2. A part of \texttt{W}, 2 bytes, is in the aligned word containing the least-significant byte at 5. First, LWRE loads these 2 bytes into the right part of the destination register. Next, the complementary LWLE loads the remainder of the unaligned word.

The LWRE instruction functions in exactly the same fashion as the LWR instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the \texttt{Config5EVA} field being set to one.

\textbf{Figure 3.10 Unaligned Word Load Using LWLE and LWRE}

The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address \texttt{(vAddr\_1,0)}, and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded for every combination of offset and byte ordering.
Figure 3.11 Bytes Loaded by LWRE Instruction

<table>
<thead>
<tr>
<th>Memory contents and byte offsets</th>
<th>Initial contents of Dest Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 ←big-endian</td>
<td>e f g h ←little-endian</td>
</tr>
<tr>
<td>offset (vAddr1..0)</td>
<td>most</td>
</tr>
<tr>
<td>most</td>
<td>least</td>
</tr>
<tr>
<td>Destination register contents after instruction (shaded is unchanged)</td>
<td></td>
</tr>
<tr>
<td>Big-endian</td>
<td>vAddr1..0</td>
</tr>
<tr>
<td>e f g l</td>
<td>0</td>
</tr>
<tr>
<td>e f i J</td>
<td>1</td>
</tr>
<tr>
<td>e i J K</td>
<td>2</td>
</tr>
<tr>
<td>i J K l</td>
<td>3</td>
</tr>
</tbody>
</table>

Restrictions:

Operation:

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReverseEndian^2)
if BigEndianMem = 0 then
  pAddr ← pAddrPSIZE-1..2 || 0^2
endif
byte ← vAddr1..0 xor BigEndianCPU^2
memword ← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
temp ← memword31..32−8*byte || GPR[rt]31−8*byte..0
GPR[rt] ← temp

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error, Watch, Reserved Instruction, Coprocessor Unusable

Programming Notes:

The architecture provides no direct support for treating unaligned words as unsigned values, that is, zeroing bits 63..32 of the destination register when bit 31 is loaded.

Historical Information:

In the MIPS I architecture, the LWL and LWR instructions were exceptions to the load-delay scheduling restriction. A LWL or LWR instruction which was immediately followed by another LWL or LWR instruction, and used the same destination register would correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction. All such restrictions were removed from the architecture in MIPS II.
Load Word Indexed to Floating Point

**Format:** LWXC1 fd, index(base)

**Purpose:** Load Word Indexed to Floating Point

To load a word from memory to an FPR (GPR+GPR addressing)

**Description:** FPR[fd] ← memory[GPR[base] + GPR[index]]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed into the low word of FPR fd. If FPRs are 64 bits wide, bits 63..32 of FPR fs become UNPREDICTABLE. The contents of GPR index and GPR base are added to form the effective address.

**Restrictions:**

An Address Error exception occurs if EffectiveAddress1..0 ≠ 0 (not word-aligned).

**Compatibility and Availability:**

LWXC1: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode (FIR F64=0 or 1, Status FR=0 or 1).

**Operation:**

\[
\text{vAddr} \leftarrow \text{GPR[base]} + \text{GPR[index]}
\]

if vAddr1..0 ≠ 0² then

  SignalException(AddressError)
endif

(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)

memword ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)

StoreFPR(fd, UNINTERPRETED_WORD, memword)

**Exceptions:**

TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Coprocessor Unusable, Watch
Multiply and Add Word to Hi, Lo

**MADD**

**Format:**

```
MADD rs, rt
```

**Purpose:** Multiply and Add Word to Hi, Lo

To multiply two words and add the result to Hi, Lo

**Description:**

\[(\text{HI,LO}) \leftarrow (\text{HI,LO}) + (\text{GPR[rs]} \times \text{GPR[rt]})\]

The 32-bit word value in GPR rs is multiplied by the 32-bit word value in GPR rt, treating both operands as signed values, to produce a 64-bit result. The product is added to the 64-bit concatenated values of HI and LO. The most significant 32 bits of the result are written into HI and the least significant 32 bits are written into LO. No arithmetic exception occurs under any circumstances.

**Restrictions:**

None

This instruction does not provide the capability of writing directly to a target GPR.

**Operation:**

```plaintext
temp \leftarrow (\text{HI} || \text{LO}) + (\text{GPR[rs]} \times \text{GPR[rt]})
HI \leftarrow \text{temp}_{63..32}
LO \leftarrow \text{temp}_{31..0}
```

**Exceptions:**

None

**Programming Notes:**

Where the size of the operands are known, software should place the shorter operand in GPR rt. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
Floating Point Multiply Add  MADD.fmt

<table>
<thead>
<tr>
<th>Format:</th>
<th>MADD.fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MADD.S fd, fr, fs, ft</td>
</tr>
<tr>
<td></td>
<td>MADD.D fd, fr, fs, ft</td>
</tr>
<tr>
<td></td>
<td>MADD.PS fd, fr, fs, ft</td>
</tr>
<tr>
<td>Purpose:</td>
<td>Floating Point Multiply Add</td>
</tr>
<tr>
<td>To perform a combined multiply-then-add of FP values</td>
<td></td>
</tr>
<tr>
<td>Description:</td>
<td>FPR[fd] ← (FPR[fs] × FPR[ft]) + FPR[fr]</td>
</tr>
<tr>
<td>The value in FPR fs is multiplied by the value in FPR ft to produce an intermediate product.</td>
<td></td>
</tr>
<tr>
<td>The intermediate product is rounded according to the current rounding mode in FCSR. The value in FPR fr is added to the product. The result sum is calculated to infinite precision, rounded according to the current rounding mode in FCSR, and placed into FPR fd. The operands and result are values in format fmt. The results and flags are as if separate floating-point multiply and add instructions were executed.</td>
<td></td>
</tr>
<tr>
<td>MADD.PS multiplies then adds the upper and lower halves of FPR fr, FPR fs, and FPR ft independently, and ORs together any generated exceptional conditions.</td>
<td></td>
</tr>
<tr>
<td>Cause bits are ORed into the Flag bits if no exception is taken.</td>
<td></td>
</tr>
<tr>
<td>Restrictions:</td>
<td></td>
</tr>
<tr>
<td>The fields fr, fs, ft, and fd must specify FPRs valid for operands of type fmt; if they are not valid, the result is UNPREDICTABLE.</td>
<td></td>
</tr>
<tr>
<td>The operands must be values in format fmt; if they are not, the result is UNPREDICTABLE and the value of the operand FPRs becomes UNPREDICTABLE.</td>
<td></td>
</tr>
<tr>
<td>The result of MADD.PS is UNPREDICTABLE if the processor is executing in the FR=0 32-bit FPU register model; i.e. it is predictable if executing on a 64-bit FPU in the FR=1 mode, but not with FR=0, and not on a 32-bit FPU.</td>
<td></td>
</tr>
<tr>
<td>Compatibility and Availability:</td>
<td></td>
</tr>
<tr>
<td>MADD.S and MADD.D: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode (FIR F64=0 or 1, Status FR=0 or 1).</td>
<td></td>
</tr>
<tr>
<td>Operation:</td>
<td></td>
</tr>
<tr>
<td>vfr ← ValueFPR(fr, fmt)</td>
<td></td>
</tr>
<tr>
<td>vfs ← ValueFPR(fs, fmt)</td>
<td></td>
</tr>
<tr>
<td>vft ← ValueFPR(ft, fmt)</td>
<td></td>
</tr>
<tr>
<td>StoreFPR(fd, fmt, (vfs × fmt vft) + fmt vfr)</td>
<td></td>
</tr>
<tr>
<td>Exceptions:</td>
<td></td>
</tr>
<tr>
<td>Coprocessor Unusable, Reserved Instruction</td>
<td></td>
</tr>
<tr>
<td>Floating Point Exceptions:</td>
<td></td>
</tr>
<tr>
<td>Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow</td>
<td></td>
</tr>
</tbody>
</table>

MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set, Revision 5.04 188
Multiply and Add Unsigned Word to Hi,Lo

**MADDU**

### Format:

\[ \text{MADDU } rs, rt \]

### Purpose:
Multiply and Add Unsigned Word to Hi,Lo

To multiply two unsigned words and add the result to \( HI \), \( LO \).

### Description:

\[ (HI, LO) \leftarrow (HI, LO) + (GPR[rs] \times GPR[rt]) \]

The 32-bit word value in GPR \( rs \) is multiplied by the 32-bit word value in GPR \( rt \), treating both operands as unsigned values, to produce a 64-bit result. The product is added to the 64-bit concatenated values of \( HI \) and \( LO \). The most significant 32 bits of the result are written into \( HI \) and the least significant 32 bits are written into \( LO \). No arithmetic exception occurs under any circumstances.

### Restrictions:
None

This instruction does not provide the capability of writing directly to a target GPR.

### Operation:

\[
\begin{align*}
\text{temp} & \leftarrow (HI \ || \ LO) + (GPR[rs] \times GPR[rt]) \\
HI & \leftarrow \text{temp}_{63..32} \\
LO & \leftarrow \text{temp}_{31..0}
\end{align*}
\]

### Exceptions:
None

### Programming Notes:

Where the size of the operands are known, software should place the shorter operand in GPR \( rt \). This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
Move from Coprocessor 0

**Format:**
MFC0 rt, rd
MFC0 rt, rd, sel

**Purpose:** Move from Coprocessor 0
To move the contents of a coprocessor 0 register to a general register.

**Description:**
GPR[rt] ← CPR[0, rd, sel]

The contents of the coprocessor 0 register specified by the combination of rd and sel are loaded into general register rt. Note that not all coprocessor 0 registers support the sel field. In those instances, the sel field must be zero.

**Restrictions:**
The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by rd and sel.

**Operation:**
```
reg = rd
data ← CPR[0, reg, sel]
GPR[rt] ← data
```

**Exceptions:**
Coprocessor Unusable
Reserved Instruction
Move Word From Floating Point

**Format:**  
MFC1 rt, fs

**Purpose:** Move Word From Floating Point
To copy a word from an FPU (CP1) general register to a GPR

**Description:**  
GPR[rt] ← FPR[fs]  
The contents of FPR fs are loaded into general register rt.

**Restrictions:**

**Operation:**

```
data ← ValueFPR(fs, UNINTERPRETED_WORD)  
GPR[rt] ← data
```

**Exceptions:**  
Coprocessor Unusable, Reserved Instruction

**Historical Information:**
For MIPS I, MIPS II, and MIPS III the contents of GPR rt are **UNPREDICTABLE** for the instruction immediately following MFC1.
Move Word From Coprocessor 2

**Format:**  
MFC2 rt, Impl  
MIPS32  
MFC2, rt, Impl, sel  
MIPS32

The syntax shown above is an example using MFC1 as a model. The specific syntax is implementation dependent.

**Purpose:** Move Word From Coprocessor 2  
To copy a word from a COP2 general register to a GPR

**Description:**  
GPR[rt] ← CP2CPR[Impl]  
The contents of the coprocessor 2 register denoted by the **Impl** field are and placed into general register **rt**. The interpretation of the **Impl** field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

**Restrictions:**  
The results are **UNPREDICTABLE** if **Impl** specifies a coprocessor 2 register that does not exist.

**Operation:**  
\[
\text{data} \leftarrow \text{CP2CPR[Impl]}
\]
\[
\text{GPR[rt]} \leftarrow \text{data}
\]

**Exceptions:**  
Coprocessor Unusable
Move Word From Coprocessor 2
Move from High Coprocessor 0

Purpose: Move from High Coprocessor 0

To move the contents of the upper 32 bits of a Coprocessor 0 register, extended by 32-bits, to a general register.

Description: $\text{GPR}[rt] \leftarrow \text{CPR}[0,rd,sel][63:32]$

The contents of the Coprocessor 0 register specified by the combination of $rd$ and $sel$ are loaded into general register $rt$. Note that not all Coprocessor 0 registers support the $sel$ field, and in those instances, the $sel$ field must be zero.

Restrictions:

The results are UNDEFINED if Coprocessor 0 does not contain a register as specified by $rd$ and $sel$, or the register exists but is not extended by 32-bits, or the register is extended for XPA, but XPA is not supported or enabled.

Operation:

```plaintext
reg \leftarrow rd
data \leftarrow \text{CPR}[0,reg,sel]$

\text{GPR}[rt] \leftarrow \text{data}_{63..32}$
```

Exceptions:

Coprocessor Unusable

Reserved Instruction
Move Word From High Half of Floating Point Register

**Format:** MFHC1 rt, fs

**Purpose:** Move Word From High Half of Floating Point Register

To copy a word from the high half of an FPU (CP1) general register to a GPR

**Description:**

GPR[rt] ← FPR[fs]63..32

The contents of the high word of FPR fs are loaded into general register rt. This instruction is primarily intended to support 64-bit floating point units on a 32-bit CPU, but the semantics of the instruction are defined for all cases.

**Restrictions:**

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

The results are UNPREDICTABLE if Status_{FR} = 0 and fs is odd.

**Operation:**

\[
\text{data} \leftarrow \text{ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)}_{63..32} \\
\text{GPR[rt]} \leftarrow \text{data}
\]

**Exceptions:**

Coprocessor Unusable

Reserved Instruction
Move Word From High Half of Coprocessor 2 Register  

**MFHC2**

The syntax shown above is an example using MFHC1 as a model. The specific syntax is implementation dependent.

**Purpose:** Move Word From High Half of Coprocessor 2 Register

To copy a word from the high half of a COP2 general register to a GPR

**Description:**

\[ \text{GPR}[rt] \leftarrow \text{CP2CPR}[\text{Impl}]_{63..32} \]

The contents of the high word of the coprocessor 2 register denoted by the \( \text{Impl} \) field are placed into GPR \( rt \). The interpretation of the \( \text{Impl} \) field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

**Restrictions:**

The results are **UNPREDICTABLE** if \( \text{Impl} \) specifies a coprocessor 2 register that does not exist, or if that register is not 64 bits wide.

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

**Operation:**

\[
\begin{align*}
data & \leftarrow \text{CP2CPR}[\text{Impl}]_{63..32} \\
\text{GPR}[rt] & \leftarrow data
\end{align*}
\]

**Exceptions:**

Coprocessor Unusable

Reserved Instruction
Move From HI Register

**Move From HI Register IMFHI**

**Format:**

```
        31 26 25 16 15 11 10 6 5 0
     +--------------------+
SPECIAL 000000
     |                   |
     |                   |
     |                   |
     +--------------------+
     | 0                 |
     | 00 0000 0000      |
     |                   |
     +--------------------+
     | rd                |
     | 5                 |
     | 5                 |
     +--------------------+
     | 0                 |
     | 00000             |
     |                   |
     | MFHI              |
     | 010000            |
```

**Purpose:** Move From HI Register

To copy the special purpose HI register to a GPR

**Description:**

\[ \text{GPR}[\text{rd}] \leftarrow \text{HI} \]

The contents of special register HI are loaded into GPR rd.

**Restrictions:**

None

**Operation:**

\[ \text{GPR}[\text{rd}] \leftarrow \text{HI} \]

**Exceptions:**

None

**Historical Information:**

In the MIPS I, II, and III architectures, the two instructions which follow the MFHI must not modify the HI register. If this restriction is violated, the result of the MFHI is UNPREDICTABLE. This restriction was removed in MIPS IV and MIPS32, and all subsequent levels of the architecture.
Move From LO Register

**Format:**  MFLO rd

**Purpose:** Move From LO Register

To copy the special purpose LO register to a GPR

**Description:**  GPR[rd] ← LO

The contents of special register LO are loaded into GPR rd.

**Restrictions:**

None

**Operation:**

GPR[rd] ← LO

**Exceptions:**

None

**Historical Information:**

In the MIPS I, II, and III architectures, the two instructions which follow the MFLO must not modify the HI register. If this restriction is violated, the result of the MFLO is **UNPREDICTABLE**. This restriction was removed in MIPS IV and MIPS32, and all subsequent levels of the architecture.
Floating Point Move

**Format:** MOV.fmt
- MOV.S fd, fs
- MOV.D fd, fs
- MOV.PS fd, fs
  
  **Purpose:** Floating Point Move
  
  To move an FP value between FPRs

  **Description:** FPR[fd] ← FPR[fs]
  
  The value in FPR fs is placed into FPR fd. The source and destination are values in format fmt. In paired-single format, both the halves of the pair are copied to fd.
  
  The move is non-arithmetic; it causes no IEEE 754 exceptions.

  **Restrictions:**
  
  The fields fs and fd must specify FPRs valid for operands of type fmt; if they are not valid, the result is UNPREDICTABLE.
  
  The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.
  
  The result of MOV.PS is UNPREDICTABLE if the processor is executing in the FR=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the FR=1 mode, but not with FR=0, and not on a 32-bit FPU.

  **Operation:**
  
  StoreFPR(fd, fmt, ValueFPR(fs, fmt))

  **Exceptions:**
  
  Coprocessor Unusable, Reserved Instruction

  **Floating Point Exceptions:**
  
  Unimplemented Operation
Move Conditional on Floating Point False

**Format:** MOVF rd, rs, cc

**Purpose:** Move Conditional on Floating Point False
To test an FP condition code then conditionally move a GPR

**Description:** if FPConditionCode(cc) = 0 then GPR[rd] ← GPR[rs]
If the floating point condition code specified by CC is zero, then the contents of GPR rs are placed into GPR rd.

**Restrictions:**

**Operation:**

```
if FPConditionCode(cc) = 0 then
    GPR[rd] ← GPR[rs]
endif
```

**Exceptions:**
Reserved Instruction, Coprocessor Unusable
Floating Point Move Conditional on Floating Point False

MOVF.fmt

**Format:**

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>cc</th>
<th>tf</th>
<th>fs</th>
<th>fd</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>0100001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MOVF.S fd, fs, cc  
MOVF.D fd, fs, cc  
MOVF.PS fd, fs, cc

**MIPS32**  
**MIPS32 Release 2**  
**MIPS64**  

**Purpose:** Floating Point Move Conditional on Floating Point False

To test an FP condition code then conditionally move an FP value

**Description:** if FPConditionCode(cc) = 0 then FPR[fd] ← FPR[fs]

If the floating point condition code specified by CC is zero, then the value in FPR fs is placed into FPR fd. The source and destination are values in format fmt.

If the condition code is not zero, then FPR fs is not copied and FPR fd retains its previous value in format fmt. If fd did not contain a value either in format fmt or previously unused data from a load or move-to operation that could be interpreted in format fmt, then the value of fd becomes UNPREDICTABLE.

MOVF.PS conditionally merges the lower half of FPR fs into the lower half of FPR fd if condition code CC is zero, and independently merges the upper half of FPR fs into the upper half of FPR fd if condition code CC+1 is zero. The CC field must be even; if it is odd, the result of this operation is UNPREDICTABLE.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

The fields fs and fd must specify FPRs valid for operands of type fmt; if they are not valid, the result is UNPREDICTABLE. The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

The result of MOVF.PS is UNPREDICTABLE if the processor is executing in the FR=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the FR=1 mode, but not with FR=0, and not on a 32-bit FPU.

**Operation:**

```
if FPConditionCode(cc) = 0 then
  StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
  StoreFPR(fd, fmt, ValueFPR(fd, fmt))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation
Move Conditional on Not Zero

Format: MOVN rd, rs, rt

Purpose: Move Conditional on Not Zero
To conditionally move a GPR after testing a GPR value

Description: if GPR[rt] ≠ 0 then GPR[rd] ← GPR[rs]
If the value in GPR rt is not equal to zero, then the contents of GPR rs are placed into GPR rd.

Restrictions:
None

Operation:
if GPR[rt] ≠ 0 then
    GPR[rd] ← GPR[rs]
endif

Exceptions:
None

Programming Notes:
The non-zero value tested might be the condition true result from the SLT, SLTI, SLTU, and SLTIU comparison instructions or a boolean value read from memory.
Floating Point Move Conditional on Not Zero  

**Format:**  
MOVN.fmt  
MOVN.S fd, fs, rt  
MOVN.D fd, fs, rt  
MOVN.PS fd, fs, rt  

**Purpose:** Floating Point Move Conditional on Not Zero  
To test a GPR then conditionally move an FP value  

**Description:**  
if GPR[rt] ≠ 0 then FPR[fd] ← FPR[fs]  
If the value in GPR rt is not equal to zero, then the value in FPR fs is placed in FPR fd. The source and destination are values in format fmt.  
If GPR rt contains zero, then FPR fs is not copied and FPR fd contains its previous value in format fmt. If fd did not contain a value either in format fmt or previously unused data from a load or move-to operation that could be interpreted in format fmt, then the value of fd becomes UNPREDICTABLE.  
The move is non-arithmetic; it causes no IEEE 754 exceptions.  

**Restrictions:**  
The fields fs and fd must specify FPRs valid for operands of type fmt; if they are not valid, the result is UNPREDICTABLE.  
The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.  
The result of MOVN.PS is UNPREDICTABLE if the processor is executing in the FR=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the FR=1 mode, but not with FR=0, and not on a 32-bit FPU.  

**Operation:**  
if GPR[rt] ≠ 0 then  
  StoreFPR(fd, fmt, ValueFPR(fs, fmt))  
else  
  StoreFPR(fd, fmt, ValueFPR(fd, fmt))  
endif  

**Exceptions:**  
Coprocessor Unusable, Reserved Instruction  
Floating Point Exceptions:  
Unimplemented Operation
Move Conditional on Floating Point True

**Format:**

```
MOVT rd, rs, cc
```

**Purpose:** Move Conditional on Floating Point True

To test an FP condition code then conditionally move a GPR

**Description:**

\[ \text{if FPConditionCode}(cc) = 1 \text{ then } \text{GPR}[rd] \leftarrow \text{GPR}[rs] \]

If the floating point condition code specified by \( CC \) is one, then the contents of GPR \( rs \) are placed into GPR \( rd \).

**Restrictions:**

**Operation:**

\[
\text{if FPConditionCode}(cc) = 1 \text{ then }
\text{GPR}[rd] \leftarrow \text{GPR}[rs]
\text{endif}
\]

**Exceptions:**

Reserved Instruction, Coprocessor Unusable
Floating Point Move Conditional on Floating Point True

**MOVT.fmt**

**Format:**

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>cc</th>
<th>tf</th>
<th>fs</th>
<th>fd</th>
<th>MOVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>65 3 1 1 5 56</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Purpose:** Floating Point Move Conditional on Floating Point True

To test an FP condition code then conditionally move an FP value

**Description:** if FPConditionCode(cc) = 1 then FPR[fd] ← FPR[fs]

If the floating point condition code specified by $CC$ is one, then the value in FPR $fs$ is placed into FPR $fd$. The source and destination are values in format $fmt$.

If the condition code is not one, then FPR $fs$ is not copied and FPR $fd$ contains its previous value in format $fmt$. If $fd$ did not contain a value either in format $fmt$ or previously unused data from a load or move-to operation that could be interpreted in format $fmt$, then the value of $fd$ becomes UNPREDICTABLE.

MOVT.PS conditionally merges the lower half of FPR $fs$ into the lower half of FPR $fd$ if condition code $CC$ is one, and independently merges the upper half of FPR $fs$ into the upper half of FPR $fd$ if condition code $CC+1$ is one. The $CC$ field should be even; if it is odd, the result of this operation is UNPREDICTABLE.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

The fields $fs$ and $fd$ must specify FPRs valid for operands of type $fmt$; if they are not valid, the result is UNPREDICTABLE. The operand must be a value in format $fmt$; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

The result of MOVT.PS is UNPREDICTABLE if the processor is executing in the $FR=0$ 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the $FR=1$ mode, but not with $FR=0$, and not on a 32-bit FPU.

**Operation:**

```c
if FPConditionCode(cc) = 1 then
    StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
    StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation
Move Conditional on Zero

Format: MOVZ rd, rs, rt

Purpose: Move Conditional on Zero
To conditionally move a GPR after testing a GPR value

Description: if GPR[rt] = 0 then GPR[rd] ← GPR[rs]

If the value in GPR rt is equal to zero, then the contents of GPR rs are placed into GPR rd.

Restrictions:
None

Operation:

if GPR[rt] = 0 then
    GPR[rd] ← GPR[rs]
endif

Exceptions:
None

Programming Notes:
The zero value tested might be the condition false result from the SLT, SLTI, SLTU, and SLTIU comparison instructions or a boolean value read from memory.
Floating Point Move Conditional on Zero

Format: MOVZ.fmt

MOVD.S fd, fs, rt  
MOVD.D fd, fs, rt  
MOVD.PS fd, fs, rt  

Purpose: Floating Point Move Conditional on Zero

To test a GPR then conditionally move an FP value

Description: if GPR[rt] = 0 then FPR[fd] ← FPR[fs]

If the value in GPR rt is equal to zero then the value in FPR fs is placed in FPR fd. The source and destination are values in format fmt.

If GPR rt is not zero, then FPR fs is not copied and FPR fd contains its previous value in format fmt. If fd did not contain a value either in format fmt or previously unused data from a load or move-to operation that could be interpreted in format fmt, then the value of fd becomes UNPREDICTABLE.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The fields fs and fd must specify FPRs valid for operands of type fmt; if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

The result of MOVZ.PS is UNPREDICTABLE if the processor is executing in the FR=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the FR=1 mode, but not with FR=0, and not on a 32-bit FPU.

Operation:

if GPR[rt] = 0 then
    StoreFPR(fd, fmt, ValueFPR(fs, fmt))
else
    StoreFPR(fd, fmt, ValueFPR(fd, fmt))
endif

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation
### Multiply and Subtract Word to Hi,Lo

**Format:** MSUB rs, rt

**Purpose:** Multiply and Subtract Word to Hi,Lo

To multiply two words and subtract the result from HI, LO

**Description:** (HI,LO) ← (HI,LO) - (GPR[rs] × GPR[rt])

The 32-bit word value in GPR rs is multiplied by the 32-bit value in GPR rt, treating both operands as signed values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values of HI and LO. The most significant 32 bits of the result are written into HI and the least significant 32 bits are written into LO. No arithmetic exception occurs under any circumstances.

**Restrictions:**

None

This instruction does not provide the capability of writing directly to a target GPR.

**Operation:**

\[
\begin{align*}
\text{temp} & \leftarrow (\text{HI} \ || \ \text{LO}) - (\text{GPR}[rs] \times \text{GPR}[rt]) \\
\text{HI} & \leftarrow \text{temp}_{63..32} \\
\text{LO} & \leftarrow \text{temp}_{31..0}
\end{align*}
\]

**Exceptions:**

None

**Programming Notes:**

Where the size of the operands are known, software should place the shorter operand in GPR rt. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
Floating Point Multiply Subtract

**Format:**  
MSUB.fmt
- MSUB.S fd, fr, fs, ft
- MSUB.D fd, fr, fs, ft
- MSUB.PS fd, fr, fs, ft

**Purpose:** Floating Point Multiply Subtract

To perform a combined multiply-then-subtract of FP values

**Description:**  
FPR[fd] ← (FPR[fs] × FPR[ft]) − FPR[fr]

The value in FPR fs is multiplied by the value in FPR ft to produce an intermediate product. The intermediate product is rounded according to the current rounding mode in $FCSR$. The subtraction result is calculated to infinite precision, rounded according to the current rounding mode in $FCSR$, and placed into FPR fd. The operands and result are values in format $fmt$. The results and flags are as if separate floating-point multiply and subtract instructions were executed.

MSUB.PS multiplies then subtracts the upper and lower halves of FPR fr, FPR fs, and FPR ft independently, and ORs together any generated exceptional conditions.

*Cause* bits are ORed into the *Flag* bits if no exception is taken.

**Restrictions:**

The fields fr, fs, ft, and fd must specify FPRs valid for operands of type $fmt$; if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format $fmt$; if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

The result of MSUB.PS is **UNPREDICTABLE** if the processor is executing in the $FR=0$ 32-bit FPU register model; i.e. it is predictable if executing on a 64-bit FPU in the $FR=1$ mode, but not with $FR=0$, and not on a 32-bit FPU.

**Compatibility and Availability:**

MSUB.S and MSUB.D: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode ($FIR_{F64}=0$ or 1, $Status_{FR}=0$ or 1).

**Operation:**

\[
\begin{align*}
&vfr \leftarrow \text{ValueFPR}(fr, fmt) \\
&vfs \leftarrow \text{ValueFPR}(fs, fmt) \\
&vft \leftarrow \text{ValueFPR}(ft, fmt) \\
&\text{StoreFPR}(fd, fmt, (vfs \times_{f_{fmt}} vft) -_{f_{fmt}} vfr)
\end{align*}
\]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow
Multiply and Subtract Word to Hi,Lo

**MSUBU**

<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>SPECIAL2</td>
<td>011100</td>
<td>rs</td>
<td>rt</td>
<td>0</td>
<td>00000</td>
<td>0</td>
<td>00000</td>
<td>MSUBU</td>
<td>000101</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** MSUBU rs, rt

**Purpose:** Multiply and Subtract Word to Hi,Lo

To multiply two words and subtract the result from HI, LO

**Description:**

\[(\text{HI,LO}) \leftarrow (\text{HI,LO}) - (\text{GPR}[rs] \times \text{GPR}[rt])\]

The 32-bit word value in GPR rs is multiplied by the 32-bit word value in GPR rt, treating both operands as unsigned values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values of HI and LO. The most significant 32 bits of the result are written into HI and the least significant 32 bits are written into LO. No arithmetic exception occurs under any circumstances.

**Restrictions:**

None

This instruction does not provide the capability of writing directly to a target GPR.

**Operation:**

\[
\begin{align*}
\text{temp} & \leftarrow (\text{HI} | | \text{LO}) - (\text{GPR}[rs] \times \text{GPR}[rt]) \\
\text{HI} & \leftarrow \text{temp}_{63..32} \\
\text{LO} & \leftarrow \text{temp}_{31..0}
\end{align*}
\]

**Exceptions:**

None

**Programming Notes:**

Where the size of the operands are known, software should place the shorter operand in GPR rt. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
Move to Coprocessor 0

**Format:**

MTC0 rt, rd

MTC0 rt, rd, sel

**Purpose:** Move to Coprocessor 0

To move the contents of a general register to a coprocessor 0 register.

**Description:** CPR[0, rd, sel] ← GPR[rt]

The contents of general register rt are loaded into the coprocessor 0 register specified by the combination of rd and sel. Not all coprocessor 0 registers support the sel field. In those instances, the sel field must be set to zero.

In Release 5, for a 32-bit processor, the MTC0 instruction writes all zeroes to the high-order bits of selected COP0 registers that have been extended beyond 32 bits. This is required for compatibility with legacy software that does not use MTHC0, yet has hardware support for extended COP0 registers (such as for Extended Physical Addressing (XPA)). Because MTC0 overwrites the result of MTHC0, software must first read the high-order bits before writing the low-order bits, then write the high-order bits back either modified or unmodified. For initialization of an extended register, software may first write the low-order bits, then the high-order bits, without first reading the high-order bits.

