MIPS64® Architecture For Programmers
Volume II: The MIPS64® Instruction Set

Document Number: MD00087
Revision 2.50
July 1, 2005

MIPS Technologies, Inc.
1225 Charleston Road
Mountain View, CA 94043-1353

Copyright © 2001-2003,2005 MIPS Technologies Inc. All rights reserved.
# Table of Contents

Chapter 1 About This Book .............................................................................................................. 1
  1.1 Typographical Conventions ..................................................................................................... 1
    1.1.1 Italic Text ....................................................................................................................... 1
    1.1.2 Bold Text ...................................................................................................................... 1
    1.1.3 Courier Text .................................................................................................................. 1
  1.2 UNPREDICTABLE and UNDEFINED .................................................................................... 2
    1.2.1 UNPREDICTABLE .......................................................................................................... 2
    1.2.2 UNDEFINED .................................................................................................................. 2
  1.3 Special Symbols in Pseudocode Notation ................................................................................. 3
  1.4 For More Information ............................................................................................................. 5

Chapter 2 Guide to the Instruction Set ............................................................................................ 7
  2.1 Understanding the Instruction Fields ....................................................................................... 7
    2.1.1 Instruction Fields .......................................................................................................... 8
    2.1.2 Instruction Descriptive Name and Mnemonic ................................................................. 9
    2.1.3 Format Field ................................................................................................................ 9
    2.1.4 Purpose Field ............................................................................................................... 10
    2.1.5 Description Field ......................................................................................................... 10
    2.1.6 Restrictions Field ....................................................................................................... 10
    2.1.7 Operation Field ........................................................................................................... 11
    2.1.8 Exceptions Field ......................................................................................................... 11
    2.1.9 Programming Notes and Implementation Notes Fields .............................................. 11
    2.2 Operation Section Notation and Functions ......................................................................... 12
    2.2.1 Instruction Execution Ordering .................................................................................... 12
    2.2.2 Pseudocode Functions ................................................................................................ 12
  2.3 Op and Function Subfield Notation ......................................................................................... 22
  2.4 FPU Instructions ................................................................................................................... 23

Chapter 3 The MIPS64® Instruction Set ......................................................................................... 25
  3.1 Compliance and Subsetting .................................................................................................. 25
  3.2 Alphabetical List of Instructions .......................................................................................... 26
    ABS.fmt .................................................................................................................................. 37
    ADD ....................................................................................................................................... 38
    ADD.fmt ............................................................................................................................... 39
    ADDI ....................................................................................................................................... 40
    ADDIU .................................................................................................................................... 41
    ADDU .................................................................................................................................... 42
    ALNAV.PS ............................................................................................................................ 43
    AND ....................................................................................................................................... 46
    ANDI ....................................................................................................................................... 47
    B ........................................................................................................................................... 48
    BAL ....................................................................................................................................... 49
    BCIF ....................................................................................................................................... 50
    BC1FL ..................................................................................................................................... 52
    BC1T ....................................................................................................................................... 54
    BC1TL ..................................................................................................................................... 56
    BC2F ....................................................................................................................................... 58
    BC2FL ..................................................................................................................................... 59
    BC2T ....................................................................................................................................... 61
    BC2TL ..................................................................................................................................... 62
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td>64</td>
</tr>
<tr>
<td>BEQL</td>
<td>65</td>
</tr>
<tr>
<td>BGEZ</td>
<td>67</td>
</tr>
<tr>
<td>BGEOZALL</td>
<td>68</td>
</tr>
<tr>
<td>BGEOZL</td>
<td>69</td>
</tr>
<tr>
<td>BGTZ</td>
<td>71</td>
</tr>
<tr>
<td>BGTZL</td>
<td>72</td>
</tr>
<tr>
<td>BLEZ</td>
<td>73</td>
</tr>
<tr>
<td>BLEZL</td>
<td>74</td>
</tr>
<tr>
<td>BLTZ</td>
<td>76</td>
</tr>
<tr>
<td>BLTZL</td>
<td>77</td>
</tr>
<tr>
<td>BNE</td>
<td>79</td>
</tr>
<tr>
<td>BNEL</td>
<td>80</td>
</tr>
<tr>
<td>BREAK</td>
<td>81</td>
</tr>
<tr>
<td>C.cond.fmt</td>
<td>82</td>
</tr>
<tr>
<td>CACHE</td>
<td>83</td>
</tr>
<tr>
<td>CEIL.L.fmt</td>
<td>84</td>
</tr>
<tr>
<td>CEIL.W.fmt</td>
<td>85</td>
</tr>
<tr>
<td>CFC1</td>
<td>86</td>
</tr>
<tr>
<td>CFC2</td>
<td>87</td>
</tr>
<tr>
<td>CLO</td>
<td>88</td>
</tr>
<tr>
<td>COP2</td>
<td>89</td>
</tr>
<tr>
<td>CLZ</td>
<td>90</td>
</tr>
<tr>
<td>CTC1</td>
<td>91</td>
</tr>
<tr>
<td>CTC2</td>
<td>92</td>
</tr>
<tr>
<td>CVT.D.fmt</td>
<td>93</td>
</tr>
<tr>
<td>CVT.L.fmt</td>
<td>94</td>
</tr>
<tr>
<td>CVT.P.S.S.</td>
<td>95</td>
</tr>
<tr>
<td>CVT.S.fmt</td>
<td>96</td>
</tr>
<tr>
<td>CVT.S.PL</td>
<td>97</td>
</tr>
<tr>
<td>CVT.S.PU</td>
<td>98</td>
</tr>
<tr>
<td>CVT.W.fmt</td>
<td>99</td>
</tr>
<tr>
<td>DADD</td>
<td>100</td>
</tr>
<tr>
<td>DADDI</td>
<td>101</td>
</tr>
<tr>
<td>DADDIU</td>
<td>102</td>
</tr>
<tr>
<td>DADDU</td>
<td>103</td>
</tr>
<tr>
<td>DCLZ</td>
<td>104</td>
</tr>
<tr>
<td>DCLZ</td>
<td>105</td>
</tr>
<tr>
<td>DDIV</td>
<td>106</td>
</tr>
<tr>
<td>DDIVU</td>
<td>107</td>
</tr>
<tr>
<td>DERET</td>
<td>108</td>
</tr>
<tr>
<td>DEXT</td>
<td>109</td>
</tr>
<tr>
<td>DEXTM</td>
<td>110</td>
</tr>
<tr>
<td>DEXTU</td>
<td>111</td>
</tr>
<tr>
<td>DI</td>
<td>112</td>
</tr>
<tr>
<td>DINS</td>
<td>113</td>
</tr>
<tr>
<td>DINSN</td>
<td>114</td>
</tr>
<tr>
<td>DINSM</td>
<td>115</td>
</tr>
<tr>
<td>DINSU</td>
<td>116</td>
</tr>
<tr>
<td>DIV</td>
<td>117</td>
</tr>
<tr>
<td>DIV.fmt</td>
<td>118</td>
</tr>
<tr>
<td>DIVU</td>
<td>119</td>
</tr>
<tr>
<td>DMFC0</td>
<td>120</td>
</tr>
<tr>
<td>Instruction</td>
<td>Page</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>LWR</td>
<td>225</td>
</tr>
<tr>
<td>LWC2</td>
<td>221</td>
</tr>
<tr>
<td>LDXC1</td>
<td>210</td>
</tr>
<tr>
<td>LDL</td>
<td>205</td>
</tr>
<tr>
<td>LBU</td>
<td>201</td>
</tr>
<tr>
<td>LB</td>
<td>200</td>
</tr>
<tr>
<td>JR.HB</td>
<td>197</td>
</tr>
<tr>
<td>JR</td>
<td>195</td>
</tr>
<tr>
<td>J</td>
<td>188</td>
</tr>
<tr>
<td>JAL</td>
<td>189</td>
</tr>
<tr>
<td>JALR</td>
<td>190</td>
</tr>
<tr>
<td>JALR.HB</td>
<td>192</td>
</tr>
<tr>
<td>DMFC1</td>
<td>151</td>
</tr>
<tr>
<td>DMFC2</td>
<td>152</td>
</tr>
<tr>
<td>DMTC0</td>
<td>153</td>
</tr>
<tr>
<td>DMTC1</td>
<td>154</td>
</tr>
<tr>
<td>DMTC2</td>
<td>155</td>
</tr>
<tr>
<td>DMULT</td>
<td>156</td>
</tr>
<tr>
<td>DMULTU</td>
<td>157</td>
</tr>
<tr>
<td>DROTR</td>
<td>158</td>
</tr>
<tr>
<td>DROTR32</td>
<td>159</td>
</tr>
<tr>
<td>DROTRV</td>
<td>160</td>
</tr>
<tr>
<td>DSBH</td>
<td>161</td>
</tr>
<tr>
<td>DSHD</td>
<td>163</td>
</tr>
<tr>
<td>DSLL</td>
<td>165</td>
</tr>
<tr>
<td>DSLL32</td>
<td>166</td>
</tr>
<tr>
<td>DSLLV</td>
<td>167</td>
</tr>
<tr>
<td>DSRA</td>
<td>168</td>
</tr>
<tr>
<td>DSRA32</td>
<td>169</td>
</tr>
<tr>
<td>DSRAV</td>
<td>170</td>
</tr>
<tr>
<td>DSRL</td>
<td>171</td>
</tr>
<tr>
<td>DSRL32</td>
<td>172</td>
</tr>
<tr>
<td>DSRLV</td>
<td>173</td>
</tr>
<tr>
<td>DSUB</td>
<td>174</td>
</tr>
<tr>
<td>DSUBU</td>
<td>175</td>
</tr>
<tr>
<td>EHB</td>
<td>176</td>
</tr>
<tr>
<td>EI</td>
<td>177</td>
</tr>
<tr>
<td>ERET</td>
<td>179</td>
</tr>
<tr>
<td>EXT</td>
<td>181</td>
</tr>
<tr>
<td>FLOOR.L.fmt</td>
<td>183</td>
</tr>
<tr>
<td>FLOOR.W.fmt</td>
<td>185</td>
</tr>
<tr>
<td>INS</td>
<td>186</td>
</tr>
<tr>
<td>DMULT</td>
<td>156</td>
</tr>
<tr>
<td>DMULTU</td>
<td>157</td>
</tr>
<tr>
<td>DROTR</td>
<td>158</td>
</tr>
<tr>
<td>DROTR32</td>
<td>159</td>
</tr>
<tr>
<td>DROTRV</td>
<td>160</td>
</tr>
<tr>
<td>DSBH</td>
<td>161</td>
</tr>
<tr>
<td>DSHD</td>
<td>163</td>
</tr>
<tr>
<td>DSLL</td>
<td>165</td>
</tr>
<tr>
<td>DSLL32</td>
<td>166</td>
</tr>
<tr>
<td>DSLLV</td>
<td>167</td>
</tr>
<tr>
<td>DSRA</td>
<td>168</td>
</tr>
<tr>
<td>DSRA32</td>
<td>169</td>
</tr>
<tr>
<td>DSRAV</td>
<td>170</td>
</tr>
<tr>
<td>DSRL</td>
<td>171</td>
</tr>
<tr>
<td>DSRL32</td>
<td>172</td>
</tr>
<tr>
<td>DSRLV</td>
<td>173</td>
</tr>
<tr>
<td>DSUB</td>
<td>174</td>
</tr>
<tr>
<td>DSUBU</td>
<td>175</td>
</tr>
<tr>
<td>EHB</td>
<td>176</td>
</tr>
<tr>
<td>EI</td>
<td>177</td>
</tr>
<tr>
<td>ERET</td>
<td>179</td>
</tr>
<tr>
<td>EXT</td>
<td>181</td>
</tr>
<tr>
<td>FLOOR.L.fmt</td>
<td>183</td>
</tr>
<tr>
<td>FLOOR.W.fmt</td>
<td>185</td>
</tr>
<tr>
<td>INS</td>
<td>186</td>
</tr>
<tr>
<td>J</td>
<td>188</td>
</tr>
<tr>
<td>JAL</td>
<td>189</td>
</tr>
<tr>
<td>JALR</td>
<td>190</td>
</tr>
<tr>
<td>JALR.HB</td>
<td>192</td>
</tr>
<tr>
<td>JR</td>
<td>195</td>
</tr>
<tr>
<td>JR.HB</td>
<td>197</td>
</tr>
<tr>
<td>LB</td>
<td>200</td>
</tr>
<tr>
<td>LBU</td>
<td>201</td>
</tr>
<tr>
<td>LD</td>
<td>202</td>
</tr>
<tr>
<td>LDC1</td>
<td>203</td>
</tr>
<tr>
<td>LDC2</td>
<td>204</td>
</tr>
<tr>
<td>LDL</td>
<td>205</td>
</tr>
<tr>
<td>LDR</td>
<td>207</td>
</tr>
<tr>
<td>LDXC1</td>
<td>210</td>
</tr>
<tr>
<td>LH</td>
<td>211</td>
</tr>
<tr>
<td>LHU</td>
<td>212</td>
</tr>
<tr>
<td>LL</td>
<td>213</td>
</tr>
<tr>
<td>LLD</td>
<td>215</td>
</tr>
<tr>
<td>LUI</td>
<td>217</td>
</tr>
<tr>
<td>LUXC1</td>
<td>218</td>
</tr>
<tr>
<td>LW</td>
<td>219</td>
</tr>
<tr>
<td>LWC1</td>
<td>220</td>
</tr>
<tr>
<td>LWC2</td>
<td>221</td>
</tr>
<tr>
<td>LWL</td>
<td>222</td>
</tr>
<tr>
<td>LWR</td>
<td>225</td>
</tr>
<tr>
<td>Instruction</td>
<td>Page</td>
</tr>
<tr>
<td>--------------</td>
<td>------</td>
</tr>
<tr>
<td>LWU</td>
<td>229</td>
</tr>
<tr>
<td>LWXC1</td>
<td>230</td>
</tr>
<tr>
<td>MADD</td>
<td>231</td>
</tr>
<tr>
<td>MADD.fmt</td>
<td>232</td>
</tr>
<tr>
<td>MADDU</td>
<td>234</td>
</tr>
<tr>
<td>MFC0</td>
<td>235</td>
</tr>
<tr>
<td>MFC1</td>
<td>236</td>
</tr>
<tr>
<td>MFC2</td>
<td>237</td>
</tr>
<tr>
<td>MFHC1</td>
<td>238</td>
</tr>
<tr>
<td>MFHC2</td>
<td>239</td>
</tr>
<tr>
<td>MFHI</td>
<td>240</td>
</tr>
<tr>
<td>MFLO</td>
<td>241</td>
</tr>
<tr>
<td>MOV.fmt</td>
<td>242</td>
</tr>
<tr>
<td>MOVF</td>
<td>243</td>
</tr>
<tr>
<td>MOVF.fmt</td>
<td>244</td>
</tr>
<tr>
<td>MOVN</td>
<td>246</td>
</tr>
<tr>
<td>MOVN.fmt</td>
<td>247</td>
</tr>
<tr>
<td>MOVT</td>
<td>249</td>
</tr>
<tr>
<td>MOVT.fmt</td>
<td>250</td>
</tr>
<tr>
<td>MOVZ</td>
<td>252</td>
</tr>
<tr>
<td>MOVZ.fmt</td>
<td>253</td>
</tr>
<tr>
<td>MSUB</td>
<td>255</td>
</tr>
<tr>
<td>MSUB.fmt</td>
<td>256</td>
</tr>
<tr>
<td>MSUBU</td>
<td>258</td>
</tr>
<tr>
<td>MTC0</td>
<td>259</td>
</tr>
<tr>
<td>MTC1</td>
<td>260</td>
</tr>
<tr>
<td>MTC2</td>
<td>261</td>
</tr>
<tr>
<td>MTHC1</td>
<td>262</td>
</tr>
<tr>
<td>MTHC2</td>
<td>263</td>
</tr>
<tr>
<td>MTHI</td>
<td>264</td>
</tr>
<tr>
<td>MTLO</td>
<td>265</td>
</tr>
<tr>
<td>MUL.fmt</td>
<td>266</td>
</tr>
<tr>
<td>MUL.fmt</td>
<td>267</td>
</tr>
<tr>
<td>MUL</td>
<td>268</td>
</tr>
<tr>
<td>MULT</td>
<td>269</td>
</tr>
<tr>
<td>MULTU</td>
<td>270</td>
</tr>
<tr>
<td>NEG.fmt</td>
<td>271</td>
</tr>
<tr>
<td>NMADD.fmt</td>
<td>271</td>
</tr>
<tr>
<td>NMSUB.fmt</td>
<td>273</td>
</tr>
<tr>
<td>NOP</td>
<td>275</td>
</tr>
<tr>
<td>NOR</td>
<td>276</td>
</tr>
<tr>
<td>OR</td>
<td>277</td>
</tr>
<tr>
<td>ORI</td>
<td>278</td>
</tr>
<tr>
<td>PLL.PS</td>
<td>279</td>
</tr>
<tr>
<td>PLU.PS</td>
<td>280</td>
</tr>
<tr>
<td>PREF</td>
<td>281</td>
</tr>
<tr>
<td>PREFIX</td>
<td>285</td>
</tr>
<tr>
<td>PUL.PS</td>
<td>286</td>
</tr>
<tr>
<td>PUU.PS</td>
<td>287</td>
</tr>
<tr>
<td>RDHWR</td>
<td>288</td>
</tr>
<tr>
<td>RDPGPR</td>
<td>290</td>
</tr>
<tr>
<td>RECIP.fmt</td>
<td>291</td>
</tr>
<tr>
<td>ROTR</td>
<td>293</td>
</tr>
<tr>
<td>ROTRV</td>
<td>294</td>
</tr>
<tr>
<td>ROUND.L.fmt</td>
<td>295</td>
</tr>
<tr>
<td>ROUND.W.fmt</td>
<td>297</td>
</tr>
<tr>
<td>Instruction</td>
<td>Page</td>
</tr>
<tr>
<td>--------------</td>
<td>------</td>
</tr>
<tr>
<td>RSQRT.fmt</td>
<td>299</td>
</tr>
<tr>
<td>SB</td>
<td>301</td>
</tr>
<tr>
<td>SC</td>
<td>302</td>
</tr>
<tr>
<td>SCD</td>
<td>305</td>
</tr>
<tr>
<td>SDi</td>
<td>308</td>
</tr>
<tr>
<td>SDBBP</td>
<td>309</td>
</tr>
<tr>
<td>SDC1</td>
<td>310</td>
</tr>
<tr>
<td>SDC2</td>
<td>311</td>
</tr>
<tr>
<td>SDL</td>
<td>312</td>
</tr>
<tr>
<td>SDR</td>
<td>315</td>
</tr>
<tr>
<td>SDXC1</td>
<td>318</td>
</tr>
<tr>
<td>SEB</td>
<td>319</td>
</tr>
<tr>
<td>SEH</td>
<td>320</td>
</tr>
<tr>
<td>SH</td>
<td>322</td>
</tr>
<tr>
<td>SLL</td>
<td>323</td>
</tr>
<tr>
<td>SLLV</td>
<td>324</td>
</tr>
<tr>
<td>SLT</td>
<td>325</td>
</tr>
<tr>
<td>SLTI</td>
<td>326</td>
</tr>
<tr>
<td>SLTIU</td>
<td>327</td>
</tr>
<tr>
<td>SLTU</td>
<td>328</td>
</tr>
<tr>
<td>SQRT.fmt</td>
<td>329</td>
</tr>
<tr>
<td>SRA</td>
<td>330</td>
</tr>
<tr>
<td>SRAV</td>
<td>331</td>
</tr>
<tr>
<td>SRL</td>
<td>332</td>
</tr>
<tr>
<td>SRLV</td>
<td>333</td>
</tr>
<tr>
<td>SSNOP</td>
<td>334</td>
</tr>
<tr>
<td>SUB</td>
<td>335</td>
</tr>
<tr>
<td>SUB.fmt</td>
<td>336</td>
</tr>
<tr>
<td>SUBU</td>
<td>337</td>
</tr>
<tr>
<td>SUXC1</td>
<td>338</td>
</tr>
<tr>
<td>SW</td>
<td>339</td>
</tr>
<tr>
<td>SWC1</td>
<td>340</td>
</tr>
<tr>
<td>SWC2</td>
<td>341</td>
</tr>
<tr>
<td>SWL</td>
<td>342</td>
</tr>
<tr>
<td>SWR</td>
<td>344</td>
</tr>
<tr>
<td>SWXC1</td>
<td>346</td>
</tr>
<tr>
<td>SYNC</td>
<td>347</td>
</tr>
<tr>
<td>SYNCI</td>
<td>351</td>
</tr>
<tr>
<td>SYSCALL</td>
<td>354</td>
</tr>
<tr>
<td>TEQ</td>
<td>355</td>
</tr>
<tr>
<td>TEQI</td>
<td>356</td>
</tr>
<tr>
<td>TGE</td>
<td>357</td>
</tr>
<tr>
<td>TGEI</td>
<td>358</td>
</tr>
<tr>
<td>TGEIU</td>
<td>359</td>
</tr>
<tr>
<td>TGEU</td>
<td>360</td>
</tr>
<tr>
<td>TLBP</td>
<td>361</td>
</tr>
<tr>
<td>TLBR</td>
<td>362</td>
</tr>
<tr>
<td>TLBWI</td>
<td>364</td>
</tr>
<tr>
<td>TLBWR</td>
<td>366</td>
</tr>
<tr>
<td>TLT</td>
<td>368</td>
</tr>
<tr>
<td>TLTI</td>
<td>369</td>
</tr>
<tr>
<td>TLTIU</td>
<td>370</td>
</tr>
<tr>
<td>TLTU</td>
<td>371</td>
</tr>
<tr>
<td>TNEI</td>
<td>372</td>
</tr>
<tr>
<td>TNEI</td>
<td>373</td>
</tr>
</tbody>
</table>
Appendix A Instruction Bit Encodings ................................................................................................. 385
  A.1 Instruction Encodings and Instruction Classes ............................................................................... 385
  A.2 Instruction Bit Encoding Tables .................................................................................................... 385
  A.3 Floating Point Unit Instruction Format Encodings ....................................................................... 392
Appendix B Revision History ................................................................................................................ 395
List of Figures

Figure 2-1: Example of Instruction Description .................................................................................................. 8
Figure 2-2: Example of Instruction Fields ........................................................................................................ 9
Figure 2-3: Example of Instruction Descriptive Name and Mnemonic .............................................................. 9
Figure 2-4: Example of Instruction Format ......................................................................................................... 9
Figure 2-5: Example of Instruction Purpose ..................................................................................................... 10
Figure 2-6: Example of Instruction Exception .................................................................................................. 11
Figure 2-7: Example of Instruction Operation .................................................................................................. 11
Figure 2-8: Example of Instruction Restrictions ............................................................................................... 11
Figure 2-9: Example of Instruction Exception ................................................................................................. 11
Figure 2-10: Example of Instruction Programming Notes ................................................................................ 12
Figure 2-11: COP_LW Pseudocode Function .................................................................................................... 13
Figure 2-12: COP_LD Pseudocode Function .................................................................................................... 13
Figure 2-13: COP_SW Pseudocode Function .................................................................................................... 13
Figure 2-14: COP_SD Pseudocode Function .................................................................................................... 14
Figure 2-15: CoprocessorOperation Pseudocode Function ............................................................................... 14
Figure 2-16: AddressTranslation Pseudocode Function .................................................................................... 15
Figure 2-17: LoadMemory Pseudocode Function ............................................................................................. 15
Figure 2-18: StoreMemory Pseudocode Function ............................................................................................ 16
Figure 2-19: Prefetch Pseudocode Function .................................................................................................... 16
Figure 2-20: SyncOperation Pseudocode Function ........................................................................................ 17
Figure 2-21: ValueFPR Pseudocode Function .................................................................................................. 18
Figure 2-22: StoreFPR Pseudocode Function .................................................................................................. 19
Figure 2-23: CheckFPException Pseudocode Function ..................................................................................... 20
Figure 2-24: FPConditionCode Pseudocode Function ....................................................................................... 20
Figure 2-25: SetFPConditionCode Pseudocode Function ................................................................................ 20
Figure 2-26: SignalException Pseudocode Function ........................................................................................ 21
Figure 2-27: SignalDebugBreakpointException Pseudocode Function ............................................................. 21
Figure 2-28: SignalDebugModeBreakpointException Pseudocode Function ..................................................... 21
Figure 2-29: NullifyCurrentInstruction Pseudocode Function .......................................................................... 21
Figure 2-30: JumpDelaySlot Pseudocode Function ......................................................................................... 22
Figure 2-31: NotWordValue Pseudocode Function ........................................................................................ 22
Figure 2-32: PolyMult Pseudocode Function .................................................................................................. 22
Figure 3-1: Example of an ALNV.PS Operation ............................................................................................... 43
Figure 3-2: Usage of Address Fields to Select Index and Way ......................................................................... 95
Figure 3-3: Operation of the DEXT Instruction ................................................................................................. 132
Figure 3-4: Operation of the DEXTM Instruction ............................................................................................. 134
Figure 3-5: Operation of the DEXTU Instruction ............................................................................................. 136
Figure 3-6: Operation of the DINS Instruction .................................................................................................. 140
Figure 3-7: Operation of the DINSM Instruction ............................................................................................ 142
Figure 3-8: Operation of the DINSU Instruction ............................................................................................. 144
Figure 3-9: Operation of the EXT Instruction .................................................................................................. 181
Figure 3-10: Operation of the INS Instruction ................................................................................................. 186
Figure 3-11: Unaligned Doubleword Load Using LDL and LDR ................................................................. 205
Figure 3-12: Bytes Loaded by LDL Instruction ................................................................................................ 206
Figure 3-13: Unaligned Doubleword Load Using LDR and LDL ................................................................. 207
Figure 3-14: Bytes Loaded by LDR Instruction ................................................................................................ 208
Figure 3-15: Unaligned Word Load Using LWL and LWR .............................................................................. 222
Figure 3-16: Bytes Loaded by LWL Instruction ................................................................................................ 223
Figure 3-17: Unaligned Word Load Using LWL and LWR .............................................................................. 226
Figure 3-18: Bytes Loaded by LWR Instruction ............................................................................................... 227
List of Tables

Table 1-1: Symbols Used in Instruction Operation Statements ................................................................. 3
Table 2-1: AccessLength Specifications for Loads/Stores ........................................................................ 16
Table 3-1: CPU Arithmetic Instructions .................................................................................................. 26
Table 3-2: CPU Branch and Jump Instructions ......................................................................................... 27
Table 3-3: CPU Instruction Control Instructions ....................................................................................... 28
Table 3-4: CPU Load, Store, and Memory Control Instructions ................................................................. 28
Table 3-5: CPU Logical Instructions ......................................................................................................... 29
Table 3-6: CPU Insert/Extract Instructions ............................................................................................... 29
Table 3-7: CPU Move Instructions ........................................................................................................... 29
Table 3-8: CPU Shift Instructions ............................................................................................................ 30
Table 3-9: CPU Trap Instructions ............................................................................................................. 30
Table 3-10: Obsolete CPU Branch Instructions ........................................................................................ 31
Table 3-11: FPU Arithmetic Instructions ................................................................................................ 31
Table 3-12: FPU Branch Instructions ....................................................................................................... 32
Table 3-13: FPU Compare Instructions .................................................................................................... 32
Table 3-14: FPU Convert Instructions ...................................................................................................... 32
Table 3-15: FPU Load, Store, and Memory Control Instructions ............................................................... 33
Table 3-16: FPU Move Instructions .......................................................................................................... 33
Table 3-17: Obsolete FPU Branch Instructions ......................................................................................... 34
Table 3-18: Coprocessor Branch Instructions .......................................................................................... 34
Table 3-19: Coprocessor Execute Instructions ......................................................................................... 34
Table 3-20: Coprocessor Load and Store Instructions ............................................................................. 34
Table 3-21: Coprocessor Move Instructions ............................................................................................. 34
Table 3-22: Obsolete Coprocessor Branch Instructions ........................................................................... 35
Table 3-23: Privileged Instructions .......................................................................................................... 35
Table 3-24: EJTAG Instructions ................................................................................................................ 35
Table 3-25: FPU Comparisons Without Special Operand Exceptions .................................................... 90
Table 3-26: FPU Comparisons With Special Operand Exceptions for QNaNs ...................................... 91
Table 3-27: Usage of Effective Address .................................................................................................. 94
Table 3-28: Encoding of Bits[17:16] of the CACHE Instruction ............................................................... 95
Table 3-29: Encoding of Bits [20:18] of the CACHE Instruction .............................................................. 96
Table 3-30: Values of the hint Field for the PREF Instruction ................................................................. 282
Table 3-31: Hardware Register List ......................................................................................................... 288
Table A-1: Symbols Used in the Instruction Encoding Tables ................................................................. 386
Table A-2: MIPS64 Encoding of the Opcode Field ................................................................................... 387
Table A-3: MIPS64 SPECIAL Opcode Encoding of Function Field .......................................................... 388
Table A-4: MIPS64 REGIMM Encoding of rt Field .................................................................................... 388
Table A-5: MIPS64 SPECIAL2 Encoding of Function Field ..................................................................... 388
Table A-6: MIPS64 SPECIAL3 Encoding of Function Field for Release 2 of the Architecture ............... 388
Table A-7: MIPS64 MOVCI Encoding of tf Bit .......................................................................................... 389
Table A-8: MIPS64 SRL Encoding of Shift/Rotate ................................................................................... 389
Table A-9: MIPS64 SRLV Encoding of Shift/Rotate ................................................................................ 389
Table A-10: MIPS64 DSRLV Encoding of Shift/Rotate ............................................................................ 389
Table A-11: MIPS64 DSRL Encoding of Shift/Rotate ................................................................................ 389
Table A-12: MIPS64 DSRL32 Encoding of Shift/Rotate .......................................................................... 390
Table A-13: MIPS64 BSHFL and DBSHFL Encoding of sa Field ............................................................... 390
Table A-14: MIPS64 COP0 Encoding of rs Field ....................................................................................... 390
Table A-15: MIPS64 COP0 Encoding of Function Field When rs=CO .................................................... 390
Table A-16: MIPS64 COP1 Encoding of rs Field ....................................................................................... 391
Table A-17: MIPS64 COP1 Encoding of Function Field When rs=S ........................................................ 391
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-18</td>
<td>MIPS64 COP1 Encoding of Function Field When rs=D</td>
<td>391</td>
</tr>
<tr>
<td>A-19</td>
<td>MIPS64 COP1 Encoding of Function Field When rs=W or L</td>
<td>391</td>
</tr>
<tr>
<td>A-20</td>
<td>MIPS64 COP1 Encoding of Function Field When rs=PS</td>
<td>392</td>
</tr>
<tr>
<td>A-21</td>
<td>MIPS64 COP1 Encoding of tf Bit When rs=S, D, or PS, Function=MOVCF</td>
<td>392</td>
</tr>
<tr>
<td>A-22</td>
<td>MIPS64 COP2 Encoding of rs Field</td>
<td>392</td>
</tr>
<tr>
<td>A-23</td>
<td>MIPS64 COP1X Encoding of Function Field</td>
<td>392</td>
</tr>
<tr>
<td>A-24</td>
<td>Floating Point Unit Instruction Format Encodings</td>
<td>393</td>
</tr>
</tbody>
</table>
Chapter 1

About This Book

The MIPS64® Architecture For Programmers Volume II comes as a multi-volume set.

- Volume I describes conventions used throughout the document set, and provides an introduction to the MIPS64® Architecture
- Volume II provides detailed descriptions of each instruction in the MIPS64® instruction set
- Volume III describes the MIPS64® Privileged Resource Architecture which defines and governs the behavior of the privileged resources included in a MIPS64® processor implementation
- Volume IV-a describes the MIPS16e™ Application-Specific Extension to the MIPS64® Architecture
- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS64® Architecture
- Volume IV-c describes the MIPS-3D® Application-Specific Extension to the MIPS64® Architecture
- Volume IV-d describes the SmartMIPS®Application-Specific Extension to the MIPS32® Architecture and is not applicable to the MIPS64® document set

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.
- **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>x⁰</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 “b#” 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>x_y..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>
<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>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, 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>FPR[x]</td>
<td>Floating Point (Coprocessor unit 1), general register x</td>
</tr>
</tbody>
</table>
Table 1-1 Symbols Used in Instruction Operation Statements

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>BigEndianMem</td>
<td>Endian mode as configured at chip reset (0 (\rightarrow) Little-Endian, 1 (\rightarrow) Big-Endian). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory pseudocode function descriptions), and the endianness of Kernel and Supervisor mode execution.</td>
</tr>
<tr>
<td>BigEndianCPU</td>
<td>The endianness for load and store instructions (0 (\rightarrow) Little-Endian, 1 (\rightarrow) Big-Endian). In User mode, this endianness may be switched by setting the RE bit in the Status register. Thus, BigEndianCPU may be computed as (BigEndianMem XOR ReverseEndian).</td>
</tr>
<tr>
<td>ReverseEndian</td>
<td>Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is implemented by setting the RE bit of the Status register. Thus, ReverseEndian may be computed as (SRRE and User mode).</td>
</tr>
<tr>
<td>LLbit</td>
<td>Bit of virtual state used to specify operation for instructions that provide atomic read-modify-write. LLbit is set when a linked load occurs and is tested by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions.</td>
</tr>
<tr>
<td>I, I+n, I-n</td>
<td>This occurs as a prefix to Operation description lines and functions as a label. It indicates the instruction time during which the pseudocode appears to “execute.” Unless otherwise indicated, all effects of the current instruction appear to occur during the instruction time of the current instruction. No label is equivalent to a time label of I. Sometimes effects of an instruction appear to occur 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.</td>
</tr>
<tr>
<td>PC</td>
<td>The Program Counter value. During the instruction time of an instruction, this is the address of the instruction word. The address of the instruction that occurs during the next instruction time is determined by assigning a value to PC during an instruction time. If no value is assigned to PC during an instruction time by any pseudocode statement, it is automatically incremented by either 2 (in the case of a 16-bit MIPS16e instruction) or 4 before the next instruction time. A taken branch assigns the target address to the 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 64-bit address all of which are significant during a memory reference.</td>
</tr>
<tr>
<td>ISA Mode</td>
<td>In processors that implement the MIPS16e Application Specific Extension, the ISA Mode is a single-bit register that determines in which mode the processor is executing, as follows:</td>
</tr>
<tr>
<td>Encoding</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0</td>
<td>The processor is executing 32-bit MIPS instructions</td>
</tr>
<tr>
<td>1</td>
<td>The processor is executing MIIPS16e instructions</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>
Various MIPS RISC processor manuals and additional information about MIPS products can be found at the MIPS URL:

http://www.mips.com

Comments or questions on the MIPS64® Architecture or this document should be directed to

MIPS Architecture Group
MIPS Technologies, Inc.
1225 Charleston Road
Mountain View, CA 94043

or via E-mail to architecture@mips.com.
Chapter 2

Guide to the Instruction Set

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

2.1 Understanding the Instruction Fields

Figure 2-1 shows an example instruction. Following the figure are descriptions of the fields listed below:

- “Instruction Fields” on page 8
- “Instruction Descriptive Name and Mnemonic” on page 9
- “Format Field” on page 9
- “Purpose Field” on page 10
- “Description Field” on page 10
- “Restrictions Field” on page 10
- “Operation Field” on page 11
- “Exceptions Field” on page 11
- “Programming Notes and Implementation Notes Fields” on page 11
### Chapter 2 Guide to the Instruction Set

#### Figure 2-1 Example of Instruction Description

- **Instruction Mnemonic and Descriptive Name**: EXAMPLE

- **Instruction encoding constant and variable field names and values**:
  - **Format**: EXAMPLE rd, rs, rt

- **Architecture level at which instruction was defined/redefined and assembler format(s) for each definition**:
  - **MIPS32**

- **Short description**:

- **Symbolic description**:

- **Full description of instruction operation**:

- **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.

