PM0045
Programming manual

Signal processing engine (SPE) APU programming interface manual

Introduction

The primary objective of this manual is to help programmers provide software that is compatible across the family of processors that use the signal processing engine (SPE) auxiliary processing unit (APU).

Scope

The scope of this manual does not include a description of individual SPE implementations. Each PowerPC™ processor is unique in its implementation of the SPE.

Audience

This manual supports system software and application programmers who want to use the SPE APU to develop products. Users should understand the following concepts:

- Operating systems
- Microprocessor system design
- Basic principles of RISC processing
- SPE instruction set

The major sections of this manual provide a general understanding of what the programming model defines in the SPE APU.

It is useful for software engineers who need to understand how to access SPE functionality from high level languages such as C and C++.

It will describe all instructions in the e500 core complex as well as Book E instructions that are defined for 32-bit implementations, along with data manipulation, SPE floating-point status and control register (SPEFSCR) operations, ABI extensions (malloc(), realloc(), calloc(), and new), a printf example, and additional library routines.

Examples are given of valid and invalid initializations of the SPE data types.

Contents

1 Overview 14

1.1 High-level language interface 14

1.2 Application binary interface (ABI) 14

2 High-level language interface 15

2.1 Introduction 15

2.2 High-level language interface 15

3 SPE operations 19

3.1 Signal processing engine (SPE) APU registers 19

3.2 Notation 22

3.3 Instruction fields 22

3.4 Description of instruction operation 25

3.5 Intrinsics 28

3.6 Basic instruction mapping 285

4 Additional operations 295

4.1 Data manipulation 295

4.2 Signal processing engine (SPE) APU registers 298

4.3 Application binary interface (ABI) extensions 303

5 Programming interface examples 306

5.1 Data type initialization 306

5.2 Fixed-point accessors 308

5.3 Loads 309

6 Glossary of terms and abbreviations 311

7 Revision history 314

List of tables

Table 1. Data types . . . 15
Table 2. SPEFSCR field descriptions . . . 20
Table 3. ACC field descriptions . . . 22
Table 4. Notation conventions . . . 22
Table 5. Instruction field descriptions . . . 23
Table 6. RTL notation . . . 25
Table 7. Operator precedence . . . 28
Table 8. Data samples and sizes . . . 31
Table 9. __brinc (registers altered by) . . . 31
Table 10. __ev_abs (registers altered by) . . . 32
Table 11. __ev_addiw (registers altered by) . . . 33
Table 12. __ev_addsmiaaw (registers altered by) . . . 34
Table 13. __ev_addssiaaw (registers altered by) . . . 35
Table 14. __ev_addumiaaw (registers altered by) . . . 36
Table 15. __ev_addusiaaw (registers altered by) . . . 37
Table 16. __ev_addw (registers altered by) . . . 38
Table 17. __ev_all_eq (registers altered by) . . . 39
Table 18. __ev_all_fs_eq (registers altered by) . . . 40
Table 19. __ev_all_fs_gt (registers altered by) . . . 41
Table 20. __ev_all_fs_lt (registers altered by) . . . 42
Table 21. __ev_all_fs_tst_eq (registers altered by) . . . 43
Table 22. __ev_all_fs_tst_gt (registers altered by) . . . 44
Table 23. __ev_all_fs_tst_lt (registers altered by) . . . 45
Table 24. __ev_all_gts (registers altered by) . . . 46
Table 25. __ev_all_gtu (registers altered by) . . . 47
Table 26. __ev_all_lts (registers altered by) . . . 48
Table 27. __ev_all_ltu (registers altered by) . . . 49
Table 28. __ev_and (registers altered by) . . . 50
Table 29. __ev_andc (registers altered by) . . . 51
Table 30. __ev_any_eq (registers altered by) . . . 52
Table 31. __ev_any_fs_eq (registers altered by) . . . 53
Table 32. __ev_any_fs_gt (registers altered by) . . . 54
Table 33. __ev_any_fs_lt (registers altered by) . . . 55
Table 34. __ev_any_fs_tst_eq (registers altered by) . . . 56
Table 35. __ev_any_fs_tst_gt (registers altered by) . . . 57
Table 36. __ev_any_fs_tst_lt (registers altered by) . . . 58
Table 37. __ev_any_gts (registers altered by) . . . 59
Table 38. __ev_any_gtu (registers altered by) . . . 60
Table 39. __ev_any_lts (registers altered by) . . . 61
Table 40. __ev_any_ltu (registers altered by) . . . 62
Table 41. __ev_cntlsw (registers altered by) . . . 63
Table 42. __ev_cntlzw (registers altered by) . . . 64
Table 43. __ev_divws (registered altered by) . . . 66
Table 44. __ev_divwu (registers altered by) . . . 67
Table 45. __ev_eqv (registers altered by) . . . 68
Table 46. __ev_extsb (registers altered by) . . . 69
Table 47. __ev_extsh (registers altered by) . . . 70
Table 48. __ev_fsabs (registers altered by) . . . 71

Table 49. __ev_fsadd (registers altered by). . . . 72
Table 50. __ev_fscfsf (registers altered by). . . . 73
Table 51. __ev_fscfsi (registers altered by). . . . 74
Table 52. __ev_fscfuf (registers altered by). . . . 75
Table 53. __ev_fscfui (registers altered by). . . . 76
Table 54. __ev_fsctsf (registers altered by). . . . 77
Table 55. __ev_fsctsi (registers altered by). . . . 78
Table 56. __ev_fsctsiz (registers altered by). . . . 79
Table 57. __ev_fsctuf (registers altered by). . . . 80
Table 58. __ev_fsctui (registers altered by). . . . 81
Table 59. __ev_fsctuiz (registers altered by). . . . 82
Table 60. __ev_fsdiv (registers altered by). . . . 83
Table 61. __ev_fsmul (registers altered by). . . . 84
Table 62. __ev_fsnabs (registers altered by). . . . 85
Table 63. __ev_fsneg (registers altered by). . . . 86
Table 64. __ev_fssub (registers altered by). . . . 87
Table 65. __ev_ldd (registers altered by). . . . 88
Table 66. __ev_lddx__ev_lddx (registers altered by). . . . 89
Table 67. __ev_ldh (registers altered by). . . . 90
Table 68. __ev_ldhx (registers altered by). . . . 91
Table 69. __ev_ldw (registers altered by). . . . 92
Table 70. __ev_ldwx (registers altered by). . . . 93
Table 71. __ev_lhhesplat (registers altered by). . . . 94
Table 72. __ev_lhhesplatx (registers altered by). . . . 95
Table 73. __ev_lhhossplat (registers altered by). . . . 96
Table 74. __ev_lhhossplatx (registers altered by). . . . 97
Table 75. __ev_lhhousplat (registers altered by). . . . 98
Table 76. __ev_lhhousplatx (registers altered by). . . . 99
Table 77. __ev_lower_eq (registers altered by). . . . 100
Table 78. __ev_lower_fs_eq (registers altered by). . . . 101
Table 79. __ev_lower_fs_gt (registers altered by). . . . 102
Table 80. __ev_lower_fs_lt (registers altered by). . . . 103
Table 81. __ev_lower_fs_tst_eq (registers altered by). . . . 104
Table 82. __ev_lower_fs_tst_gt (registers altered by). . . . 105
Table 83. __ev_lower_fs_tst_lt (registers altered by). . . . 106
Table 84. __ev_lower__gts (registers altered by). . . . 107
Table 85. __ev_lower_gtu (registers altered by). . . . 108
Table 86. __ev_lower_lts (registers altered by). . . . 109
Table 87. __ev_lower_ltu (registers altered by). . . . 110
Table 88. __ev_lwhe (registers altered by). . . . 111
Table 89. __ev_lwhex (registers altered by). . . . 112
Table 90. __ev_lwhos (registers altered by). . . . 113
Table 91. __ev_lwhosx (registers altered by). . . . 114
Table 92. __ev_lwhou (registers altered by). . . . 115
Table 93. __ev_lwhoux (registers altered by). . . . 116
Table 94. __ev_lwhsplat (registers altered by). . . . 117
Table 95. __ev_lwhsplatx (registers altered by). . . . 118
Table 96. __ev_lwwsplat (registers altered by). . . . 119
Table 97. __ev_lwwsplatx (registers altered by). . . . 120
Table 98. __ev_mergehi (registers altered by). . . . 121
Table 99. __ev_mergehilo (registers altered by). . . . 122
Table 100. __ev_mergelo (registers altered by). . . . 123

Table 101. __ev_mergelohi (registers altered by). . . . 124
Table 102. __ev_mhegsmfaa (registers altered by). . . . 125
Table 103. __ev_mhegsmfan (registers altered by). . . . 126
Table 104. __ev_mhegsmiaa (registers altered by). . . . 127
Table 105. __ev_mhegsmian (registers altered by). . . . 128
Table 106. __ev_mhegumfaa (registers altered by). . . . 129
Table 107. __ev_mhegumiaa (registers altered by). . . . 130
Table 108. __ev_mhegumfan (registers altered by). . . . 131
Table 109. __ev_mhegumian (registers altered by). . . . 132
Table 110. __ev_mhesmf (registers altered by). . . . 133
Table 111. __ev_mhesmfaaw (registers altered by). . . . 134
Table 112. __ev_mhesmfanw (registers altered by). . . . 135
Table 113. __ev_mhesmi (registers altered by). . . . 136
Table 114. __ev_mhesmiaaw (registers altered by). . . . 137
Table 115. __ev_mhesmianw (registers altered by). . . . 138
Table 116. __ev_mhessf (registers altered by). . . . 140
Table 117. __ev_mhessfaaw (registers altered by). . . . 142
Table 118. __ev_mhessfanw (registers altered by). . . . 144
Table 119. __ev_mhessiaaw (registers altered by). . . . 145
Table 120. __ev_mhessianw (registers altered by). . . . 146
Table 121. __ev_mheumf (registers altered by). . . . 147
Table 122. __ev_mheumi (registers altered by). . . . 148
Table 123. __ev_mheumfaaw (registers altered by). . . . 149
Table 124. __ev_mheumiaaw (registers altered by). . . . 150
Table 125. __ev_mheumfanw (registers altered by). . . . 151
Table 126. __ev_mheumianw (registers altered by). . . . 152
Table 127. __ev_mheusfaaw (registers altered by). . . . 154
Table 128. __ev_mheusiaaw (registers altered by). . . . 155
Table 129. __ev_mheusfanw (registers altered by). . . . 157
Table 130. __ev_mheusianw (registers altered by). . . . 159
Table 131. __ev_mhogsmfaa (registers altered by). . . . 160
Table 132. __ev_mhogsmfan (registers altered by). . . . 161
Table 133. __ev_mhogsmiaa (registers altered by). . . . 162
Table 134. __ev_mhogsmian (registers altered by). . . . 163
Table 135. __ev_mhogumfaa (registers altered by). . . . 164
Table 136. __ev_mhogumiaa (registers altered by). . . . 165
Table 137. __ev_mhogumfan (registers altered by). . . . 166
Table 138. __ev_mhogumian (registers altered by). . . . 167
Table 139. __ev_mhosmf (registers altered by). . . . 168
Table 140. __ev_mhosmfaaw (registers altered by). . . . 169
Table 141. __ev_mhosmfanw (registers altered by). . . . 170
Table 142. __ev_mhosmi (registers altered by). . . . 171
Table 143. __ev_mhosmiaaw (registers altered by). . . . 172
Table 144. __ev_mhosmianw (registers altered by). . . . 173
Table 145. __ev_mhossf (registers altered by). . . . 175
Table 146. __ev_mhossfaaw (registers altered by) 177
Table 147. __ev_mhossfanw (registers altered by). . . . 179
Table 148. __ev_mhossiaaw (registers altered by). . . . 180
Table 149. __ev_mhossianw (registers altered by). . . . 181
Table 150. __ev_mhoumf (registers altered by). . . . 182
Table 151. __ev_mhoumi (registers altered by). . . . 183
Table 152. __ev_mhoumfaaw (registers altered by). . . . 184

Table 153. __ev_mhoumiaaw (registers altered by).. 185
Table 154. __ev_mhoumfanw (registers altered by). 186
Table 155. __ev_mhoumianw (registers altered by).. 187
Table 156. __ev_mhousfaaw (registers altered by).. 189
Table 157. __ev_mhousiaaw (registers altered by).. 191
Table 158. __ev_mhousfanw (registers altered by).. 193
Table 159. __ev_mhousianw (registers altered by).. 195
Table 160. __ev_mra (registers altered by). 196
Table 161. __ev_mwhsmf (registers altered by). 197
Table 162. __ev_mwhsmi (registers altered by).. 198
Table 163. __ev_mwhssf (registers altered by). 200
Table 164. __ev_mwhumf (registers altered by). 201
Table 165. __ev_mwhumi (registers altered by).. 202
Table 166. __ev_mwlsmiaaw (registers altered by). 203
Table 167. __ev_mwlsmianw (registers altered by). 204
Table 168. __ev_mwlssiaaw (registers altered by).. 205
Table 169. __ev_mwlssianw (registers altered by).. 206
Table 170. __ev_mwlumi (registers altered by). 207
Table 171. __ev_mwlumiaaw (registers altered by). 208
Table 172. __ev_mwlumianw (registers altered by). 209
Table 173. __ev_mwlusiaaw (registers altered by). 210
Table 174. __ev_mwlusianw (registers altered by). 211
Table 175. __ev_mwsmf (registers altered by).. 212
Table 176. __ev_mwsmf (registers altered by).. 212
Table 177. __ev_mwsmfaa (registers altered by). 213
Table 178. __ev_mwsmfan (registers altered by). 214
Table 179. __ev_mwsmi (registers altered by). 215
Table 180. __ev_mwsmiaa (registers altered by). 216
Table 181. __ev_mwsmian (registers altered by). 217
Table 182. __ev_mwssf (registers altered by).. 218
Table 183. __ev_mwssfaa (registers altered by).. 220
Table 184. __ev_mwssfan (registers altered by).. 222
Table 185. __ev_mwumi (registers altered by). 223
Table 186. __ev_mwumiaa (registers altered by). 224
Table 187. __ev_mwumian (registers altered by). 225
Table 188. __ev_nand (registers altered by).. 226
Table 189. __ev_neg (registers altered by).. 227
Table 190. __ev_nor (registers altered by). 228
Table 191. __ev_or (registers altered by). 229
Table 192. __ev_orc (registers altered by). 230
Table 193. __ev_rlw (registers altered by). 231
Table 194. __ev_rlwi (registers altered by). 232
Table 195. __ev_rndw (registers altered by). 233
Table 196. __ev_select_eq (registers altered by). 234
Table 197. __ev_select_fs_eq (registers altered by).. 235
Table 198. __ev_select_fs_gt (registers altered by). 236
Table 199. __ev_select_fs_lt (registers altered by).. 237
Table 200. __ev_select_fs_tst_eq (registers altered by).. 238
Table 201. __ev_select_fs_tst_gt (registers altered by). 239
Table 202. __ev_select_fs_tst_lt (registers altered by).. 240
Table 203. __ev_select_gts (registers altered by).. 241
Table 204. __ev_select_gtu (registers altered by).. 242

Table 205. __ev_select_lts (registers altered by). . . . 243
Table 206. __ev_select_ltu (registers altered by). . . . 244
Table 207. __ev_slw (registers altered by). . . . 245
Table 208. __ev_slwi (registers altered by). . . . 246
Table 209. __ev_splatfi (registers altered by). . . . 247
Table 210. __ev_splati (registers altered by). . . . 248
Table 211. __ev_srwis (registers altered by). . . . 249
Table 212. __ev_srwiu (registers altered by). . . . 250
Table 213. __ev_srws (registers altered by). . . . 251
Table 214. __ev_srwu (registers altered by). . . . 252
Table 215. __ev_stdd (registers altered by). . . . 253
Table 216. __ev_stddx (registers altered by). . . . 254
Table 217. __ev_stdh (registers altered by). . . . 255
Table 218. __ev_stdhx (registers altered by). . . . 256
Table 219. __ev_stdw (registers altered by). . . . 257
Table 220. __ev_stdwx (registers altered by). . . . 258
Table 221. __ev_stwhe (registers altered by). . . . 259
Table 222. __ev_stwhex (registers altered by). . . . 260
Table 223. __ev_stwho (registers altered by). . . . 261
Table 224. __ev_stwhox (registers altered by). . . . 262
Table 225. __ev_stwwe (registers altered by). . . . 263
Table 226. __ev_stwwex (registers altered by). . . . 264
Table 227. __ev_stwwo (registers altered by). . . . 265
Table 228. __ev_stwwox (registers altered by). . . . 266
Table 229. __ev_subfsmiaaw (registers altered by). . . . 267
Table 230. __ev_subfssiaaw (registers altered by). . . . 268
Table 231. __ev_subfumiaaw (registers altered by). . . . 269
Table 232. __ev_subfusiaaw (registers altered by). . . . 270
Table 233. __ev_subfw (registers altered by). . . . 271
Table 234. __ev_subifw (registers altered by). . . . 272
Table 235. __ev_upper_eq (registers altered by). . . . 273
Table 236. __ev_upper_fs_eq (registers altered by). . . . 274
Table 237. __ev_upper_fs_gt (registers altered by). . . . 275
Table 238. __ev_upper_fs_lt (registers altered by). . . . 276
Table 239. __ev_upper_fs_tst_eq (registers altered by). . . . 277
Table 240. __ev_upper_fs_tst_gt (registers altered by). . . . 278
Table 241. __ev_upper_fs_tst_lt (registers altered by). . . . 279
Table 242. __ev_upper_gts (registers altered by). . . . 280
Table 243. __ev_upper_gtu (registers altered by). . . . 281
Table 244. __ev_upper_lts (registers altered by). . . . 282
Table 245. __ev_upper_ltu (registers altered by). . . . 283
Table 246. __ev_xor (registers altered by). . . . 284
Table 247. SPEFSCR field descriptions 299
Table 248. New tokens for fixed-point data types. . . . 303
Table 249. Document revision history 314

List of figures

Figure 1. Signal processing and embedded floating-point status and control register (SPEFSCR) . 19
Figure 2. Accumulator (ACC) . . . 22
Figure 3. Instruction description . . . 29
Figure 4. Vector absolute value (__ev_abs) . . . 32
Figure 5. Vector add immediate word (__ev_addiw) . . . 33
Figure 6. Vector add signed, modulo, integer to accumulator word (ev_addsmiaaw) . . . 34
Figure 7. Vector add signed, saturate, integer to accumulator word (ev_addssiaaw) . . . 35
Figure 8. Vector add unsigned,modulo,integer to accumulator word (ev_addumiaaw) . . . 36
Figure 9. Vector add unsigned, saturate, integer to accumulator word (ev_addusiaaw) . . . 37
Figure 10. Vector add word (__ev_addw) . . . 38
Figure 11. Vector all equal (__ev_all_eq) . . . 39
Figure 12. Vector all floating-point equal (__ev_all_fs_eq) . . . 40
Figure 13. Vector all floating-point greater than (__ev_all_fs_gt) . . . 41
Figure 14. Vector all floating-point less than (__ev_all_fs_lt) . . . 42
Figure 15. Vector all floating-point test equal (__ev_all_fs_tst_eq) . . . 43
Figure 16. Vector all floating-point test greater than (__ev_all_fs_tst_gt) . . . 44
Figure 17. Vector all floating-point test less than (__ev_all_fs_tst_lt) . . . 45
Figure 18. Vector all greater than signed (__ev_all_gts) . . . 46
Figure 19. Vector all greater than unsigned (__ev_all_gtu) . . . 47
Figure 20. Vector all less than signed (__ev_all_lts) . . . 48
Figure 21. Vector all less than unsigned (__ev_all_ltu) . . . 49
Figure 22. Vector AND (__ev_and) . . . 50
Figure 23. Vector AND with complement (__ev_andc) . . . 51
Figure 24. Vector any equal (__ev_any_eq) . . . 52
Figure 25. Vector any floating-point equal (__ev_any_fs_eq) . . . 53
Figure 26. Vector any floating-point greater than (__ev_any_fs_gt) . . . 54
Figure 27. Vector any floating-point less than (__ev_any_fs_lt) . . . 55
Figure 28. Vector any floating-point test equal (__ev_any_fs_tst_eq) . . . 56
Figure 29. Vector any floating-point test greater than (__ev_any_fs_tst_gt) . . . 57
Figure 30. Vector any floating-point test less than (__ev_any_fs_tst_lt) . . . 58
Figure 31. Vector any greater than signed (__ev_any_gts) . . . 59
Figure 32. Vector any greater than unsigned (__ev_any_gtu) . . . 60
Figure 33. Vector any less than signed(__ev_any_lts) . . . 61
Figure 34. Vector any less than unsigned (__ev_any_ltu) . . . 62
Figure 35. Vector count leading signed bits word (__ev_cntlsw) . . . 63
Figure 36. Vector Count Leading Signed Bits Word (__ev_cntlzw) . . . 64
Figure 37. Vector divide word signed (__ev_divws) . . . 66
Figure 38. Vector divide word unsigned (__ev_divwu) . . . 67
Figure 39. Vector equivalent (__ev_eqv) . . . 68
Figure 40. Vector extend sign byte (__ev_extsb) . . . 69
Figure 41. Vector extend sign half word (__ev_extsh) . . . 70
Figure 42. Vector floating-point absolute value (__ev_fsabs) . . . 71
Figure 43. Vector floating-point add (__ev_fsadd) . . . 72
Figure 44. Vector convert floating-point from signed fraction (__ev_fscfsf) . . . 73
Figure 45. Vector convert floating-point from signed integer (__ev_fscfsi) . . . 74
Figure 46. Vector convert floating-point from unsigned fraction (__ev_fscfuf) . . . 75
Figure 47. Vector convert floating-point from unsigned integer (__ev_fscfui) . . . 76
Figure 48. Vector convert floating-point to signed fraction (__ev_x) . . . 77