**Restrictions:**

The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by rd and sel.

**Operation:**

\[
\text{data} \leftarrow \text{GPR}[\text{rt}]
\]

\[
\text{reg} \leftarrow \text{rd}
\]

if (Config5MVH = 1) then

// The most-significant bit may vary by register. Only supported
// bits should be written 0.
// Extended LLAddr is not written with 0s, as it is a read-only register.
// BadVAddr is not written with 0s, as it is read-only

if (Config3LPA = 1) then

if (reg, sel = EntryLo0 or EntryLo1) then CPR[0, reg, sel]_{63:32} = 0^{32}

if (reg, sel = MAAR) then CPR[0, reg, sel]_{63:32} = 0^{32}

// TagLo is zeroed only if the implementation-dependent bits are
// writeable

if (reg, sel = TagLo) then CPR[0, reg, sel]_{63:32} = 0^{32}

if (Config3VZ = 1) then

if (reg, sel = EntryHi) then CPR[0, reg, sel]_{63:32} = 0^{32}

endif

endif

endif

**Exceptions:**

Coprocessor Unusable

Reserved Instruction
Move Word to Floating Point

Format: \texttt{MTC1 rt, fs}

Purpose: Move Word to Floating Point
To copy a word from a GPR to an FPU (CP1) general register

Description: \( \text{FPR}[fs] \leftarrow \text{GPR}[rt] \)
The low word in GPR \( rt \) is placed into the low word of FPR \( fs \).

Restrictions:

Operation:

\[
\text{data} \leftarrow \text{GPR}[rt]_{31..0} \\
\text{StoreFPR}(fs, \text{UNINTERPRETED\_WORD}, \text{data})
\]

Exceptions:
Coprocessor Unusable

Historical Information:
For MIPS I, MIPS II, and MIPS III the value of FPR \( fs \) is \textbf{UNPREDICTABLE} for the instruction immediately following MTC1.
Move Word to Coprocessor 2

The syntax shown above is an example using MTC1 as a model. The specific syntax is implementation-dependent.

**Purpose:** Move Word to Coprocessor 2
To copy a word from a GPR to a COP2 general register

**Description:**
\[\text{CP2CPR}[\text{Impl}] \leftarrow \text{GPR}[rt]\]

The low word in GPR \(rt\) is placed into the low word of a Coprocessor 2 general register denoted by the \(\text{Impl}\) field. The interpretation of the \(\text{Impl}\) field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

**Restrictions:**
The results are **UNPREDICTABLE** if \(\text{Impl}\) specifies a Coprocessor 2 register that does not exist.

**Operation:**
\[
\text{data} \leftarrow \text{GPR}[rt] \\
\text{CP2CPR}[\text{Impl}] \leftarrow \text{data}
\]

**Exceptions:**
Coprocessor Unusable
Reserved Instruction
Move to High Coprocessor 0

Purpose: Move to High Coprocessor 0

To copy a word from a GPR to the upper 32 bits of a COP2 general register that has been extended by 32 bits.

Description: \( \text{CPR}[0, \text{rd}, \text{sel}][63:32] \leftarrow \text{GPR}[\text{rt}] \)

The contents of general register \( \text{rt} \) are loaded into the Coprocessor 0 register specified by the combination of \( \text{rd} \) and \( \text{sel} \). Not all Coprocessor 0 registers support the \( \text{sel} \) field, and when this is the case, the \( \text{sel} \) field must be set to zero.

Restrictions:

The results are \textbf{UNDEFINED} if Coprocessor 0 does not contain a register as specified by \( \text{rd} \) and \( \text{sel} \), or if the register exists but is not extended by 32 bits, or the register is extended for XPA, but XPA is not supported or enabled.

Operation:

\[
\begin{align*}
\text{data} & \leftarrow \text{GPR}[\text{rt}] \\
\text{reg} & \leftarrow \text{rd} \\
\text{CPR}[0, \text{reg}, \text{sel}][63:32] & \leftarrow \text{data}
\end{align*}
\]

Exceptions:

Coprocessor Unusable

Reserved Instruction
**Move Word to High Half of Floating Point Register**

**MTHC1**

**Format:**

\[
\text{MTHC1 \ rt, fs}
\]

**Purpose:** Move Word to High Half of Floating Point Register

To copy a word from a GPR to the high half of an FPU (CP1) general register

**Description:**

\[
\text{FPR[fs]}_{63..32} \leftarrow \text{GPR[rt]}
\]

The word in GPR \( rt \) is placed into the high word of FPR \( fs \). This instruction is primarily intended to support 64-bit floating point units on a 32-bit CPU, but the semantics of the instruction are defined for all cases.

**Restrictions:**

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

The results are \text{UNPREDICTABLE} if \( \text{Status}_{FR} = 0 \) and \( fs \) is odd.

**Operation:**

\[
\begin{align*}
\text{newdata} & \leftarrow \text{GPR[rt]} \\
\text{olddata} & \leftarrow \text{ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)}_{31..0} \\
\text{StoreFPR(fs, UNINTERPRETED_DOUBLEWORD, newdata | | olddata)}
\end{align*}
\]

**Exceptions:**

Coprocessor Unusable

Reserved Instruction

**Programming Notes**

When paired with MTC1 to write a value to a 64-bit FPR, the MTC1 must be executed first, followed by the MTHC1. This is because of the semantic definition of MTC1, which is not aware that software will be using an MTHC1 instruction to complete the operation, and sets the upper half of the 64-bit FPR to an **UNPREDICTABLE** value.
Move Word to High Half of Coprocessor 2 Register

**MTHC2**

<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>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP2</td>
<td>MTH</td>
<td>rt</td>
<td>Impl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010010</td>
<td>00111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

- MTHC2 rt, Impl
- MTHC2 rt, Impl, sel

The syntax shown above is an example using MTHC1 as a model. The specific syntax is implementation dependent.

**Purpose:** Move Word to High Half of Coprocessor 2 Register

To copy a word from a GPR to the high half of a COP2 general register

**Description:**

\[
\text{CP2CPR[Impl]}_{63..32} \leftarrow \text{GPR}[rt]
\]

The word in GPR \(rt\) is placed into the high word of coprocessor 2 general register denoted by the \(\text{Impl}\) field. The interpretation of the \(\text{Impl}\) field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

**Restrictions:**

The results are **UNPREDICTABLE** if \(\text{Impl}\) specifies a coprocessor 2 register that does not exist, or if that register is not 64 bits wide.

In implementations prior to Release 2 of the architecture, this instruction resulted in a **Reserved Instruction Exception**.

**Operation:**

\[
data \leftarrow \text{GPR}[rt] \\
\text{CP2CPR[Impl]} \leftarrow data \ || \ \text{CPR}[2, rd, sel]_{31..0}
\]

**Exceptions:**

- Coprocessor Unusable
- Reserved Instruction

**Programming Notes**

When paired with MTC2 to write a value to a 64-bit CPR, the MTC2 must be executed first, followed by the MTHC2. This is because of the semantic definition of MTC2, which is not aware that software will be using an MTHC2 instruction to complete the operation, and sets the upper half of the 64-bit CPR to a **UNPREDICTABLE** value.
### Move to HI Register

**Format:** MTHI rs

**Purpose:** Move to HI Register

To copy a GPR to the special purpose HI register

**Description:** HI ← GPR[rs]

The contents of GPR rs are loaded into special register HI.

**Restrictions:**

A computed result written to the HI/LO pair by DIV, DIVU,MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either HI or LO.

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:**

HI ← GPR[rs]

**Exceptions:**

None

**Historical Information:**

In MIPS I-III, if either of the two preceding instructions is MFHI, the result of that MFHI is UNPREDICTABLE. Reads of the HI or LO special register must be separated from any subsequent instructions that write to them by two or more instructions. In MIPS IV and later, including MIPS32 and MIPS64, this restriction does not exist.
Move to LO Register

Format: MTLO rs

Purpose: Move to LO Register
To copy a GPR to the special purpose LO register

Description: LO ← GPR[rs]
The contents of GPR rs are loaded into special register LO.

Restrictions:
A computed result written to the HI/LO pair by DIV, DIVU, MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either HI or LO.

If an MTLO instruction is executed following one of these arithmetic instructions, but before an MFLO or MFHI instruction, the contents of HI 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
MTLO r6 # code not containing mfhi
... MFHI r3 # this mfhi would get an UNPREDICTABLE value

Operation:
LO ← GPR[rs]

Exceptions:
None

Historical Information:
In MIPS I-III, if either of the two preceding instructions is MFHI, the result of that MFHI is UNPREDICTABLE. Reads of the HI or LO special register must be separated from any subsequent instructions that write to them by two or more instructions. In MIPS IV and later, including MIPS32 and MIPS64, this restriction does not exist.
Multiply Word to GPR

**Format:**
MUL rd, rs, rt

**Purpose:**
To multiply two words and write the result to a GPR.

**Description:**
GPR[rd] ← GPR[rs] × GPR[rt]

The 32-bit word value in GPR rs is multiplied by the 32-bit value in GPR rt, treating both operands as signed values, to produce a 64-bit result. The least significant 32 bits of the product are written to GPR rd. The contents of HI and LO are UNPREDICTABLE after the operation. No arithmetic exception occurs under any circumstances.

**Restrictions:**
Note that this instruction does not provide the capability of writing the result to the HI and LO registers.

**Operation:**

```
temp ← GPR[rs] × GPR[rt]
GPR[rd] ← temp31..0
HI ← UNPREDICTABLE
LO ← UNPREDICTABLE
```

**Exceptions:**
None

**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 GPR rd 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 GPR rt. This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
Floating Point Multiply

**Format:**

\[
\text{MUL.fmt}
\]

\[
\begin{array}{cccccc}
\text{COP1} & \text{fmt} & \text{ft} & \text{fs} & \text{fd} & \text{MUL} \\
010001 & 5 & 5 & 5 & 5 & 000010
\end{array}
\]

**MIPS32 Release 2**

**Purpose:** Floating Point Multiply

To multiply FP values

**Description:**

\[
\text{FPR}[fd] \leftarrow \text{FPR}[fs] \times \text{FPR}[ft]
\]

The value in FPR \(fs\) is multiplied by the value in FPR \(ft\). The result is calculated to infinite precision, rounded according to the current rounding mode in \(FCSR\), and placed into FPR \(fd\). The operands and result are values in format \(fmt\).

MUL.PS multiplies the upper and lower halves of FPR \(fs\) and FPR \(ft\) independently, and ORs together any generated exceptional conditions.

**Restrictions:**

The fields \(fs\), \(ft\), and \(fd\) must specify FPRs valid for operands of type \(fmt\); if they are not valid, the result is \textit{UNPREDICTABLE}. The operands must be values in format \(fmt\); if they are not, the result is \textit{UNPREDICTABLE} and the value of the operand FPRs becomes \textit{UNPREDICTABLE}.

The result of MUL.PS is \textit{UNPREDICTABLE} if the processor is executing in the \(FR=0\) 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the \(FR=1\) mode, but not with \(FR=0\), and not on a 32-bit FPU.

**Operation:**

\[
\text{StoreFPR} (fd, fmt, \text{ValueFPR} (fs, fmt) \times \text{fmt} \text{ValueFPR} (ft, fmt))
\]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow
Multiply Word

Format:\[MULT\ \text{rs}, \text{rt}\]

MIPS32

Purpose: Multiply Word

To multiply 32-bit signed integers

Description: \[(\text{HI}, \text{LO}) \leftarrow \text{GPR}[\text{rs}] \times \text{GPR}[\text{rt}]\]

The 32-bit word value in GPR \(\text{rt}\) is multiplied by the 32-bit value in GPR \(\text{rs}\), treating both operands as signed values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register \(\text{LO}\), and the high-order 32-bit word is placed into special register \(\text{HI}\).

No arithmetic exception occurs under any circumstances.

Restrictions:

None

Operation:

\[
\begin{align*}
\text{prod} & \leftarrow \text{GPR}[\text{rs}]_{31..0} \times \text{GPR}[\text{rt}]_{31..0} \\
\text{LO} & \leftarrow \text{prod}_{31..0} \\
\text{HI} & \leftarrow \text{prod}_{63..32}
\end{align*}
\]

Exceptions:

None

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 GPR \(\text{rt}\). This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
 Multiply Unsigned Word

Format: \texttt{MULTU rs, rt}

Purpose: Multiply Unsigned Word

To multiply 32-bit unsigned integers

Description: \((\text{HI, LO}) \leftarrow \text{GPR[rs]} \times \text{GPR[rt]}\)

The 32-bit word value in GPR \(rt\) is multiplied by the 32-bit value in GPR \(rs\), treating both operands as unsigned values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register \(LO\), and the high-order 32-bit word is placed into special register \(HI\).

No arithmetic exception occurs under any circumstances.

Restrictions:

None

Operation:

\[
\begin{align*}
\text{prod} & \leftarrow (0 \ || \ \text{GPR[rs]}_{31..0}) \times (0 \ || \ \text{GPR[rt]}_{31..0}) \\
\text{LO} & \leftarrow \text{prod}_{31..0} \\
\text{HI} & \leftarrow \text{prod}_{63..32}
\end{align*}
\]

Exceptions:

None

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 GPR \(rt\). This may reduce the latency of the instruction on those processors which implement data-dependent instruction latencies.
Floating Point Negate

**Purpose:** Floating Point Negate

To negate an FP value

**Description:**

\[ \text{FPR}[fd] \leftarrow -\text{FPR}[fs] \]

The value in FPR \(fs\) is negated and placed into FPR \(fd\). The value is negated by changing the sign bit value. The operand and result are values in format \(fmt\). NEG.PS negates the upper and lower halves of FPR \(fs\) independently, and ORs together any generated exceptional conditions.

If \(\text{FIR}_{\text{Has2008}}=0\) or \(\text{FCSR}_{\text{ABS2008}}=0\) then this operation is arithmetic. For this case, any NaN operand signals invalid operation.

If \(\text{FCSR}_{\text{ABS2008}}=1\) then this operation is non-arithmetic. For this case, both regular floating point numbers and NaN values are treated alike, only the sign bit is affected by this instruction. No IEEE exception can be generated for this case.

**Restrictions:**

The fields \(fs\) and \(fd\) must specify FPRs valid for operands of type \(fmt\); if they are not valid, the result is UNPREDICTABLE. The operand must be a value in format \(fmt\); if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

The result of NEG.PS is UNPREDICTABLE if the processor is executing in the \(FR=0\) 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the \(FR=1\) mode, but not with \(FR=0\), and not on a 32-bit FPU.

**Operation:**

\[ \text{StoreFPR}(fd, fmt, \text{Negate(ValueFPR}(fs, fmt))) \]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation, Invalid Operation
Floating Point Negative Multiply Add

**NMADD.fmt**

**Format:**

| NMADD.S | fd, fr, fs, ft |
| NMADD.D | fd, fr, fs, ft |
| NMADD.PS | fd, fr, fs, ft |

**MIPS64, MIPS32 Release 2**

**Purpose:** Floating Point Negative Multiply Add

To negate a combined multiply-then-add of FP values

**Description:**

\[ FPR[fd] \leftarrow -((FPR[fs] \times FPR[ft]) + FPR[fr]) \]

The value in FPR \( fs \) is multiplied by the value in FPR \( ft \) to produce an intermediate product. The intermediate product is rounded according to the current rounding mode in \( FCSR \). The value in FPR \( fr \) is added to the product.

The result sum is calculated to infinite precision, rounded according to the current rounding mode in \( FCSR \), negated by changing the sign bit, and placed into FPR \( fd \). The operands and result are values in format \( fmt \). The results and flags are as if separate floating-point multiply and add and negate instructions were executed.

**NMADD.PS** applies the operation to the upper and lower halves of FPR \( fr \), FPR \( fs \), and FPR \( ft \) independently, and ORs together any generated exceptional conditions.

**Cause** bits are ORed into the Flag bits if no exception is taken.

**Restrictions:**

The fields \( fr, fs, ft \), and \( fd \) must specify FPRs valid for operands of type \( fmt \); if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format \( fmt \); if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

The result of **NMADD.PS** is **UNPREDICTABLE** if the processor is executing in the \( FR=0 \) 32-bit FPU register model; i.e. it is predictable if executing on a 64-bit FPU in the \( FR=1 \) mode, but not with \( FR=0 \), and not on a 32-bit FPU.

**Compatibility and Availability:**

**NMADD.S** and **NMADD.D**: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode (\( FIR_{F64}=0 \) or 1, \( Status_{FR}=0 \) or 1).

**Operation:**

\[
\begin{align*}
  &vfr \leftarrow \text{ValueFPR}(fr, fmt) \\
  &vfs \leftarrow \text{ValueFPR}(fs, fmt) \\
  &vft \leftarrow \text{ValueFPR}(ft, fmt) \\
  &\text{StoreFPR}(fd, fmt, -(vfr +_{fmt} (vfs \times_{fmt} vft)))
\end{align*}
\]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow
Floating Point Negative Multiply Subtract  

**Format:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMSUB.fmt</td>
<td></td>
</tr>
<tr>
<td>NMSUB.S fd, fr, fs, ft</td>
<td>MIPS64, MIPS32 Release 2</td>
</tr>
<tr>
<td>NMSUB.D fd, fr, fs, ft</td>
<td>MIPS64, MIPS32 Release 2</td>
</tr>
<tr>
<td>NMSUB.PS fd, fr, fs, ft</td>
<td>MIPS64, MIPS32 Release 2</td>
</tr>
</tbody>
</table>

**Purpose:** Floating Point Negative Multiply Subtract

To negate a combined multiply-then-subtract of FP values

**Description:**

\[ \text{FPR}[fd] \leftarrow -((\text{FPR}[fs] \times \text{FPR}[ft]) - \text{FPR}[fr]) \]

The value in FPR \(fs\) is multiplied by the value in FPR \(ft\) to produce an intermediate product. The intermediate product is rounded according to the current rounding mode in FCSR. The value in FPR \(fr\) is subtracted from the product.

The result is calculated to infinite precision, rounded according to the current rounding mode in FCSR, negated by changing the sign bit, and placed into FPR \(fd\). The operands and result are values in format \(fmt\). The results and flags are as if separate floating-point multiply and subtract and negate instructions were executed.

NMSUB.PS applies the operation to the upper and lower halves of FPR \(fr\), FPR \(fs\), and FPR \(ft\) independently, and ORs together any generated exceptional conditions.

_Cause_ bits are ORed into the Flag bits if no exception is taken.

**Restrictions:**

The fields \(fr\), \(fs\), \(ft\), and \(fd\) must specify FPRs valid for operands of type \(fmt\); if they are not valid, the result is UNPREDICTABLE.

The operands must be values in format \(fmt\); if they are not, the result is UNPREDICTABLE and the value of the operand FPRs becomes UNPREDICTABLE.

The result of NMSUB.PS is UNPREDICTABLE if the processor is executing in the \(FR=0\) 32-bit FPU register model; i.e. it is predictable if executing on a 64-bit FPU in the \(FR=1\) mode, but not with \(FR=0\), and not on a 32-bit FPU.

**Compatibility and Availability:**

NMSUB.S and NMSUB.D: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode (\(FlR_{F6}=0\) or 1, \(Status_{FR}=0\) or 1).

**Operation:**

\[
\begin{align*}
\text{vfr} & \leftarrow \text{ValueFPR}(fr, \text{fmt}) \\
\text{vfs} & \leftarrow \text{ValueFPR}(fs, \text{fmt}) \\
\text{vft} & \leftarrow \text{ValueFPR}(ft, \text{fmt}) \\
\text{StoreFPR}(fd, \text{fmt}, -(\text{vfs} \times_{\text{fmt}} \text{vft}) -_{\text{fmt}} \text{vfr}))
\end{align*}
\]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow
No Operation

**Format:** NOP

**Purpose:** No Operation

To perform no operation.

**Description:**

NOP is the assembly idiom used to denote no operation. The actual instruction is interpreted by the hardware as SLL r0, r0, 0.

**Restrictions:**

None

**Operation:**

None

**Exceptions:**

None

**Programming Notes:**

The zero instruction word, which represents SLL, r0, r0, 0, is the preferred NOP for software to use to fill branch and jump delay slots and to pad out alignment sequences.
**Not Or**

Format: NOR rd, rs, rt

**Purpose:** Not Or

To do a bitwise logical NOT OR

**Description:**

\[ GPR[rd] \leftarrow GPR[rs] \text{ NOR } GPR[rt] \]

The contents of GPR rs are combined with the contents of GPR rt in a bitwise logical NOR operation. The result is placed into GPR rd.

**Restrictions:**

None

**Operation:**

\[ GPR[rd] \leftarrow GPR[rs] \text{ nor } GPR[rt] \]

**Exceptions:**

None
**Or**

**Format:** \( \text{OR} \ rd, \ rs, \ rt \)

**Purpose:** Or
To do a bitwise logical OR

**Description:** \( \text{GPR}[rd] \leftarrow \text{GPR}[rs] \text{ or } \text{GPR}[rt] \)
The contents of GPR \( rs \) are combined with the contents of GPR \( rt \) in a bitwise logical OR operation. The result is placed into GPR \( rd \).

**Restrictions:**
None

**Operation:**
\( \text{GPR}[rd] \leftarrow \text{GPR}[rs] \text{ or } \text{GPR}[rt] \)

**Exceptions:**
None
Or Immediate

**Format:** ORI rt, rs, immediate

**Purpose:** Or Immediate

To do a bitwise logical OR with a constant

**Description:** GPR[rt] ← GPR[rs] or immediate

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR rs in a bitwise logical OR operation. The result is placed into GPR rt.

**Restrictions:**

None

**Operation:**

GPR[rt] ← GPR[rs] or zero_extend(immediate)

**Exceptions:**

None

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORI</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS32
Wait for the LLBit to clear

**Purpose:** Wait for the LLBit to clear

**Description:**
Locks implemented using the LL/SC instructions are a common method of synchronization between threads of control. A typical lock implementation does a load-linked instruction and checks the value returned to determine whether the software lock is set. If it is, the code branches back to retry the load-linked instruction, thereby implementing an active busy-wait sequence. The PAUSE instructions is intended to be placed into the busy-wait sequence to block the instruction stream until such time as the load-linked instruction has a chance to succeed in obtaining the software lock.

The precise behavior of the PAUSE instruction is implementation-dependent, but it usually involves descheduling the instruction stream until the LLBit is zero. In a single-threaded processor, this may be implemented as a short-term WAIT operation which resumes at the next instruction when the LLBit is zero or on some other external event such as an interrupt. On a multi-threaded processor, this may be implemented as a short term YIELD operation which resumes at the next instruction when the LLBit is zero. In either case, it is assumed that the instruction stream which gives up the software lock does so via a write to the lock variable, which causes the processor to clear the LLBit as seen by this thread of execution.

The encoding of the instruction is such that it is backward compatible with all previous implementations of the architecture. The PAUSE instruction can therefore be placed into existing lock sequences and treated as a NOP by the processor, even if the processor does not implement the PAUSE instruction.

**Restrictions:**
The operation of the processor is **UNPREDICTABLE** if a PAUSE instruction is placed in the delay slot of a branch or a jump.

**Operation:**

```plaintext
if LLBit ≠ 0 then
    EPC ← PC + 4 /* Resume at the following instruction */
    DescheduleInstructionStream()
endif
```

**Exceptions:**
None

**Programming Notes:**
The PAUSE instruction is intended to be inserted into the instruction stream after an LL instruction has set the LLBit and found the software lock set. The program may wait forever if a PAUSE instruction is executed and there is no possibility that the LLBit will ever be cleared.

An example use of the PAUSE instruction is included in the following example:

```plaintext
acquire_lock:
```
**Wait for the LLBit to clear**

```
11    t0, 0(a0)  /* Read software lock, set hardware lock */
bnez t0, acquire_lock_retry: /* Branch if software lock is taken */
addiu t0, t0, 1  /* Set the software lock */
sc t0, 0(a0)      /* Try to store the software lock */
bnez t0, 10f  /* Branch if lock acquired successfully */
sync
acquire_lock_retry:
pause            /* Wait for LLBIT to clear before retry */
b acquire_lock   /* and retry the operation */
nop
10:

    Critical region code

release_lock:
    sync
    sw zero, 0(a0)  /* Release software lock, clearing LLBIT */
    /* for any PAUSEd waiters */
```
**Format:** PLL.PS fd, fs, ft

**MIPS64, MIPS32 Release 2**

**Purpose:** Pair Lower Lower

To merge a pair of paired single values with realignment

**Description:** FPR[fd] ← lower(FPR[fs]) || lower(FPR[ft])

A new paired-single value is formed by catenating the lower single of FPR fs (bits 31..0) and the lower single of FPR ft (bits 31..0).

The move is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

The fields fs, ft, and fd must specify FPRs valid for operands of type PS. If they are not valid, the result is UNPREDICTABLE.

The result of this instruction is UNPREDICTABLE if the processor is executing in the FR=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the FR=1 mode, but not with FR=0, and not on a 32-bit FPU.

**Operation:**

\[ \text{StoreFPR}(fd, \text{PS}, \text{ValueFPR}(fs, \text{PS})_{31..0} || \text{ValueFPR}(ft, \text{PS})_{31..0}) \]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction
Pair Lower Upper

**Format:** PLU.PS fd, fs, ft

**MIPS64, MIPS32 Release 2**

**Purpose:** Pair Lower Upper

To merge a pair of paired single values with realignment

**Description:**

\[
\text{FPR}[fd] \leftarrow \text{lower}(\text{FPR}[fs]) \mid \mid \text{upper}(\text{FPR}[ft])
\]

A new paired-single value is formed by concatenating the lower single of FPR \(fs\) (bits 31..0) and the upper single of FPR \(ft\) (bits 63..32).

The move is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

The fields \(fs\), \(ft\), and \(fd\) must specify FPRs valid for operands of type \(PS\). If they are not valid, the result is **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in the \(FR=0\) 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the \(FR=1\) mode, but not with \(FR=0\), and not on a 32-bit FPU.

**Operation:**

\[
\text{StoreFPR}(fd, \text{PS}, \text{ValueFPR}(fs, \text{PS})_{31..0} \mid \mid \text{ValueFPR}(ft, \text{PS})_{63..32})
\]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction
Prefetch

**Purpose:** Prefetch

To move data between memory and cache.

**Description:** `prefetch_memory(GPR[base] + offset)`

PREF adds the 16-bit signed `offset` to the contents of GPR `base` to form an effective byte address. The `hint` field supplies information about the way that the data is expected to be used.

PREF enables the processor to take some action, typically causing data to be moved to or from the cache, to improve program performance. The action taken for a specific PREF instruction is both system and context dependent. Any action, including doing nothing, is permitted as long as it does not change architecturally visible state or alter the meaning of a program. Implementations are expected either to do nothing, or to take an action that increases the performance of the program. The PrepareForStore function is unique in that it may modify the architecturally visible state.

PREF does not cause addressing-related exceptions, including TLB exceptions. If the address specified would cause an addressing exception, the exception condition is ignored and no data movement occurs. However even if no data is moved, some action that is not architecturally visible, such as writeback of a dirty cache line, can take place.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a byproduct of the action taken by the PREF instruction.

PREF neither generates a memory operation nor modifies the state of a cache line for a location with an `uncached` memory access type, whether this type is specified by the address segment (e.g., `kseg1`), the programmed cacheability and coherency attribute of a segment (e.g., the use of the `K0`, `KU`, or `K23` fields in the `Config` register), or the per-page cacheability and coherency attribute provided by the TLB.

If PREF results in a memory operation, the memory access type and cacheability&coherency attribute used for the operation are determined by the memory access type and cacheability&coherency attribute of the effective address, just as it would be if the memory operation had been caused by a load or store to the effective address.

For a cached location, the expected and useful action for the processor is to prefetch a block of data that includes the effective address. The size of the block and the level of the memory hierarchy it is fetched into are implementation specific.

In coherent multiprocessor implementations, if the effective address uses a coherent Cacheability and Coherency Attribute (CCA), then the instruction causes a coherent memory transaction to occur. This means a prefetch issued on one processor can cause data to be evicted from the cache in another processor.

The PREF instruction and the memory transactions which are sourced by the PREF instruction, such as cache refill or cache writeback, obey the ordering and completion rules of the SYNC instruction.

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Data Use and Desired Prefetch Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>load</td>
<td>Use: Prefetched data is expected to be read (not modified). Action: Fetch data as if for a load.</td>
</tr>
</tbody>
</table>

**Table 4.4 Values of hint Field for PREF Instruction**
### Table 4.4 Values of hint Field for PREF Instruction

<table>
<thead>
<tr>
<th>Value</th>
<th>hint Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>store</td>
<td>Use: Prefetched data is expected to be stored or modified. Action: Fetch data as if for a store.</td>
</tr>
<tr>
<td>2-3</td>
<td>Reserved</td>
<td>Reserved for future use - not available to implementations.</td>
</tr>
<tr>
<td>4</td>
<td>load_streamed</td>
<td>Use: Prefetched data is expected to be read (not modified) but not reused extensively; it “streams” through cache. Action: Fetch data as if for a load and place it in the cache so that it does not displace data prefetched as “retained.”</td>
</tr>
<tr>
<td>5</td>
<td>store_streamed</td>
<td>Use: Prefetched data is expected to be stored or modified but not reused extensively; it “streams” through cache. Action: Fetch data as if for a store and place it in the cache so that it does not displace data prefetched as “retained.”</td>
</tr>
<tr>
<td>6</td>
<td>load_retained</td>
<td>Use: Prefetched data is expected to be read (not modified) and reused extensively; it should be “retained” in the cache. Action: Fetch data as if for a load and place it in the cache so that it is not displaced by data prefetched as “streamed.”</td>
</tr>
<tr>
<td>7</td>
<td>store_retained</td>
<td>Use: Prefetched data is expected to be stored or modified and reused extensively; it should be “retained” in the cache. Action: Fetch data as if for a store and place it in the cache so that it is not displaced by data prefetched as “streamed.”</td>
</tr>
<tr>
<td>8-20</td>
<td>Reserved</td>
<td>Reserved for future use - not available to implementations.</td>
</tr>
<tr>
<td>21-24</td>
<td>Implementation Dependent</td>
<td>Unassigned by the Architecture - available for implementation-dependent use.</td>
</tr>
<tr>
<td>25</td>
<td>writeback_invalidate (also known as “nudge”)</td>
<td>Use: Data is no longer expected to be used. Action: For a writeback cache, schedule a writeback of any dirty data. At the completion of the writeback, mark the state of any cache lines written back as invalid. If the cache line is not dirty, it is implementation dependent whether the state of the cache line is marked invalid or left unchanged. If the cache line is locked, no action is taken.</td>
</tr>
<tr>
<td>26-29</td>
<td>Implementation Dependent</td>
<td>Unassigned by the Architecture - available for implementation-dependent use.</td>
</tr>
<tr>
<td>30</td>
<td>PrepareForStore</td>
<td>Use: Prepare the cache for writing an entire line, without the overhead involved in filling the line from memory. Action: If the reference hits in the cache, no action is taken. If the reference misses in the cache, a line is selected for replacement, any valid and dirty victim is written back to memory, the entire line is filled with zero data, and the state of the line is marked as valid and dirty. Programming Note: Because the cache line is filled with zero data on a cache miss, software must not assume that this action, in and of itself, can be used as a fast bzero-type function.</td>
</tr>
<tr>
<td>31</td>
<td>Implementation Dependent</td>
<td>Unassigned by the Architecture - available for implementation-dependent use.</td>
</tr>
</tbody>
</table>
Restrictions:
None

Operation:
\[ \text{vAddr} \leftarrow \text{GPR[base]} + \text{sign\_extend(offset)} \]
\[ (\text{pAddr, CCA}) \leftarrow \text{AddressTranslation(vAddr, DATA, LOAD)} \]
\[ \text{Prefetch}(\text{CCA, pAddr, vAddr, DATA, hint}) \]

Exceptions:
Bus Error, Cache Error
Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

Programming Notes:
Prefetch cannot move data to or from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch does not cause addressing exceptions. A prefetch may be used using an address pointer before the validity of the pointer is determined without worrying about an addressing exception.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a byproduct of the action taken by the PREF instruction. Typically, this only occurs in systems which have high-reliability requirements.