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

- **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] ← sign_extend(temp[31..0])
  ```

- **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

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](image)

### 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](image)

### 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.

**Format:** ADD rd, rs, rt

![Figure 2-4 Example of Instruction Format](image)

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.

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

**Figure 2-5 Example of Instruction Purpose**

### 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.

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

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

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

**Figure 2-6 Example of Instruction Description**

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

If either GPR rt or GPR rs does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Figure 2-7 Example of Instruction Restrictions

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.

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

Figure 2-8 Example of Instruction Operation

See Section 2.2, "Operation Section Notation and Functions" on page 12 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.

Exceptions:

Integer Overflow

Figure 2-9 Example of Instruction Exception

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

2.1.9 Programming Notes and Implementation Notes Fields
Chapter 2 Guide to the Instruction Set

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.

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

Figure 2-10 Example of Instruction Programming Notes

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 12
• “Pseudocode Functions” on page 12

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 12
• “Memory Operation Functions” on page 14
• “Floating Point Functions” on page 17
• “Miscellaneous Functions” on page 20

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.
**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 \( \text{memword} \) in coprocessor general register \( rt \).

\[
\text{COP_LW} \left( z, rt, \text{memword} \right)
\]
- \( z \): The coprocessor unit number
- \( rt \): Coprocessor general register specifier
- \( \text{memword} \): A 32-bit word value supplied to the coprocessor

/* Coprocessor-dependent action */

endfunction COP_LW

Figure 2-11 COP_LW Pseudocode Function

**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 \( \text{memdouble} \) in coprocessor general register \( rt \).

\[
\text{COP_LD} \left( z, rt, \text{memdouble} \right)
\]
- \( z \): The coprocessor unit number
- \( rt \): Coprocessor general register specifier
- \( \text{memdouble} \): 64-bit doubleword value supplied to the coprocessor.

/* Coprocessor-dependent action */

endfunction COP_LD

Figure 2-12 COP_LD Pseudocode Function

**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 \( rt \).

\[
dataword \leftarrow \text{COP_SW} \left( z, rt \right)
\]
- \( z \): The coprocessor unit number
- \( rt \): Coprocessor general register specifier
- \( dataword \): 32-bit word value

/* Coprocessor-dependent action */

endfunction COP_SW

Figure 2-13 COP_SW Pseudocode Function
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.

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

- \( z \): The coprocessor unit number
- \( rt \): Coprocessor general register specifier
- \( data_{\text{double}} \): 64-bit doubleword value

/* Coprocessor-dependent action */

endfunction COP_SD

Figure 2-14 COP_SD Pseudocode Function

CoprocessorOperation

The CoprocessorOperation function performs the specified Coprocessor operation.

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

- \( z \): Coprocessor unit number */
- \( \text{cop\_fun} \): Coprocessor function from function field of instruction */

/* Transmit the \( \text{cop\_fun} \) value to coprocessor \( z \) */

endfunction CoprocessorOperation

Figure 2-15 CoprocessorOperation Pseudocode Function

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 cache coherence algorithm, describing the mechanism used to resolve the memory reference.

Given the virtual address \( vAddr \), and whether the reference is to Instructions or Data (IorD), find the corresponding physical address \( pAddr \) and the cache coherence algorithm (CCA) used to resolve the reference. If the virtual address is in one of the unmapped address spaces, the physical address and CCA are determined directly by the virtual address. If the virtual address is in one of the mapped address spaces then the TLB 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.

\[(p\text{Addr}, CCA) \leftarrow \text{AddressTranslation} (v\text{Addr}, IorD, LorS)\]

\[/* p\text{Addr}: physical address */
/* CCA: Cache Coherence Algorithm, the method used to access caches*/
/* and memory and resolve the reference */

/* v\text{Addr}: virtual address */
/* IorD: Indicates whether access is for INSTRUCTION or DATA */
/* LorS: Indicates whether access is for LOAD or STORE */

/* See the address translation description for the appropriate MMU */
/* type in Volume III of this book for the exact translation mechanism */

endfunction AddressTranslation

**Figure 2-16 AddressTranslation Pseudocode Function**

**LoadMemory**

The LoadMemory function loads a value from memory.

This action uses cache and main memory as specified in both the Cache Coherence Algorithm (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.

\[\text{MemElem} \leftarrow \text{LoadMemory} (CCA, \text{AccessLength}, p\text{Addr}, v\text{Addr}, IorD)\]

\[/* \text{MemElem}: Data is returned in a fixed width with a natural alignment. The */
/* width is the same size as the CPU general-purpose register, */
/* 32 or 64 bits, aligned on a 32- or 64-bit boundary, */
/* respectively. */
/* CCA: Cache Coherence Algorithm, the method used to access caches */
/* and memory and resolve the reference */

/* AccessLength: Length, in bytes, of access */
/* p\text{Addr}: physical address */
/* v\text{Addr}: virtual address */
/* IorD: Indicates whether access is for Instructions or Data */

endfunction LoadMemory

**Figure 2-17 LoadMemory Pseudocode Function**

**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 Cache Coherence Algorithm (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 \textit{AccessLength} field indicate which of the bytes within the \textit{MemElem} data should be stored; only these bytes in memory will actually be changed.

\begin{verbatim}
StoreMemory (CCA, AccessLength, MemElem, pAddr, vAddr)

/* CCA: Cache Coherence Algorithm, the method used to access */
/* caches and memory and resolve the reference. */
/* AccessLength: Length, in bytes, of access */
/* MemElem: Data in the width and alignment of a memory element. */
/* The width is the same size as the CPU general */
/* purpose register, either 4 or 8 bytes, */
/* aligned on a 4- or 8-byte boundary. For a */
/* partial-memory-element store, only the bytes that will be*/
/* stored must be valid.*/
/* pAddr: physical address */
/* vAddr: virtual address */

endfunction StoreMemory

Figure 2-18 StoreMemory Pseudocode Function
\end{verbatim}

\textit{Prefetch}

The \textit{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.

\begin{verbatim}
Prefetch (CCA, pAddr, vAddr, DATA, hint)

/* CCA: Cache Coherence Algorithm, the method used to access */
/* caches and memory and resolve the reference. */
/* pAddr: physical address */
/* vAddr: virtual address */
/* DATA: Indicates that access is for DATA */
/* hint: hint that indicates the possible use of the data */

endfunction Prefetch

Figure 2-19 Prefetch Pseudocode Function
\end{verbatim}

Table 2-1 lists the data access lengths and their labels for loads and stores.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{AccessLength Name} & \textbf{Value} & \textbf{Meaning} \\
\hline
DOUBLEWORD & 7 & 8 bytes (64 bits) \\
SEPTIBYTE & 6 & 7 bytes (56 bits) \\
SEXTIBYTE & 5 & 6 bytes (48 bits) \\
QUINTIBYTE & 4 & 5 bytes (40 bits) \\
WORD & 3 & 4 bytes (32 bits) \\
TRIPLEBYTE & 2 & 3 bytes (24 bits) \\
HALFWORD & 1 & 2 bytes (16 bits) \\
BYTE & 0 & 1 byte (8 bits) \\
\hline
\end{tabular}
\caption{Table 2-1 AccessLength Specifications for Loads/Stores}
\end{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 stype occur in the same order for all processors.

```
SyncOperation(stype)

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

definition SyncOperation
```

Figure 2-20 SyncOperation Pseudocode Function

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.

\[
\text{value} \leftarrow \text{ValueFPR}(\text{fpr}, \text{fmt})
\]

/* 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 ← UNPREDICTABLE [32] FPR[fpr]31..0
  D, UNINTERPRETED_DOUBLEWORD:
    if (FP32RegistersMode = 0)
      if (fpr3 ≠ 0) then
        valueFPR ← UNPREDICTABLE
      else
        valueFPR ← FPR[fpr+1]31..0 FPR[fpr]31..0
      endif
    else
      valueFPR ← FPR[fpr]
    endif
  L, PS, OB, QH:
    if (FP32RegistersMode = 0) then
      valueFPR ← UNPREDICTABLE
    else
      valueFPR ← FPR[fpr]
    endif
  DEFAULT:
    valueFPR ← UNPREDICTABLE
endcase
endfunction ValueFPR
```

**Figure 2-21 ValueFPR Pseudocode Function**

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

StoreFPR (fpr, fmt, value)

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

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

case fmt of
  S, W, UNINTERPRETED_WORD:
    FPR[fpr] ← \texttt{UNPREDICTABLE}^{32} \parallel \texttt{value}_{31..0}
  D, UNINTERPRETED_DOUBLEWORD:
    if (FP32RegistersMode = 0)
      if (fpr0 \neq 0) then
        \texttt{UNPREDICTABLE}
      else
        FPR[fpr] ← \texttt{UNPREDICTABLE}^{32} \parallel \texttt{value}_{31..0}
        FPR[fpr+1] ← \texttt{UNPREDICTABLE}^{32} \parallel \texttt{value}_{63..32}
      endif
    else
      FPR[fpr] ← value
    endif
  L, PS, OB, QH:
    if (FP32RegistersMode = 0) then
      \texttt{UNPREDICTABLE}
    else
      FPR[fpr] ← value
    endif
endcase
endfunction StoreFPR

Figure 2-22 StoreFPR Pseudocode Function

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

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

definition CheckFPException

Figure 2-23 CheckFPException Pseudocode Function

FPConditionCode

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

tf ← FPConditionCode(cc)

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

if cc = 0 then
  FPConditionCode ← FCSR23
else
  FPConditionCode ← FCSR24+cc
endif

definition FPConditionCode

Figure 2-24 FPConditionCode Pseudocode Function

SetFPConditionCode

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

SetFPConditionCode(cc)

if cc = 0 then
  FCSR ← FCSR31..24 || tf || FCSR22..0
else
  FCSR ← FCSR31..25+cc || tf || FCSR23+cc..0
endif

definition SetFPConditionCode

Figure 2-25 SetFPConditionCode Pseudocode Function

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.

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

**Figure 2-26 SignalException Pseudocode Function**

**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.

```
SignalDebugBreakpointException()
endfunction SignalDebugBreakpointException
```

**Figure 2-27 SignalDebugBreakpointException Pseudocode Function**

**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.

```
SignalDebugModeBreakpointException()
endfunction SignalDebugModeBreakpointException
```

**Figure 2-28 SignalDebugModeBreakpointException Pseudocode Function**

**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.

```
NullifyCurrentInstruction()
endfunction NullifyCurrentInstruction
```

**Figure 2-29 NullifyCurrentInstruction Pseudocode Function**
Chapter 2 Guide to the Instruction Set

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.

```
JumpDelaySlot(vAddr)

   /* vAddr: Virtual address */

endfunction JumpDelaySlot
```

Figure 2-30 JumpDelaySlot Pseudocode Function

NotWordValue

The NotWordValue function returns a boolean value that determines whether the 64-bit value contains a valid word (32-bit) value. Such a value has bits 63..32 equal to bit 31.

```
result ← NotWordValue(value)

   /* result: True if the value is not a correct sign-extended word value; */
   /* False otherwise */

   /* value: A 64-bit register value to be checked */

NotWordValue ← value_{63..32} ≠ (value_{31})_{32}

endfunction NotWordValue
```

Figure 2-31 NotWordValue Pseudocode Function

PolyMult

The PolyMult function multiplies two binary polynomial coefficients.

```
PolyMult(x, y)

   temp ← 0
   for i in 0 .. 31
      if $x_i = 1$ then
         temp ← temp xor (y_{(31-i)..0} || 0^i)
      endif
   endfor

PolyMult ← temp

endfunction PolyMult
```

Figure 2-32 PolyMult Pseudocode Function

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 Section 2.3, "Op and Function Subfield Notation" on page 22 for a description of the op and function subfields.
Chapter 3

The MIPS64® Instruction Set

3.1 Compliance and Subsetting

To be compliant with the MIPS64 Architecture, designs must implement a set of required features, as described in this document set. To allow flexibility in implementations, the MIPS64 Architecture does provide subsetting rules. An implementation that follows these rules is compliant with the MIPS64 Architecture as long as it adheres strictly to the rules, and fully implements the remaining instructions. Supersetting of the MIPS64 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, and/or COP3 opcodes, or via the addition of approved Application Specific Extensions.

The instruction set subsetting rules are as follows:

• All CPU instructions must be implemented - no subsetting is allowed.

• 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, the paired single (PS) format is optional. 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 MIPS64 architecture:
  – No FPU
  – FPU with S, D, W, and L formats and all supporting instructions
  – FPU with S, D, PS, W, and L 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, DMFC2, DMTC2, LDC2, LWC2, MFC2, MTC2, SDC2, and SWC2) may be omitted on an instruction-by-instruction basis.

• Implementation of the full 64-bit address space is optional. The processor may implement 64-bit data and operations with a 32-bit only address space. In this case, the MMU acts as if 64-bit addressing is always disabled. Software may determine if the processor implements a 32-bit or 64-bit address space by checking the AT field in the Config CP0 register.

• Supervisor Mode is optional. If Supervisor Mode is not implemented, bit 3 of the Status register must be ignored on write and read as zero.

• 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 memory management unit is implemented, it must be the standard TLB-based MMU as described in the Privileged Resource Architecture chapter. Software may determine the type of the MMU by checking the MT field in the Config CP0 register.

• The Privileged Resource Architecture includes several implementation options and may be subsetted in accordance with those options.

• 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 ASEs are optional and may be subsetted out. If most cases, software may determine if a supported ASE is implemented by checking the appropriate bit in the Config1 or Config3 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 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 Alphabetical 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>DADD</td>
<td>Doubleword Add</td>
</tr>
<tr>
<td>DADDI</td>
<td>Doubleword Add immediate</td>
</tr>
<tr>
<td>DADDIU</td>
<td>Doubleword Add Immediate Unsigned</td>
</tr>
<tr>
<td>DADDU</td>
<td>Doubleword Add Unsigned</td>
</tr>
<tr>
<td>DCLO</td>
<td>Count Leading Ones in Doubleword</td>
</tr>
<tr>
<td>DCLZ</td>
<td>Count Leading Zeros in Doubleword</td>
</tr>
<tr>
<td>DDIV</td>
<td>Doubleword Divide</td>
</tr>
<tr>
<td>DDIVU</td>
<td>Doubleword Divide Unsigned</td>
</tr>
<tr>
<td>DIV</td>
<td>Divide Word</td>
</tr>
<tr>
<td>DIVU</td>
<td>Divide Unsigned Word</td>
</tr>
<tr>
<td>DMULT</td>
<td>Doubleword Multiply</td>
</tr>
<tr>
<td>DMULTU</td>
<td>Doubleword Multiply Unsigned</td>
</tr>
<tr>
<td>DSUB</td>
<td>Doubleword Subtract</td>
</tr>
<tr>
<td>DSUBU</td>
<td>Doubleword Subtract Unsigned</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>
</tbody>
</table>
### Table 3-1 CPU Arithmetic Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<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>
<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>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>
<th>Release 2 Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>EHB</td>
<td>Execution Hazard Barrier</td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
<td></td>
</tr>
<tr>
<td>SSNOP</td>
<td>Superscalar No Operation</td>
<td></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>LBU</td>
<td>Load Byte Unsigned</td>
</tr>
<tr>
<td>LD</td>
<td>Load Doubleword</td>
</tr>
<tr>
<td>LDL</td>
<td>Load Doubleword LEft</td>
</tr>
<tr>
<td>LDR</td>
<td>Load Doubleword Right</td>
</tr>
<tr>
<td>LH</td>
<td>Load Halfword</td>
</tr>
<tr>
<td>LHU</td>
<td>Load Halfword Unsigned</td>
</tr>
<tr>
<td>LL</td>
<td>Load Linked Word</td>
</tr>
<tr>
<td>LLD</td>
<td>Load Linked Doubleword</td>
</tr>
<tr>
<td>LW</td>
<td>Load Word</td>
</tr>
<tr>
<td>LWL</td>
<td>Load Word Left</td>
</tr>
<tr>
<td>LWR</td>
<td>Load Word Right</td>
</tr>
<tr>
<td>LWU</td>
<td>Load Word Unsigned</td>
</tr>
<tr>
<td>PREF</td>
<td>Prefetch</td>
</tr>
<tr>
<td>SB</td>
<td>Store Byte</td>
</tr>
<tr>
<td>SC</td>
<td>Store Conditional Word</td>
</tr>
<tr>
<td>SCD</td>
<td>Store Conditional Doubleword</td>
</tr>
<tr>
<td>SD</td>
<td>Store Doubleword</td>
</tr>
<tr>
<td>SDL</td>
<td>Store Doubleword LEft</td>
</tr>
<tr>
<td>SDR</td>
<td>Store Doubleword Right</td>
</tr>
<tr>
<td>SH</td>
<td>Store Halfword</td>
</tr>
<tr>
<td>SW</td>
<td>Store Word</td>
</tr>
<tr>
<td>SWL</td>
<td>Store Word Left</td>
</tr>
<tr>
<td>SWR</td>
<td>Store Word Right</td>
</tr>
<tr>
<td>SYNC</td>
<td>Synchronize Shared Memory</td>
</tr>
<tr>
<td>SYNCI</td>
<td>Synchronize Caches to Make Instruction Writes Effective</td>
</tr>
</tbody>
</table>
### 3.2 Alphabetical List of Instructions

#### 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>
<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>DEXT</td>
<td>Doubleword Extract Bit Field</td>
<td>Release 2 Only</td>
</tr>
<tr>
<td>DEXTM</td>
<td>Doubleword Extract Bit Field Middle</td>
<td>Release 2 Only</td>
</tr>
<tr>
<td>DEXTU</td>
<td>Doubleword Extract Bit Field Upper</td>
<td>Release 2 Only</td>
</tr>
<tr>
<td>DINS</td>
<td>Doubleword Insert Bit Field</td>
<td>Release 2 Only</td>
</tr>
<tr>
<td>DINSM</td>
<td>Doubleword Insert Bit Field Middle</td>
<td>Release 2 Only</td>
</tr>
<tr>
<td>DINSU</td>
<td>Doubleword Insert Bit Field Upper</td>
<td>Release 2 Only</td>
</tr>
<tr>
<td>DSBH</td>
<td>Doubleword Swap Bytes Within Halfwords</td>
<td>Release 2 Only</td>
</tr>
<tr>
<td>DSHD</td>
<td>Doubleword Swap Halfwords Within Doublewords</td>
<td>Release 2 Only</td>
</tr>
<tr>
<td>EXT</td>
<td>Extract Bit Field</td>
<td>Release 2 Only</td>
</tr>
<tr>
<td>INS</td>
<td>Insert Bit Field</td>
<td>Release 2 Only</td>
</tr>
<tr>
<td>WSBH</td>
<td>Word Swap Bytes Within Halfwords</td>
<td>Release 2 Only</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>MOVT</td>
<td>Move Conditional on Floating Point True</td>
</tr>
<tr>
<td>MOVZ</td>
<td>Move Conditional on Zero</td>
</tr>
<tr>
<td>MTHI</td>
<td>Move To HI Register</td>
</tr>
</tbody>
</table>
Chapter 3 The MIPS64® Instruction Set

### Table 3-7 CPU Move Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTLO</td>
<td>Move To LO Register</td>
</tr>
<tr>
<td>RDHWR</td>
<td>Read Hardware Register</td>
</tr>
<tr>
<td></td>
<td>Release 2 Only</td>
</tr>
</tbody>
</table>

### Table 3-8 CPU Shift Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROTR</td>
<td>Doubleword Rotate Right</td>
</tr>
<tr>
<td>DROTR32</td>
<td>Doubleword Rotate Right Plus 32</td>
</tr>
<tr>
<td>DROTRV</td>
<td>Doubleword Rotate Right Variable</td>
</tr>
<tr>
<td>DSLL</td>
<td>Doubleword Shift Left Logical</td>
</tr>
<tr>
<td>DSLL32</td>
<td>Doubleword Shift Left Logical Plus 32</td>
</tr>
<tr>
<td>DSLLV</td>
<td>Doubleword Shift Left Logical Variable</td>
</tr>
<tr>
<td>DSRA</td>
<td>Doubleword Shift Right Arithmetic</td>
</tr>
<tr>
<td>DSRA32</td>
<td>Doubleword Shift Right Arithmetic Plus 32</td>
</tr>
<tr>
<td>DSRAV</td>
<td>Doubleword Shift Right Arithmetic Variable</td>
</tr>
<tr>
<td>DSRL</td>
<td>Doubleword Shift Right Logical</td>
</tr>
<tr>
<td>DSRL32</td>
<td>Doubleword Shift Right Logical Plus 32</td>
</tr>
<tr>
<td>DSRLV</td>
<td>Doubleword Shift Right Logical Variable</td>
</tr>
<tr>
<td>ROTR</td>
<td>Rotate Word Right</td>
</tr>
<tr>
<td>ROTRV</td>
<td>Rotate Word Right Variable</td>
</tr>
<tr>
<td>SLL</td>
<td>Shift Word Left Logical</td>
</tr>
<tr>
<td>SLLV</td>
<td>Shift Word Left Logical Variable</td>
</tr>
<tr>
<td>SRA</td>
<td>Shift Word Right Arithmetic</td>
</tr>
<tr>
<td>SRAV</td>
<td>Shift Word Right Arithmetic Variable</td>
</tr>
<tr>
<td>SRL</td>
<td>Shift Word Right Logical</td>
</tr>
<tr>
<td>SRLV</td>
<td>Shift Word Right Logical Variable</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>
</tbody>
</table>
### Table 3-9 CPU Trap Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<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 Obsolete\(^1\) 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>BGTLZL</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 MIPS64 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>
<tr>
<td>MSUB.fmt</td>
<td>Floating Point Multiply Subtract</td>
</tr>
<tr>
<td>MUL.fmt</td>
<td>Floating Point Multiply</td>
</tr>
<tr>
<td>NEG.fmt</td>
<td>Floating Point Negate</td>
</tr>
<tr>
<td>NMADD.fmt</td>
<td>Floating Point Negative Multiply Add</td>
</tr>
</tbody>
</table>
### Table 3-11 FPU Arithmetic Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMSUB.fmt</td>
<td>Floating Point Negative Multiply Subtract</td>
</tr>
<tr>
<td>RECIP.fmt</td>
<td>Reciprocal Approximation</td>
</tr>
<tr>
<td>RSQRT.fmt</td>
<td>Reciprocal Square Root Approximation</td>
</tr>
<tr>
<td>SQRT.fmt</td>
<td>Floating Point Square Root</td>
</tr>
<tr>
<td>SUB.fmt</td>
<td>Floating Point Subtract</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>64-bit FPU Only</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>64-bit FPU Only</td>
</tr>
<tr>
<td>CVT.PS.S</td>
<td>Floating Point Convert Pair to Paired Single</td>
<td>64-bit FPU Only</td>
</tr>
<tr>
<td>CVT.S.PL</td>
<td>Floating Point Convert Pair Lower to Single Floating Point</td>
<td>64-bit FPU Only</td>
</tr>
<tr>
<td>CVT.S.PU</td>
<td>Floating Point Convert Pair Upper to Single Floating Point</td>
<td>64-bit FPU Only</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>64-bit FPU Only</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>64-bit FPU Only</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>
</tbody>
</table>
### Table 3-14 FPU Convert Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th></th>
</tr>
</thead>
<tbody>
<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></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>64-bit FPU Only</td>
</tr>
<tr>
<td>LUXC1</td>
<td>Load Doubleword Indexed Unaligned to Floating Point</td>
<td>64-bit FPU Only</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>64-bit FPU Only</td>
</tr>
<tr>
<td>PFXML</td>
<td>Prefetch Indexed</td>
<td></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>64-bit FPU Only</td>
</tr>
<tr>
<td>SUXC1</td>
<td>Store Doubleword Indexed Unaligned from Floating Point</td>
<td>64-bit FPU Only</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>64-bit FPU Only</td>
</tr>
</tbody>
</table>

### Table 3-16 FPU Move Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th></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>DMFC1</td>
<td>Doubleword Move from Floating Point</td>
<td></td>
</tr>
<tr>
<td>DMTC1</td>
<td>Doubleword Move 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 Only</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>MOVT.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>
</tbody>
</table>
### Table 3-16 FPU Move Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTC1</td>
<td>Move Word to Floating Point</td>
</tr>
<tr>
<td>MTHC1</td>
<td>Move Word to High Half of Floating Point Register</td>
</tr>
</tbody>
</table>

### 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 MIPS64 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>DMFC2</td>
<td>Doubleword Move from Coprocessor 2</td>
</tr>
<tr>
<td>DMTC2</td>
<td>Doubleword Move to Coprocessor 2</td>
</tr>
<tr>
<td>MFC2</td>
<td>Move Word from Coprocessor 2</td>
</tr>
</tbody>
</table>
### Table 3-21 Coprocessor Move Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFHC2</td>
<td>Move Word from High Half of Coprocessor 2 Register</td>
<td>Release 2 Only</td>
</tr>
<tr>
<td>MTC2</td>
<td>Move Word to Coprocessor 2</td>
<td></td>
</tr>
<tr>
<td>MTHC2</td>
<td>Move Word to High Half of Coprocessor 2 Register</td>
<td>Release 2 Only</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 MIPS64 architecture.

### Table 3-23 Privileged Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHE</td>
<td>Perform Cache Operation</td>
</tr>
<tr>
<td>DI</td>
<td>Disable Interrupts</td>
</tr>
<tr>
<td>DMFC0</td>
<td>Doubleword Move from Coprocessor 0</td>
</tr>
<tr>
<td>DMTC0</td>
<td>Doubleword Move to Coprocessor 0</td>
</tr>
<tr>
<td>EI</td>
<td>Enable Interrupts</td>
</tr>
<tr>
<td>ERET</td>
<td>Exception Return</td>
</tr>
<tr>
<td>MFC0</td>
<td>Move from Coprocessor 0</td>
</tr>
<tr>
<td>MTC0</td>
<td>Move to Coprocessor 0</td>
</tr>
<tr>
<td>RDPGPR</td>
<td>Read GPR from Previous Shadow Set</td>
</tr>
<tr>
<td>TLBP</td>
<td>Probe TLB for Matching Entry</td>
</tr>
<tr>
<td>TLBR</td>
<td>Read Indexed TLB Entry</td>
</tr>
<tr>
<td>TLBWI</td>
<td>Write Indexed TLB Entry</td>
</tr>
<tr>
<td>TLBWR</td>
<td>Write Random TLB Entry</td>
</tr>
<tr>
<td>WAIT</td>
<td>Enter Standby Mode</td>
</tr>
<tr>
<td>WRPGPR</td>
<td>Write GPR to Previous Shadow Set</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

<table>
<thead>
<tr>
<th>COP1</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>010001</td>
<td>fmt</td>
<td>0</td>
<td>fs</td>
<td>fd</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

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

**MIPS32**  
**MIPS32**  
**MIPS64, MIPS32 Release 2**

**Purpose:**

To compute the absolute value of an FP 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.

This operation is arithmetic; a NaN operand signals invalid operation.

**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 16 FP registers mode.

**Operation:**

StoreFPR(fd, fmt, AbsoluteValue(ValueFPR(fs, fmt)))

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation, Invalid Operation
ADD

**Format:** ADD rd, rs, rt

**MIPS32**

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

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

The 32-bit word value in GPR rt is added to the 32-bit value in GPR rs to produce a 32-bit result.

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

**Restrictions:**
If either GPR rt or GPR rs does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

**Operation:**
```java
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif

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

**Exceptions:**
Integer Overflow

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

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

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

**Purpose:**
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 16 FP registers mode.

**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

<table>
<thead>
<tr>
<th>Format:</th>
<th>ADDI rt, rs, immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>To add a constant to a 32-bit integer. If overflow occurs, then trap.</td>
</tr>
</tbody>
</table>

**Description:**

\[ GPR[rt] \leftarrow GPR[rs] + \text{immediate} \]

The 16-bit signed \text{immediate} is added to the 32-bit value in GPR \text{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 sign-extended and placed into GPR \text{rt}.

**Restrictions:**

If GPR \text{rs} does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

**Operation:**

```
if NotWordValue(GPR[rs]) then
    UNPREDICTABLE
endif

temp \leftarrow (GPR[rs]31\|GPR[rs]31..0) + \text{sign} \_ \text{extend}(\text{immediate})
if temp32 \neq temp31 then
    SignalException(IntegerOverflow)
else
    GPR[rt] \leftarrow \text{sign} \_ \text{extend}(\text{temp31..0})
endif
```

**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:
To add a constant to a 32-bit integer

Description:
GPR[rt] ← GPR[rs] + immediate
The 16-bit signed immediate is added to the 32-bit value in GPR rs and the 32-bit arithmetic result is sign-extended and placed into GPR rt.
No Integer Overflow exception occurs under any circumstances.

Restrictions:
If GPR rs does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

if NotWordValue(GPR[rs]) then
    UNPREDICTABLE
endif

temp ← GPR[rs] + sign_extend(immediate)
GPR[rt] ← sign_extend(temp31..0)

Exceptions:
None

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

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>ADDU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: ADDU rd, rs, rt

MIPS32

Purpose:
To add 32-bit integers

Description: GPR[rd] ← 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 sign-extended and placed into GPR rd.

No Integer Overflow exception occurs under any circumstances.

Restrictions:
If either GPR rt or GPR rs does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```plaintext
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif

temp ← GPR[rs] + GPR[rt]
GPR[rd] ← sign_extend(temp31..0)
```

Exceptions:
None

Programming Notes:
The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. 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

| Format: | ALNV.PS fd, fs, ft, rs |
| Purpose: | To align a misaligned pair of paired single values |
| Description: | FPR[fd] ← 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.

The move is nonarithmetic; it causes no IEEE 754 exceptions.
Restrictions:
The fields \( f_s, f_t, \) and \( f_d \) 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 16 FP registers mode.

Operation:

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

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:

\[
\begin{align*}
& \text{/* Copy T2 bytes (a multiple of 16) of data T0 to T1, T0 unaligned, T1 aligned.} \\
& \text{Reads one dw beyond the end of T0. */} \\
& \text{LUXC1} \ F0, 0(T0) /* set up by reading 1st src dw */ \\
& \text{LI} \ T3, 0 /* index into src and dst arrays */ \\
& \text{ADDIU} \ T4, T0, 8 /* base for odd dw loads */ \\
& \text{ADDIU} \ T5, T1, -8 /* base for odd dw stores */ \\
& \text{LOOP:} \\
& \text{LUXC1} \ F1, T3(T4) \\
& \text{ALNV.PS} \ F2, F0, F1, T0/* switch F0, F1 for little-endian */ \\
& \text{SDC1} \ F2, T3(T1) \\
& \text{ADDIU} \ T3, T3, 16 \\
& \text{LUXC1} \ F0, T3(T0) \\
& \text{ALNV.PS} \ F2, F1, F0, T0/* switch F1, F0 for little-endian */ \\
& \text{BNE} \ T3, T2, \text{LOOP} \\
& \text{SDC1} \ F2, T3(T5) \\
& \text{DONE:}
\end{align*}
\]
ALNVP.S 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 */
AND

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

Purpose:
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**

<table>
<thead>
<tr>
<th>ANDI</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>001100</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Format:** ANDI rt, rs, immediate  

**MIPS32**

**Purpose:**
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
Format: \texttt{B\ offset}

Purpose:
To do an unconditional branch

Description: \texttt{branch}

\texttt{B\ offset} is the assembly idiom used to denote an unconditional branch. The actual instruction is interpreted by the hardware as \texttt{BEQ r0, r0, offset}.

An 18-bit signed offset (the 16-bit \texttt{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 \texttt{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{target\_offset} & \leftarrow & \text{sign\_extend}(\text{offset} \| 0^2) \\
\text{I+1:} & & \text{PC} & \leftarrow & \text{PC} + \text{target\_offset}
\end{align*}

Exceptions:
None

Programming Notes:
With the 18-bit signed instruction offset, the conditional branch range is $\pm 128$ Kbytes. Use \texttt{jump (J)} or \texttt{jump register (JR)} instructions to branch to addresses outside this range.
Branch and Link

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

**Purpose:**
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 \text{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 \text{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:**

\[
\begin{align*}
\text{I:} & \quad \text{target_offset} \leftarrow \text{sign extend}(\text{offset} \mid\!\mid 0^2) \\
& \quad \text{GPR}[31] \leftarrow \text{PC} + 8 \\
\text{I+1:} & \quad \text{PC} \leftarrow \text{PC} + \text{target_offset}
\end{align*}
\]

**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.
Format:

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

Purpose:
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 con-
dition 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 ← (offset_{15})^{GPRLEN-(16+2)} || offset || 0^2 |

I+1:
| if condition then |
| PC ← PC + target_offset |
| endif |
Branch on FP False (cont.)

**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

**Format:**

<table>
<thead>
<tr>
<th>BC1FL</th>
<th>offset (cc = 0 implied)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC1FL</td>
<td>cc, offset</td>
</tr>
</tbody>
</table>

**MIPS32**

**Purpose:**

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 ← (offset15)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

MIPS32

MIPS32

Purpose:
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)\text{GPRLEN-(16+2)} || offset || 0^2

I+1:  
if condition then
    PC ← PC + target_offset
endif
Branch on FP True (cont.)

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

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>BC</td>
<td>cc</td>
<td>nd</td>
<td>tf</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010001</td>
<td>01000</td>
<td>1</td>
<td>1</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

BC1TL  offset (cc = 0 implied)  
BC1TL  cc, offset  

**Purpose:**

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.

I:

\[
\text{condition} \leftarrow \text{FPConditionCode}(cc) = 1 \\
\text{target_offset} \leftarrow (\text{offset}_{15})_{\text{GPRLEN}} - (16+2) \| \text{offset} \| 0^2
\]

I+1:

if condition then
    PC \leftarrow PC + target_offset
else
    NullifyCurrentInstruction()
endif
Branch on FP True Likely (cont.)

<table>
<thead>
<tr>
<th>Exceptions:</th>
<th>BCITL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coprocessor Unusable, Reserved Instruction</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Floating Point Exceptions:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unimplemented Operation</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programming Notes:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Historical Information:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>
### Branch on COP2 False

**Format:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC2F</td>
<td>offset (cc = 0 implied)</td>
</tr>
<tr>
<td>BC2F</td>
<td>cc, offset</td>
</tr>
</tbody>
</table>

**Purpose:**

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 || 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 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.
### Branch on COP2 False Likely

**Format:**

- **BC2FL**   offset (cc = 0 implied)
- **MIPS32**
- **BC2FL**   cc, offset
- **MIPS32**

**Purpose:**

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:**

if COP2Condition(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 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. 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.

```
I:   condition ← COP2Condition(cc) = 0
     target_offset ← (offset_{15})^{GPRLLEN-{(16+2)}} || offset || 0^2
I+1: if condition then
     PC ← PC + target_offset
else
    NullifyCurrentInstruction()
endif
```

### Table: BC2FL

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP2</td>
<td>BC</td>
<td>cc</td>
<td>nd</td>
<td>tf</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010010</td>
<td>01000</td>
<td>1</td>
<td>0</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **MIPS32**
- **BC2FL**   offset (cc = 0 implied)
- **MIPS32**
- **BC2FL**   cc, offset

Branch on COP2 False Likely (cont.)

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 BC2F instruction instead.
Branch on COP2 True

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP2</td>
<td>BC</td>
<td>cc</td>
<td>nd</td>
<td>tf</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010010</td>
<td>01000</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**Purpose:**  
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 ← (offset15)^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 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.
Branch on COP2 True Likely

BC2TL

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP2</td>
<td>BC</td>
<td>cc</td>
<td>nd</td>
<td>tf</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010010</td>
<td>01000</td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:  
BC2TL offset (cc = 0 implied)  
BC2TL cc, offset

MIPS32
MIPS32

Purpose:  
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: 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.

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

I+1:
  if condition then
    PC ← PC + target_offset
  else
    NullifyCurrentInstruction()
  endif
Branch on COP2 True Likely (cont.)  

BC2TL

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

### Format:
```
BEQ rs, rt, offset
```

### Purpose:
To compare GPRs then do a PC-relative conditional branch

### Description:
\(\text{if GPR}[rs] = \text{GPR}[rt] \text{ then branch}\)

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

### Operation:
```
I: \quad \text{target\_offset} \leftarrow \text{sign\_extend}(\text{offset} \mid \mid 0^2)
    \text{condition} \leftarrow (\text{GPR}[rs] = \text{GPR}[rt])

I+1: \quad \text{if} \ \text{condition} \ \text{then}
       \quad \text{PC} \leftarrow \text{PC} + \text{target\_offset}
       \quad \text{endif}
```

### Exceptions:
None

### 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.

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

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQL</td>
<td>rs</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010100</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

**Description:**
if 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:**

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

**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:**
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 || 02)
   condition ← GPR[rs] ≥ 0
   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

<table>
<thead>
<tr>
<th>Format:</th>
<th>BGEZAL rs, offset</th>
</tr>
</thead>
</table>

Purpose:
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 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 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:

I:

\[
target\_offset \leftarrow \text{sign\_extend}(\text{offset} || 0^2) \\
\text{condition} \leftarrow \text{GPR}[rs] \geq 0^{\text{GPR\_LEN}} \\
\text{GPR}[31] \leftarrow \text{PC} + 8
\]

I+1:

if condition then

\[
\text{PC} \leftarrow \text{PC} + \text{target\_offset}
\]
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.

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

Format: BGEZALL rs, offset

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

Description:
if 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] ≥ 0GPRLEN
    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 $\pm 128$ 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.

Historical Information:

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

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGIMM</td>
<td>rs</td>
<td>BGEZL</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** BGEZL rs, offset

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

**Description:** if 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^GPRLEN

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.
<table>
<thead>
<tr>
<th>Format:</th>
<th>BGTZ $rs$, offset</th>
</tr>
</thead>
</table>

**Purpose:**
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:**

```
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 Zero Likely

Format: \texttt{BGTZL \textit{rs}, \textit{offset}}

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

Description: \texttt{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 \textit{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 \textbf{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{target\_offset} \leftarrow \text{sign\_extend} (\text{offset} \mid| 0^2) \\
& \text{condition} \leftarrow \text{GPR}[\text{rs}] > 0^{\text{GPRLEN}} \\
\text{I+1: } & \text{if condition then} \\
& \quad \text{PC} \leftarrow \text{PC} + \text{target\_offset} \\
& \quad \text{else} \\
& \quad \text{NullifyCurrentInstruction()} \\
& \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

**BLEZ**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLEZ</td>
<td>000110</td>
<td>rs</td>
<td>0</td>
<td>00000</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** BLEZ rs, offset

**MIPS32**

**Purpose:**

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 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 **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
```

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 Less Than or Equal to Zero Likely

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

**MIPS32**

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

**Description:** if \( \text{GPR}[\text{rs}] \leq 0 \) 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 \( \text{rs} \) 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. If the branch is not taken, the instruction in the delay slot is not executed.

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

**Operation:**

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

**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 BLEZ instruction instead.

Historical Information:

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

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

**Purpose:**  
To test a GPR then do a PC-relative conditional branch  

**Description:**  if \texttt{GPR[rs]} < 0 then branch  

An 18-bit signed offset (the 16-bit \texttt{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 \texttt{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 \textbf{UNPREDICTABLE} if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.  

**Operation:**  
\begin{align*}  
\textbf{I}: & \quad \text{target\_offset} \leftarrow \text{sign\_extend}(\text{offset} \mid\mid 0^2) \\
& \quad \text{condition} \leftarrow \text{GPR[rs]} < 0^{\text{GPRLEN}} \\
\textbf{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.
Branch on Less Than Zero and Link

**Format:**  
`BLTZAL rs, offset`

**Purpose:**  
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 | | 0^2)  
|   | condition ← GPR[rs] < 0^GPRLEN  
|   | 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:**  BLTZALL rs, offset

**MIPS32**

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

**Description:** if 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 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 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:**

$I$:  

\[
target\_offset \leftarrow \text{sign\_extend}(\text{offset} \mid\mid 0^2)
\]

\[
\text{condition} \leftarrow \text{GPR[rs]} < 0^\text{GPRLEN}
\]

\[
\text{GPR}[31] \leftarrow \text{PC} + 8
\]

$I+1$:  

if condition then

\[
\text{PC} \leftarrow \text{PC} + \text{target\_offset}
\]

else

\[
\text{NullifyCurrentInstruction()}
\]

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 BLTZAL instruction instead.