Figure 49. Vector convert floating-point to signed integer (__ev_fsctsi) . . . 78
Figure 50. Vector convert floating-point to signed integer with roundtoward zero (__ev_fsctsiz) 79
Figure 51. Vector convert floating-point to unsigned fraction (__ev_fsctuf) . . . 80
Figure 52. Vector convert floating-point to unsigned integer (__ev_fsctui) . . . 81
Figure 53. Vector convert floating-point to unsigned integer with roundtoward zero (__ev_fsctuiz) . . 82
Figure 54. Vector floating-point divide (__ev_fsdiv) . . . 83
Figure 55. Vector floating-point multiply (__ev_fsmul) . . . 84
Figure 56. Vector floating-point negative absolute value (__ev_fsnabs) . . . 85
Figure 57. Vector floating-point negate (__ev_fsneg) . . . 86
Figure 58. Vector Floating-Point subtract (__ev_fssub) . . . 87
Figure 59. __ev_ldd results in big- and little-endian modes . . . 88
Figure 60. __ev_lddx results in big- and little-endian modes . . . 89
Figure 61. __ev_ldh results in big- and little-endian modes . . . 90
Figure 62. __ev_ldhx results in big- and little-endian modes . . . 91
Figure 63. __ev_ldw results in big- and little-endian modes . . . 92
Figure 64. __ev_ldwx results in big- and little-endian modes . . . 93
Figure 65. __ev_lhhesplat results in big- and little-endian modes . . . 94
Figure 66. __ev_lhhesplatx results in big- and little-endian modes . . . 95
Figure 67. __ev_lhhossplat results in big- and little-endian modes . . . 96
Figure 68. __ev_lhhossplatx results in big- and little-endian modes . . . 97
Figure 69. __ev_lhhousplat results in big- and little-endian modes . . . 98
Figure 70. __ev_lhhousplatx results in big- and little-endian modes . . . 99
Figure 71. Vector lower equal (__ev_lower_eq) . . . 100
Figure 72. Vector lower floating-point equal (__ev_lower_fs_eq) . . . 101
Figure 73. Vector lower floating-point greater than (__ev_lower_fs_gt) . . . 102
Figure 74. Vector lower floating-point less than (__ev_lower_fs_lt) . . . 103
Figure 75. Vector lower floating-point test equal (__ev_lower_fs_tst_eq) . . . 104
Figure 76. Vector lower floating-point test greater than (__ev_lower_fs_tst_gt) . . . 105
Figure 77. Vector lower floating-point test less than (__ev_lower_fs_tst_lt) . . . 106
Figure 78. Vector lower greater than signed (__ev_lower_gts) . . . 107
Figure 79. Vector lower greater than unsigned (__ev_lower_gtu) . . . 108
Figure 80. Vector lower less than signed (__ev_lower_lts) . . . 109
Figure 81. Vector lower less than unsigned (__ev_lower_ltu) . . . 110
Figure 82. __ev_lwhe results in big- and little-endian modes . . . 111
Figure 83. __ev_lwhex results in big- and little-endian modes . . . 112
Figure 84. __ev_lwhos results in big- and little-endian modes . . . 113
Figure 85. __ev_lwhosx results in big- and little-endian modes . . . 114
Figure 86. __ev_lwhou results in big- and little-endian modes . . . 115
Figure 87. __ev_lwhoux results in big- and little-endian modes . . . 116
Figure 88. __ev_lwhsplat results in big- and little-endian modes . . . 117
Figure 89. __ev_lwhsplatx results in big- and little-endian modes . . . 118
Figure 90. __ev_lwwsplat results in big- and little-endian modes . . . 119
Figure 91. __ev_lwwsplatx results in big- and little-endian modes . . . 120
Figure 92. High-order element merging (__ev_mergehi) . . . 121
Figure 93. High-order element merging (__ev_mergehilo) . . . 122
Figure 94. Low-order element merging (__ev_mergelo) . . . 123
Figure 95. Low-order element merging (__ev_mergelohi) . . . 124
Figure 96. __ev_mhegsmfaa (even form) . . . 125
Figure 97. __ev_mhegsmfan (even form) . . . 126
Figure 98. __ev_mhegsmiaa (even form) . . . 127
Figure 99. __ev_mhegsmian (even form) . . . 128
Figure 100. __ev_mhegumfaa (even form) . . . 129

Figure 101. __ev_mhegumiaa (even form) . . . 130
Figure 102. __ev_mhegumfan (even form) . . . 131
Figure 103. __ev_mhegumian (even form) . . . 132
Figure 104. Even multiply of two signed modulo fractional elements (to accumulator) (__ev_mhesmf) . . . 133
Figure 105. Even form of vector half-word multiply (__ev_mhesmfaaw) . . . 134
Figure 106. Even form of vector half-word multiply (__ev_mhesmfanw) . . . 135
Figure 107. Even form for vector multiply (to accumulator) (__ev_mhesmi) . . . 136
Figure 108. Even form of vector half-word multiply (__ev_mhesmiaaw) . . . 137
Figure 109. Even form of vector half-word multiply (__ev_mhesmianw) . . . 138
Figure 110. Even multiply of two signed saturate fractional elements (to accumulator) (__ev_mhessf) . . . 140
Figure 111. Even form of vector half-word multiply (__ev_mhessfaaw) . . . 142
Figure 112. Even form of vector half-word multiply (__ev_mhessfanw) . . . 144
Figure 113. Even form of vector half-word multiply (__ev_mhessiaaw) . . . 145
Figure 114. Even form of vector half-word multiply (__ev_mhessianw) . . . 146
Figure 115. Vector multiply half words, even, unsigned, modulo, fractional (to accumulator) (__ev_mheumf) . . . 147
Figure 116. Vector multiply half words, even, unsigned, modulo, integer (to accumulator) (__ev_mheumi) . . . 148
Figure 117. Even form of vector half-word multiply (__ev_mheumfaaw) . . . 149
Figure 118. Even form of vector half-word multiply (__ev_mheumiaaw) . . . 150
Figure 119. Even form of vector half-word multiply (__ev_mheumfanw) . . . 151
Figure 120. Even form of vector half-word multiply (__ev_mheumianw) . . . 152
Figure 121. Even form of vector half-word multiply (__ev_mheusfaaw) . . . 154
Figure 122. Even form of vector half-word multiply (__ev_mheusiaaw) . . . 155
Figure 123. Even form of vector half-word multiply (__ev_mheusfanw) . . . 157
Figure 124. Even form of vector half-word multiply (__ev_mheusianw) . . . 159
Figure 125. __ev_mhogsmfaa (odd form) . . . 160
Figure 126. __ev_mhogsmfan (odd form) . . . 161
Figure 127. __ev_mhogsmiaa (odd form) . . . 162
Figure 128. __ev_mhogsmian (odd form) . . . 163
Figure 129. __ev_mhogumfaa (odd form) . . . 164
Figure 130. __ev_mhogumiaa (odd form) . . . 165
Figure 131. __ev_mhogumfan (odd form) . . . 166
Figure 132. __ev_mhogumian (odd form) . . . 167
Figure 133. Vector multiply half words, odd, signed, modulo, fractional (to accumulator) (__ev_mhosmf) . . . 168
Figure 134. Odd form of vector half-word multiply (__ev_mhosmfaaw) . . . 169
Figure 135. Odd form of vector half-word multiply (__ev_mhosmfanw) . . . 170
Figure 136. Vector multiply half words, odd, signed, modulo, integer (to accumulator) (__ev_mhosmi) . . . 171
Figure 137. Odd form of vector half-word multiply (__ev_mhosmiaaw) . . . 172
Figure 138. Odd form of vector half-word multiply (__ev_mhosmianw) . . . 173
Figure 139. Vector multiply half words, odd, signed, saturate, fractional (to Accumulator) (__ev_mhossf) . . . 175
Figure 140. Odd form of vector half-word multiply (__ev_mhossfaaw) . . . 177
Figure 141. Odd form of vector half-word multiply (__ev_mhossfanw) . . . 179
Figure 142. Odd form of vector half-word multiply (__ev_mhossiaaw) . . . 180
Figure 143. Odd form of vector half-word multiply (__ev_mhossianw) . . . 181
Figure 144. Vector multiply half words, odd, unsigned, modulo, fractional (to accumulator) (__ev_mhoumf) . . . 182

Figure 145. Vector multiply half words, odd, unsigned, modulo, integer (to Accumulator) (__ev_mhoumi) . . . 183
Figure 146. Odd form of vector half-word multiply (__ev_mhoumfaaw) . . . 184
Figure 147. Odd form of vector half-Word multiply (__ev_mhoumiaaw) . . . 185
Figure 148. Odd form of vector half-word multiply (__ev_mhoumfanw) . . . 186
Figure 149. Odd form of vector half-word multiply (__ev_mhoumianw). . . . 187
Figure 150. Odd form of vector half word multiply (__ev_mhousfaaw) . . . 189
Figure 151. Odd form of vector half word multiply (__ev_mhousiaaw) . . . 191
Figure 152. Odd form of vector half word multiply (__ev_mhousfanw) . . . 193
Figure 153. Odd form of vector half word multiply (__ev_mhousianw) . . . 195
Figure 154. Initialize accumulator (__ev_mra). . . . 196
Figure 155. Vector multiply word high signed, modulo, fractional (to accumulator) (__ev_mwhsmf) . 197
Figure 156. Vector multiply word high signed, modulo, integer (to Accumulator) (__ev_mwhsmi) . . . 198
Figure 157. Vector multiply word high signed, saturate, fractional (to Accumulator)(__ev_mwhssf). . 200
Figure 158. Vector multiply word high unsigned, modulo, integer (to accumulator) (__ev_mwhumi) . 201
Figure 159. Vector multiply word high unsigned, modulo, integer (to accumulator) (__ev_mwhumi) . 202
Figure 160. Vector multiply word low signed, modulo, integer and accumulate in words (__ev_mwlsmiaaw) . . . 203
Figure 161. Vector multiply word low signed, modulo, integer and accumulate negative in words (__ev_mwlsmianw) . . . 204
Figure 162. Vector multiply word low signed, saturate, integer and accumulate in words (__ev_mwlssiaaw) . . . 205
Figure 163. Vector multiply word low signed, saturate, integer and accumulate negative in words (__ev_mwlssianw) . . . 206
Figure 164. Vector multiply word low unsigned, modulo, integer (__ev_mwlumi) . . . 207
Figure 165. Vector multiply word low unsigned, modulo, integer and accumulate in words (__ev_mwlumiaaw) . . . 208
Figure 166. Vector multiply word low unsigned, modulo, integer and accumulate negative in words (__ev_mwlumianw) . . . 209
Figure 167. Vector multiply word low unsigned, saturate, integer and accumulate in words (__ev_mwlusiaaw) . . . 210
Figure 168. Vector multiply word low unsigned, saturate, integer and accumulate negative in words (__ev_mwlusianw) . . . 211
Figure 169. Vector multiply word signed, modulo, fractional (to Accumulator) (__ev_mwsmf) . . . 212
Figure 170. Vector multiply word signed, modulo, fractional and Accumulate (__ev_mwsmfaa) 213
Figure 171. Vector multiply word signed, modulo, fractional, and accumulate Negative (__ev_mwsmfan) . . . 214
Figure 172. Vector multiply word signed, modulo, integer (to Accumulator) (__ev_mwsmi) 215
Figure 173. Vector multiply word signed, modulo, integer and accumulate (__ev_mwsmiaa) 216
Figure 174. Vector multiply word signed, modulo, integer and accumulate Negative (__ev_mwsmian) . . . 217
Figure 175. Vector multiply word signed, saturate, fractional (to Accumulator) (__ev_mwssf) 218
Figure 176. Vector multiply word signed, saturate, fractional and accumulate (__ev_mwssfaa). 220
Figure 177. Vector multiply word signed, saturate, fractional and accumulate Negative (__ev_mwssfan) . . . 222
Figure 178. Vector multiply word unsigned, modulo, integer (to Accumulator) (__ev_mwumi) 223
Figure 179. Vector multiply word unsigned, modulo, integer and accumulate (__ev_mwumiaa) 224
Figure 180. Vector multiply word unsigned, modulo, integer and accumulate Negative (__ev_mwumian) . . . 225
Figure 181. Vector NAND (__ev_nand) . . . 226
Figure 182. Vector negate (__ev_neg) . . . 227

Figure 183. Vector NOR (__ev_nor) . . . 228
Figure 184. Vector OR (__ev_or) . . . 229
Figure 185. Vector OR with complement (__ev_orc) . . . 230
Figure 186. Vector rotate left word (__ev_rlw) . . . 231
Figure 187. Vector rotate left word immediate (__ev_rlwi) . . . 232
Figure 188. Vector round word (__ev_rndw) . . . 233
Figure 189. Vector select equal (__ev_select_eq) . . . 234
Figure 190. Vector select Floating-Point equal (__ev_select_fs_eq) . . . 235
Figure 191. Vector select Floating-Point greater than (__ev_select_fs_gt) . . . 236
Figure 192. Vector select Floating-Point less than (__ev_select_fs_lt) . . . 237
Figure 193. Vector select Floating-Point test equal (__ev_select_fs_tst_eq) . . . 238
Figure 194. Vector select Floating-Point test greater than (__ev_select_fs_tst_gt) . . . 239
Figure 195. Vector select Floating-Point test less than (__ev_select_fs_tst_lt) . . . 240
Figure 196. Vector select greater than signed (__ev_select_gts) . . . 241
Figure 197. Vector select greater than unsigned (__ev_select_gtu) . . . 242
Figure 198. Vector select less than signed (__ev_select_lts) . . . 243
Figure 199. Vector select less than unsigned (__ev_select_ltu) . . . 244
Figure 200. Vector shift left word (__ev_slw) . . . 245
Figure 201. Vector shift left word immediate (__ev_slwi) . . . 246
Figure 202. Vector splat fractional immediate (__ev_splatfi) . . . 247
Figure 203. __ev_splati sign extend . . . 248
Figure 204. Vector shift right word immediate signed (__ev_srwis) . . . 249
Figure 205. Vector shift right word immediate unsigned (__ev_srwiu) . . . 250
Figure 206. Vector shift right word signed (__ev_srws) . . . 251
Figure 207. Vector shift right word unsigned (__ev_srwu) . . . 252
Figure 208. __ev_stdd results in big- and little-endian modes . . . 253
Figure 209. __ev_stdd[x] results in Big- and Little-Endian modes . . . 254
Figure 210. __ev_stdh results in Big- and Little-Endian modes . . . 255
Figure 211. __ev_stdhx results in Big- and Little-Endian modes . . . 256
Figure 212. __ev_stdw results in Big- and Little-Endian modes . . . 257
Figure 213. __ev_stdwx results in Big- and Little-Endian modes . . . 258
Figure 214. __ev_stwhe results in Big- and Little-Endian modes . . . 259
Figure 215. __ev_stwhex results in Big- and Little-Endian modes . . . 260
Figure 216. __ev_stwho results in Big- and Little-Endian modes . . . 261
Figure 217. __ev_stwhox results in Big- and Little-Endian modes . . . 262
Figure 218. __ev_stwwe results in Big- and Little-Endian modes . . . 263
Figure 219. __ev_stwwex results in Big- and Little-Endian modes . . . 264
Figure 220. __ev_stwwo results in Big- and Little-Endian modes . . . 265
Figure 221. __ev_stwwox results in Big- and Little-Endian modes . . . 266
Figure 222. Vector subtract signed, modulo, integer to accumulator Word (__ev_subfsmiaaw) . . . 267
Figure 223. Vector subtract signed, saturate, integer to accumulator Word (__ev_subfssiaaw) . . . 268
Figure 224. Vector subtract unsigned, modulo, integer to accumulator Word (__ev_subfumiaaw) . . . 269
Figure 225. Vector subtract unsigned, saturate, integer to accumulator Word (__ev_subfusiaaw) . . . 270
Figure 226. Vector subtract from word (__ev_subfw) . . . 271
Figure 227. Vector subtract immediate from word (__ev_subifw) . . . 272
Figure 228. Vector upper Equal(__ev_upper_eq) . . . 273
Figure 229. Vector upper Floating-Point Equal(__ev_upper_fs_eq) . . . 274
Figure 230. Vector upper Floating-Point greater than (__ev_upper_fs_gt) . . . 275
Figure 231. Vector upper Floating-Point less than (__ev_upper_fs_lt) . . . 276
Figure 232. Vector upper Floating-Point test equal (__ev_upper_fs_tst_eq) . . . 277
Figure 233. Vector upper Floating-Point test greater than (__ev_upper_fs_tst_gt) . . . 278
Figure 234. Vector upper Floating-Point test less than (__ev_upper_fs_tst_lt) . . . 279

Figure 235. Vector upper greater than signed (__ev_upper_gts) 280
Figure 236. Vector upper greater than unsigned (__ev_upper_gtu) 281
Figure 237. Vector upper less than signed (__ev_upper_lts) 282
Figure 238. Vector upper less than unsigned (__ev_upper_ltu) 283
Figure 239. Vector XOR (__ev_xor) 284
Figure 240. Big-endian word ordering 295
Figure 241. Big-endian half-word ordering. 295
Figure 242. Signal processing and embedded floating-point status and control register (SPEFSCR) 299

1 Overview

This document defines a programming model to use with the signal processing engine (SPE) auxiliary processing unit (APU). This document describes three types of programming interfaces:

- A high-level language interface that is intended for use within programming languages, such as C or C++
- An application binary interface (ABI) that defines low-level coding conventions
- An assembly language interface

1.1 High-level language interface

The high-level language interface enables programmers to use the SPE APU from programming languages such as C and C++, and describes fundamental data types for the SPE programming model. See *Chapter 2: High-level language interface on page 15*," for details about this interface.

1.2 Application binary interface (ABI)

The SPE programming model extends the existing PowerPC ABIs. The extension is independent of the endian mode. The ABI reviews the data types and register usage conventions for vector register files and describes setup of the stack frame. Save and Restore functions for the vector register are included in the ABI section to advocate uniformity of method among compilers for saving and restoring vector registers.

The *AltiVec™ Technology Programming Interface Manual*, provides the valid set of argument types for specific AltiVec operations and predicates as well as specific AltiVec instructions that are generated for that set of arguments. The AltiVec operations and predicates are organized alphabetically in *Chapter 4: Additional operations on page 295*"

2 High-level language interface

2.1 Introduction

This document defines a programming model to use with the signal processing engine (SPE) auxiliary processing unit (APU) instruction set. The purpose of the programming model is to give users the ability to write code that utilizes the APU in a high-level language, such as C or C++.

Users should not be concerned with issues such as register allocation, scheduling, and conformity to the underlying ABI, which are all associated with writing code at the assembly level.

2.2 High-level language interface

The high-level language interface for SPE is intended to accomplish the following goals:

- Provide an efficient and expressive mechanism to access SPE functionality from programming languages such as C and C++
- Define a minimal set of language extensions that unambiguously describe the intent of the programmer while minimizing the impact on existing PowerPC compilers and development tools
- Define a minimal set of library extensions that are needed to support SPE

2.2.1 Data types

Table 1 describes a set of fundamental data types that the SPE programming model introduces.

Note: The type _ev64_ stands for embedded vector of data width 64 bits.

Table 1. Data types

New C/C++ type	Interpretation of contents	Values
__ev64_u16__	4 unsigned 16-bit integers	0...65535
__ev64_s16__	4 signed 16-bit integers	-32768...32767
__ev64_u32__	2 unsigned 32-bit integers	$0...2^{32} - 1$
__ev64_s32__	2 signed 32-bit integers	$-2^{31}...2^{31} - 1$
__ev64_u64__	1 unsigned 64-bit integer	$0...2^{64} - 1$
__ev64_s64__	1 signed 64-bit integer	$-2^{63}...2^{63} - 1$
__ev64_fs__	2 floats	IEEE-754 single-precision values
__ev64_opaque__	any of the above	—

The __ev64_opaque__ data type is an opaque data type that can represent any of the specified __ev64_*__ data types. All of the __ev64_*__ data types are available to programmers.

2.2.2 Alignment

Refer to the e500 ABI for full alignment details.

Alignment of __ev64_*__ types

A defined data item of any __ev64_*__ data type in memory is always aligned on an 8-byte boundary. A pointer to any __ev64_*__ data type always points to an 8-byte boundary. The compiler is responsible for aligning any __ev64_*__ data types on an 8-byte boundary. When __ev64_*__ data is correctly aligned, a program is incorrect if it attempts to dereference a pointer to an __ev64_*__ type if the pointer does not contain an 8-byte aligned address.

In the SPE architecture, an unaligned load/store causes an alignment exception.

Alignment of aggregates and unions containing __ev64_*__ types

Aggregates (structures and arrays) and unions containing __ev64_*__ variables must be aligned on 8-byte boundaries and their internal organization must be padded, if necessary, so that each internal __ev64_*__ variable is aligned on an 8-byte boundary.

2.2.3 Extensions of C/C++ operators for the new types

Most C/C++ operators do not permit any of their arguments to be one of the __ev64_*__ types. Let 'a' and 'b' be variables of any __ev64_*__ type, and 'p' be a pointer to any __ev64_*__ type. The normal C/C++ operators are extended to include the operations in the following sections.

sizeof()

The functions sizeof(a) and sizeof(*p) return 8.

Assignment

Assignment is allowed only if both the left- and right-hand sides of an expression are the same __ev64_*__ type. For example, the expression a=b is valid and represents assignment of 'b' to 'a'. The one exception to the rule occurs when 'a' or 'b' is of type __ev64_opaque__. Let 'o' be of type __ev64_opaque__ and let 'a' be of any __ev64_*__ type.

The assignments a=o and o=a are allowed and have implicit casts. Otherwise, the expression is invalid, and the compiler must signal an error.

Address operator

The operation &a is valid if 'a' is an __ev64_*__ type. The result of the operation is a pointer to 'a'.

Pointer arithmetic

The usual pointer arithmetic can be performed on p. In particular, p+1 is a pointer to the next __ev64_*__ element after p.

Pointer dereferencing

If 'p' is a pointer to an __ev64_*__ type, *p implies either a 64-bit SPE load from the address, equivalent to the intrinsic __ev_ldd(p,0), or a 64-bit SPE store to that address,

equivalent to the intrinsic __ev_stdd(p,0). Dereferencing a pointer to a non-__ev64_*__ type produces the standard behavior of either a load or a copy of the corresponding type.