Prefetch operations have no effect on cache lines that were previously locked with the CACHE instruction.

*Hint* field encodings whose function is described as “streamed” or “retained” convey usage intent from software to hardware. Software should not assume that hardware will always prefetch data in an optimal way. If data is to be truly retained, software should use the Cache instruction to lock data into the cache.
Prefetch EVA

**Format:**  
PREFE hint,offset(base)

**Purpose:** Prefetch EVA  
To move data between user mode virtual address space memory and cache while operating in kernel mode.

**Description:** `prefetch_memory(GPR[base] + offset)`  
PREFE adds the 9-bit signed `offset` to the contents of GPR `base` to form an effective byte address. The `hint` field supplies information about the way that the data is expected to be used.

PREFE enables the processor to take some action, typically causing data to be moved to or from the cache, to improve program performance. The action taken for a specific PREFE instruction is both system and context dependent. Any action, including doing nothing, is permitted as long as it does not change architecturally visible state or alter the meaning of a program. Implementations are expected either to do nothing, or to take an action that increases the performance of the program. The PrepareForStore function is unique in that it may modify the architecturally visible state.

PREFE does not cause addressing-related exceptions, including TLB exceptions. If the address specified would cause an addressing exception, the exception condition is ignored and no data movement occurs. However even if no data is moved, some action that is not architecturally visible, such as writeback of a dirty cache line, can take place.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a byproduct of the action taken by the PREFE instruction.

PREFE neither generates a memory operation nor modifies the state of a cache line for a location with an uncached memory access type, whether this type is specified by the address segment (e.g., kseg1), the programmed cacheability and coherency attribute of a segment (e.g., the use of the `K0`, `KU`, or `K23` fields in the `Config` register), or the per-page cacheability and coherency attribute provided by the TLB.

If PREFE results in a memory operation, the memory access type and cacheability&coherency attribute used for the operation are determined by the memory access type and cacheability&coherency attribute of the effective address, just as it would be if the memory operation had been caused by a load or store to the effective address.

For a cached location, the expected and useful action for the processor is to prefetch a block of data that includes the effective address. The size of the block and the level of the memory hierarchy it is fetched into are implementation specific.

In coherent multiprocessor implementations, if the effective address uses a coherent Cacheability and Coherency Attribute (CCA), then the instruction causes a coherent memory transaction to occur. This means a prefetch issued on one processor can cause data to be evicted from the cache in another processor.

The PREFE instruction and the memory transactions which are sourced by the PREFE instruction, such as cache refill or cache writeback, obey the ordering and completion rules of the SYNC instruction.

The PREFE instruction functions in exactly the same fashion as the PREF instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the `Config5EVA` field being set to one.
### Table 4.5 Values of hint Field for PREFE Instruction

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Data Use and Desired Prefetch Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>load</td>
<td>Use: Prefetched data is expected to be read (not modified). Action: Fetch data as if for a load.</td>
</tr>
<tr>
<td>1</td>
<td>store</td>
<td>Use: Prefetched data is expected to be stored or modified. Action: Fetch data as if for a store.</td>
</tr>
<tr>
<td>2-3</td>
<td>Reserved</td>
<td>Reserved for future use - not available to implementations.</td>
</tr>
<tr>
<td>4</td>
<td>load_streamed</td>
<td>Use: Prefetched data is expected to be read (not modified) but not reused extensively; it “streams” through cache. Action: Fetch data as if for a load and place it in the cache so that it does not displace data prefetched as “retained.”</td>
</tr>
<tr>
<td>5</td>
<td>store_streamed</td>
<td>Use: Prefetched data is expected to be stored or modified but not reused extensively; it “streams” through cache. Action: Fetch data as if for a store and place it in the cache so that it does not displace data prefetched as “retained.”</td>
</tr>
<tr>
<td>6</td>
<td>load_retained</td>
<td>Use: Prefetched data is expected to be read (not modified) and reused extensively; it should be “retained” in the cache. Action: Fetch data as if for a load and place it in the cache so that it is not displaced by data prefetched as “streamed.”</td>
</tr>
<tr>
<td>7</td>
<td>store_retained</td>
<td>Use: Prefetched data is expected to be stored or modified and reused extensively; it should be “retained” in the cache. Action: Fetch data as if for a store and place it in the cache so that it is not displaced by data prefetched as “streamed.”</td>
</tr>
<tr>
<td>8-20</td>
<td>Reserved</td>
<td>Reserved for future use - not available to implementations.</td>
</tr>
<tr>
<td>21-24</td>
<td>Implementation Dependent</td>
<td>Unassigned by the Architecture - available for implementation-dependent use.</td>
</tr>
<tr>
<td>25</td>
<td>writeback_invalidate (also known as “nudge”)</td>
<td>Use: Data is no longer expected to be used. Action: For a writeback cache, schedule a writeback of any dirty data. At the completion of the writeback, mark the state of any cache lines written back as invalid. If the cache line is not dirty, it is implementation dependent whether the state of the cache line is marked invalid or left unchanged. If the cache line is locked, no action is taken.</td>
</tr>
<tr>
<td>26-29</td>
<td>Implementation Dependent</td>
<td>Unassigned by the Architecture - available for implementation-dependent use.</td>
</tr>
</tbody>
</table>
Table 4.5 Values of hint Field for PREFE Instruction

<table>
<thead>
<tr>
<th>hint</th>
<th>Operation</th>
<th>Programming Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>PrepareForStore</td>
<td>Use: Prepare the cache for writing an entire line, without the overhead involved in filling the line from memory. Action: If the reference hits in the cache, no action is taken. If the reference misses in the cache, a line is selected for replacement, any valid and dirty victim is written back to memory, the entire line is filled with zero data, and the state of the line is marked as valid and dirty. Programming Note: Because the cache line is filled with zero data on a cache miss, software must not assume that this action, in and of itself, can be used as a fast bzero-type function.</td>
</tr>
<tr>
<td>31</td>
<td>Implementation Dependent</td>
<td>Unassigned by the Architecture - available for implementation-dependent use.</td>
</tr>
</tbody>
</table>

Restrictions:

Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Operation:

\[
\text{vAddr} \leftarrow \text{GPR[base]} + \text{signExtend}(\text{offset})
\]

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

\[
\text{Prefetch}(\text{CCA}, \text{pAddr}, \text{vAddr}, \text{DATA, hint})
\]

Exceptions:

Bus Error, Cache Error, Address Error, Reserved Instruction, Coprocessor Usable

Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

Programming Notes:

Prefetch cannot move data to or from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch does not cause addressing exceptions. A prefetch may be used using an address pointer before the validity of the pointer is determined without worrying about an addressing exception.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a byproduct of the action taken by the PREFE instruction. Typically, this only occurs in systems which have high-reliability requirements.

Prefetch operations have no effect on cache lines that were previously locked with the CACHE instruction.

*Hint* field encodings whose function is described as “streamed” or “retained” convey usage intent from software to hardware. Software should not assume that hardware will always prefetch data in an optimal way. If data is to be truly retained, software should use the Cache instruction to lock data into the cache.
Prefetch Indexed

**Purpose:** Prefetch Indexed

To move data between memory and cache.

**Description:** `prefetch_memory[GPR[base] + GPR[index]]`

PREFIX adds the contents of GPR `index` to the contents of GPR `base` to form an effective byte address. The `hint` field supplies information about the way the data is expected to be used.

The only functional difference between the PREF and PREFIX instructions is the addressing mode implemented by the two. Refer to the PREF instruction for all other details, including the encoding of the `hint` field.

**Restrictions:**

**Compatibility and Availability:**

PREFIX: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode ($FIR_{F64}=0$ or 1, $Status_{FR}=0$ or 1).

**Operation:**

```
vAddr ← GPR[base] + GPR[index]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction, Bus Error, Cache Error

**Programming Notes:**

The PREFIX instruction is only available on processors that implement floating point and should never be generated by compilers in situations other than those in which the corresponding load and store indexed floating point instructions are generated.

Refer to the corresponding section in the PREF instruction description.
### Format: PUL.PS fd, fs, ft

**MIPS64, MIPS32 Release 2**

**Purpose:** Pair Upper Lower

To merge a pair of paired single values with realignment

**Description:**

\[
FPR[fd] \leftarrow \text{upper}(FPR[fs]) \ || \ \text{lower}(FPR[ft])
\]

A new paired-single value is formed by catenating the upper single of FPR \(fs\) (bits \(63..32\)) and the lower single of FPR \(ft\) (bits \(31..0\)).

The move is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

The fields \(fs\), \(ft\), and \(fd\) must specify FPRs valid for operands of type \(PS\). If they are not valid, the result is **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in the \(FR=0\) 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the \(FR=1\) mode, but not with \(FR=0\), and not on a 32-bit FPU.

**Operation:**

\[
\text{StoreFPR}(fd, \ PS, \ \text{ValueFPR}(fs, \ PS)_{63..32} \ || \ \text{ValueFPR}(ft, \ PS)_{31..0})
\]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction
Pair Upper Upper

**Format:** PUU.PS fd, fs, ft

**MIPS64, MIPS32 Release 2**

**Purpose:** Pair Upper Upper

To merge a pair of paired single values with realignment

**Description:**

\[ \text{FPR}[fd] \leftarrow \text{upper}(\text{FPR}[fs]) || \text{upper}(\text{FPR}[ft]) \]

A new paired-single value is formed by catenating the upper single of FPR fs (bits 63..32) and the upper single of FPR ft (bits 63..32).

The move is non-arithmetic; it causes no IEEE 754 exceptions.

**Restrictions:**

The fields fs, ft, and fd must specify FPRs valid for operands of type PS. If they are not valid, the result is **UNPREDICTABLE**.

The result of this instruction is **UNPREDICTABLE** if the processor is executing in the FR=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the FR=1 mode, but not with FR=0, and not on a 32-bit FPU.

**Operation:**

\[ \text{StoreFPR}(fd, PS, \text{ValueFPR}(fs, PS)_{63..32} || \text{ValueFPR}(ft, PS)_{63..32}) \]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction
**Format:** \texttt{RDHWR rt,rd}  

**Purpose:** Read Hardware Register  
To move the contents of a hardware register to a general purpose register (GPR) if that operation is enabled by privileged software.

The purpose of this instruction is to give user mode access to specific information that is otherwise only visible in kernel mode.

**Description:** \( \text{GPR}[rt] \leftarrow \text{HWR}[rd] \)

If access is allowed to the specified hardware register, the contents of the register specified by \( rd \) is loaded into general register \( rt \). Access control for each register is selected by the bits in the coprocessor 0 \( \text{HWREna} \) register.

The available hardware registers, and the encoding of the \( rd \) field for each, are shown in Table 4.6.

### Table 4.6 RDHWR Register Numbers

<table>
<thead>
<tr>
<th>Register Number (( rd ) Value)</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CPUNum</td>
<td>Number of the CPU on which the program is currently running. The register provides read access to the coprocessor 0 ( EBase_{CPUNum} ) field.</td>
</tr>
<tr>
<td>1</td>
<td>SYNCI Step</td>
<td>Address step size to be used with the SYNCI instruction, or zero if no caches need be synchronized. See that instruction’s description for the use of this value.</td>
</tr>
<tr>
<td>2</td>
<td>CC</td>
<td>High-resolution cycle counter. This register provides read access to the coprocessor 0 ( \text{Count} ) Register.</td>
</tr>
<tr>
<td>3</td>
<td>CCRRes</td>
<td>Resolution of the CC register. This value denotes the number of cycles between update of the register. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\begin{tabular}{</td>
</tr>
<tr>
<td>4-28</td>
<td></td>
<td>These registers numbers are reserved for future architecture use. Access results in a Reserved Instruction Exception.</td>
</tr>
<tr>
<td>29</td>
<td>ULR</td>
<td>User Local Register. This register provides read access to the coprocessor 0 ( \text{UserLocal} ) register, if it is implemented. In some operating environments, the ( \text{UserLocal} ) register is a pointer to a thread-specific storage block.</td>
</tr>
<tr>
<td>30-31</td>
<td></td>
<td>These register numbers are reserved for implementation-dependent use. If they are not implemented, access results in a Reserved Instruction Exception.</td>
</tr>
</tbody>
</table>
Restrictions:
In implementations of Release 1 of the Architecture, this instruction resulted in a Reserved Instruction Exception.

Access to the specified hardware register is enabled if Coprocessor 0 is enabled, or if the corresponding bit is set in the \texttt{HWREna} register. If access is not allowed or the register is not implemented, a Reserved Instruction Exception is signaled.

Operation:

\begin{verbatim}
    case rd
        0: temp ← EBaseCPUNum
        1: temp ← SYNCI_StepSize()
        2: temp ← Count
        3: temp ← CountResolution()
        29: temp ← UserLocal
        30: temp ← Implementation-Dependent-Value
        31: temp ← Implementation-Dependent-Value
        otherwise: SignalException(ReservedInstruction)
    endcase
GPR[rt] ← temp
\end{verbatim}

Exceptions:

Reserved Instruction
Read GPR from Previous Shadow Set

RDPGPR

Purpose: Read GPR from Previous Shadow Set
To move the contents of a GPR from the previous shadow set to a current GPR.

Description: GPR[rd] ← SGPR[SRSCtl\textsubscript{PSS}, rt]
The contents of the shadow GPR register specified by \textit{SRSCtl\textsubscript{PSS}} (signifying the previous shadow set number) and \textit{rt} (specifying the register number within that set) is moved to the current GPR \textit{rd}.

Restrictions:
In implementations prior to Release 2 of the Architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:
GPR[rd] ← SGPR[SRSCtl\textsubscript{PSS}, rt]

Exceptions:
Coprocessor Unusable
Reserved Instruction
Reciprocal Approximation

**Purpose:** Reciprocal Approximation
To approximate the reciprocal of an FP value (quickly)

**Description:**
\[
\text{FPR}[fd] \leftarrow 1.0 / \text{FPR}[fs]
\]
The reciprocal of the value in FPR \( fs \) is approximated and placed into FPR \( fd \). The operand and result are values in format \( fmt \).

The numeric accuracy of this operation is implementation dependent; it does not meet the accuracy specified by the IEEE 754 Floating Point standard. The computed result differs from the both the exact result and the IEEE-mandated representation of the exact result by no more than one unit in the least-significant place (ULP).

It is implementation dependent whether the result is affected by the current rounding mode in \( FCSR \).

**Restrictions:**
The fields \( fs \) and \( fd \) must specify FPRs valid for operands of type \( fmt \); if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format \( fmt \); if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

**Compatibility and Availability:**
RECIP.S and RECIP.D: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode (\( FIR_{F06}=0 \) or 1, \( Status_{FK}=0 \) or 1).

**Operation:**
\[
\text{StoreFPR}(fd, fmt, 1.0 / \text{valueFPR}(fs, fmt))
\]

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Inexact, Division-by-zero, Unimplemented Op, Invalid Op, Overflow, Underflow
Rotate Word Right

Format: \texttt{ROTR \textit{rd}, \textit{rt}, \textit{sa}}

Purpose: Rotate Word Right

To execute a logical right-rotate of a word by a fixed number of bits.

Description: \texttt{GPR[rd] \leftarrow GPR[rt] \leftrightarrow (right) \textit{sa}}

The contents of the low-order 32-bit word of GPR \textit{rt} are rotated right; the word result is placed in GPR \textit{rd}. The bit-rotate amount is specified by \textit{sa}.

Restrictions:

Operation:

\[
\begin{align*}
\text{if } \left( \text{ArchitectureRevision() < 2} \right) \text{ and } (\text{Config3SM} = 0) \text{ then} \\
\quad \text{UNPREDICTABLE} \\
\text{endif} \\
\quad s \leftarrow \textit{sa} \\
\quad \text{temp} \leftarrow \text{GPR[rt]}_{s-1..0} \text{ || GPR[rt]}_{31..s} \\
\quad \text{GPR[rd]} \leftarrow \text{temp}
\end{align*}
\]

Exceptions:

Reserved Instruction
Rotate Word Right Variable

Format:  \texttt{ROTRV \textit{rd}, \textit{rt}, \textit{rs}}

Purpose: Rotate Word Right Variable

To execute a logical right-rotate of a word by a variable number of bits

Description: \( \text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rt}] \leftrightarrow \text{(right)} \text{GPR}[\text{rs}] \)

The contents of the low-order 32-bit word of GPR \( \text{rt} \) are rotated right; the word result is placed in GPR \( \text{rd} \). The bit-rotate amount is specified by the low-order 5 bits of GPR \( \text{rs} \).

Restrictions: 

Operation:

\begin{verbatim}
if ((ArchitectureRevision() < 2) and (Config3SM = 0)) then
    UNPREDICTABLE
endif
s \leftarrow \text{GPR}[\text{rs}]_{4..0}
temp \leftarrow \text{GPR}[\text{rt}]_{s-1..0} \mid \mid \text{GPR}[\text{rt}]_{31..s}
\text{GPR}[\text{rd}] \leftarrow \text{temp}
\end{verbatim}

Exceptions:

Reserved Instruction

SmartMIPS Crypto, MIPS32 Release 2
Floating Point Round to Long Fixed Point

**Format:**
ROUND.L.fmt

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>ROUND.L</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>001000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set, Revision 5.04 254**

**Purpose:**
Floating Point Round to Long Fixed Point

To convert an FP value to 64-bit fixed point, rounding to nearest

**Description:**
FPR[fd] ← convert_and_round(FPR[fs])

The value in FPR fs, in format fmt, is converted to a value in 64-bit long fixed point format and rounded to nearest/even (rounding mode 0). The result is placed in FPR fd.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to 2^{63}-1, the result cannot be represented correctly and an IEEE Invalid Operation condition exists. In this case the Invalid Operation flag is set in the FCSR. If the Invalid Operation Enable bit is set in the FCSR, no result is written to fd and an Invalid Operation exception is taken immediately. Otherwise, the default result, 2^{63}–1, is written to fd.

**Restrictions:**
The fields fs and fd must specify valid FPRs; fs for type fmt and fd for long fixed point; if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

The result of this instruction is UNPREDICTABLE if the processor is executing in the FR=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the FR=1 mode, but not with FR=0, and not on a 32-bit FPU.

**Operation:**

StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Inexact, Unimplemented Operation, Invalid Operation
Floating Point Round to Word Fixed Point  

**ROUND.W.fmt**

**Format:**  
ROUND.W.S  fd, fs  \[MIPS32\]  
ROUND.W.D  fd, fs  \[MIPS32\]

**Purpose:** Floating Point Round to Word Fixed Point  
To convert an FP value to 32-bit fixed point, rounding to nearest

**Description:**  
\[FPR[fd] \leftarrow convert_and_round(FPR[fs])\]  
The value in FPR fs, in format fmt, is converted to a value in 32-bit word fixed point format rounding to nearest/even (rounding mode 0). The result is placed in FPR fd.

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{31}\) to \(2^{31}-1\), the result cannot be represented correctly and an IEEE Invalid Operation condition exists. In this case the Invalid Operation flag is set in the FCSR. If the Invalid Operation Enable bit is set in the FCSR, no result is written to fd and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{31}-1\), is written to fd.

**Restrictions:**  
The fields fs and fd must specify valid FPRs; fs for type fmt and fd for word fixed point; if they are not valid, the result is \(UNPREDICTABLE\).

The operand must be a value in format fmt; if it is not, the result is \(UNPREDICTABLE\) and the value of the operand FPR becomes \(UNPREDICTABLE\).

**Operation:**  
\[
\text{StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))}
\]

**Exceptions:**  
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**  
Inexact, Unimplemented Operation, Invalid Operation
Reciprocal Square Root Approximation

**Format:** RSQRT.fmt

- **RSQRT.S** \( fd, fs \)
- **RSQRT.D** \( fd, fs \)

**MIPS64, MIPS32 Release 2**

**Purpose:** Reciprocal Square Root Approximation

To approximate the reciprocal of the square root of an FP value (quickly)

**Description:**

\[ \text{FPR}[fd] \leftarrow 1.0 / \sqrt{\text{FPR}[fs]} \]

The reciprocal of the positive square root of the value in FPR \( fs \) is approximated and placed into FPR \( fd \). The operand and result are values in format \( fmt \).

The numeric accuracy of this operation is implementation dependent; it does not meet the accuracy specified by the IEEE 754 Floating Point standard. The computed result differs from both the exact result and the IEEE-mandated representation of the exact result by no more than two units in the least-significant place (ULP).

The effect of the current FCSR rounding mode on the result is implementation dependent.

**Restrictions:**

- The fields \( fs \) and \( fd \) must specify FPRs valid for operands of type \( fmt \); if they are not valid, the result is **UNPREDICTABLE**.
- The operand must be a value in format \( fmt \); if it is not, the result is **UNPREDICTABLE** and the value of the operand FPR becomes **UNPREDICTABLE**.

**Compatibility and Availability:**

RSQRT.S and RSQRT.D: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode (\( FIR_{F0} = 0 \) or 1, \( Status_{FR} = 0 \) or 1).

**Operation:**

\[ \text{StoreFPR}(fd, fmt, 1.0 / \text{SquareRoot(valueFPR}(fs, fmt))) \]

**Exceptions:**

- Coprocessor Unusable, Reserved Instruction
- Floating Point Exceptions:
  - Inexact, Division-by-zero, Unimplemented Operation, Invalid Operation, Overflow, Underflow
**Store Byte**

<table>
<thead>
<tr>
<th>Format:</th>
<th>SB rt, offset(base)</th>
</tr>
</thead>
</table>

**Purpose:** Store Byte

To store a byte to memory

**Description:**

```
memory[baseGPR + offset] ← GPR[rt]
```

The least-significant 8-bit byte of GPR rt is stored in memory at the location specified by the effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

**Restrictions:**

None

**Operation:**

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReverseEndian2)
bytesel ← vAddr1..0 xor BigEndianCPU2
dataword ← GPR[rt]31-8*bytesel..0 || 08*bytesel
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
```

**Exceptions:**

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Watch
Store Byte EVA

**Format:**  
SBE rt, offset(base)

**Purpose:** Store Byte EVA  
To store a byte to user mode virtual address space when executing in kernel mode.

**Description:**  
memory[GPR[base] + offset] ← GPR[rt]

The least-significant 8-bit byte of GPR rt is stored in memory at the location specified by the effective address. The 9-bit signed offset is added to the contents of GPR base to form the effective address.

The SBE instruction functions in exactly the same fashion as the SB instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the Config5_EVA field being set to one.

**Restrictions:**  
Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

**Operation:**

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
pAddr ← pAddr_{PSIZE-1..2} || (pAddr_{1..0} xor ReverseEndian^2)
bytesel ← vAddr_{1..0} xor BigEndianCPU^2
dataword ← GPR[rt]_{31-8*bytesel..0} || 0^{8*bytesel}
StoreMemory(CCA, BYTE, dataword, pAddr, vAddr, DATA)
```

**Exceptions:**  
TLB Refill  
TLB Invalid  
Bus Error  
Address Error  
Watch  
Reserved Instruction  
Coprocessor Unusable
Store Conditional Word

**Format:** \( SC \ rt, \ offset(base) \)

**Purpose:** Store Conditional Word

To store a word to memory to complete an atomic read-modify-write

**Description:**

\[
\text{if } \text{atomic_update} \text{ then } \text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{GPR}[\text{rt}], \ GPR[\text{rt}] \leftarrow 1 \\
\text{else } \text{GPR}[\text{rt}] \leftarrow 0
\]

The LL and SC instructions provide primitives to implement atomic read-modify-write (RMW) operations on synchronizable memory locations. In Release 5, the behaviour of SC is modified when \( \text{Config5}_{\text{LLB}} = 1 \).

The 32-bit word in GPR \( rt \) is conditionally stored in memory at the location specified by the aligned effective address. The 16-bit signed \( offset \) is added to the contents of GPR \( base \) to form an effective address.

The SC completes the RMW sequence begun by the preceding LL instruction executed on the processor. To complete the RMW sequence atomically, the following occur:

- The 32-bit word of GPR \( rt \) is stored to memory at the location specified by the aligned effective address.

- A one, indicating success, is written into GPR \( rt \).

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR \( rt \).

If either of the following events occurs between the execution of LL and SC, the SC fails:

- A coherent store is completed by another processor or coherent I/O module into the block of synchronizable physical memory containing the word. The size and alignment of the block is implementation-dependent, but it is at least one word and at most the minimum page size.

- A coherent store is executed between an LL and SC sequence on the same processor to the block of synchronizable physical memory containing the word (if \( \text{Config5}_{\text{LLB}} = 1 \); else whether such a store causes the SC to fail is not predictable).

- An ERET instruction is executed. (Release 5 includes ERETNC, which will not cause the SC to fail.)

Furthermore, an SC must always compare its address against that of the LL. An SC will fail if the aligned address of the SC does not match that of the preceding LL.

A load that executes on the processor executing the LL/SC sequence to the block of synchronizable physical memory containing the word, will not cause the SC to fail (if \( \text{Config5}_{\text{LLB}} = 1 \); else such a load may cause the SC to fail).

If any of the events listed below occurs between the execution of LL and SC, the SC may fail where it could have succeeded, i.e., success is not predictable. Portable programs should not cause any of these events.

- A load or store executed on the processor executing the LL and SC that is not to the block of synchronizable physical memory containing the word. (The load or store may cause a cache eviction between the LL and SC that results in SC failure. The load or store does not necessarily have to occur between the LL and SC.)
• Any prefetch that is executed on the processor executing the LL and SC sequence (due to a cache eviction between the LL and SC).

• A non-coherent store executed between an LL and SC sequence to the block of synchronizable physical memory containing the word.

• The instructions executed starting with the LL and ending with the SC do not lie in a 2048-byte contiguous region of virtual memory. (The region does not have to be aligned, other than the alignment required for instruction words.)

CACHE operations that are local to the processor executing the LL/SC sequence will result in unpredictable behaviour of the SC if executed between the LL and SC, that is, they may cause the SC to fail where it could have succeeded. Non-local CACHE operations (address-type with coherent CCA) may cause an SC to fail on either the local processor or on the remote processor in multiprocessor or multi-threaded systems. This definition of the effects of CACHE operations is mandated if \( \text{Config5}_{\text{LLB}} = 1 \). If \( \text{Config5}_{\text{LLB}} = 0 \), then CACHE effects are implementation-dependent.

The following conditions must be true or the result of the SC is not predictable—the SC may fail or succeed (if \( \text{Config5}_{\text{LLB}} = 1 \), then either success or failure is mandated, else the result is UNPREDICTABLE):

• Execution of SC must have been preceded by execution of an LL instruction.

• An RMW sequence executed without intervening events that would cause the SC to fail must use the same address in the LL and SC. The address is the same if the virtual address, physical address, and cacheability & coherency attribute are identical.

Atomic RMW is provided only for synchronizable memory locations. A synchronizable memory location is one that is associated with the state and logic necessary to implement the LL/SC semantics. Whether a memory location is synchronizable depends on the processor and system configurations, and on the memory access type used for the location:

• **Uniprocessor atomicity:** To provide atomic RMW on a single processor, all accesses to the location must be made with memory access type of either cached noncoherent or cached coherent. All accesses must be to one or the other access type, and they may not be mixed.

• **MP atomicity:** To provide atomic RMW among multiple processors, all accesses to the location must be made with a memory access type of cached coherent.

• **I/O System:** To provide atomic RMW with a coherent I/O system, all accesses to the location must be made with a memory access type of cached coherent. If the I/O system does not use coherent memory operations, then atomic RMW cannot be provided with respect to the I/O reads and writes.

**Restrictions:**

The addressed location must have a memory access type of cached noncoherent or cached coherent; if it does not, the result is UNPREDICTABLE.

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

**Operation:**

\[
\text{vAddr} \leftarrow \text{sign_extend} (\text{offset}) + \text{GPR} [\text{base}]
\]

\[
\text{if } \text{vAddr}_{1...0} \neq 0^2 \text{ then}
\]

\[
\text{SignalException} (\text{AddressError})
\]
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
if LLbit then
    StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
endif
GPR[rt] ← 031 || LLbit
LLbit ← 0 // if Config5LLB=1, SC always clears LLbit regardless of address match.

Exceptions:
TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

Programming Notes:
LL and SC are used to atomically update memory locations, as shown below.

L1:
    LL    T1, (T0)  # load counter
    ADDI   T2, T1, 1  # increment
    SC    T2, (T0)  # try to store, checking for atomicity
    BEQ   T2, 0, L1  # if not atomic (0), try again
    NOP   # branch-delay slot

Exceptions between the LL and SC cause SC to fail, so persistent exceptions must be avoided. Some examples of these are arithmetic operations that trap, system calls, and floating point operations that trap or require software emulation assistance.

LL and SC function on a single processor for cached noncoherent memory so that parallel programs can be run on uniprocessor systems that do not support cached coherent memory access types.
Store Conditional Word EVA

**Format:**  \( \text{SCE rt, offset(base)} \)

**Purpose:** Store Conditional Word EVA

To store a word to user mode virtual memory while operating in kernel mode to complete an atomic read-modify-write.

**Description:**

```plaintext
if atomic_update then memory[\text{GPR[base]} + \text{offset}] \leftarrow \text{GPR[rt]}, \text{GPR[rt]} \leftarrow 1
else \text{GPR[rt]} \leftarrow 0
```

The LL and SC instructions provide primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The 32-bit word in GPR \( rt \) is conditionally stored in memory at the location specified by the aligned effective address. The 9-bit signed \( offset \) is added to the contents of GPR \( base \) to form an effective address.

The SCE completes the RMW sequence begun by the preceding LLE instruction executed on the processor. To complete the RMW sequence atomically, the following occur:

- The 32-bit word of GPR \( rt \) is stored to memory at the location specified by the aligned effective address.
- A 1, indicating success, is written into GPR \( rt \).

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR \( rt \).

If either of the following events occurs between the execution of LL and SC, the SC fails:

- A coherent store is completed by another processor or coherent I/O module into the block of synchronizable physical memory containing the word. The size and alignment of the block is implementation dependent, but it is at least one word and at most the minimum page size.
- An ERET instruction is executed.

If either of the following events occurs between the execution of LLE and SCE, the SCE may succeed or it may fail; the success or failure is not predictable. Portable programs should not cause one of these events.

- A memory access instruction (load, store, or prefetch) is executed on the processor executing the LLE/SCE.
- The instructions executed starting with the LLE and ending with the SCE do not lie in a 2048-byte contiguous region of virtual memory. (The region does not have to be aligned, other than the alignment required for instruction words.)

The following conditions must be true or the result of the SCE is **UNPREDICTABLE:**

- Execution of SCE must have been preceded by execution of an LLE instruction.
- An RMW sequence executed without intervening events that would cause the SCE to fail must use the same address in the LLE and SCE. The address is the same if the virtual address, physical address, and cacheability & coherency attribute are identical.

Atomic RMW is provided only for synchronizable memory locations. A synchronizable memory location is one that is associated with the state and logic necessary to implement the LLE/SCE semantics. Whether a memory location is
synchronizable depends on the processor and system configurations, and on the memory access type used for the location:

- **Uniprocessor atomicity:** To provide atomic RMW on a single processor, all accesses to the location must be made with memory access type of either *cached noncoherent* or *cached coherent*. All accesses must be to one or the other access type, and they may not be mixed.

- **MP atomicity:** To provide atomic RMW among multiple processors, all accesses to the location must be made with a memory access type of *cached coherent*.

- **I/O System:** To provide atomic RMW with a coherent I/O system, all accesses to the location must be made with a memory access type of *cached coherent*. If the I/O system does not use coherent memory operations, then atomic RMW cannot be provided with respect to the I/O reads and writes.

The SCE instruction functions in exactly the same fashion as the SC instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the $\text{Config5}_{\text{EVA}}$ field being set to one.

**Restrictions:**

The addressed location must have a memory access type of *cached noncoherent* or *cached coherent*; if it does not, the result is **UNPREDICTABLE**.

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

**Operation:**

\[
\text{vAddr} \leftarrow \text{sign\_extend}\text{(offset)} + \text{GPR}\[\text{base}\]
\]

if $\text{vAddr}_{1..0} \neq 0^2$ then

  SignalException(AddressError)
endif

(pAddr, CCA) $\leftarrow$ AddressTranslation (vAddr, DATA, STORE)

dataword $\leftarrow$ GPR[rt]

if LLbit then

  StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
endif

GPR[rt] $\leftarrow$ 0$^{31} ||$ LLbit

**Exceptions:**

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch, Reserved Instruction, Coprocessor Unusable

**Programming Notes:**

LLE and SCE are used to atomically update memory locations, as shown below.