Historical Information:

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

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGIMM</td>
<td>rs</td>
<td>BLTZL</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001</td>
<td>5</td>
<td>0010</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** BLTZL rs, offset

**MIPS32**

**Purpose:**
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 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:**
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:  
\[
\text{target_offset} \leftarrow \text{sign_extend}(\text{offset} \mid 0^2)
\]

condition \( \leftarrow \text{GPR}[\text{rs}] < 0^\text{GPRLEN} \)

I+1:  
\[
\text{if condition then}
\]
\[
\text{PC} \leftarrow \text{PC} + \text{target_offset}
\]
\[
\text{else}
\]
\[
\text{NullifyCurrentInstruction()}
\]
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 BLTZ instruction instead.

Historical Information:

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

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

**Purpose:**
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 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:**
```
I:    target_offset ← sign_extend(offset || 0^2)
     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.
Branch on Not Equal Likely

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNEL</td>
<td>rs</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010101</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**MIPS32**

**Purpose:**
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:**

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

**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 BNE instruction instead.

## Historical Information:

In the MIPS I architecture, this instruction signaled a Reserved Instruction Exception.
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>SPECIAL</td>
<td>code</td>
<td></td>
<td>BREAK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td>001101</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** BREAK

**Purpose:**
To cause a Breakpoint exception

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

**Restrictions:**
None

**Operation:**

```
SignalException(Breakpoint)
```

**Exceptions:**
Breakpoint
Floating Point Compare

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

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

**Description:**
FPUCConditionCode(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 cond2..1 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.

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 . 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 3-25 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 3-25 FPU Comparisons Without Special Operand Exceptions

<table>
<thead>
<tr>
<th>Instruction</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>Cond Mnemonic</td>
<td>Name of Predicate and Logically Negated Predicate (Abbreviation)</td>
<td>If Predicate Is True</td>
<td>False [this predicate is always False]</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>True (T)</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>Unordered</td>
<td></td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>Ordered (OR)</td>
<td></td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>Equal</td>
<td></td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>Not Equal (NEQ)</td>
<td></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>Unordered or Equal</td>
<td></td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>Ordered or Greater Than or Less Than (OGL)</td>
<td></td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>Ordered or Less Than</td>
<td></td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>Unordered or Greater Than or Equal (UGE)</td>
<td></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>Ordered or Less Than</td>
<td></td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>Ordered or Greater Than or Equal (OGE)</td>
<td></td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>Ordered or Less Than or Equal</td>
<td></td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>Ordered or Greater Than (OGT)</td>
<td></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>Unordered or Less Than or Equal</td>
<td></td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>Ordered or Greater Than (UGT)</td>
<td></td>
<td>T</td>
<td>F</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>Instruction</th>
<th>Comparison Predicate</th>
<th>Name of Predicate and Logically Negated Predicate (Abbreviation)</th>
<th>Relation Values</th>
<th>If Predicate Is True</th>
<th>Inv Op Excp If QNaN?</th>
<th>Condition Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td></td>
<td>Signaling False [this predicate always False]</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Signaling True (ST)</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>NGLE</td>
<td></td>
<td>Not Greater Than or Less Than or Equal</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Greater Than or Less Than or Equal (GLE)</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>SEQ</td>
<td></td>
<td>Signaling Equal</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Signaling Not Equal (SNE)</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>NGL</td>
<td></td>
<td>Not Greater Than or Less Than</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Greater Than or Less Than (GL)</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>LT</td>
<td></td>
<td>Less Than</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not Less Than (NLT)</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>NGE</td>
<td></td>
<td>Not Greater Than or Equal</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Greater Than or Equal (GE)</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>LE</td>
<td></td>
<td>Less Than Equal</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not Less Than Equal (NLE)</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>NGT</td>
<td></td>
<td>Not Greater Than</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Greater Than (GT)</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

Key: ? = unordered, > = greater than, < = less than, = is equal, T = True, F = False
Floating Point Compare (cont.)

Restrictions:

The fields $fs$ and $ft$ 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 $C$.cond.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode, or if the condition code number is odd.

Operation:

if $SNaN$($ValueFPR(fs, fmt)$) or $SNaN$($ValueFPR(ft, fmt)$) or $QNaN$($ValueFPR(fs, fmt)$) or $QNaN$($ValueFPR(ft, fmt)$) then
  less $\leftarrow$ false
  equal $\leftarrow$ false
  unordered $\leftarrow$ true
endif

if ($SNaN$($ValueFPR(fs,fmt)$) or $SNaN$($ValueFPR(ft,fmt)$)) or ($cond_3$ and ($QNaN$($ValueFPR(fs,fmt)$) or $QNaN$($ValueFPR(ft,fmt)$))) then
  SignalException(InvalidOperation)
endif
else
  less $\leftarrow$ $ValueFPR(fs, fmt) <_{fmt} ValueFPR(ft, fmt)$
  equal $\leftarrow$ $ValueFPR(fs, fmt) =_{fmt} ValueFPR(ft, fmt)$
  unordered $\leftarrow$ false
endif

condition $\leftarrow$ ($cond_2$ and less) or ($cond_1$ and equal)

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 values. 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.

# 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
Format: CACHE op, offset(base)  

Purpose:
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>
<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:

\[
\text{OffsetBit} \leftarrow \log_2(\text{BPT}) \\
\text{IndexBit} \leftarrow \log_2(\frac{\text{CS}}{A}) \\
\text{WayBit} \leftarrow \text{IndexBit} + \text{Ceiling}(\log_2(\text{A})) \\
\text{Way} \leftarrow \text{Addr}_{\text{WayBit}-1..\text{IndexBit}} \\
\text{Index} \leftarrow \text{Addr}_{\text{IndexBit}-1..\text{OffsetBit}}
\]

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.
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.

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.

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.
## 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</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</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 powerup.</td>
<td>Optional</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

<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>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.</td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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.</td>
<td>Required (Instruction Cache Encoding Only), Recommended otherwise</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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></td>
</tr>
<tr>
<td></td>
<td>S, T</td>
<td>Hit Invalidate</td>
<td>Address</td>
<td></td>
<td>Optional</td>
</tr>
</tbody>
</table>
### 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>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></td>
<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.</td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>S, T</td>
<td>Hit Writeback Invalidate / Hit Invalidate</td>
<td>Address</td>
<td>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.</td>
<td>Optional</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.</td>
<td>Recommended</td>
</tr>
<tr>
<td></td>
<td>S, T</td>
<td>Hit Writeback</td>
<td>Address</td>
<td></td>
<td>Optional</td>
</tr>
</tbody>
</table>
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.

<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>
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 CACHE instruction is the target of an invalidate or a writeback invalidate.
If access to Coprocessor 0 is not enabled, a Coprocessor Usable Exception is signaled.

Operation:

\[
\text{vAddr} \leftarrow \text{GPR}[\text{base}] + \text{sign\_extend}(\text{offset}) \\
(\text{pAddr, uncached}) \leftarrow \text{AddressTranslation} (\text{vAddr, DataReadReference}) \\
\text{CacheOp} (\text{op, vAddr, pAddr})
\]

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 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:

```
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

**Format:**

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>CEIL.L</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

- **CEIL.L.S** \( fd, fs \)  
  - **MIPS64, MIPS32 Release 2**

- **CEIL.L.D** \( fd, fs \)  
  - **MIPS64, MIPS32 Release 2**

**Purpose:**

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

**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 rounding toward \(+\infty\) (rounding mode 2). The result is placed in FPR \( fd \).

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{63}\) to \(2^{63}-1\), the result cannot be represented correctly, 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 16 FP registers mode.

**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, Overflow
Floating Point Ceiling Convert to Word Fixed Point

Format:

CEIL.W.S   fd, fs  
CEIL.W.D   fd, fs

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

Description:
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, Overflow
Move Control Word From Floating Point

**CFC1**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>010001</td>
<td>CF</td>
<td>00010</td>
<td>rt</td>
<td>5</td>
<td>fs</td>
<td>5</td>
<td>0</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0000 0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  
CFC1 rt, fs

**MIPS32**

**Purpose:**
To copy a word from an FPU control register to a GPR

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

Copy the 32-bit word from FP (coprocessor 1) control register fs into GPR rt, sign-extending it to 64 bits.

**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.

**Operation:**
```plaintext
if fs = 0 then  
temp ← FIR  
elseif fs = 25 then  
temp ← 024 || FCSR31..25 || FCSR23  
elseif fs = 26 then  
temp ← 014 || FCSR17..12 || 05 || FCSR6..2 || 02  
elseif fs = 28 then  
temp ← 020 || FCSR11.7 || 04 || FCSR24 || FCSR1..0  
elseif fs = 31 then  
temp ← FCSR  
else  
temp ← UNPREDICTABLE  
endif  
GPR[rt] ← sign_extend(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 immediately 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.
Move Control Word From Coprocessor 2

Format: CFC2 rt, rd

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

Purpose:
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, sign-extending it to 64 bits. 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:

\[
\text{temp} \leftarrow \text{CP2CCR[Impl]}
\]
\[
\text{GPR[rt]} \leftarrow \text{sign\_extend(temp)}
\]

Exceptions:
Coprocessor Unusable, Reserved Instruction
Count Leading Ones in Word

Format: CLO rd, rs

Purpose:
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.

If GPR rs does not contain a sign-extended 32-bit value (bits 63..31 equal), then the results of the operation are UNPREDICTABLE.

Operation:

if NotWordValue(GPR[rs]) then
    UNPREDICTABLE
endif
temp ← 32
for i in 31 .. 0
    if GPR[rs]_i = 0 then
        temp ← 31 - i
        break
    endif
endfor
GPR[rd] ← temp

Exceptions:
None
## Coprocessor Operation to Coprocessor 2

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP2</td>
<td>CO</td>
<td>cofun</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010010</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Format:

\[
\text{COP2 func}
\]

### Purpose:

To perform an operation to Coprocessor 2

### Description:

\[
\text{CoprocessorOperation}(2, \text{cofun})
\]

An implementation-dependent operation is performed to Coprocessor 2, with the \text{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:**
  
  \[
  \text{CoprocessorOperation}(2, \text{cofun})
  \]

- **Exceptions:**
  - Coprocessor Unusable
  - Reserved Instruction
**Count Leading Zeros in Word**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPECIAL2</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>CLZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>00000</td>
<td>100000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** CLZ rd, rs

**Purpose**

Count the number of leading zeros in a word

**Description:**

GPR[rd] ← count_leading_zeros GPR[rs]

Bits 31..0 of GPR rs are scanned from most significant to least significant bit. The number of leading zeros is counted and the result is written to GPR rd. If no bits were set in GPR rs, the result written to GPR rt 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.

If GPR rs does not contain a sign-extended 32-bit value (bits 63..31 equal), then the results of the operation are UNPREDICTABLE.

**Operation:**

if NotWordValue(GPR[rs]) then

UNPREDICTABLE
endif

temp ← 32
for i in 31 .. 0
    if GPR[rs]_i = 1 then
        temp ← 31 - i
        break
    endif
endfor
GPR[rd] ← temp

**Exceptions:**

None
Move Control Word to Floating Point

<table>
<thead>
<tr>
<th>Format:</th>
<th>CTC1</th>
<th>rt, fs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>To copy a word from a GPR to an FPU control register</td>
<td></td>
</tr>
<tr>
<td>Description:</td>
<td>FP_Control[fs] ← GPR[rt]</td>
<td></td>
</tr>
<tr>
<td>Copy the low word from GPR rt into the FP (coprocessor 1) control register indicated by fs.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>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 the EPC register contains the address of the CTC1 instruction.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Restrictions:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Operation:

\[
\text{temp} \leftarrow \text{GPR}[rt]_{31..0}
\]

if \(fs = 25\) then /* FCCR */
  if \(\text{temp}_{31..8} \neq 0^{24}\) then
    UNPREDICTABLE
  else
    \text{FCSR} \leftarrow \text{temp}_{7..1} \parallel \text{FCSR}_{24} \parallel \text{temp}_{0} \parallel \text{FCSR}_{22..0}
  endif
endif
elseif \(fs = 26\) then /* FEXR */
  if \(\text{temp}_{22..18} \neq 0\) then
    UNPREDICTABLE
  else
    \text{FCSR} \leftarrow \text{FCSR}_{31..18} \parallel \text{temp}_{17..12} \parallel \text{FCSR}_{11..7} \parallel \text{temp}_{6..2} \parallel \text{FCSR}_{1..0}
  endif
endif
elseif \(fs = 28\) then /* FENR */
  if \(\text{temp}_{22..18} \neq 0\) then
    UNPREDICTABLE
  else
    \text{FCSR} \leftarrow \text{FCSR}_{31..25} \parallel \text{temp}_{2} \parallel \text{FCSR}_{23..12} \parallel \text{temp}_{11..7} \parallel \text{FCSR}_{6..2} \parallel \text{temp}_{1..0}
  endif
endif
elseif \(fs = 31\) then /* FCSR */
  if \(\text{temp}_{22..18} \neq 0\) then
    UNPREDICTABLE
  else
    \text{FCSR} \leftarrow \text{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 undefined 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.
Move Control Word to Coprocessor 2  CTC2

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP2</td>
<td>CT</td>
<td>rt</td>
<td>Impl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010010</td>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:  CTC2 rt, rd  

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

Purpose:
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]31..0
CP2CCR[Impl] ← temp
```

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

CVT.D.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>fs</td>
<td>fd</td>
<td>CVT.D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010001</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:  
CVT.D.S fd, fs  
CVT.D.W fd, fs  
CVT.D.L fd, fs

MIPS32  
MIPS32  
MIPS64, MIPS32 Release 2

Purpose:  
To convert an FP or fixed point value to double FP

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

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 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.D.L, the result of this instruction is UNPREDICTABLE if the processor is executing in 16 FP registers mode.

Operation:  
StoreFPR (fd, D, ConvertFmt(ValueFPR(fs, fmt), 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.S fd, fs
- CVT.L.D fd, fs

**MIPS64, MIPS32 Release 2**
- MIPS64, MIPS32 Release 2

**Purpose:**
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 16 FP registers mode.

**Operation:**
```
StoreFPR (fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```
Floating Point Convert to Long Fixed Point, cont.

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

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

**CVT.PS.S**

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>CVT.PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>10000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>100110</td>
</tr>
</tbody>
</table>

**Format:**  
CVT.PS.S fd, fs, ft

**Purpose:**  
To convert two FP values to a paired single value

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

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 16 FP registers mode.
Floating Point Convert Pair to Paired Single (cont.)

**Operation:**

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

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

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

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>CVT.S</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td>100000</td>
</tr>
</tbody>
</table>

**Format:**

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

**MIPS32**

**MIPS64, MIPS32 Release 2**

**Purpose:**

To convert an FP or fixed point value to single FP

**Description:**

FPR[fd] ← convert_and_round(GPR[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 16 FP registers mode.

**Operation:**

StoreFPR(fd, S, 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 CVT.S.PL

<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>fs</td>
<td>fd</td>
<td>CVT.S.PL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010001</td>
<td>10110</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** CVT.S.PL fd, fs

**MIPS64, MIPS32 Release 2**

**Purpose:**
To convert one half of a paired single FP value to single FP

**Description:**
\[ \text{GPR}[fd] \leftarrow \text{convert\_and\_round}(\text{GPR}[fs]) \]

The lower paired single value in FPR \( fs \), in format \( PS \), 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 \( fd \). This instruction can be used to isolate the lower half of a paired single value.

**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 16 FP registers mode.

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

**Exceptions:**
Coprocessor Usable, Reserved Instruction

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

**CVT.S.PU**

<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>fs</td>
<td>fd</td>
<td>CVT.S.PU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010001</td>
<td>10110</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** CVT.S.PU fd, fs

**Purpose:**
To convert one half of a paired single FP value to single FP

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

The upper paired single value in FPR fs, in format PS, 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. This instruction can be used to isolate the upper half of a paired single value.

**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 in 16 FP registers mode.

**Operation:**

StoreFPR (fd, S, ConvertFmt(ValueFPR(fs, PS), PU, S))

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

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

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>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>fs</td>
<td>fd</td>
<td>CVT.W</td>
<td>010001</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Format:
CVT.W.S fd, fs  
CVT.W.D fd, fs

MIPS32  
MIPS32

Purpose:
To convert an FP value to 32-bit fixed point

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

Exceptions:
Coprocessor Usable, Reserved Instruction

Floating Point Exceptions:
Invalid Operation, Unimplemented Operation, Inexact, Overflow
**Doubleword Add**

**Format:**  
DADD rd, rs, rt  

**MIPS64**

**Purpose:**  
To add 64-bit integers. If overflow occurs, then trap.

**Description:**  
GPR[rd] ← GPR[rs] + GPR[rt]  
The 64-bit doubleword value in GPR rt is added to the 64-bit value in GPR rs to produce a 64-bit result. If the addition results in 64-bit 2’s complement arithmetic overflow, then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 64-bit result is placed into GPR rd.

**Restrictions:**

**Operation:**
```
temp ← (GPR[rs]63||GPR[rs]) + (GPR[rt]63||GPR[rt])
if (temp64 ≠ temp63) then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp63..0
endif
```

**Exceptions:**  
Integer Overflow, Reserved Instruction

**Programming Notes:**  
DADDU performs the same arithmetic operation but does not trap on overflow.
Doubleword Add Immediate

<table>
<thead>
<tr>
<th>DADDI</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>011000</td>
<td>6</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Format: DADDI rt, rs, immediate

Purpose:
To add a constant to a 64-bit integer. If overflow occurs, then trap.

Description: GPR [rt] ← GPR [rs] + immediate

The 16-bit signed immediate is added to the 64-bit value in GPR rs to produce a 64-bit result. If the addition results in 64-bit 2’s complement arithmetic overflow, then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 64-bit result is placed into GPR rt.

Restrictions:

Operation:

```
temp ← (GPR[rs]63||GPR[rs]) + sign_extend(immediate)
if (temp64 ≠ temp63) then
  SignalException(IntegerOverflow)
else
  GPR[rt] ← temp63..0
endif
```

Exceptions:

Integer Overflow, Reserved Instruction

Programming Notes:

DADDIU performs the same arithmetic operation but does not trap on overflow.
Doubleword Add Immediate Unsigned

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DADDIU</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011001</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  DADDIU rt, rs, immediate  

**Purpose:**  
To add a constant to a 64-bit integer

**Description:**  
GPR[rt] ← GPR[rs] + immediate  
The 16-bit signed *immediate* is added to the 64-bit value in GPR *rs* and the 64-bit arithmetic result is placed into GPR *rt*.  
No Integer Overflow exception occurs under any circumstances.

**Restrictions:**

**Operation:**

GPR[rt] ← GPR[rs] + sign_extend(immediate)

**Exceptions:**
Reserved Instruction

**Programming Notes:**

The term “unsigned” in the instruction name is a misnomer; this operation is 64-bit modulo arithmetic that does not trap on overflow. It is appropriate for unsigned arithmetic such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.
### Doubleword Add Unsigned

|   | 31   | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0   |
|---|------|----|----|----|----|----|----|----|----|----|----|----|----|
|   | SPECIAL | rs | rt | rd | 0 |   |    |    |    |    |    |    | DADDU |
|   | 000000  | 5  | 5  | 5  | 0 |    |    |    |    |    |    |    | 101101 |

**Format:** DADDU rd, rs, rt

**Purpose:**
To add 64-bit integers

**Description:**

GPR[rd] ← GPR[rs] + GPR[rt]

The 64-bit doubleword value in GPR rt is added to the 64-bit value in GPR rs and the 64-bit arithmetic result is placed into GPR rd.

No Integer Overflow exception occurs under any circumstances.

**Restrictions:**

**Operation:**

GPR[rd] ← GPR[rs] + GPR[rt]

**Exceptions:**
Reserved Instruction

**Programming Notes:**

The term "unsigned" in the instruction name is a misnomer; this operation is 64-bit modulo arithmetic that does not trap on overflow. It is appropriate for unsigned arithmetic such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.
### Count Leading Ones in Doubleword

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL2</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>DCLO</td>
<td>100101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** DCLO rd, rs

**Purpose:**
To count the number of leading ones in a doubleword

**Description:** GPR[rd] ← count_leading_ones GPR[rs]

The 64-bit word in GPR rs is 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 64 bits were set in GPR rs, the result written to GPR rd is 64.

**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 ← 64
for i in 63..0
    if GPR[rs]_i = 1 then
        temp ← 63 - i
        break
    endif
endfor
GPR[rd] ← temp
```

**Exceptions:**
None
Count Leading Zeros in Doubleword

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL2</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>DCLZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \( \text{DCLZ } rd, rs \)

**Purpose:**
To count the number of leading zeros in a doubleword

**Description:** \( GPR[rd] \leftarrow \text{count\_leading\_zeros } GPR[rs] \)
The 64-bit word in GPR \( rs \) is scanned from most significant to least significant bit. The number of leading zeros is counted and the result is written to GPR \( rd \). If no bits were set in GPR \( rs \), the result written to GPR \( rd \) is 64.

**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:**
```
temp <- 64
for i in 63..0
    if GPR[rs]_i = 0 then
        temp <- 63 - i
        break
    endif
endfor
GPR[rd] <- temp
```

**Exceptions:**
None
Doubleword Divide

### DDIV

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>0</td>
<td>00 0000 0000</td>
<td>DDIV</td>
<td>011110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** DDIV  rs, rt  

**Purpose:**
To divide 64-bit signed integers

**Description:** \((LO, HI) \leftarrow GPR[rs] \div GPR[rt]\)

The 64-bit doubleword in GPR \(rs\) is divided by the 64-bit doubleword in GPR \(rt\), treating both operands as signed values. The 64-bit quotient is placed into special register \(LO\) and the 64-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:**

\[
LO \leftarrow GPR[rs] \div GPR[rt] \\
HI \leftarrow GPR[rs] \mod GPR[rt]
\]

**Exceptions:**
Reserved Instruction

**Programming Notes:**
See “Programming Notes” for the DIV instruction.

**Historical Perspective:**
In 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.
### Doubleword Divide Unsigned

**DDIVU**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>0</td>
<td>000000 0000</td>
<td>DDIVU</td>
<td>011111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>10</th>
<th>6</th>
</tr>
</thead>
</table>

**Format:** DDIVU rs, rt

**Purpose:**
To divide 64-bit unsigned integers

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

The 64-bit doubleword in GPR rs is divided by the 64-bit doubleword in GPR rt, treating both operands as unsigned values. The 64-bit quotient is placed into special register LO and the 64-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 undefined.

**Operation:**

\[
q \leftarrow (0 \mid\mid \text{GPR}[rs]) \div (0 \mid\mid \text{GPR}[rt]) \\
r \leftarrow (0 \mid\mid \text{GPR}[rs]) \mod (0 \mid\mid \text{GPR}[rt]) \\
\text{LO} \leftarrow q_{63\ldots0} \\
\text{HI} \leftarrow r_{63\ldots0}
\]

**Exceptions:**
Reserved Instruction

**Programming Notes:**
See “Programming Notes” for the DIV instruction.

**Historical Perspective:**
In 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.
**Debug Exception Return**

<table>
<thead>
<tr>
<th>COP0</th>
<th>CO</th>
<th>DEPC</th>
<th>DERET</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>1</td>
<td>000 0000 0000 0000 0000</td>
<td>011111</td>
</tr>
</tbody>
</table>

**Format:** DERET

**Purpose:**

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.
Debug Exception Return (cont.)

Operation:

\[
\begin{align*}
\text{DebugDM} & \leftarrow 0 \\
\text{DebugIEXI} & \leftarrow 0 \\
\text{if } \text{IsMIPS16Implemented()} \text{ then} \\
& \quad \text{PC} \leftarrow \text{DEPC}_{63..1} \parallel 0 \\
& \quad \text{ISAMode} \leftarrow \text{DEPC}_0 \\
\text{else} \\
& \quad \text{PC} \leftarrow \text{DEPC} \\
\text{endif} \\
& \quad \text{ClearHazards}()
\end{align*}
\]

Exceptions:

Coprocessor Unusable Exception
Reserved Instruction Exception
Doubleword Extract Bit Field

Format: \texttt{dext rt, rs, pos, size}  

Purpose:
To extract a bit field from GPR \textit{rs} and store it right-justified into GPR \textit{rt}.

Description:
\texttt{GPR[rt] \leftarrow ExtractField(GPR[rs], msbd, lsb)}

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

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

For this instruction, the values of \textit{pos} and \textit{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 63
\end{align*}

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

Three instructions are required to access any legal bit field within the doubleword, as a function of the \textit{msb} (as derived from \textit{msbd} and \textit{lsb}) and \textit{lsb} of the field (which implies restrictions on \textit{pos} and \textit{size}), as follows:
**Doubleword Extract Bit Field, cont.**

<table>
<thead>
<tr>
<th>msbd</th>
<th>lsb</th>
<th>msb</th>
<th>pos</th>
<th>size</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ≤ msbd &lt; 32</td>
<td>0 ≤ lsb &lt; 32</td>
<td>0 ≤ msb &lt; 63</td>
<td>0 ≤ pos &lt; 32</td>
<td>1 ≤ size ≤ 32</td>
<td>DEXT</td>
<td>The field is 32 bits or less and starts in the right-most word of the doubleword</td>
</tr>
<tr>
<td>0 ≤ msbd &lt; 32</td>
<td>32 ≤ lsb &lt; 64</td>
<td>32 ≤ msb &lt; 64</td>
<td>32 ≤ pos &lt; 64</td>
<td>1 ≤ size ≤ 32</td>
<td>DEXTU</td>
<td>The field is 32 bits or less and starts in the left-most word of the doubleword</td>
</tr>
<tr>
<td>32 ≤ msbd &lt; 64</td>
<td>0 ≤ lsb &lt; 32</td>
<td>32 ≤ msb &lt; 64</td>
<td>0 ≤ pos &lt; 32</td>
<td>32 &lt; size ≤ 64</td>
<td>DEXTM</td>
<td>The field is larger than 32 bits and starts in the right-most word of the doubleword</td>
</tr>
</tbody>
</table>

**Restrictions:**

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

Because of the limits on the values of `msbd` and `lsb`, there is no **UNPREDICTABLE** case for this instruction.

**Operation:**

\[
GPR[rt] \leftarrow 0^{63-(msbd+1)} \parallel GPR[rs]_{msbd+1} = lsb
\]

**Exceptions:**

Reserved Instruction

**Programming Notes**

The assembler will accept any value of `pos` and `size` that satisfies the relationship \(0 < pos + size \leq 64\) and emit DEXT, DEXTM, or DEXTU as appropriate to the values. Programmers should always specify the DEXT mnemonic and let the assembler select the instruction to use.
Doubleword Extract Bit Field Middle

**Format:** \( \text{dextm\ rt, rs, pos, size} \)

**Purpose:**
To extract a bit field from GPR \( rs \) and store it right-justified into GPR \( rt \).

**Description:**
\[
\text{GPR}[\text{rt}] \leftarrow \text{ExtractField}(\text{GPR}[\text{rs}], \text{msbd}, \text{lsb})
\]

The bit field starting at bit \( \text{pos} \) and extending for \( \text{size} \) bits is extracted from GPR \( rs \) and stored zero-extended and right-justified in GPR \( rt \). The assembly language arguments \( \text{pos} \) and \( \text{size} \) are converted by the assembler to the instruction fields \( \text{msbdminus32} \) (the most significant bit of the destination field in GPR \( rt \), minus 32), in instruction bits 15..11, and \( lsb \) (least significant bit of the source field in GPR \( rs \)), in instruction bits 10..6, as follows:

\[
\begin{align*}
\text{msbdminus32} & \leftarrow \text{size}-1-32 \\
\text{lsb} & \leftarrow \text{pos} \\
\text{msbd} & \leftarrow \text{msbdminus32} + 32 \\
\text{msb} & \leftarrow \text{lsb}+\text{msbd}
\end{align*}
\]

For this instruction, the values of \( \text{pos} \) and \( \text{size} \) must satisfy all of the following relations:

\[
\begin{align*}
0 & \leq \text{pos} < 32 \\
32 & < \text{size} \leq 64 \\
32 & < \text{pos}+\text{size} \leq 64
\end{align*}
\]

**Figure 3-4** shows the symbolic operation of the instruction.

**Figure 3-4 Operation of the DEXTM Instruction**

Three instructions are required to access any legal bit field within the doubleword, as a function of the \( \text{msb} \) (as derived from \( \text{msbd} \) and \( \text{lsb} \)) and \( \text{lsb} \) of the field (which implies restrictions on \( \text{pos} \) and \( \text{size} \)), as follows:
Doubleword Extract Bit Field Middle, cont.

<table>
<thead>
<tr>
<th>msbd</th>
<th>lsb</th>
<th>msb</th>
<th>pos</th>
<th>size</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ≤ msbd &lt; 32</td>
<td>0 ≤ lsb &lt; 32</td>
<td>0 ≤ msb &lt; 63</td>
<td>0 ≤ pos &lt; 32</td>
<td>1 ≤ size ≤ 32</td>
<td>DEXT</td>
<td>The field is 32 bits or less and starts in the right-most word of the doubleword</td>
</tr>
<tr>
<td>0 ≤ msbd &lt; 32</td>
<td>32 ≤ lsb &lt; 64</td>
<td>32 ≤ msb &lt; 64</td>
<td>32 ≤ pos &lt; 64</td>
<td>1 ≤ size ≤ 32</td>
<td>DEXTU</td>
<td>The field is 32 bits or less and starts in the left-most word of the doubleword</td>
</tr>
<tr>
<td>32 ≤ msbd &lt; 64</td>
<td>0 ≤ lsb &lt; 32</td>
<td>32 ≤ msb &lt; 64</td>
<td>0 ≤ pos &lt; 32</td>
<td>32 ≤ size ≤ 64</td>
<td>DEXTM</td>
<td>The field is larger than 32 bits and starts in the right-most word of the doubleword</td>
</tr>
</tbody>
</table>

Restrictions:

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

The operation is **UNPREDICTABLE** if \((lsb + msbd + 1) > 64\).

Operation:

\[
\begin{align*}
\text{msbd} &\leftarrow \text{msbdminus32} + 32 \\
&\text{if } ((\text{lsb} + \text{msbd} + 1) > 64) \text{ then} \\
&\text{UNPREDICTABLE} \\
&\text{endif} \\
\text{GPR[rt]} &\leftarrow 0^{63-(\text{msbd}+1)} || \text{GPR[rs]}_{\text{msbd}+\text{lsb}..\text{pos}}
\end{align*}
\]

Exceptions:

Reserved Instruction

Programming Notes

The assembler will accept any value of \(pos\) and \(size\) that satisfies the relationship \(0 < pos+size \leq 64\) and emit DEXT, DEXTM, or DEXTU as appropriate to the values. Programmers should always specify the DEXT mnemonic and let the assembler select the instruction to use.
Doubleword Extract Bit Field Upper

**Format:** \texttt{dextu rt, rs, pos, size}

**Purpose:**
To extract a bit field from GPR \(rs\) and store it right-justified into GPR \(rt\).

**Description:** \(\text{GPR}[rt] \leftarrow \text{ExtractField(}\text{GPR}[rs], \text{msbd}, \text{lsb})\)

The bit field starting at bit \(\text{pos}\) and extending for \(\text{size}\) bits is extracted from GPR \(rs\) and stored zero-extended and right-justified in GPR \(rt\). The assembly language arguments \(\text{pos}\) and \(\text{size}\) are converted by the assembler to the instruction fields \(\text{msbd}\) (the most significant bit of the destination field in GPR \(rt\)), in instruction bits 15..11, and \(\text{lsbminus32}\) (least significant bit of the source field in GPR \(rs\), minus32), in instruction bits 10..6, as follows:

\[
\begin{align*}
\text{msbd} & \leftarrow \text{size}-1 \\
\text{lsbminus32} & \leftarrow \text{pos}-32 \\
\text{lsb} & \leftarrow \text{lsbminus32} + 32 \\
\text{msb} & \leftarrow \text{lsb}+\text{msbd}
\end{align*}
\]

For this instruction, the values of \(\text{pos}\) and \(\text{size}\) must satisfy all of the following relations:

\[
\begin{align*}
32 & \leq \text{pos} < 64 \\
0 & < \text{size} \leq 32 \\
32 & < \text{pos}+\text{size} \leq 64
\end{align*}
\]

Figure 3-5 shows the symbolic operation of the instruction.

![Figure 3-5 Operation of the DEXTU Instruction](image)

Three instructions are required to access any legal bit field within the doubleword, as a function of the \(\text{msb}\) (as derived from \(\text{msbd}\) and \(\text{lsb}\)) and \(\text{lsb}\) of the field (which implies restrictions on \(\text{pos}\) and \(\text{size}\)), as follows:
**Restrictions:**

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

The operation is **UNPREDICTABLE** if \((\text{lsb} + \text{msbd} + 1) > 64\).

**Operation:**

\[
\text{lsb} \leftarrow \text{lsbminus32} + 32 \\
\text{if } ((\text{lsb} + \text{msbd} + 1) > 64) \text{ then} \\
\quad \text{UNPREDICTABLE} \\
\text{endif} \\
\text{GPR}[rt] \leftarrow 0^{63-(\text{msbd}+1)} \parallel \text{GPR}[rs]_{\text{msbd}+\text{lsb}..\text{pos}}
\]

**Exceptions:**

Reserved Instruction

**Programming Notes**

The assembler will accept any value of \(\text{pos}\) and \(\text{size}\) that satisfies the relationship \(0 < \text{pos} + \text{size} \leq 64\) and emit DEXT, DEXTM, or DEXTU as appropriate to the values. Programmers should always specify the DEXT mnemonic and let the assembler select the instruction to use.

<table>
<thead>
<tr>
<th>(\text{msbd})</th>
<th>(\text{lsb})</th>
<th>(\text{msb})</th>
<th>(\text{pos})</th>
<th>(\text{size})</th>
<th>\textbf{Instruction}</th>
<th>\textbf{Comment}</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0 \leq \text{msbd} &lt; 32)</td>
<td>(0 \leq \text{lsb} &lt; 32)</td>
<td>(0 \leq \text{msb} &lt; 63)</td>
<td>(0 \leq \text{pos} &lt; 32)</td>
<td>(1 \leq \text{size} \leq 32)</td>
<td>DEXT</td>
<td>The field is 32 bits or less and starts in the right-most word of the doubleword</td>
</tr>
<tr>
<td>(0 \leq \text{msbd} &lt; 32)</td>
<td>(32 \leq \text{lsb} &lt; 64)</td>
<td>(32 \leq \text{msb} &lt; 64)</td>
<td>(32 \leq \text{pos} &lt; 64)</td>
<td>(1 \leq \text{size} \leq 32)</td>
<td>DEXTU</td>
<td>The field is 32 bits or less and starts in the left-most word of the doubleword</td>
</tr>
<tr>
<td>(32 \leq \text{msbd} &lt; 64)</td>
<td>(0 \leq \text{lsb} &lt; 32)</td>
<td>(32 \leq \text{msb} &lt; 64)</td>
<td>(0 \leq \text{pos} &lt; 32)</td>
<td>(32 &lt; \text{size} \leq 64)</td>
<td>DEXTM</td>
<td>The field is larger than 32 bits and starts in the right-most word of the doubleword</td>
</tr>
</tbody>
</table>
Disable Interrupts

**Purpose:**
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 sign-extended and 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.

\[
\text{data} \leftarrow \text{Status} \\
\text{GPR}[rt] \leftarrow \text{sign\_extend(data)} \\
\text{Status}_{IE} \leftarrow 0
\]
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 can not 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.
**Doubleword Insert Bit Field**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPECIAL3</td>
<td>rs</td>
<td>rt</td>
<td>msb (pos+size-1)</td>
<td>lsb (pos)</td>
<td>DINS</td>
<td>000111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \( \text{dins } rt, rs, \text{pos}, \text{size} \)

**MIPS64 Release 2**

**Purpose:**

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

**Description:**

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

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 \( \text{msb} \) (the most significant bit of the field), in instruction bits 15..11, and \( \text{lsb} \) (least significant bit of the field), in instruction bits 10..6, as follows:

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

For this instruction, 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 3-6** shows the symbolic operation of the instruction.

---

**Figure 3-6 Operation of the DINS Instruction**

---

Copyright © 2001-2003,2005 MIPS Technologies Inc. All rights reserved.
Three instructions are required to access any legal bit field within the doubleword, as a function of the \textit{msb} and \textit{lsb} of the field (which implies restrictions on \textit{pos} and \textit{size}), as follows:

<table>
<thead>
<tr>
<th>(msb)</th>
<th>(lsb)</th>
<th>(pos)</th>
<th>(size)</th>
<th>\textbf{Instruction}</th>
<th>\textbf{Comment}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (\leq ) (msb) &lt; 32</td>
<td>0 (\leq ) (lsb) &lt; 32</td>
<td>0 (\leq ) (pos) &lt; 32</td>
<td>1 (\leq ) (size) &lt; 32</td>
<td>DINS</td>
<td>The field is entirely contained in the right-most word of the doubleword</td>
</tr>
<tr>
<td>32 (\leq ) (msb) &lt; 64</td>
<td>0 (\leq ) (lsb) &lt; 32</td>
<td>0 (\leq ) (pos) &lt; 32</td>
<td>2 (\leq ) (size) (\leq) 64</td>
<td>DINSM</td>
<td>The field straddles the words of the doubleword</td>
</tr>
<tr>
<td>32 (\leq ) (msb) &lt; 64</td>
<td>32 (\leq ) (lsb) &lt; 64</td>
<td>32 (\leq ) (pos) &lt; 64</td>
<td>1 (\leq ) (size) &lt; 32</td>
<td>DINSU</td>
<td>The field is entirely contained in the left-most word of the doubleword</td>
</tr>
</tbody>
</table>

\textbf{Restrictions:}

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

The operation is \textbf{UNPREDICTABLE} if \(lsb > msb\).