Alignment of __ev64_*__ types on page 16, describes unaligned accesses.

Type casting

Pointers to __ev64_*__ and existing types may be cast back and forth to each other. Casting a pointer to an __ev64_*__ type represents an (unchecked) assertion that the address is 8-byte aligned.

Casting from a integral type to a pointer to an __ev64_*__ type is allowed.

For example:

```
__ev64_u16__ *a = (__ev64_u16__ *) 0x48;
```

Casting between __ev64_*__ types and existing types is not allowed.

Casting between __ev64_*__ types and pointers to existing types is not allowed.

The behaviors expected from such casting are provided instead of using intrinsics.

The intrinsics provide the ability to extract existing data types out of __ev64_*__ variables as well as the ability to insert into and/or create __ev64_*__ variables from existing data types. Normal C casts provide casts from one __ev64_*__ type to another.

An implicit cast is performed when going to __ev64_opaque__ from any other __ev64_*__ type. An implicit cast occurs when going from __ev64_opaque__ to any other __ev64_*__ type. The implicit casts that occur when going between __ev64_opaque__ and any other __ev64_*__ type also apply to pointers of type __ev64_opaque__. When casting between any two __ev64_*__ types not including __ev64_opaque__, an explicit cast is required. When casting between pointers to any two __ev64_*__ types not including __ev64_opaque__, an explicit cast is required. No cast or promotion performs a conversion; the bit pattern of the result is the same as the bit pattern of the argument that is cast.

2.2.4 New operators

New operators are introduced to construct __ev64_*__ values and allow full access to the functionality that the SPE architecture provides.

__ev64_*__ Initialization and literals

The __ev64_opaque__ type is the only __ev64_*__ type that cannot be initialized. The remaining __ev64_*__ types can be initialized using the C99 array initialization syntax. Each type is treated as an array of the specified data contents of the appropriate size. The following code exemplifies the initialization of these types:

```
__ev64_u16__ a = { 0, 1, 2, 3 };
__ev64_s16__ b = { -1, -2, -3, 4 };
__ev64_u32__ c = { 3, 4 };
__ev64_s32__ d = { -2, 4 };
__ev64_u64__ e = { 17  } ;
__ev64_s64__ f = { 23 } ;
__ev64_fs__ g = { 2.4, -3.2 };

c = __ev_addw(a, (__ev64_s16__){2,1,5,2});
```

New operators representing SPE operations

New operators are introduced to allow full access to the functionality that the SPE architecture provides. Language structures that parse like function calls represent these operators in the programming language.

The names associated with these operations are all prefixed with "__ev_". The appearance of one of these forms can indicate one of the following:

- A specific SPE operation, like __ev_addw(__ev64_opaque__ a, __ev64_opaque__ b)
- A predicate computed from a SPE operation, like __ev_all_eq(__ev64_opaque__ a, __ev64_opaque__ b)
- Creation, insertion, extraction of __ev64_opaque__ values

Each operator representing an SPE operation takes a list of arguments representing the input operands (in the order in which they are shown in the architecture specification) and returns a result that could be void. The programming model restricts the operand types that are permitted for each SPE operation. Predicate intrinsics handle comparison operations in the SPE programming model.

Each compare operation has the following predicate intrinsics associated with it:

- _any_
- _all_
- _upper_
- _lower_
- _select_

Each predicate returns an integer (0/1) with the result of the compare. The compiler allocates a CR field for use in the comparison and to optimize conditional statements.

2.2.5 Programming interface

This document does not prohibit or require an implementation to provide any set of include files to provide access to the intrinsics. If an implementation requires that an include file be used before the use of the intrinsics described in this document, that file should be <spe.h>.

This document does require that prototypes for the additional library routines described are accessible by means of the include file <spe.h>. If an implementation should provide a __SPE__, define it with a nonzero value. That definition should not occur in the <spe.h> header file.

3 SPE operations

This chapter describes the following instructions:

- All instructions in the e500 core complex, including numerous instructions that Book E does not define.
- Book E instructions that are defined for 32-bit implementations, including many instructions that are not implemented on the e500 core complex.

3.1 Signal processing engine (SPE) APU registers

The SPE includes the following two registers:

- The signal processing and embedded floating-point status and control register (SPEFSCR), which is described in *Chapter 3.1.1*."
- A 64-bit accumulator, which is described in *Chapter 3.1.2*."

3.1.1 Signal processing and embedded floating-point status and control register (SPEFSCR)

The SPEFSCR, which is shown in *Figure 1*, is used for status and control of SPE instructions.

Figure 1. Signal processing and embedded floating-point status and control register (SPEFSCR)

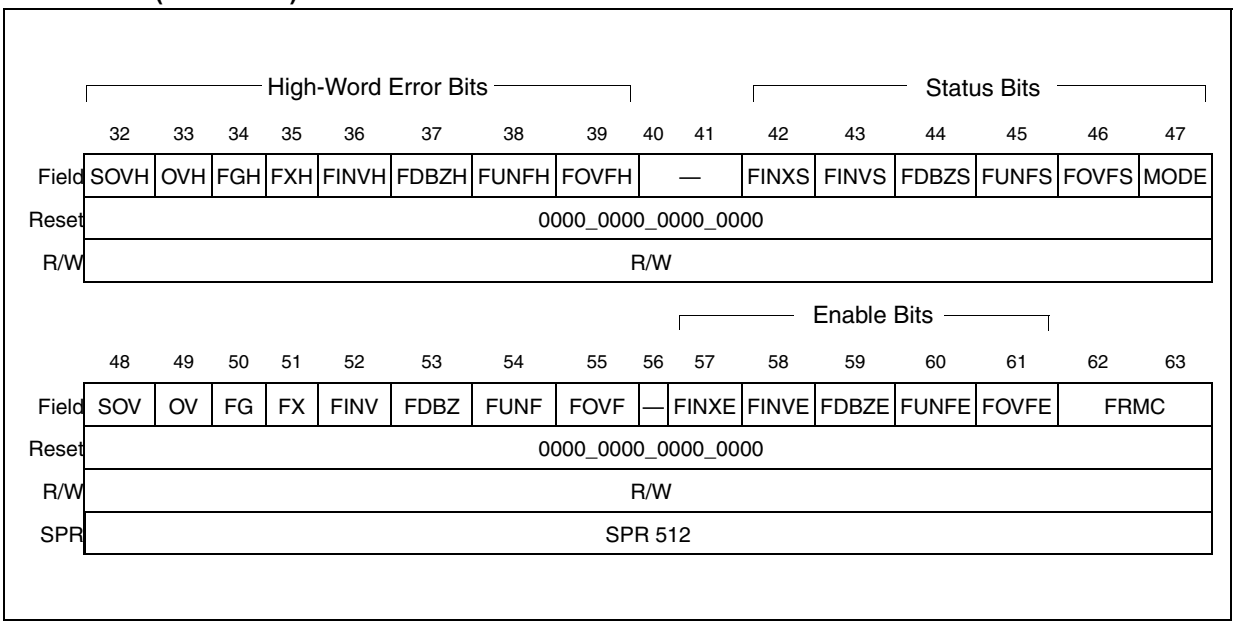


Table 2. SPEFSCR field descriptions

Bits	Name	Function
32	SOVH	Summary integer overflow high, which is set whenever an instruction other than **mtspr** sets OVH. SOVH remains set until a **mtspr[SPEFSCR]** clears it.
33	OVH	Integer overflow high. An overflow occurred in the upper half of the register while executing a SPE integer instruction.
34	FGH	Embedded floating-point guard bit high. Floating-point guard bit from the upper half. The value is undefined if the processor takes a floating-point exception caused by input error, floating-point overflow, or floating-point underflow.
35	FXH	Embedded floating-point sticky bit high. Floating bit from the upper half. The value is undefined if the processor takes a floating-point exception caused by input error, floating-point overflow, or floating-point underflow.
36	FINVH	Embedded floating-point invalid operation error high. Set when an input value on the high side is a NaN, Inf, or Denorm. Also set on a divide if both the dividend and divisor are zero.
37	FDBZH	Embedded floating-point divide by zero error high. Set if the dividend is non-zero and the divisor is zero.
38	FUNFH	Embedded floating-point underflow error high
39	FOVFH	Embedded floating-point overflow error high
40–41	—	Reserved and should be cleared
42	FINXS	Embedded floating-point inexact sticky. FINXS = FINXS | FGH | FXH | FG | FX
43	FINVS	Embedded floating-point invalid operation sticky. Location for software to use when implementing true IEEE floating-point.
44	FDBZS	Embedded floating-point divide by zero sticky. FDBZS = FDBZS | FDBZH | FDBZ
45	FUNFS	Embedded floating-point underflow sticky. Storage location for software to use when implementing true IEEE floating-point.
46	FOVFS	Embedded floating-point overflow sticky. Storage location for software to use when implementing true IEEE floating-point.
47	MODE	Embedded floating-point mode (read-only on e500)
48	SOV	Integer summary overflow. Set whenever an SPE instruction other than **mtspr** sets OV. SOV remains set until **mtspr[SPEFSCR]** clears it.
49	OV	Integer overflow. An overflow occurred in the lower half of the register while a SPE integer instruction was executed.
50	FG	Embedded floating-point guard bit. Floating-point guard bit from the lower half. The value is undefined if the processor takes a floating-point exception caused by input error, floating-point overflow, or floating-point underflow.
51	FX	Embedded floating-point sticky bit. Floating bit from the lower half. The value is undefined if the processor takes a floating-point exception caused by input error, floating-point overflow, or floating-point underflow.
52	FINV	Embedded floating-point invalid operation error. Set when an input value on the high side is a NaN, Inf, or Denorm. Also set on a divide if both the dividend and divisor are zero.
53	FDBZ	Embedded floating-point divide by zero error. Set if the dividend is non-zero and the divisor is zero.
54	FUNF	Embedded floating-point underflow error

Table 2. SPEFSCR field descriptions (continued)

Bits	Name	Function
55	FOVF	Embedded floating-point overflow error
56	—	Reserved and should be cleared
57	FINXE	Embedded floating-point inexact enable
58	FINVE	Embedded floating-point invalid operation/input error exception enable 0: Exception disabled 1: Exception enabled If the exception is enabled, a floating-point data exception is taken if a floating-point instruction sets FINV or FINVH.
59	FDBZE	Embedded floating-point divide-by-zero exception enable 0: Exception disabled 1: Exception enabled If the exception is enabled, a floating-point data exception is taken if a floating-point instruction sets FDBZ or FDBZH.
60	FUNFE	Embedded floating-point underflow exception enable 0: Exception disabled 1: Exception enabled If the exception is enabled, a floating-point data exception is taken if a floating-point instruction sets FUNF or FUNFH.
61	FOVFE	Embedded floating-point overflow exception enable 0: Exception disabled 1: Exception enabled If the exception is enabled, a floating-point data exception is taken if a floating-point instruction sets FOVF or FOVFH.
62–63	FRMC	Embedded floating-point rounding mode control 00: Round to nearest 01: Round toward zero 10: Round toward +infinity 11: Round toward –infinity

3.1.2 Accumulator (ACC)

The 64-bit architectural accumulator register shown in *Figure 2* holds the results of multiply accumulate (MAC) forms of SPE integer instructions. The ACC allows back-to-back execution of dependent MAC instructions that are in inner loops of DSP code such as FIR filters. The ACC is partially visible to the programmer; its results need not be read explicitly to be used. Instead, the results are always copied into a 64-bit destination GPR specified by the instruction. The ACC, however, must be explicitly cleared when starting a new MAC loop. Depending on the instruction type, the ACC can hold either a 64-bit value or a vector of two 32-bit elements.

The Initialize Accumulator instruction (**evmra**), which is described in the Instruction Set chapter of *Programmer's reference manual for Book E processors,* initializes the ACC.

Figure 2. Accumulator (ACC)

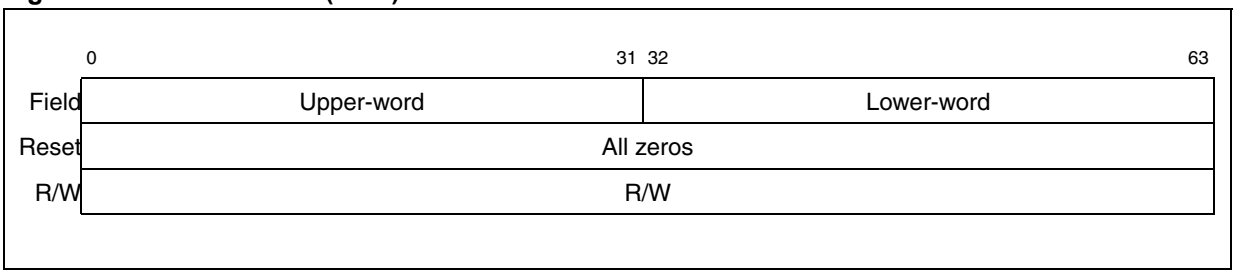


Table 3. ACC field descriptions

Bits	Name	Function
0–31	Upper word	Holds the upper-word accumulate value for SPE multiply with accumulate instructions
32–63	Lower word	Holds the lower-word accumulate value for SPE multiply with accumulate instructions

3.2 Notation

Table 4 shows definitions and notation that appear throughout this document.

Table 4. Notation conventions

Symbol	Meaning
X_p	Bit p of register/field X
$X_{p:q}$	Bits p through q of register/field X
$X_{p\ q\ ...}$	Bits p, q,... of register/field X
¬X	The ones complement of the contents of X
Field i	Bits 4×i through 4×i+3 of a register
.	As the last character of an instruction mnemonic, this character indicates that the instruction records status information in certain fields of the condition register as a side effect of execution, as described in the Register Model chapter of *EREF: Programmer's reference manual for Book E processors*.
||	Describes the concatenation of two values. For example, 010 || 111 is the same as 010111.
x^n	x raised to the n^{th} power.
^nx	Replication of x, n times (i.e., x concatenated to itself n–1 times). $^n0$ and $^n1$ are special cases: $^n0$ means a field of n bits with each bit equal to 0. Thus, ^{5}0 is equivalent to 0b0_0000. $^n1$ means a field of n bits with each bit equal to 1. Thus, ^{5}1 is equivalent to 0b1_1111.
/, //, ///,	Reserved field in an instruction or in a register. Each bit and field in instructions, in status and control registers (such as the XER), and in SPRs is defined, allocated, or reserved.

3.3 Instruction fields

Table 5 describes instruction fields.

Table 5. Instruction field descriptions

Field	Description
AA (30)	Absolute address bit. 0: The immediate field represents an address relative to the current instruction address. For I-form branch instructions, the effective address of the branch target is the sum 320 ll (CIA+EXTS(LIll0b00))32–63. For B-form branch instructions, the effective address of the branch target is the sum 320 ll (CIA+EXTS(BDll0b00))32–63. For I-form branch extended instructions, the effective address of the branch target is the sum CIA+EXTS(LIll0b00). For B-form branch extended instructions, the effective address of the branch target is the sum CIA+EXTS(BDll0b00). 1: The immediate field represents an absolute address. For I-form branch instructions, the effective address of the branch target is the value 320 ll EXTS(LIll0b00)32–63. For B-form branch instructions, the effective address of the branch target is the value 320 ll EXTS(BDll0b00)32–63. For I-form branch extended instructions, the effective address of the branch target is the value EXTS(LIll0b00). For B-form branch extended instructions, the effective address of the branch target is the value EXTS(BDll0b00).
crbA (11–15)	Specifies a condition register bit to be used as a source
crbB (16–20)	Specifies a condition register bit to be used as a source
crbD (16–29)	Immediate field specifying a 14-bit signed two's complement branch displacement that is concatenated on the right with 0b00 and sign-extended to 64 bits.
crfD (6–8)	Specifies a CR field to be used as a target
crfS (11–13)	Specifies a CR field to be used as a source
BI (11–15)	Specifies a condition register bit to be used as the condition of a branch conditional instruction
BO (6–10)	Specifies options for branch conditional instructions
crbD (6–10)	Specifies a CR bit for use as a target
CT (6–10)	Cache touch instructions (**dcbt**, **dcbtst**, and **icbt**) use this field to specify the target portion of the cache facility to place the prefetched data or instructions. This field is implementation-dependent.
D (16–31)	Immediate field that specifies a 16-bit signed two's complement integer that is sign-extended to 64 bits
DE (16–27)	Immediate field that specifies a 12-bit signed two's complement integer that is sign-extended to 64 bits
DES (16–27)	Immediate field that specifies a 12-bit signed two's complement integer that is concatenated on the right with 0b00 and sign-extended to 64 bits

Table 5. Instruction field descriptions (continued)

Field	Description
E (15)	Immediate field that specifies a 1-bit value that **wrteei** uses to place in MSR[EE] (external input enable bit)
CRM (12–19)	Field mask that identifies the condition register fields that the **mtcrf** instruction updates
LI (6–29)	Immediate field that specifies a 24-bit signed two's complement integer that is concatenated on the right with 0b00 and sign-extended to 64 bits
LK (31)	Link bit that indicates whether the link register (LR) is set. 0: Do not set the LR. 1: Set the LR. The sum of the value 4 and the address of the branch instruction is placed into the LR.
MB (21–25) and ME (26–30)	Fields that M-form rotate instructions use to specify a 64-bit mask consisting of 1s from bit MB+32 through bit ME+32 inclusive and 0s elsewhere
mb (26 || 21–25)	Used in MD-form and MDS-form rotate instructions to specify the first 1-bit of a 64-bit mask
me (26 || 21–25)	Used in MD-form and MDS-form rotate instructions to specify the last 1-bit of a 64-bit mask
MO (6–10)	Specifies the subset of memory accesses that a Memory Barrier instruction (**mbar**) ordered
NB (16–20)	Specifies the number of bytes to move in an immediate Move Assist instruction
OPCD (0–5)	Primary opcode field
rA (11–15)	Specifies a GPR to be used as a source or as a target
rB (16–20)	Specifies a GPR to be used as a source
Rc (31)	Record bit. 0: Do not alter the condition register. 1: Set condition register field 0 or field 1.
RS (6–10)	Specifies a GPR to be used as a source
rD (6–10)	Specifies a GPR to be used as a target
SH (16–20)	Specifies a shift amount in Rotate Word Immediate and Shift Word Immediate instructions
sh (30 || 16–20)	Specifies a shift amount in Rotate Doubleword Immediate and Shift Doubleword Immediate instructions
SIMM (16–31)	Immediate field that specifies a 16-bit signed integer
SPRN (16–20||11–15)	Specifies an SPR for **mtspr** and **mfspr** instructions
TO (6–10)	Specifies the conditions on which to trap
UIMM (16–31)	Immediate field that specifies a 16-bit unsigned integer
WS (18–20)	Specifies a word in the TLB entry that is being accessed
XO (21–29, 21–30, 22–30, 26–30, 27–29, 27–30, 28–31)	Extended opcode field

3.4 Description of instruction operation

A series of statements that use a semi-formal language at the register transfer level (RTL) describes the operation of most instructions. RTL uses the general notation that is shown in *Table 4* and *Table 5* and conventions that are specific to RTL, shown in *Table 6*. *Figure 3 on page 29* gives an example. Some of this notation is used in the formal descriptions of instructions.

The RTL descriptions cover the normal execution of the instruction, except that the standard settings of the condition register, integer exception register, floating-point status, and control register are not always shown. (Nonstandard setting of these registers, such as the setting of the condition register field 0 by the **stwcx** instruction, is shown.) The RTL descriptions do not cover all cases in which the interrupt may be invoked, or for which the results are boundedly undefined, and may not cover all invalid forms.

RTL descriptions specify the architectural transformation that the execution of an instruction performs. They do not imply any particular implementation.

Table 6. RTL notation

Notation	Meaning
$\leftarrow$	Assignment
$\leftarrow_f$	Assignment in which the data may be reformatted in the target location
$\neg$	NOT logical operator (one's complement)
$+$	Two's complement addition
$-$	Two's complement subtraction, unary minus
$\times$	Multiplication
$\div$	Division (yielding quotient)
$+_{dp}$	Floating-point addition, result rounded to double-precision
$-_{dp}$	Floating-point subtraction, result rounded to double-precision
$\times_{dp}$	Floating-point multiplication, product rounded to double-precision
$\div_{dp}$	Floating-point division quotient, rounded to double-precision
$+_{sp}$	Floating-point addition, result rounded to single-precision
$-_{sp}$	Floating-point subtraction, result rounded to single-precision
$\times_{sf}$	Signed fractional multiplication
$\times_{si}$	Signed integer multiplication
$\times_{sp}$	Floating-point multiplication, result rounded to single-precision
$\div_{sp}$	Floating-point division, result rounded to single-precision
$\times_{fp}$	Floating-point multiplication to infinite precision (no rounding)
$\times_{ui}$	Unsigned integer multiplication
FPSquareRoot-Double(x)	Floating-point $\sqrt{x}$, result rounded to double-precision
FPSquareRoot-Single(x)	Floating-point $\sqrt{x}$, result rounded to single-precision
FPReciprocal-Estimate(x)	Floating-point estimate of $\frac{1}{x}$

Table 6. RTL notation (continued)

Notation	Meaning
FPReciprocal-SquareRoot-Estimate(x)	Floating-point estimate of $\frac{1}{\sqrt{x}}$
Allocate-DataCache-Block(x)	If the block containing the byte addressed by x does not exist in the data cache, allocate a block in the data cache and set the contents of the block to 0.
Flush-DataCache-Block(x)	If the block containing the byte addressed by x exists in the data cache and is dirty, the block is written to main memory and is removed from the data cache.
Invalidate-DataCache-Block(x)	If the block containing the byte addressed by x exists in the data cache, the block is removed from the data cache.
Store-DataCache-Block(x)	If the block containing the byte addressed by x exists the data cache and is dirty, the block is written to main memory but may remain in the data cache.
Prefetch-DataCache-Block(x,y)	If the block containing the byte addressed by x does not exist in the portion of the data cache specified by y, the block in memory is copied into the data cache.
Prefetch-ForStore-DataCache-Block(x,y)	If the block containing the byte addressed by x does not exist in the portion of the data cache specified by y, the block in memory is copied into the data cache and made exclusive to the processor that is executing the instruction.
ZeroDataCache-Block(x)	The contents of the block containing the byte addressed by x in the data cache is cleared.
Invalidate-Instruction-CacheBlock(x)	If the block containing the byte addressed by x is in the instruction cache, the block is removed from the instruction cache.
Prefetch-Instruction-CacheBlock(x,y)	If the block containing the byte addressed by x does not exist in the portion of the instruction cache specified by y, the block in memory is copied into the instruction cache.
=, ≠	Equal to, Not Equal to relations
<, ≤, >, ≥	Signed comparison relations
$<_u$, $>_u$	Unsigned comparison relations
?	Unordered comparison relation
&, |	AND, OR logical operators
⊕, ≡	Exclusive OR, Equivalence logical operators ((a≡b) = (a⊕¬b))
ABS(x)	Absolute value of x
APID(x)	Returns an implementation-dependent information on the presence and status of the auxiliary processing extensions specified by x
CEIL(x)	Least integer ≥ x
CnvtFP32ToI32Sat(fp, signed,upper_lower,round, fractional)	Converts a 32 bit floating point number to a 32 bit integer if possible, otherwise it saturates.