```
L1:
  LLE  T1, (T0)  # load counter
  ADDI T2, T1, 1 # increment
  SCE  T2, (T0)  # try to store, checking for atomicity
  B EQ T2, 0, L1 # if not atomic (0), try again
  NOP               # branch-delay slot
```
Exceptions between the LLE and SCE cause SCE to fail, so persistent exceptions must be avoided. Some examples of these are arithmetic operations that trap, system calls, and floating point operations that trap or require software emulation assistance.

LLE and SCE function on a single processor for *cached noncoherent* memory so that parallel programs can be run on uniprocessor systems that do not support *cached coherent* memory access types.
Software Debug Breakpoint

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL2</td>
<td>code - use syscall</td>
<td>SDBBP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td></td>
<td>111111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  
`SDBBP code`  
`EJTAG`  

**Purpose:**  
Software Debug Breakpoint  
To cause a debug breakpoint exception

**Description:**  
This instruction causes a debug exception, passing control to the debug exception handler. If the processor is executing in Debug Mode when the SDBBP instruction is executed, the exception is a Debug Mode Exception, which sets the `DebugDExcCode` field to the value 0x9 (Bp). The code field can be used for passing information to the debug exception handler, and is retrieved by the debug exception handler only by loading the contents of the memory word containing the instruction, using the DEPC register. The CODE field is not used in any way by the hardware.

**Restrictions:**

**Operation:**

```plaintext
If DebugDM = 0 then
    SignalDebugBreakpointException()
else
    SignalDebugModeBreakpointException()
endif
```

**Exceptions:**  
Debug Breakpoint Exception  
Debug Mode Breakpoint Exception
Store Doubleword from Floating Point

Format: SDC1 ft, offset(base)

Purpose: Store Doubleword from Floating Point

To store a doubleword from an FPR to memory

Description: memory[GPR[base] + offset] ← FPR[ft]

The 64-bit doubleword in FPR ft is stored in memory at the location specified by the aligned effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

Restrictions:

An Address Error exception occurs if EffectiveAddress2..0 ≠ 0 (not doubleword-aligned).

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr2..0 ≠ 0 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
datadoubleword ← ValueFPR(ft, UNINTERPRETED_DOUBLEWORD)
paddr ← paddr xor ((BigEndianCPU xor ReverseEndian) || 02)
StoreMemory(CCA, WORD, datadoubleword31..0, pAddr, vAddr, DATA)
paddr ← paddr xor 0b100
StoreMemory(CCA, WORD, datadoubleword63..32, pAddr, vAddr+4, DATA)
```

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch
Store Doubleword from Coprocessor 2

**Format:**  SDC2 rt, offset(base)

**Purpose:** Store Doubleword from Coprocessor 2
To store a doubleword from a Coprocessor 2 register to memory

**Description:** memory[GPR[base] + offset] ← CPR[2,rt,0]
The 64-bit doubleword in Coprocessor 2 register rt is stored in memory at the location specified by the aligned effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

**Restrictions:**
An Address Error exception occurs if EffectiveAddress2..0 ≠ 0 (not doubleword-aligned).

**Operation:**
```c
vAddr ← signextend(offset) + GPR[base]
if vAddr2..0 ≠ 0 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
lsw ← CPR[2,rt,0]
msw ← CPR[2,rt+1,0]
paddr ← paddr xor ((BigEndianCPU xor ReverseEndian) || 0^2)
StoreMemory(CCA, WORD, lsw, pAddr, vAddr, DATA)
paddr ← paddr xor 0b100
StoreMemory(CCA, WORD, msw, pAddr, vAddr+4, DATA)
```

**Exceptions:**
Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch
Store Doubleword Indexed from Floating Point

**Format:**  
SDXC1 $fs$, $index(base)$

**Purpose:** Store Doubleword Indexed from Floating Point  
To store a doubleword from an FPR to memory (GPR+GPR addressing)

**Description:**  
$memory[\text{GPR}[\text{base}]+\text{GPR}[\text{index}]] \leftarrow \text{FPR}[fs]$  
The 64-bit doubleword in FPR $fs$ is stored in memory at the location specified by the aligned effective address. The contents of GPR $index$ and GPR $base$ are added to form the effective address.

**Restrictions:**  
An Address Error exception occurs if $\text{EffectiveAddress}_{2,0} \neq 0$ (not doubleword-aligned).

**Compatibility and Availability:**  
SDXC1: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode ($FIR_{F64}=0$ or 1, $Status_{FR}=0$ or 1).

**Operation:**  
\begin{verbatim}
  vAddr ← GPR[base] + GPR[index]
  if vAddr2..0 ≠ 03 then
    SignalException(AddressError)
  endif
  (pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)
  datadoubleword ← ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)
  paddr ← paddr xor ((BigEndianCPU xor ReverseEndian) || 02)
  StoreMemory(CCA, WORD, datadoubleword31..0, pAddr, vAddr, DATA)
  paddr ← paddr xor 0b100
  StoreMemory(CCA, WORD, datadoubleword63..32, pAddr, vAddr+4, DATA)
\end{verbatim}

**Exceptions:**  
TLB Refill, TLB Invalid, TLB Modified, Coprocessor Unusable, Address Error, Reserved Instruction, Watch.
Sign-Extend Byte

SEB

Format: SEB rd, rt

Purpose: Sign-Extend Byte
To sign-extend the least significant byte of GPR rt and store the value into GPR rd.

Description: GPR[rd] ← SignExtend(GPR[rt]7..0)
The least significant byte from GPR rt is sign-extended and stored in GPR rd.

Restrictions:
In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:
GPR[rd] ← sign_extend(GPR[rt]7..0)

Exceptions:
Reserved Instruction

Programming Notes:
For symmetry with the SEB and SEH instructions, one would expect that there would be ZEB and ZEH instructions that zero-extend the source operand. Similarly, one would expect that the SEW and ZEW instructions would exist to sign- or zero-extend a word to a doubleword. These instructions do not exist because there are functionally-equivalent instructions already in the instruction set. The following table shows the instructions providing the equivalent functions.

<table>
<thead>
<tr>
<th>Expected Instruction</th>
<th>Function</th>
<th>Equivalent Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZEB rx,ry</td>
<td>Zero-Extend Byte</td>
<td>ANDI rx,ry,0xFF</td>
</tr>
<tr>
<td>ZEH rx,ry</td>
<td>Zero-Extend Halfword</td>
<td>ANDI rx,ry,0xFFFF</td>
</tr>
</tbody>
</table>
Sign-Extend Halfword

Format: \text{SEH \(rd, rt\)}

MIPS32 Release 2

Purpose: Sign-Extend Halfword

To sign-extend the least significant halfword of GPR \(rt\) and store the value into GPR \(rd\).

Description: \(GPR[rd] \leftarrow \text{SignExtend}(GPR[rt]_{15..0})\)

The least significant halfword from GPR \(rt\) is sign-extended and stored in GPR \(rd\).

Restrictions:

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:

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

Exceptions:

Reserved Instruction

Programming Notes:

The SEH instruction can be used to convert two contiguous halfwords to sign-extended word values in three instructions. For example:

\begin{verbatim}
  lw  t0, 0(al)  /* Read two contiguous halfwords */
  seh t1, t0    /* t1 = lower halfword sign-extended to word */
  sra t0, t0, 16 /* t0 = upper halfword sign-extended to word */
\end{verbatim}

Zero-extended halfwords can be created by changing the SEH and SRA instructions to ANDI and SRL instructions, respectively.

For symmetry with the SEB and SEH instructions, one would expect that there would be ZEB and ZEH instructions that zero-extend the source operand. Similarly, one would expect that the SEW and ZEW instructions would exist to sign- or zero-extend a word to a doubleword. These instructions do not exist because there are functionally-equivalent instructions already in the instruction set. The following table shows the instructions providing the equivalent functions.

<table>
<thead>
<tr>
<th>Expected Instruction</th>
<th>Function</th>
<th>Equivalent Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZEB (rx, ry)</td>
<td>Zero-Extend Byte</td>
<td>ANDI (rx, ry, 0xFF)</td>
</tr>
<tr>
<td>ZEH (rx, ry)</td>
<td>Zero-Extend Halfword</td>
<td>ANDI (rx, ry, 0xFFFF)</td>
</tr>
</tbody>
</table>
Store Halfword

**Format:** \( \text{SH} \text{ rt, offset(base)} \)  

**Purpose:** Store Halfword  
To store a halfword to memory  

**Description:** \( \text{memory}\{\text{GPR}\{\text{base}\} + \text{offset}\} \leftarrow \text{GPR}\{\text{rt}\} \)  
The least-significant 16-bit halfword of register \( \text{rt} \) is stored in memory at the location specified by the aligned effective address. The 16-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form the effective address.

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

**Operation:**
\[
\begin{align*}
vAddr & \leftarrow \text{sign}\_\text{extend}(\text{offset}) + \text{GPR}\{\text{base}\} \\
\text{if } vAddr_0 & \neq 0 \text{ then} \\
& \quad \text{SignalException(AddressError)} \\
\text{endif} \\
(pAddr, CCA) & \leftarrow \text{AddressTranslation}(vAddr, \text{DATA, STORE}) \\
pAddr & \leftarrow pAddr_{\text{PSIZE} - 1..2} || (pAddr_{1..0} \text{xor (ReverseEndian || 0)}) \\
bytesel & \leftarrow vAddr_{1..0} \text{xor (BigEndianCPU || 0)} \\
dataword & \leftarrow \text{GPR}\{\text{rt}\}_{31-8*\text{bytesel}.0 || g^8*\text{bytesel}} \\
\text{StoreMemory(CCA, HALFWORD, dataword, pAddr, vAddr, DATA)}
\end{align*}
\]

**Exceptions:**  
TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch
**Store Halfword EVA**

**Format:** \( \text{SHE} \ rt, \ offset(\text{base}) \)

**Purpose:** Store Halfword EVA

To store a halfword to user mode virtual address space when executing in kernel mode.

**Description:** \( \text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{GPR}[\text{rt}] \)

The least-significant 16-bit halfword of register \( rt \) is stored in memory at the location specified by the aligned effective address. The 9-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form the effective address.

The SHE instruction functions in exactly the same fashion as the SH instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the \( \text{Config5}_{\text{EVA}} \) field being set to one.

**Restrictions:**

Only usable in kernel mode when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

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

**Operation:**

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign} \_ \text{extend}(\text{offset}) + \text{GPR}[\text{base}] \\
(\text{pAddr}, \text{CCA}) & \leftarrow \text{AddressTranslation (vAddr, DATA, STORE)} \\
\text{pAddr} & \leftarrow \text{pAddr}_{\text{PSIZE}-1..2} \ || \ (\text{pAddr}_{1..0} \ xor (\text{ReverseEndian} \ || \ 0)) \\
\text{bytesel} & \leftarrow \text{vAddr}_{1..0} \ xor (\text{BigEndianCPU} \ || \ 0) \\
\text{dataword} & \leftarrow \text{GPR}[\text{rt}]_{31-8*\text{bytesel},0} \ || \ 8*\text{bytesel} \\
\text{StoreMemory (CCA, HALFWORD, dataword, pAddr, vAddr, DATA)}
\end{align*}
\]

**Exceptions:**

- TLB Refill
- TLB Invalid
- Bus Error
- Address Error
- Watch
- Reserved Instruction
- Coprocessor Unusable
Shift Word Left Logical

Format: \texttt{SLL rd, rt, sa} \hspace{1cm} \text{MIPS32}

Purpose: Shift Word Left Logical
To left-shift a word by a fixed number of bits

Description: \( \text{GPR}[rd] \leftarrow \text{GPR}[rt] \ll sa \)
The contents of the low-order 32-bit word of GPR \( rt \) are shifted left, inserting zeros into the emptied bits; the word result is placed in GPR \( rd \). The bit-shift amount is specified by \( sa \).

Restrictions:
None

Operation:
\[
\begin{align*}
s & \leftarrow sa \\
\text{temp} & \leftarrow \text{GPR}[rt]_{(31-s)}..0 | | 0^s \\
\text{GPR}[rd] & \leftarrow \text{temp}
\end{align*}
\]

Exceptions:
None

Programming Notes:
SLL \( r0, r0, 0 \), expressed as NOP, is the assembly idiom used to denote no operation.
SLL \( r0, r0, 1 \), expressed as SSNOP, is the assembly idiom used to denote no operation that causes an issue break on superscalar processors.
Shift Word Left Logical Variable

**Format:**  \( \text{SLLV rd, rt, rs} \)

**MIPS32**

**Purpose:** Shift Word Left Logical Variable
To left-shift a word by a variable number of bits

**Description:** \( \text{GPR[rd]} \leftarrow \text{GPR[rt]} \ll rs \)
The contents of the low-order 32-bit word of GPR \( rt \) are shifted left, inserting zeros into the emptied bits; the result word is placed in GPR \( rd \). The bit-shift amount is specified by the low-order 5 bits of GPR \( rs \).

**Restrictions:**
None

**Operation:**
\[
\begin{align*}
s & \leftarrow \text{GPR[rs]}_{4..0} \\
\text{temp} & \leftarrow \text{GPR[rt]}_{(31-s)\.0} \ | | \ 0^s \\
\text{GPR[rd]} & \leftarrow \text{temp}
\end{align*}
\]

**Exceptions:**
None

**Programming Notes:**
None
### Set on Less Than

**Format:**  
SLT rd, rs, rt

**MIPS32**

**Purpose:** Set on Less Than  
To record the result of a less-than comparison

**Description:**  
GPR[rd] ← (GPR[rs] < GPR[rt])

Compare the contents of GPR rs and GPR rt as signed integers and record the Boolean result of the comparison in GPR rd. If GPR rs is less than GPR rt, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:**
None

**Operation:**

```plaintext
if GPR[rs] < GPR[rt] then
    GPR[rd] ← 0^GPRLEN-1 || 1
else
    GPR[rd] ← 0^GPRLEN
endif
```

**Exceptions:**
None

---

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>GPR[rd]</th>
<th>GPR[rs]</th>
<th>GPR[rt]</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>SPECIAL</td>
<td>rd</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>SLT</td>
<td>00000</td>
<td>101010</td>
</tr>
</tbody>
</table>

- 6 | 5 | 5 | 5 | 6
Set on Less Than Immediate

**Format:**  
SLTI rt, rs, immediate

**MIPS32**

**Purpose:**  
Set on Less Than Immediate

To record the result of a less-than comparison with a constant

**Description:**  
GPR[rt] ← (GPR[rs] < immediate)

Compare the contents of GPR rs and the 16-bit signed immediate as signed integers and record the Boolean result of the comparison in GPR rt. If GPR rs is less than immediate, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:**  
None

**Operation:**  
\[
\begin{align*}
\text{if } \text{GPR[rs]} &< \text{sign_extend(immediate)} \text{ then} \\
\text{GPR[rt]} &\leftarrow 0^{\text{GPRLEN}-1} \mid 1 \\
\text{else} \\
\text{GPR[rt]} &\leftarrow 0^\text{GPRLEN}
\end{align*}
\]

**Exceptions:**  
None
**Set on Less Than Immediate Unsigned**  

**Format:**  
```
SLTIU rt, rs, immediate
```

**Purpose:** Set on Less Than Immediate Unsigned  
To record the result of an unsigned less-than comparison with a constant

**Description:**  
```
GPR[rt] ← (GPR[rs] < immediate)
```

Compare the contents of GPR `rs` and the sign-extended 16-bit `immediate` as unsigned integers and record the Boolean result of the comparison in GPR `rt`. If GPR `rs` is less than `immediate`, the result is 1 (true); otherwise, it is 0 (false).

Because the 16-bit `immediate` is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:**  
None

**Operation:**
```java
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
    GPR[rt] ← 0^{GPRLEN-1} || 1
else
    GPR[rt] ← 0^{GPRLEN}
endif
```

**Exceptions:**  
None
Set on Less Than Unsigned

**Format:** `SLTU rd, rs, rt`

**MIPS32**

**Purpose:** Set on Less Than Unsigned

To record the result of an unsigned less-than comparison

**Description:** `GPR[rd] ← (GPR[rs] < GPR[rt])`

Compare the contents of GPR `rs` and GPR `rt` as unsigned integers and record the Boolean result of the comparison in GPR `rd`. If GPR `rs` is less than GPR `rt`, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:**

None

**Operation:**

```plaintext
if (0 || GPR[rs]) < (0 || GPR[rt]) then
    GPR[rd] ← 0GPRLEN-1 || 1
else
    GPR[rd] ← 0GPRLEN
endif
```

**Exceptions:**

None
Floating Point Square Root

**Format:**  
\[
\begin{array}{cccc}
\text{COP1} & \text{fmt} & \text{fs} & \text{fd} & \text{SQRT} \\
010001 & 00000 & 0 & 0 & 000100 \\
\end{array}
\]

**Purpose:** Floating Point Square Root  
To compute the square root of an FP value

**Description:**  
\[
\text{FP}[fd] \leftarrow \text{SQRT(FP}[fs])
\]
The square root of the value in FP \(fs\) is calculated to infinite precision, rounded according to the current rounding mode in FCSR, and placed into FP \(fd\). The operand and result are values in format \(fmt\).

If the value in FP \(fs\) corresponds to \(-0\), the result is \(-0\).

**Restrictions:**  
If the value in FP \(fs\) is less than 0, an Invalid Operation condition is raised.  
The fields \(fs\) and \(fd\) must specify FPRs valid for operands of type \(fmt\); if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format \(fmt\); if it is not, the result is UNPREDICTABLE and the value of the operand FP becomes UNPREDICTABLE.

**Operation:**  
\[
\text{StoreFP}(fd, fmt, \text{SquareRoot(ValueFP}(fs, fmt)))
\]

**Exceptions:**  
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**  
Invalid Operation, Inexact, Unimplemented Operation
## Shift Word Right Arithmetic

**Format:**  
```
SRA rd, rt, sa
```

**Purpose:** Shift Word Right Arithmetic  
To execute an arithmetic right-shift of a word by a fixed number of bits

**Description:**  
```
GPR[rd] ← GPR[rt] >> sa  (arithmetic)
```

The contents of the low-order 32-bit word of GPR rt are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR rd. The bit-shift amount is specified by sa.

**Restrictions:**
None

**Operation:**
```
s ← sa
```
```
temp ← (GPR[rt]31)^s || GPR[rt]31..s
```
```
GPR[rd] ← temp
```

**Exceptions:**
None
Shift Word Right Arithmetic Variable  SRAV

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>SRAV</td>
<td>000111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** SRAV rd, rt, rs

**Purpose:** Shift Word Right Arithmetic Variable

To execute an arithmetic right-shift of a word by a variable number of bits

**Description:**

\[ GPR[rd] \leftarrow GPR[rt] >> rs \quad \text{(arithmetic)} \]

The contents of the low-order 32-bit word of GPR rt are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR rd. The bit-shift amount is specified by the low-order 5 bits of GPR rs.

**Restrictions:**

None

**Operation:**

\[
\begin{align*}
    s & \leftarrow GPR[rs]_{4..0} \\
    \text{temp} & \leftarrow (GPR[rt]_{31})^s || GPR[rt]_{31..s} \\
    GPR[rd] & \leftarrow \text{temp}
\end{align*}
\]

**Exceptions:**

None
Shift Word Right Logical

**Format:**  \( \text{SRL } rd, rt, sa \)

**Purpose:** Shift Word Right Logical

To execute a logical right-shift of a word by a fixed number of bits

**Description:** \( \text{GPR}[rd] \leftarrow \text{GPR}[rt] \gg sa \) (logical)

The contents of the low-order 32-bit word of GPR \( rt \) are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR \( rd \). The bit-shift amount is specified by \( sa \).

**Restrictions:**
None

**Operation:**

\[
\begin{align*}
\text{s} & \leftarrow sa \\
\text{temp} & \leftarrow 0^n \| \text{GPR}[rt]_{31..s} \\
\text{GPR}[rd] & \leftarrow \text{temp}
\end{align*}
\]

**Exceptions:**

None
Shift Word Right Logical Variable

**Format:**  SRLV rd, rt, rs

**Purpose:**  Shift Word Right Logical Variable
To execute a logical right-shift of a word by a variable number of bits

**Description:**  GPR[rd] ← GPR[rt] >> GPR[rs]  (logical)
The contents of the low-order 32-bit word of GPR rt are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR rd. The bit-shift amount is specified by the low-order 5 bits of GPR rs.

**Restrictions:**  None

**Operation:**

```plaintext
s ← GPR[rs]4..0
temp ← 0s || GPR[rt]31..s
GPR[rd] ← temp
```

**Exceptions:**  None
Superscalar No Operation

**Format:** SSNOP

**Purpose:** Superscalar No Operation

Break superscalar issue on a superscalar processor.

**Description:**

SSNOP is the assembly idiom used to denote superscalar no operation. The actual instruction is interpreted by the hardware as SLL r0, r0, 1.

This instruction alters the instruction issue behavior on a superscalar processor by forcing the SSNOP instruction to single-issue. The processor must then end the current instruction issue between the instruction previous to the SSNOP and the SSNOP. The SSNOP then issues alone in the next issue slot.

On a single-issue processor, this instruction is a NOP that takes an issue slot.

**Restrictions:**

None

**Operation:**

None

**Exceptions:**

None

**Programming Notes:**

SSNOP is intended for use primarily to allow the programmer control over CP0 hazards by converting instructions into cycles in a superscalar processor. For example, to insert at least two cycles between an MTC0 and an ERET, one would use the following sequence:

```
mtc0 x, y
ssnop
ssnop
eret
```

Based on the normal issues rules of the processor, the MTC0 issues in cycle T. Because the SSNOP instructions must issue alone, they may issue no earlier than cycle T+1 and cycle T+2, respectively. Finally, the ERET issues no earlier than cycle T+3. Note that although the instruction after an SSNOP may issue no earlier than the cycle after the SSNOP is issued, that instruction may issue later. This is because other implementation-dependent issue rules may apply that prevent an issue in the next cycle. Processors should not introduce any unnecessary delay in issuing SSNOP instructions.
Subtract Word

**Format:** \texttt{SUB rd, rs, rt}

**Purpose:** Subtract Word

To subtract 32-bit integers. If overflow occurs, then trap

**Description:** \( \text{GPR}[rd] \leftarrow \text{GPR}[rs] - \text{GPR}[rt] \)

The 32-bit word value in GPR \( rt \) is subtracted from the 32-bit value in GPR \( rs \) to produce a 32-bit result. If the subtraction results in 32-bit 2’s complement arithmetic overflow, then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR \( rd \).

**Restrictions:**
None

**Operation:**

\[
\text{temp} \leftarrow (\text{GPR}[rs]_{31}|\text{GPR}[rs]_{31..0}) - (\text{GPR}[rt]_{31}|\text{GPR}[rt]_{31..0})
\]

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

**Exceptions:**

Integer Overflow

**Programming Notes:**

SUBU performs the same arithmetic operation but does not trap on overflow.
### Floating Point Subtract

#### Format:

<table>
<thead>
<tr>
<th>SUB.fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB.S fd, fs, ft</td>
</tr>
<tr>
<td>SUB.D fd, fs, ft</td>
</tr>
<tr>
<td>SUB.PS fd, fs, ft</td>
</tr>
</tbody>
</table>

#### Purpose:

Floating Point Subtract

To subtract FP values

#### Description:

\[ \text{FPR}[fd] \leftarrow \text{FPR}[fs] - \text{FPR}[ft] \]

The value in FPR \( ft \) is subtracted from the value in FPR \( fs \). The result is calculated to infinite precision, rounded according to the current rounding mode in \( FCSR \), and placed into FPR \( fd \). The operands and result are values in format \( fmt \). SUB.PS subtracts the upper and lower halves of FPR \( fs \) and FPR \( ft \) independently, and ORs together any generated exceptional conditions.

#### Restrictions:

The fields \( fs, ft, \) and \( fd \) must specify FPRs valid for operands of type \( fmt \). If they are not valid, the result is \textit{UNPREDICTABLE}.

The operands must be values in format \( fmt \); if they are not, the result is \textit{UNPREDICTABLE} and the value of the operand FPRs becomes \textit{UNPREDICTABLE}.

The result of SUB.PS is \textit{UNPREDICTABLE} if the processor is executing in the FR=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the \( FR=1 \) mode, but not with \( FR=0 \), and not on a 32-bit FPU.

#### Operation:

\[ \text{StoreFPR} (fd, fmt, \text{ValueFPR}(fs, fmt) -_{fmt} \text{ValueFPR}(ft, fmt)) \]

#### CPU Exceptions:

Coprocessor Unusable, Reserved Instruction

#### FPU Exceptions:

Inexact, Overflow, Underflow, Invalid Op, Unimplemented Op
Subtract Unsigned Word

Subtract Unsigned Word

Purpose: Subtract Unsigned Word
To subtract 32-bit integers

Description: \( \text{GPR}[rd] \leftarrow \text{GPR}[rs] - \text{GPR}[rt] \)
The 32-bit word value in GPR \( rt \) is subtracted from the 32-bit value in GPR \( rs \) and the 32-bit arithmetic result is and placed into GPR \( rd \).
No integer overflow exception occurs under any circumstances.

Restrictions:
None

Operation:
\[
\text{temp} \leftarrow \text{GPR}[rs] - \text{GPR}[rt] \\
\text{GPR}[rd] \leftarrow \text{temp}
\]

Exceptions:
None

Programming Notes:
The term "unsigned" in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.
Store Doubleword Indexed Unaligned from Floating Point

Format: SUXC1 fs, index(base)

Purpose: Store Doubleword Indexed Unaligned from Floating Point
To store a doubleword from an FPR to memory (GPR+GPR addressing) ignoring alignment

Description: memory[(GPR[base] + GPR[index])_{PSIZE-1..3}] ← FPR[fs]

The contents of the 64-bit doubleword in FPR fs is stored at the memory location specified by the effective address. The contents of GPR index and GPR base are added to form the effective address. The effective address is double-word-aligned; EffectiveAddress_{2,0} are ignored.

Restrictions:
The result of this instruction is UNPREDICTABLE if the processor is executing in the FR=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the FR=1 mode, but not with FR=0, and not on a 32-bit FPU.

Operation:
\[ \text{vAddr} \leftarrow \text{(GPR[base]+GPR[index])}_{63..3} | | 0^3 \]
\[ \text{(pAddr, CCA)} \leftarrow \text{AddressTranslation(vAddr, DATA, STORE)} \]
\[ \text{datadoubleword} \leftarrow \text{ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)} \]
\[ \text{paddr} \leftarrow \text{paddr xor ((BigEndianCPU xor ReverseEndian) | | 0^2)} \]
\[ \text{StoreMemory(CCA, WORD, datadoubleword}_{31..0}, \text{pAddr, vAddr, DATA)} \]
\[ \text{paddr} \leftarrow \text{paddr xor 0b100} \]
\[ \text{StoreMemory(CCA, WORD, datadoubleword}_{63..32}, \text{pAddr, vAddr+4, DATA)} \]

Exceptions:
Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Watch
Store Word

Format: \textit{SW rt, offset(base)}

\textbf{Purpose:} Store Word

To store a word to memory

\textbf{Description:} \texttt{memory[GP[R\text{base}] + offset] \leftarrow GPR[rt]}

The least-significant 32-bit word of GPR \textit{rt} is stored in memory at the location specified by the aligned effective address. The 16-bit signed \textit{offset} is added to the contents of GPR \textit{base} to form the effective address.