\textbf{Operation:}

\begin{verbatim}
if (lsb > msb) then
  UNPREDICTABLE
endif
GPR[rt] \(\leftarrow\) GPR[rt]_{63..msb+1} \(\parallel\) GPR[rs]_{msb-lsb..0} \(\parallel\) GPR[rt]_{lsb-1..0}
\end{verbatim}

\textbf{Exceptions:}

Reserved Instruction

\textbf{Programming Notes}

The assembler will accept any value of \textit{pos} and \textit{size} that satisfies the relationship \(0 < \textit{pos}+\textit{size} \leq 64\) and emit DINS, DINSM, or DINSU as appropriate to the values. Programmers should always specify the DINS mnemonic and let the assembler select the instruction to use.
DINSM

Format: dinsm rt, rs, pos, size

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

Description: GPR[rt] ← InsertField(GPR[rt], GPR[rs], msb, lsb)

The right-most size bits from GPR rs are inserted 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 msbminus32 (the most significant bit of the field, minus 32), in instruction bits 15..11, and lsb (least significant bit of the field), in instruction bits 10..6, as follows:

\[
\begin{align*}
msbminus32 & \leftarrow pos+size-1-32 \\
lsb & \leftarrow pos \\
msb & \leftarrow msbminus32 + 32
\end{align*}
\]

For this instruction, the values of pos and size must satisfy all of the following relations:

\[
\begin{align*}
0 & \leq pos < 32 \\
2 & \leq size \leq 64 \\
32 & < pos+size \leq 64
\end{align*}
\]

Figure 3-7 shows the symbolic operation of the instruction.

---

**Figure 3-7 Operation of the DINSM Instruction**
Three instructions are required to access any legal bit field within the doubleword, as a function of the \textit{msb} and \textit{lsb} of the field (which implies restrictions on \textit{pos} and \textit{size}), as follows:

<table>
<thead>
<tr>
<th>\textit{msb}</th>
<th>\textit{lsb}</th>
<th>\textit{pos}</th>
<th>\textit{size}</th>
<th>\textit{Instruction}</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ≤ \textit{msb} &lt; 32</td>
<td>0 ≤ \textit{lsb} &lt; 32</td>
<td>0 ≤ \textit{pos} &lt; 32</td>
<td>1 ≤ \textit{size} ≤ 32</td>
<td>DINS</td>
<td>The field is entirely contained in the right-most word of the doubleword</td>
</tr>
<tr>
<td>32 ≤ \textit{msb} &lt; 64</td>
<td>0 ≤ \textit{lsb} &lt; 32</td>
<td>0 ≤ \textit{pos} &lt; 32</td>
<td>2 ≤ \textit{size} ≤ 64</td>
<td>DINSM</td>
<td>The field straddles the words of the doubleword</td>
</tr>
<tr>
<td>32 ≤ \textit{msb} &lt; 64</td>
<td>32 ≤ \textit{lsb} &lt; 64</td>
<td>32 ≤ \textit{pos} &lt; 64</td>
<td>1 ≤ \textit{size} ≤ 32</td>
<td>DINSU</td>
<td>The field is entirely contained in the left-most word of the doubleword</td>
</tr>
</tbody>
</table>

\textbf{Restrictions:}

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

Because of the instruction format, \textit{lsb} can never be greater than \textit{msb}, so there is no \textbf{UNPREDICATABLE} case for this instruction.

\textbf{Operation:}

\[ \text{msb} \leftarrow \text{msbminuss32} + 32 \]

\[ \text{GPR}[rt] \leftarrow \text{GPR}[rt]_{63..\text{msb}+1} \parallel \text{GPR}[rs]_{\text{msb}-\text{lsb}..0} \parallel \text{GPR}[rt]_{\text{lsb}-1..0} \]

\textbf{Exceptions:}

Reserved Instruction

\textbf{Programming Notes}

The assembler will accept any value of \textit{pos} and \textit{size} that satisfies the relationship $0 < \text{pos} + \text{size} \leq 64$ and emit DINS, DINSM, or DINSU as appropriate to the values. Programmers should always specify the DINS mnemonic and let the assembler select the instruction to use.
Doubleword Insert Bit Field Upper

**Format:** `dinsu rt, rs, pos, size`  

**Purpose:**
To merge a right-justified bit field from GPR `rs` into a specified position in GPR `rt`.

**Description:**
GPR[rt] ← InsertField(GPR[rt], GPR[rs], msb, lsb)

The right-most `size` bits from GPR `rs` are inserted 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 `msbminus32` (the most significant bit of the field, minus 32), in instruction bits 15..11, and `lsbminus32` (least significant bit of the field, minus 32), in instruction bits 10..6, as follows:

- `msbminus32 ← pos+size-1-32`
- `lsbminus32 ← pos-32`
- `msb ← msbminus32 + 32`
- `lsb ← lsbminus32 + 32`

For this instruction, the values of `pos` and `size` must satisfy all of the following relations:

- `32 ≤ pos < 64`
- `1 ≤ size ≤ 32`
- `32 < pos+size ≤ 64`

Figure 3-8 shows the symbolic operation of the instruction.

Three instructions are required to access any legal bit field within the doubleword, as a function of the `msb` and `lsb` of the field (which implies restrictions on `pos` and `size`), as follows:
Doubleword Insert Bit Field Upper, cont.

<table>
<thead>
<tr>
<th>$msb$</th>
<th>$lsb$</th>
<th>$pos$</th>
<th>$size$</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 \leq msb &lt; 32$</td>
<td>$0 \leq lsb &lt; 32$</td>
<td>$0 \leq pos &lt; 32$</td>
<td>$1 \leq size \leq 32$</td>
<td>DINS</td>
<td>The field is entirely contained in the right-most word of the doubleword</td>
</tr>
<tr>
<td>$32 \leq msb &lt; 64$</td>
<td>$0 \leq lsb &lt; 32$</td>
<td>$0 \leq pos &lt; 32$</td>
<td>$2 \leq size \leq 64$</td>
<td>DINSM</td>
<td>The field straddles the words of the doubleword</td>
</tr>
<tr>
<td>$32 \leq msb &lt; 64$</td>
<td>$32 \leq lsb &lt; 64$</td>
<td>$32 \leq pos &lt; 64$</td>
<td>$1 \leq size \leq 32$</td>
<td>DINSU</td>
<td>The field is entirely contained in the left-most word of the doubleword</td>
</tr>
</tbody>
</table>

**Restrictions:**

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

The operation is **UNPREDICTABLE** if $lsb > msb$.

**Operation:**

\[
\begin{align*}
\text{lsb} & \leftarrow \text{lsbminus32} + 32 \\
\text{msb} & \leftarrow \text{msbminus32} + 32 \\
\text{if} \ (\text{lsb} > \text{msb}) \text{ then} & \\
\text{UNPREDICTABLE} & \\
\text{endif} & \\
\text{GPR}[rt] & \leftarrow \text{GPR}[rt]_{63..msb+1} \parallel \text{GPR[rs]}_{msb-1..0} \parallel \text{GPR[rt]}_{lsb-1..0}
\end{align*}
\]

**Exceptions:**

Reserved Instruction

**Programming Notes**

The assembler will accept any value of $pos$ and $size$ that satisfies the relationship $0 < pos+size \leq 64$ and emit DINS, DINSM, or DINSU as appropriate to the values. Programmers should always specify the DINS mnemonic and let the assembler select the instruction to use.
DIV

**Format:**  DIV rs, rt

**Purpose:**
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 sign-extended and placed into special register LO and the 32-bit remainder is sign-extended and placed into special register HI.
No arithmetic exception occurs under any circumstances.

**Restrictions:**
If either GPR rt or GPR rs does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.
If the divisor in GPR rt is zero, the arithmetic result value is **UNPREDICTABLE**.

**Operation:**
```c
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then
    UNPREDICTABLE
endif
q ← GPR[rs]31..0 div GPR[rt]31..0
LO ← sign_extend(q31..0)
result ← GPR[rs]31..0 mod GPR[rt]31..0
HI ← sign_extend(result)
```

**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-specified signal handler. C does not expect overflow to cause any exceptional condition. If the C compiler uses a divide instruction, it also emits code to test for a zero divisor and execute a BREAK instruction to inform the operating system if a zero is detected.

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 1 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.S fd, fs, ft
DIV.D fd, fs, ft

MIPS32

Purpose:
To divide FP values

Description:
FPR[fd] ← FPR[fs] / 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 UNPREDICABLE.
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:
StoreFPR (fd, fmt, ValueFPR(fs, fmt) / ValueFPR(ft, fmt))

Exceptions:
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:
Inexact, Invalid Operation, Unimplemented Operation, Division-by-zero, Overflow, Underflow
**Divide Unsigned Word (DIVU)**

**Format:**  
DIVU rs, rt  

**MIPS32**

**Purpose:**  
To divide a 32-bit unsigned integers

**Description:**  
(HI, LO) ← GPR[r] / GPR[r]  
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 sign-extended and placed into special register LO and the 32-bit remainder is sign-extended and placed into special register HI.

No arithmetic exception occurs under any circumstances.

**Restrictions:**  
If either GPR rt or GPR rs does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

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

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

**Exceptions:**  
None

**Programming Notes:**  
See “Programming Notes” for the DIV instruction.

**Historical Perspective:**  
In MIPS 1 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.
Doubleword Move from Coprocessor 0

**Format:**

- DMFC0 rt, rd
- DMFC0 rt, rd, sel

**MIPS64**

**Purpose:**
To move the contents of a coprocessor 0 register to a general purpose register (GPR).

**Description:**

\[ \text{GPR}[rt] \leftarrow \text{CPR}[0,rd,sel] \]

The contents of the coprocessor 0 register are loaded into GPR \( rt \). Note that not all coprocessor 0 registers support the \( sel \) field. In those instances, the \( sel \) field must be zero.

**Restrictions:**

The results are **UNPREDICTABLE** if coprocessor 0 does not contain a register as specified by \( rd \) and \( sel \), or if the coprocessor 0 register specified by \( rd \) and \( sel \) is a 32-bit register.

**Operation:**

\[ \text{datadoubleword} \leftarrow \text{CPR}[0,rd,sel] \]
\[ \text{GPR}[rt] \leftarrow \text{datadoubleword} \]

**Exceptions:**

- Coprocessor Unusable
- Reserved Instruction
Doubleword Move from Floating Point

DMFC1

Format: DMFC1 rt,fs

Purpose:
To move a doubleword from an FPR to a GPR.

Description: GPR[rt] ← FPR[fs]
The contents of FPR fs are loaded into GPR rt.

Restrictions:

Operation:

datadoubleword ← ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)
GPR[rt] ← datadoubleword

Exceptions:
Coprocessor Unusable
Reserved Instruction

Historical Information:
For MIPS III, the contents of GPR rt are undefined for the instruction immediately following DMFC1.
Doubleword Move from Coprocessor 2

| Format: | DMFC2 rt, rd | MIPS64  
          | DMFC2 rt, rd, sel | MIPS64 |

The syntax shown above is an example using DMFC1 as a model. The specific syntax is implementation dependent.

**Purpose:**
To move a doubleword from a coprocessor 2 register to a GPR.

**Description:**
GPR[rt] ← CP2CPR[Impl]

The contents of the coprocessor 2 register denoted by the Impl field is loaded into GPR 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, or if the coprocessor 2 register specified by rd and sel is a 32-bit register.

**Operation:**

datadoubleword ← CP2CPR[Impl]
GPR[rt] ← datadoubleword

**Exceptions:**
Coprocessor Unusable
Reserved Instruction
Doubleword Move to Coprocessor 0

**Format:**
DMTC0 rt, rd
DMTC0 rt, rd, sel

**Purpose:**
To move a doubleword from a GPR to a coprocessor 0 register.

**Description:**
CPR[0,rd,sel] ← GPR[rt]
The contents of GPR rt are loaded into the coprocessor 0 register specified in the rd and sel fields. Note that not all coprocessor 0 registers support the sel field. In those instances, the sel field must be zero.

**Restrictions:**
The results are **UNPREDICTABLE** if coprocessor 0 does not contain a register as specified by rd and sel, or if the coprocessor 0 register specified by rd and sel is a 32-bit register.

**Operation:**

datadoubleword ← GPR[rt]
CPR[0,rd,sel] ← datadoubleword

**Exceptions:**
Coprocessor Unusable
Reserved Instruction
Doubleword Move to Floating Point  DMTC1

Format:  DMTC1 rt, fs

Purpose:
To copy a doubleword from a GPR to an FPR

Description:  FPR[fs] ← GPR[rt]
The doubleword contents of GPR rt are placed into FPR fs.

Restrictions:

Operation:

datadoubleword ← GPR[rt]
StoreFPFR(fs, UNINTERPRETED_DOUBLEWORD, datadoubleword)

Exceptions:
Coprocessor Usable
Reserved Instruction

Historical Information:
For MIPS III, the contents of FPR fs are undefined for the instruction immediately following DMTC1.
Doubleword Move to Coprocessor 2

DMTC2

<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>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP2</td>
<td>DMT</td>
<td>rt</td>
<td>Impl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010010</td>
<td>00101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**
- DMTC2 rt, rd
- DMTC2 rt, rd, sel

The syntax shown above is an example using DMTC1 as a model. The specific syntax is implementation dependent.

**Purpose:**
To move a doubleword from a GPR to a coprocessor 2 register.

**Description:**
\[\text{CPR}[2, \text{rd, sel}] \leftarrow \text{GPR}[\text{rt}]\]

The contents GPR \(rt\) are loaded into the coprocessor 2 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 the coprocessor 2 register specified by \(rd\) and \(sel\) is a 32-bit register.

**Operation:**
- \(\text{datadoubleword} \leftarrow \text{GPR}[\text{rt}]\)
- \(\text{CP2CPR}[\text{Impl}] \leftarrow \text{datadoubleword}\)

**Exceptions:**
- Coprocessor Usable
- Reserved Instruction
Doubleword Multiply

<table>
<thead>
<tr>
<th>Format:</th>
<th>DMULT rs, rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>To multiply 64-bit signed integers</td>
</tr>
<tr>
<td>Description:</td>
<td>(LO, HI) ← GPR[rs] × GPR[rt]</td>
</tr>
</tbody>
</table>

The 64-bit doubleword value in GPR rt is multiplied by the 64-bit value in GPR rs, treating both operands as signed values, to produce a 128-bit result. The low-order 64-bit doubleword of the result is placed into special register LO, and the high-order 64-bit doubleword is placed into special register HI.

No arithmetic exception occurs under any circumstances.

Restrictions:

Operation:

```plaintext
prod ← GPR[rs] × GPR[rt]
LO ← prod63..0
HI ← prod127..64
```

Exceptions:

Reserved Instruction

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.

Historical Perspective:

In 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 all subsequent levels of the architecture.
**Doubleword Multiply Unsigned**

<table>
<thead>
<tr>
<th>Format:</th>
<th>DMULTU rs, rt</th>
</tr>
</thead>
</table>

**Purpose:**
To multiply 64-bit unsigned integers

**Description:** \((LO, HI) \leftarrow GPR[rs] \times GPR[rt]\)

The 64-bit doubleword value in GPR \(rt\) is multiplied by the 64-bit value in GPR \(rs\), treating both operands as unsigned values, to produce a 128-bit result. The low-order 64-bit doubleword of the result is placed into special register \(LO\), and the high-order 64-bit doubleword is placed into special register \(HI\). No arithmetic exception occurs under any circumstances.

**Restrictions:**

**Operation:**
\[
\text{prod} \leftarrow (0|GPR[rs]) \times (0|GPR[rt])
\]
\[
LO \leftarrow \text{prod}_{63..0}
\]
\[
HI \leftarrow \text{prod}_{127..64}
\]

**Exceptions:**
Reserved Instruction

**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.

**Historical Perspective:**
In 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 all subsequent levels of the architecture.
### Doubleword Rotate Right

**Format:** DROTR \( rd, \) \( rt, \) \( sa \)

**Purpose:**
To execute a logical right-rotate of a doubleword by a fixed amount—0 to 31 bits

**Description:**
GPR\( [rd] \leftarrow \) GPR\( [rt] \leftrightarrow \) (right) \( sa \)

The doubleword contents of GPR \( rt \) are rotated right; the result is placed in GPR \( rd \). The bit-rotate amount in the range 0 to 31 is specified by \( sa \).

**Restrictions:**

**Operation:**

\[
\begin{align*}
\text{s} &\leftarrow 0 \parallel \text{sa} \\
\text{GPR}[rd] &\leftarrow \text{GPR}[rt]_{s-1..0} \parallel \text{GPR}[rt]_{63..s}
\end{align*}
\]

**Exceptions:**

Reserved Instruction

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>0000</td>
<td>R</td>
<td>1</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>DSRL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Doubleword Rotate Right Plus 32

**Format:**  DROTR32  rd, rt, sa

**Purpose:**
To execute a logical right-rotate of a doubleword by a fixed amount—32 to 63 bits

**Description:**
GPR[rd] ← GPR[rt] ↔(right) (saminus32+32)
The 64-bit doubleword contents of GPR rt are rotated right; the result is placed in GPR rd. The bit-rotate amount in the range 32 to 63 is specified by saminus32+32.

**Restrictions:**

**Operation:**
\[
\begin{align*}
  s &\leftarrow 1 \parallel sa \quad /* 32+saminus32 */ \\
  \text{GPR}[rd] &\leftarrow \text{GPR}[rt]_{s-1..0} \parallel \text{GPR}[rt]_{63..s}
\end{align*}
\]

**Exceptions:**
Reserved Instruction
Doubleword Rotate Right Variable

**DROTRV**

<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>7</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>0000</td>
<td>R</td>
<td>DSRLV</td>
<td>010110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** DROTRV rd, rt, rs

**MIPS64 Release 2**

**Purpose:**
To execute a logical right-rotate of a doubleword by a variable number of bits

**Description:**
GPR[rd] ← GPR[rt] ↔ (right) GPR[rs]

The 64-bit doubleword contents of GPR rt are rotated right; the result is placed in GPR rd. The bit-rotate amount in the range 0 to 63 is specified by the low-order 6 bits in GPR rs.

**Restrictions:**

**Operation:**

\[
\begin{align*}
    s & \leftarrow \text{GPR[rs]}_{5..0} \\
    \text{GPR[rd]} & \leftarrow \text{GPR[rt]}_{s-1..0} \parallel \text{GPR[rt]}_{63..s}
\end{align*}
\]

**Exceptions:**

Reserved Instruction
Doubleword Swap Bytes Within Halfwords  

| Format:  | dsbh rd, rt |
| Purpose: | To swap the bytes within each halfword of GPR rt and store the value into GPR rd. |

Description:

\[ GPR[rd] \leftarrow \text{SwapBytesWithinHalfwords}(GPR[rt]) \]

Within each halfword of GPR rt the bytes are swapped and stored in GPR rd.

Restrictions:

In implementations Release 1 of the architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:

\[ GPR[rd] \leftarrow GPR[rt]_{55.48} \parallel GPR[rt]_{63..56} \parallel GPR[rt]_{39..32} \parallel GPR[rt]_{47..40} \parallel GPR[rt]_{23..16} \parallel GPR[rt]_{31..24} \parallel GPR[rt]_{7..0} \parallel GPR[rt]_{15..8} \]

Exceptions:

Reserved Instruction
Programming Notes:
The DSBH and DSHD instructions can be used to convert doubleword data of one endianness to the other endianness. For example:

```
  ld   t0, 0(a1)  /* Read doubleword value */
  dsbh t0, t0    /* Convert endianness of the halfwords */
  dshd t0, t0    /* Swap the halfwords within the doublewords */
```
Doubleword Swap Halfwords Within Doublewords

Format: dshd rd, rt

Purpose:
To swap the halfwords of GPR rt and store the value into GPR rd.

Description:
GPR[rd] ← SwapHalfwordsWithinDoublewords(GPR[rt])
The halfwords of GPR rt are swapped and stored in GPR rd.

Restrictions:
In implementations of Release 1 of the architecture, this instruction resulted in a Reserved Instruction Exception.

Operation:

Exceptions:
Reserved Instruction
Programming Notes:

The DSBH and DSHD instructions can be used to convert doubleword data of one endianness to the other endianness. For example:

```assembly
ld t0, 0(a1)        /* Read doubleword value */
dsbh t0, t0         /* Convert endianness of the halfwords */
dshd t0, t0         /* Swap the halfwords within the doublewords */
```
## Doubleword Shift Left Logical

### Format:

\[
\text{DSLL} \quad \text{rd, rt, sa}
\]

### MIPS64

### Purpose:
To execute a left-shift of a doubleword by a fixed amount—0 to 31 bits

### Description:

\[
\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rt}] \ll \text{sa}
\]

The 64-bit doubleword contents of GPR \( rt \) are shifted left, inserting zeros into the emptied bits; the result is placed in GPR \( rd \). The bit-shift amount in the range 0 to 31 is specified by \( sa \).

### Restrictions:

### Operation:

\[
s \leftarrow 0 \ || \ sa \\
\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rt}]_{(63-s)} \ || \ 0^{n}
\]

### Exceptions:

Reserved Instruction
Doubleword Shift Left Logical Plus 32

**Format:** DSLL32  rd, rt, sa

**Purpose:**
To execute a left-shift of a doubleword by a fixed amount—32 to 63 bits

**Description:** GPR[rd] ← GPR[rt] << (sa+32)

The 64-bit doubleword contents of GPR rt are shifted left, inserting zeros into the emptied bits; the result is placed in GPR rd. The bit-shift amount in the range 0 to 31 is specified by sa.

**Restrictions:**

**Operation:**

\[
\begin{align*}
s &\leftarrow 1 || sa /* 32+sa */ \\
GPR[rd] &\leftarrow GPR[rt]_{(63-s)}..0 || 0^s
\end{align*}
\]

**Exceptions:**

Reserved Instruction
Doubleword Shift Left Logical Variable

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>DSLLV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** DSLLV rd, rt, rs

**Purpose:**
To execute a left-shift of a doubleword by a variable number of bits

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

The 64-bit doubleword contents of GPR rt are shifted left, inserting zeros into the emptied bits; the result is placed in GPR rd. The bit-shift amount in the range 0 to 63 is specified by the low-order 6 bits in GPR rs.

**Restrictions:**

**Operation:**

```
s ← GPR[rs]5..0
GPR[rd] ← GPR[rt](63–s)..0 || 0s
```

**Exceptions:**

Reserved Instruction
Doubleword Shift Right Arithmetic

| Format: | DSRA  rd, rt, sa |
| Purpose: | To execute an arithmetic right-shift of a doubleword by a fixed amount—0 to 31 bits |
| Description: | GPR[rd] ← GPR[rt] >> sa  (arithmetic) |
| Restrictions: |
| Operation: |
| Exceptions: |
| Reserved Instruction |
### Doubleword Shift Right Arithmetic Plus 32

**Format:**

\[
\text{DSRA32} \quad \text{rd}, \text{rt}, \text{sa}
\]

**Purpose:**

To execute an arithmetic right-shift of a doubleword by a fixed amount—32 to 63 bits

**Description:**

\[
\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rt}] \gg (\text{sa}+32) \quad \text{(arithmetic)}
\]

The doubleword contents of GPR \(rt\) are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR \(rd\). The bit-shift amount in the range 32 to 63 is specified by \(sa+32\).

**Restrictions:**

**Operation:**

\[
\begin{align*}
\text{s} & \leftarrow 1 || \text{sa} /* 32+sa */ \\
\text{GPR}[\text{rd}] & \leftarrow (\text{GPR}[\text{rt}]_{63})^s || \text{GPR}[\text{rt}]_{63..s}
\end{align*}
\]

**Exceptions:**

Reserved Instruction
Doubleword Shift Right Arithmetic Variable

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>DSRAV</td>
<td>010111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  DSRAV rd, rt, rs  

**Purpose:**  To execute an arithmetic right-shift of a doubleword by a variable number of bits  

**Description:**  GPR[rd] ← GPR[rt] >> GPR[rs] (arithmetic)  

The doubleword contents of GPR rt are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR rd. The bit-shift amount in the range 0 to 63 is specified by the low-order 6 bits in GPR rs.

**Restrictions:**

**Operation:**

\[
\begin{align*}
\text{s} & \leftarrow \text{GPR[rs]}_{5..0} \\
\text{GPR[rd]} & \leftarrow (\text{GPR[rt]}_{63})^\text{s} \parallel \text{GPR[rt]}_{63..s}
\end{align*}
\]

**Exceptions:**

Reserved Instruction
Doubleword Shift Right Logical

**Format:** DSRL \( rd, rt, sa \)

**Purpose:**
To execute a logical right-shift of a doubleword by a fixed amount—0 to 31 bits

**Description:**
\[ GPR[rd] \leftarrow GPR[rt] \gg sa \quad \text{(logical)} \]

The doubleword contents of GPR \( rt \) are shifted right, inserting zeros into the emptied bits; the result is placed in GPR \( rd \). The bit-shift amount in the range 0 to 31 is specified by \( sa \).

**Restrictions:**

**Operation:**
\[
\begin{align*}
    s & \leftarrow 0 || sa \\
    GPR[rd] & \leftarrow 0^s || GPR[rt]_{63..s}
\end{align*}
\]

**Exceptions:**
Reserved Instruction
**Doubleword Shift Right Logical Plus 32 (DSRL32)**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>22</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>00000</td>
<td>R</td>
<td>0</td>
<td>rt</td>
<td>rd</td>
<td>saminus32</td>
<td>DSRL32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** DSRL32  rd, rt, sa

**MIPS64**

**Purpose:**
To execute a logical right-shift of a doubleword by a fixed amount—32 to 63 bits

**Description:**

\[
\text{GPR}[rd] \leftarrow \text{GPR}[rt] \gg (\text{saminus32}+32) \quad \text{(logical)}
\]

The 64-bit doubleword contents of GPR rt are shifted right, inserting zeros into the emptied bits; the result is placed in GPR rd. The bit-shift amount in the range 32 to 63 is specified by \( \text{saminus32}+32 \).

**Restrictions:**

**Operation:**

\[
\begin{align*}
s & \leftarrow 1 \mid\mid \text{sa} /* 32+\text{saminus32} */ \\
\text{GPR}[rd] & \leftarrow 0^s \mid\mid \text{GPR}[rt]_{63..s}
\end{align*}
\]

**Exceptions:**

Reserved Instruction
### Doubleword Shift Right Logical Variable (DSRLV)

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
<td>15</td>
<td>11</td>
<td>10</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SPECIAL</td>
<td></td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0000</td>
<td>R</td>
<td>1</td>
<td>DSRLV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** DSRLV rd, rt, rs

**Purpose:**
To execute a logical right-shift of a doubleword by a variable number of bits

**Description:**
GPR[rd] ← GPR[rt] >> GPR[rs] (logical)

The 64-bit doubleword contents of GPR rt are shifted right, inserting zeros into the emptied bits; the result is placed in GPR rd. The bit-shift amount in the range 0 to 63 is specified by the low-order 6 bits in GPR rs.

**Restrictions:**

**Operation:**

\[
\begin{align*}
    s & \leftarrow GPR[rs]_{5..0} \\
    GPR[rd] & \leftarrow 0^s || GPR[rt]_{63..s}
\end{align*}
\]

**Exceptions:**
Reserved Instruction
Doubleword Subtract

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>DSUB</td>
<td>10110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: \( \text{DSUB} \ rd, \ rs, \ rt \)

Purpose:
To subtract 64-bit integers; trap on overflow

Description: \( \text{GPR}[rd] \leftarrow \text{GPR}[rs] - \text{GPR}[rt] \)
The 64-bit doubleword value in GPR \( rt \) is subtracted from the 64-bit value in GPR \( rs \) to produce a 64-bit result. If the subtraction results in 64-bit 2’s complement arithmetic overflow, then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 64-bit result is placed into GPR \( rd \).

Restrictions:

Operation:
\[
\text{temp} \leftarrow (\text{GPR}[rs]_{63} | \text{GPR}[rs]) - (\text{GPR}[rt]_{63} | \text{GPR}[rt])
\]
if (\( \text{temp}_{64} \neq \text{temp}_{63} \)) then
  \( \text{SignalException(IntegerOverflow)} \)
else
  \( \text{GPR}[rd] \leftarrow \text{temp}_{63..0} \)
endif

Exceptions:
Integer Overflow, Reserved Instruction

Programming Notes:
DSUBU performs the same arithmetic operation but does not trap on overflow.
Doubleword Subtract Unsigned

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>DSUBU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** DSUBU rd, rs, rt

**Purpose:**
To subtract 64-bit integers

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

The 64-bit doubleword value in GPR rt is subtracted from the 64-bit value in GPR rs and the 64-bit arithmetic result is placed into GPR rd.

No Integer Overflow exception occurs under any circumstances.

**Restrictions:**
- **Operation:** 64-bit processors
  - GPR[rd] ← GPR[rs] − GPR[rt]

**Exceptions:**
Reserved Instruction

**Programming Notes:**
The term “unsigned” in the instruction name is a misnomer; this operation is 64-bit modulo arithmetic that does not trap on overflow. It is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.
Execution Hazard Barrier

EHB

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>0</td>
<td>0</td>
<td>000000</td>
<td>000000</td>
<td>3</td>
<td>00011</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: EHB

Purpose:
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 MIPS64 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:  
EI rt  
MIPS32 Release 2
MIPS32 Release 2

Purpose:
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 sign-extended and 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] ← sign_extend(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 can not 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>COP0</th>
<th>CO</th>
<th>0</th>
<th>6 5 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>1</td>
<td>000 0000 0000 0000 0000 0000</td>
<td>ERET</td>
</tr>
<tr>
<td>6 1 19</td>
<td>011000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** `ERET`  

**Purpose:**  
To return from interrupt, exception, or error trap.

**Description:**  
ERET clears execution and instruction hazards, conditionally restores SRSCtlCSS from SRSCtlPSS 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 SRSCtlCSS from SRSCtlPSS if StatusBEV = 1, or if StatusERL = 1 because any exception that sets StatusERL to 1 (Reset, Soft Reset, NMI, or cache error) does not save SRSCtlCSS in SRSCtlPSS. If software sets StatusERL to 1, it must be aware of the operation of an ERET that may be subsequently executed.
Exception Return  

<table>
<thead>
<tr>
<th>Operation:</th>
<th>ERET</th>
</tr>
</thead>
<tbody>
<tr>
<td>if StatusERL = 1 then</td>
<td></td>
</tr>
<tr>
<td>temp ← ErrorEPC</td>
<td></td>
</tr>
<tr>
<td>StatusERL ← 0</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>temp ← EPC</td>
<td></td>
</tr>
<tr>
<td>StatusEXL ← 0</td>
<td></td>
</tr>
<tr>
<td>if (ArchitectureRevision ≥ 2) and (SRSCtlHSS &gt; 0) and (StatusREV = 0) then</td>
<td></td>
</tr>
<tr>
<td>SRSCtlCSS ← SRSCtlPSS</td>
<td></td>
</tr>
<tr>
<td>endif</td>
<td></td>
</tr>
<tr>
<td>endif</td>
<td></td>
</tr>
<tr>
<td>if IsMIPS16Implemented() then</td>
<td></td>
</tr>
<tr>
<td>PC ← temp63..1</td>
<td></td>
</tr>
<tr>
<td>ISAMode ← temp0</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>PC ← temp</td>
<td></td>
</tr>
<tr>
<td>endif</td>
<td></td>
</tr>
<tr>
<td>LLbit ← 0</td>
<td></td>
</tr>
<tr>
<td>ClearHazards()</td>
<td></td>
</tr>
</tbody>
</table>

| Exceptions: |  
| Coprocessor Unusable Exception |
### Extract Bit Field

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL3</td>
<td>rs</td>
<td>rt</td>
<td>msbd (size-1)</td>
<td>lsb (pos)</td>
<td>EXT</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \texttt{ext rt, rs, pos, size}

**Purpose:**
To extract a bit field from GPR \(rs\) and store it right-justified into GPR \(rt\).

**Description:**
\[ \text{GPR}[rt] \leftarrow \text{ExtractField}(\text{GPR}[rs], \text{msbd}, \text{lsb}) \]

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:

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

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 3-9** shows the symbolic operation of the instruction.

![Figure 3-9 Operation of the EXT Instruction](image)

**Restrictions:**
In implementations prior to Release of the architecture, this instruction resulted in a Reserved Instruction Exception. The operation is \textbf{UNPREDICTABLE} if \(lsb+msbd > 31\).

If GPR \(rs\) does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is \textbf{UNPREDICTABLE}.
Extract Bit Field, cont.

Operation:

```plaintext
if ((lsb + msbd) > 31) or (NotWordValue(GPR[rs])) then
   UNPREDICTABLE
endif

temp ← sign_extend(0^{32-(msbd+1)} || GPR[rs]_{msbd+1sb..lsb})
GPR[rt] ← temp
```

Exceptions:

Reserved Instruction
Floating Point Floor Convert to Long Fixed Point

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>FLOOR.L.fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Format:  
FLOOR.L.S fd, fs  
FLOOR.L.D fd, fs  

MIPS64, MIPS32 Release 2  
MIPS64, MIPS32 Release 2

Purpose:
To convert an FP value to 64-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 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 16 FP registers mode.

Operation:
StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
Exceptions:
Coprocessor Unusable, Reserved Instruction

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

**FLOOR.W.fmt**

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>FLOOR.W</th>
<th>001111</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td></td>
<td>00000</td>
<td></td>
<td></td>
<td>fd, fs</td>
<td>MIPS32</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>MIPS32</td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

FLOOR.W.S fd, fs

FLOOR.W.D fd, fs

**Purpose:**

To convert an 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 \(-\infty\) (rounding mode 3). The result is placed in FPR fd.

When the source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{31}\) to \(2^{31}-1\), the result cannot be represented correctly, 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, Overflow
**Insert Bit Field**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL3</td>
<td>rs</td>
<td>rt</td>
<td>msb</td>
<td>lsb</td>
<td>INS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \( \text{ins } rt, rs, pos, size \)

**Purpose:**
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.

---

Figure 3-10 Operation of the INS Instruction
Restrictions:
In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception.

The operation is **UNPREDICTABLE** if $lsb > msb$.

If either GPR $rs$ or GPR $rt$ does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

Operation:

```c
if (lsb > msb) or (NotWordValue(GPR[rs])) or (NotWordValue(GPR[rt]))) then
    UNPREDICTABLE
endif
GPR[rt] ← sign_extend(GPR[rt]31..msb+1 || GPR[rs]msb-lsb..0 || GPR[rt]lsb-1..0)
```

Exceptions:

Reserved Instruction
Jump

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>000010</td>
<td>instr_index</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>26</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  \[ J \ target \]

**Purpose:**
To branch within the current 256 MB-aligned region

**Description:**
This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256 MB-aligned region. The low 28 bits of the target address is the \textit{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 \textbf{UNPREDICTABLE} if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**
\[
\begin{align*}
I: & \\
I+1: & PC \leftarrow PC_{GPRLEN-1...28} \ || \ instr\_index \ || \ 0^2
\end{align*}
\]

**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 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 jump 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 and Link

<table>
<thead>
<tr>
<th>Format:</th>
<th>JAL target</th>
<th>MIPS32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>To execute a procedure call within the current 256 MB-aligned region</td>
<td></td>
</tr>
<tr>
<td>Description:</td>
<td>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” 256 MB-aligned region. The low 28 bits of the target address is the $\text{instr_index}$ 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.</td>
<td></td>
</tr>
<tr>
<td>Restrictions:</td>
<td>Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.</td>
<td></td>
</tr>
</tbody>
</table>
| Operation: | $I$: GPR[31] ← PC + 8  
$\text{I+1:PC} \leftarrow \text{PC}_{\text{GPRLEN}-1..28} \mid \mid \text{instr\_index} \mid \mid 0^2$ |
| 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 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 and Link Register JALR

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>0</td>
<td>rd</td>
<td>hint</td>
<td>JALR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000</td>
<td>00000</td>
<td>0000</td>
<td>0000</td>
<td>001001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  
JALR rs (rd = 31 implied)  
JALR rd, rs

**MIPS32**

**Purpose:**
To execute a procedure call to an instruction address in a register

**Description:**  
GPR[rd] ← return_addr, PC ← GPR[rs]

Place the return address link in GPR 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 MIPS16e ASE:*

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

*For processors that do implement the MIPS16e ASE:*

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

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 rs and 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 UNPREDICTABLE. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

The effective target address in GPR rs must be naturally-aligned. For processors that do not implement the MIPS16e ASE, 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, 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.

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 ← GPR[rs]
- GPR[rd] ← PC + 8

**I+1:** if Config₁CA = 0 then
  - PC ← temp
else
  - PC ← tempGPRLEN-1..1 || 0
  - ISAMode ← temp₅
endif

Exceptions:

None

Programming Notes:

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

JALR.HB

Format:

JALR.HB rs (rd = 31 implied)
JALR.HB rd, rs

MIPS32 Release 2
MIPS32 Release 2

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

Description:
GPR[rd] ← return_addr, PC ← GPR[rs], clear execution and instruction hazards

Place the return address link in GPR 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:

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

For processors that do implement the MIPS16 ASE:

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

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 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 JALR.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 JALR.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.