Table 6. RTL notation (continued)

Notation	Meaning
CnvtI32ToFP32Sat (v,signed,upper_lower, fractional)	Converts a 32 bit integer to a 32 bit floating point number if possible, otherwise it saturates.
EXTS(x)	Result of extending x on the left with signed bits
EXTZ(x)	Result of extending x on the left with zeros
GPR(x)	General purpose register x
MASK(x, y)	Mask that has ones in bit positions x through y (wrapping if x>y) and zeros elsewhere
MEM(x,1)	Contents of the byte of memory located at address x
MEM(x,y)(for y={2,4,8})	Contents of y bytes of memory starting at address x. If big-endian memory, the byte at address x is the MSB and the byte at address x+y–1 is the LSB of the value being accessed. If little-endian memory, the byte at address x is the LSB and the byte at address x+y– 1 is the MSB of the value being accessed.
MOD(x,y)	Modulo y of x (remainder of x divided by y)
ROTL32(x, y)	Result of rotating the value x||x left y positions, where x is 32 bits long
SINGLE(x)	Result of converting x from floating-point double format to floating-point single format
SPREG(x)	Special-purpose register x
TRAP	Invoke a trap-type program interrupt
characterization	Reference to setting status bits in a standard way that is explained in the text
undefined	Undefined value that may vary between implementations and between different executions on the same implementation
CIA	Current instruction address, which is the address of the instruction that is described in RTL. Used by relative branches to set the next instruction address (NIA) and by branch instructions with LK=1 to set the LR. CIA does not correspond to any architected register.
NIA	Next instruction address, and the address of the next instruction to be executed. For a successful branch, the next instruction address is the branch target address: in RTL, indicated by assigning a value to NIA. For other instructions that cause non-sequential instruction fetching, the RTL is similar. For instructions that do not branch, and do not otherwise cause instruction fetching to be non-sequential, the next instruction address is CIA+4. NIA does not correspond to any architected register.
if … then … else …	Conditional execution indenting shows range; else is optional.
do	Do loop, indenting shows range. ‘To’ and/or ‘by’ clauses specify incrementing an iteration variable, and a ‘while’ clause gives termination conditions.
leave	Leave innermost do loop, or do loop described in leave statement

Table 7 summarizes precedence rules for RTL operators. Operators that are higher in the table are applied before those that are lower in the table. Operators at the same level in the table associate from left to right, from right to left, or not at all, as shown. (For example, the – operator associates from left to right, so a–b–c = (a–b)–c.) Using parentheses can increase clarity or override the evaluation order that the table implies; parenthesized expressions are evaluated before serving as parameters.

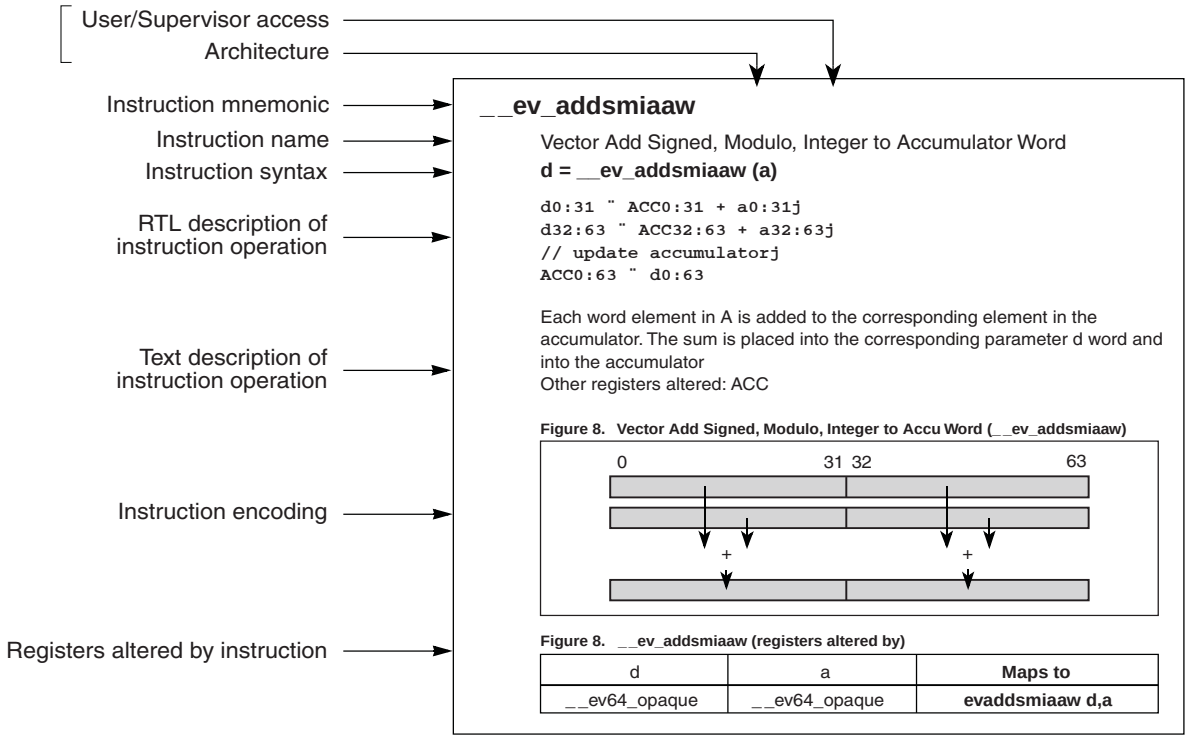
Table 7. Operator precedence

Operators	Associativity
Subscript, function evaluation	Left to right
Pre-superscript (replication), post-superscript (exponentiation)	Right to left
unary –, ¬	Right to left
×, ÷	Left to right
+, –	Left to right
||	Left to right
=, ≠, <, ≤, >, ≥, $<_u$, $>_u$, ?	Left to right
&, ⊕, ≡	Left to right
|	Left to right
: (range)	None
←	None

3.5 Intrinsics

The rest of this chapter describes individual instructions, which are listed in alphabetical order by mnemonic. *Figure 3* shows the format for instruction description pages.

Figure 3. Instruction description



3.5.1 Intrinsic definitions

For saturation, left shifts, and bit reversal, the pseudo RTL is provided here to more accurately describe those functions that are referenced in the instruction pseudo RTL.

Saturation

```
SATURATE(overflow, carry, saturated_underflow, saturated_overflow,
value)
if overflow then
    if carry then
        return saturated_underflow
    else
        return saturated_overflow
else
    return value
```

Shift

```
SL(value, cnt)
if cnt > 31 then
    return 0
else
    return (value << cnt)
```

Bit reverse

```
BITREVERSE(value)
result ← 0
mask ← 1
shift ← 31
cnt ← 32
while cnt > 0 then do
    t ← data & mask
    if shift >= 0 then
        result ← (t << shift) | result
    else
        result ← (t >> -shift) | result
    cnt ← cnt - 1
    shift ← shift - 2
    mask ← mask << 1
return result
```

__brinc

Bit reversed increment

d = __brinc(a,b)

```
n ← MASKBITS            // Imp dependent # of mask bits
mask ← b64-n:63         // Least sig. n bits of register
temp0 ← a64-n:63
temp1 ← bitreverse(1 + bitreverse(a | (¬mask)))
d ← a0:63-n || (d & mask)
```

brinc provides a way for software to access FFT data in a bit-reversed manner. Parameter a contains the index into a buffer that contains data on which FFT is to be performed. Parameter b contains a mask that allows the index to be updated with bit-reversed addressing. Typically this instruction precedes a load with index instruction; for example,

```
brinc r2, r3, r4
lhax r8, r5, r2
```

Parameter b contains a bit-mask that is based on the number of points in an FFT. To access a buffer containing n byte sized data that is to be accessed with bit-reversed addressing, the mask has $\log_2 n$ 1s in the least significant bit positions and 0s in the remaining most significant bit positions. If, however, the data size is a multiple of a half word or a word, the mask is constructed so that the 1s are shifted left by $\log_2$ (size of the data) and 0s are placed in the least significant bit positions. *Table 8* shows example values of masks for different data sizes and number of data.

Table 8. Data samples and sizes

Number of data samples	Byte	Half word	Word	Double word
8	000...00000111	000...00001110	000...000011100	000...0000111000
16	000...00001111	000...00011110	000...000111100	000...0001111000
32	000...00011111	000...00111110	000...001111100	000...0011111000
64	000...00111111	000...01111110	000...011111100	000...0111111000

Table 9. __brinc (registers altered by).

d	a	b	Maps to
uint32_t	uint32_t	uint32_t	**brinc d,a,b**

Architecture Note: An implementation can restrict the number of bits specified in a mask. The number of bits in a mask may not exceed 32.

Architecture Note: This instruction only modifies the lower 32 bits of the destination register in 32-bit implementations. For 64-bit implementations in 32-bit mode, the contents of the upper 32 bits of the destination register are undefined.

Architecture Note: Execution of brinc does not cause SPE Unavailable exceptions, regardless of the state of MSRSPE.

__ev_abs

Vector Absolute Value

d = __ev_abs(a)

$d_{0:31} \leftarrow \mathbf{ABS}(a_{0:31})$
$d_{32:63} \leftarrow \mathbf{ABS}(a_{32:63})$

The absolute value of each element of a parameter is placed in the corresponding elements of parameter d. An absolute value of 0x8000_0000 (most negative number) returns 0x8000_0000. No overflow is detected.

Figure 4. Vector absolute value (__ev_abs)

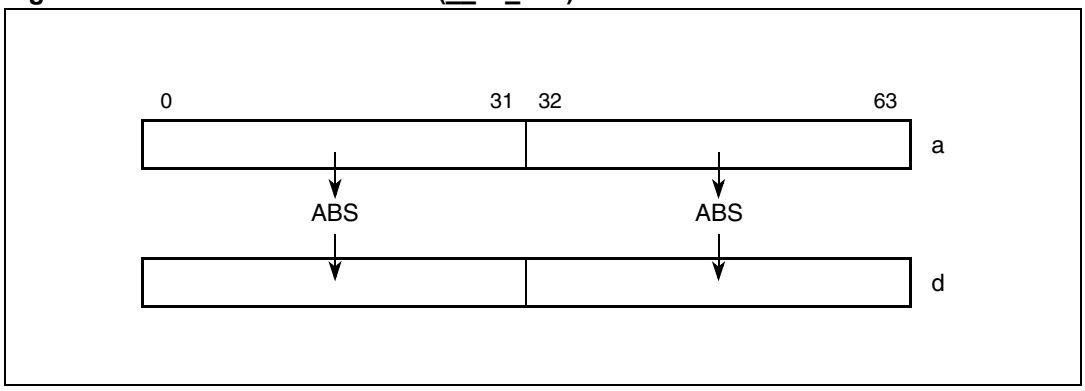


Table 10. __ev_abs (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evabs d,a**

__ev_addiw

Vector Add Immediate Word

d= __ev_addiw (a,b)

```
d0:31 ← a0:31  + EXTZ(b)// Modulo sum
d32:63 ←a32:63  + EXTZ(b)// Modulo sum
```

Parameter b is zero-extended and added to both the high and low elements of parameter a and the results are placed in the parameter d.

Note: The same value is added to both elements of the register.

Figure 5. Vector add immediate word (__ev_addiw)

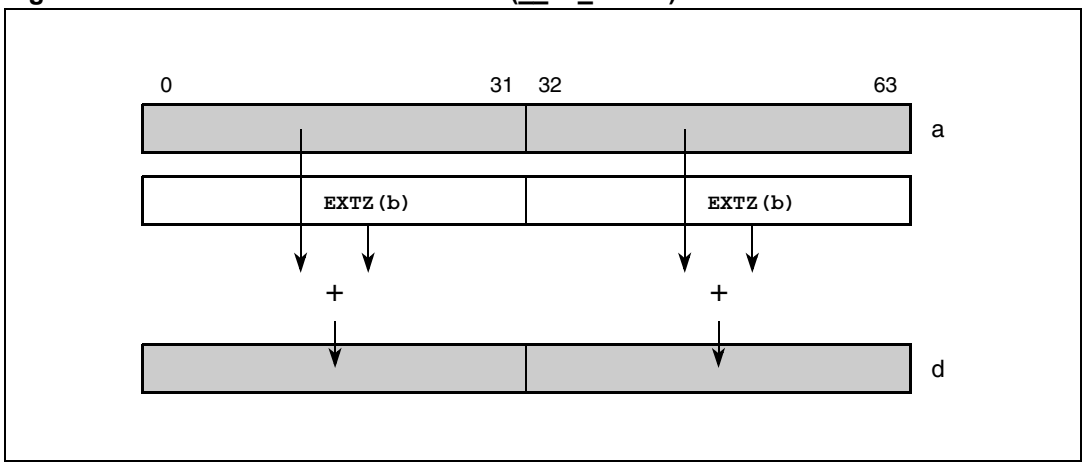


Table 11. __ev_addiw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	5-bit unsigned literal	**evaddiw d,a,b**

__ev_addsmiaaw

Vector Add Signed, Modulo, Integer to Accumulator Word

d = __ev_addsmiaaw (a)

```
// high
d0:31 ← ACC0:31 + a0:31// low
d32:63 ← ACC32:63 + a32:63
// update accumulator
ACC0:63 ← d0:63
```

Each word element in parameter a is added to the corresponding element in the accumulator and the results are placed in parameter d and into the accumulator.

Other registers altered: ACC

Figure 6. Vector add signed, modulo, integer to accumulator word (ev_addsmiaaw)

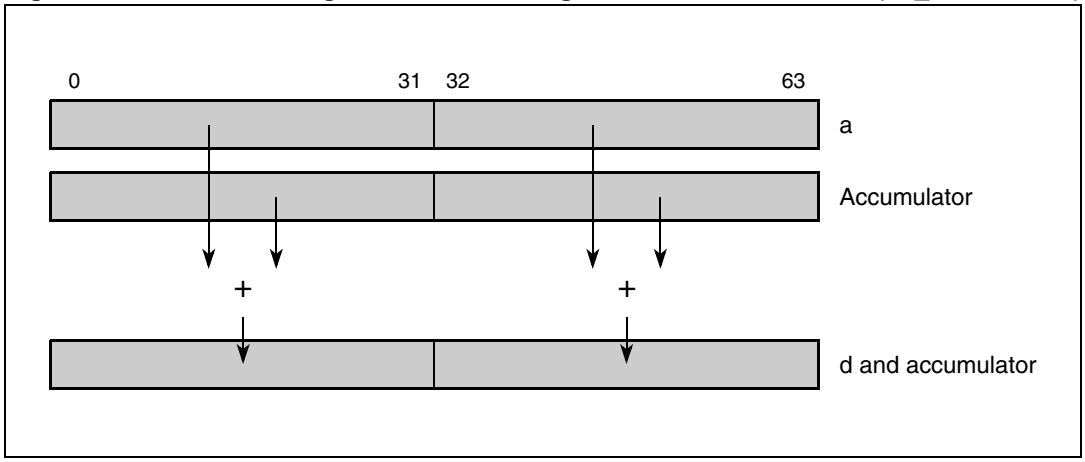


Table 12. __ev_addsmiaaw (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evaddsmiaaw d,a**

__ev_addssiaaw

Vector Add Signed, Saturate, Integer to Accumulator Word

d = __ev_addssiaaw (a)

```
// high
temp0:63 ← EXTS(ACC0:31) + EXTS(a0:31)
ovh ← temp31 ⊕ temp32
d0:31 ← SATURATE(ovh, temp31, 0x80000000, 0x7fffffff, temp32:63)
// low
temp0:63 ← EXTS(ACC32:63) + EXTS(a32:63)
ovl ← temp31 ⊕ temp32
d32:63 ← SATURATE(ovl, temp31, 0x80000000, 0x7fffffff, temp32:63)

ACC0:63 ← d0:63

SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

Each signed integer word element in parameter a is sign-extended and added to the corresponding sign-extended element in the accumulator, saturating if overflow or underflow occurs, and the results are placed in parameter d and the accumulator. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

Figure 7. Vector add signed, saturate, integer to accumulator word (ev_addssiaaw)

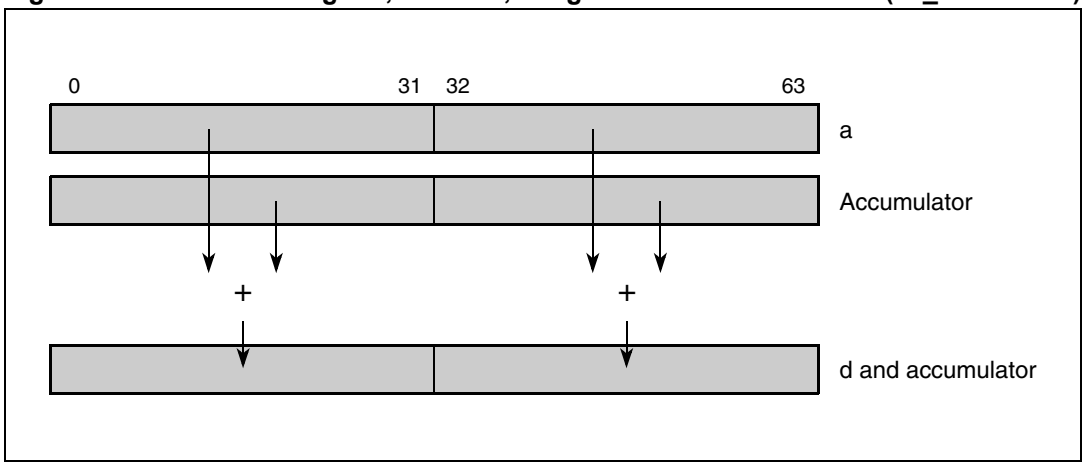


Table 13. __ev_addssiaaw (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evaddssiaaw d,a**

__ev_addumiaaw

Vector Add Unsigned, Modulo, Integer to Accumulator Word

d = __ev_addumiaaw (a)

$d_{0:31} \leftarrow ACC_{0:31} + a_{0:31}$
$d_{32:63} \leftarrow ACC_{32:63} + a_{32:63}$

$ACC_{0:63} \leftarrow d_{0:63}$

Each unsigned integer word element in the parameter a is added to the corresponding element in the accumulator and the results are placed in the parameter d and the accumulator.

Other registers altered: ACC

Figure 8. Vector add unsigned,modulo,integer to accumulator word (ev_addumiaaw)

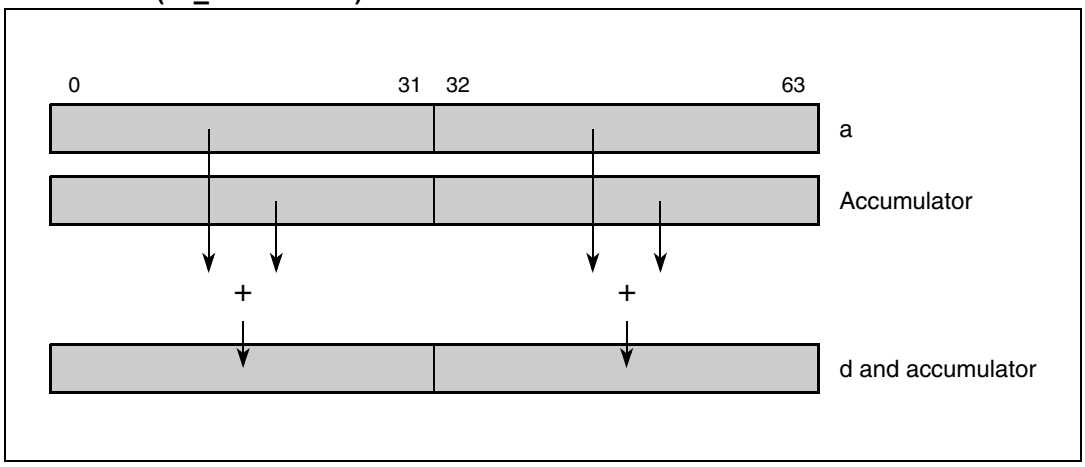


Table 14. __ev_addumiaaw (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evaddumiaaw d,a**

__ev_addusiaaw

Vector Add Unsigned, Saturate, Integer to Accumulator Word

d = __ev_addusiaaw (a)

```
// high
temp0:63 ← EXTZ(ACC0:31) + EXTZ(a0:31)
ovh ← temp31
d0:31 ← SATURATE(ovh, temp31, 0xffffffff, 0xffffffff, temp32:63)
// low
temp0:63 ← EXTZ(ACC32:63) + EXTZ(a32:63)
ovl ← temp31
d32:63 ← SATURATE(ovl, temp31, 0xffffffff, 0xffffffff, temp32:63)

ACC0:63 ← d0:63

SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

Each unsigned integer word element in parameter a is zero-extended and added to the corresponding zero-extended element in the accumulator, saturating if overflow occurs, and the results are placed in parameter d and the accumulator. Any overflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

Figure 9. Vector add unsigned, saturate, integer to accumulator word (ev_addusiaaw)

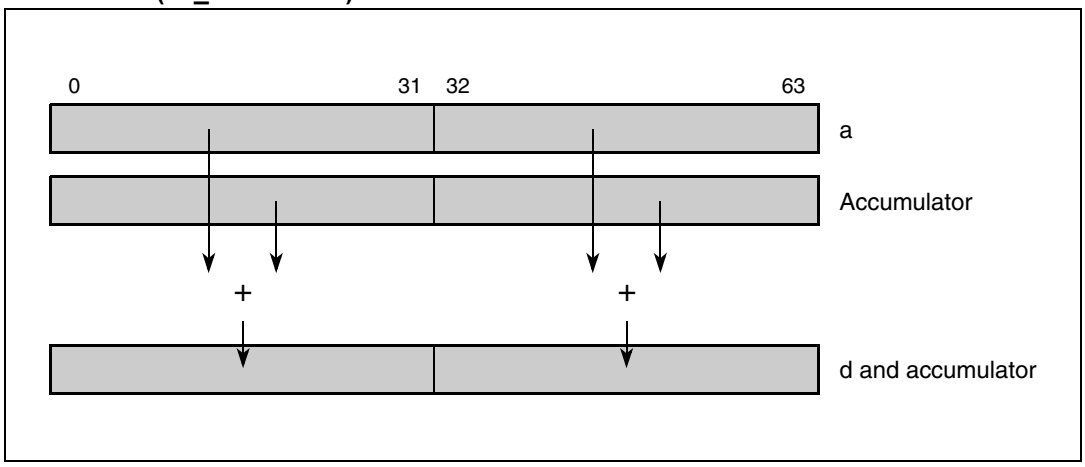


Table 15. __ev_addusiaaw (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evaddusiaaw d,a**

__ev_addw

Vector Add Word

d = __ev_addw (a,b)

$d_{0:31} \leftarrow a_{0:31}$ + $b_{0:31}$// Modulo sum
$d_{32:63} \leftarrow a_{32:63}$ + $b_{32:63}$ // Modulo sum

The corresponding elements of parameters a and b are added, and the results are placed in parameter d. The sum is a modulo sum.