\textbf{Restrictions:}

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

\textbf{Operation:}

\begin{verbatim}
vAddr \leftarrow \text{sign\_extend(offset)} + GPR[base]
if vAddr_{1...0} \neq 0^2 then
    \text{SignalException(AddressError)}
endif
(pAddr, CCA) \leftarrow \text{AddressTranslation (vAddr, DATA, STORE)}
dataword \leftarrow GPR[rt]
\text{StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)}
\end{verbatim}

\textbf{Exceptions:}

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch
**Store Word from Floating Point**

**Purpose:** Store Word from Floating Point

To store a word from an FPR to memory

**Description:** memory[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{FPR}[\text{ft}]

The low 32-bit word from FPR $ft$ is stored in memory at the location specified by the aligned effective address. The 16-bit signed $\text{offset}$ is added to the contents of GPR $\text{base}$ to form the effective address.

**Restrictions:**

An Address Error exception occurs if EffectiveAddress$1..0 \neq 0$ (not word-aligned).

**Operation:**

$$\text{vAddr} \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}]$$

if $\text{vAddr}_{1..0} \neq 0^3$ then

   SignalException(AddressError)

endif

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

$\text{dataword} \leftarrow \text{ValueFPR}(\text{ft}, \text{UNINTERPRETED\_WORD})$

$\text{StoreMemory}(\text{CCA}, \text{WORD}, \text{dataword}, \text{pAddr}, \text{vAddr}, \text{DATA})$

**Exceptions:**

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch
Store Word from Coprocessor 2

Format: \texttt{SWC2 \, rt, \, offset(base)}

Purpose: Store Word from Coprocessor 2

To store a word from a COP2 register to memory

Description: \texttt{memory[\text{GPR}[\text{base}] + offset] \leftarrow \text{CPR}[2,rt,0]}

The low 32-bit word from COP2 (Coprocessor 2) register \texttt{rt} is stored in memory at the location specified by the aligned effective address. The 16-bit signed \texttt{offset} is added to the contents of GPR \texttt{base} to form the effective address.

Restrictions:

An Address Error exception occurs if EffectiveAddress_{1,0} \neq 0 (not word-aligned).

Operation:

\begin{verbatim}
     vAddr \leftarrow \text{sign\_extend}(offset) + \text{GPR}[\text{base}]
     \text{if } vAddr_{1,0} \neq 0^2 \text{ then}
         \text{SignalException(AddressError)}
     \text{endif}
     (pAddr, CCA) \leftarrow \text{AddressTranslation}(vAddr, DATA, STORE)
     \text{dataword} \leftarrow \text{CPR}[2,rt,0]
     \text{StoreMemory}(CCA, \text{WORD, dataword, pAddr, vAddr, DATA})
\end{verbatim}

Exceptions:

Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
31 & 26 & 25 & 21 20 16 15  0  \\
\hline
SWC2 & base & rt & offset \\
\hline
111010 & 5 5 & 16 \\
\hline
\end{tabular}
\end{center}
Store Word EVA

**Format:**  
*SWE rt, offset(base)*

**Purpose:**  
To store a word to user mode virtual address space when executing in kernel mode.

**Description:**  
memory[\(GPR[base] + offset\)] ← \(GPR[rt]\)

The least-significant 32-bit word of GPR \(rt\) is stored in memory at the location specified by the aligned effective address. The 9-bit signed \(offset\) is added to the contents of GPR \(base\) to form the effective address.

The SWE instruction functions in exactly the same fashion as the SW instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the \(Config_{EVA}\) field being set to one.

**Restrictions:**

Only usable in kernel mode when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

**Operation:**

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign
double}_\text{extend}(\text{offset}) + \text{GPR}\[\text{base}] \\
(\text{pAddr, CCA}) & \leftarrow \text{AddressTranslation (vAddr, DATA, STORE)} \\
\text{dataword} & \leftarrow \text{GPR}\[\text{rt}] \\
\text{StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)}
\end{align*}
\]

**Exceptions:**

- TLB Refill
- TLB Invalid
- Bus Error
- Address Error
- Watch
- Reserved Instruction
- Coprocessor Unusable
Store Word Left

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWL</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** SWL rt, offset(base)

**Purpose:** Store Word Left

To store the most-significant part of a word to an unaligned memory address

**Description:** memory[GPR[base] + offset] ← GPR[rt]

The 16-bit signed offset is added to the contents of GPR base to form an effective address (EffAddr). EffAddr is the address of the most-significant of 4 consecutive bytes forming a word (W) in memory starting at an arbitrary byte boundary.

A part of W, the most-significant 1 to 4 bytes, is in the aligned word containing EffAddr. The same number of the most-significant (left) bytes from the word in GPR rt are stored into these bytes of W.

The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2.5 form an unaligned word starting at location 2. A part of W, 2 bytes, is located in the aligned word containing the most-significant byte at 2. First, SWL stores the most-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWR stores the remainder of the unaligned word.

**Figure 4.12 Unaligned Word Store Using SWL and SWR**

| Word at byte 2 in memory, big-endian byte order; each memory byte contains its own address |
| most | — significance — | least |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

| Memory: Initial contents |
| GPR 24 |
| E | F | G | H |
| 0 | 1 | E | F | 4 | 5 | 6 | ... |

After executing SWL $24, 2 ($0)

Then after SWR $24, 5 ($0)

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address (vAddrL_0)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte ordering.
Figure 4.13 Bytes Stored by an SWL Instruction

<table>
<thead>
<tr>
<th>Memory contents and byte offsets</th>
<th>Initial contents of Dest Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>ABCDEFGH</td>
</tr>
<tr>
<td>Big-endian byte ordering</td>
<td>Little-endian byte ordering</td>
</tr>
<tr>
<td>vAddr ← sign_extend(offset) + GPR[base]</td>
<td></td>
</tr>
<tr>
<td>(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)</td>
<td></td>
</tr>
<tr>
<td>pAddr ← pAddr_{PSIZE-1..2}</td>
<td></td>
</tr>
<tr>
<td>If BigEndianMem = 0 then</td>
<td></td>
</tr>
<tr>
<td>pAddr ← pAddr_{PSIZE-1..2}</td>
<td></td>
</tr>
<tr>
<td>endif</td>
<td></td>
</tr>
<tr>
<td>byte ← vAddr_{1..0} xor BigEndianCPU^2</td>
<td></td>
</tr>
<tr>
<td>dataword ← 0_{24-8*byte}</td>
<td></td>
</tr>
<tr>
<td>StoreMemory(CCA, byte, dataword, pAddr, vAddr, DATA)</td>
<td></td>
</tr>
</tbody>
</table>

Exceptions:
TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Watch

Restrictions:
None

Operation:

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_{PSIZE-1..2} || (pAddr_{1..0} xor ReverseEndian^2)
If BigEndianMem = 0 then
    pAddr ← pAddr_{PSIZE-1..2} || 0^2
endif
byte ← vAddr_{1..0} xor BigEndianCPU^2
dataword ← 0_{24-8*byte} || GPR[rt]_{31..24-8*byte}
StoreMemory(CCA, byte, dataword, pAddr, vAddr, DATA)
**Format:** SWLE rt, offset(base)  

**Purpose:** Store Word Left EVA  
To store the most-significant part of a word to an unaligned user mode virtual address while operating in kernel mode.

**Description:** memory[GPR(base) + offset] ← GPR[rt]  
The 9-bit signed offset is added to the contents of GPR base to form an effective address (EffAddr). EffAddr is the address of the most-significant of 4 consecutive bytes forming a word (W) in memory starting at an arbitrary byte boundary.

A part of W, the most-significant 1 to 4 bytes, is in the aligned word containing EffAddr. The same number of the most-significant (left) bytes from the word in GPR rt are stored into these bytes of W.

The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2.5 form an unaligned word starting at location 2. A part of W, 2 bytes, is located in the aligned word containing the most-significant byte at 2. First, SWLE stores the most-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWRE stores the remainder of the unaligned word.

**Figure 4.14 Unaligned Word Store Using SWLE and SWRE**

![Diagram of memory and registers illustrating the operation of SWLE and SWRE](image)

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address (vAddr1,0)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte ordering.

The SWLE instruction functions in exactly the same fashion as the SWL instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the Config5EVA field being set to one.
Restrictions:
Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Operation:
\[
\begin{align*}
vAddr & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR[base]} \\
(pAddr, CCA) & \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{STORE}) \\
pAddr & \leftarrow pAddr_{\text{PSIZE}-1..2} \| (pAddr_{1..0} \text{ xor ReverseEndian}^2) \\
\text{If BigEndianMem} = 0 \text{ then} \\
pAddr & \leftarrow pAddr_{\text{PSIZE}-1..2} \| 0^2 \\
\text{endif} \\
\text{byte} & \leftarrow vAddr_{1..0} \text{ xor BigEndianCPU}^2 \\
\text{dataword} & \leftarrow 0_{24-8^*\text{byte}} \| \text{GPR[rt]}_{31..24-8^*\text{byte}} \\
\text{StoreMemory}(CCA, \text{byte, dataword, pAddr, vAddr, DATA})
\end{align*}
\]

Exceptions:
TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Watch, Reserved Instruction, Coprocessor Unusable
**Store Word Right**

**Format:**  \( \text{SWR} \ rt, \ \text{offset}(\text{base}) \)  

**Purpose:** Store Word Right  
To store the least-significant part of a word to an unaligned memory address

**Description:**  
\[ \text{memory}[\text{GPR[base]} + \ \text{offset}] \leftarrow \text{GPR}[rt] \]

The 16-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form an effective address \( \text{EffAddr} \). \( \text{EffAddr} \) is the address of the least-significant of 4 consecutive bytes forming a word \( \( W \) \) in memory starting at an arbitrary byte boundary.

A part of \( W \), the least-significant 1 to 4 bytes, is in the aligned word containing \( \text{EffAddr} \). The same number of the least-significant (right) bytes from the word in GPR \( rt \) are stored into these bytes of \( W \).

The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of \( W \), 2 bytes, is contained in the aligned word containing the least-significant byte at 5. First, SWR stores the least-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWL stores the remainder of the unaligned word.

**Figure 4.16 Unaligned Word Store Using SWR and SWL**

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address \( (\text{Addr}_{1,0}) \)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte-ordering.
Figure 4.17 Bytes Stored by SWR Instruction

<table>
<thead>
<tr>
<th>Memory contents and byte offsets</th>
<th>Initial contents of Dest Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>i j k l offset (vAddr_1, 0)</td>
<td>A B C D E F G H</td>
</tr>
<tr>
<td>3 2 1 0 ← little-endian</td>
<td>most — significance — least</td>
</tr>
<tr>
<td>most least — significance —</td>
<td>32-bit register E F G H</td>
</tr>
</tbody>
</table>

Memory contents after instruction (shaded is unchanged)

<table>
<thead>
<tr>
<th>Big-endian byte ordering</th>
<th>vAddr_1, 0</th>
<th>Little-endian byte ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>H j k l</td>
<td>0</td>
<td>E F G H</td>
</tr>
<tr>
<td>G H k l</td>
<td>1</td>
<td>F G H l</td>
</tr>
<tr>
<td>F G H i</td>
<td>2</td>
<td>G H k l</td>
</tr>
<tr>
<td>E F G H</td>
<td>3</td>
<td>H j k l</td>
</tr>
</tbody>
</table>

Restrictions:
None

Operation:

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_{PSIZE-1..2} || (pAddr_1..0 xor ReverseEndian^2)
If BigEndianMem = 0 then
  pAddr ← pAddr_{PSIZE-1..2} || 0^2
endif
byte ← vAddr_1..0 xor BigEndianCPU^2
dataword ← GPR[rt]_{31-8}*byte || 0^8*byte
StoreMemory(CCA, WORD-byte, dataword, pAddr, vAddr, DATA)

Exceptions:
TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Watch
Store Word Right EVA

![32-bit Table](image)

**Format:** \( \text{SWRE \ rt, offset(base)} \)

**Purpose:** Store Word Right EVA

To store the least-significant part of a word to an unaligned user mode virtual address while operating in kernel mode.

**Description:** \( \text{memory[GRP[base] + offset]} \leftarrow \text{GPR[rt]} \)

The 9-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form an effective address (\( \text{EffAddr} \)). \( \text{EffAddr} \) is the address of the least-significant of 4 consecutive bytes forming a word (\( W \)) in memory starting at an arbitrary byte boundary.

A part of \( W \), the least-significant 1 to 4 bytes, is in the aligned word containing \( \text{EffAddr} \). The same number of the least-significant (right) bytes from the word in GPR \( rt \) are stored into these bytes of \( W \).

The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2.5 form an unaligned word starting at location 2. A part of \( W \), 2 bytes, is contained in the aligned word containing the least-significant byte at 5. First, \( \text{SWRE} \) stores the least-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary \( \text{SWLE} \) stores the remainder of the unaligned word.

**Figure 4.18 Unaligned Word Store Using SWRE and SWLE**

![Diagram](image)

The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address (\( \text{vAddr}_{1,0} \))—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte-ordering.

The \( \text{LWE} \) instruction functions in exactly the same fashion as the \( \text{LW} \) instruction, except that address translation is performed using the user mode virtual address space mapping in the TLB when accessing an address within a memory segment configured to use the MUSUK access mode. Memory segments using UUSK or MUSK access modes are also accessible. Refer to Volume III, Enhanced Virtual Addressing section for additional information.

Implementation of this instruction is specified by the \( \text{Config}_{EVA} \) field being set to one.
Restrictions:
Only usable when access to Coprocessor0 is enabled and when accessing an address within a segment configured using UUSK, MUSK or MUSUK access mode.

Operation:
\[
vAddr \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}]
\]
\[
(pAddr, \text{CCA}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{STORE})
\]
\[
pAddr \leftarrow pAddr_{\text{SIZE}-1..2} || (pAddr_{1..0} \text{ xor ReverseEndian}^2)
\]
If BigEndianMem = 0 then
\[
pAddr \leftarrow pAddr_{\text{SIZE}-1..2} || 0^2
\]
endif
byte $\leftarrow vAddr_{1..0} \text{ xor BigEndianCPU}^2$
dataword $\leftarrow \text{GPR}[rt]_{31-8*\text{byte}} || 0^8*\text{byte}$
StoreMemory(CCA, WORD=byte, dataword, pAddr, vAddr, DATA)

Exceptions:
TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Watch, Coprocessor Unusable
Store Word Indexed from Floating Point

**Format:** \( \text{SWXC1} \ fs, \text{index(base)} \)

**Purpose:** Store Word Indexed from Floating Point
To store a word from an FPR to memory (GPR+GPR addressing)

**Description:** \( \text{memory}[GPR[base] + GPR[index]] \leftarrow FPR[fs] \)
The low 32-bit word from FPR \( fs \) is stored in memory at the location specified by the aligned effective address. The contents of GPR \( index \) and GPR \( base \) are added to form the effective address.

**Restrictions:**
An Address Error exception occurs if EffectiveAddress\(_{1,0} \neq 0 \) (not word-aligned).

**Compatibility and Availability:**
SWXC1: Required in all versions of MIPS64 since MIPS64r1. Not available in MIPS32r1. Required by MIPS32r2 and subsequent versions of MIPS32. When required, required whenever FPU is present, whether a 32-bit or 64-bit FPU, whether in 32-bit or 64-bit FP Register Mode \((FIR_{F64}=0 \text{ or } 1, \ FR=0 \text{ or } 1,)\)

**Operation:**
\[ \text{vAddr} \leftarrow GPR[base] + GPR[index] \]
if \( \text{vAddr}_{1,0} \neq 0 \) then
   Exception(AddressError)
endif
\( (\text{pAddr}, \text{CCA}) \leftarrow \text{AddressTranslation}(\text{vAddr}, \text{DATA}, \text{STORE}) \)
\( \text{dataword} \leftarrow \text{ValueFPR}(fs, \text{UNINTERPRETED\_WORD}) \)
\( \text{StoreMemory(CCA, WORD, dataword, pAddr, vAddr, DATA)} \)

**Exceptions:**
TLB Refill, TLB Invalid, TLB Modified, Address Error, Reserved Instruction, Coprocessor Unusable, Watch
Format:  
SYNC {stype = 0 implied}  
SYNC stype

Purpose:  To order loads and stores for shared memory.

Description:

These types of ordering guarantees are available through the SYNC instruction:

- Completion Barriers
- Ordering Barriers

Simple Description for Completion Barrier:

- The barrier affects only *uncached* and *cached coherent* loads and stores.
- The specified memory instructions (loads or stores or both) that occur before the SYNC instruction must be completed before the specified memory instructions after the SYNC are allowed to start.
- Loads are completed when the destination register is written. Stores are completed when the stored value is visible to every other processor in the system.

Detailed Description for Completion Barrier:

- Every synchronizable specified memory instruction (loads or stores or both) that occurs in the instruction stream before the SYNC instruction must be already globally performed before any synchronizable specified memory instructions that occur after the SYNC are allowed to be performed, with respect to any other processor or coherent I/O module.
- The barrier does not guarantee the order in which instruction fetches are performed.
- A stype value of zero will always be defined such that it performs the most complete set of synchronization operations that are defined. This means stype zero always does a completion barrier that affects both loads and stores preceding the SYNC instruction and both loads and stores that are subsequent to the SYNC instruction. Non-zero values of stype may be defined by the architecture or specific implementations to perform synchronization behaviors that are less complete than that of stype zero. If an implementation does not use one of these non-zero values to define a different synchronization behavior, then that non-zero value of stype must act the same as stype zero completion barrier. This allows software written for an implementation with a lighter-weight barrier to work on another implementation which only implements the stype zero completion barrier.
- A completion barrier is required, potentially in conjunction with SSNOP (in Release 1 of the Architecture) or EHB (in Release 2 of the Architecture), to guarantee that memory reference results are visible across operating mode changes. For example, a completion barrier is required on some implementations on entry to and exit from Debug Mode to guarantee that memory effects are handled correctly.
SYNC behavior when the stype field is zero:

- A completion barrier that affects preceding loads and stores and subsequent loads and stores.

Simple Description for Ordering Barrier:

- The barrier affects only uncached and cached coherent loads and stores.
- The specified memory instructions (loads or stores or both) that occur before the SYNC instruction must always be ordered before the specified memory instructions after the SYNC.
- Memory instructions which are ordered before other memory instructions are processed by the load/store datapath first before the other memory instructions.

Detailed Description for Ordering Barrier:

- Every synchronizable specified memory instruction (loads or stores or both) that occurs in the instruction stream before the SYNC instruction must reach a stage in the load/store datapath after which no instruction re-ordering is possible before any synchronizable specified memory instruction which occurs after the SYNC instruction in the instruction stream reaches the same stage in the load/store datapath.

- If any memory instruction before the SYNC instruction in program order, generates a memory request to the external memory and any memory instruction after the SYNC instruction in program order also generates a memory request to external memory, the memory request belonging to the older instruction must be globally performed before the time the memory request belonging to the younger instruction is globally performed.

- The barrier does not guarantee the order in which instruction fetches are performed.

As compared to the completion barrier, the ordering barrier is a lighter-weight operation as it does not require the specified instructions before the SYNC to be already completed. Instead it only requires that those specified instructions which are subsequent to the SYNC in the instruction stream are never re-ordered for processing ahead of the specified instructions which are before the SYNC in the instruction stream. This potentially reduces how many cycles the barrier instruction must stall before it completes.

The Acquire and Release barrier types are used to minimize the memory orderings that must be maintained and still have software synchronization work.

Implementations that do not use any of the non-zero values of stype to define different barriers, such as ordering barriers, must make those stype values act the same as stype zero.

For the purposes of this description, the CACHE, PREF and PREFX instructions are treated as loads and stores. That is, these instructions and the memory transactions sourced by these instructions obey the ordering and completion rules of the SYNC instruction.
Table 4.7 lists the available completion barrier and ordering barriers behaviors that can be specified using the stype field.

**Table 4.7 Encodings of the Bits[10:6] of the SYNC instruction; the SType Field**

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Older instructions which must reach the load/store ordering point before the SYNC instruction completes.</th>
<th>Younger instructions which must reach the load/store ordering point only after the SYNC instruction completes.</th>
<th>Older instructions which must be globally performed when the SYNC instruction completes</th>
<th>Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>SYNC or SYNC 0</td>
<td>Loads, Stores</td>
<td>Loads, Stores</td>
<td>Loads, Stores</td>
<td>Required</td>
</tr>
<tr>
<td>0x4</td>
<td>SYNC_WMB or SYNC 4</td>
<td>Stores</td>
<td>Stores</td>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td>0x10</td>
<td>SYNC_MB or SYNC 16</td>
<td>Loads, Stores</td>
<td>Loads, Stores</td>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td>0x11</td>
<td>SYNC_ACQUIRE or SYNC 17</td>
<td>Loads</td>
<td>Loads, Stores</td>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td>0x12</td>
<td>SYNC_RELEASE or SYNC 18</td>
<td>Loads, Stores</td>
<td>Stores</td>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td>0x13</td>
<td>SYNC_RMB or SYNC 19</td>
<td>Loads</td>
<td>Loads</td>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td>0x1-0x3, 0x5-0xF</td>
<td></td>
<td></td>
<td></td>
<td>Implementation-Specific and Vendor Specific Sync Types</td>
<td></td>
</tr>
<tr>
<td>0x14 - 0x1F</td>
<td>RESERVED</td>
<td></td>
<td></td>
<td>Required for MIPS Technologies for future extension of the architecture.</td>
<td></td>
</tr>
</tbody>
</table>

**Terms:**

*Synchronizable:* A load or store instruction is *synchronizable* if the load or store occurs to a physical location in shared memory using a virtual location with a memory access type of either *uncached* or *cached coherent*. *Shared memory* is memory that can be accessed by more than one processor or by a coherent I/O system module.

*Performed load:* A load instruction is *performed* when the value returned by the load has been determined. The result of a load on processor A has been *determined* with respect to processor or coherent I/O module B when a subsequent store to the location by B cannot affect the value returned by the load. The store by B must use the same memory access type as the load.
Performed store: A store instruction is performed when the store is observable. A store on processor A is observable with respect to processor or coherent I/O module B when a subsequent load of the location by B returns the value written by the store. The load by B must use the same memory access type as the store.

Globally performed load: A load instruction is globally performed when it is performed with respect to all processors and coherent I/O modules capable of storing to the location.

Globally performed store: A store instruction is globally performed when it is globally observable. It is globally observable when it is observable by all processors and I/O modules capable of loading from the location.

Coherent I/O module: A coherent I/O module is an Input/Output system component that performs coherent Direct Memory Access (DMA). It reads and writes memory independently as though it were a processor doing loads and stores to locations with a memory access type of cached coherent.

Load/Store Datapath: The portion of the processor which handles the load/store data requests coming from the processor pipeline and processes those requests within the cache and memory system hierarchy.

Restrictions:
The effect of SYNC on the global order of loads and stores for memory access types other than uncached and cached coherent is UNPREDICTABLE.

Operation:
SyncOperation(stype)

Exceptions:
None

Programming Notes:
A processor executing load and store instructions observes the order in which loads and stores using the same memory access type occur in the instruction stream; this is known as program order.

A parallel program has multiple instruction streams that can execute simultaneously on different processors. In multiprocessor (MP) systems, the order in which the effects of loads and stores are observed by other processors—the global order of the loads and store—determines the actions necessary to reliably share data in parallel programs.

When all processors observe the effects of loads and stores in program order, the system is strongly ordered. On such systems, parallel programs can reliably share data without explicit actions in the programs. For such a system, SYNC has the same effect as a NOP. Executing SYNC on such a system is not necessary, but neither is it an error.

If a multiprocessor system is not strongly ordered, the effects of load and store instructions executed by one processor may be observed out of program order by other processors. On such systems, parallel programs must take explicit actions to reliably share data. At critical points in the program, the effects of loads and stores from an instruction stream must occur in the same order for all processors. SYNC separates the loads and stores executed on the processor into two groups, and the effect of all loads and stores in one group is seen by all processors before the effect of any load or store in the subsequent group. In effect, SYNC causes the system to be strongly ordered for the executing processor at the instant that the SYNC is executed.

Many MIPS-based multiprocessor systems are strongly ordered or have a mode in which they operate as strongly ordered for at least one memory access type. The MIPS architecture also permits implementation of MP systems that are not strongly ordered; SYNC enables the reliable use of shared memory on such systems. A parallel program that does not use SYNC generally does not operate on a system that is not strongly ordered. However, a program that does use SYNC works on both types of systems. (System-specific documentation describes the actions needed to reliably share data in parallel programs for that system.)

The behavior of a load or store using one memory access type is UNPREDICTABLE if a load or store was previ-
ously made to the same physical location using a different memory access type. The presence of a SYNC between the references does not alter this behavior.

SYNC affects the order in which the effects of load and store instructions appear to all processors; it does not generally affect the physical memory-system ordering or synchronization issues that arise in system programming. The effect of SYNC on implementation-specific aspects of the cached memory system, such as writeback buffers, is not defined.

```
# Processor A (writer)
# Conditions at entry:
# The value 0 has been stored in FLAG and that value is observable by B
SW R1, DATA       # change shared DATA value
LI R2, 1
SYNC             # Perform DATA store before performing FLAG store
SW R2, FLAG      # say that the shared DATA value is valid

# Processor B (reader)
LI R2, 1
1: LW R1, FLAG       # Get FLAG
BNE R2, R1, 1B# if it says that DATA is not valid, poll again
NOP
SYNC             # FLAG value checked before doing DATA read
LW R1, DATA      # Read (valid) shared DATA value
```

The code fragments above show how SYNC can be used to coordinate the use of shared data between separate writer and reader instruction streams in a multiprocessor environment. The FLAG location is used by the instruction streams to determine whether the shared data item DATA is valid. The SYNC executed by processor A forces the store of DATA to be performed globally before the store to FLAG is performed. The SYNC executed by processor B ensures that DATA is not read until after the FLAG value indicates that the shared data is valid.

Software written to use a SYNC instruction with a non-zero stype value, expecting one type of barrier behavior, should only be run on hardware that actually implements the expected barrier behavior for that non-zero stype value or on hardware which implements a superset of the behavior expected by the software for that stype value. If the hardware does not perform the barrier behavior expected by the software, the system may fail.
**Synchronize Caches to Make Instruction Writes Effective**

**SYNCI**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGIMM</td>
<td>000001</td>
<td>base</td>
<td>SYNCI</td>
<td>11111</td>
<td>offset</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  
**SYNCI offset(base)**

**MIPS32 Release 2**

**Purpose:** Synchronize Caches to Make Instruction Writes Effective

To synchronize all caches to make instruction writes effective.

**Description:**

This instruction is used after a new instruction stream is written to make the new instructions effective relative to an instruction fetch, when used in conjunction with the SYNC and JALR.HB, JR.HB, or ERET instructions, as described below. Unlike the CACHE instruction, the SYNCI instruction is available in all operating modes in an implementation of Release 2 of the architecture.

The 16-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used to address the cache line in all caches which may need to be synchronized with the write of the new instructions. The operation occurs only on the cache line which may contain the effective address. One SYNCI instruction is required for every cache line that was written. See the Programming Notes below.

A TLB Refill and TLB Invalid (both with cause code equal TLBL) exception can occur as a byproduct of this instruction. This instruction never causes TLB Modified exceptions nor TLB Refill exceptions with a cause code of TLBS. This instruction never causes Execute-Inhibit nor Read-Inhibit exceptions.

A Cache Error exception may occur as a byproduct of this instruction. For example, if a writeback operation detects a cache or bus error during the processing of the operation, that error is reported via a Cache Error exception. Similarly, a Bus Error Exception may occur if a bus operation invoked by this instruction is terminated in an error.

An Address Error Exception (with cause code equal AdEL) may occur if the effective address references a portion of the kernel address space which would normally result in such an exception. It is implementation dependent whether such an exception does occur.

It is implementation dependent whether a data watch is triggered by a SYNCI instruction whose address matches the Watch register address match conditions. In multiprocessor implementations where instruction caches are not coherently maintained by hardware, the SYNCI instruction may optionally affect all coherent icaches within the system. If the effective address uses a coherent Cacheability and Coherency Attribute (CCA), then the operation may be globalized, meaning it is broadcast to all of the coherent instruction caches within the system. If the effective address does not use one of the coherent CCAs, there is no broadcast of the SYNCI operation. If multiple levels of caches are to be affected by one SYNCI instruction, all of the affected cache levels must be processed in the same manner - either all affected cache levels use the globalized behavior or all affected cache levels use the non-globalized behavior.

In multiprocessor implementations where instruction caches are coherently maintained by hardware, the SYNCI instruction should behave as a NOP instruction.

**Restrictions:**

The operation of the processor is **UNPREDICTABLE** if the effective address references any instruction cache line that contains instructions to be executed between the SYNCI and the subsequent JALR.HB, JR.HB, or ERET instruction required to clear the instruction hazard.

The SYNCI instruction has no effect on cache lines that were previously locked with the CACHE instruction. If correct software operation depends on the state of a locked line, the CACHE instruction must be used to synchronize the caches.

The SYNCI instruction acts on the current processor at a minimum. It is implementation specific whether it affects...
the caches on other processors in a multiprocessor system, except as required to perform the operation on the current processor (as might be the case if multiple processors share an L2 or L3 cache).

Full visibility of the new instruction stream requires execution of a subsequent SYNC instruction, followed by a JALR.HB, JR.HB, DERET, or ERET instruction. The operation of the processor is **UNPREDICTABLE** if this sequence is not followed.

**Operation:**

\[
\text{vaddr} \leftarrow \text{GPR[base]} + \text{sign_extend(offset)} \\
\text{SynchronizeCacheLines}(\text{vaddr}) \quad /* \text{Operate on all caches} */
\]

**Exceptions:**

Reserved Instruction Exception (Release 1 implementations only)
TLB Refill Exception
TLB Invalid Exception
Address Error Exception
Cache Error Exception
Bus Error Exception

**Programming Notes:**

When the instruction stream is written, the SYNCI instruction should be used in conjunction with other instructions to make the newly-written instructions effective. The following example shows a routine which can be called after the new instruction stream is written to make those changes effective. Note that the SYNCI instruction could be replaced with the corresponding sequence of CACHE instructions (when access to Coprocessor 0 is available), and that the JR.HB instruction could be replaced with JALR.HB, ERET, or DERET instructions, as appropriate. A SYNC instruction is required between the final SYNCI instruction in the loop and the instruction that clears instruction hazards.

```assembly
/*
 * This routine makes changes to the instruction stream effective to the
 * hardware. It should be called after the instruction stream is written.
 * On return, the new instructions are effective.
 *
 * Inputs:
 * a0 = Start address of new instruction stream
 * a1 = Size, in bytes, of new instruction stream
 */
beq a1, zero, 20f /* If size==0, */
nop /* branch around */
addu a1, a0, a1 /* Calculate end address + 1 */
rdhwr v0, HW_SYNCI_Step /* Get step size for SYNCI from new */
\quad /* Release 2 instruction */
beq v0, zero, 20f /* If no caches require synchronization, */
nop /* branch around */
10: synci 0(a0) /* Synchronize all caches around address */
addu a0, a0, v0 /* Add step size in delay slot */
sltue v1, a0, al /* Compare current with end address */
bne v1, zero, 10b /* Branch if more to do */
nop /* branch around */
sync /* Clear memory hazards */
20: jr.hb ra /* Return, clearing instruction hazards */
nop
```
### System Call

**Format:** SYSCALL

**Purpose:** System Call

To cause a System Call exception

**Description:**

A system call exception occurs, immediately and unconditionally transferring control to the exception handler.