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

Restrictions:

Register specifiers rs and 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 UNPREDICTABLE. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

The effective target address in GPR rs must be naturally-aligned. For processors that do not implement the MIPS16 ASE, 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, 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.
Restrictions, cont.:

After modifying an instruction stream mapping or writing to the instruction stream, execution of those instructions has \texttt{UNPREDICTABLE} behavior until the instruction hazard has been cleared with JALR.HB, JR.HB, ERET, or DERET. Further, the operation is \texttt{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 \texttt{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 \leftarrow \text{GPR}[rs]
    \text{GPR}[rd] \leftarrow \text{PC} + 8
I+1: if Config1CA = 0 then
        \text{PC} \leftarrow \text{temp}
    else
        \text{PC} \leftarrow \text{temp} \& \text{GPRLEN-1..1} \mid | \text{0}
        \text{ISAMode} \leftarrow \text{temp}_0
    endif
    \text{ClearHazard()}
\end{verbatim}

Exceptions:
None

Programming Notes:

JALR and JALR.HB are the only branch-and-link instructions that can select a register for the return link; all other link instructions use GPR 31. The default register for GPR \texttt{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
* a1 = 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 a1 /* Call routine, clearing the hazard */
nop
Jump Register

Format: \texttt{JR rs}

Purpose:
To execute a branch to an instruction address in a register

Description: $PC \leftarrow \text{GPR}[rs]$
Jump to the effective target address in GPR \texttt{rs}. Execute the instruction following the jump, in the branch delay slot, before jumping.

For processors that implement the MIPS16e ASE, 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

Restrictions:
The effective target address in GPR \texttt{rs} must be naturally-aligned. For processors that do not implement the MIPS16e ASE, 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, 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 \texttt{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 \leftarrow \text{GPR}[rs]
I+1: if Config1CA = 0 then
      PC \leftarrow temp
   else
      PC \leftarrow temp_{\text{GPRLEN-1..1}} | | 0
      \text{ISAMode} \leftarrow temp_0
endif
\end{verbatim}

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

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>0</td>
<td>000000</td>
<td>00 0000 0000</td>
<td>1</td>
<td>Any other legal hint value</td>
<td>JR</td>
<td>001000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \( \text{JR.HB} \quad rs \)

**Purpose:**
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 MIPS16 ASE, 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:**

The effective target address in GPR \( rs \) must be naturally-aligned. For processors that do not implement the MIPS16 ASE, 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, 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 JALR.HB. Only hazards created by instructions executed before the JR.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:

I: \( \text{temp} \leftarrow \text{GPR}[rs] \)
I+1: if Config1CA = 0 then
    PC \leftarrow \text{temp}
else
    PC \leftarrow \text{tempGPRLEN}-1..1 \mid 0
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

/*
 * Routine called to modify ASID and return with the new
 * mapping established.
 *
 * a0 = New ASID to establish
 */
  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 */
  jr.hb ra  /* Return, clearing the hazard */
  nop

Example: Making a write to the instruction stream visible

/*
 * Routine called after new instructions are written to
 * make them visible and return with the hazards cleared.
 */
{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

```
la AT, 10f
jr.hb AT /* Jump to next instruction, clearing */
nop /* hazards */
10:
```
Load Byte

Format: \( LB \ rt, \ offset(\text{base}) \)

Purpose:
To load a byte from memory as a signed value

Description: \( GPR[rt] \leftarrow memory[GPR[base] + offset] \)
The contents of the 8-bit byte at the memory location specified by the effective address are fetched, sign-extended, and placed in GPR \( rt \). The 16-bit signed \( offset \) is added to the contents of GPR \( base \) to form the effective address.

Restrictions:
None

Operation:
\[
\begin{align*}
vAddr & \leftarrow \text{sign\_extend}(offset) + GPR[base] \\
pAddr & \leftarrow pAddr_{\text{PSIZE}-1..3} \mid \mid (pAddr_{2..0} \text{xor ReverseEndian}^3) \\
\text{memdoubleword} & \leftarrow \text{LoadMemory}(CCA, \text{BYTE}, pAddr, vAddr, DATA) \\
\text{byte} & \leftarrow vAddr_{2..0} \text{xor BigEndianCPU}^3 \\
GPR[rt] & \leftarrow \text{sign\_extend}((\text{memdoubleword}_{7..8} \text{byte}_{8..9})
\end{align*}
\]

Exceptions:
TLB Refill, TLB Invalid, Address Error, Watch
Load Byte Unsigned LBU

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBU</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100100</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: LBU rt, offset(base)

MIPS32

Purpose:
To load a byte from memory as an unsigned value

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 16-bit signed offset is added to the contents of GPR base to form the effective address.

Restrictions:
None

Operation:

\[
\begin{align*}
vAddr & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR[base]} \\
(pAddr, CCA) & \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{LOAD}) \\
pAddr & \leftarrow pAddr_{\text{PSIZE}-1..1} || (pAddr_{2..0} \oplus \text{ReverseEndian}^3) \\
\text{memdoubleword} & \leftarrow \text{LoadMemory}(CCA, \text{BYTE}, pAddr, vAddr, \text{DATA}) \\
\text{byte} & \leftarrow vAddr_{2..0} \oplus \text{BigEndianCPU}^3 \\
\text{GPR[rt]} & \leftarrow \text{zero\_extend}(\text{memdoubleword}_{7..8*\text{byte}..8*\text{byte}})
\end{align*}
\]

Exceptions:
TLB Refill, TLB Invalid, Address Error, Watch
Load Doubleword

<table>
<thead>
<tr>
<th>LD</th>
<th>base</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>110111</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

**Format:**  $LD \ rt, \ offset(base)$  

**Purpose:**
To load a doubleword from memory

**Description:**  $GPR[rt] \leftarrow \text{memory}[GPR[base] + offset]$  
The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in GPR $rt$. The 16-bit signed $offset$ is added to the contents of GPR $base$ to form the effective address.

**Restrictions:**
The effective address must be naturally-aligned. If any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

**Operation:**

\[
\begin{align*}
vAddr & \leftarrow \text{sign\_extend}(offset) + GPR[base] \\
\text{if } vAddr_{2..0} & \neq 0 \text{ then} \\
& \quad \text{SignalException(AddressError)} \\
\text{endif}
\end{align*}
\]

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

\[
\text{memdoubleword} \leftarrow \text{LoadMemory}(CCA, \text{DOUBLEWORD}, pAddr, vAddr, \text{DATA})
\]

\[
GPR[rt] \leftarrow \text{memdoubleword}
\]

**Exceptions:**
TLB Refill, TLB Invalid, Bus Error, Address Error, Reserved Instruction, Watch
Load Doubleword to Floating Point

**Format:** LDC1 ft, offset(base)

**Purpose:**
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:**

\[
\text{vAddr} \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR[base]}
\]

\[
\begin{align*}
&\text{if vAddr}_{2..0} \neq 0^3 \text{ then} \\
&\quad \text{SignalException(AddressError)} \\
&\text{endif} \\
&(\text{pAddr}, \text{CCA}) \leftarrow \text{AddressTranslation (vAddr, DATA, LOAD)} \\
&\text{memdoubleword} \leftarrow \text{LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)}
\end{align*}
\]

\[
\text{StoreFPR(ft, UNINTERPRETED\_DOUBLEWORD, memdoubleword)}
\]

**Exceptions:**
Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, Address Error, Watch
Load Doubleword to Coprocessor 2

**Format:** LDC2 rt, offset(base)

**Purpose:**
To load a doubleword from memory to a Coprocessor 2 register

**Description:**
\[
\text{CPR}[2, rt, 0] \leftarrow \text{memory}[\text{GPR}[\text{base}] + \text{offset}]
\]

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:**
\[
\text{vAddr} \leftarrow \text{sign}_{\text{extend}}(\text{offset}) + \text{GPR[base]}
\]
\[
\text{if vAddr}_{2,0} \neq 0^{3} \text{ then SignalException(AddressError) endif}
\]
\[
(\text{pAddr, CCA}) \leftarrow \text{AddressTranslation(vAddr, DATA, LOAD)}
\]
\[
\text{memdoubleword} \leftarrow \text{LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)}
\]
\[
\text{CPR}[2, rt, 0] \leftarrow \text{memdoubleword}
\]

**Exceptions:**
Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, Address Error, Watch
LDL

Format: LDL rt, offset(base)

Purpose:
To load the most-significant part of a doubleword 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 8 consecutive bytes forming a doubleword (DW) in memory, starting at an arbitrary byte boundary.

A part of DW, the most-significant 1 to 8 bytes, is in the aligned doubleword containing EffAddr. This part of DW is loaded appropriately into the most-significant (left) part of GPR rt, leaving the remainder of GPR rt unchanged.

Figure 3-11 Unaligned Doubleword Load Using LDL and LDR

Figure 3-11 illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of DW, 6 bytes, is located in the aligned doubleword starting with the most-significant byte at 2. LDL first loads these 6 bytes into the left part of the destination register and leaves the remainder of the destination unchanged. The complementary LDR next loads the remainder of the unaligned doubleword.
The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned doubleword—the low 3 bits of the address \( (v\text{Addr}2..0) \)—and the current byte-ordering mode of the processor (big- or little-endian). Figure 3-12 shows the bytes loaded for every combination of offset and byte ordering.

**Figure 3-12 Bytes Loaded by LDL Instruction**

---

### Restrictions:

#### Operation:

\[
\begin{align*}
v\text{Addr} & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
(p\text{Addr}, \text{CCA}) & \leftarrow \text{AddressTranslation}(v\text{Addr}, \text{DATA}, \text{LOAD}) \\
p\text{Addr} & \leftarrow p\text{Addr}_{\text{PSIZE}-1..3} || (p\text{Addr}_{2..0} \text{xor ReverseEndian}^3) \\
& \text{if BigEndianMem} = 0 \text{ then} \\
& \quad p\text{Addr} \leftarrow p\text{Addr}_{\text{PSIZE}-1..3} || 0^3 \\
& \text{endif} \\
\text{byte} & \leftarrow v\text{Addr}_{2..0} \text{xor BigEndianCPU}^3 \\
\text{memdoublworde} & \leftarrow \text{LoadMemory}(\text{CCA}, \text{byte}, \text{pAddr}, \text{vAddr}, \text{DATA}) \\
\text{GPR}[rt] & \leftarrow \text{memdoublworde}_{7*8+\text{byte}..0} || \text{GPR}[rt]_{55-8*\text{byte}..0}
\end{align*}
\]

#### Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error, Reserved Instruction, Watch
Load Doubleword Right

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>011011</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \( \text{LDR } rt, \text{offset}(\text{base}) \)  

**Purpose:**  
To load the least-significant part of a doubleword from an unaligned memory address

**Description:**  
\( \text{GPR}[rt] \leftarrow \text{GPR}[rt] \text{ MERGE memory}[\text{GPR}[\text{base}] + \text{offset}] \)  
The 16-bit signed \( \text{offset} \) is added to the contents of \( \text{GPR base} \) to form an effective address (\( \text{EffAddr} \)). \( \text{EffAddr} \) is the address of the least-significant of 8 consecutive bytes forming a doubleword (\( \text{DW} \)) in memory, starting at an arbitrary byte boundary.

A part of \( \text{DW} \), the least-significant 1 to 8 bytes, is in the aligned doubleword containing \( \text{EffAddr} \). This part of \( \text{DW} \) is loaded appropriately into the least-significant (right) part of \( \text{GPR rt} \) leaving the remainder of \( \text{GPR rt} \) unchanged.

Figure 3-13 illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in \( 2..9 \) form an unaligned doubleword starting at location 2. Two bytes of the \( \text{DW} \) are located in the aligned doubleword containing the least-significant byte at 9. LDR first loads these 2 bytes into the right part of the destination register, and leaves the remainder of the destination unchanged. The complementary LDL next loads the remainder of the unaligned doubleword.

**Figure 3-13 Unaligned Doubleword Load Using LDR and LDL**

Doubleword 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>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

GPR 24 initial contents

GPR 24 after LDL \( \$24,2(\$0) \)

GPR 24 after LDR \( \$24,9(\$0) \)
The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned doubleword—the low 3 bits of the address (vAddr2..0)—and the current byte-ordering mode of the processor (big- or little-endian).

Figure 3-14 shows the bytes loaded for every combination of offset and byte ordering.

**Figure 3-14 Bytes Loaded by LDR Instruction**

<table>
<thead>
<tr>
<th>Memory contents and byte offsets (vAddr2..0)</th>
<th>Initial contents of Destination Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>most — significance — least</td>
<td>Destination Register</td>
</tr>
<tr>
<td>0   1   2   3   4   5   6   7  ← big-endian</td>
<td>a   b   c   d   e   f   g   h</td>
</tr>
<tr>
<td>7   6   5   4   3   2   1   0  ← little-endian offset</td>
<td></td>
</tr>
</tbody>
</table>

Restrictions:
Load Doubleword Right (cont.)

**Operation: 64-bit processors**

\[
\begin{align*}
v	ext{Addr} & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
(p\text{Addr}, \text{CCA}) & \leftarrow \text{Address\_Translation}(v\text{Addr}, \text{DATA}, \text{LOAD}) \\
p\text{Addr} & \leftarrow \text{pAddr}_{\text{PSIZE}-1..3} || (\text{pAddr}_{2..0} \oplus \text{Reverse\_Endian}^3) \\
\text{if Big\_Endian\_Mem} = 1 \text{ then} \\
\quad p\text{Addr} & \leftarrow \text{pAddr}_{\text{PSIZE}-1..3} || 0^3 \\
\text{endif} \\
\text{byte} & \leftarrow v\text{Addr}_{2..0} \oplus \text{Big\_Endian\_CPU}^3 \\
\text{mem\_double\_word} & \leftarrow \text{Load\_Memory}(\text{CCA}, \text{byte}, \text{pAddr}, v\text{Addr}, \text{DATA}) \\
\text{GPR}[rt] & \leftarrow \text{GPR}[rt]_{63..64-8\times\text{byte}} || \text{mem\_double\_word}_{63..8\times\text{byte}}
\end{align*}
\]

**Exceptions:**

TLB Refill, TLB Invalid, Bus Error, Address Error, Reserved Instruction, Watch
Load Doubleword Indexed to Floating Point

**Format:**  
LDXC1 fd, index(base)

**MIPS64**  
**MIPS32 Release 2**

**Purpose:**  
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).

**Operation:**  

vAddr ← GPR[base] + GPR[index]  
if vAddr2..0 ≠ 0 then  
   SignalException(AddressError)  
endif  
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)  
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)

StoreFPR(ft, UNINTERPRETED_DOUBLEWORD, memdoubleword)

**Exceptions:**  
TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Coprocessor Unusable, Watch
Load Halfword

**Format:** \( \text{LH} \ rt, \text{offset(base)} \)  

**MIPS32**

**Purpose:**
To load a halfword from memory as a signed value

**Description:** \( \text{GPR}[rt] \leftarrow \text{memory}[\text{GPR}[base] + \text{offset}] \)

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 16-bit signed \( \text{offset} \) 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:**

\[
vAddr \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[base]
\]

if \( vAddr_0 \neq 0 \) then
    \( \text{SignalException(AddressError)} \)
endif

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

\[pAddr \leftarrow pAddr_{\text{PSIZE}-1..3} || (pAddr_{2..0} \text{xor (ReverseEndian}^2 || 0))\]

\[\text{memdoubleword} \leftarrow \text{LoadMemory}(CCA, \text{HALFWORD}, pAddr, vAddr, \text{DATA})\]

\[\text{byte} \leftarrow vAddr_{2..0} \text{xor (BigEndianCPU}^2 || 0)\]

\[\text{GPR}[rt] \leftarrow \text{sign\_extend(\text{memdoubleword}_{15+8*\text{byte}..8*\text{byte}})}\]

**Exceptions:**
TLB Refill, TLB Invalid, Bus Error, Address Error, Watch
Load Halfword Unsigned

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LHU</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100101</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** LHU rt, offset(base)

**MIPS32**

**Purpose:**
To load a halfword from memory as an unsigned 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, zero-extended, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

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

**Operation:**

```
  vAddr ← sign_extend(offset) + GPR[base]
  if vAddr₀ ≠ 0 then
    SignalException(AddressError)
  endif
  (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
  pAddr ← pAddr_PSIZE-1..3 || (pAddr₂..0 xor (ReverseEndian₂ || 0))
  memdoubleword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
  byte ← vAddr₂..0 xor (BigEndianCPU₂ || 0)
  GPR[rt] ← zero_extend(memdoubleword₁₅+8*byte..8*byte)
```

**Exceptions:**
TLB Refill, TLB Invalid, Address Error, Watch
Load Linked Word

Format: \( \text{LL } rt, \text{offset}(\text{base}) \)

Purpose:
To load a word from memory for an atomic read-modify-write

Description:
\[
\text{GPR}[rt] \leftarrow \text{memory}[\text{GPR}[\text{base}] + \text{offset}]
\]
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, sign-extended to the GPR register length, and written into GPR \( rt \). The 16-bit signed offset is added to the contents of GPR \( \text{base} \) 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 \text{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:
\[
\begin{align*}
v\text{Addr} & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
& \text{if } v\text{Addr}_{1..0} \neq 0^2 \text{ then} \\
& \quad \text{SignalException(AddressError)} \\
& \text{endif} \\
(p\text{Addr}, \text{CCA}) & \leftarrow \text{AddressTranslation}(v\text{Addr}, \text{DATA}, \text{LOAD}) \\
p\text{Addr} & \leftarrow p\text{Addr}_{\text{PSIZE}-1..3} || (p\text{Addr}_{2..0} \text{xor ReverseEndian} || 0^2) \\
\text{mem\_doubleword} & \leftarrow \text{Load\_Memory}(CCA, \text{WORD}, p\text{Addr}, v\text{Addr}, \text{DATA}) \\
\text{byte} & \leftarrow v\text{Addr}_{2..0} \text{xor BigEndianCPU} || 0^2 \\
\text{GPR}[rt] & \leftarrow \text{sign\_extend}(\text{mem\_doubleword}_{31\times8*\text{byte}..8*\text{byte}}) \\
\text{LLbit} & \leftarrow 1
\end{align*}
\]
Load Linked Word (cont.)

Exceptions:
TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Watch

Programming Notes:
There is no Load Linked Word Unsigned operation corresponding to Load Word Unsigned.
Load Linked Doubleword  

**Format:**

\[
\text{LLD} \quad rt, \ offset(\text{base})
\]

**MIPS64**

**Purpose:**

To load a doubleword from memory for an atomic read-modify-write

**Description:**

\[
\text{GPR}[rt] \leftarrow \text{memory}[\text{GPR}[\text{base}] + offset]
\]

The LLD and SCD instructions provide primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed 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 LLD is executed it starts the active RMW sequence and replaces any other sequence that was active. The RMW sequence is completed by a subsequent SCD instruction that either completes the RMW sequence atomically and succeeds, or does not complete and fails.

Executing LLD on one processor does not cause an action that, by itself, would cause an SCD for the same block to fail on another processor.

An execution of LLD does not have to be followed by execution of SCD; 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 SCD instruction for the formal definition.

The effective address must be naturally-aligned. If any of the 3 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

**Operation:**

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{signExtend}(\text{offset}) + \text{GPR}[\text{base}] \\
\text{if vAddr}_2..0 & \neq 0^3 \text{ then} \\
& \text{SignalException(AddressError)} \\
\text{endif} \\
(\text{pAddr}, \text{CCA}) & \leftarrow \text{AddressTranslation} (\text{vAddr}, \text{DATA}, \text{LOAD}) \\
\text{memdoubleword} & \leftarrow \text{LoadMemory} (\text{CCA}, \text{DOUBLEWORD}, \text{pAddr}, \text{vAddr}, \text{DATA}) \\
\text{GPR}[\text{rt}] & \leftarrow \text{memdoubleword} \\
\text{LLbit} & \leftarrow 1
\end{align*}
\]

MIPS64® Architecture For Programmers Volume II, Revision 2.50  

Copyright © 2001-2003,2005 MIPS Technologies Inc. All rights reserved.
Load Linked Doubleword (cont.)

Exceptions:
TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Watch
### Load Upper Immediate

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUI</td>
<td>0</td>
<td>0</td>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
<tr>
<td>001111</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 6 | 5 | 5 | 16 |

#### Format:
LUI rt, immediate

#### Purpose:
To load a constant into the upper half of a word

#### Description:
GPR[rt] \(\leftarrow\) immediate \(\|\) 0\(^{16}\)

The 16-bit `immediate` is shifted left 16 bits and concatenated with 16 bits of low-order zeros. The 32-bit result is sign-extended and placed into GPR `rt`.

#### Restrictions:
None

#### Operation:
GPR[rt] \(\leftarrow\) signExtend(immediate \(\|\) 0\(^{16}\))

#### Exceptions:
None
Load Doubleword Indexed Unaligned to Floating Point LUXC1

<table>
<thead>
<tr>
<th>COP1X</th>
<th>base</th>
<th>index</th>
<th>0</th>
<th>fd</th>
<th>LUXC1</th>
</tr>
</thead>
<tbody>
<tr>
<td>010011</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Format:** LUXC1 fd, index(base)

**MIPS64**

**MIPS32 Release 2**

**Purpose:**
To load a doubleword from memory to an FPR (GPR+GPR addressing), ignoring alignment

**Description:**
FPR[fd] ← memory[(GPR[base] + 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; EffectiveAddress2..0 are ignored.

**Restrictions:**
The result of this instruction is UNPREDICTABLE if the processor is executing in 16 FP registers mode.

**Operation:**

\[ \text{vAddr} \leftarrow (GPR[\text{base}] + GPR[\text{index}])_{63..3} || 0^3 \]

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

\( \text{memdoubleword} \leftarrow \text{LoadMemory} \ (\text{CCA}, \text{DOUBLEWORD}, \text{pAddr}, \text{vAddr}, \text{DATA}) \)

\( \text{StoreFPR(f, UNINTERPRETED\_DOUBLEWORD, memdoubleword)} \)

**Exceptions:**
Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, Watch
Load Word

<table>
<thead>
<tr>
<th>LW</th>
<th>base</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>100011</td>
<td>6</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

**Format:** \( \text{LW} \ rt, \text{offset} (\text{base}) \)

**Purpose:**
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:**
\[
\begin{align*}
v\text{Addr} & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \\
\text{if} & \ v\text{Addr}_{1..0} \neq 0^2 \ \text{then} \\
& \ \text{SignalException}(\text{AddressError}) \\
\text{endif} \\
(p\text{Addr}, CCA) & \leftarrow \text{AddressTranslation}(v\text{Addr}, \text{DATA, LOAD}) \\
p\text{Addr} & \leftarrow p\text{Addr}_{\text{PSIZE}-1..3} \ || \ (p\text{Addr}_{2..0} \text{xor ReverseEndian} \ || \ 0^2)) \\
\text{memdoubleword} & \leftarrow \text{LoadMemory}(CCA, \text{WORD, pAddr, vAddr, DATA}) \\
\text{byte} & \leftarrow v\text{Addr}_{2..0} \text{xor BigEndianCPU} \ || \ 0^2) \\
\text{GPR}[rt] & \leftarrow \text{sign\_extend(}\text{memdoubleword}_{31..8\text{\_byte}}..8\text{\_byte})
\end{align*}
\]

**Exceptions:**
TLB Refill, TLB Invalid, Bus Error, Address Error, Watch
Load Word to Floating Point

**Format:** LWC1 ft, offset(base)

**Purpose:**
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. 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:**

\[
\begin{align*}
\text{vAddr} &\leftarrow \text{sign\_extend}\text{(offset)} + \text{GPR[base]} \\
\text{if vAddr}_{1..0} &\neq 0^2 \text{ then} \\
&\quad \text{SignalException(AddressError)} \\
\text{endif} \\
\text{pAddr} &\leftarrow \text{AddressTranslation} \left(\text{vAddr, DATA, LOAD}\right) \\
\text{memdoubleword} &\leftarrow \text{LoadMemory}(\text{CCA, WORD, pAddr, vAddr, DATA}) \\
\text{bytesel} &\leftarrow \text{vAddr}_{2..0} \text{ xor (BigEndianCPU || 0}^2) \\
\text{StoreFPR} &\leftarrow \text{StoreFPR}(\text{ft, UNINTERPRETED\_WORD,}} \text{sign\_extend}\left(\text{memdoubleword}_{31+8\times\text{bytesel}..8\times\text{bytesel}}\right) \\
\end{align*}
\]

**Exceptions:**
TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Coprocessor Unusable, Watch
Load Word to Coprocessor 2

**Format:** LWC2 rt, offset(base)

**Purpose:**
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:**

```plaintext
vAddr ← sign_extend(offset) + GPR[base]
if vAddr12..0 ≠ 0² then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0²))
memdoubleword ← LoadMemory(CCA, DOUBLEWORD, pAddr, vAddr, DATA)
bytesel ← vAddr2..0 xor (BigEndianCPU || 0²)
CPR[2,rt,0] ← sign_extend(memdoubleword31+8*bytesel..8*bytesel)
```

**Exceptions:**
TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Coprocessor Unusable, Watch
LWL

Format: \texttt{LWL \textit{rt}, offset(base)}

Purpose:
To load the most-significant part of a word as a signed value from an unaligned memory address

Description: \texttt{GPR[rt] \leftarrow GPR[rt] \text{ MERGE memory}[GPR[base] + offset]}

The 16-bit signed \textit{offset} is added to the contents of GPR \textit{base} to form an effective address \textit{(EffAddr)}. \textit{EffAddr} is the address of the most-significant of 4 consecutive bytes forming a word \textit{(W)} in memory starting at an arbitrary byte boundary.

The most-significant 1 to 4 bytes of \textit{W} is in the aligned word containing the \textit{EffAddr}. This part of \textit{W} is loaded into the most-significant (left) part of the word in GPR \textit{rt}. The remaining least-significant part of the word in GPR \textit{rt} is unchanged.

For 64-bit GPR \textit{rt} registers, the destination word is the low-order word of the register. The loaded value is treated as a signed value; the word sign bit (bit 31) is always loaded from memory and the new sign bit value is copied into bits 63..32.

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 \textit{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.

\textbf{Figure 3-15 Unaligned Word Load Using LWL and LWR}

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LWL</td>
<td>100010</td>
<td>base</td>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>offset</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

Word at byte 2 in big-endian memory; each memory byte contains its own address

\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
most & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
least & & & & & & & & & & \\
\hline
\end{tabular}

Memory initial contents

\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
a & b & c & d & e & f & g & h \\
\hline
\end{tabular}

GPR 24 initial contents

After executing \texttt{LWL \$24,2($0)}

Then after \texttt{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 3-16 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>a b c d e f g h</td>
</tr>
<tr>
<td>offset (vAddr_{1..0})</td>
<td>most — significance — least</td>
</tr>
<tr>
<td>3 2 1 0 ←little-endian</td>
<td>I J K L</td>
</tr>
<tr>
<td>most — significance — least</td>
<td></td>
</tr>
</tbody>
</table>

| Destination register contents after instruction (shaded is unchanged) |
|--------------------------------------------------|-----------------|
| Big-endian byte ordering | vAddr_{1..0} | Little-endian byte ordering |
| sign bit (31) extended | I J K L | 0 | sign bit (31) extended | L f g h |
| sign bit (31) extended | J K L h | 1 | sign bit (31) extended | K L g h |
| sign bit (31) extended | K L g h | 2 | sign bit (31) extended | J K L h |
| sign bit (31) extended | L f g h | 3 | sign bit (31) extended | I J K L |

The word sign (31) is always loaded and the value is copied into bits 63..32.
Load Word Left (con't)

Restrictions:
None

Operation:

\[
\begin{align*}
vAddr & \leftarrow \text{sign}\_\text{extend}(\text{offset}) + \text{GPR[base]} \\
(p\text{Addr}, \text{CCA}) & \leftarrow \text{AddressTranslation}(v\text{Addr}, \text{DATA, LOAD}) \\
p\text{Addr} & \leftarrow p\text{Addr}_{\text{PSIZE}-1..3} \mid\mid (p\text{Addr}_{2..0} \text{ xor ReverseEndian}^3) \\
& \quad \text{if } \text{BigEndianMem} = 0 \text{ then} \\
& \quad \quad p\text{Addr} \leftarrow p\text{Addr}_{\text{PSIZE}-1..3} \mid\mid 0^3 \\
& \quad \text{endif} \\
\text{byte} & \leftarrow 0 \mid\mid (v\text{Addr}_{1..0} \text{ xor BigEndianCPU}^2) \\
\text{word} & \leftarrow v\text{Addr}_2 \text{ xor BigEndianCPU} \\
\text{memdoubleword} & \leftarrow \text{LoadMemory}(\text{CCA, byte, p\text{Addr, v\text{Addr, DATA}}}) \\
\text{temp} & \leftarrow \text{memdoubleword}_{31+32\times\text{word}-8\times\text{byte}..32\times\text{word}} \mid\mid \text{GPR[rt]}_{23-8\times\text{byte}..0} \\
\text{GPR[rt]} & \leftarrow (\text{temp}_{31})^{32} \mid\mid \text{temp}
\end{align*}
\]

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 Right**

**LWR**

```
<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>LWR</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100110</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Format:** $\text{LWR } rt, \text{ offset}(base) \text{ MIPS32}$

**Purpose:**

To load the least-significant part of a word from an unaligned memory address as a signed value.

**Description:**

$\text{GPR}[rt] \leftarrow \text{GPR}[rt] \text{ MERGE memory}[\text{GPR}[base] + \text{offset}]$

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 $\text{EffAddr}$. 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.

If GPR $rt$ is a 64-bit register, the destination word is the low-order word of the register. The loaded value is treated as a signed value; if the word sign bit (bit 31) is loaded (that is, when all 4 bytes are loaded), then the new sign bit value is copied into bits 63..32. If bit 31 is not loaded, the value of bits 63..32 is implementation dependent; the value is either unchanged or a copy of the current value of bit 31.

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.
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 (vAddr1,0), 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 3-18 Bytes Loaded by LWR 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 I J K L offset (vAddr1,0)</td>
<td>a b c d e f g h</td>
</tr>
<tr>
<td>←big-endian</td>
<td>most — significance — least</td>
</tr>
<tr>
<td>3 2 1 0 I J K L offset (vAddr1,0)</td>
<td>no cng or sign extend</td>
</tr>
<tr>
<td>←little-endian</td>
<td>e f g</td>
</tr>
<tr>
<td>most</td>
<td>0</td>
</tr>
<tr>
<td>least</td>
<td>sign bit (31) extended</td>
</tr>
<tr>
<td>— significance —</td>
<td>I J K L</td>
</tr>
</tbody>
</table>

Destination 64-bit register contents after instruction (shaded is unchanged)

<table>
<thead>
<tr>
<th>Big-endian byte ordering</th>
<th>vAddr1,0</th>
<th>Little-endian byte ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>no cng or sign extend</td>
<td>e f g I</td>
<td>sign bit (31) extended I J K L</td>
</tr>
<tr>
<td>no cng or sign extend</td>
<td>e f I J</td>
<td>no cng or sign extend e I J K</td>
</tr>
<tr>
<td>no cng or sign extend</td>
<td>e I J K</td>
<td>no cng or sign extend e f I J</td>
</tr>
<tr>
<td>sign bit (31) extended</td>
<td>I J K L</td>
<td>no cng or sign extend e f g I</td>
</tr>
</tbody>
</table>

The word sign (31) is always loaded and the value is copied into bits 63..32.
Restrictions:

None

Operation:

\[
\begin{align*}
v Addr & \leftarrow \text{sign}_\text{extend}(\text{offset}) + \text{GPR}[\text{base}] \\
(\text{pAddr}, \text{CCA}) & \leftarrow \text{AddressTranslation}(v Addr, \text{DATA, LOAD}) \\
p Addr & \leftarrow \text{pAddr}_{\text{PSize}-1..3} || (\text{pAddr}_{2..0} \text{xor ReverseEndian}^3) \\
& \quad \text{if } \text{BigEndianMem} = 0 \text{ then} \\
& \quad \quad \text{pAddr} \leftarrow \text{pAddr}_{\text{PSize}-1..3} || 0^3 \\
& \quad \text{endif} \\
\text{byte} & \leftarrow v Addr_{1..0} \text{xor BigEndianCPU}^2 \\
\text{word} & \leftarrow v Addr_{2} \text{xor BigEndianCPU} \\
\text{memdoubleword} & \leftarrow \text{LoadMemory}(\text{CCA, byte, pAddr, vAddr, DATA}) \\
\text{temp} & \leftarrow \text{GPR}[\text{rt}]_{31..32-8*\text{byte} || \text{memdoubleword}_{31+32*\text{word}..32*\text{word}+8*\text{byte}} \\
& \quad \text{if } \text{byte} = 4 \text{ then} \\
& \quad \quad \text{utemp} \leftarrow (\text{temp}_{31})^{32} /* \text{loaded bit 31, must sign extend} */ \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{utemp} \leftarrow \text{GPR}[\text{rt}]_{63..32} /* \text{leave what was there alone} */ \\
& \quad \quad \quad \text{utemp} \leftarrow (\text{GPR}[\text{rt}]_{31})^{32} /* \text{sign-extend bit 31} */ \\
& \quad \text{endif} \\
\text{GPR}[\text{rt}] & \leftarrow \text{utemp} || \text{temp}
\end{align*}
\]

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 Unsigned

<table>
<thead>
<tr>
<th>Format:</th>
<th>LWU rt, offset(base)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>To load a word from memory as an unsigned value</td>
</tr>
<tr>
<td>Description:</td>
<td>GPR[rt] ← memory[GPR[base] + offset]</td>
</tr>
<tr>
<td>The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, zero-extended, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.</td>
<td></td>
</tr>
<tr>
<td>Restrictions:</td>
<td>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.</td>
</tr>
<tr>
<td>Operation:</td>
<td>vAddr ← sign_extend(offset) + GPR[base]</td>
</tr>
<tr>
<td>if vAddr_{1,0} ≠ 02 then</td>
<td></td>
</tr>
<tr>
<td>SignalException(AddressError)</td>
<td></td>
</tr>
<tr>
<td>endif</td>
<td></td>
</tr>
<tr>
<td>(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)</td>
<td></td>
</tr>
<tr>
<td>pAddr ← pAddr_{PSIZE-1..3}</td>
<td></td>
</tr>
<tr>
<td>memdoubleword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)</td>
<td></td>
</tr>
<tr>
<td>byte ← vAddr_{2..0} xor (BigEndianCPU</td>
<td></td>
</tr>
<tr>
<td>GPR[rt] ← 0^{32}</td>
<td></td>
</tr>
<tr>
<td>Exceptions:</td>
<td>TLB Refill, TLB Invalid, Bus Error, Address Error, Reserved Instruction, Watch</td>
</tr>
</tbody>
</table>
Load Word Indexed to Floating Point

**Format:** \( \text{LWXC1} \ fd, \ \text{index(base)} \)

**Purpose:**
To load a word from memory to an FPR (GPR+GPR addressing)

**Description:**
\[ \text{FPR}[fd] \leftarrow \text{memory}[\text{GPR}[base] + \text{GPR}[index]] \]
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 \( fd \). The contents of GPR \( index \) and GPR \( base \) are added to form the effective address.

**Restrictions:**
An Address Error exception occurs if EffectiveAddress\text{1..0} ≠ 0 (not word-aligned).

**Operation:**
\[
\begin{align*}
v\text{Addr} & \leftarrow \text{GPR}[base] + \text{GPR}[index] \\
& \text{if } v\text{Addr}_{1..0} \neq 0^2 \text{ then} \\
& \quad \text{SignalException(AddressError)} \\
& \text{endif} \\
\text{pAddr} & \leftarrow \text{AddressTranslation}(v\text{Addr}, \text{DATA}, \text{LOAD}) \\
\text{memdoubleword} & \leftarrow \text{LoadMemory}(\text{CCA}, \text{WORD}, \text{pAddr}, \text{vAddr}, \text{DATA}) \\
\text{bytesel} & \leftarrow v\text{Addr}_{2..0} \text{xor (BigEndianCPU } || 0^2) \\
\text{StoreFPR}(ft, \text{UNINTERPRETED_WORD}, \\
& \quad \text{sign\_extend(memdoubleword}_{31+8\text{bytesel}..8\text{bytesel}}))
\end{align*}
\]

**Exceptions:**
TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Coprocessor Unusable, Watch
Multiply and Add Word to Hi,Lo  

<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>rs</td>
<td>rt</td>
<td>0</td>
<td>0</td>
<td>MADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**  MADD rs, rt  

**Purpose:**  
To multiply two words and add the result to 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 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 HI31..0 and LO31..0. The most significant 32 bits of the result are sign-extended and written into HI and the least significant 32 bits are sign-extended and written into LO. No arithmetic exception occurs under any circumstances.  

**Restrictions:**  
If GPRs rs or rt do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are UNPREDICTABLE.  
This instruction does not provide the capability of writing directly to a target GPR.  

**Operation:**  
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then  
UNPREDICTABLE  
endif  
temp ← (HI31..0 || LO31..0) + (GPR[rs]31..0 × GPR[rt]31..0)  
HI ← sign_extend(temp31..0)  
LO ← sign_extend(temp31..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

<table>
<thead>
<tr>
<th>COP1X</th>
<th>fr</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>MADD</th>
<th>fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>010011</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

**Format:**
- MADD.S fd, fr, fs, ft
- MADD.D fd, fr, fs, ft
- MADD.PS fd, fr, fs, ft

**Purpose:**
To perform a combined multiply-then-add 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 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.

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.

*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 MADD.PS is UNPREDICTABLE if the processor is executing in 16 FP registers mode.

**Operation:**
- \( vfr \leftarrow \text{ValueFPR}(fr, \text{fmt}) \)
- \( vfs \leftarrow \text{ValueFPR}(fs, \text{fmt}) \)
- \( vft \leftarrow \text{ValueFPR}(ft, \text{fmt}) \)
- \( \text{StoreFPR}(fd, \text{fmt}, (vfs \times_{\text{fmt}} vft) +_{\text{fmt}} vfr) \)
Floating Point Multiply Add (cont.)

Exceptions:
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:
Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow
Multiply and Add Unsigned Word to Hi,Lo  

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</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>
<td>10</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>SPECIAL2</td>
<td>rs</td>
<td>rt</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>MADDU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** MADDU rs, rt  

**Purpose:**  
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\_{31..0} and LO\_{31..0}. The most significant 32 bits of the result are sign-extended and written into HI and the least significant 32 bits are sign-extended and written into LO. No arithmetic exception occurs under any circumstances.  

**Restrictions:**  
If GPRs rs or rt do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are UNPREDICTABLE.  
This instruction does not provide the capability of writing directly to a target GPR.  

**Operation:**  
\[
\text{if NotWordValue}(GPR[rs]) \text{ or NotWordValue}(GPR[rt]) \text{ then UNPREDICTABLE}
\]
\[
temp \leftarrow (HI_{31..0} || LO_{31..0}) + ((0^{32} || GPR[rs]_{31..0}) \times (0^{32} || GPR[rt]_{31..0}))
\]
\[
HI \leftarrow \text{sign\_extend}(temp_{63..32})
\]
\[
LO \leftarrow \text{sign\_extend}(temp_{31..0})
\]

**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.
## Move from Coprocessor 0

<table>
<thead>
<tr>
<th>COP0</th>
<th>MF</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
<th>sel</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

**Format:**
- MFC0 rt, rd, sel
- MFC0 rt, rd

**MIPS32**

**MIPS32**

**Purpose:**
To move the contents of a coprocessor 0 register to a general register.

**Description:**
\[ \text{GPR}[rt] \leftarrow \text{CPR}[0, rd, sel] \]

The contents of the coprocessor 0 register specified by the combination of rd and sel are sign-extended and 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:**
\[
\begin{align*}
\text{data} & \leftarrow \text{CPR}[0, rd, sel]_{31..0} \\
\text{GPR}[rt] & \leftarrow \text{sign\_extend(data)}
\end{align*}
\]

**Exceptions:**
- Coprocessor Unusable
- Reserved Instruction
Move Word From Floating Point

**Format:**  \texttt{MFC1 \textit{rt}, \textit{fs}}

**Purpose:**
To copy a word from an FPU (CP1) general register to a GPR

**Description:**  \texttt{GPR[rt] \rightarrow FPR[fs]}
The contents of FPR \textit{fs} are sign-extended and loaded into general register \textit{rt}.

**Restrictions:**

**Operation:**
\[\text{data} \leftarrow \text{ValueFPR} (\textit{fs}, \text{UNINTERPRETED\_WORD})_{31..0}\]
\[\text{GPR[rt]} \leftarrow \text{sign\_extend} (\text{data})\]

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Historical Information:**
For MIPS I, MIPS II, and MIPS III the contents of GPR \textit{rt} are \textbf{UNPREDICTABLE} for the instruction immediately following MFC1.
Move Word From Coprocessor 2

<table>
<thead>
<tr>
<th>COP2</th>
<th>MF</th>
<th>rt</th>
<th>Impl</th>
</tr>
</thead>
<tbody>
<tr>
<td>010010</td>
<td>00000</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Format:  
MFC2 rt, rd  
MFC2, rt, rd, sel

The syntax shown above is an example using MFC1 as a model. The specific syntax is implementation dependent.

Purpose:  
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 sign-extended 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:  
\[
\text{data} \leftarrow \text{CP2CPR}[\text{Impl}]_{31..0} \\
\text{GPR}[\text{rt}] \leftarrow \text{sign} \_\text{extend}(\text{data})
\]

Exceptions:  
Coprocessor Unusable
Move Word From High Half of Floating Point Register

<table>
<thead>
<tr>
<th>Format:</th>
<th>MFHC1 rt, fs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>To copy a word from the high half of an FPU (CP1) general register to a GPR</td>
</tr>
<tr>
<td>Description:</td>
<td>GPR[rt] ← sign_extend(FPR[fs]63..32)</td>
</tr>
<tr>
<td>The contents of the high word of FPR fs are sign-extended and 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.</td>
<td></td>
</tr>
<tr>
<td>Restrictions:</td>
<td>In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction Exception. The results are UNPREDICTABLE if StatusFR = 0 and fs is odd.</td>
</tr>
<tr>
<td>Operation:</td>
<td>data ← ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)63..32</td>
</tr>
<tr>
<td>GPR[rt] ← sign_extend(data)</td>
<td></td>
</tr>
<tr>
<td>Exceptions:</td>
<td>Coprocessor Unusable</td>
</tr>
<tr>
<td>Reserved Instruction</td>
<td></td>
</tr>
</tbody>
</table>
Move Word From High Half of Coprocessor 2 Register  

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>3</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP2</td>
<td>010010</td>
<td>MFH</td>
<td>00011</td>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Impl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

**Format:**  
MFHC2 rt, rd  
MFHC2, rt, rd, sel  

MIPS32 Release 2  
MIPS32 Release 2  

The syntax shown above is an example using MFHC1 as a model. The specific syntax is implementation dependent.

**Purpose:**  
To copy a word from the high half of a COP2 general register to a GPR

**Description:**  
\[
\text{GPR}[rt] \leftarrow \text{sign} \_\text{extend}(\text{CP2CPR}[\text{Impl}]_{63..32})
\]

The contents of the high word of the coprocessor 2 register denoted by the \textit{Impl} field are sign-extended and placed into GPR \textit{rt}. The interpretation of the \textit{Impl} field is left entirely to the Coprocessor 2 implementation and is not specified by the architecture.

**Restrictions:**  
The results are \textbf{UNPREDICTABLE} if \textit{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:**  
\[
\text{data} \leftarrow \text{CP2CPR}[\text{Impl}]_{63..32}
\]
\[
\text{GPR}[rt] \leftarrow \text{sign} \_\text{extend}(\text{data})
\]

**Exceptions:**  
Coprocessor Unusable

Reserved Instruction
### Move From HI Register

<table>
<thead>
<tr>
<th>Format: MFHI rd</th>
<th>MIPS32</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose:</strong></td>
<td>To copy the special purpose HI register to a GPR</td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>GPR[rd] ← HI</td>
</tr>
<tr>
<td></td>
<td>The contents of special register HI are loaded into GPR rd.</td>
</tr>
<tr>
<td><strong>Restrictions:</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>Operation:</strong></td>
<td>GPR[rd] ← HI</td>
</tr>
<tr>
<td><strong>Exceptions:</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>Historical Information:</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
Move From LO Register

<table>
<thead>
<tr>
<th></th>
<th>MIPS32</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MFLO</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>0</td>
<td>rd</td>
<td>0</td>
<td>000000</td>
<td>010010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** MFLO rd

**Purpose:**
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 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.
Floating Point Move

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>MOV</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>0</td>
<td>0000</td>
<td>5</td>
<td>5</td>
<td>000110</td>
</tr>
</tbody>
</table>

**Format:**
- MOV.S fd, fs
- MOV.D fd, fs
- MOV.PS fd, fs

**MIPS32**
- MOV.S fd, fs
- MOV.D fd, fs
- MOV.PS fd, fs

**MIPS64, MIPS32 Release 2**

**Purpose:**
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 16 FP registers mode.

**Operation:**

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

**Exceptions:**
- Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
- Unimplemented Operation
Move Conditional on Floating Point False

**Format:** MOVF rd, rs, cc

**Purpose:**
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:**