Figure 10. Vector add word (__ev_addw)

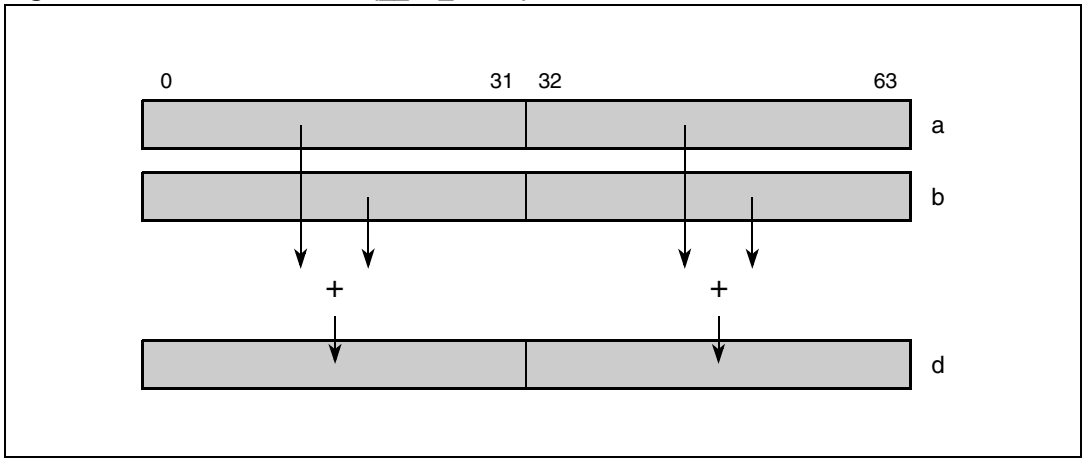


Table 16. __ev_addw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evaddw d,a,b**

__ev_all_eq

Vector All Equal

d = __ev_all_eq(a,b)

```
if ( a0:31 = b0:31) & (a32:63 = b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b and the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b.

Figure 11. Vector all equal (__ev_all_eq)

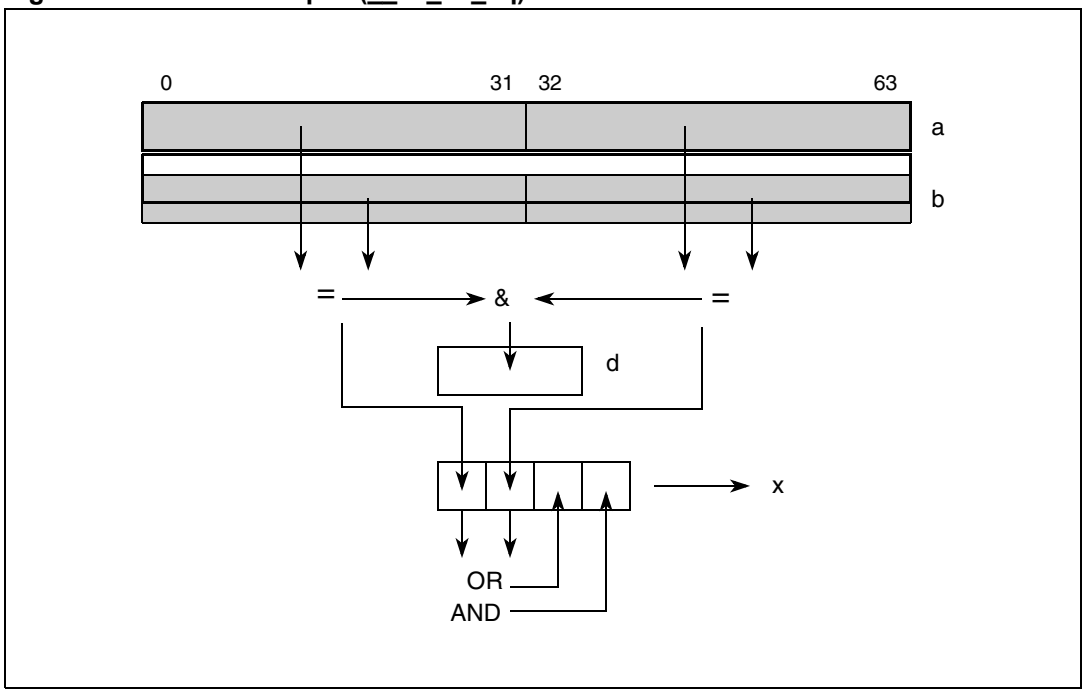


Table 17. __ev_all_eq (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpeq x,a,b**

__ev_all_fs_eq

Vector All Floating-Point Equal

d = __ev_all_fs_eq(a,b)

```
if ( (a0:31 = b0:31) & (a32:63 = b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b and the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b.

Figure 12. Vector all floating-point equal (__ev_all_fs_eq)

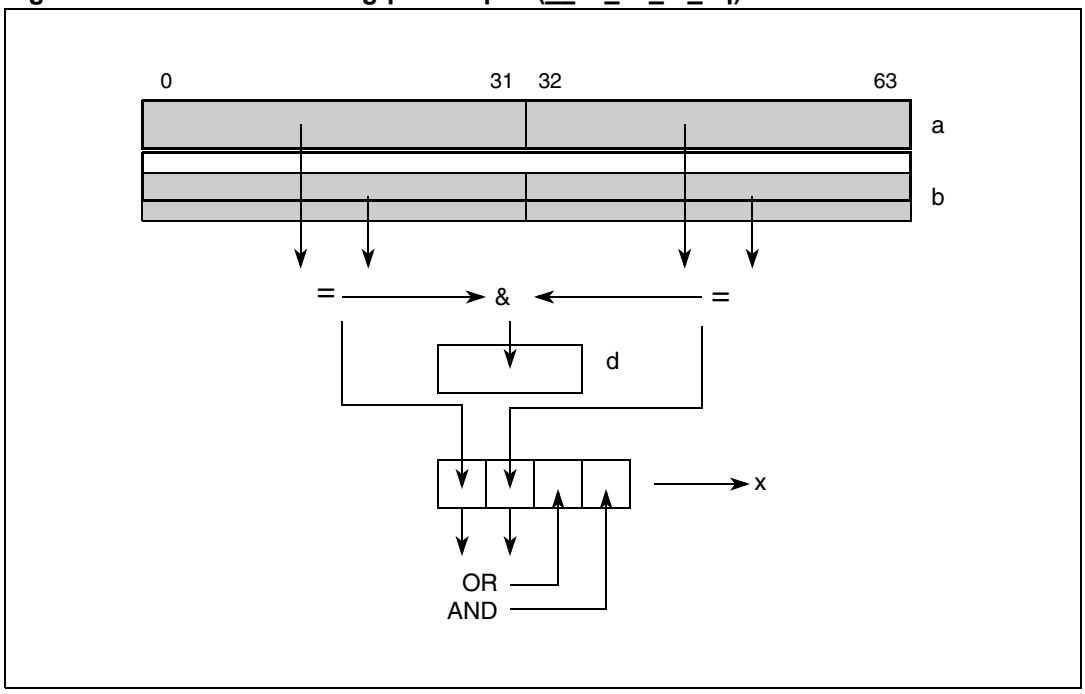


Table 18. __ev_all_fs_eq (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmplt x,a,b**

__ev_all_fs_gt

Vector All Floating-Point Greater Than

d = __ev_all_fs_gt(a,b)

```
if ( a0:31 > b0:31) & (a32:63 > b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b and the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

Figure 13. Vector all floating-point greater than (__ev_all_fs_gt)

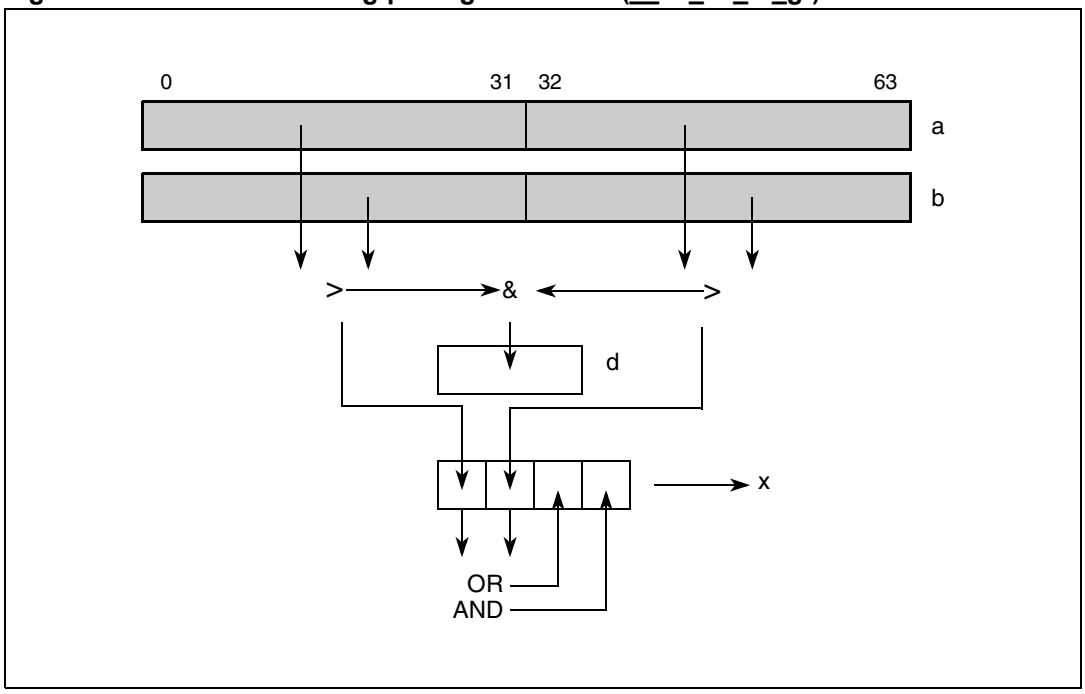


Table 19. __ev_all_fs_gt (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmpgt x,a,b**

__ev_all_fs_lt

Vector All Floating-Point Less Than

d = __ev_all_fs_lt(a,b)

```
if ( (a0:31 < b0:31) & (a32:63 < b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are less than the upper 32 bits of parameter b, and the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

Figure 14. Vector all floating-point less than (__ev_all_fs_lt)

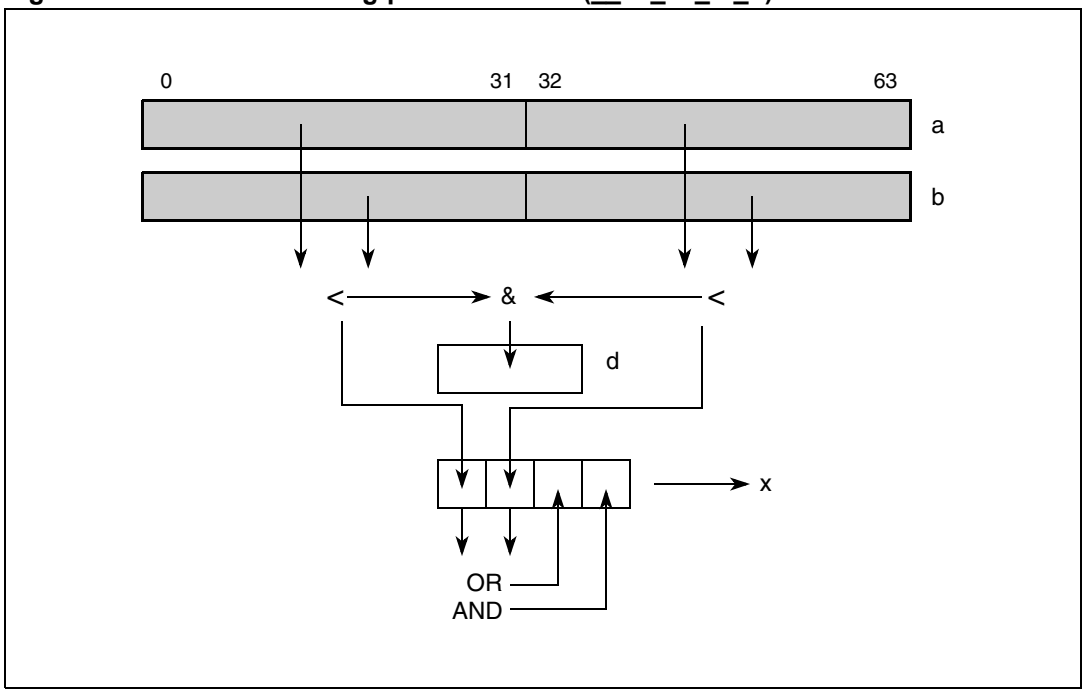


Table 20. __ev_all_fs_lt (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmplt x,a,b**

__ev_all_fs_tst_eq

Vector All Floating-Point Test Equal

d = __ev_all_fs_tst_eq(a,b)

```
if ( (a0:31 = unsigned b0:31) & (a32:63 = unsigned b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b, and the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b. This intrinsic differs from __ev_all_fs_eq because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_all_fs_eq instead.

Figure 15. Vector all floating-point test equal (__ev_all_fs_tst_eq)

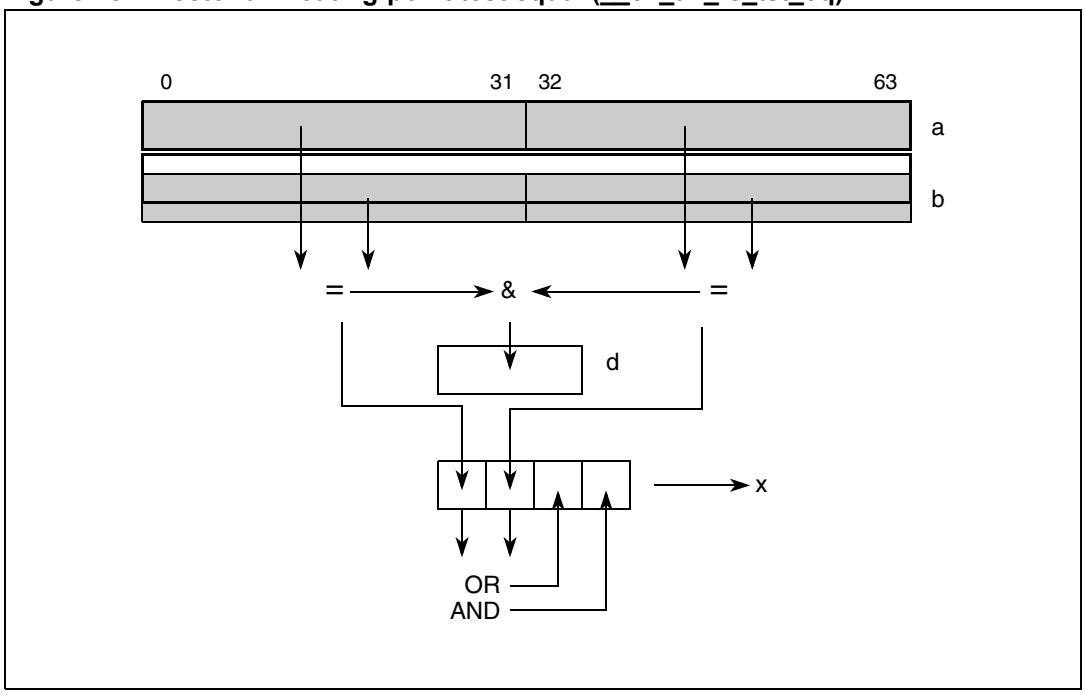


Table 21. __ev_all_fs_tst_eq (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststeq x,a,b**

__ev_all_fs_tst_gt

Vector All Floating-Point Test Greater Than

d = __ev_all_fs_tst_gt(a,b)

```
if ( (a0:31 > b0:31) & (a32:63 > b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b and the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b. This intrinsic differs from __ev_all_fs_gt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_all_fs_gt instead.

Figure 16. Vector all floating-point test greater than (__ev_all_fs_tst_gt)

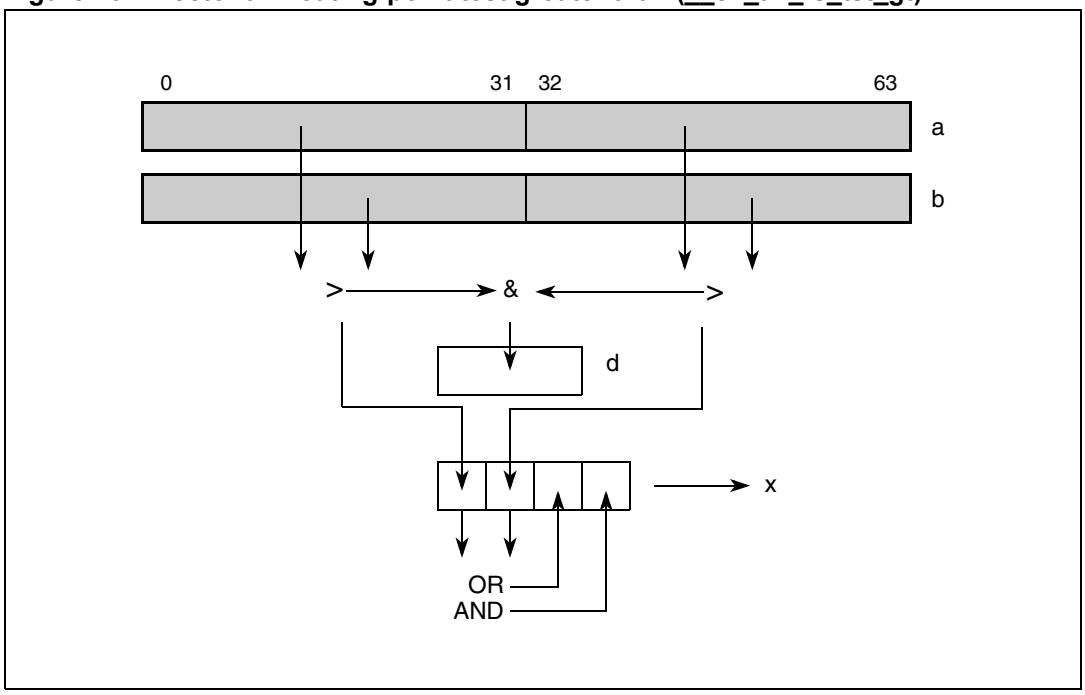


Table 22. __ev_all_fs_tst_gt (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststgt x,a,b**

__ev_all_fs_tst_lt

Vector All Floating-Point Test Less Than

d = __ev_all_fs_tst_lt(a,b)

```
if ( (a0:31 < b0:31) & (a32:63 < b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are less than the upper 32 bits of parameter b and the lower 32 bits of parameter a are less than the lower 32 bits of parameter b. This intrinsic differs from __ev_all_fs_lt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_all_fs_lt instead.

Figure 17. Vector all floating-point test less than (__ev_all_fs_tst_lt)

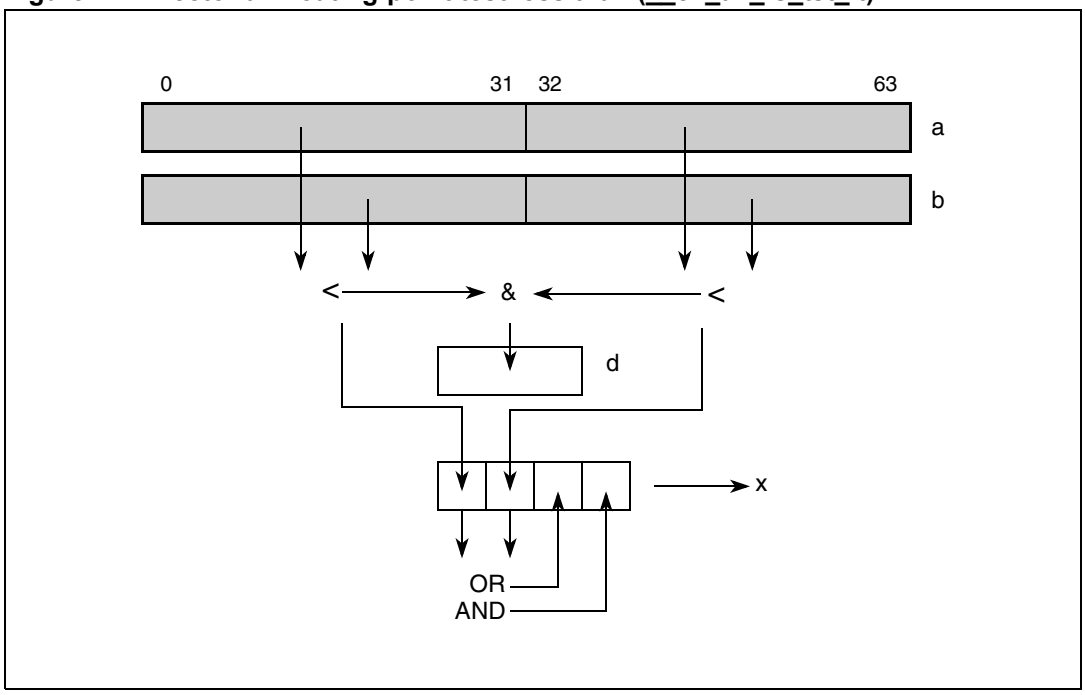


Table 23. __ev_all_fs_tst_lt (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststlt x,a,b**

__ev_all_gts

Vector All Greater Than Signed

d = __ev_all_gts(a,b)