The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

**Restrictions:**

None

**Operation:**

`SignalException(SystemCall)`

**Exceptions:**

System Call

---

![System Call Format](Image)

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>Bit 25</th>
<th>Bit 26</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL 00000</td>
<td></td>
<td></td>
<td>code</td>
<td></td>
<td>SYSCALL 001100</td>
</tr>
</tbody>
</table>

MIPS32
**Trap if Equal**

**Format:**  
`TEQ rs, rt`  

**Purpose:** Trap if Equal  
To compare GPRs and do a conditional trap  

**Description:**  
if GPR[rs] = GPR[rt] then Trap  
Compare the contents of GPR `rs` and GPR `rt` as signed integers; if GPR `rs` is equal to GPR `rt`, then take a Trap exception.  
The contents of the `code` field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**  
None

**Operation:**  
```c
if GPR[rs] = GPR[rt] then
    SignalException(Trap)
endif
```

**Exceptions:**  
Trap
**Trap if Equal Immediate**

**TEQI**

**Format:**  
TEQI rs, immediate

**Purpose:** Trap if Equal Immediate  
To compare a GPR to a constant and do a conditional trap

**Description:**  
if GPR[rs] = immediate then Trap

Compare the contents of GPR rs and the 16-bit signed immediate as signed integers; if GPR rs is equal to immediate, then take a Trap exception.

**Restrictions:**  
None

**Operation:**  
if GPR[rs] = sign_extend(immediate) then  
    SignalException(Trap)
endif

**Exceptions:**  
Trap
### Trap if Greater or Equal

**Format:** `TGE rs, rt`  
**Purpose:** Trap if Greater or Equal  
To compare GPRs and do a conditional trap  

**Description:**  
if GPR[rs] ≥ GPR[rt] then Trap  

Compare the contents of GPR `rs` and GPR `rt` as signed integers; if GPR `rs` is greater than or equal to GPR `rt`, then take a Trap exception.  

The contents of the `code` field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.  

**Restrictions:**  
None  

**Operation:**  
```
if GPR[rs] ≥ GPR[rt] then
    SignalException(Trap)
endif
```

**Exceptions:**  
Traps
# Trap if Greater or Equal Immediate

**Format:** \( \text{TGEI}\, rs,\, \text{immediate} \)

**Purpose:** Trap if Greater or Equal Immediate

To compare a GPR to a constant and do a conditional trap

**Description:** \( \text{if GPR}[rs] \geq \text{immediate then Trap} \)

Compare the contents of GPR \( rs \) and the 16-bit signed \( \text{immediate} \) as signed integers; if GPR \( rs \) is greater than or equal to \( \text{immediate} \), then take a Trap exception.

**Restrictions:**
None

**Operation:**

\[
\text{if GPR}[rs] \geq \text{sign}\_\text{extend}(\text{immediate}) \text{ then}
\]

\[
\text{SignalException(Trap)}
\]

**Exceptions:**

Trap
# Trap if Greater or Equal Immediate Unsigned

**Format:** \( \text{TGEIU} \) rs, \( \text{immediate} \)

**Purpose:** Trap if Greater or Equal Immediate Unsigned

To compare a GPR to a constant and do a conditional trap

**Description:** if GPR[\( \text{rs} \)] \( \geq \) immediate then Trap

Compare the contents of GPR \( \text{rs} \) and the 16-bit sign-extended \( \text{immediate} \) as unsigned integers; if GPR \( \text{rs} \) is greater than or equal to \( \text{immediate} \), then take a Trap exception.

Because the 16-bit \( \text{immediate} \) is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum \([0, 32767]\) or maximum \([\text{max}_{\text{unsigned}}-32767, \text{max}_{\text{unsigned}}]\) end of the unsigned range.

**Restrictions:**

None

**Operation:**

\[
\text{if} (0 || \text{GPR[rs]} \geq (0 || \text{sign}_\text{extend}(\text{immediate}))) \text{ then} \\
\quad \text{SignalException(Trap)} \\
\text{endif}
\]

**Exceptions:**

Trap

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>TGEIU</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td>5</td>
<td>01001</td>
<td>16</td>
</tr>
</tbody>
</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>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Trap if Greater or Equal Unsigned

Format:  TGEU rs, rt

Purpose: Trap if Greater or Equal Unsigned
To compare GPRs and do a conditional trap

Description: if GPR[rs] ≥ GPR[rt] then Trap
Compare the contents of GPR rs and GPR rt as unsigned integers; if GPR rs is greater than or equal to GPR rt, then take a Trap exception.
The contents of the code field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:
None

Operation:
if (0 || GPR[rs]) ≥ (0 || GPR[rt]) then
   SignalException(Trap)
endif

Exceptions:
Trap
Trap if Greater or Equal Unsigned

TGEU
Format: TLBINV

Purpose: TLB Invalidate

TLBINV invalidates a set of TLB entries based on ASID and Index match. The virtual address is ignored in the entry match. TLB entries which have their G bit set to 1 are not modified.

Implementation of the TLBINV instruction is optional. The implementation of this instruction is indicated by the IE field in Config4.

Implementation of EntryHiEHINV field is required for implementation of TLBGINV instruction.

Support for TLBINV is recommend for implementations supporting VTLB/FTLB type of MMU.

Description:

On execution of the TLBINV instruction, the set of TLB entries with matching ASID are marked invalid, excluding those TLB entries which have their G bit set to 1.

The EntryHiASID field has to be set to the appropriate ASID value before executing the TLBINV instruction.

Behavior of the TLBINV instruction applies to all applicable TLB entries and is unaffected by the setting of the Wired register.

For JTLB-based MMU (ConfigMT=1):

All matching entries in the JTLB are invalidated. Index is unused.

For VTLB/FTLB-based MMU (ConfigMT=4):

A TLBINV with Index set in VTLB range causes all matching entries in the VTLB to be invalidated.

A TLBINV with Index set in FTLB range causes all matching entries in the single corresponding FTLB set to be invalidated.

If TLB invalidate walk is implemented in software (Config4IE=2), then software must do these steps to flush the entire MMU:

1. one TLBINV instruction is executed with an index in VTLB range (invalidates all matching VTLB entries)
2. a TLBINV instruction is executed for each FTLB set (invalidates all matching entries in FTLB set)

If TLB invalidate walk is implemented in hardware (Config4IE=3), then software must do these steps to flush the entire MMU:

1. one TLBINV instruction is executed (invalidates all matching entries in both FTLB & VTLB). In this case, Index is unused.

Restrictions:
The operation is **UNDEFINED** if the contents of the *Index* register are greater than or equal to the number of available TLB entries (For the case of *Config\(_{MT}\)*=4).

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

**Operation:**

```c
if (Config\(_{MT}\)=1 or (Config\(_{MT}\)=4 & Config4\(_{IE}\)=2 & Index ≤ Config1\(_{MMU\_SIZE}\)-1))
    startnum ← 0
    endnum ← Config1\(_{MMU\_SIZE}\)-1
endif

// treating VTLB and FTLB as one array
if (Config\(_{MT}\)=4 & Config4\(_{IE}\)=2 & Index > Config1\(_{MMU\_SIZE}\)-1)
    startnum ← start of selected FTLB set // implementation specific
    endnum ← end of selected FTLB set - 1 // implementation specific
endif

if (Config\(_{MT}\)=4 & Config4\(_{IE}\)=3))
    startnum ← 0
    endnum ← Config1\(_{MMU\_SIZE}\)-1 + ((Config4\(_{FTLB\_Ways}\) + 2) * Config4\(_{FTLB\_Sets}\))
endif

for (i = startnum to endnum)
    if (TLB[i].ASID = EntryHi\_ASID & TLB[i].G = 0)
        TLB[i].VP\_V2_invalid ← 1
    endif
endfor
```

**Exceptions:**

Coprocessor Unusable
TLB Invalidate Flush

**Purpose:** TLB Invalidate Flush

TLBINVF invalidates a set of TLB entries based on `Index` match. The virtual address and ASID are ignored in the entry match.

Implementation of the TLBINVF instruction is optional. The implementation of this instruction is indicated by the IE field in `Config4`

Implementation of the `EntryHIEHINV` field is required for implementation of TLBINV and TLBINVF instructions.

Support for TLBINVF is recommend for implementations supporting VTLB/FTLB type of MMU.

**Description:**

On execution of the TLBINVF instruction, all entries within range of `Index` are invalidated.

Behavior of the TLBINVF instruction applies to all applicable TLB entries and is unaffected by the setting of the `Wired` register.

For JTLB-based MMU (`ConfigMT=1`):

TLBINVF causes all entries in the JTLB to be invalidated. `Index` is unused.

For VTLB/FTLB-based MMU (`ConfigMT=4`):

TLBINVF with `Index` in VTLB range causes all entries in the VTLB to be invalidated.

TLBINVF with `Index` in FTLB range causes all entries in the single corresponding set in the FTLB to be invalidated.

If TLB invalidate walk is implemented in software (`Config4IE=2`), then software must do these steps to flush the entire MMU:

1. one TLBINV instruction is executed with an index in VTLB range (invalidates all VTLB entries)
2. a TLBINV instruction is executed for each FTLB set (invalidates all entries in FTLB set)

If TLB invalidate walk is implemented in hardware (`Config4IE=3`), then software must do these steps to flush the entire MMU:

1. one TLBINV instruction is executed (invalidates all entries in both FTLB & VTLB). In this case, `Index` is unused.

**Restrictions:**

The operation is **UNDEFINED** if the contents of the `Index` register are greater than or equal to the number of available TLB entries (`Config4IE=2`).

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

### MIPS32

<table>
<thead>
<tr>
<th>COP0</th>
<th>CO</th>
<th>0</th>
<th>TLBINVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>1</td>
<td>000 0000 0000 0000 0000</td>
<td>000100</td>
</tr>
</tbody>
</table>

**Format:** TLBINVF
Operation:

if (ConfigMT=1 or (ConfigMT=4 & Config4IE=2 & Index ≤ Config1MMU_SIZE-1))
    startnum ← 0
    endnum ← Config1MMU_SIZE-1
endif

// treating VTLB and FTLB as one array
if (ConfigMT=4 & Config4IE=2 & Index > Config1MMU_SIZE-1)
    startnum ← start of selected FTLB set // implementation specific
    endnum ← end of selected FTLB set - 1 // implementation specific
endif

if (ConfigMT=4 & Config4IE=3))
    startnum ← 0
    endnum ← Config1MMU_SIZE-1 + ((Config4FTLBWays + 2) * Config4FTLBSets)
endif

for (i = startnum to endnum)
    TLB[i].VPN2_invalid ← 1
endfor

Exceptions:

Coprocessor Unusable
Format: TLBP

Purpose: Probe TLB for Matching Entry

To find a matching entry in the TLB.

Description:
The Index register is loaded with the address of the TLB entry whose contents match the contents of the EntryHi register. If no TLB entry matches, the high-order bit of the Index register is set. In Release 1 of the Architecture, it is implementation dependent whether multiple TLB matches are detected on a TLBP. However, implementations are strongly encouraged to report multiple TLB matches only on a TLB write. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. In Release 3 of the Architecture, multiple TLB matches may be reported on either TLB write or TLB probe.

Restrictions:
If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:
\[
\text{Index} \leftarrow 1 \mid \text{UNPREDICTABLE}^{31} \\
\text{for } i \in 0 \ldots \text{TLBEntries}-1 \\
\quad \text{if } \{(\text{TLB}[i]_{\text{VPN2}} \text{ and not } (\text{TLB}[i]_{\text{Mask}})) = \\
\quad \quad \quad \{(\text{EntryHi}_{\text{VPN2}} \text{ and not } (\text{TLB}[i]_{\text{Mask}})) \text{ and } \\
\quad \quad \quad \quad \quad ((\text{TLB}[i]_{\text{G}} = 1) \text{ or } (\text{TLB}[i]_{\text{ASID}} = \text{EntryHi}_{\text{ASID}}))\text{then} \\
\quad \quad \quad \text{Index} \leftarrow i \\
\quad \quad \text{endif} \\
\text{endfor}
\]

Exceptions:
Coprocessor Unusable

Machine Check
Purpose: Read Indexed TLB Entry

To read an entry from the TLB.

Description:
The EntryHi, EntryLo0, EntryLo1, and PageMask registers are loaded with the contents of the TLB entry pointed to by the Index register. In Release 1 of the Architecture, it is implementation dependent whether multiple TLB matches are detected on a TLBR. However, implementations are strongly encouraged to report multiple TLB matches only on a TLB write. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. In Release 3 of the Architecture, multiple TLB matches may be detected on a TLBR.

In an implementation supporting TLB entry invalidation (Config4IE = 2 or Config4IE = 3), reading an invalid TLB entry causes 0 to be written to EntryHi, EntryLo0, EntryLo1 registers and the PageMaskMASK register field.

Note that the value written to the EntryHi, EntryLo0, and EntryLo1 registers may be different from that originally written to the TLB via these registers in that:

- The value returned in the VPN2 field of the EntryHi register may have those bits set to zero corresponding to the one bits in the Mask field of the TLB entry (the least-significant bit of VPN2 corresponds to the least-significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed after a TLB entry is written and then read.

- The value returned in the PFN field of the EntryLo0 and EntryLo1 registers may have those bits set to zero corresponding to the one bits in the Mask field of the TLB entry (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed after a TLB entry is written and then read.

- The value returned in the G bit in both the EntryLo0 and EntryLo1 registers comes from the single G bit in the TLB entry. Recall that this bit was set from the logical AND of the two G bits in EntryLo0 and EntryLo1 when the TLB was written.

Restrictions:
The operation is UNDEFINED if the contents of the Index register are greater than or equal to the number of TLB entries in the processor.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```c
i ← Index
if i > (TLBEntries - 1) then
    UNDEFINED
endif
if ( (Config4IE = 2 or Config4IE = 3) and TLB[i]VPN2_invalid = 1) then
    PageMask_MASK ← 0
    EntryHi ← 0
    EntryLo0 ← 0
```

TLBR

<table>
<thead>
<tr>
<th></th>
<th>COP0</th>
<th>CO</th>
<th>0</th>
<th>TLBR</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>010000</td>
<td>1</td>
<td>000 0000 0000 0000 0000</td>
<td>000001</td>
</tr>
<tr>
<td>26</td>
<td>25</td>
<td>24</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

COP0 010000

CO 1

000 0000 0000 0000 0000

TLBR 000001

MIPS32
EntryLo0 ← 0
EntryHiInv ← 1
else
    PageMaskMask ← TLB[i]_Mask
    EntryHi ←
        (TLB[i]_VPN2 and not TLB[i]_Mask) || # Masking implem dependent
        0^5 || TLB[i]_ASID
    EntryLo1 ← 0^2 ||
        (TLB[i]_PFN1 and not TLB[i]_Mask) || # Masking implem dependent
        TLB[i]_Cl || TLB[i]_D1 || TLB[i]_V1 || TLB[i]_G
    EntryLo0 ← 0^2 ||
        (TLB[i]_PFN0 and not TLB[i]_Mask) || # Masking implem dependent
        TLB[i]_C0 || TLB[i]_D0 || TLB[i]_V0 || TLB[i]_G
endif

Exceptions:
Coprocessor Unusable
Machine Check
Write Indexed TLB Entry

**TLBWI**

<table>
<thead>
<tr>
<th>COP0</th>
<th>CO</th>
<th>0</th>
<th>TLBWI</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>1</td>
<td>000 0000 0000 0000 0000</td>
<td>000010</td>
</tr>
</tbody>
</table>

**Format:** TLBWI

**MIPS32**

**Purpose:** Write Indexed TLB Entry

To write or invalidate a TLB entry indexed by the Index register.

**Description:**

If $Config4IE < 2$ or $EntryHi_{EHINV}=0$:

The TLB entry pointed to by the Index register is written from the contents of the $EntryHi$, $EntryLo0$, $EntryLo1$, and $PageMask$ registers. It is implementation dependent whether multiple TLB matches are detected on a TLBWI. In such an instance, a Machine Check Exception is signaled. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. The information written to the TLB entry may be different from that in the $EntryHi$, $EntryLo0$, and $EntryLo1$ registers, in that:

- The value written to the VPN2 field of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of the $PageMask$ register (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.

- The value written to the PFN0 and PFN1 fields of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of $PageMask$ register (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.

- The single G bit in the TLB entry is set from the logical AND of the G bits in the $EntryLo0$ and $EntryLo1$ registers.

If $Config4IE > 1$ and $EntryHi_{EHINV}=1$:

The TLB entry pointed to by the Index register has its VPN2 field marked as invalid. This causes the entry to be ignored on TLB matches for memory accesses. No Machine Check is generated.

**Restrictions:**

The operation is **UNDEFINED** if the contents of the Index register are greater than or equal to the number of TLB entries in the processor.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

**Operation:**

```plaintext
i ← Index
if (Config4IE = 2 or Config4IE = 3) then
    TLB[i]VPN2_invalid ← 0
    if (EntryHI_{EHINV}=1) then
        TLB[i]VPN2_invalid ← 1
        break
    endif
endif
```
Write Indexed TLB Entry

TLB[i]Mask ← PageMaskMask
TLB[i]VPN2 ← EntryHiVPN2 and not PageMaskMask # Implementation dependent
TLB[i]ASID ← EntryHiASID
TLB[i]G ← EntryLo1G and EntryLo0G
TLB[i]PFN1 ← EntryLo1PFN and not PageMaskMask # Implementation dependent
TLB[i]C1 ← EntryLo1C
TLB[i]D1 ← EntryLo1D
TLB[i]V1 ← EntryLo1V
TLB[i]PFN0 ← EntryLo0PFN and not PageMaskMask # Implementation dependent
TLB[i]C0 ← EntryLo0C
TLB[i]D0 ← EntryLo0D
TLB[i]V0 ← EntryLo0V

Exceptions:

Coprocessor Unusable

Machine Check
Write Random TLB Entry

Purpose: Write Random TLB Entry

To write a TLB entry indexed by the Random register.

Description:

The TLB entry pointed to by the Random register is written from the contents of the EntryHi, EntryLo0, EntryLo1, and PageMask registers. It is implementation dependent whether multiple TLB matches are detected on a TLBWR. In such an instance, a Machine Check Exception is signaled. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. The information written to the TLB entry may be different from that in the EntryHi, EntryLo0, and EntryLo1 registers, in that:

- The value written to the VPN2 field of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of the PageMask register (the least significant bit of VPN2 corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.

- The value written to the PFN0 and PFN1 fields of the TLB entry may have those bits set to zero corresponding to the one bits in the Mask field of PageMask register (the least significant bit of PFN corresponds to the least significant bit of the Mask field). It is implementation dependent whether these bits are preserved or zeroed during a TLB write.

- The single G bit in the TLB entry is set from the logical AND of the G bits in the EntryLo0 and EntryLo1 registers.

Restrictions:

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

\[
i \leftarrow \text{Random}
\]

if (Config4IE = 2 or Config4IE = 3) then
   \[\text{TLB}[i]_{\text{VPN2}} \_\text{valid} \leftarrow 0\]
endif
\[\text{TLB}[i]_{\text{Mask}} \leftarrow \text{PageMask}_{\text{Mask}}\]
\[\text{TLB}[i]_{\text{VPN2}} \leftarrow \text{EntryHi}_{\text{VPN2}} \text{ and not } \text{PageMask}_{\text{Mask}} \text{ # Implementation dependent}\]
\[\text{TLB}[i]_{\text{ASID}} \leftarrow \text{EntryHi}_{\text{ASID}}\]
\[\text{TLB}[i]_{G} \leftarrow \text{EntryLo1}_{G} \text{ and EntryLo0}_{G}\]
\[\text{TLB}[i]_{\text{PFN1}} \leftarrow \text{EntryLo1}_{\text{PFN}} \text{ and not } \text{PageMask}_{\text{Mask}} \text{ # Implementation dependent}\]
\[\text{TLB}[i]_{C1} \leftarrow \text{EntryLo1}_{C}\]
\[\text{TLB}[i]_{D1} \leftarrow \text{EntryLo1}_{D}\]
\[\text{TLB}[i]_{V1} \leftarrow \text{EntryLo1}_{V}\]
\[\text{TLB}[i]_{\text{PFN0}} \leftarrow \text{EntryLo0}_{\text{PFN}} \text{ and not } \text{PageMask}_{\text{Mask}} \text{ # Implementation dependent}\]
\[\text{TLB}[i]_{C0} \leftarrow \text{EntryLo0}_{C}\]
\[\text{TLB}[i]_{D0} \leftarrow \text{EntryLo0}_{D}\]
\[\text{TLB}[i]_{V0} \leftarrow \text{EntryLo0}_{V}\]
Exceptions:
Coprocessor Unusable
Machine Check
### Trap if Less Than

**Format:** \[ \text{TLT } rs, rt \]  

**MIPS32**

**Purpose:** Trap if Less Than  
To compare GPRs and do a conditional trap

**Description:**  
if GPR[rs] < GPR[rt] then Trap  
Compare the contents of GPR \( rs \) and GPR \( rt \) as signed integers; if GPR \( rs \) is less than GPR \( rt \), then take a Trap exception.  
The contents of the \( code \) field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**  
None

**Operation:**  
if GPR[rs] < GPR[rt] then  
    SignalException(Trap)  
endif

**Exceptions:**  
Trap
Trap if Less Than Immediate

**Format:** TLTI rs, immediate

**Purpose:** Trap if Less Than Immediate
To compare a GPR to a constant and do a conditional trap

**Description:** if GPR[rs] < immediate then Trap
Compare the contents of GPR rs and the 16-bit signed immediate as signed integers; if GPR rs is less than immediate, then take a Trap exception.

**Restrictions:** None

**Operation:**

```c
if GPR[rs] < sign_extend(immediate) then
    SignalException(Trap)
endif
```

**Exceptions:** Trap
Format: TLTIU rs, immediate

Purpose: Trap if Less Than Immediate Unsigned
To compare a GPR to a constant and do a conditional trap

Description: if GPR[rs] < immediate then Trap
Compare the contents of GPR rs and the 16-bit sign-extended immediate as unsigned integers; if GPR rs is less than immediate, then take a Trap exception.

Because the 16-bit immediate is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

Restrictions:
None

Operation:
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
   SignalException(Trap)
endif

Exceptions:
Trap
**Trap if Less Than Unsigned**

**TLTU**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>rs</td>
<td>rt</td>
<td>code</td>
<td>TLTU</td>
<td>110011</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** TLTU rs, rt

**MIPS32**

**Purpose:** Trap if Less Than Unsigned

To compare GPRs and do a conditional trap

**Description:** if GPR[rs] < GPR[rt] then Trap

Compare the contents of GPR rs and GPR rt as unsigned integers; if GPR rs is less than GPR rt, then take a Trap exception.

The contents of the code field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**

None

**Operation:**

\[
\text{if (0 || GPR[rs]) < (0 || GPR[rt]) then}
\]

\[
\text{SignalException(Trap)}
\]

\[
\text{endif}
\]

**Exceptions:**

Trap
Trap if Not Equal

**Format:** \texttt{TNE \text{rs}, \text{rt}}

**Purpose:** Trap if Not Equal

To compare GPRs and do a conditional trap

**Description:** if \texttt{GPR[rs] \neq GPR[rt]} then Trap

Compare the contents of GPR \texttt{rs} and GPR \texttt{rt} as signed integers; if GPR \texttt{rs} is not equal to GPR \texttt{rt}, then take a Trap exception.

The contents of the \texttt{code} field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**

None

**Operation:**

\begin{verbatim}
    if \texttt{GPR[rs] \neq GPR[rt]} then
        SignalException(Trap)
    endif
\end{verbatim}

**Exceptions:**

Trap
Trap if Not Equal Immediate

**Format:**  TNEI rs, immediate

**Purpose:** Trap if Not Equal Immediate
To compare a GPR to a constant and do a conditional trap

**Description:** if GPR[rs] ≠ immediate then Trap
Compare the contents of GPR rs and the 16-bit signed immediate as signed integers; if GPR rs is not equal to immediate, then take a Trap exception.

**Restrictions:**
None

**Operation:**

```plaintext
if GPR[rs] ≠ sign_extend(immediate) then
    SignalException(Trap)
endif
```

**Exceptions:**
Trap
Floating Point Truncate to Long Fixed Point

**Format:**
- TRUNC.L.fmt
- TRUNC.L.S fd, fs
- TRUNC.L.D fd, fs

**MIPS64, MIPS32 Release 2**

**Purpose:**
Floating Point Truncate to Long Fixed Point

To convert an FP value to 64-bit fixed point, rounding toward zero

**Description:**
FPR[fd] ← convert_and_round(FPR[fs])

The value in FPR fs, in format fmt, is converted to a value in 64-bit long fixed point format and rounded toward zero (rounding mode 1). The result is placed in FPR fd.

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{63}\) to \(2^{63}-1\), the result cannot be represented correctly and an IEEE Invalid Operation condition exists. In this case the Invalid Operation flag is set in the FCSR. If the Invalid Operation Enable bit is set in the FCSR, no result is written to fd and an Invalid Operation exception is taken immediately. Otherwise, the default result, \(2^{63}-1\), is written to fd.

**Restrictions:**
The fields fs and fd must specify valid FPRs; fs for type fmt and fd for long fixed point; if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

The result of this instruction is UNPREDICTABLE if the processor is executing in the FR=0 32-bit FPU register model; it is predictable if executing on a 64-bit FPU in the FR=1 mode, but not with FR=0, and not on a 32-bit FPU.

**Operation:**

\[
\text{StoreFPR}(fd, L, \text{ConvertFmt(ValueFPR(fs, fmt), fmt, L)})
\]

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Unimplemented Operation, Invalid Operation, Inexact
Floating Point Truncate to Word Fixed Point

**Format:**

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>TRUNC.W</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td></td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>01101</td>
</tr>
</tbody>
</table>

**Purpose:** Floating Point Truncate to Word Fixed Point

To convert an FP value to 32-bit fixed point, rounding toward zero

**Description:**

\[
\text{FPR}[fd] \leftarrow \text{convert\_and\_round}(\text{FPR}[fs])
\]

The value in FPR \( fs \), in format \( fmt \), is converted to a value in 32-bit word fixed point format using rounding toward zero (rounding mode 1). The result is placed in FPR \( fd \).

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{31} \) to \( 2^{31} - 1 \), the result cannot be represented correctly and an IEEE Invalid Operation condition exists. In this case the Invalid Operation flag is set in the FCSR. If the Invalid Operation Enable bit is set in the FCSR, no result is written to \( fd \) and an Invalid Operation exception is taken immediately. Otherwise, the default result, \( 2^{31} - 1 \), is written to \( fd \).

**Restrictions:**

The fields \( fs \) and \( fd \) must specify valid FPRs; \( fs \) for type \( fmt \) and \( fd \) for word fixed point; if they are not valid, the result is UNPREDICTABLE.

The operand must be a value in format \( fmt \); if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

**Operation:**

\[
\text{StoreFPR}(fd, W, \text{ConvertFmt}(\text{ValueFPR}(fs, fmt), fmt, W))
\]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Inexact, Invalid Operation, Unimplemented Operation
Enter Standby Mode

WAIT

Format: \texttt{WAIT} \hfill MIPS32

Purpose: Enter Standby Mode

Wait for Event

Description:
The \texttt{WAIT} instruction performs an implementation-dependent operation, usually involving a lower power mode. Software may use the code bits of the instruction to communicate additional information to the processor, and the processor may use this information as control for the lower power mode. A value of zero for code bits is the default and must be valid in all implementations.

The \texttt{WAIT} instruction is typically implemented by stalling the pipeline at the completion of the instruction and entering a lower power mode. The pipeline is restarted when an external event, such as an interrupt or external request occurs, and execution continues with the instruction following the \texttt{WAIT} instruction. It is implementation-dependent whether the pipeline restarts when a non-enabled interrupt is requested. In this case, software must poll for the cause of the restart. The assertion of any reset or NMI must restart the pipeline and the corresponding exception must be taken.

If the pipeline restarts as the result of an enabled interrupt, that interrupt is taken between the \texttt{WAIT} instruction and the following instruction (EPC for the interrupt points at the instruction following the \texttt{WAIT} instruction).

Restrictions:
The operation of the processor is \texttt{UNDEFINED} if a \texttt{WAIT} instruction is placed in the delay slot of a branch or a jump.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

\begin{itemize}
\item \texttt{I: Enter implementation dependent lower power mode}
\item \texttt{I+1: } /* Potential interrupt taken here */
\end{itemize}

Exceptions:

Coprocessor Unusable Exception
Write to GPR in Previous Shadow Set

**Format:** \( \text{WRPGPR} \ rd, \ rt \)

**Purpose:** Write to GPR in Previous Shadow Set
To move the contents of a current GPR to a GPR in the previous shadow set.

**Description:** \( \text{SGPR}[\text{SRSCtlPSS}, \ rd] \leftarrow \text{GPR}[rt] \)
The contents of the current GPR \( rt \) is moved to the shadow GPR register specified by \( \text{SRSCtlPSS} \) (signifying the previous shadow set number) and \( rd \) (specifying the register number within that set).

**Restrictions:**
In implementations prior to Release 2 of the Architecture, this instruction resulted in a Reserved Instruction Exception.

**Operation:**
\[ \text{SGPR}[\text{SRSCtlPSS}, \ rd] \leftarrow \text{GPR}[rt] \]

**Exceptions:**
Coprocessor Unusable
Reserved Instruction
Word Swap Bytes Within Halfwords  

**Format:**  
WSBH rd, rt

**Purpose:**  
Word Swap Bytes Within Halfwords  
To swap the bytes within each halfword of GPR rt and store the value into GPR rd.

**Description:**  
GPR[rd] ← SwapBytesWithinHalfwords(GPR[rt])  
Within each halfword of GPR rt the bytes are swapped, and stored in GPR rd.

**Restrictions:**  
In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

**Operation:**  
GPR[rd] ← GPR[r]23..16 || GPR[r]31..24 || GPR[r]7..0 || GPR[r]15..8

**Exceptions:**  
Reserved Instruction

**Programming Notes:**  
The WSBH instruction can be used to convert halfword and word data of one endianness to another endianness. The endianness of a word value can be converted using the following sequence:

```
lw t0, 0(al)     /* Read word value */
wsbh t0, t0      /* Convert endianness of the halfwords */
rotr t0, t0, 16  /* Swap the halfwords within the words */
```

Combined with SEH and SRA, two contiguous halfwords can be loaded from memory, have their endianness converted, and be sign-extended into two word values in four instructions. For example:

```
lw t0, 0(al)     /* Read two contiguous halfwords */
wsbh t0, t0      /* Convert endianness of the halfwords */
seh t1, t0       /* t1 = lower halfword sign-extended to word */
sra t0, t0, 16   /* t0 = upper halfword sign-extended to word */
```

Zero-extended words can be created by changing the SEH and SRA instructions to ANDI and SRL instructions, respectively.
**Format:** XOR rd, rs, rt

**Purpose:** Exclusive OR

To do a bitwise logical Exclusive OR

**Description:** GPR[rd] ← GPR[rs] XOR GPR[rt]

Combine the contents of GPR rs and GPR rt in a bitwise logical Exclusive OR operation and place the result into GPR rd.

**Restrictions:**
None

**Operation:**

GPR[rd] ← GPR[rs] xor GPR[rt]

**Exceptions:**
None
### Exclusive OR Immediate

**Format:** XORI rt, rs, immediate

**Purpose:** Exclusive OR Immediate

To do a bitwise logical Exclusive OR with a constant

**Description:**

\[
\text{GPR[rt]} \leftarrow \text{GPR[rs]} \ XOR \ \text{immediate}
\]

Combine the contents of GPR \( rs \) and the 16-bit zero-extended \( \text{immediate} \) in a bitwise logical Exclusive OR operation and place the result into GPR \( rt \).

**Restrictions:**

None

**Operation:**

\[
\text{GPR[rt]} \leftarrow \text{GPR[rs]} \ XOR \ \text{zero_extend(immediate)}
\]

**Exceptions:**

None
Instruction Bit Encodings

A.1 Instruction Encodings and Instruction Classes

Instruction encodings are presented in this section; field names are printed here and throughout the book in italics.

When encoding an instruction, the primary *opcode* field is encoded first. Most *opcode* values completely specify an instruction that has an *immediate* value or offset.

*Opcode* values that do not specify an instruction instead specify an instruction class. Instructions within a class are further specified by values in other fields. For instance, *opcode* REGIMM specifies the *immediate* instruction class, which includes conditional branch and trap *immediate* instructions.

A.2 Instruction Bit Encoding Tables

This section provides various bit encoding tables for the instructions of the MIPS32® ISA.

*Figure A.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 leftmost columns of the table. Bits 28..26 of the *opcode* field are listed along the topmost 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).
Tables A.2 through A.20 describe the encoding used for the MIPS32 ISA. Table A.1 describes the meaning of the symbols used in the tables.

### Table A.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 or a new revision of the Architecture. Executing such an instruction must cause a Reserved Instruction Exception.</td>
</tr>
<tr>
<td>∇</td>
<td>Operation or field codes marked with this symbol represent instructions which were only legal if 64-bit operations were enabled on implementations of Release 1 of the Architecture. In Release 2 of the architecture, operation or field codes marked with this symbol represent instructions which are legal if 64-bit floating point operations are enabled. In other cases, executing such an instruction must cause a Reserved Instruction Exception (non-coprocessor encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Usable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).</td>
</tr>
</tbody>
</table>
Table A.1 Symbols Used in the Instruction Encoding Tables (Continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Δ</td>
<td>Instructions formerly marked ∇ in some earlier versions of manuals, corrected and marked ∆ in revision 5.03. Legal on MIPS64r1 but not MIPS32r1; in release 2 and above, legal in both MIPS64 and MIPS32, in particular even when running in “32-bit FPU Register File mode”, FR=0, as well as FR=1.</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 encodings if requested by the partner. The partner is not required to consult with MIPS Technologies when one of these encodings is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction Exception (SPECIAL2 encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Usable Exception (coprocessor instruction encodings 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 optional Module or 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 MIPS32 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 A.2 MIPS32 Encoding of the Opcode Field

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>β</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 encodings if requested by the partner. The partner is not required to consult with MIPS Technologies when one of these encodings is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction Exception (SPECIAL2 encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Usable Exception (coprocessor instruction encodings 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 optional Module or 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 MIPS32 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>

1. In Release 1 of the Architecture, the COP1X opcode was called COP3, and was available as another user-available coprocessor. In Release 2 of the Architecture, a full 64-bit floating point unit is available with 32-bit CPUs, and the COP1X opcode is reserved for that purpose on all Release 2 CPUs. 32-bit implementations of Release 1 of the architecture are strongly discouraged from using this opcode for a user-available coprocessor as doing so will limit the potential for an upgrade path for the FPU.
2. Release 2 of the Architecture added the SPECIAL3 opcode. Implementations of Release 1 of the Architecture signaled a Reserved Instruction Exception for this opcode.
### Table A.3 MIPS32 SPECIAL Opcode Encoding of Function Field

<table>
<thead>
<tr>
<th>Function</th>
<th>Bits 2-0</th>
<th>Bits 5-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>

1. Specific encodings of the \( rt \), \( rd \), and \( sa \) fields are used to distinguish among the SLL, NOP, SSNOP, EHB and PAUSE functions.

2. Specific encodings of the \( hint \) field are used to distinguish JR from JR.HB and JALR from JALR.HB

### Table A.4 MIPS32 REGIMM Encoding of rt Field

<table>
<thead>
<tr>
<th>rt</th>
<th>Bits 18-16</th>
<th>Bits 20-19</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>110</td>
</tr>
</tbody>
</table>

### Table A.5 MIPS32 SPECIAL2 Encoding of Function Field

<table>
<thead>
<tr>
<th>Function</th>
<th>Bits 2-0</th>
<th>Bits 5-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>
### Table A.6 MIPS32 SPECIAL3 Encoding of Function Field for Release 2 of the Architecture

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 5..3</td>
<td>000 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td>0 000 EXT ⊕ β β β INS ⊕ β β β</td>
<td></td>
</tr>
<tr>
<td>1 001 ε ε ε * ε ε ε * ε ε ε</td>
<td></td>
</tr>
<tr>
<td>2 010 ε ε ε ε ε ε ε ε ε ε ε</td>
<td></td>
</tr>
<tr>
<td>3 011 ε LWLE LWRE CACHEE SBE SHE SCE SWE</td>
<td></td>
</tr>
<tr>
<td>4 100 BSHFL ⊕δ SWLE SWRE PREFE β * * *</td>
<td></td>
</tr>
<tr>
<td>5 101 LBUE LHUE * * * LBE LHE LLE LWE</td>
<td></td>
</tr>
<tr>
<td>6 110 ε ε * * ε ε * * ε ε *</td>
<td></td>
</tr>
<tr>
<td>7 111 ε * * * RDHWR ⊕ ε * * *</td>
<td></td>
</tr>
</tbody>
</table>

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.

### Table A.7 MIPS32 MOVCI Encoding of tf Bit

<table>
<thead>
<tr>
<th>tf bit 16</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVF</td>
<td>MOVT</td>
<td></td>
</tr>
</tbody>
</table>

### Table A.8 MIPS32 SRL Encoding of Shift/Rotate

<table>
<thead>
<tr>
<th>R bit 21</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRL</td>
<td>ROTR</td>
<td></td>
</tr>
</tbody>
</table>

1. Release 2 of the Architecture added the ROTR instruction. Implementations of Release 1 of the Architecture ignored bit 21 and treated the instruction as an SRL.

### Table A.9 MIPS32 SRLV Encoding of Shift/Rotate

<table>
<thead>
<tr>
<th>R bit 6</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRLV</td>
<td>ROTRV</td>
<td></td>
</tr>
</tbody>
</table>

1. Release 2 of the Architecture added the ROTRV instruction. Implementations of Release 1 of the Architecture ignored bit 6 and treated the instruction as an SRLV.
### Table A.10 MIPS32 BSHFL Encoding of sa Field

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

1. The sa field is sparsely 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 A.11 MIPS32 COP0 Encoding of rs Field

<table>
<thead>
<tr>
<th>rs bits 25..21</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>0</td>
<td>00</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>1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Release 2 of the Architecture added the MFMC0 function, which is further decoded as the DI (bit 5 = 0) and EI (bit 5 = 1) instructions.

### Table A.12 MIPS32 COP0 Encoding of Function Field When rs=CO

<table>
<thead>
<tr>
<th>function bits 5..3</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>0</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>1</td>
<td>100</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>110</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>000</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>011</td>
<td>010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Release 2 of the Architecture added the MFMC0 function, which is further decoded as the DI (bit 5 = 0) and EI (bit 5 = 1) instructions.
### Table A.13 MIPS32 COP1 Encoding of rs Field

<table>
<thead>
<tr>
<th>rs</th>
<th>bits 23..21</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000 MFC1 β</td>
</tr>
<tr>
<td>1</td>
<td>BC1 δ BC1ANY2 δεV</td>
</tr>
<tr>
<td>2</td>
<td>S δ D δ * * W δ L δ PS δ *</td>
</tr>
<tr>
<td>3</td>
<td>BZ.B ε BZ.H ε BZ.W ε BZ.D ε BNZ.B ε BNZ.H ε BNZ.W ε BNZ.D ε</td>
</tr>
</tbody>
</table>

### Table A.14 MIPS32 COP1 Encoding of Function Field When rs=S

<table>
<thead>
<tr>
<th>function</th>
<th>bits 5..3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000 ADD SUB MUL DIV SORT ABS MOV NEG</td>
<td></td>
</tr>
<tr>
<td>1 001 ROUND.L V TRUNC.L V CEILL.V FLOORL.V ROUND.W TRUNC.W CEIL.W FLOOR.W</td>
<td></td>
</tr>
<tr>
<td>2 010 MOVCF δ MOVZ MOVN + + + + RECIP1 εV RECIP1 εV RSQRT1 εV RSQRT2 εV</td>
<td></td>
</tr>
<tr>
<td>3 011 MOVCF δ MOVZ MOVN + + + + RECIP2 εV RECIP2 εV</td>
<td></td>
</tr>
<tr>
<td>4 100 CVT.D + + + + CVT.W CVTL V CVTPS V +</td>
<td></td>
</tr>
<tr>
<td>5 101 + + + + + + + +</td>
<td></td>
</tr>
<tr>
<td>6 110 CF CABS.F εV CABS.UN εV C.EQ CABS.EQ εV C.UEQ CABS.UEQ εV C.OLT CABS.OLT εV C.ULT CABS.ULT εV C.OLE CABS.OLE εV C.ULE CABS.ULE εV</td>
<td></td>
</tr>
<tr>
<td>7 111 CSF CABS.SF εV C.NGLE CABS.NGLE εV C.SQO CABS.SSEQ εV C.NGL CABS.NGL εV C.LT CABS.LT εV C.NGE CABS.NGE εV C.LE CABS.LE εV C.NGT CABS.NGT εV</td>
<td></td>
</tr>
</tbody>
</table>

### Table A.15 MIPS32 COP1 Encoding of Function Field When rs=D

<table>
<thead>
<tr>
<th>function</th>
<th>bits 5..3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000 ADD SUB MUL DIV SORT ABS MOV NEG</td>
<td></td>
</tr>
<tr>
<td>1 001 ROUND.L V TRUNC.L V CEILL.V FLOORL.V ROUND.W TRUNC.W CEIL.W FLOOR.W</td>
<td></td>
</tr>
<tr>
<td>2 010 MOVCF δ MOVZ MOVN + + + + RECIP1 εV RECIP1 εV RSQRT1 εV RSQRT2 εV</td>
<td></td>
</tr>
<tr>
<td>3 011 + + + + + + + +</td>
<td></td>
</tr>
<tr>
<td>4 100 CVT.S + + + + CVT.W CVTL V + +</td>
<td></td>
</tr>
<tr>
<td>5 101 + + + + + + + +</td>
<td></td>
</tr>
<tr>
<td>6 110 CF CABS.F εV C.UN CABS.UN εV C.EQ CABS.EQ εV C.UEQ CABS.UEQ εV C.OLT CABS.OLT εV C.ULT CABS.ULT εV C.OLE CABS.OLE εV C.ULE CABS.ULE εV</td>
<td></td>
</tr>
<tr>
<td>7 111 CSF CABS.SF εV C.NGLE CABS.NGLE εV C.SQO CABS.SSEQ εV C.NGL CABS.NGL εV C.LT CABS.LT εV C.NGE CABS.NGE εV C.LE CABS.LE εV C.NGT CABS.NGT εV</td>
<td></td>
</tr>
</tbody>
</table>
### Table A.16 MIPS32 COP1 Encoding of Function Field When rs=W or L

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
<th>bits 5..3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7</td>
<td>000 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>CVT.S</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>111</td>
</tr>
</tbody>
</table>

1. Format type L is legal only if 64-bit floating point operations are enabled.

### Table A.17 MIPS32 COP1 Encoding of Function Field When rs=PS

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
<th>bits 5..3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7</td>
<td>000 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td>0</td>
<td>ADD</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>SUB</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>MOVC</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>ADDR</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>CVT.S.PU</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>CVT.S.PL</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>C.F</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>C.SF</td>
<td>111</td>
</tr>
</tbody>
</table>

1. Format type PS is legal only if 64-bit floating point operations are enabled.

### Table A.18 MIPS32 COP1 Encoding of tf Bit When rs=S, D, or PS, Function=MOVCF

<table>
<thead>
<tr>
<th>tf</th>
<th>bit 16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>MOVF:fmt</td>
<td>MOVTF:fmt</td>
</tr>
</tbody>
</table>
### Table A.19 MIPS32 COP2 Encoding of rs Field

<table>
<thead>
<tr>
<th>bits 25..24</th>
<th>rs</th>
<th>bits 23..21</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>000</td>
<td>000</td>
<td>MFC2</td>
<td>θ</td>
<td>β</td>
<td>CFC2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td>BC2</td>
<td>θ</td>
<td>∗∗∗∗∗∗∗</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>C2</td>
<td>θδ</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>C2</td>
<td>θβ</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>SWXC1</td>
<td>∆</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>101</td>
<td>SWXC1</td>
<td>∆</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>110</td>
<td>NMADD.S</td>
<td>∆</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>111</td>
<td>NMSUB.S</td>
<td>∆</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table A.20 MIPS32 COP1X Encoding of Function Field

<table>
<thead>
<tr>
<th>bits 5..3</th>
<th>function</th>
<th>bits 2..0</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>000</td>
<td>LWXC1</td>
<td>∆</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>SWXC1</td>
<td>∆</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>SWXC1</td>
<td>∆</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>011</td>
<td>SWXC1</td>
<td>∆</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>SWXC1</td>
<td>∆</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>SWXC1</td>
<td>∆</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>SWXC1</td>
<td>∆</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>SWXC1</td>
<td>∆</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### A.3 Floating Point Unit Instruction Format Encodings

Instruction format encodings for the floating point unit are presented in this section. This information is a tabular presentation of the encodings described in tables Table A.13 and Table A.20 above.

#### Table A.21 Floating Point Unit Instruction Format Encodings

<table>
<thead>
<tr>
<th>fmt field (bits 25..21 of COP1 opcode)</th>
<th>fmt3 field (bits 2..0 of COP1X opcode)</th>
<th>Mnemonic</th>
<th>Name</th>
<th>Bit Width</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>Hex</td>
<td>Decimal</td>
<td>Hex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0..15</td>
<td>00..0F</td>
<td>0</td>
<td>0</td>
<td>Used to encode Coprocessor 1 interface instructions (MFC1, CTC1, etc.). Not used for format encoding.</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>Single</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>Double</td>
</tr>
<tr>
<td>18..19</td>
<td>12..13</td>
<td>2..3</td>
<td>2..3</td>
<td>Reserved for future use by the architecture.</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>4</td>
<td>4</td>
<td>W</td>
<td>Word</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>5</td>
<td>5</td>
<td>L</td>
<td>Long</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>6</td>
<td>6</td>
<td>PS</td>
<td>Paired Single</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>7</td>
<td>7</td>
<td>Reserved for future use by the architecture.</td>
<td></td>
</tr>
<tr>
<td>24..31</td>
<td>18..1F</td>
<td>0</td>
<td>0</td>
<td>Reserved for future use by the architecture. Not available for fmt3 encoding.</td>
<td></td>
</tr>
</tbody>
</table>
Instruction Bit Encodings
Misaligned Memory Accesses

Prior to Release 5 the MIPS architectures\(^1\) require “natural” alignment of memory operands for most memory operations. Instructions such as LWL and LWR are provided so that unaligned accesses can be performed via instruction sequences. As of Release 5 of the Architecture the MSA (MIPS SIMD Architecture) supports 128 bit vector memory accesses, and does NOT require these MSA vector load and store instructions to be naturally aligned. The behavior, semantics, and architecture specifications of such misaligned accesses are described in this appendix.

B.1 Terminology

This document uses the following terminology:

- “Unaligned” and “misaligned” are used generically refer to any memory value not naturally aligned.

- The term “split” is used to refer to operations which cross important boundaries, whether architectural (e.g. “page split” or “segment split”) or microarchitectural (e.g. “cache line split”).

- The MIPS Architecture specifications have contained, since its beginning, special so-called Unaligned Load and Store instructions such as LWL/LWR and SWL/SWR (Load Word Left/Right, etc.)

- When necessary, we will call these “explicit unaligned memory access instructions”, as distinct from “instructions that permit implicit misaligned memory accesses”, such as MSA vector loads and stores.

- But where it is obvious from the context what we are talking about, we may say simply “unaligned” rather than the longer “explicit unaligned memory access instructions”, and “misaligned” rather than “instructions that permit implicit misaligned memory accesses”.

- Release 5 of the MIPS Architecture defines instructions, the MSA vector loads and stores, which may be aligned (e.g. 128-bits on a 128 bit boundary), partially aligned (e.g. “element aligned”, see below), or misaligned. These may be called verbosely “instructions that permit implicit misaligned memory accesses”.

- Misalignment is dynamic, known only when the address is computed (rather than static, explicit in the instruction as it is for LWL/LWR, etc.). We distinguish accesses for which the alignment is not yet known (“potentially misaligned”), from those whose alignment is known to be misaligned (“actually misaligned”), and from those for which the alignment is known to be naturally aligned (“actually aligned”).

  E.g. LL/SC instructions are never potentially misaligned, i.e. are always actually aligned (if they do not trap). MSA vector loads and stores are potentially misaligned, although the programmer or compiler may arrange so that particular instances will never be actually misaligned.

B.2 Hardware versus software support for misaligned memory accesses

Processors that implement versions of the MIPS Architectures prior to Release 5 require “natural” alignment of memory operands for most memory operations: apart from unaligned load and store instructions such as LW/LWR and SWL/SWR, all memory accesses that are not naturally aligned are required to signal an Address Error Exception.

Systems that implement Release 5 or higher of the MIPS Architectures require support for misaligned memory operands for the following instructions:

- MSA (MIPS SIMD Architecture) vector loads and stores (128-bit quantities)

In Release 5 all misaligned memory accesses other than MSA continue to produce the Address Error Exception with the appropriate ErrorCode.

In particular, misalignment support is NOT provided for the unaligned memory accesses, LWL/LWR and SWL/SWR. Nor is it provided for LL/SC. Nor for MIPS64 LDL/LDR and SDL/SDR, and LLD/SCD. Nor for the EVA versions LWLE/SWLE, LWRE/SWRE, LLE/SCE. All such instructions continue to produce the Address Error Exception if misaligned.

Note the phrasing “Systems that implement Release 5 or higher”. Processor hardware may provide varying degrees of support for misaligned accesses, producing the Address Error Exception in certain cases. The software Address Error Exception handler may then emulate the required misaligned memory access support in software. The term “systems that implement Release 5 or higher” includes such systems that combine hardware and software support. The processor in such a system by itself may not be fully Release 5 compliant because it does not support all misaligned memory references, but the combination of hardware and exception handler software may be.

Here are some examples of processor hardware providing varying degrees of support for misaligned accesses. The examples are named so that the different implementations can be discussed.

Full Misaligned Support:
Some processors may implement all the required misaligned memory access support in hardware.

Trap (and Emulate) All Misaligneds:
E.g. it is permitted for a processor implementation to produce the Address Error Exception for all misaligned accesses. I.e. with the appropriate exception handler software,

Trap (and Emulate) All Splits:
Intra-Cache-Line Misaligneds Support:
more accurately: Misaligneds within aligned 64B regions Support:
E.g. it is permitted for an implementation to perform misaligned accesses that fall entirely within a cache line in hardware, but to produce the Address Error Exception for all cache line splits and page splits.

Intra-Page Misaligneds Support:
more accurately: Misaligneds within aligned 4KB regions Support:
E.g. it is permitted for a processor implementation to perform cache line splits in hardware, but to produce the Address Error Exception for page splits.

Trap (and Emulate Page) Splits:
Distinct misaligned handling by memory type:
E.g. an implementation may perform misaligned accesses as described above for WB (Writeback) memory, but may produce the Address Error Exception for all misaligned accesses involving the UC memory type.
Misaligned Memory Accesses

Other mixes of hardware and software support are possible.

It is expected that Full Misaligned Support and Trap and Emulate Page Splits will be the most common implementations.

In general, actually misaligned memory accesses may be significantly slower than actually aligned memory accesses, even if an implementation provides Full Misaligned Support in hardware. Programmers and compilers should avoid actually misaligned memory accesses. Potentially but not actually misaligned memory accesses should suffer no performance penalty.

B.3 Detecting misaligned support

It is sufficient to check that MSA is present, as defined by the appropriate reference manual: i.e. support for misaligned MSA vector load and store instructions is required if the Config3 MSAP bit is set (CP0 Register 16, Select 3, bit 28).

The need for software to emulate misaligned support as described in the previous section must be detected by an implementation specific manner, and is not defined by the Architecture.

B.4 Misaligned semantics

B.4.1 Misaligned Fundamental Rules: Single Thread Atomic, but not Multi-thread

The following principles are fundamental for the other architecture rules relating to misaligned support.

Architecture Rule B-1: Misaligned memory accesses are atomic with respect to a single thread (with limited exceptions noted in other rules).

E.g. all interrupts and exceptions are delivered either completely before or completely after a misaligned (split) memory access. Such an exception handler is not entered with part of a misaligned load destination register written, and part unwritten. Similarly, it is not entered with part of a misaligned store memory destination written, and part unwritten.

E.g. uncorrectable ECC errors that occur halfway through a split store may violate single thread atomicity.

Hardware page table walking is not considered to covered by single thread atomicity.

Architecture Rule B-2: Memory accesses that are actually misaligned are not guaranteed to be atomic as observed from other threads, processors, and I/O devices.

B.4.2 Permissions and misaligned memory accesses

Architecture Rule B-3: It must be permitted to access every byte specified by a memory access.

Architecture Rule B-4: It is NOT required that permissions, etc., be uniform across all bytes.

2. E.g. MIPS® Architecture for Programmers, Volume IV-j: The MIPS32® SIMD Architecture Module, Document Number MD00866, 2013; or the corresponding documents for other MIPS Architectures such as MIPS64®
B.4 Misaligned semantics

This applies to all memory accesses, but in particular applies to misaligned split accesses, which can cross page boundaries and/or other boundaries that have different permissions. It *IS* permitted for a misaligned, in particular a page split memory access, to cross permission boundaries, as long as the access is permitted by permissions on both sides of the boundary. I.e. it is not required that the permissions be identical, for all parts, just that all parts are permitted.

**Architecture Rule B-5:** If any part of the misaligned memory access is not permitted, then the entire access must take the appropriate exception.

**Architecture Rule B-6:** If multiple exceptions arise for a given part of a misaligned memory access, then the same prioritization rules apply as for a non-misaligned memory access.

**Architecture Rule B-7:** If different exceptions are mandated for different parts of a split misaligned access, it is UNPREDICTABLE which takes priority and is actually delivered. But at least one of them must be delivered.

E.g. if a misaligned load is a page split, and one part of the load is to a page marked read-only, while the other is to a page marked invalid, the entire access must take the TLB Invalid Exception. The destination register will NOT be partially written.

E.g. if a misaligned store is a page split, and one part of the store is to a page marked writable, while the other is to a page marked read-only, the entire store must take the TLB Modified Exception. It is NOT permitted to write part of the access to memory, but not the other part.

E.g. if a misaligned memory access is a page split, and part is in the TLB and the other part is not - if software TLB miss handling is enabled then none of the access shall be performed before the TLB Refill Exception is entered.

E.g. if a misaligned load is a page split, and one part of the load is to a page marked read-only, while the other is to a page marked read-write, the entire access is permitted. I.e. a hardware implementation MUST perform the entire access. A hardware/software implementation may perform the access or take an Address Error Exception, but if it takes an Address Error Exception trap no part of the access may have been performed on arrival to the trap handler.

**B.4.3 Misaligned Memory Accesses Past the End of Memory**

**Architecture Rule B-8:** Misaligned memory accesses past the end of virtual memory are permitted, and behave as if a first partial access was done from the starting address to the virtual address limit, and a second partial access was done from the low virtual address for the remaining bytes.

E.g. an N byte misaligned memory access (N=16 for 128-bit MSA) starting M bytes below the end of the virtual address space “VMax” will access M bytes in the range [VMax-M+1,VMax], and in addition will access N-M bytes starting at the lowest virtual address “VMin”, the range [VMin, VMin+N-M-1].

E.g. for 32 bit virtual addresses, VMin=0 and VMax = 2^{32}-1, and an N byte access beginning M bytes below the top of the virtual address space expands to two separate accesses as follows: 2^{32} - M \Rightarrow [2^{32}-M,2^{32} - 1] \cup [0, 0 + N - M]

E.g. for 64 bit virtual addresses, VMin=0 and VMax = 2^{64}-1, and an N byte access beginning M bytes below the top of the virtual address space expands to two separate accesses as follows: 2^{64} - M \Rightarrow [2^{64}-M,2^{64} - 1] \cup [0, 0 + N - M]

Similarly, both 32 and 64 bit accesses can cross the corresponding signed boundaries, e.g. from, 0xFFFF_FFFF to 0x8000_0000 or from 0xFFFF_FFFF_FFFF_FFFF to 0x8000_0000_0000_0000.

**Architecture Rule B-9:** Beyond the wrapping at 32 or 64 bits mentioned, above, there is no special handling of accesses that cross MIPS segment boundaries, or which exceed SEGBITS within a MIPS segment.
Misaligned Memory Accesses

E.g. a 16 byte MSA access may begin in xuseg with a single byte at address 0x3FFF_FFFF_FFFF_FFFF and cross to xsseg, e.g. 15 bytes starting from 0x4000_000_0000_0000 - assuming consistent permissions and CCAs.

Architecture Rule B-10: Misaligned memory accesses must signal Address Error Exception if any part of the access would lie outside the physical address space.

E.g. if in an unmapped segment such as kseg0, and the start of the misaligned is below the PABITS limit, but the access size crosses the PABITS limit.

B.4.4 TLBs and Misaligned Memory Accesses

A specific case of rules stated above:

Architecture Rule B-11: if any part of a misaligned memory access involves a TLB miss, then none of the access shall be performed before the TLB miss handling exception is entered.

Here “performed” the actual store, changing memory or cache data values, or the actual load, writing a destination register, or load side effects related to memory mapped I/O. It does not refer to microarchitectural side effects such as changing cache line state from M in another processor to S locally, nor to TLB state.

Note: this rules does NOT disallow emulating misaligned memory accesses via a trap handler that performs the access a byte at a time, even though a TLB miss may occur for a later byte after an earlier byte has been written. Such a trap handler is emulating the entire misaligned. A TLB miss in the emulation code will return to the emulation code, not to the original misaligned memory instruction.

However, this rule DOES disallow handling permissions errors in this manner. Write permission must be checked in advance for all parts of a page split store.

Architecture Rule B-12: Misaligned memory accesses are not atomic with respect to hardware page table walking for TLB miss handling (as is added in MIPS Release 5).

Overall, TLBs, in particular hardware page table walking, are not considered to be part of “single thread atomicity”, and hardware page table walks are not ordered with the memory accesses of the loads and stores that trigger them.

E.g. the different parts of a split may occur at different times, and speculatively. If another processor is modifying the page tables without performing a TLB shootdown, the TLB entries found for a split may not have both occurred in memory at the same time.

E.g. on an exception triggered by a misaligned access, it is UNPREDICTABLE which TLB entries for a page split are in the TLB: both, one but not the other, or none.

Implementations must provide mechanisms to accommodate all parts of a misaligned load or store in order to guarantee forward progress. E.g. a certain minimum number of TLB entries may be required for the split parts of a misaligned memory access, and/or associated software TLB miss handlers or hardware TLB miss page table walkers. Other such mechanisms may not require extra TLB entries.

Architecture Rule B-13: Misaligned memory accesses are not atomic with respect to setting of PTE access and dirty
B.4 Misaligned semantics

bits.

E.g. if a hardware page table walker sets PTE dirty bit for both parts of a page split misaligned store, then it may be possible to observe one bit being set while the other is still not set.

**Architecture Rule B-14:** Misaligned memory accesses that affect any part of the page tables in memory that are used in performing the virtual to physical address translation of any part of the split access are UNPREDICTABLE.

E.g. a split store that writes one of its own PTEs - whether the hardware page table walker PTE, or whatever data structure a software PTE miss handler uses. (This means that a simple Address Error Exception handler can implement misaligneds without having to check page table addresses.)

**B.4.5 Memory Types and Misaligned Memory Accesses**

**Architecture Rule B-15:** Misaligned memory accesses are defined and are expected to be used for the following CCAs: WB (Writeback) and UCA (Uncached Accelerated), i.e. write combining.

**Architecture Rule B-16:** Misaligned memory accesses are defined for UC. Instructions that are potentially misaligned, but which are not actually misaligned, may safely be used with UC memory including MMIO. But instructions which are actually misaligned should not be used with MMIO - their results may be UNPREDICTABLE or worse.

Misaligned memory accesses are defined for the UC (Uncached) memory type, but their use is recommended only for ordinary uncached memory, DRAM or SRAM. The use of misaligned memory accesses is discouraged for uncached memory mapped I/O (MMIO) where accesses have side effects, because the specification of misaligned memory accesses does not specify the order or the atomicity in which the parts of the misaligned access are performed, which it makes it very difficult to use these accesses to control memory-mapped I/O devices with side effects.

**Architecture Rule B-17:** Misaligned memory accesses that cross two different CCA memory types are UNPREDICTABLE. (Reasons for this may include crossing of page boundaries, segment boundaries, etc.)

**Architecture Rule B-18:** Misaligned memory accesses that cross page boundaries, but with the same memory type in both pages, are permitted.

**Architecture Rule B-19:** Misaligned memory accesses that cross segment boundaries are well defined, so long as the memory types in both segments are the same and are otherwise permitted.

**B.4.6 Misaligneds, Memory Ordering, and Coherence**

This section discusses single and multithread atomicity and multithread memory ordering for misaligned memory accesses. But the overall Misaligned Memory Accesses specification, does not address issues for potentially but not actually misaligned memory references. Documents such as the MIPS Coherence Protocol Specification define such behavior.³

**B.4.6.1 Misaligneds are Single Thread Atomic**

Recall the first fundamental rule of misaligned support, single-thread atomicity:

---

³. E.g. MIPS Coherence Protocol Specification (AFP Version), Document Number MD00605, Revision 0.100. June 25, 2008. Updates and revisions of this document are pending.
Architecture Rule B-1: “Misaligned memory accesses are atomic with respect to a single thread (with limited exceptions noted in other rules).” on page 355.

E.g. all interrupts and exceptions are delivered either completely before or completely after a misaligned (split) memory access. Such an exception handler is not entered with part of a misaligned load destination register written, and part unwritten. Similarly, it is not entered with part of a misaligned store memory destination written, and part unwritten.

Architecture Rule B-20: However, an implementation may not be able to enforce single thread atomicity for certain error conditions.

Architecture Rule B-21: E.g. single thread atomicity for a misaligned, cache line or page split store, MAY be violated when an uncorrectable ECC error detected when performing a later part of a misaligned, when an part has already been performed, updating memory or cache.

Architecture Rule B-22: Nevertheless, implementations should avoid violating single thread atomicity whenever possible, even for error conditions.

Here are some exceptional or error conditions for which violating single thread atomicity for misaligneds is NOT acceptable: any event involving instruction access rather than data access, Debug data breakpoints, Watch address match, Address Error, TLB Refill, TLB Invalid, TLB Modified, Cache Error on load or LL, Bus Error on load or LL.

Machine Check Exceptions (a) are implementation dependent, (b) could potentially include a wide number of processor internal inconsistencies. However, at the time of writing the only Machine Check Exceptions that are defined are (a) detection of multiple matching entries in the TLB, and (b) inconsistencies in memory data structures encountered by the hardware page walker page table. Neither of these should cause a violation of single thread atomicity for misaligneds. In general, no errors related to virtual memory addresses should cause violations of single thread atomicity.

Architecture Rule B-23: Reset (Cold Reset) and Soft Reset are not required to respect single thread atomicity for misaligned memory accesses. E.g. Reset may be delivered when a store is only partly performed.

However, implementations are encouraged to make Reset and, in particular, Soft Reset, single instruction atomic whenever possible. E.g. a Soft Reset may be delivered to a processor that is not hung, when a misaligned store is only partially performed. If possible, the rest of the misaligned store should be performed. However, if the processor is stays hung with the misaligned store only partially performed, then the hang should time out and reset handling be completed.

Non-Maskable Interrupt (NMI) is required to respect single thread atomicity for misaligned memory accesses, since NMIs are defined to only be delivered at instruction boundaries.

B.4.6.2 Misaligneds are not Multithread/Multiprocessor Atomic

Recall the second fundamental rule of misaligneds - lack of multiprocessor atomicity:

Architecture Rule B-2: “Memory accesses that are actually misaligned are not guaranteed to be atomic as observed from other threads, processors, and I/O devices.” on page 355.

The rules in this section provide further detail.

Architecture Rule B-24: Instructions that are potentially but not actually misaligned memory accesses but which are not actually misaligned may be atomic, as observed from other threads, processors, or I/O devices.

The overall Misaligned Memory Accesses specification, does not address issues for potentially but not actually misaligned memory references. Documents such as the MIPS Coherence Protocol Specification define such behavior...
Architecture Rule B-25: Actually misaligned memory accesses may be performed more than one part. The order of these parts is not defined.

Architecture Rule B-26: It is UNPREDICTABLE and implementation dependent how many parts may be used to implement an actually misaligned memory access.

E.g. a page split store may be performed as two separate accesses, one for the low part, and one for the high part.

E.g. a misaligned access that is not split may be performed as a single access.

E.g. or a misaligned access - any misaligned access, not necessarily a split - may be performed a byte at a time.

Although most of this section has been emphasizing behavior that software cannot rely on, we can make the following guarantees:

Architecture Rule B-27: every byte written in a misaligned store will be written once and only once.

Architecture Rule B-28: a misaligned store will not be observed to write any bytes that are not specified; in particular, it will not do a read of memory that includes part of a split, merge, and then write the old and new data back.

Note the term “observed” in the rule above. E.g. memory and cache systems using word or line oriented ECC may perform read-modify-write in order to write a subword such as a byte. However, such ECC RMWs are atomic from the point of view of other processors, and do not affect bytes not written.

B.4.6.3 Misaligneds and Multiprocessor Memory Ordering

Preceding sections have defined misaligned memory accesses as having single thread atomicity but not multithread atomicity. Furthermore, there are issues related to memory ordering overall:

Architecture Rule B-29: Instructions that are potentially but not actually misaligned memory accesses comply with the MIPS Architecture rules for memory consistency, memory ordering, and synchronization.

This section Misaligned Memory Accesses, does not address issues for potentially but not actually misaligned memory references. Documents such as the MIPS Coherence Protocol Specification define such behavior.

Architecture Rule B-30: Although actually misaligned memory references may be split into several smaller references, as described in previous sections, these smaller references behave as described for any memory references in documents such as the MIPS Coherence Protocol Specification. In particular, misaligned subcomponent references respect the ordering and completion types of the SYNC instruction, legal and illegal sequences described in that document.

B.5 Pseudocode

Pseudocode can be convenient for describing the operation of instructions. Pseudocode is not necessarily a full specification, since it may not express all error conditions, all parallelism, or all non-determinism - all behavior left up to the implementation. Also, pseudocode may overspecify an operation, and appear to make guarantees that software should not rely on.

The first stage pseudocode provides functions LoadPossiblyMisaligned and StorePossiblyMisaligned that interface with other pseudocode via virtual address vAddr, the memory request size nbytes (=16 for 128b MSA), and arrays of byte data inbytes[nbytes] and inbytes[nbytes].
Misaligned Memory Accesses

The byte data is assumed to be permuted as required by the Big and Little endian byte ordering modes as required by the different instructions - thus permitting the pseudocode for misalignment support to be separated from the endianness considerations. I.e. outbytes[0] contains the value that a misaligned store will write to address vAddr+0, and so on.

The simplest thing that could possibly work would be to operate as follows:

```plaintext
for i in 0 .. nbytes-1
  (pAddr, CCA) ← AddressTranslation (vAddr+i, DATA, LOAD)
  inbytes[i] ← LoadRawMemory (CCA, nbytes, pAddr, vAddr+i, DATA)
endfor

for i in 0 .. nbytes-1
  (pAddr, CCA) ← AddressTranslation (vAddr+i, DATA, STORE)
  StoreRawMemory (CCA, 1, outbytes[i], pAddr, vAddr+i, DATA)
endfor
```

but this simplest possible pseudocode does not express the atomicity constraints and certain checks.

### B.5.1 Pseudocode distinguishing Actually Aligned from Actually Misaligned

The top level pseudocode functions LoadPossiblyMisaligned/StorePossiblyMisaligned take different paths depending on whether actually aligned or actually misaligned - to reflect the fact that aligned and misaligned have different semantics, different atomicity properties, etc.

**Figure B.1 LoadPossiblyMisaligned / StorePossiblyMisaligned pseudocode**

```plaintext
inbytes[nbytes] ← LoadPossiblyMisaligned(vaddr, nbytes)
  if naturally_aligned(vaddr, nbytes)
    return LoadAligned(vaddr, nbytes)
  else
    return LoadMisaligned(caddr, nbytes)
endfunction LoadPossiblyMisaligned

StorePossiblyMisaligned(vaddr, outbytes[nbytes])
  if naturally_aligned(vaddr, nbytes)
    StoreAligned(vaddr, nbytes)
  else
    StoreMisaligned(caddr, nbytes)
endfunction StorePossiblyMisaligned
```

### B.5.2 Actually Aligned

The aligned cases are very simple, and are defined to be a single standard operation from the existing pseudocode repertoire (except for byte swapping), reflecting the fact that actually aligned memory operations may have certain atomicity properties in both single and multithread situations.

**Figure B.2 LoadAligned / StoreAligned pseudocode**

```plaintext
inbytes[nbytes] ← LoadAligned(vaddr, nbytes)
  assert naturally_aligned(vaddr, nbytes)
  (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
  return inbytes[] ← LoadRawMemory (CCA, nbytes, pAddr, vAddr, DATA)
endfunction LoadAligned
```
B.5 Pseudocode

```
StoreAligned(vaddr, outbytes[nbytes])
    assert naturally_aligned(vaddr, nbytes)
    (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
    StoreRawMemory (CCA, nbytes, outbytes, pAddr, vAddr, DATA)
endfunction StoreAligned

B.5.3 Byte Swapping

The existing pseudocode uses functions LoadMemory and StoreMemory to access memory, which are declared but not defined. These functions implicitly perform any byteswapping needed by the Big and Little endian modes of the MIPS processor, which is acceptable for naturally aligned scalar data memory load and store operations. However, with vector operations and misaligned support, it is necessary to assemble the bytes from a memory load instruction, and only then to byteswap them - i.e.byteswapping must be exposed in the pseudocode. And conversely for stores.

**Figure B.3 LoadRawMemory Pseudocode Function**

```
MemElem ← LoadRawMemory (CCA, AccessLength, pAddr, vAddr, IorD)
    /* like the original pseudocode LoadMemory, except no byteswapping */
    /* MemElem: A vector of AccessLength bytes, in memory order. */
    /* 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 LoadRawMemory
```

**Figure B.4 StoreRawMemory Pseudocode Function**

```
StoreRawMemory (CCA, AccessLength, MemElem, pAddr, vAddr)
    /* like the original pseudocode StoreMemory, except no byteswapping */
    /* CCA: Cacheability&Coherency Attribute, the method used to access */
    /* caches and memory and resolve the reference. */
    /* AccessLength: Length, in bytes, of access */
    /* MemElem: A vector of AccessLength bytes, in memory order. */
    /* pAddr: physical address */
    /* vAddr: virtual address */
endfunction StoreRawMemory
```

Helper functions for byte swapping according to endianness:

**Figure B.5 Byteswapping pseudocode functions**

```
outbytes[nbytes] ← ByteReverse(inbytes[nbytes], nbytes)
    for i in 0 .. nbytes-1
        outbytes[nbytes-i] ← inbytes[i]
    endfor
    return outbytes[]
endfunction ByteReverse
```
Misaligned Memory Accesses

```c
outbytes[nbytes] ← ByteSwapIfNeeded(inbytes[nbytes], nbytes)
if BigEndianCPU then
    return ByteReverse(inbytes)
else
    return inbytes
endfunction ByteSwapIfNeeded
```

### B.5.4 Pseudocode Expressing Most General Misaligned Semantics

The misaligned cases have fewer constraints and more implementation freedom. The very general pseudocode below makes explicit some of the architectural rules that software can rely on, as well as many things that software should NOT rely on: lack of atomicity both between and within splits, etc. However, we emphasize that only the behavior guaranteed by the architecture rules should be relied on.

**Figure B.6 LoadMisaligned most general pseudocode**

```c
inbytes[nbytes] ← LoadMisaligned(vaddr, nbytes)
if any part of [vaddr,vaddr+nbytes) lies outside valid virtual address range
    then SignalException(...)
for i in 0 .. nbytes-1
    (pAddr[i], CCA[i]) ← AddressTranslation (vAddr+i, DATA, LOAD)
    if any pAddr[i] is invalid or not permitted then SignalException(...)
    if any CCA[i] != CCA[j], where i, j are in [0,nbytes) then UNPREDICTABLE
loop // in any order, and possibly in parallel
    pick an arbitrary subset S of [0,nbytes) that has not yet been loaded
    load inbytes[S] from memory with the corresponding CCA[i], pAddr[i], vAddr+i
    remove S from consideration
    until set of byte numbers remaining unloaded is empty.
return inbytes[]
endfunction LoadMisaligned
// ...similarly for StoreMisaligned...
```

### B.5.5 Example Pseudocode for Possible Implementations

This section provides alternative implementations of LoadMisaligned and StoreMisaligned that emphasize some of the permitted behaviors.

It is emphasized that these are not specifications, just examples. Examples to emphasize that particular implementations of misaligneds may be permitted. But these examples should not be relied on. Only the guarantees of the architecture rules should be relied on. The most general pseudocode seeks to express these in the most general possible form.

#### B.5.5.1 Example Byte-by-byte Pseudocode

The simplest possible implementation is to operate byte by byte. Here presented more formally than above, because the separate byte loads and stores expresses the desired lack of guaranteed atomicity (whereas for {Load,Store}PossiblyMisaligned the separate byte loads and stores would not express possible guarantees of atomicity). Similarly, the pseudocode translates the addresses twice, a first pass to check if there are any permissions errors, a second pass to actually use ordinary stores. UNPREDICTABLE behavior if the translations change between the two passes.

This pseudocode tries to indicate that it is permissible to use such a 2-phase approach in an exception handler to emulate misaligneds in software. It is not acceptable to use a single pass of byte by byte stores, unless split stores half per-
formed can be withdrawn, transactionally. But it is not required to save the translations of the first pass to reuse in the second pass (which would be extremely slow). If virtual addresses translations or

Figure B.7 Byte-by-byte pseudocode for LoadMisaligned / StoreMisaligned

```c
inbytes[nbytes] ← LoadMisaligned(vaddr, nbytes)
for i in 0 .. nbytes-1
   (ph1.pAddr[i], ph1.CCA[i]) ← AddressTranslation (vAddr+i, DATA, LOAD)
/* ... checks ... */
for i in 0 .. nbytes-1
   (ph2.pAddr[i], ph2.CCA[i]) ← AddressTranslation (vAddr+i, DATA, LOAD)
if ph1.pAddr[i] != ph2.addr or ph1.CCA[i] != ph2.CCA[i] then UNPREDICTABLE
inbytes[i] ← LoadRawMemory(ph2.CCA[i], nbytes, ph2.pAddr[i], vAddr+i, DATA)
return inbytes[]
endfunction LoadMisaligned

StoreMisaligned(vaddr, outbytes[nbytes])
for i in 0 .. nbytes-1
   (ph1.pAddr[i], ph1.CCA[i]) ← AddressTranslation (vAddr+i, DATA, LOAD)
/* ... checks ... */
for i in 0 .. nbytes-1
   (ph2.pAddr[i], ph2[i].CCA) ← AddressTranslation (vAddr+i, DATA, LOAD)
if ph1.pAddr[i] != ph2.addr or ph1.CCA[i] != ph2.CCA[i] then UNPREDICTABLE
StoreRawMemory(ph2[i].CCA, nbytes, outbytes[i], ph2.pAddr[i], vAddr+i, DATA)
endfunction StoreMisaligned
```

B.5.5.2 Example Pseudocode Handling Splits and non-Splits Separately

A more aggressive implementation, which is probably the preferred implementation on typical hardware, may:

- if a misaligned request is not split, it is performed as a single operation
- whereas if it is split it is performed as two separate operations, with cache line and page splits handled separately.

B.6 Misalignment and MSA vector memory accesses

B.6.1 Semantics

Misalignment support is defined by Release 5 of the MIPS Architecture only for MSA (MIPS SIMD Architecture)\(^4\) vector load and store instructions, including Vector Load (LD.df), Vector Load Indexed (LDX.df), Vector Store (ST.df) and Vector Store Indexed (STX.df). Each vector load and store has associated with it a data format, “df”, which can be byte/halfword/word/doubleword (B/H/W/D) (8/16/32/64 bits). The data format defines the vector element size.

The data format is used to determine Big-endian versus Little-endian byte swapping, and also influences multiprocessor atomicity as described here.

---

Architecture Rule B-31: Vector memory reference instructions are single thread atomic, as defined above.

Architecture Rule B-32: Vector memory reference instructions have element atomicity.

If the vector is aligned on the element boundary, i.e. if the vector address is equal to zero modulo 2, 4, 8 for H/W/D respectively, then for the purposes of multiprocessor memory ordering the vector memory reference instruction can be considered a set of vector element memory operations. The vector element memory operations may be performed in any order, but each vector element operation, since naturally aligned, has the atomicity of the corresponding scalar.

On MIPS32r5 16 and 32 bit scalar accesses are defined to be atomic, so e.g. each of the 32-bit elements of word vector loaded using LD.W would be atomic. However, on MIPS32r5 64 bit accesses are not defined to be atomic, so LD.D would not have element atomicity.

On MIPS64r5 16, 32, ad 64 bit scalar accesses are atomic. So vector LD.H, LD.W, LD.D, and the corresponding stores would be element atomic.

All of the rules in sections B.4.2 “Permissions and misaligned memory accesses”, B.4.4 “TLBs and Misaligned Memory Accesses”, B.4.5 “Memory Types and Misaligned Memory Accesses”, and B.4.6.1 “Misaligneds are Single Thread Atomic” apply to the vector load or store instructions as a whole.

E.g. a misaligned vector load instruction will never leave its vector destination register half written, if part of a page split succeeds and the other part takes an exception. It is either all done, or not at all.

E.g. misaligned vector memory references that partly fall outside the virtual address space are UNPREDICTABLE.

However, the multiprocessor and multithread oriented rules of section B.4.6.2 “Misaligneds are not Multithread/Multiprocessor Atomic” and B.4.6.3 “Misaligneds and Multiprocessor Memory Ordering” do NOT apply to the vector memory reference instruction as a whole. These rules only apply to vector element accesses.

In fact, all of the rules of B.4 “Misaligned semantics” apply to all vector element accesses - except where “overridden” for the vector as a whole.

E.g. a misaligned vector memory reference that crosses a memory type boundary, e.g. which is page split between WB and UCA CCA,s, is UNPREDICTABLE. Even though, if the vector as whole is vector element aligned, no vector element crosses such a boundary, so that if the vector element memory accesses were considered individually, each would be predictable.

B.6.2 Pseudocode for MSA memory operations with misalignment

The MSA specification uses the following pseudocode functions to access memory:

**Figure B.8 LoadTYPEVector / StoreTYPEVector used by MSA specification**

```plaintext
function LoadTYPEVector(ts, a, n)
    /* Implementation defined
       load ts, a vector of n TYPE elements
       from virtual address a.
    */
endfunction LoadTYPEVector

function StoreTYPEVector(tt, a, n)
    /* Implementation defined
       store tt, a vector of n TYPE elements
       to virtual address a.
    */
endfunction StoreTYPEVector
```
B.6 Misalignment and MSA vector memory accesses

These can be expressed in terms of the misaligned pseudocode operations as follows - passing the TYPE (Byte, Halfword, Word, Doubleword) as a parameter:

**Figure B.9 Pseudocode for LoadVector**

```plaintext
function LoadVector(vregdest, vAddr, nelem, TYPE)
    vector_wide_assertions(vAddr, nelem, TYPE)
    for all i in 0 to nelem-1 do /* in any order, any combination */
        rawtmp[i] ← LoadPossiblyMisaligned( vAddr + i*sizeof(TYPE), sizeof(TYPE) )
        btmp[i] ← ByteSwapIfNeeded( rawtmp[i], sizeof(TYPE) )
    /* vregdest.TYPE[i] ← btmp[i] */
    vregdestnbits(TYPE)*i+nbits(TYPE)-1..nbits(TYPE)*i = btmp[i]
    endfor
endfunction LoadVector
```

**Figure B.10 Pseudocode for StoreVector**

```plaintext
function StoreVector(vregsrc, vAddr, nelem, TYPE)
    vector_wide_assertions(vAddr, nelem, TYPE)
    for i in 0 .. nelem-1 /* in any order, any combination */
        btmp[i] ← vregsrcnbits(TYPE)*i+nbits(TYPE)-1..nbits(TYPE)*i
        rawtmp[i] ← ByteSwapIfNeeded( btmp[i], sizeof(TYPE) )
        StorePossiblyMisaligned( vAddr + i*sizeof(TYPE), sizeof(TYPE) )
    endfor
endfunction StoreVector
```
Misaligned Memory Accesses
## Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.90</td>
<td>November 1, 2000</td>
<td>Internal review copy of reorganized and updated architecture documentation.</td>
</tr>
<tr>
<td>0.91</td>
<td>November 15, 2000</td>
<td>Internal review copy of reorganized and updated architecture documentation.</td>
</tr>
<tr>
<td>0.92</td>
<td>December 15, 2000</td>
<td>Changes in this revision:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Correct sign in description of MSUBU.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Update JR and JALR instructions to reflect the changes required by MIPS16.</td>
</tr>
<tr>
<td>0.95</td>
<td>March 12, 2001</td>
<td>Update for second external review release</td>
</tr>
<tr>
<td>1.00</td>
<td>August 29, 2002</td>
<td>Update based on all review feedback:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Add missing optional select field syntax in mtc0/mfc0 instruction descriptions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Correct the PREF instruction description to acknowledge that the Prepare-ForStore function does, in fact, modify architectural state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• To provide additional flexibility for Coprocessor 2 implementations, extend the \texttt{sel} field for DMFC0, DMTC0, MFC0, and MTC0 to be 8 bits.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Update the PRE instruction to note that it may not update the state of a locked cache line.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Remove obviously incorrect documentation in DIV and DIVU with regard to putting smaller numbers in register \texttt{rt}.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Fix the description for MFC2 to reflect data movement from the coprocessor 2 register to the GPR, rather than the other way around.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Correct the pseudo code for LDC1, LDC2, SDC1, and SDC2 for a MIPS32 implementation to show the required word swapping.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Indicate that the operation of the CACHE instruction is UNPREDICTABLE if the cache line containing the instruction is the target of an invalidate or writeback invalidate.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Indicate that an Index Load Tag or Index Store Tag operation of the CACHE instruction must not cause a cache error exception.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Make the entire right half of the MFC2, MTC2, CFC2, CTC2, DMFC2, and DMTC2 instructions implementation dependent, thereby acknowledging that these fields can be used in any way by a Coprocessor 2 implementation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Clean up the definitions of LL, SC, LLD, and SCD.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Add a warning that software should not use non-zero values of the stype field of the SYNC instruction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Update the compatibility and subsetting rules to capture the current requirements.</td>
</tr>
<tr>
<td>Revision</td>
<td>Date</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 1.90     | September 1, 2002 | Merge the MIPS Architecture Release 2 changes in for the first release of a Release 2 processor. Changes in this revision include:  
• All new Release 2 instructions have been included: DI, EHB, EI, EXT, INS, JALR.HB, JR.HB, MFHC1, MFHC2, MTHC1, MTHC2, RDHWR, RDPGPR, ROTR, ROTRV, SEB, SEH, SYNCI, WRPGPR, WSBH.  
• The following instruction definitions changed to reflect Release 2 of the Architecture: DERET, ERET, JAL, JALR, JR, SRL, SRLV  
• With support for 64-bit FPUs on 32-bit CPUs in Release 2, all floating point instructions that were previously implemented by MIPS64 processors have been modified to reflect support on either MIPS32 or MIPS64 processors in Release 2.  
• All pseudo-code functions have been updated, and the Are64BitFP0OperationsEnabled function was added.  
• Update the instruction encoding tables for Release 2.  |
| 2.00     | June 9, 2003  | Continue with updates to merge Release 2 changes into the document. Changes in this revision include:  
• Correct the target GPR (from rd to rt) in the SLTI and SLTIU instructions. This appears to be a day-one bug.  
• Correct CPR number, and missing data movement in the pseudocode for the MTC0 instruction.  
• Add note to indicate that the CACHE instruction does not take Address Error Exceptions due to mis-aligned effective addresses.  
• Update SRL, ROTR, SRLV, ROTRV, DSRL, DROTR, DSRLV, DROTRV, DSRL32, and DROTR32 instructions to reflect a 1-bit, rather than a 4-bit decode of shift vs. rotate function.  
• Add programming note to the PrepareForStore PREF hint to indicate that it cannot be used alone to create a bzero-like operation.  
• Add note to the PREF and PREFX instruction indicating that they may cause Bus Error and Cache Error exceptions, although this is typically limited to systems with high-reliability requirements.  
• Update the SYNCI instruction to indicate that it should not modify the state of a locked cache line.  
• Establish specific rules for when multiple TLB matches can be reported (on writes only). This makes software handling easier.  |
| 2.50     | July 1, 2005  | Changes in this revision:  
• Correct figure label in LWR instruction (it was incorrectly specified as LWL).  
• Update all files to FrameMaker 7.1.  
• Include support for implementation-dependent hardware registers via RDHWR.  
• Indicate that it is implementation-dependent whether prefetch instructions cause EJTAG data breakpoint exceptions on an address match, and suggest that the preferred implementation is not to cause an exception.  
• Correct the MIPS32 pseudocode for the LDC1, LDXC1, LUXC1, SDC1, SDCX1, and SUXC1 instructions to reflect the Release 2 ability to have a 64-bit FPU on a 32-bit CPU. The correction simplifies the code by using the ValueFPR and StoreFPR functions, which correctly implement the Release 2 access to the FPRs.  
• Add an explicit recommendation that all cache operations that require an index be done by converting the index to a kseg0 address before performing the cache operation.  
• Expand on restrictions on the PREF instruction in cases where the effective address has an uncached coherency attribute.  
• |
<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
</table>
| 2.60     | June 25, 2008 | Changes in this revision:  
  • Applied the new B0.01 template.  
  • Update RDHWR description with the UserLocal register.  
  • added PAUSE instruction  
  • Ordering SYNCs  
  • CMP behavior of CACHE, PREF*, SYNCI  
  • CVT.S.PL, CVT.S.PU are non-arithmetic (no exceptions)  
  • *MADD.fmt & *MSUB fmt are non-fused.  
  • various typos fixed |
| 2.61     | July 10, 2008 |  
  • Revision History file was incorrectly copied from Volume III.  
  • Removed index conditional text from PAUSE instruction description.  
  • SYNC instruction - added additional format “SYNC stype” |
| 2.62     | January 2, 2009 |  
  • LWC1, LWXC1 - added statement that upper word in 64bit registers are UNDEFINED.  
  • CVT.S.PL and CVT.S.PU descriptions were still incorrectly listing IEEE exceptions.  
  • Typo in CFC1 Description.  
  • CCRes is accessed through $3 for RDHWR, not $4. |
| 3.00     | March 25, 2010 |  
  • JALX instruction description added.  
  • Sub-setting rules updated for JALX.  
  •  |
| 3.01     | June 01, 2010 |  
  • Copyright page updated.  
  • User mode instructions not allowed to produce UNDEFINED results, only UNPREDICTABLE results. |
| 3.02     | March 21, 2011 |  
  • RECP, RSQRT instructions do not require 64-bit FPU.  
  • MADD/MSUB/NMADD/NMSUB psuedo-code was incorrect for PS format check. |
| 3.50     | September 20, 2012 |  
  • Added EVA load/store instructions: LBE, LBU, LHE, LHUE, LWE, SBE, SHE, SWE, CACHEE, PREFE, LLE, SCE, LWLE, LWRE, SWLE, SWRE.  
  • TLBW1 - can be used to invalidate the VPN2 field of a TLB entry.  
  • FCSR.MAC2008 bit affects intermediate rounding in MADD fmt, MSUB fmt, NMADD fmt and NMSUB fmt.  
  • FCSR.ABS2008 bit defines whether ABS fmt and NEG fmt are arithmetic or not (how they deal with QNAN inputs). |
| 3.51     | October 20, 2012 |  
  • CACHE and SYNCI ignore RI and XI exceptions.  
  • CVT, CEIL, FLOOR, ROUND, TRUNC to integer can’t generate FP-Overflow exception. |
| 5.00     | December 14, 2012 |  
  • R5 changes: DSP and MT ASEs -> Modules  
  • NMADD.fmt, NMSUB fmt - for IEEE2008 negative portion is arithmetic. |
| 5.01     | December 15, 2012 |  
  • No technical content changes:  
  • Update logos on Cover.  
  • Update copyright page. |
<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
</table>
| 5.02     | April 22, 2013| • Fix: Figure 2.26 Are64BitFPOperationsEnabled Pseudocode Function - “Enabled” was missing.  
• R5 change retroactive to R3: removed FCSR.MCA2008 bit: no architectural support for fused multiply add with no intermediate rounding. Applies to MADD fmt, MSUB fmt, NMADD fmt, NMSUB fmt.  
• Clarification: references to “16 FP registers mode” changed to “the FR=0 32-bit register model”; specifically, paired single (PS) instructions and long (L) format instructions have UNPREDICTABLE results if FR=0, as well as LUXC1and SUXC1.  
• Clarification: C.cond fmt instruction page: cond bits 2..1 specify the comparison, cond bit 0 specifies ordered versus unordered, while cond bit 3 specifies signalling versus non-signalling.  
• R5 change: UFR (User mode FR change): CFC1, CTC1 changes.          |
| 5.03     | August 21, 2013| • Resolved inconsistencies with regards to the availability of instructions in MIPS32r2: MADD fmt family (MADD.S, MADD.D, NMADD.S, NMADD.D, MSUB.S, MSUB.D, NMSUB.S, NMSUB.D), RECIP fmt family (RECIPS, RECIPD, RSQRT.S, RSQRT.D), and indexed FP loads and stores (LWXC1, LDXC1, SWXC1, SDXC1). The appendix section A.2 “Instruction Bit Encoding Tables”, shared between Volume I and Volume II of the ARM, was updated, in particular the new upright delta ∆ is added to Table A.2 “Symbols Used in the Instruction Encoding Tables”, replacing the inverse delta marking ∇ for these instructions. Similar updates made to microMIPS’s corresponding sections. Instruction set descriptions and pseudocode in Volume II, Basic Instruction Set Architecture, updated. These instructions are required in MIPS32r2 if an FPU is implemented.  
• Misaligned memory access support for MSA: see Volume II, Appendix B “Misaligned Memory Accesses”.  
• Has2008 is is required as of release 5 - Table 5.4, “FIR Register Descriptions”.  
• ABS2008 and NAN2008 fields of Table 5.7 “FCSR Register Field Descriptions” were optional in release 3 and could be R/W, but as of release 5 are required, read-only, and preset by hardware.  
• FPU FCSR.FS Flush Subnormals / Flush to Zero behaviour is made consistent with MSA behaviour, in MSACSR.FS: Table 5.7, “FCSR Register Field Descriptions”, updated. New section 5.8.1.4 “Alternate Flush to Zero Underflow Handling”.  
• Volume I, Section 2.2 “Compliance ad Subsetting” noted that the L format is required in MIPS FPUs, to be consistent with Table 5.4 “FIR Register Field Definitions”.  
• Noted that UFR and UNFR can only be written with the value 0 from GPR[0]. See section 5.6.5 “User accessible FPU Register model control (UFR, CP1 Control Register 1)” and section 5.6.5 “User accessible Negated FPU Register model control (UNFR, CP1 Control Register 4)” |
| 5.04     | December 11, 2013| LLSC Related Changes  
• Added ERETN. New.  
• Modified SC handling: refined, added, and elaborated cases where SC can fail or was UNPREDICTABLE.  
XPA Related Changes  
• Added MTHC0, MFHC0 to access extensions. All new.  
• Modified MTC0 for MIPS32 to zero out the extended bits which are writable. This is to support compatibility of XPA hardware with non XPA software. In pseudo-code, added registers that are impacted.  
• MTHC0 and MFHC0 - Added RI conditions.  