```markdown
if FPConditionCode(cc) = 0 then
    GPR[rd] ← GPR[rs]
endif
```

**Exceptions:**
Reserved Instruction, Coprocessor Unusable
Floating Point Move Conditional on Floating Point False

**Format:**

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>cc</th>
<th>tf</th>
<th>fs</th>
<th>fd</th>
<th>MOVCF</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>010001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS32

MIPS32

MIPS64

MIPS32 Release 2

**Purpose:**

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 16 FP registers mode.
Operation:

if fmt ≠ PS
  if FPConditionCode(cc) = 0 then
    StoreFPR(fd, fmt, ValueFPR(fs, fmt))
  else
    StoreFPR(fd, fmt, ValueFPR(fd, fmt))
  endif
else
  mask ← 0
  if FPConditionCode(cc+0) = 0 then mask ← mask or 0xF0 endif
  if FPConditionCode(cc+1) = 0 then mask ← mask or 0x0F endif
  StoreFPR(fd, PS, ByteMerge(mask, fd, fs))
endif

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation
Move Conditional on Not Zero

Format: MOVN rd, rs, rt

Purpose:
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 here is the condition true result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.
## Floating Point Move Conditional on Not Zero

### MOVN.fmt

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>rt</th>
<th>fs</th>
<th>fd</th>
<th>MOVN</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Format:</th>
<th>MOVN.S fd, fs, rt</th>
<th>MIPS32</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVN.D fd, fs, rt</td>
<td>MIPS32</td>
<td></td>
</tr>
<tr>
<td>MOVN.PS fd, fs, rt</td>
<td>MIPS64, MIPS32 Release 2</td>
<td></td>
</tr>
</tbody>
</table>

### Purpose:

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 16 FP registers mode.
Floating Point Move Conditional on Not Zero

**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:**  
To test an FP condition code then conditionally move a GPR

**Description:**  
if FPConditionCode(cc) = 1 then GPR[rd] ← 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:**

```c
if FPConditionCode(cc) = 1 then
    GPR[rd] ← GPR[rs]
endif
```

**Exceptions:**

Reserved Instruction, Coprocessor Unusable
Floating Point Move Conditional on Floating Point True

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>cc</th>
<th>0</th>
<th>tf</th>
<th>0</th>
<th>1</th>
<th>fs</th>
<th>fd</th>
<th>MOVCF</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: MOVT.S fd, fs, cc
MOVT.D fd, fs, cc
MOVT.PS fd, fs, cc

Purpose:
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 undefined.
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 16 FP registers mode.
Operation:

```c
if fmt ≠ PS
    if FPConditionCode(cc) = 0 then
        StoreFPR(fd, fmt, ValueFPR(fs, fmt))
    else
        StoreFPR(fd, fmt, ValueFPR(fd, fmt))
    endif
else
    mask ← 0
    if FPConditionCode(cc+0) = 0 then mask ← mask or 0xF0 endif
    if FPConditionCode(cc+1) = 0 then mask ← mask or 0x0F endif
    StoreFPR(fd, PS, ByteMerge(mask, fd, fs))
endif
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation
## Move Conditional on Zero

**Format:**  
MOVZ rd, rs, rt

**Purpose:**  
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 here is the condition false result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.
Floating Point Move Conditional on Zero

**MOVZ.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>rt</td>
<td>fs</td>
<td>fd</td>
<td>MOEZ</td>
<td>010010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

**Format:** MOVZ.S fd, fs, rt  
MOVZ.D fd, fs, rt  
MOVZ.PS fd, fs, rt

**MIPS32**  
**MIPS32**  
**MIPS64, MIPS32 Release 2**

**Purpose:**
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 16 FP registers mode.
Floating Point Move Conditional on Zero (cont.)

**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  

\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
31 & 26 & 25 & 21 & 20 & 16 & 15 \\
\hline
SPECIAL2 & rs & rt & 0 & 0 & 0 & 0 \\
011100 & 6 & 5 & 5 & 5 & 6 & 0 \\
\hline
\end{tabular}

**Format:**  \texttt{MSUB rs, rt}  

**Purpose:**  
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 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 \(HI_{31..0}\) and \(LO_{31..0}\). The most significant 32 bits of the result are sign-extended and written into \(HI\) and the least significant 32 bits are sign-extended and written into \(LO\). No arithmetic exception occurs under any circumstances.  

**Restrictions:**  
If GPRs \(rs\) or \(rt\) do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are **UNPREDICTABLE**.  

This instruction does not provide the capability of writing directly to a target GPR.  

**Operation:**  
\[
\text{if NotWordValue(GPR}[rs] \) or NotWordValue(GPR}[rt] \) then  
\text{UNPREDICTABLE}  
\text{endif}  
\text{temp} \leftarrow (\text{HI}_{31..0} || \text{LO}_{31..0}) - (\text{GPR}[rs]_{31..0} \times \text{GPR}[rt]_{31..0})  
\text{HI} \leftarrow \text{sign\_extend(temp}_{63..32})  
\text{LO} \leftarrow \text{sign\_extend(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 Subtract

<table>
<thead>
<tr>
<th>COP1X</th>
<th>fr</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>MSUB</th>
<th>fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>010011</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>101</td>
<td>3</td>
</tr>
</tbody>
</table>

**Format:**

- MSUB.S fd, fr, fs, ft
- MSUB.D fd, fr, fs, ft
- MSUB.PS fd, fr, fs, ft

**Purpose:**
To perform a combined multiply-then-subtract of FP values

**Description:**

\[ \text{FPFR}[fd] \leftarrow (\text{FPFR}[fs] \times \text{FPFR}[ft]) - \text{FPFR}[fr] \]

The value in FPR fs is multiplied by the value in FPR ft to produce an intermediate product. The value in FPR fr is subtracted from the product. 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.

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 16 FP registers mode.

**Operation:**

\[
\begin{align*}
\text{vfr} & \leftarrow \text{ValueFPR}(fr, \text{fmt}) \\
\text{vfs} & \leftarrow \text{ValueFPR}(fs, \text{fmt}) \\
\text{vft} & \leftarrow \text{ValueFPR}(ft, \text{fmt}) \\
\text{StoreFPR}(fd, \text{fmt}, (\text{vfs} \times_{\text{fmt}} \text{vft}) -_{\text{fmt}} \text{vfr}))
\end{align*}
\]
Floating Point Multiply Subtract (cont.)

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow
Multiply and Subtract Word to Hi,Lo

<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>rs</td>
<td>rt</td>
<td>0</td>
<td>0</td>
<td>MSUBU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>00000</td>
<td>00000</td>
<td>000101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: MSUBU rs, rt

Purpose:
To multiply two words and subtract the result from 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 subtracted from the 64-bit concatenated values of \(HI_{31..0}\) and \(LO_{31..0}\). The most significant 32 bits of the result are sign-extended and written into \(HI\) and the least significant 32 bits are sign-extended and written into \(LO\). No arithmetic exception occurs under any circumstances.

Restrictions:
If GPRs rs or rt do not contain sign-extended 32-bit values (bits 63..31 equal), then the results of the operation are UNPREDICTABLE.

This instruction does not provide the capability of writing directly to a target GPR.

Operation:

\[
\text{if NotWordValue}(GPR[rs]) \text{ or NotWordValue}(GPR[rt]) \text{ then UNPREDICTABLE}\n\]

\[
temp \leftarrow (HI_{31..0} || LO_{31..0}) - ((0^{32} || GPR[rs]_{31..0}) \times (0^{32} || GPR[rt]_{31..0}))
\]

\[
HI \leftarrow \text{sign\_extend}(temp_{63..32})
\]

\[
LO \leftarrow \text{sign\_extend}(temp_{31..0})
\]

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.
Move to Coprocessor 0

<table>
<thead>
<tr>
<th>COP0</th>
<th>MT</th>
<th>rd</th>
<th>rd</th>
<th>0</th>
<th>sel</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>00100</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Format:</th>
<th>MTC0 rt, rd</th>
<th>MIPS32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MTC0 rt, rd, sel</td>
<td>MIPS32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Purpose:</th>
</tr>
</thead>
<tbody>
<tr>
<td>To move the contents of a general register to a coprocessor 0 register.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description:</th>
<th>CPR[0, rd, sel] ← GPR[rt]</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Restrictions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The results are UNDEFINED if coprocessor 0 does not contain a register as specified by rd and sel.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>data ← GPR[rt]</td>
</tr>
<tr>
<td>if (Width(CPR[0,rd,sel]) = 64) then</td>
</tr>
<tr>
<td>CPR[0,rd,sel] ← data</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>CPR[0,rd,sel] ← data31..0</td>
</tr>
<tr>
<td>endif</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coprocessor Usable</td>
</tr>
<tr>
<td>Reserved Instruction</td>
</tr>
</tbody>
</table>
### Move Word to Floating Point

<table>
<thead>
<tr>
<th>Format:</th>
<th>MTC1 rt, fs</th>
<th>MIPS32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>To copy a word from a GPR to an FPU (CP1) general register</td>
<td></td>
</tr>
<tr>
<td>Description:</td>
<td>FPR[fs] ← GPR[rt]</td>
<td></td>
</tr>
<tr>
<td>The low word in GPR rt is placed into the low word of FPR fs. If FPRs are 64 bits wide, bits 63..32 of FPR fs become undefined.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Restrictions:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Operation: | data ← GPR[rt]31..0  
StoreFPR(fs, UNINTERPRETED_WORD, data) |
| Exceptions: | Coprocessor Unusable |
| Historical Information: | For MIPS I, MIPS II, and MIPS III the value of FPR fs is UNPREDICTABLE for the instruction immediately following MTC1. |
Move Word to Coprocessor 2

<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>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP2</td>
<td>MT</td>
<td>rt</td>
<td>Impl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010010</td>
<td>00100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MPC2**

**Format:**
- MTC2 rt, rd
- MTC2 rt, rd, sel

The syntax shown above is an example using MTC1 as a model. The specific syntax is implementation dependent.

**Purpose:**
To copy a word from a GPR to a COP2 general register

**Description:**
\[
\text{CP2CPR[Impl]} \leftarrow \text{GPR[rt]}
\]

The low word in GPR rt is placed into the low word of coprocessor 2 general register denoted by the Impl field. If coprocessor 2 general registers are 64 bits wide, bits 63..32 of the register denoted by the Impl field become undefined. 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:**
\[
data \leftarrow \text{GPR[rt]}_{31..0}
\]
\[
\text{CP2CPR[Impl]} \leftarrow data
\]

**Exceptions:**
- Coprocessor Unusable
- Reserved Instruction
Move Word to High Half of Floating Point Register

**MTHC1**

**Format:**  
MTHC1 rt, fs

**Purpose:**  
To copy a word from a GPR to the high half of an FPU (CP1) general register

**Description:**  
FPR[fs]63..32 ← GPR[rt]31..0  
The low 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 **UNPREDICTABLE** if StatusFR = 0 and fs is odd.

**Operation:**  
newdata ← GPR[rt]31..0  
olddata ← ValueFPR(fs, UNINTERPRETED_DOUBLEWORD)31..0  
StoreFPR(fs, UNINTERPRETED_DOUBLEWORD, newdata || olddata)

**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>COP2</th>
<th>MTH</th>
<th>rt</th>
<th>Impl</th>
</tr>
</thead>
<tbody>
<tr>
<td>010010</td>
<td>00111</td>
<td>rt</td>
<td>Impl</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

**Format:**

MTHC2 rt, rd

MTHC2 rt, rd, sel

MIPS32 Release 2

MIPS32 Release 2

The syntax shown above is an example using MTHC1 as a model. The specific syntax is implementation dependent.

**Purpose:**

To copy a word from a GPR to the high half of a COP2 general register

**Description:**

\[ CP2CPR[\text{Impl}]_{63..32} \leftarrow GPR[rt]_{31..0} \]

The low 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:**

\[
\begin{align*}
\text{data} & \leftarrow GPR[rt]_{31..0} \\
CP2CPR[\text{Impl}] & \leftarrow \text{data} \parallel CPR[2, rd, sel]_{31..0}
\end{align*}
\]

**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 an **UNPREDICTABLE** value.
Move to HI Register

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>0</th>
<th>6</th>
<th>5</th>
<th>15</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>0</td>
<td>0000000000000000</td>
<td>MTHI</td>
<td>010001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** MTHI rs

**Purpose:**
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, DDIV, DDIVU, DMULT, DMULTU, MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either HI or LO.
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:
```
MUL 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:
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, DDIV, DDIVU, DMULT, DMULTU, MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either HI or LO.

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:

```
MUL 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.
MUL

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 sign-extended and written to GPR rd. The contents of HI and LO are UNPREDICTABLE after the operation. No arithmetic exception occurs under any circumstances.

Restrictions:
On 64-bit processors, if either GPR rt or GPR rs does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Note that this instruction does not provide the capability of writing the result to the HI and LO registers.

Operation:

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then
   UNPREDICTABLE
endif

   temp ← GPR[rs] * GPR[rt]
   GPR[rd] ← sign_extend(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

MUL.fmt

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>MUL</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>000010</td>
</tr>
</tbody>
</table>

**Format:**
- MUL.S fd, fs, ft
- MUL.D fd, fs, ft
- MUL.PS fd, fs, ft

**Purpose:**
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 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 MUL.PS is UNPREDICTABLE if the processor is executing in 16 FP registers mode.

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

**Exceptions:**
Coprocessor Usable, Reserved Instruction

**Floating Point Exceptions:**
Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow
**Format:** \[ \text{MULT } rs, rt \]

**Purpose:**
To multiply 32-bit signed integers

**Description:** \((HI, LO) \leftarrow \text{GPR}[rs] \times \text{GPR}[rt]\)

The 32-bit word value in GPR \(rt\) is multiplied by the 32-bit value in GPR \(rs\), treating both operands as signed values, to produce a 64-bit result. The low-order 32-bit word of the result is sign-extended and placed into special register \(LO\), and the high-order 32-bit word is sign-extended and placed into special register \(HI\).

No arithmetic exception occurs under any circumstances.

**Restrictions:**

On 64-bit processors, if either GPR \(rt\) or GPR \(rs\) does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

**Operation:**

\[
\begin{align*}
\text{if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then} \\
\text{UNPREDICTABLE} \\
\text{endif} \\
\text{prod} \leftarrow \text{GPR}[rs]_{31..0} \times \text{GPR}[rt]_{31..0} \\
\text{LO} \leftarrow \text{sign_extend(prod}_{31..0}) \\
\text{HI} \leftarrow \text{sign_extend(prod}_{63..32})
\end{align*}
\]

**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.
**MULTU**

**Format:**  
MULTU rs, rt

**MIPS32**

**Purpose:**
To multiply 32-bit unsigned integers

**Description:**  
(HI, LO) ← GPR[rs] × 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 sign-extended and placed into special register LO, and the high-order 32-bit word is sign-extended and placed into special register HI.

No arithmetic exception occurs under any circumstances.

**Restrictions:**

On 64-bit processors, if either GPR rt or GPR rs does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

**Operation:**

if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
  UNPREDICTABLE
endif

  prod ← (0 || GPR[rs]31..0) × (0 || GPR[rt]31..0)
  LO ← sign_extend(prod31..0)
  HI ← sign_extend(prod63..32)

**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

**Format:**

<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>NEG</td>
<td>000111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NEG.S fd, fs  
NEG.D fd, fs  
NEG.PS fd, fs

**MIPS32**  
**MIPS32**  
**MIPS64, MIPS32 Release 2**

**Purpose:**

To negate an FP value

**Description:** $FPR[fd] \leftarrow -FPR[fs]$

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.

This operation is arithmetic; a NaN operand signals invalid operation.

**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 16 FP registers mode.

**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

<table>
<thead>
<tr>
<th>COP1X</th>
<th>fr</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>NMADD</th>
<th>fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>010011</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

**Format:**  
NMADD.S fd, fr, fs, ft  
NMADD.D fd, fr, fs, ft  
NMADD.PS fd, fr, fs, ft

**Purpose:**  
To negate a combined multiply-then-add of FP values

**Description:**  
\[ FPR[fd] ← − (FPR[fs] \times FPR[ft]) + FPR[fr] \]

The value in FPR \( fs \) is multiplied by the value in FPR \( ft \) to produce an intermediate product. The 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 \( fmt \).

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 16 FP registers mode.

**Operation:**

\[
\begin{align*}
\text{vfr} & \leftarrow \text{ValueFPR}(fr, fmt) \\
\text{vfs} & \leftarrow \text{ValueFPR}(fs, fmt) \\
\text{vft} & \leftarrow \text{ValueFPR}(ft, fmt) \\
\text{StoreFPR}(fd, fmt, −(vfr +_{fmt} (vfs \times_{fmt} vft)))
\end{align*}
\]
Floating Point Negative Multiply Add (cont.)

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow
Floating Point Negative Multiply Subtract  

<table>
<thead>
<tr>
<th>COP1X</th>
<th>fr</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>NMSUB</th>
<th>fmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>010011</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>111</td>
<td>33</td>
</tr>
</tbody>
</table>

**Format:**

- NMSUB.S fd, fr, fs, ft
- NMSUB.D fd, fr, fs, ft
- NMSUB.PS fd, fr, fs, ft

**Purpose:**

To negate a combined multiply-then-subtract 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 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.

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 16 FP registers mode.

**Operation:**

\[
\begin{align*}
 vfr & \leftarrow \text{ValueFPR}(fr, fmt) \\
 vfs & \leftarrow \text{ValueFPR}(fs, fmt) \\
 vft & \leftarrow \text{ValueFPR}(ft, fmt) \\
 \text{StoreFPR}(fd, fmt, -(vfs \times_{fmt} vft) -_{fmt} vfr)
\end{align*}
\]
### Floating Point Negative Multiply Subtract (cont.)

<table>
<thead>
<tr>
<th>NMSUB.fmt</th>
</tr>
</thead>
</table>

**Exceptions:**
- Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
- Inexact, Unimplemented Operation, Invalid Operation, Overflow, Underflow
No Operation

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SLL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** NOP

**Purpose:**
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**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>NOR</td>
<td>100111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** NOR rd, rs, rt  
**MIPS32**

**Purpose:**  
To do a bitwise logical NOT OR

**Description:** GPR[rd] ← GPR[rs] 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] ← GPR[rs] nor GPR[rt]

**Exceptions:**  
None
Format: \texttt{OR rd, rs, rt} \quad \texttt{MIPS32}

Purpose:
To do a bitwise logical OR

Description: \texttt{GPR[rd] \leftarrow GPR[rs] \text{ or } GPR[rt]}

The contents of GPR \textit{rs} are combined with the contents of GPR \textit{rt} in a bitwise logical OR operation. The result is placed into GPR \textit{rd}.

Restrictions:
None

Operation:
\texttt{GPR[rd] \leftarrow GPR[rs] \text{ or } GPR[rt]}

Exceptions:
None
**Or Immediate**

<table>
<thead>
<tr>
<th>ORI</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>001101</td>
<td>6</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

**Format:** ORI \(rt, rs, \text{immediate}\)

**Purpose:**
To do a bitwise logical OR with a constant

**Description:** \(GPR[rt] \leftarrow GPR[rs] \text{ or immediate}\)

The 16-bit \textit{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:**
\[GPR[rt] \leftarrow GPR[rs] \text{ or zero}_\text{extend}(\text{immediate})\]

**Exceptions:**
None
PLL.PS

**Format:** PLL.PS fd, fs, ft

**Purpose:**
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 16 FP registers mode.

**Operation:**

```
StoreFPR(fd, PS, ValueFPR(fs, PS)31..0 || ValueFPR(ft, PS)31..0)
```

**Exceptions:**
Coprocessor Unusable, Reserved Instruction
Format: \texttt{PLU.PS fd, fs, ft} \hfill \text{MIPS64, MIPS32 Release 2}

Purpose:
To merge a pair of paired single values with realignment

Description: \texttt{FPR[fd] \leftarrow lower(FPR[fs]) \mid\rangle upper(FPR[ft])}
A new paired-single value is formed by catenating the lower single of FPR \textit{fs} (bits 31..0) and the upper single of FPR \textit{ft} (bits 63..32).
The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:
The fields \textit{fs, ft}, and \textit{fd} must specify FPRs valid for operands of type \textit{PS}. If they are not valid, the result is \texttt{UNPREDICTABLE}.
The result of this instruction is \texttt{UNPREDICTABLE} if the processor is executing in 16 FP registers mode.

Operation:
\begin{equation}
\text{StoreFPR(fd, PS, ValueFPR(fs, PS)_{31..0} \mid\rangle ValueFPR(ft, PS)_{63..32}}
\end{equation}

Exceptions:
Coprocessor Unusable, Reserved Instruction
**Prefetch**

**Format:**  
PREF hint,offset(base)

**Purpose:**  
To move data between memory and cache.

**Description:**  
prefetch_memory(GPR[base] + offset)

Prefetch 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.

Prefetch 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.

Prefetch 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.

Prefetch 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 coherency attribute of a segment (e.g., the use of the K0, KU, or K23 fields in the Config register), or the per-page coherency attribute provided by the TLB.

If PREF results in a memory operation, the memory access type and coherency attribute used for the operation are determined by the memory access type and 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.
Table 3-30 Values of the hint Field for the PREF 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>8-24</td>
<td>Reserved</td>
<td>Reserved for future use - not available to implementations.</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>-------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 25   | writeback_invalidate (also known as “nudge”) | 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. |
| 26-29| Implementation Dependent | Unassigned by the Architecture - available for implementation-dependent use. |
| 30   | PrepareForStore | 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. |
| 31   | Implementation Dependent | Unassigned by the Architecture - available for implementation-dependent use. |
Restrictions:
None

Operation:
\[
vAddr \leftarrow GPR[base] + \text{sign}_{-}\text{extend}(offset)
(\text{pAddr, CCA}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA, LOAD})
\text{Prefetch}(\text{CCA, pAddr, vAddr, DATA, hint})
\]

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 Indexed**

<table>
<thead>
<tr>
<th>Format:</th>
<th>PREFX hint, index(base)</th>
</tr>
</thead>
</table>

**Purpose:**
To move data between memory and cache.

**Description:**
prefetch_memory[GPR[base] + GPR[index]]

PREFX 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 PREFX 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:**

**Operation:**

\[ vAddr \leftarrow GPR[base] + GPR[index] \]

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

\[ \text{Prefetch}(CCA, pAddr, vAddr, \text{DATA}, \text{hint}) \]

**Exceptions:**
Coprocessor Unusable, Reserved Instruction, Bus Error, Cache Error

**Programming Notes:**
The PREFX instruction is only available on processors that implement floating point and should never by generated by compilers in situations other than those in which the corresponding load and store indexed floating point instructions are generated.

Also refer to the corresponding section in the PREF instruction description.
Purpose:
To merge a pair of paired single values with realignment

Description:
\[ \text{FPR}[fd] \leftarrow \text{upper(FPR}[fs]) \ || \ \text{lower(FPR}[ft]) \]

A new paired-single value is formed by concatenating 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 16 FP registers mode.

Operation:
\[ \text{StoreFPR}(fd, PS, \text{ValueFPR}(fs, PS)_{63..32} \ || \ \text{ValueFPR}(ft, PS)_{31..0}) \]

Exceptions:
Coprocessor Unusable, Reserved Instruction
Format:  PUU.PS fd, fs, ft

Purpose:
To merge a pair of paired single values with realignment

Description: FPR[fd] ← upper(FPR[fs]) || upper(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 16 FP registers mode.

Operation:
    StoreFPR(fd, PS, ValueFPR(fs, PS)63..32 || ValueFPR(ft, PS)63..32)

Exceptions:
Coprocessor Unusable, Reserved Instruction
Read Hardware Register RDHWR

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPECIAL3</td>
<td>0</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>0000</td>
<td>0000</td>
<td>RDHWR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111 11</td>
<td>00 000</td>
<td>11 1011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** RDHWR rt, rd

**Purpose:**

To move the contents of a hardware register to a general purpose register (GPR) if that operation is enabled by privileged software.

**Description:** GPR[rt] ← HWR[rd]

If access is allowed to the specified hardware register, the contents of the register specified by rd is sign-extended and loaded into general register rt. Access control for each register is selected by the bits in the coprocessor 0 HWREna register.

The available hardware registers, and the encoding of the rd field for each, are shown in Table 3-31.

<table>
<thead>
<tr>
<th>Table 3-31 Hardware Register List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Number (rd Value)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>4-28</td>
</tr>
<tr>
<td>29</td>
</tr>
<tr>
<td>30-31</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 HWREna register. If access is not allowed, a Reserved Instruction Exception is signaled.

Operation:
```c
    case rd
        0x00: temp ← sign_extend(EBaseCPUNum)
        0x01: temp ← sign_extend(SYNCI_StepSize())
        0x02: temp ← sign_extend(Count)
        0x03: temp ← sign_extend(CountResolution())
        0x30: temp ← sign_extend_if_32bit_op(Implementation-Dependent-Value)
        0x31: temp ← sign_extend_if_32bit_op(Implementation-Dependent-Value)
        otherwise: SignalException(ReservedInstruction)
    endcase
    GPR[rt] ← temp
```

function sign_extend_if_32bit_op(value)
    if (width(value) = 64) and Are64bitOperationsEnabled() then
        sign_extend_if_32bit_op ← value
    else
        sign_extend_if_32bit_op ← signExtend(value)
    endif
end sign_extend_if_32bit_op

Exceptions:
Reserved Instruction
Read GPR from Previous Shadow Set

### RDPGPR

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>RDPGPR</td>
<td>rt</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0100 00</td>
<td>01 010</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>rt</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** RDPGPR rd, rt

**Purpose:**
To move the contents of a GPR from the previous shadow set to a current GPR.

**Description:** GPR[rd] ← SGPR[SRSCTlPSS, rt]
The contents of the shadow GPR register specified by SRSCTlPSS (signifying the previous shadow set number) and rt (specifying the register number within that set) is moved to the current GPR rd.

**Restrictions:**
In implementations prior to Release 2 of the Architecture, this instruction resulted in a Reserved Instruction Exception.

**Operation:**
GPR[rd] ← SGPR[SRSCTlPSS, rt]

**Exceptions:**
Coprocessor Usable
Reserved Instruction
Reciprocal Approximation

Reciprocal Approximation

<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>fs</td>
<td>fd</td>
<td>RECIP</td>
<td>010001</td>
<td>00000</td>
<td>010101</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:  
RECIP.S fd, fs  
RECIP.D fd, fs  

MIPS64, MIPS32 Release 2  
MIPS64, MIPS32 Release 2

Purpose:
To approximate the reciprocal of an FP value (quickly)

Description:  
FPR[fd] ← 1.0 / 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.

The result of RECIP.D is UNPREDICTABLE if the processor is executing in 16 FP registers mode.

Operation:

\[ \text{StoreFPR(fd, fmt, 1.0 / valueFPR(fs, fmt))} \]
Reciprocal Approximation (cont.)

Exceptions:
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:
Inexact, Division-by-zero, Unimplemented Op, Invalid Op, Overflow, Underflow
Format: \texttt{ROTR rd, rt, sa} \\
SPECIAL Crypto, MIPS32 Release 2

Purpose:
To execute a logical right-rotate of a word by a fixed number of bits

Description: \( GPR[rd] \leftarrow GPR[rt] \leftrightarrow \text{(right)} \ sa \)
The contents of the low-order 32-bit word of GPR \( rt \) are rotated right; the word result is sign-extended and placed in GPR \( rd \). The bit-rotate amount is specified by \( sa \).

Restrictions:
If GPR \( rt \) does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:
\[
\begin{align*}
\text{if NotWordValue}(GPR[rt]) \text{ or } \\
((\text{ArchitectureRevision()} < 2) \text{ and } (\text{Config3SM} = 0)) \text{ then} \\
\text{UNPREDICTABLE} \\
\text{endif} \\
\text{s} \leftarrow sa \\
\text{temp} \leftarrow GPR[rt]_{s-1..0} \parallel GPR[rt]_{31..s} \\
GPR[rd] \leftarrow \text{sign\_extend}(\text{temp})
\end{align*}
\]

Exceptions:
Reserved Instruction
Rotate Word Right Variable

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>7</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>0000</td>
<td>R</td>
<td>SRLV</td>
<td>000110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 6 | 5 | 5 | 5 | 4 | 1 | 6 |

**Format:**  \texttt{ROTRV rd, rt, rs}

**Purpose:**
To execute a logical right-rotate of a word by a variable number of bits

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

The contents of the low-order 32-bit word of \text{GPR} \( rt \) are rotated right; the word result is sign-extended and placed in \text{GPR} \( rd \). The bit-rotate amount is specified by the low-order 5 bits of \text{GPR} \( rs \).

**Restrictions:**
If \text{GPR} \( rt \) does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

**Operation:**

\[
\text{if NotWordValue(GPR}[rt]) \text{ or (ArchitectureRevision() < 2) and (Config3SM = 0)) then UNPREDICTABLE}
\]

\[s \leftarrow \text{GPR}[rs]_{4..0}
\]

\[\text{temp} \leftarrow \text{GPR}[rt]_{s-1..0} \parallel \text{GPR}[rt]_{31..s}
\]

\[\text{GPR}[rd] \leftarrow \text{sign\_extend(temp)}
\]

**Exceptions:**
Reserved Instruction
Floating Point Round to Long Fixed Point

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>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>fs</td>
<td>fd</td>
<td>ROUND.L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010001</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>001000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**
- `ROUND.L.S` `fd, fs`
- `ROUND.L.D` `fd, fs`

**MIPS64, MIPS32 Release 2**

**Purpose:**
To convert an FP value to 64-bit fixed point, rounding to nearest.

**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 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 16 FP registers mode.

**Operation:**
\[
\text{StoreFPR}(fd, L, \text{ConvertFmt}(\text{ValueFPR}(fs, fmt), fmt, L))
\]
Floating Point Round to Long Fixed Point (cont.)

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Inexact, Unimplemented Operation, Invalid Operation, Overflow
Floating Point Round to Word Fixed Point

ROUND.W.fmt

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>fs</th>
<th>fd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>COP1</td>
<td>fmt</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>00000</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>001100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:
ROUND.W.S fd, fs  
ROUND.W.D fd, fs  
MIPS32

Purpose:
To convert an FP value to 32-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 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:
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
Floating Point Round to Word Fixed Point (cont.)