```
if ( (a0:31 >signed b0:31) & (a32:63 >signed b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b and the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

Figure 18. Vector all greater than signed (__ev_all_gts)

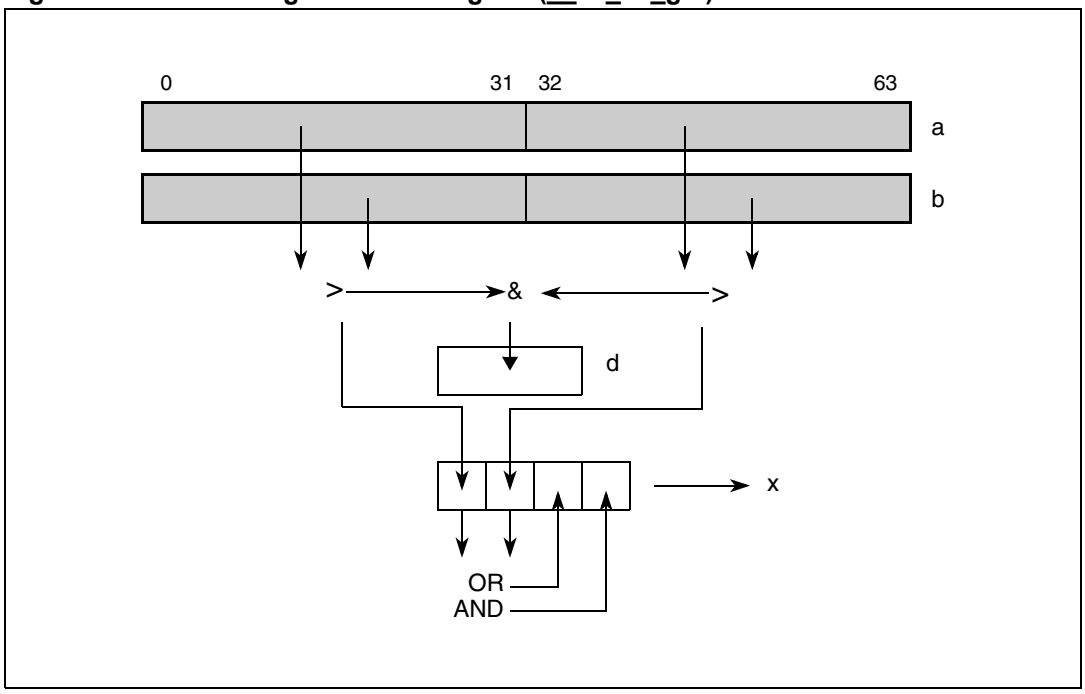


Table 24. __ev_all_gts (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpgts x,a,b**

__ev_all_gtu

Vector All Elements Greater Than Unsigned

d = __ev_all_gtu(a,b)

```
if ( (a0:31 > unsigned b0:31) & (a32:63 > unsigned b32:63)) then d ← true
else a ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b and the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

Figure 19. Vector all greater than unsigned (__ev_all_gtu)

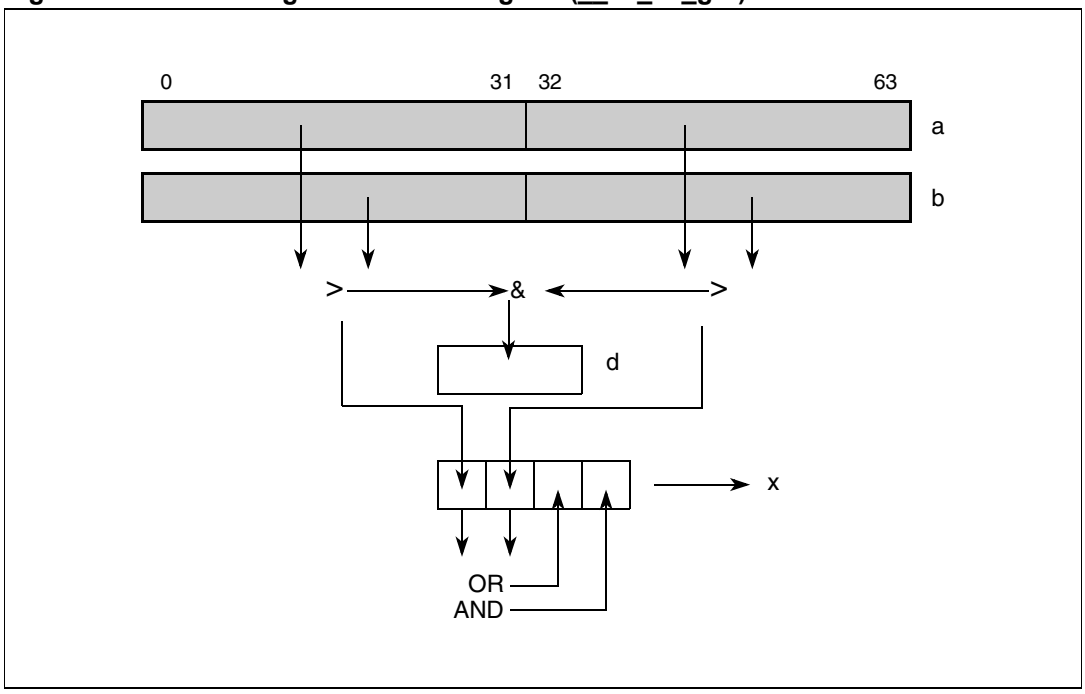


Table 25. __ev_all_gtu (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpgtu x,a,b**

__ev_all_lts

Vector All Elements Less Than Signed

d = __ev_all_lts(a,b)

```
if ( (a0:31 <signed b0:31) & (a32:63 <signed b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are less than the upper 32 bits of parameter b and the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

Figure 20. Vector all less than signed (__ev_all_lts)

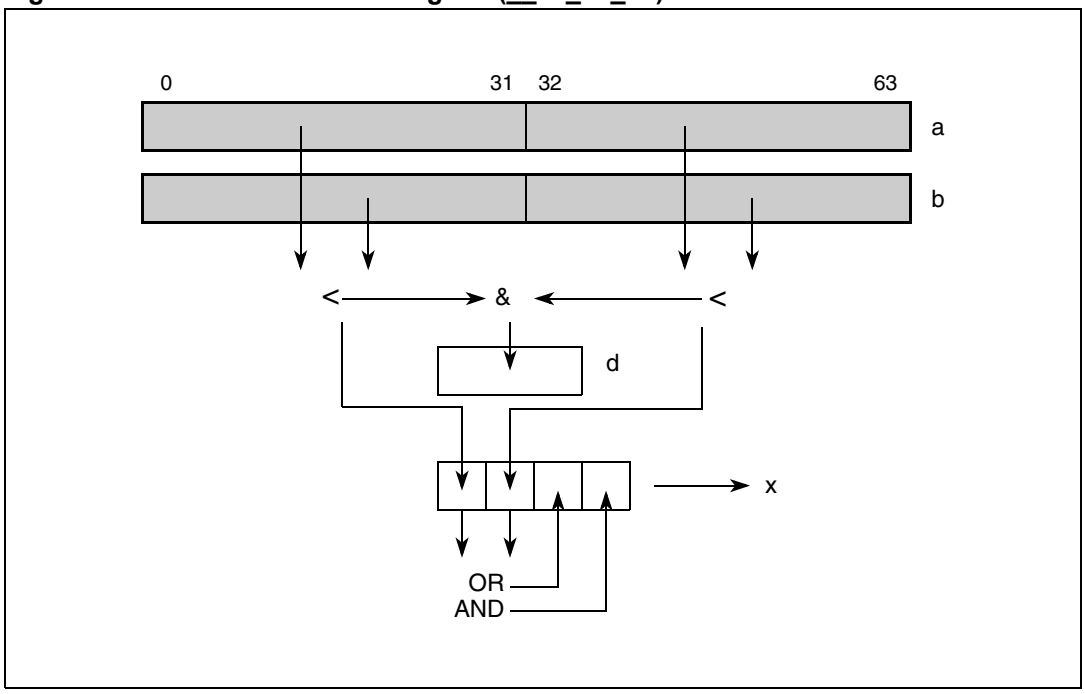


Table 26. __ev_all_lts (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmplts x,a,b**

__ev_all_ltu

Vector All Elements Less Than Unsigned

d = __ev_all_ltu(a,b)

```
if ( (a0:31 <unsigned b0:31) & (a32:63 <unsigned b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are less than the upper 32 bits of parameter b and the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

Figure 21. Vector all less than unsigned (__ev_all_ltu)

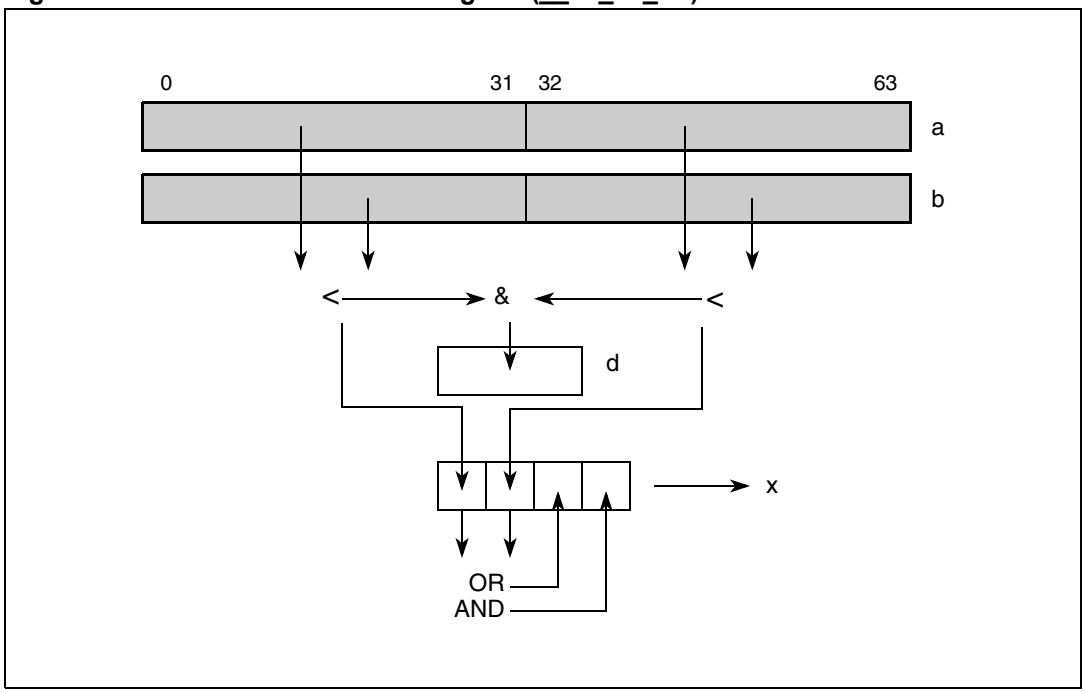


Table 27. __ev_all_ltu (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpltu x,a,b**

__ev_and

Vector AND

d = __ev_and (a,b)

```
d0:31 ← a0:31 & b0:31 // Bitwise AND
d32:63 ← a32:63 & b32:63// Bitwise AND
```

The corresponding elements of parameters a and b are ANDed bitwise, and the results are placed in the corresponding element of parameter d.

Figure 22. Vector AND (__ev_and)

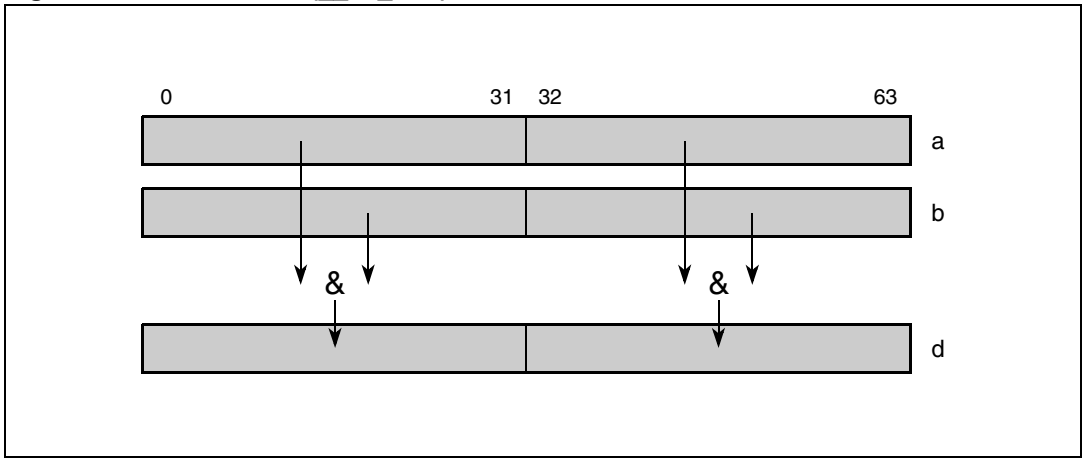


Table 28. __ev_and (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evand d,a,b**

__ev_andc

Vector AND with Complement

d = __ev_andc(a,b)

$d_{0:31} \leftarrow a_{0:31}$ `& (¬`$b_{0:31}$`) // Bitwise ANDC`
$d_{32:63} \leftarrow a_{32:63}$ `& (¬`$b_{32:63}$`) // Bitwise ANDC`

The word elements of parameter a and are ANDed bitwise with the complement of the corresponding elements of parameter b. The results are placed in the corresponding element of parameter d.

Figure 23. Vector AND with complement (__ev_andc)

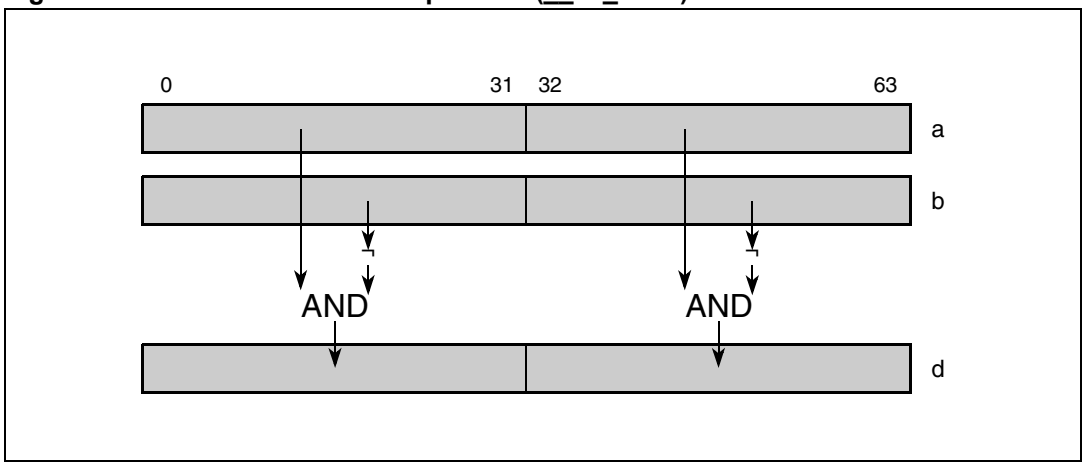


Table 29. __ev_andc (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evandc d,a,b**

__ev_any_eq

Vector Any Equal

d = __ev_any_eq(a,b)

```
if ( (a0:31 = b0:31) | (a32:63 = b32:63)) then d← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b or the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b.

Figure 24. Vector any equal (__ev_any_eq)

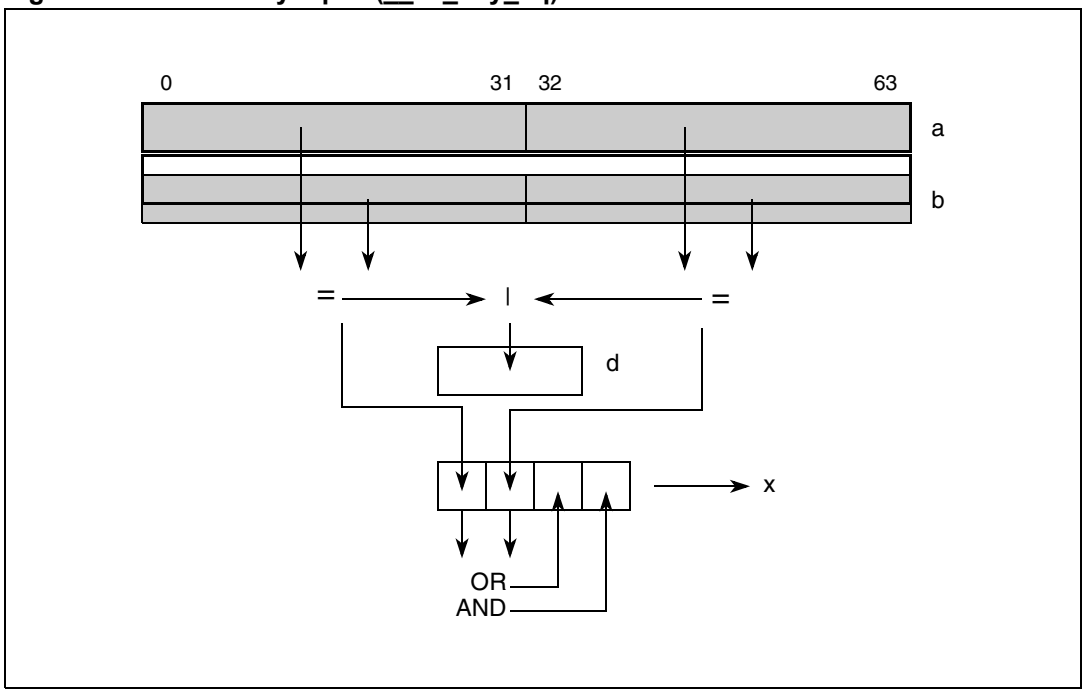


Table 30. __ev_any_eq (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpeq x,a,b**

__ev_any_fs_eq

Vector Any Floating-Point Equal

d = __ev_any_fs_eq(a,b)

```
if ( (a0:31 = b0:31) | (a32:63 = b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b or the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b.

Figure 25. Vector any floating-point equal (__ev_any_fs_eq)

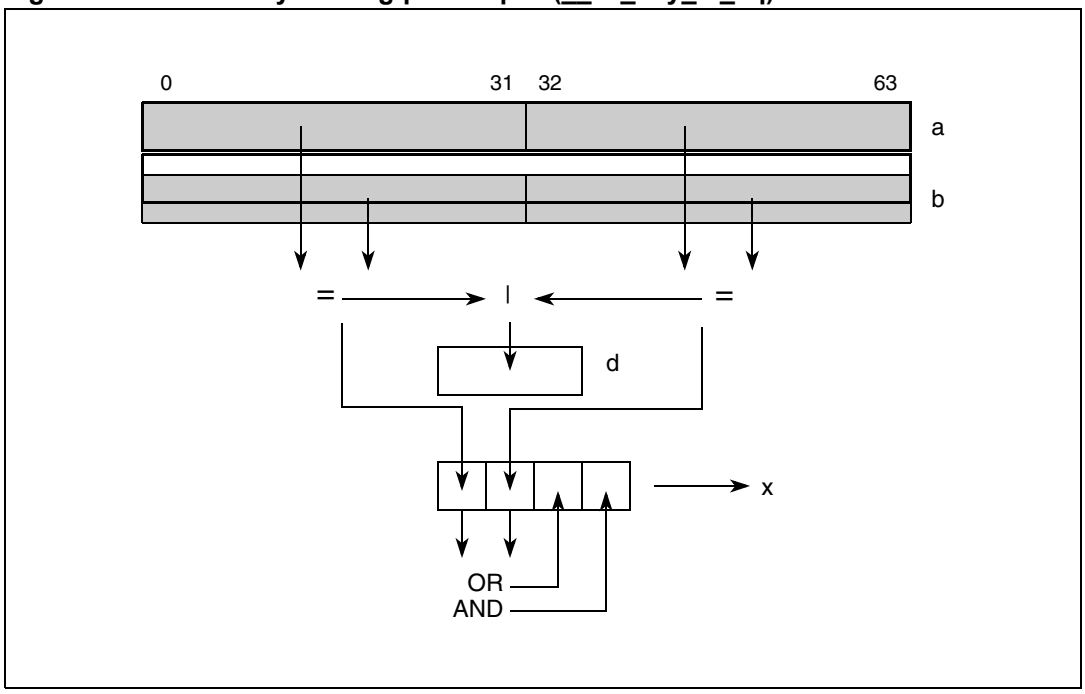


Table 31. __ev_any_fs_eq (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmpeq x,a,b**

__ev_any_fs_gt

Vector Any Floating-Point Greater Than

d = __ev_any_fs_gt(a,b)

```
if ( (a0:31 > b0:31) | (a32:63 > b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b or the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

Figure 26. Vector any floating-point greater than (__ev_any_fs_gt)

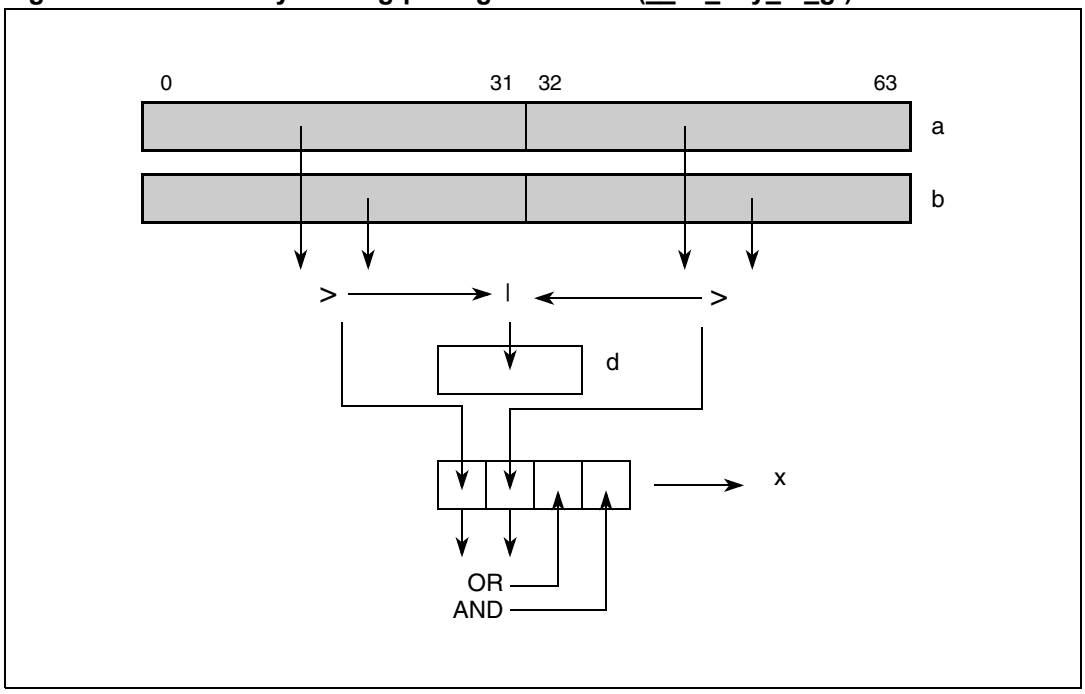


Table 32. __ev_any_fs_gt (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmpgt x,a,b**

__ev_any_fs_lt

Vector Any Floating-Point Less Than

d = __ev_any_fs_lt(a,b)

```
if ( (a0:31 < b0:31) | (a32:63 < b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are less than the upper 32 bits of parameter b or the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

Figure 27. Vector any floating-point less than (__ev_any_fs_lt)

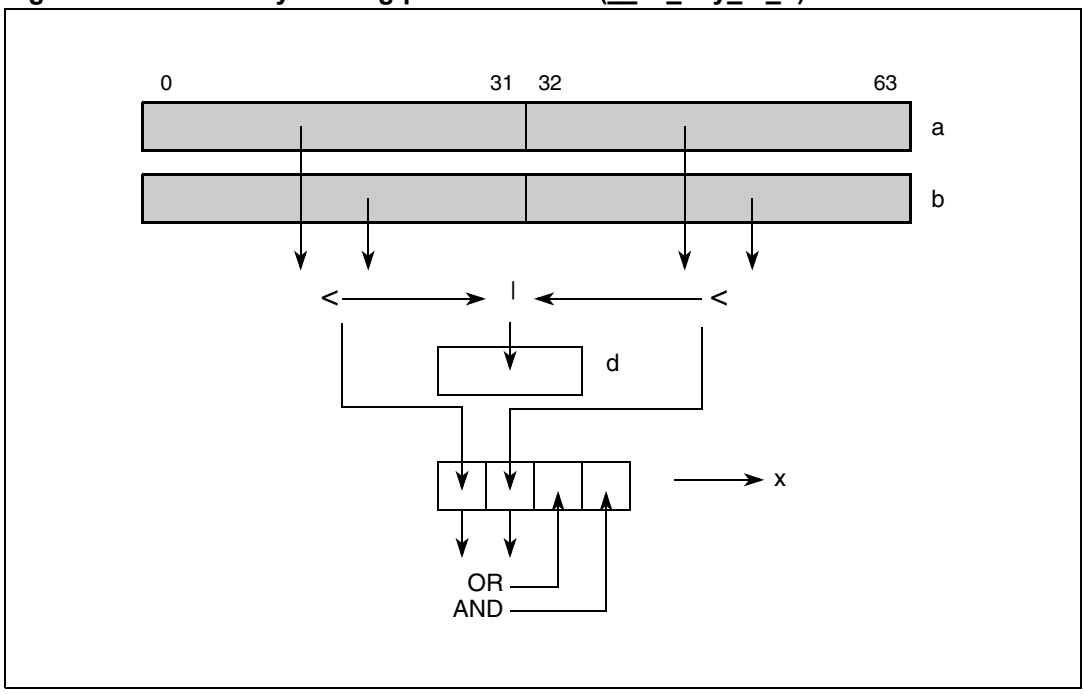


Table 33. __ev_any_fs_lt (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmplt x,a,b**

__ev_any_fs_tst_eq

Vector Any Floating-Point Test Equal

d = __ev_any_fs_tst_eq(a,b)

```
if ( (a0:31 = b0:31) | (a32:63 = b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b or the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b. This intrinsic differs from __ev_any_fs_eq because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_any_fs_eq instead.

Figure 28. Vector any floating-point test equal (__ev_any_fs_tst_eq)

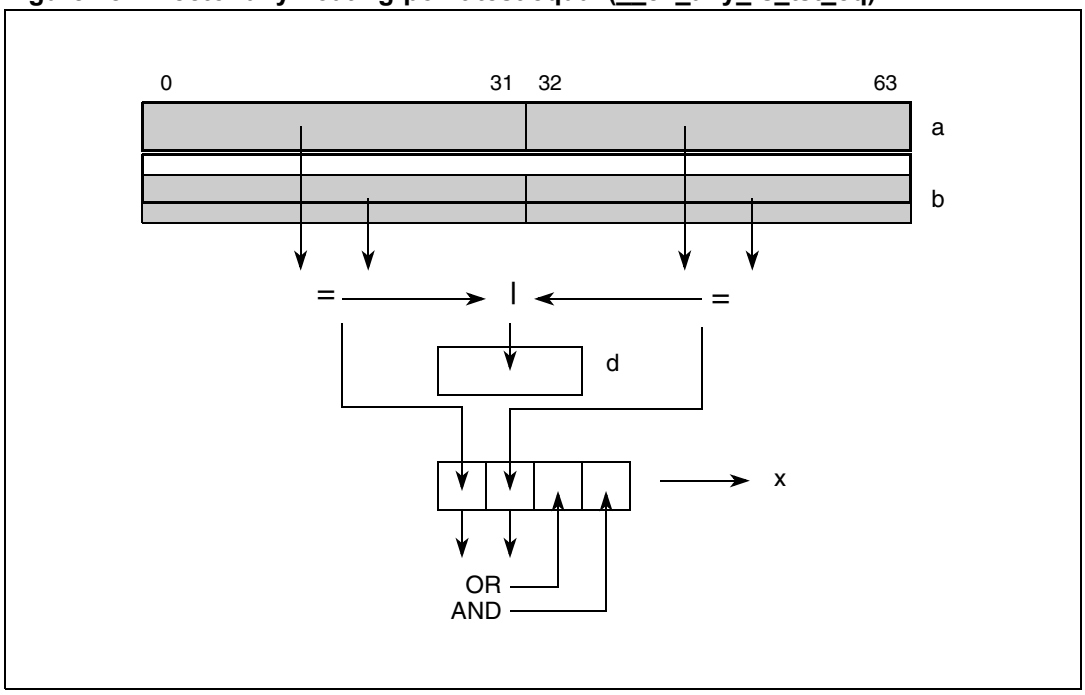


Table 34. __ev_any_fs_tst_eq (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststeq x,a,b**

__ev_any_fs_tst_gt

Vector Any Floating-Point Test Greater Than

d = __ev_any_fs_tst_gt(a,b)

```
if ( (a0:31 > b0:31) | (a32:63 > b2:63)) then d← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b or the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b. This intrinsic differs from __ev_any_fs_gt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_any_fs_gt instead.

Figure 29. Vector any floating-point test greater than (__ev_any_fs_tst_gt)

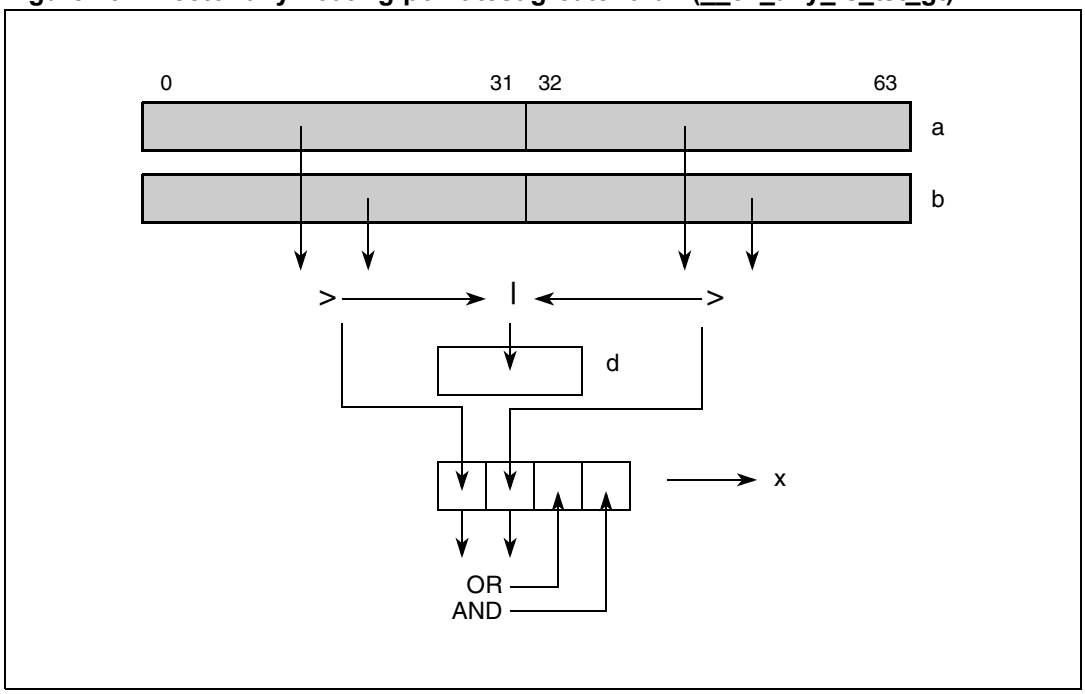


Table 35. __ev_any_fs_tst_gt (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststgt x,a,b**

__ev_any_fs_tst_lt

Vector Any Floating-Point Test Less Than

d = __ev_any_fs_tst_lt(a,b)

```
if ( (a0:31 < b0:31) || (a32:63 < b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are less than the upper 32 bits of parameter b or the lower 32 bits of parameter a are less than the lower 32 bits of parameter b. This intrinsic differs from __ev_any_fs_lt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_any_fs_lt instead.

Figure 30. Vector any floating-point test less than (__ev_any_fs_tst_lt)

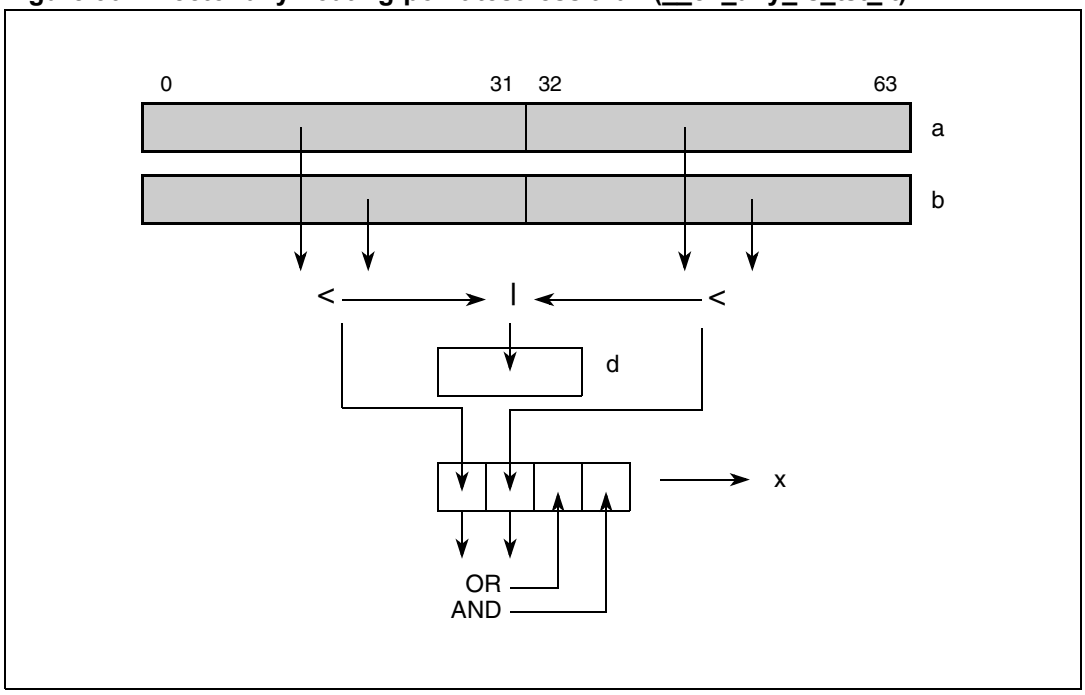


Table 36. __ev_any_fs_tst_lt (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststlt x,a,b**

__ev_any_gts

Vector AND with Complement

d = __ev_any_gts(a,b)

```
if ((a0:31 >signed b0:31)|(a32:63 >signed b32:63)) then d ← true
else d ← false
```

This intrinsic returns true if either the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b or the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

Figure 31. Vector any greater than signed (__ev_any_gts)

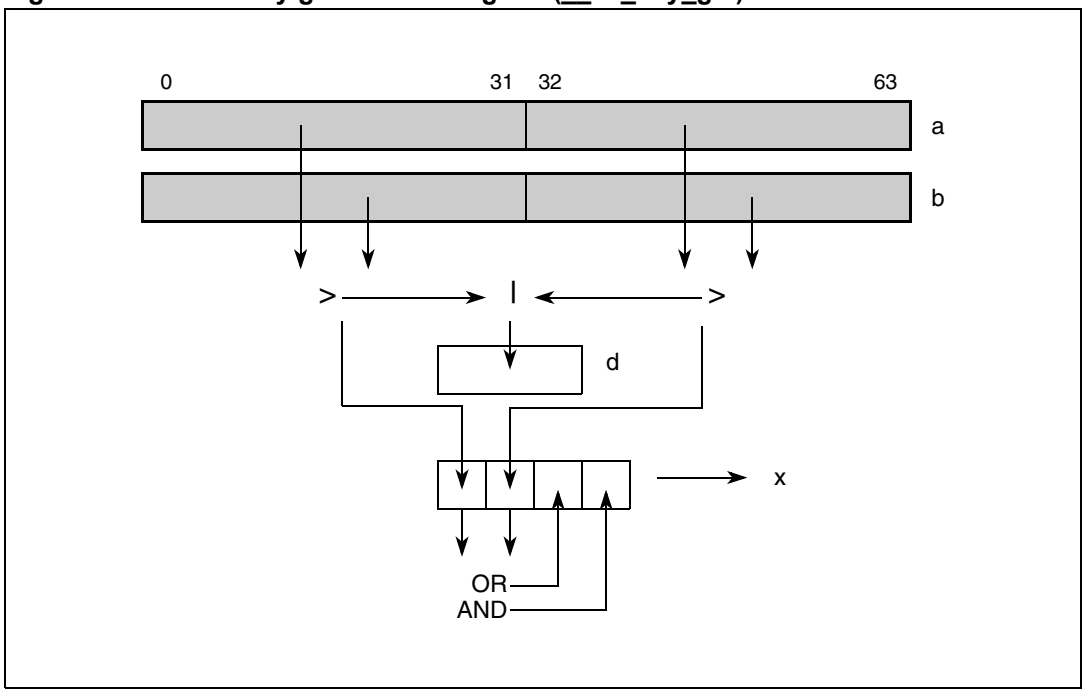


Table 37. __ev_any_gts (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpgts x,a,b**

__ev_any_gtu

Vector Any Element Greater Than Unsigned

d = __ev_any_gtu(a,b)

```
if ( (a0:31 >unsigned b0:31) | (a32:63 >unsigned b32:63)) then d ← true
else d←false
```

This intrinsic returns true if either the upper 32 bits of parameters a are greater than the upper 32 bits of parameter b or the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

Figure 32. Vector any greater than unsigned (__ev_any_gtu)

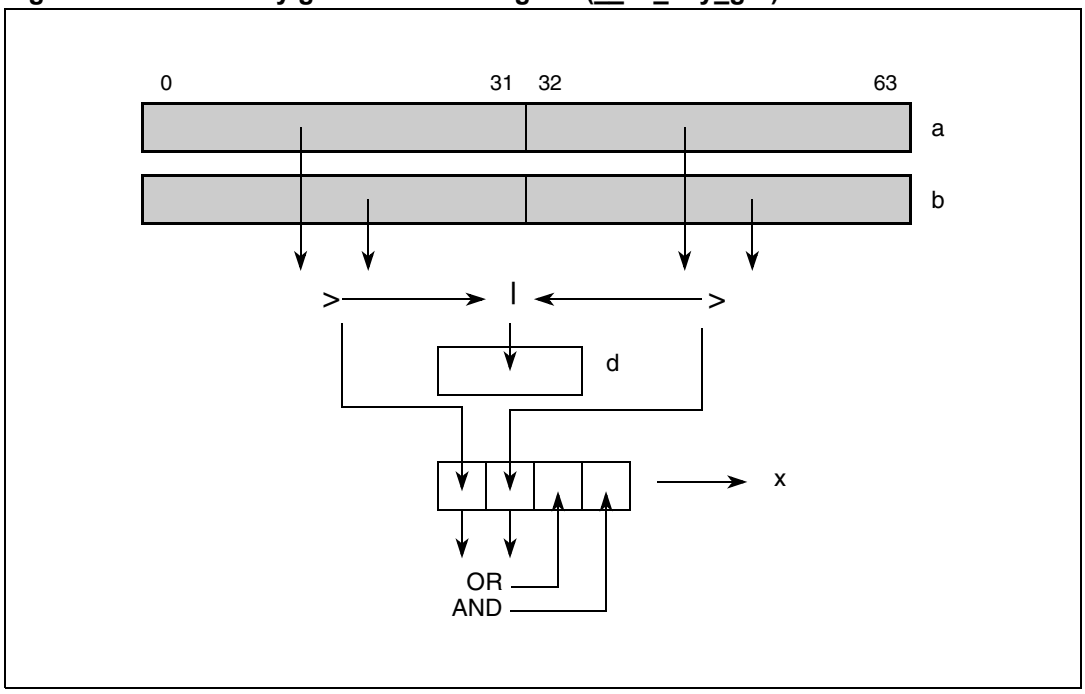


Table 38. __ev_any_gtu (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpgtu x,a,b**

__ev_any_lts

Vector Any Element Less Than Signed

d = __ev_any_lts(a,b)

```
if ( (a0:31 <signed b0:31) | (a32:63 <signed b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are less than the upper 32 bits of parameter b or the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

Figure 33. Vector any less than signed(__ev_any_lts)

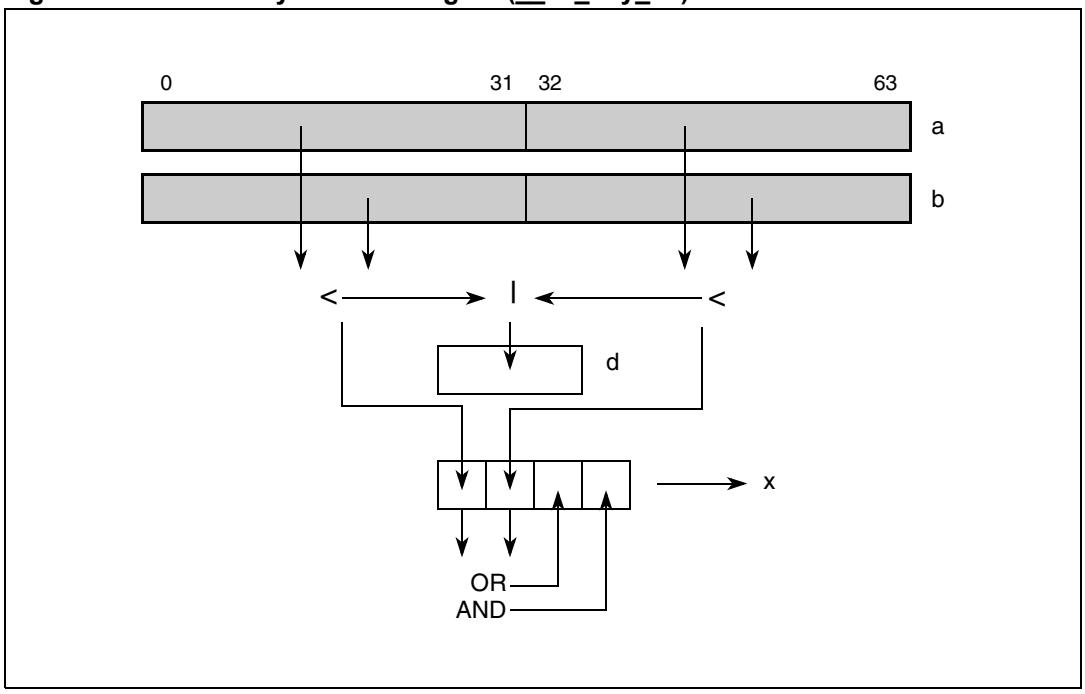


Table 39. __ev_any_lts (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmplts x,a,b**

__ev_any_ltu

Vector Any Element Less Than Unsigned

d = __ev_any_ltu(a,b)

```
if ( (a0:31 <unsigned b0:31) | (a32:63 <unsigned b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are less than the upper 32 bits of parameter b or the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

Figure 34. Vector any less than unsigned (__ev_any_ltu)

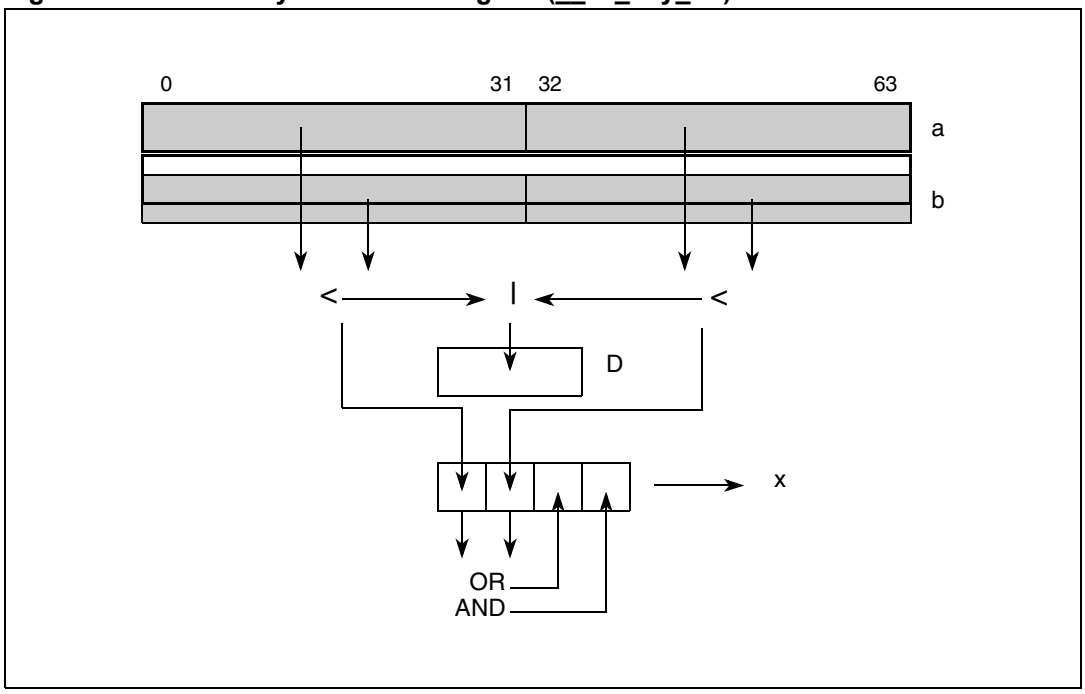


Table 40. __ev_any_ltu (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpltu x,a,b**

__ev_cntlsw

Vector Count Leading Signed Bits Word

d = __ev_cntlsw(a)

The leading signed bits in each element of parameter a are counted, and the count is placed into each element of parameter d.

evcntlzw is used for unsigned parameters; **evcntlsw** is used for signed parameters.