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Inexact, Unimplemented Operation, Invalid Operation, Overflow
Reciprocal Square Root Approximation

Format:

\[
\text{RSQRT}\cdot\text{S} \quad \text{fd, fs} \\
\text{RSQRT}\cdot\text{D} \quad \text{fd, fs}
\]

MIPS64, MIPS32 Release 2

Purpose:
To approximate the reciprocal of the square root of an FP value (quickly)

Description:

\[\text{FPR}[\text{fd}] \leftarrow \frac{1.0}{\sqrt{\text{FPR}[\text{fs}]}}\]

The reciprocal of the positive square root of the value in FPR \(\text{fs}\) is approximated and placed into FPR \(\text{fd}\). The operand and result are values in format \(\text{fmt}\).

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 \(\text{FCSR}\) rounding mode on the result is implementation dependent.

Restrictions:

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

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

The result of RSQRT.D is UNPREDICTABLE if the processor is executing in 16 FP registers mode.

Operation:

\[
\text{StoreFPR} (\text{fd}, \text{fmt}, 1.0 / \sqrt{\text{valueFPR}}(\text{fs}, \text{fmt}))
\]
Reciprocal Square Root Approximation (cont.)

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Inexact, Division-by-zero, Unimplemented Operation, Invalid Operation, Overflow, Underflow
### Store Byte

**Format:** \( SB \, rt, \, offset(base) \)  
**Purpose:** To store a byte to memory  
**Description:** \( memory[GPR[base] + offset] \leftarrow GPR[rt] \)  
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:**  
\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend}(offset) + \text{GPR}[base] \\
(pAddr, CCA) & \leftarrow \text{AddressTranslation} (\text{vAddr}, \text{DATA}, \text{STORE}) \\
pAddr & \leftarrow pAddr_{\text{pSIZE-1..3}} || (pAddr_{2..0} \text{ xor ReverseEndian}) \\
\text{bytesel} & \leftarrow pAddr_{2..0} \text{ xor BigEndianCPU} \\
\text{datadoubleword} & \leftarrow \text{GPR}[rt]_{63-8*\text{bytesel}..0} || 0^{8*\text{bytesel}} \\
\text{StoreMemory} \ (CCA, \ \text{BYTE}, \ \text{datadoubleword}, \ \text{pAddr}, \ \text{vAddr}, \ \text{DATA})
\end{align*}
\]
**Exceptions:** TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Watch
The LL and SC instructions provide primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The least-significant 32-bit word in GPR \texttt{rt} is conditionally 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 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 least-significant 32-bit word of GPR \texttt{rt} is stored into memory at the location specified by the aligned effective address.
- A 1, indicating success, is written into GPR \texttt{rt}.

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR \texttt{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 LL and SC, the SC 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 LL/SC.
- 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.)

The following conditions must be true or the result of the SC 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 cache-coherence algorithm 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:**

\[
vAddr \leftarrow \text{sign	extunderscore extend}(\text{offset}) + \text{GPR}[\text{base}]
\]
if \(vAddr_{1..0} \neq 0^2\) then
  \text{SignalException(AddressError)}
endif
(pAddr, CCA) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{STORE})
pAddr \leftarrow pAddr_{\text{PSIZE}-1..3} || (pAddr_{2..0} \text{ xor } (\text{ReverseEndian} || 0^2))
bytesel \leftarrow vAddr_{2..0} \text{ xor } (\text{BigEndianCPU} || 0^2)
datadoubleword \leftarrow \text{GPR}[rt]_{63-8\timesbytesel..0} || 0^8\timesbytesel
if LLbit then
  \text{StoreMemory}(CCA, \text{WORD}, \text{datadoubleword}, pAddr, vAddr, \text{DATA})
endif
GPR[rt] \leftarrow 0^63 || LLbit
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 Doubleword

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCD</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111100</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: $\text{SCD } rt, \text{ offset}(\text{base})$

Purpose:
To store a doubleword to memory to complete an atomic read-modify-write

Description: 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 LLD and SCD instructions provide primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The 64-bit doubleword in GPR $rt$ is conditionally 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 an effective address.

The SCD completes the RMW sequence begun by the preceding LLD instruction executed on the processor. If it would complete the RMW sequence atomically, the following occur:

- The 64-bit doubleword of GPR $rt$ is stored into 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 LLD and SCD, the SCD fails:

- A coherent store is completed by another processor or coherent I/O module into the block of synchronizable physical memory containing the doubleword. The size and alignment of the block is implementation dependent, but it is at least one doubleword and at most the minimum page size.
- An ERET instruction is executed.

If either of the following events occurs between the execution of LLD and SCD, the SCD may succeed or it may fail; success or failure is not predictable. Portable programs should not cause these events:

- A memory access instruction (load, store, or prefetch) is executed on the processor executing the LLD/SCD.
- The instructions executed starting with the LLD and ending with the SCD 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 two conditions must be true or the result of the SCD is UNPREDICTABLE:

- Execution of the SCD must be preceded by execution of an LLD instruction.
- An RMW sequence executed without intervening events that would cause the SCD to fail must use the same address in the LLD and SCD. The address is the same if the virtual address, physical address, and cache-coherence algorithm 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 any of the 3 least-significant bits of the address is non-zero, an Address Error exception occurs.

**Operation:**

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign}_\text{extend}(\text{offset}) + \text{GPR}[\text{base}] \\
\text{if } \text{vAddr}_{2..0} & \neq 0^3 \text{ then} \\
& \quad \text{SignalException(AddressError)} \\
\text{endif} \\
(\text{pAddr}, \text{CCA}) & \leftarrow \text{AddressTranslation}(\text{vAddr, DATA, STORE}) \\
\text{datadoubleword} & \leftarrow \text{GPR}[rt] \\
\text{if } \text{LLbit} \text{ then} \\
& \quad \text{StoreMemory}((\text{CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)}) \\
\text{endif} \\
\text{GPR}[rt] & \leftarrow 0^{63} \ || \ \text{LLbit}
\end{align*}
\]
Exceptions:
TLB Refill, TLB Invalid, TLB Modified, Address Error, Reserved Instruction, Watch

Programming Notes:
LLD and SCD are used to atomically update memory locations, as shown below.

L1:
  LLD  T1, (T0)  # load counter
  ADDI  T2, T1, 1  # increment
  SCD  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 LLD and SCD cause SCD to fail, so persistent exceptions must be avoided. Some examples of such exceptions are arithmetic operations that trap, system calls, and floating point operations that trap or require software emulation assistance.

LLD and SCD 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 Doubleword

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

| 6 | 5 | 5 | 16 |

#### Format: \( \text{SD r}, \text{offset(base)} \)

#### Purpose:
To store a doubleword to memory

#### Description:
\( \text{memory[GPR[base] + offset]} \leftarrow \text{GPR[rt]} \)

The 64-bit doubleword in GPR \( r_t \) 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 any of the 3 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

#### Operation:
\[
\text{vAddr} \leftarrow \text{sign_extend(\text{offset}) + GPR[base]}
\]
\[
\text{if vAddr}_{2..0} \neq 0^3 \text{ then}
\]
\[
\text{SignalException(AddressError)}
\]
\[
\text{endif}
\]
\[
(p\text{Addr}, \text{CCA}) \leftarrow \text{AddressTranslation (vAddr, DATA, STORE)}
\]
\[
\text{datadoubleword} \leftarrow \text{GPR[rt]}
\]
\[
\text{StoreMemory (CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)}
\]

#### Exceptions:
TLB Refill, TLB Invalid, TLB Modified, Address Error, Reserved Instruction, Watch
Software Debug Breakpoint

<table>
<thead>
<tr>
<th></th>
<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>011100</td>
<td>code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SDBBP</td>
<td>111111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** SDBBP code

**Purpose:**
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:**

```
If DebugDM = 0 then
    SignalDebugBreakpointException()
else
    SignalDebugModeBreakpointException()
endif
```

**Exceptions:**
Debug Breakpoint Exception
Debug Mode Breakpoint Exception
### Store Doubleword from Floating Point

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDC</td>
<td>ft</td>
<td>offset(base)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** SDC $ft$, offset(base)

**Purpose:**
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 EffectiveAddress$_{2..0} \neq 0$ (not doubleword-aligned).

**Operation:**

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign}_\text{extend}(\text{offset}) + \text{GPR[base]} \\
\text{if } \text{vAddr}_{2..0} & \neq 0^3 \text{ then} \\
& \quad \text{SignalException(AddressError)} \\
\text{endif} \\
(p\text{Addr}, \text{CCA}) & \leftarrow \text{AddressTranslation(vAddr, DATA, STORE)} \\
\text{datadoubleword} & \leftarrow \text{ValueFPR}(ft, \text{UNINTERPRETED_DOUBLEWORD}) \\
\text{StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)}
\end{align*}
\]

**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:**  
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:**  
vAddr ← sign_extend(offset) + GPR[base]  
if vAddr2..0 ≠ 0 then  
    SignalException(AddressError)  
endif  
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, STORE)  
datadoubleword ← CPR[2,rt,0]  
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)  

**Exceptions:**  
Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch
**Format:** SDL rt, offset(base)

**Purpose:**
To store the most-significant part of a doubleword to an unaligned memory address

**Description:**
memory[GPR[base] + offset] ← Some_Bytes_From 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 8 consecutive bytes forming a doubleword (DW) in memory, starting at an arbitrary byte boundary.

A part of DW, the most-significant 1 to 8 bytes, is in the aligned doubleword containing EffAddr. The same number of most-significant (left) bytes of GPR rt are stored into these bytes of DW.

The figure below illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of DW, 6 bytes, is located in the aligned doubleword containing the most-significant byte at 2. First, SDL stores the 6 most-significant bytes of the source register into these bytes in memory. Next, the complementary SDR instruction stores the remainder of DW.

**Figure 3-19 Unaligned Doubleword Store With SDL and SDR**
The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned doubleword—that is, the low 3 bits of the address ($vAddr_{2..0}$)—and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes stored for every combination of offset and byte ordering.

**Figure 3-20 Bytes Stored by an SDL Instruction**

Restrictions:

<table>
<thead>
<tr>
<th>Initial Memory Contents and Byte Offsets</th>
<th>Contents of Source Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>most — significance — least</td>
<td>i j k l m n o p</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 ← big-endian</td>
<td>A B C D E F G H</td>
</tr>
<tr>
<td>7 6 5 4 3 2 1 0 ← little-endian offset</td>
<td></td>
</tr>
</tbody>
</table>

Memory contents after instruction (shaded is unchanged)

<table>
<thead>
<tr>
<th>Big-endian byte ordering vAddr_{2..0}</th>
<th>Little-endian byte ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D E F G H 0 i j k l m n o A</td>
<td>i j k l m n A B</td>
</tr>
<tr>
<td>i A B C D E F G 1 i j k l m n A B</td>
<td>i j k l m A B C</td>
</tr>
<tr>
<td>i j A B C D E F 2 i j k l m A B C</td>
<td>i j k l A B C D</td>
</tr>
<tr>
<td>i j k A B C D E 3 i j k l A B C D</td>
<td>i j k A B C D E</td>
</tr>
<tr>
<td>i j k l A B C D 4 i j k l m A B C</td>
<td>i j A B C D E F</td>
</tr>
<tr>
<td>i j k l m A B C 5 i j k l m A B C</td>
<td>i A B C D E F G</td>
</tr>
<tr>
<td>i j k l m n A B 6 i j k l m n A B</td>
<td>A B C D E F G H</td>
</tr>
<tr>
<td>i j k l m n o A 7 i j k l m n o A</td>
<td></td>
</tr>
</tbody>
</table>
Operation:

\[
v\text{Addr} \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}]
\]
\[
(p\text{Addr}, \text{CCA}) \leftarrow \text{AddressTranslation}(v\text{Addr}, \text{DATA}, \text{STORE})
\]
\[
p\text{Addr} \leftarrow p\text{Addr}_{\text{PSIZE}-1..3} \vert\vert (p\text{Addr}_{2..0} \text{ xor ReverseEndian}^3)
\]
If BigEndianMem = 0 then
\[
p\text{Addr} \leftarrow p\text{Addr}_{\text{PSIZE}-1..3} \vert\vert 0^3
\]
endif
\[
\text{bytesel} \leftarrow \text{vAddr}_{2..0} \text{ xor BigEndianCPU}^3
\]
\[
\text{datadoubleword} \leftarrow 0^{36-8*\text{bytesel}} \vert\vert \text{GPR}[rt]_{63..56-8*\text{bytesel}}
\]
\[
\text{StoreMemory}(\text{CCA, byte, datadoubleword, pAddr, vAddr, DATA})
\]

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Reserved Instruction, Watch
Store Doubleword Right

**Format:**  \( SDR \ rt, \ offset(base) \)

**Purpose:**
To store the least-significant part of a doubleword to an unaligned memory address

**Description:**  \( \text{memory}[GPR[base] + offset] \leftarrow \text{Some}_\text{Bytes}_\text{From} \ GPR[rt] \)

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 least-significant of 8 consecutive bytes forming a doubleword \( (DW) \) in memory, starting at an arbitrary byte boundary.

A part of \( DW \), the least-significant 1 to 8 bytes, is in the aligned doubleword containing \( EffAddr \). The same number of least-significant (right) bytes of GPR \( rt \) are stored into these bytes of \( DW \).

The figure below illustrates this operation for big-endian byte ordering. The 8 consecutive bytes in 2..9 form an unaligned doubleword starting at location 2. A part of \( DW \), 2 bytes, is located in the aligned doubleword containing the least-significant byte at 9. First, \( SDR \) stores the 2 least-significant bytes of the source register into these bytes in memory. Next, the complementary \( SDL \) stores the remainder of \( DW \).

**Figure 3-21 Unaligned Doubleword Store With SDR and SDL**

<table>
<thead>
<tr>
<th>Memory</th>
<th>GPR 24</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>A B C D E F G H</td>
</tr>
</tbody>
</table>

After executing \( SDR \ $24,9($0) \)

Then after \( SDL \ $24,2($0) \)
The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned doubleword—that is, the low 3 bits of the address (vAddr2..0)—and the current byte ordering mode of the processor (big- or little-endian). Figure 3-22 shows the bytes stored for every combination of offset and byte-ordering.

**Figure 3-22 Bytes Stored by an SDR Instruction**

<table>
<thead>
<tr>
<th>Initial Memory contents and byte offsets</th>
<th>Contents of Source Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>most — significance — least</td>
<td>Source Register</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 ←big--endian most — significance — least</td>
<td></td>
</tr>
<tr>
<td>i j k l m n o p</td>
<td>A B C D E F G H</td>
</tr>
<tr>
<td>7 6 5 4 3 2 1 0 ←little-endian offset</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory contents after instruction (shaded is unchanged)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-endian byte ordering</td>
</tr>
<tr>
<td>vAddr2..0</td>
</tr>
<tr>
<td>Little-endian byte ordering</td>
</tr>
<tr>
<td>H j k l m n o p</td>
</tr>
<tr>
<td>G H k l m n o p</td>
</tr>
<tr>
<td>F G H l m n o p</td>
</tr>
<tr>
<td>E F G H m n o p</td>
</tr>
<tr>
<td>D E F G H n o p</td>
</tr>
<tr>
<td>C D E F G H o p</td>
</tr>
<tr>
<td>B C D E F G H p</td>
</tr>
<tr>
<td>A B C D E F G H</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>A B C D E F G H</td>
</tr>
<tr>
<td>B C D E F G H</td>
</tr>
<tr>
<td>C D E F G H o p</td>
</tr>
<tr>
<td>D E F G H n o p</td>
</tr>
<tr>
<td>E F G H m n o p</td>
</tr>
<tr>
<td>F G H l m n o p</td>
</tr>
<tr>
<td>G H k l m n o p</td>
</tr>
</tbody>
</table>

Restrictions:
Operation:

\[ \text{vAddr} \leftarrow \text{sign}\_\text{extend}(\text{offset}) + \text{GPR}\[\text{base}] \]

\[(\text{pAddr, CCA}) \leftarrow \text{AddressTranslation (vAddr, DATA, STORE)}\]

\[\text{pAddr} \leftarrow \text{pAddr}_{P\text{SIZE}-1..3} || (\text{pAddr}_{2..0} \oplus \text{ReverseEndian}^3)\]

\[\text{If BigEndianMem = 0 then}\]

\[\text{pAddr} \leftarrow \text{pAddr}_{P\text{SIZE}-1..3} || 0^3\]

\[\text{endif}\]

\[\text{bytesel} \leftarrow \text{vAddr}_{1..0} \oplus \text{BigEndianCPU}^3\]

\[\text{datadoubleword} \leftarrow \text{GPR}[rt]_{63..8*\text{bytesel}} || 0^8*\text{bytesel}\]

\[\text{StoreMemory (CCA, DOUBLEWORD-byte, datadoubleword, pAddr, vAddr, DATA)}\]

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Reserved Instruction, Watch
Store Doubleword Indexed from Floating Point

**Format:**  
SDXC1 fs, index(base)

**MIPS64**  
MIPS32 Release 2

**Purpose:**  
To store a doubleword from an FPR to memory (GPR+GPR addressing)

**Description:**  
memory[GPR[base] + GPR[index]] ← 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 EffectiveAddress2..0 ≠ 0 (not doubleword-aligned).

**Operation:**  
1. \[\text{vAddr} ← \text{GPR[base]} + \text{GPR[index]}\]
2. if \(\text{vAddr}_2..0 \neq 0\) then
   1. SignalException(AddressError)
3. endif
4. \((\text{pAddr}, \text{CCA}) ← \text{AddressTranslation(vAddr, DATA, STORE)}\)
5. datadoubleword ← ValueFPR(ft, UNINTERPRETED_DOUBLEWORD)
6. \(\text{StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)}\)

**Exceptions:**  
TLB Refill, TLB Invalid, TLB Modified, Coprocessor Usable, Address Error, Reserved Instruction, Watch.
### Sign-Extend Byte

**SEB**

#### Format:

```
seb rd, rt
```

#### Purpose:

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.

If GPR `rt` does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

#### Operation:

```
if NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
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 <code>rx,ry</code></td>
<td>Zero-Extend Byte</td>
<td>ANDI <code>rx,ry,0xFF</code></td>
</tr>
<tr>
<td>ZEH <code>rx,ry</code></td>
<td>Zero-Extend Halfword</td>
<td>ANDI <code>rx,ry,0xFFFF</code></td>
</tr>
<tr>
<td>SEW <code>rx,ry</code></td>
<td>Sign-Extend Word</td>
<td>SLL <code>rx,ry,0</code></td>
</tr>
<tr>
<td>ZEW <code>rx,rx</code></td>
<td>Zero-Extend Word</td>
<td>DINSP32 <code>rx,r0,32,32</code></td>
</tr>
</tbody>
</table>

1. The equivalent instruction uses `rx` for both source and destination, so the expected instruction is limited to one register.
**Sign-Extend Halfword**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL3</td>
<td>011111</td>
<td>0</td>
<td>00000</td>
<td>rt</td>
<td>rd</td>
<td>SEH</td>
<td>11000</td>
<td>BSHFL</td>
<td>100000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

**Format:** seh rd, rt

**Purpose:**
To sign-extend the least significant halfword of GPR rt and store the value into GPR rd.

**Description:**
\[ \text{GPR}[rd] \leftarrow \text{SignExtend}(\text{GPR}[rt]_{15..0}) \]

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.

If GPR rt does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

**Operation:**
\[
\begin{align*}
&\text{if NotWordValue(GPR[rt]) then} \\
&\quad \text{UNPREDICTABLE} \\
&\text{endif} \\
&\text{GPR[rd] }\leftarrow \text{sign\_extend(GPR[rt]_{15..0})}
\end{align*}
\]

**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:

```mips
lw t0, 0(a1) /* 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 */
```

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>
<tr>
<td>SEW rx,ry</td>
<td>Sign-Extend Word</td>
<td>SLL rx,ry,0</td>
</tr>
<tr>
<td>ZEW rx,rx₁</td>
<td>Zero-Extend Word</td>
<td>DINSP32 rx,r0,32,32</td>
</tr>
</tbody>
</table>

1. The equivalent instruction uses rx for both source and destination, so the expected instruction is limited to one register.
Store Halfword

**Format:** \( \text{SH } rt, \text{offset(base)} \)  

**Purpose:**
To store a halfword to memory

**Description:** memory\( [\text{GPR[base]} + \text{offset}] \) \( \leftarrow \) \( \text{GPR[rt]} \)

The least-significant 16-bit halfword of register \( rt \) 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:**
\[
\text{vAddr} \leftarrow \text{sign} \_\text{extend}(\text{offset}) + \text{GPR[base]}
\]
\[
\text{if vAddr}_0 \neq 0 \text{ then }
\text{SendException(AddressError)}
\text{endif}
\]
\[
(\text{pAddr, CCA}) \leftarrow \text{AddressTranslation(vAddr, DATA, STORE)}
\]
\[
\text{pAddr} \leftarrow \text{pAddr}_{\text{PSIZE}-1..3} || (\text{pAddr}_{12..0} \text{xor (ReverseEndian}^2 || 0))
\]
\[
\text{bytesel} \leftarrow \text{vAddr}_{12..0} \text{xor (BigEndianCPU}^2 || 0)
\]
\[
\text{datadoubleword} \leftarrow \text{GPR[rt]}_{63-8*\text{bytesel}..0} || 0^8*\text{bytesel}
\]
\[
\text{StoreMemory(CCA, HALFWORD, datadoubleword, pAddr, vAddr, DATA)}
\]

**Exceptions:**
TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch
<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>00000</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>SLL</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \texttt{SLL \textit{rd}, \textit{rt}, \textit{sa}}  

**MIPS32**

**Purpose:**  
To left-shift a word by a fixed number of bits

**Description:** \texttt{GPR[\textit{rd}] \leftarrow GPR[\textit{rt}] \ll \textit{sa}}

The contents of the low-order 32-bit word of GPR \textit{rt} are shifted left, inserting zeros into the emptied bits; the word result is sign-extended and placed in GPR \textit{rd}. The bit-shift amount is specified by \textit{sa}.

**Restrictions:**
None

**Operation:**

\begin{align*}
  s & \leftarrow \textit{sa} \\
  \text{temp} & \leftarrow GPR[\textit{rt}]_{(31-\text{s})}..0 \mid 0^s \\
  \text{GPR[\textit{rd}]} & \leftarrow \text{sign\_extend} (\text{temp})
\end{align*}

**Exceptions:**
None

**Programming Notes:**

Unlike nearly all other word operations, the SLL input operand does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign-extended into a 64-bit destination register; this instruction with a zero shift amount truncates a 64-bit value to 32 bits and sign-extends it.

\texttt{SLL \textit{r0}, \textit{r0}, 0}, expressed as \texttt{NOP}, is the assembly idiom used to denote no operation.

\texttt{SLL \textit{r0}, \textit{r0}, 1}, expressed as \texttt{SSNOP}, is the assembly idiom used to denote no operation that causes an issue break on superscalar processors.
Shift Word Left Logical Variable  

**Format:**  
SLLV rd, rt, rs  

**Purpose:**  
To left-shift a word by a variable number of bits

**Description:**  
GPR[rd] ← GPR[rt] << 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 sign-extended and placed in GPR rd. The bit-shift amount is specified by the low-order 5 bits of GPR rs.

**Restrictions:** None

**Operation:**

\[ 
\begin{align*} 
s & \leftarrow GPR[rs]_{4..0} 
\text{temp} & \leftarrow GPR[rt]_{(31-s)..0} \mid \mid 0^s 
GPR[rd] & \leftarrow \text{sign\_extend}(\text{temp}) 
\end{align*} 
\]

**Exceptions:** None

**Programming Notes:**  
Unlike nearly all other word operations, the input operand does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign-extended into a 64-bit destination register; this instruction with a zero shift amount truncates a 64-bit value to 32 bits and sign-extends it.
Set on Less Than

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>SLT</td>
<td>101010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: \texttt{SLT \textit{rd}, \textit{rs}, \textit{rt}} \textbf{MIPS32}

Purpose:
To record the result of a less-than comparison

\textbf{Description:} \texttt{GPR[rd] \leftarrow (GPR[rs] < GPR[rt])}

Compare the contents of GPR \textit{rs} and GPR \textit{rt} as signed integers and record the Boolean result of the comparison in GPR \textit{rd}. If GPR \textit{rs} is less than GPR \textit{rt}, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

\textbf{Restrictions:}
None

\textbf{Operation:}
\begin{verbatim}
if GPR[rs] < GPR[rt] then
    GPR[rd] \leftarrow 0^{GPRLEN-1} | 1
else
    GPR[rd] \leftarrow 0^{GPRLEN}
endif
\end{verbatim}

Exceptions:
None
Set on Less Than Immediate

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

MIPS32

Purpose:
To record the result of a less-than comparison with a constant

Description: \texttt{GPR[rt] \leftarrow (GPR[rs] < \textit{immediate})}

Compare the contents of GPR \textit{rs} and the 16-bit signed \textit{immediate} as signed integers and record the Boolean result of the comparison in GPR \textit{rt}. If GPR \textit{rs} is less than \textit{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{verbatim}
if GPR[rs] < sign_extend(immediate) then
  GPR[rt] \leftarrow 0^{GPRLEN-1}|| 1
else
  GPR[rt] \leftarrow 0^{GPRLEN}
endif
\end{verbatim}

Exceptions:
None
Set on Less Than Immediate Unsigned

<table>
<thead>
<tr>
<th>Format:</th>
<th>SLTIU rt, rs, immediate</th>
</tr>
</thead>
</table>

**Purpose:**
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:**

```c
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**

**SLTU**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>0000</td>
<td>SLTU</td>
<td>101011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \texttt{SLTU \textit{rd}, \textit{rs}, \textit{rt}} \quad \text{MIPS32}

**Purpose:**
To record the result of an unsigned less-than comparison

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

Compare the contents of GPR \textit{rs} and GPR \textit{rt} as unsigned integers and record the Boolean result of the comparison in GPR \textit{rd}. If GPR \textit{rs} is less than GPR \textit{rt}, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:**
None

**Operation:**
\[
\begin{align*}
\text{if } (0 \| GPR[rs]) < (0 \| GPR[rt]) \text{ then } \\
\quad \text{GPR}[rd] &\leftarrow 0^{\text{GPRLEN}-1} \| 1 \\
\text{else } \\
\quad \text{GPR}[rd] &\leftarrow 0^{\text{GPRLEN}} \\
\end{align*}
\]

**Exceptions:**
None
Floating Point Square Root

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>SQRT</th>
<th>000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

SQRT.S fd, fs
SQRT.D fd, fs

**MIPS32**

**Purpose:**

To compute the square root of an FP value

**Description:**

FPR[fd] ← SQRT(FPR[fs])

The square root of the value in FPR fs is calculated to infinite precision, rounded according to the current rounding mode in FCSR, and placed into FPR fd. The operand and result are values in format fmt.

If the value in FPR fs corresponds to –0, the result is –0.

**Restrictions:**

If the value in FPR 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 FPR becomes UNPREDICTABLE.

**Operation:**

StoreFPR(fd, fmt, SquareRoot(ValueFPR(fs, fmt)))

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Invalid Operation, Inexact, Unimplemented Operation
Shift Word Right Arithmetic  SRA

<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>5</th>
<th>5</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>SRA</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format: SRA rd, rt, sa

Purpose:
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 sign-extended and placed in GPR rd. The bit-shift amount is specified by sa.

Restrictions:
On 64-bit processors, if GPR rt does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:
```c
if NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
s ← sa
temp ← (GPR[rt]31)s || GPR[rt]31..s
GPR[rd]← sign_extend(temp)
```

Exceptions: None
### Shift Word Right Arithmetic Variable

**MIPS64® Architecture For Programmers Volume II, Revision 2.50**

**SRAV**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>SRAV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** SRAV rd, rt, rs

**MIPS32**

**Purpose:**
To execute an arithmetic right-shift of a word by a variable number of bits

**Description:**
GPR[rd] ← GPR[rt] >> rs  (arithmetic)

The contents of the low-order 32-bit word of GPR rt are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is sign-extended and placed in GPR rd. The bit-shift amount is specified by the low-order 5 bits of GPR rs.

**Restrictions:**
On 64-bit processors, if GPR rt does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

**Operation:**

```plaintext
if NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif

s ← GPR[rs]4..0
temp ← (GPR[rt]31)s || GPR[rt]31..s
GPR[rd] ← sign_extend(temp)
```

**Exceptions:**
None
### Shift Word Right Logical

**Format:**  
SRL rd, rt, sa

**MIPS32**

**Purpose:**  
To execute a logical right-shift of a word by a fixed number of bits

**Description:**  
GPR[rd] ← GPR[rt] >> sa  (logical)

The contents of the low-order 32-bit word of GPR rt are shifted right, inserting zeros into the emptied bits; the word result is sign-extended and placed in GPR rd. The bit-shift amount is specified by sa.

**Restrictions:**  
On 64-bit processors, if GPR rt does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

**Operation:**

```
if NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
s ← sa
temp ← 0^s || GPR[rt]31...s
GPR[rd] ← sign_extend(temp)
```

**Exceptions:**  
None
Shift Word Right Logical Variable  

**SRLV**

### Format:  
SRLV rd, rt, rs  

### MIPS32

### Purpose:  
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 sign-extended and placed in GPR rd. The bit-shift amount is specified by the low-order 5 bits of GPR rs.

### Restrictions:  
On 64-bit processors, if GPR rt does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

### Operation:  
if NotWordValue(GPR[rt]) then  
    UNPREDICTABLE  
endif  

s ← GPR[rs]4..0  
temp ← 0s || GPR[rt]31..s  
GPR[rd] ← sign_extend(temp)

### Exceptions:  
None
Superscalar No Operation

<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>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>SLL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0001</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** SSNOP

**Purpose:**
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:

```assembly
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} \hspace{1cm} 
**MIPS32**

**Purpose:**
To subtract 32-bit integers. If overflow occurs, then trap

**Description:** \texttt{GPR[rd] \leftarrow GPR[rs] - GPR[rt]}

The 32-bit word value in GPR \textit{rt} is subtracted from the 32-bit value in GPR \textit{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 sign-extended and placed into GPR \textit{rd}.

**Restrictions:**

On 64-bit processors, if either GPR \textit{rt} or GPR \textit{rs} does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is \texttt{UNPREDICTABLE}.

**Operation:**

```plaintext
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif

temp \leftarrow (GPR[rs]_{31} \mid GPR[rs]_{31..0}) - (GPR[rt]_{31} \mid GPR[rt]_{31..0})
if temp_{32} \neq temp_{31} then
    SignalException(IntegerOverflow)
else
    GPR[rd] \leftarrow \text{sign_extend}(temp_{31..0})
endif
```

**Exceptions:**
Integer Overflow

**Programming Notes:**

SUBU performs the same arithmetic operation but does not trap on overflow.
Floating Point Subtract

SUB.fmt

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>SUB</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000001</td>
</tr>
</tbody>
</table>

**Format:**
- SUB.S fd, fs, ft
- SUB.D fd, fs, ft
- SUB.PS fd, fs, ft

**Purpose:**
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 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 SUB.PS is UNPREDICTABLE if the processor is executing in 16 FP registers mode.

**Operation:**

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

**CPU Exceptions:**
Coprocessor Unusable, Reserved Instruction

**FPU Exceptions:**
Inexact, Overflow, Underflow, Invalid Op, Unimplemented Op
**Subtract Unsigned Word**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>00000</td>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>00000</td>
<td>SUBU</td>
</tr>
</tbody>
</table>

**Format:** SUBU rd, rs, rt

**Purpose:**
To subtract 32-bit integers

**Description:**
GPR[rd] ← GPR[rs] - 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 sign-extended and placed into GPR rd.

No integer overflow exception occurs under any circumstances.

**Restrictions:**

On 64-bit processors, if either GPR rt or GPR rs does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

**Operation:**

```c
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif

temp ← GPR[rs] - GPR[rt]
GPR[rd] ← sign_extend(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:**
To store a doubleword from an FPR to memory (GPR+GPR addressing) ignoring alignment

**Description:**
```plaintext
memory[(GPR[base] + GPR[index])_{PSIZE-1..3}] \leftarrow 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 16 FP registers mode.

**Operation:**
```plaintext
vAddr \leftarrow (GPR[base]+GPR[index])_{63..3} || 0^3
(pAddr, CCA) \leftarrow AddressTranslation(vAddr, DATA, STORE)
datadoubleword \leftarrow ValueFPR(ft, UNINTERPRETED_DOUBLEWORD)
StoreMemory(CCA, DOUBLEWORD, datadoubleword, pAddr, vAddr, DATA)
```

**Exceptions:**
Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Watch
Store Word

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW</td>
<td>101011</td>
<td>base</td>
<td>rt</td>
<td></td>
<td>offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \( SW \ rt, \ offset(base) \)

**Purpose:**
To store a word to memory

**Description:** \( memory[\text{GPR[base]} + \ offset] \leftarrow \text{GPR[rt]} \)

The least-significant 32-bit word of GPR \( rt \) is stored in memory at the location specified by the aligned effective address. The 16-bit signed \( offset \) is added to the contents of GPR \( base \) to form the effective address.

**Restrictions:**
The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

**Operation:**

\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR[base]} \\
\text{if} \ v\text{Addr}_{1..0} \neq 0^2 \text{ then} \\
& \quad \text{SignalException(AddressError)} \\
\text{endif} \\
(\text{pAddr}, \text{CCA}) & \leftarrow \text{AddressTranslation}(\text{vAddr}, \text{DATA}, \text{STORE}) \\
\text{pAddr} & \leftarrow \text{pAddr}_{\text{PSIZE}-1..3} \ || \ (\text{pAddr}_{2..0} \text{ xor (ReverseEndian}} \ || \ 0^2)) \\
\text{bytesel} & \leftarrow \text{vAddr}_{2..0} \text{ xor (BigEndianCPU} \ || \ 0^2) \\
\text{datadoubleword} & \leftarrow \text{GPR[rt]}_{63-8*\text{bytesel}..0} \ || \ 0^8*\text{bytesel} \\
\text{StoreMemory}(\text{CCA}, \text{WORD}, \text{datadoubleword}, \text{pAddr}, \text{vAddr}, \text{DATA})
\end{align*}
\]

**Exceptions:**
TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch
### Store Word from Floating Point

**Format:**  
SWC1 ft, offset(base)  

**MIPS32**

**Purpose:**  
To store a word from an FPR to memory  

**Description:**  
\[ \text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{FPR}[\text{ft}] \]  
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 \( \text{EffectiveAddress}_{1,0} \neq 0 \) (not word-aligned).

**Operation:**  
\[
\begin{align*}
\text{vAddr} & \leftarrow \text{sign} \_\text{extend}(\text{offset}) + \text{GPR}[\text{base}] \\
\text{if } \text{vAddr}_{1,0} & \neq 0^{3} \text{ then} \\
\quad \text{SignalException(AddressError)} \\
\text{endif} \\
(p\text{Addr}, \text{CCA}) & \leftarrow \text{AddressTranslation}(\text{vAddr}, \text{DATA}, \text{STORE}) \\
\text{pAddr} & \leftarrow \text{pAddr}_{\text{PSIZE}-1,3} \mid (\text{pAddr}_{2,0} \text{ xor} (\text{ReverseEndian} \mid \mid 0^{2})) \\
\text{bytesel} & \leftarrow \text{vAddr}_{2,0} \text{ xor} (\text{BigEndianCPU} \mid \mid 0^{2}) \\
\text{datadoubleword} & \leftarrow \text{ValueFPR}(\text{ft}, \text{UNINTERPRETED\_WORD}) \mid \mid 0^{8}\text{bytesel} \\
\text{StoreMemory}(\text{CCA}, \text{WORD}, \text{datadoubleword}, \text{pAddr}, \text{vAddr}, \text{DATA})
\end{align*}
\]

**Exceptions:**  
Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

<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>SWC1</td>
<td>base</td>
<td>ft</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111001</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Store Word from Coprocessor 2

**Format:** SWC2 rt, offset(base)

**Purpose:**
To store a word from a COP2 register to memory

**Description:** memory[GPR[base] + offset] ← CPR[2,rt,0]

The low 32-bit word from COP2 (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 EffectiveAddress\( _{1,0} \neq 0 \) (not word-aligned).

**Operation:**

\[
\begin{align*}
vAddr & \leftarrow \text{sign}\_\text{extend}(\text{offset}) + \text{GPR[base]} \\
& \text{if } vAddr_{2..0} \neq 0^3 \text{ then} \\
& \quad \text{SignalException(AddressError)} \\
& \text{endif} \\
(pAddr, CCA) & \leftarrow \text{AddressTranslation}(vAddr, \text{DATA, STORE}) \\
pAddr & \leftarrow pAddr_{\text{PSIZE}-1..3} || (pAddr_{2..0} \text{ xor (ReverseEndian || 02)}) \\
bytesel & \leftarrow vAddr_{2..0} \text{ xor (BigEndianCPU || 02)} \\
datadoubleword & \leftarrow CPR[2,rt,0]_{63-8*bytesel..0} || 8*bytesel \\
& \text{StoreMemory}(CCA, \text{WORD, datadoubleword, pAddr, vAddr, DATA})
\end{align*}
\]

**Exceptions:**
Coprocessor Unusable, Reserved Instruction, TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

---

<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>SWC2</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111010</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Format:  \texttt{SWL \textit{rt}, offset(base)}

Purpose:
To store the most-significant part of a word to an unaligned memory address

Description: \texttt{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.

If GPR rt is a 64-bit register, the source word is the low word of the register.

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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3-23.png}
\caption{Unaligned Word Store Using SWL and SWR}
\end{figure}

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.
Figure 3-24 Bytes Stored by an SWL Instruction

Restrictions:
None

Operation:

\[
\text{vAddr} \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}]
\]

\[
\text{pAddr, CCA} \leftarrow \text{AddressTranslation}(\text{vAddr, DATA, STORE})
\]

\[
\text{pAddr} \leftarrow \text{pAddr}_{\text{PSIZE}-1..3} | | \text{pAddr}_{2..0} \text{ xor ReverseEndian}^3
\]

If BigEndianMem = 0 then

\[
\text{pAddr} \leftarrow \text{pAddr}_{\text{PSIZE}-1..2} | | 0^2
\]
endif

\[
\text{byte} \leftarrow \text{vAddr}_{1..0} \text{ xor BigEndianCPU}^2
\]

if (vAddr\_2 xor BigEndianCPU) = 0 then

\[
\text{datadoubleword} \leftarrow 0^{32} | | 0^{24-8*\text{byte}} | | \text{GPR}[\text{rt}]_{31..24-8*\text{byte}}
\]
else

\[
\text{datadoubleword} \leftarrow 0^{24-8*\text{byte}} | | \text{GPR}[\text{rt}]_{31..24-8*\text{byte}} | | 0^{32}
\]
endif

\[
\text{StoreMemory}(\text{CCA, byte, datadoubleword, pAddr, vAddr, DATA})
\]

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Watch
### Store Word Right (SWR)

**Format:** SWR rt, offset(base)  

**MIPS32**

**Purpose:**  
To store the least-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 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. The same number of the least-significant (right) bytes from the word in GPR rt are stored into these bytes of W.

If GPR rt is a 64-bit register, the source word is the low word of the register.

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 3-25 Unaligned Word Store Using SWR and SWL

<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></th>
</tr>
</thead>
<tbody>
<tr>
<td>SWR</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101110</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Word at byte 2 in memory, big-endian byte order, each mem byte contains its address

<table>
<thead>
<tr>
<th>least — significance — least</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Memory: Initial contents

GPR 24:

| A | B | C | D | E | F | G | H |

<table>
<thead>
<tr>
<th>least — significance — least</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

After executing SWR $24, 5($0)

<table>
<thead>
<tr>
<th>least — significance — least</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Then after SWL $24, 2($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 (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.

**Figure 3-26 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>0 1 2 3 ← big-endian</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</td>
</tr>
<tr>
<td></td>
<td>E F G H</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory contents after instruction (shaded is unchanged)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-endian byte ordering</td>
</tr>
<tr>
<td>H j k l</td>
</tr>
<tr>
<td>G H k l</td>
</tr>
<tr>
<td>F G H l</td>
</tr>
<tr>
<td>E F G H</td>
</tr>
</tbody>
</table>

**Restrictions:**

None

**Operation:**

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..3 || (pAddr2..0 xor ReverseEndian3)
If BigEndianMem = 0 then
  pAddr ← pAddrPSIZE-1..2 || 0^2
endif
byte ← vAddr1..0 xor BigEndianCPU^2
if (vAddr2 xor BigEndianCPU) = 0 then
datadoubleword ← 0^32 || GPR[rt]31-8*byte..0 || 0^8*byte
else
datadoubleword ← GPR[rt]31-8*byte..0 || 0^8*byte || 0^32
endif
StoreMemory(CCA, WORD-byte, datadoubleword, pAddr, vAddr, DATA)

**Exceptions:**

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error, Watch
Store Word Indexed from Floating Point

Format: \texttt{SWXC1 fs, index(base)}

MIPS64
MIPS32 Release 2

Purpose:
To store a word from an FPR to memory (GPR+GPR addressing)

Description: memory\[GPR[base] + GPR[index]\] ← 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).

Operation:
\[
\begin{align*}
\text{vAddr} & \leftarrow \text{GPR[base]} + \text{GPR[index]} \\
\text{if vAddr}_{1,0} \neq 0^3 \text{ then} & \\
& \quad \text{SignalException(AddressError)} \\
\text{endif} \\
\text{(pAddr, CCA)} & \leftarrow \text{AddressTranslation(vAddr, DATA, STORE)} \\
\text{pAddr} & \leftarrow \text{pAddr}_{PSIZE-1..3} || (\text{pAddr}_{2..0} \text{xor (ReverseEndian} || 0^2)) \\
\text{bytesel} & \leftarrow \text{vAddr}_{2..0} \text{xor (BigEndianCPU} || 0^2) \\
\text{datadoubleword} & \leftarrow \text{ValueFPR(ft, UNINTERPRETED_WORD) || 0^8*bytesel} \\
\text{StoreMemory(CCA, WORD, datadoubleword, pAddr, vAddr, DATA)}
\end{align*}
\]

Exceptions:
TLB Refill, TLB Invalid, TLB Modified, Address Error, Reserved Instruction, Coprocessor Unusable, Watch
Synchronize Shared Memory

**SYNC**

<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>000000</td>
<td>00000000000000</td>
<td>stype</td>
<td>SYNC</td>
<td>001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** SYNC (stype = 0 implied)

**Purpose:**
To order loads and stores.

**Description:**

*Simple Description:*

- SYNC affects only uncached and cached coherent loads and stores. The loads and stores that occur before the SYNC must be completed before the loads and stores 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.
- SYNC 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 SYNC is required on some implementations on entry to and exit from Debug Mode to guarantee that memory affects are handled correctly.

*Detailed Description:*

- When the *stype* field has a value of zero, every synchronizable load and store that occurs in the instruction stream before the SYNC instruction must be globally performed before any synchronizable load or store that occurs after the SYNC can be performed, with respect to any other processor or coherent I/O module.
- SYNC does not guarantee the order in which instruction fetches are performed. The *stype* values 1-31 are reserved for future extensions to the architecture. A value of zero will always be defined such that it performs all defined synchronization operations. Non-zero values may be defined to remove some synchronization operations. As such, software should never use a non-zero value of the *stype* field, as this may inadvertently cause future failures if non-zero values remove synchronization operations.
Synchronize Shared Memory (cont.)

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*. 
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:
\[\text{SyncOperation(stype)}\]

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 previously 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. The effect of SYNC on reads or writes to memory caused by privileged implementation-specific instructions, such as CACHE, also 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
```

Prefetch operations have no effect detectable by User-mode programs, so ordering the effects of prefetch operations is not meaningful.

The code fragments above shows 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.
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>base</td>
<td>SYNCI</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001</td>
<td>6</td>
<td>11111</td>
<td>5</td>
<td></td>
<td>5</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

**Format:** \texttt{SYNCI offset(base)}

**Purpose:**
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.

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.

**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 only on the current processor. It doesn’t not affect the caches on other processors in a multi-processor 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.
<table>
<thead>
<tr>
<th>Operation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>vaddr ← GPR[base] + sign_extend(offset)</td>
</tr>
<tr>
<td>SynchronizeCacheLines(vaddr)/* Operate on all caches */</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved Instruction Exception (Release 1 implementations only)</td>
</tr>
<tr>
<td>TLB Refill Exception</td>
</tr>
<tr>
<td>TLB Invalid Exception</td>
</tr>
<tr>
<td>Address Error Exception</td>
</tr>
<tr>
<td>Cache Error Exception</td>
</tr>
<tr>
<td>Bus Error Exception</td>
</tr>
</tbody>
</table>
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.

```c
/*
 * 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
 */

addu   a1, a0, a1 /* Calculate end address + 1 */
    /* (daddu for 64-bit addressing) */
rdhwr  v0, HW_SYNCI_Step /* Get step size for SYNCI from new */
    /* Release 2 instruction */
beq    v0, zero, 20f /* If no caches require synchronization, */
    /* branch around */
        10: synci 0(a0) /* Synchronize all caches around address */
sltu   v1, a0, a1 /* Compare current with end address */
bne    v1, zero, 10b /* Branch if more to do */
addu   a0, a0, v0 /* Add step size in delay slot */
    /* (daddu for 64-bit addressing) */
sync   /* Clear memory hazards */
    20: jr.hb ra /* Return, clearing instruction hazards */
    nop
```
System Call

**Format:** SYSCALL

**Purpose:**
To cause a System Call exception

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

**Restrictions:**
None

**Operation:**
```
SignalException(SystemCall)
```

**Exceptions:**
System Call
**Trap if Equal**

**TEQ**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>code</td>
<td>TEQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td>110100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** TEQ rs, rt

**MIPS32**

**Purpose:**
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:**

if GPR[rs] = GPR[rt] then
    SignalException(Trap)
endif

**Exceptions:**
Trap
## Trap if Equal Immediate

### Format:
```
TEQI rs, immediate
```

### MIPS32

### Purpose:
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

<table>
<thead>
<tr>
<th></th>
<th>REGIMM</th>
<th>rs</th>
<th>TEQI</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000001</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

---

Format: TEQI `rs`, `immediate`  
Purpose: 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

---

356  
MIPS64® Architecture For Programmers Volume II, Revision 2.50  
Copyright © 2001-2003,2005 MIPS Technologies Inc. All rights reserved.
Trap if Greater or Equal

| Format: | TGE rs, rt |
| Purpose: | To compare GPRs and do a conditional trap |
| Description: | if GPR[rs] ≥ GPR[rt] then Trap |
| Restrictions: | None |
| Operation: | if GPR[rs] ≥ GPR[rt] then \( \text{SignalException(Trap)} \) endif |
| Exceptions: | Trap |
**Trap if Greater or Equal Immediate**

**Format:** TGEI rs, immediate

**Purpose:**
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 greater than or 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 Immediate Unsigned

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>TGEIU</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>

**Format:** TGEIU rs, immediate

**Purpose:** 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 greater than or equal to 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 Greater or Equal Unsigned

Format: \texttt{TGEU\,rs,\,rt}

Purpose:
To compare GPRs and do a conditional trap

Description: if $\text{GPR}[rs] \geq \text{GPR}[rt]$ 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 \textit{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 (0 || GPR[rs]) \geq (0 || GPR[rt]) then
    SignalException(Trap)
endif
\end{verbatim}

Exceptions:
Trap
Probe TLB for Matching Entry

<table>
<thead>
<tr>
<th>COP0</th>
<th>CO</th>
<th>0</th>
<th>TLBP</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>1</td>
<td>000 0000 0000 0000 0000</td>
<td>001000</td>
</tr>
</tbody>
</table>

**Format:** TLBP

**Purpose:**
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.

**Restrictions:**
If access to Coprocessor 0 is not enabled, a Coprocessor Un usable Exception is signaled.

**Operation:**

\[
\text{Index} \leftarrow 1 \bigg|\bigg| \text{UNPREDICTABLE}^{31}
\]

\[
\text{for } i \text{ in } 0 \ldots \text{TLBEntries-1}
\]

\[
\text{if } ((\text{TLB}[i]_{\text{VPN}2} \text{ and not } (\text{TLB}[i]_{\text{Mask}})) = (\text{EntryHi}_{\text{VPN}2} \text{ and not } (\text{TLB}[i]_{\text{Mask}}))) \text{ and } \\
(\text{TLB}[i]_{R} = \text{EntryHi}_{R}) \text{ and } \\
((\text{TLB}[i]_{G} = 1) \text{ or } (\text{TLB}[i]_{\text{ASID}} = \text{EntryHi}_{\text{ASID}})) \text{ then}
\]

\[
\text{Index} \leftarrow i
\]

**Exceptions:**
Coprocessor Un usable
Machine Check
**Format:** TLBR

**MIPS32**

**Purpose:**

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. 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:**

\[
i \leftarrow \text{Index} \\
\text{if } i > (\text{TLBEntries} - 1) \text{ then UNDEFINED} \\
\text{endif} \\
\text{PageMaskMask } \leftarrow \text{TLB}[i]_{\text{Mask}} \\
\text{EntryHi } \leftarrow \text{TLB}[i]_R \| 0_{\text{Fill}} \| (\text{TLB}[i]_{\text{VPN2}} \text{ and not TLB}[i]_{\text{Mask}}) \| \text{# Masking implementation dependent} \\
\text{PageMaskMask } \leftarrow \text{TLB}[i]_{\text{ASID}} \\
\text{EntryLo1 } \leftarrow 0_{\text{Fill}} \| (\text{TLB}[i]_{\text{PFN1}} \text{ and not TLB}[i]_{\text{Mask}}) \| \text{# Masking implementation dependent} \\
\text{EntryLo0 } \leftarrow 0_{\text{Fill}} \| (\text{TLB}[i]_{\text{PFN0}} \text{ and not TLB}[i]_{\text{Mask}}) \| \text{# Masking implementation dependent} \\
\text{Exceptions:} \\
\text{Coprocessor Unusable} \\
\text{Machine Check}
Write Indexed TLB Entry

<table>
<thead>
<tr>
<th></th>
<th>COP0</th>
<th>INDEX</th>
<th>Page Mask</th>
<th>Entry Lo0</th>
<th>Entry Lo1</th>
<th>Entry Hi</th>
<th>TLBWI</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>010000</td>
<td>0000000000000000</td>
<td>000010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** TLBW

**MIPS32**

**Purpose:**
To write a TLB entry indexed by the Index register.

**Description:**
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 TLBW. 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:**
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.
Write Indexed TLB Entry

Operation:

\[
i \leftarrow \text{Index} \\
\text{TLB}[i]_{\text{Mask}} \leftarrow \text{PageMask}_{\text{Mask}} \\
\text{TLB}[i]_{R} \leftarrow \text{EntryHi}_{R} \\
\text{TLB}[i]_{\text{VPN2}} \leftarrow \text{EntryHi}_{\text{VPN2}} \text{ and not PageMask}_{\text{Mask}} \# \text{Implementation dependent} \\
\text{TLB}[i]_{\text{ASID}} \leftarrow \text{EntryHi}_{\text{ASID}} \\
\text{TLB}[i]_{G} \leftarrow \text{EntryLo1}_G \text{ and EntryLo0}_G \\
\text{TLB}[i]_{\text{PFN1}} \leftarrow \text{EntryLo1}_{\text{PFN}} \text{ and not PageMask}_{\text{Mask}} \# \text{Implementation dependent} \\
\text{TLB}[i]_{C1} \leftarrow \text{EntryLo1}_C \\
\text{TLB}[i]_{D1} \leftarrow \text{EntryLo1}_D \\
\text{TLB}[i]_{V1} \leftarrow \text{EntryLo1}_V \\
\text{TLB}[i]_{\text{PFN0}} \leftarrow \text{EntryLo0}_{\text{PFN}} \text{ and not PageMask}_{\text{Mask}} \# \text{Implementation dependent} \\
\text{TLB}[i]_{C0} \leftarrow \text{EntryLo0}_C \\
\text{TLB}[i]_{D0} \leftarrow \text{EntryLo0}_D \\
\text{TLB}[i]_{V0} \leftarrow \text{EntryLo0}_V
\]

Exceptions:

Coprocessor Unusable

Machine Check
**Write Random TLB Entry**

<table>
<thead>
<tr>
<th>COP0</th>
<th>CO</th>
<th>0</th>
<th>000 0000 0000 0000 0000</th>
<th>TLBWR</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>1</td>
<td></td>
<td>0000 0000 0000 0000 0000</td>
<td>000110</td>
</tr>
</tbody>
</table>

**Format:** TLBWR

**MIPS32**

**Purpose:**
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.
Write Random TLB Entry

Operation:

\[
i \leftarrow \text{Random} \\
\text{TLB}[i]_{\text{Mask}} \leftarrow \text{PageMask}_{\text{Mask}} \\
\text{TLB}[i]_{\text{R}} \leftarrow \text{EntryHi}_{\text{R}} \\
\text{TLB}[i]_{\text{VPN2}} \leftarrow \text{EntryHi}_{\text{VPN2}} \text{ and not PageMask}_{\text{Mask}} \quad \# \text{Implementation dependent} \\
\text{TLB}[i]_{\text{ASID}} \leftarrow \text{EntryHi}_{\text{ASID}} \\
\text{TLB}[i]_{\text{G}} \leftarrow \text{EntryLo}_1\text{G} \text{ and EntryLo}_0\text{G} \\
\text{TLB}[i]_{\text{PFN1}} \leftarrow \text{EntryLo}_1\text{PFN} \text{ and not PageMask}_{\text{Mask}} \quad \# \text{Implementation dependent} \\
\text{TLB}[i]_{\text{C1}} \leftarrow \text{EntryLo}_1\text{C} \\
\text{TLB}[i]_{\text{D1}} \leftarrow \text{EntryLo}_1\text{D} \\
\text{TLB}[i]_{\text{V1}} \leftarrow \text{EntryLo}_1\text{V} \\
\text{TLB}[i]_{\text{PFN0}} \leftarrow \text{EntryLo}_0\text{PFN} \text{ and not PageMask}_{\text{Mask}} \quad \# \text{Implementation dependent} \\
\text{TLB}[i]_{\text{C0}} \leftarrow \text{EntryLo}_0\text{C} \\
\text{TLB}[i]_{\text{D0}} \leftarrow \text{EntryLo}_0\text{D} \\
\text{TLB}[i]_{\text{V0}} \leftarrow \text{EntryLo}_0\text{V}
\]

Exceptions:

Coprocessor Unusable
Machine Check
Trap if Less Than

Format: TLT rs, rt

Purpose:
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

<table>
<thead>
<tr>
<th>Format:</th>
<th>TLTI rs, immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>To compare a GPR to a constant and do a conditional trap</td>
</tr>
<tr>
<td>Description:</td>
<td>if GPR[rs] &lt; immediate then Trap</td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Restrictions:</td>
<td>None</td>
</tr>
<tr>
<td>Operation:</td>
<td>if GPR[rs] &lt; sign_extend(immediate) then SignalException(Trap) endif</td>
</tr>
<tr>
<td>Exceptions:</td>
<td>Trap</td>
</tr>
</tbody>
</table>

**MIPS32**
### Trap if Less Than Immediate Unsigned

**Format:** TLTIU rs, immediate

**Purpose:**
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:**
```plaintext
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
    SignalException(Trap)
endif
```

**Exceptions:**
Trap

---

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>TLTIU</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td></td>
<td>01011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31 26 25 21 20 16 15          0</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGIMM</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>000001</td>
</tr>
</tbody>
</table>
Trap if Less Than Unsigned

**Format:** \texttt{TLTU rs, rt}  \hspace{1cm} \textbf{MIPS32}

**Purpose:**
To compare GPRs and do a conditional trap

**Description:** if GPR[rs] < GPR[rt] then Trap

Compare the contents of GPR \textit{rs} and GPR \textit{rt} as unsigned integers; if GPR \textit{rs} is less than GPR \textit{rt}, then take a Trap exception.

The contents of the \textit{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:**  \( \text{TNE} \ rs, \ rt \)  

**MIPS32**

**Purpose:**
To compare GPRs and do a conditional trap

**Description:** \( \text{if } \text{GPR}[rs] \neq \text{GPR}[rt] \text{ then Trap} \)

Compare the contents of GPR \( rs \) and GPR \( rt \) as signed integers; if GPR \( rs \) is not equal to GPR \( rt \), then take a Trap exception.

The contents of the code field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

**Restrictions:**
None

**Operation:**

\[
\text{if } \text{GPR}[rs] \neq \text{GPR}[rt] \text{ then}
\]

\[
\text{SignalException(Trap)}
\]

**Exceptions:**

Trap
Trap if Not Equal Immediate

Format: TNEI rs, immediate

Purpose:
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:
if GPR[rs] ≠ sign_extend(immediate) then
    SignalException(Trap)
endif

Exceptions:
Trap
Floating Point Truncate to Long Fixed Point

**Format:**

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>fs</th>
<th>fd</th>
<th>TRUNC.L</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>001001</td>
</tr>
</tbody>
</table>

- **TRUNC.L.S fd, fs**
- **TRUNC.L.D fd, fs**

**MIPS64, MIPS32 Release 2**

**MIPS64, MIPS32 Release 2**

**Purpose:**

To convert an FP value to 64-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 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 16 FP registers mode.

**Operation:**

\[
\text{StoreFPR}(fd, L, \text{ConvertFmt}(	ext{ValueFPR}(fs, fmt), fmt, L))
\]
Floating Point Truncate to Long Fixed Point (cont.)

Exceptions:
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:
Unimplemented Operation, Invalid Operation, Overflow, Inexact
TRUNC.W.fmt

Floating Point Truncate to Word Fixed Point

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>fs</td>
<td>fd</td>
<td>TRUNC.W</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31 26 25 21 20 16 15 11 10 6 5 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
</tr>
<tr>
<td>010001</td>
</tr>
</tbody>
</table>

**Format:** TRUNC.W.S fd, fs
TRUNC.W.D fd, fs

**Purpose:**
To convert an FP value to 32-bit fixed point, rounding toward zero

**Description:**
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 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:**
StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
Exceptions:
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:
Inexact, Invalid Operation, Overflow, Unimplemented Operation
Enter Standby Mode

<table>
<thead>
<tr>
<th>COP0</th>
<th>CO</th>
<th>Implementation-Dependent Code</th>
<th>WAIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>1</td>
<td></td>
<td>100000</td>
</tr>
</tbody>
</table>

**Format:** WAIT

**Purpose:**
Wait for Event

**Description:**
The WAIT instruction performs an implementation-dependent operation, usually involving a lower power mode. Software may use bits 24:6 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 bits 24:6 is the default and must be valid in all implementations.

The 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 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 WAIT instruction and the following instruction (EPC for the interrupt points at the instruction following the WAIT instruction).

**Restrictions:**
The operation of the processor is UNDEFINED if a 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.
### Enter Standby Mode (cont.)

<table>
<thead>
<tr>
<th>Operation:</th>
<th>WAIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>I: Enter implementation dependent lower power mode</td>
<td></td>
</tr>
<tr>
<td>I+1: /* Potential interrupt taken here */</td>
<td></td>
</tr>
</tbody>
</table>

#### Exceptions:

- Coprocessor Unusable Exception
Write to GPR in Previous Shadow Set

<table>
<thead>
<tr>
<th>COP0</th>
<th>WRPGPR</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100 00</td>
<td>01 110</td>
<td>5</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>000 0000 0000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:  WRPGPR rd, rt

Purpose:
To move the contents of a current GPR to a GPR in the previous shadow set.

Description:  SGPR[SRSCtlPSS, rd] ← GPR[rt]

The contents of the current GPR rt is moved to the shadow GPR register specified by 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:

SGPR[SRSCtlPSS, rd] ← GPR[rt]

Exceptions:
Coprocessor Unusable
Reserved Instruction
**Word Swap Bytes Within Halfwords**

**Format:** `wsbh rd, rt`  
**Purpose:** To swap the bytes within each halfword of GPR `rt` and store the value into GPR `rd`.

**Description:**  
Within each halfword of the lower word of GPR `rt` the bytes are swapped, the result 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.

If GPR `rs` does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

**Operation:**

```plaintext
if NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif

```

**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:

```plaintext
lw  t0, 0(a1)      /* 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:

```plaintext
lw  t0, 0(a1)      /* 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.
### XOR

**Format:**  XOR rd, rs, rt  

**Purpose:**  
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

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>XOR</td>
<td>100110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS32
**XORI**

**Format:** XORI rt, rs, immediate

**Purpose:**
To do a bitwise logical Exclusive OR with a constant

**Description:**
\[ GPR[rt] \leftarrow GPR[rs] \text{ XOR immediate} \]

Combine the contents of GPR \( rs \) and the 16-bit zero-extended \textit{immediate} in a bitwise logical Exclusive OR operation and place the result into GPR \( rt \).

**Restrictions:**
None

**Operation:**
\[ GPR[rt] \leftarrow GPR[rs] \text{ xor zero_extend(immediate)} \]

**Exceptions:**
None
Appendix A

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 MIPS64® 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).
Appendix A Instruction Bit Encodings

Tables A-2 through A-23 describe the encoding used for the MIPS64 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 are not legal if the processor is configured to be backward compatible with MIPS32 processors. If the processor is executing with 64-bit operations enabled, execution proceeds normally. 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 Unusable 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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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 Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).</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 Unusable 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.</td>
</tr>
<tr>
<td>ε</td>
<td>Operation or field codes marked with this symbol are reserved for MIPS Application Specific Extensions. If the 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 MIPS64 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 MIPS64 Encoding of the Opcode Field

<table>
<thead>
<tr>
<th>opcode</th>
<th>bits 28..26</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>SPECIAL δ</td>
</tr>
<tr>
<td>1 001</td>
<td>ADDI δ</td>
</tr>
<tr>
<td>2 010</td>
<td>COP0 δ</td>
</tr>
<tr>
<td>3 011</td>
<td>DADDI ⊥</td>
</tr>
<tr>
<td>4 100</td>
<td>LB</td>
</tr>
<tr>
<td>5 101</td>
<td>SB</td>
</tr>
<tr>
<td>6 110</td>
<td>LL</td>
</tr>
<tr>
<td>7 111</td>
<td>SC</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.
### Table A-3 MIPS64 SPECIAL Opcode Encoding of Function Field

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 5..3</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>bits 5..3</td>
<td>000 001 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td>0 000</td>
<td>SLL</td>
</tr>
<tr>
<td>1 001</td>
<td>JR</td>
</tr>
<tr>
<td>2 010</td>
<td>MFHI</td>
</tr>
<tr>
<td>3 011</td>
<td>MULT</td>
</tr>
<tr>
<td>4 100</td>
<td>ADD</td>
</tr>
<tr>
<td>5 101</td>
<td>*</td>
</tr>
<tr>
<td>6 110</td>
<td>TGE</td>
</tr>
<tr>
<td>7 111</td>
<td>DSSL</td>
</tr>
</tbody>
</table>

1. Specific encodings of the rt, rd, and so fields are used to distinguish among the SLL, NOP, SSNOP and EHB functions.
2. Specific encodings of the hint field are used to distinguish JR from JR.HB and JALR from JALR.HB.

### Table A-4 MIPS64 REGIMM Encoding of rt Field

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

### Table A-5 MIPS64 SPECIAL2 Encoding of Function Field

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 5..3</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>bits 5..3</td>
<td>000 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td>0 000</td>
<td>MADD</td>
</tr>
<tr>
<td>1 001</td>
<td>0</td>
</tr>
<tr>
<td>2 010</td>
<td>0</td>
</tr>
<tr>
<td>3 011</td>
<td>0</td>
</tr>
<tr>
<td>4 100</td>
<td>CLZ</td>
</tr>
<tr>
<td>5 101</td>
<td>0</td>
</tr>
<tr>
<td>6 110</td>
<td>0</td>
</tr>
<tr>
<td>7 111</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table A-6 MIPS64 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>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>bits 5..3</td>
<td>000 001 010 011 100 101 110 111</td>
</tr>
<tr>
<td>0 000</td>
<td>EXT</td>
</tr>
<tr>
<td>1 001</td>
<td>*</td>
</tr>
<tr>
<td>2 010</td>
<td>*</td>
</tr>
<tr>
<td>3 011</td>
<td>*</td>
</tr>
<tr>
<td>4 100</td>
<td>BSHFL</td>
</tr>
<tr>
<td>5 101</td>
<td>*</td>
</tr>
<tr>
<td>6 110</td>
<td>*</td>
</tr>
<tr>
<td>7 111</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 MIPS64 MOVCI Encoding of tf Bit

<table>
<thead>
<tr>
<th>tf</th>
<th>bit 16</th>
<th>MOVF</th>
<th>MOVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table A-8 MIPS641 SRL Encoding of Shift/Rotate

<table>
<thead>
<tr>
<th>R</th>
<th>bit 21</th>
<th>SRL</th>
<th>ROTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></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 MIPS641 SRLV Encoding of Shift/Rotate

<table>
<thead>
<tr>
<th>R</th>
<th>bit 6</th>
<th>SRLV</th>
<th>ROTRV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></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 MIPS641 DSRLV Encoding of Shift/Rotate

<table>
<thead>
<tr>
<th>R</th>
<th>bit 6</th>
<th>DSRLV</th>
<th>DROTRV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Release 2 of the Architecture added the DROTRV instruction. Implementations of Release 1 of the Architecture ignored bit 6 and treated the instruction as a DSRLV.

### Table A-11 MIPS641 DSRL Encoding of Shift/Rotate

<table>
<thead>
<tr>
<th>R</th>
<th>bit 21</th>
<th>DSRL</th>
<th>DROTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Release 2 of the Architecture added the DROTR instruction. Implementations of Release 1 of the Architecture ignored bit 21 and treated the instruction as a DSRL.
### Table A-12 MIPS64 DSRL32 Encoding of Shift/Rotate

<table>
<thead>
<tr>
<th>R</th>
<th>bit 21</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DSRL32</td>
</tr>
<tr>
<td>1</td>
<td>DROTR32</td>
</tr>
</tbody>
</table>

1. Release 2 of the Architecture added the DROTR32 instruction. Implementations of Release 1 of the Architecture ignored bit 21 and treated the instruction as a DSRL32.

### Table A-13 MIPS64 BSHFL and DBSHFL Encoding of sa Field

<table>
<thead>
<tr>
<th>sa</th>
<th>bits 8..6</th>
<th>bits 10..9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>WSBH (BSHFL)</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>DSBH (DBSHFL)</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>DSHD (DBSHFL)</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-14 MIPS64 COP0 Encoding of rs Field

<table>
<thead>
<tr>
<th>rs</th>
<th>bits 23..21</th>
<th>bits 25..24</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>MFC0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>RDPGR ⊕</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>WRPGPR ⊕</td>
</tr>
</tbody>
</table>

1. Release 2 of the Architecture added the MFMC0 function, which is further decoded as the DI and EI instructions.

### Table A-15 MIPS64 COP0 Encoding of Function Field When rs=CO

<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>010</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>011</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>5</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

1. Release 2 of the Architecture added the MFMC0 function, which is further decoded as the DI and EI instructions.
### Table A-16 MIPS64 COP1 Encoding of rs Field

<table>
<thead>
<tr>
<th>rs</th>
<th>bits 23..21</th>
<th>bits 25..24</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>011</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>101</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>110</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>MFC1</td>
</tr>
<tr>
<td>001</td>
<td>DMFC1 ⊥</td>
</tr>
<tr>
<td>010</td>
<td>CFC1</td>
</tr>
<tr>
<td>011</td>
<td>MFHC1 ⊕</td>
</tr>
<tr>
<td>100</td>
<td>TMT1 ⊥</td>
</tr>
<tr>
<td>101</td>
<td>DMTC1 ⊥</td>
</tr>
<tr>
<td>110</td>
<td>CTC1</td>
</tr>
<tr>
<td>111</td>
<td>MTHC1 ⊕</td>
</tr>
</tbody>
</table>

### Table A-17 MIPS64 COP1 Encoding of Function Field When rs=5

<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>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ADD</td>
</tr>
<tr>
<td>001</td>
<td>SUB</td>
</tr>
<tr>
<td>010</td>
<td>MUL</td>
</tr>
<tr>
<td>011</td>
<td>DIV</td>
</tr>
<tr>
<td>100</td>
<td>SQRT</td>
</tr>
<tr>
<td>101</td>
<td>ABS</td>
</tr>
<tr>
<td>110</td>
<td>MOV</td>
</tr>
<tr>
<td>111</td>
<td>NEG</td>
</tr>
</tbody>
</table>

### Table A-18 MIPS64 COP1 Encoding of Function Field When rs=D

<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>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ADD</td>
</tr>
<tr>
<td>001</td>
<td>SUB</td>
</tr>
<tr>
<td>010</td>
<td>MUL</td>
</tr>
<tr>
<td>011</td>
<td>DIV</td>
</tr>
<tr>
<td>100</td>
<td>SQRT</td>
</tr>
<tr>
<td>101</td>
<td>ABS</td>
</tr>
<tr>
<td>110</td>
<td>MOV</td>
</tr>
<tr>
<td>111</td>
<td>NEG</td>
</tr>
</tbody>
</table>

### Table A-19 MIPS64 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>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>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ADD</td>
</tr>
<tr>
<td>001</td>
<td>SUB</td>
</tr>
<tr>
<td>010</td>
<td>MUL</td>
</tr>
<tr>
<td>011</td>
<td>DIV</td>
</tr>
<tr>
<td>100</td>
<td>SQRT</td>
</tr>
<tr>
<td>101</td>
<td>ABS</td>
</tr>
<tr>
<td>110</td>
<td>MOV</td>
</tr>
<tr>
<td>111</td>
<td>NEG</td>
</tr>
</tbody>
</table>

---

1. Format type L is legal only if 64-bit floating point operations are enabled.
### Appendix A Instruction Bit Encodings

#### 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-16 and Table A-23 above.

---

#### Table A-20 MIPS64 COP1 Encoding of Function Field When rs=PS

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 5..3</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>0 000</td>
<td>ADD V</td>
</tr>
<tr>
<td>1 001</td>
<td>*</td>
</tr>
<tr>
<td>2 010</td>
<td>MOVCF eV</td>
</tr>
<tr>
<td>3 011</td>
<td>ADDR eV</td>
</tr>
<tr>
<td>4 100</td>
<td>CVT.S.PU</td>
</tr>
<tr>
<td>5 101</td>
<td>CVT.S.PL</td>
</tr>
<tr>
<td>6 110</td>
<td>C.F V</td>
</tr>
<tr>
<td></td>
<td>ABS.F V</td>
</tr>
<tr>
<td>7 111</td>
<td>C.SF V</td>
</tr>
<tr>
<td></td>
<td>ABS.SF V</td>
</tr>
</tbody>
</table>

1. Format type PS is legal only if 64-bit floating point operations are enabled.

#### Table A-21 MIPS64 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>0</td>
<td>MOVF.fmt</td>
</tr>
<tr>
<td>1</td>
<td>MOVT.fmt</td>
</tr>
</tbody>
</table>

#### Table A-22 MIPS64 COP2 Encoding of rs Field

<table>
<thead>
<tr>
<th>rs</th>
<th>bits 23..21</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 25..24</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>0 00</td>
<td>MFC2 Θ</td>
</tr>
<tr>
<td>1 01</td>
<td>BC2 Θ</td>
</tr>
<tr>
<td>2 10</td>
<td>SWXC1 V</td>
</tr>
<tr>
<td>3 11</td>
<td>*</td>
</tr>
</tbody>
</table>

#### Table A-23 MIPS64 COP1X Encoding of Function Field

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 5..3</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>0 000</td>
<td>LWXC1 V</td>
</tr>
<tr>
<td>1 001</td>
<td>SWXC1 V</td>
</tr>
<tr>
<td>2 010</td>
<td>*</td>
</tr>
<tr>
<td>3 011</td>
<td>*</td>
</tr>
<tr>
<td>4 100</td>
<td>MADD.S V</td>
</tr>
<tr>
<td>5 101</td>
<td>MSUB.S V</td>
</tr>
<tr>
<td>6 110</td>
<td>NMADD.S V</td>
</tr>
<tr>
<td>7 111</td>
<td>NMSUB.S V</td>
</tr>
</tbody>
</table>

1. COP1X instructions are legal only if 64-bit floating point operations are enabled.
### Table A-24 Floating Point Unit Instruction Format Encodings

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Mnemonic</th>
<th>Name</th>
<th>Bit Width</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..15</td>
<td>00..0F</td>
<td>—</td>
<td>—</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>S</td>
<td>Single</td>
<td>32</td>
<td>Floating Point</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>D</td>
<td>Double</td>
<td>64</td>
<td>Floating Point</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>W</td>
<td>Word</td>
<td>32</td>
<td>Fixed Point</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>L</td>
<td>Long</td>
<td>64</td>
<td>Fixed Point</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>PS</td>
<td>Paired Single</td>
<td>2 × 32</td>
<td>Floating Point</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>—</td>
<td>—</td>
<td>Reserved for future use by the architecture.</td>
<td></td>
</tr>
<tr>
<td>24..31</td>
<td>18..1F</td>
<td>—</td>
<td>—</td>
<td>Reserved for future use by the architecture. Not available for \textit{fmt3} encoding.</td>
<td></td>
</tr>
</tbody>
</table>
Revision History

In the left hand page margins of this document you may find vertical change bars to note the location of significant changes to this document since its last release. Significant changes are defined as those which you should take note of as you use the MIPS IP. Changes to correct grammar, spelling errors or similar may or may not be noted with change bars. Change bars will be removed for changes which are more than one revision old.

Please note: Limitations on the authoring tools make it difficult to place change bars on changes to figures. Change bars on figure titles are used to denote a potential change in the figure itself.

<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>External 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>Updated based on feedback from all reviews.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Add missing optional select field syntax in mtc0/mfc0 instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>descriptions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Correct the PREF instruction description to acknowledge that the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PrepareForStore function does, in fact, modify architectural state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• To provide additional flexibility for Coprocessor 2 implementations,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>extend the sel field for DMFC0, DMTC0, MFC0, and MTC0 to be 8 bits.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Update the PREF instruction to note that it may not update the state of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a locked cache line.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Remove obviously incorrect documentation in DIV and DIVU with regard</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to putting smaller numbers in register rt.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Fix the description for MFC2 to reflect data movement from the coprocessor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if the cache line containing the instruction is the target of an invalidate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DMTC2 instructions implementation dependent, thereby acknowledging that</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>field of the SYNC instruction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Update the compatibility and subsetting rules to capture the current</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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: DEXT, DEXTM, DEXTU, DI, DINS, DINSU, DINSM, DROTR, DROTR32, DROTRV, DSBH, DSHD, EHB, EI, EXT, INS, JALR.HB, JR.HB, MFHC1, MFHC2, MTHC1, MTHC2, RDHWR, RDPGPR, ROTR, ROTRV, SEB, SEH, SYNCI, WRFGPR, WSBH.  
• The following instruction definitions changed to reflect Release 2 of the Architecture: DERET, DSRL, DSRL32, DSRLV, 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 Are64bitFPOperationsEnabled 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, DSR32, 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 can not 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. |
## 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, SDXC1, 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>
<tbody>
<tr>
<td>2.50</td>
<td>July 1, 2005</td>
<td>Changes in this revision:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Correct figure label in LWR instruction (it was incorrectly specified as LWL).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Update all files to FrameMaker 7.1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Include support for implementation-dependent hardware registers via RDHWR.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Correct the MIPS32 pseudocode for the LDC1, LDXC1, LUXC1, SDC1, SDXC1, 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Expand on restrictions on the PREF instruction in cases where the effective address has an uncached coherency attribute.</td>
</tr>
</tbody>
</table>