Figure 35. Vector count leading signed bits word (__ev_cntlsw)

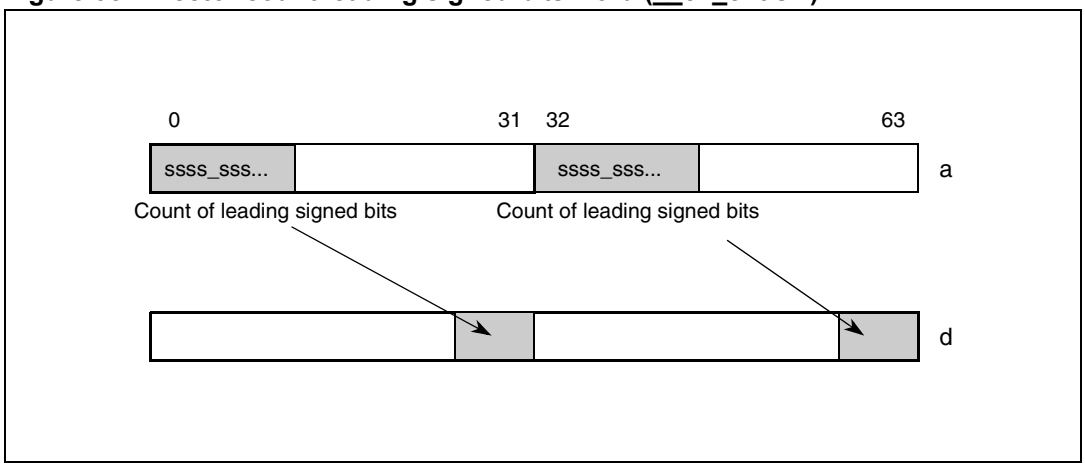


Table 41. __ev_cntlsw (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evcntlsw d,a**

__ev_cntlzw

Vector Count Leading Zeros Word

d = __ev_cntlzw(a)

The leading zero bits in each element of parameter a are counted, and the respective count is placed into each element of parameter d.

Figure 36. Vector Count Leading Signed Bits Word (__ev_cntlzw)

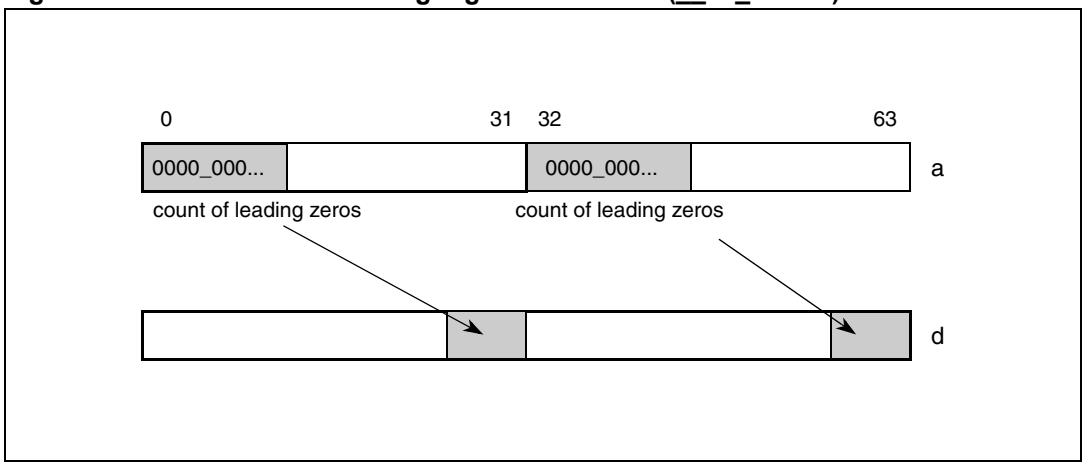


Table 42. __ev_cntlzw (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evcntlzw d,a**

__ev_divws

Vector Divide Word Signed

d = __ev_divws (a,b)

```
dividendh ← a0:31
dividendl ← a32:63
divisorh ← b0:31
divisorl ← b32:63
d0:31 ← dividendh ÷ divisorh
d32:63 ← dividendl ÷ divisorl
ovh ← 0
ovl ← 0
if ((dividendh < 0) & (divisorh = 0)) then
  d0:31 ← 0x80000000
  ovh ← 1
else if ((dividendh >= 0) & (divisorh = 0)) then
  d0:31 ← 0x7FFFFFFF
  ovh ← 1
else if ((dividendh = 0x80000000) & (divisorh = 0xFFFF_FFFF))
then
  d0:31 ← 0x7FFFFFFF
  ovh ← 1
if ((dividendl < 0) & (divisorl = 0)) then
  d32:63 ← 0x80000000
  ovl ← 1
else if ((dividendl >= 0) & (divisorl = 0)) then
  d32:63 ← 0x7FFFFFFF
  ovl ← 1
else if ((dividendl = 0x80000000) & (divisorl = 0xFFFF_FFFF))
then
  d32:63 ← 0x7FFFFFFF
  ovl ← 1
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The two dividends are the two elements of the contents of parameter a. The two divisors are the two elements of the contents of parameter b. The resulting two 32-bit quotients on each element are placed into parameter d. The remainders are not supplied. Parameters and quotients are interpreted as signed integers. If overflow, underflow, or divide by zero occurs, the overflow and summary overflow SPEFSCR bits are set. Note that any overflow indication is always set as a side effect of this instruction. No form is defined that disables the setting of the overflow bits. In case of overflow, a saturated value is delivered into the destination register.

Figure 37. Vector divide word signed (__ev_divws)

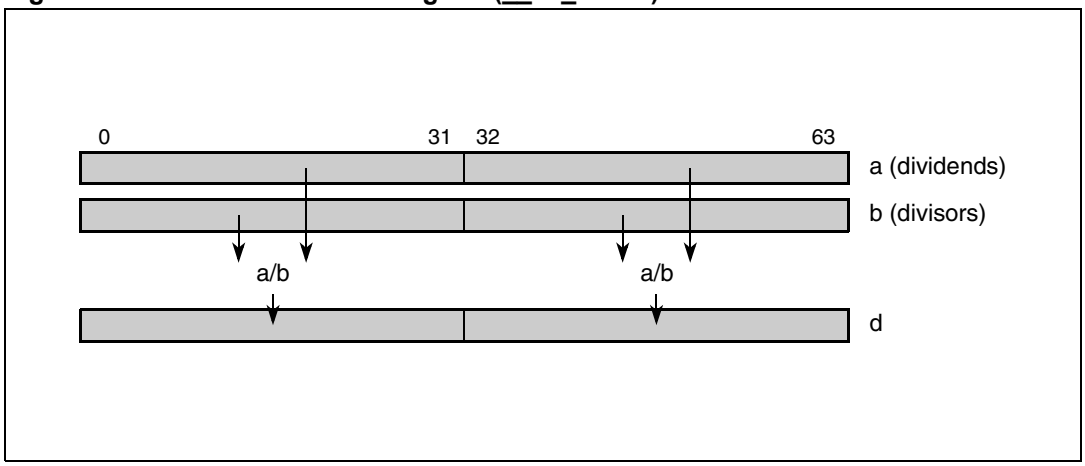


Table 43. __ev_divws (registered altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evdivws d,a,b**

__ev_divwu

Vector Divide Word Unsigned

d = __ev_divwu (a,b)

```
dividendh ← a0:31
dividendl ← a32:63
divisorh ← b0:31
divisorl ← b32:63
d0:31 ← dividendh ÷ divisorh
d32:63 ← dividendl ÷ divisorl
ovh ← 0
ovl ← 0
if (divisorh = 0) then
   d0:31 = 0xFFFFFFFF
   ovh ← 1
if (divisorl = 0) then
    d32:63 ← 0xFFFFFFFF
    ovl ← 1
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The two dividends are the two elements of the contents of parameter a. The two divisors are the two elements of the contents of parameter b. Two 32-bit quotients are formed as a result of the division on each of the high and low elements and the quotients are placed into parameter d. Remainders are not supplied. Parameters and quotients are interpreted as unsigned integers. If a divide by zero occurs, the overflow and summary overflow SPEFSCR bits are set. Note that any overflow indication is always set as a side effect of this instruction. No form is defined that disables the setting of the overflow bits. In case of overflow, a saturated value is delivered into the destination register.

Figure 38. Vector divide word unsigned (__ev_divwu)

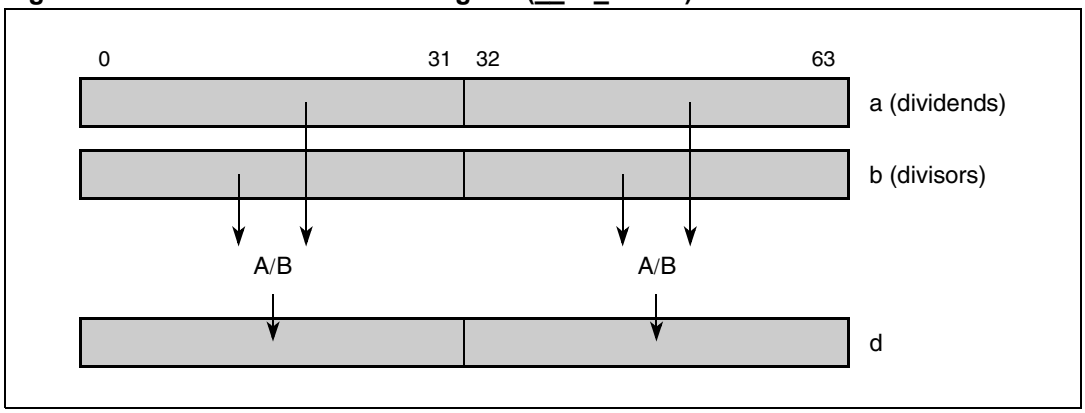


Table 44. __ev_divwu (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evdivwu d,a,b**

__ev_eqv

Vector Equivalent

d = __ev_eqv (a,b)

$d_{0:31} \leftarrow a_{0:31} \equiv b_{0:31}$ // Bitwise XNOR
$d_{32:63} \leftarrow a_{32:63} \equiv b_{32:63}$ // Bitwise XNOR

The corresponding elements of parameters a and b are XNORed bitwise, and the results are placed in the parameter d.

Figure 39. Vector equivalent (__ev_eqv)

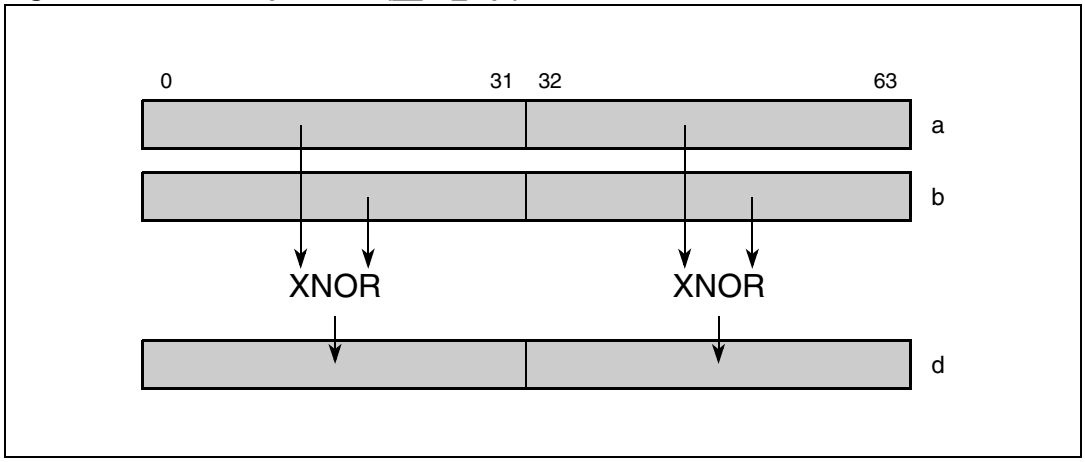


Table 45. __ev_eqv (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**eveqv d,a,b**

__ev_extsb

Vector Extend Sign Byte

d = __ev_extsb (a)

$d_{0:31} \leftarrow \mathbf{EXTS}(a_{24:31})$
$d_{32:63} \leftarrow \mathbf{EXTS}(a_{56:63})$

The signs of the byte in each of the elements in parameter a are extended, and the results are placed in the parameter d.

Figure 40. Vector extend sign byte (__ev_extsb)

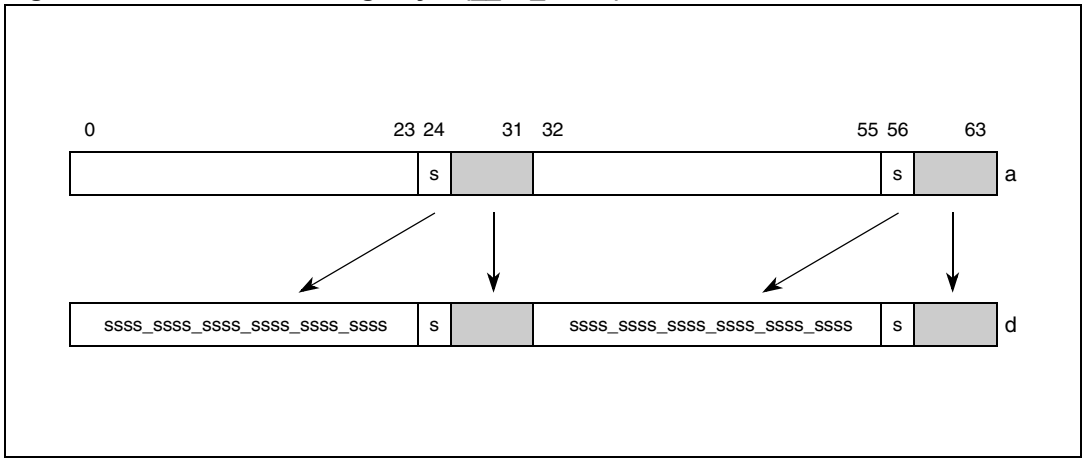


Table 46. __ev_extsb (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evextsb d,a**

__ev_extsh

Vector Extend Sign Half Word

d = __ev_extsh (a)

$d_{0:31} \leftarrow \mathbf{EXTS}(a_{16:31})$
$d_{32:63} \leftarrow \mathbf{EXTS}(a_{48:63})$

The signs of the half words in each of the elements in parameter a are extended, and the results are placed into parameter d.

Figure 41. Vector extend sign half word (__ev_extsh)

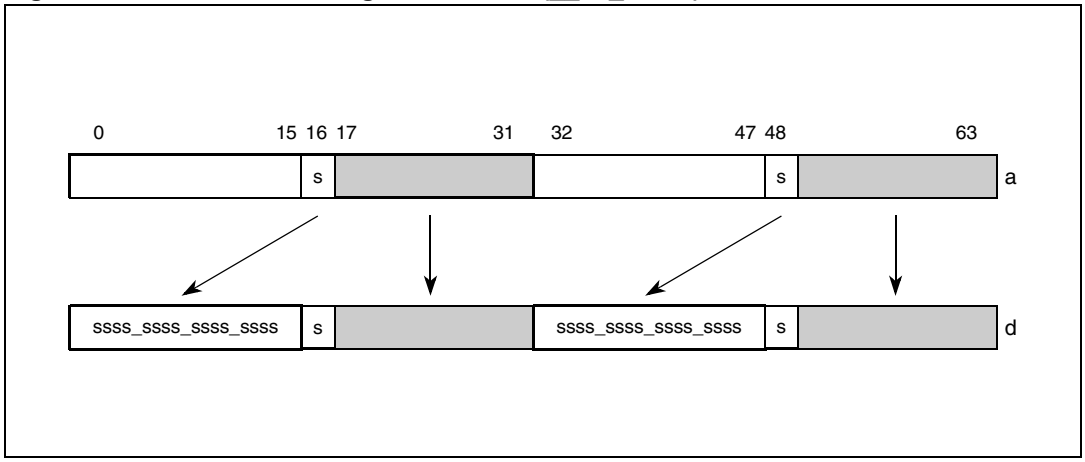


Table 47. __ev_extsh (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evextsh d,a**

__ev_fsabs

Vector Floating-Point Absolute Value

d = __ev_fsabs (a)

$d_{0:31} \leftarrow 0b0 \;||\; a_{1:31}$
$d_{32:63} \leftarrow 0b0 \;||\; a_{33:63}$

The signed bits of each element of parameter a are cleared, and the result is placed into parameter d. No exceptions are taken during the execution of this instruction.

Figure 42. Vector floating-point absolute value (__ev_fsabs)

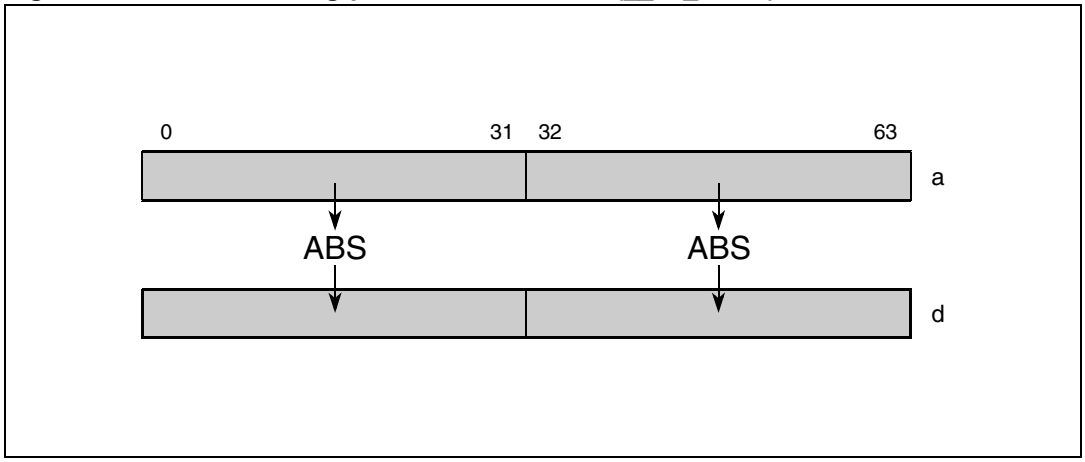


Table 48. __ev_fsabs (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsabs d,a**

__ev_fsadd

Vector Floating-Point Add

d = __ev_fsadd (a,b)

$d_{0:31} \leftarrow a_{0:31} +_{sp} b_{0:31}$
$d_{32:63} \leftarrow a_{32:63} +_{sp} b_{32:63}$

The single-precision floating-point value of each element of parameter a is added to the corresponding element in parameter b, and the results are placed in parameter d.

If an overflow condition is detected or the contents of parameters a or b are NaN or Infinity, the result is an appropriately signed maximum floating-point value.

If an underflow condition is detected, the result is an appropriately signed floating-point 0.

The following status bits are set in the SPEFSCR:

- FINV, FINVH if the contents of **r**A or **r**B are +inf, -inf, Denorm, or NaN
- FOFV, FOFVH if an overflow occurs
- FUNF, FUNFH if an underflow occurs
- FINXS, FG, FGH, FX, FXH if the result is inexact or overflow occurred and overflow exceptions are disabled

Figure 43. Vector floating-point add (__ev_fsadd)

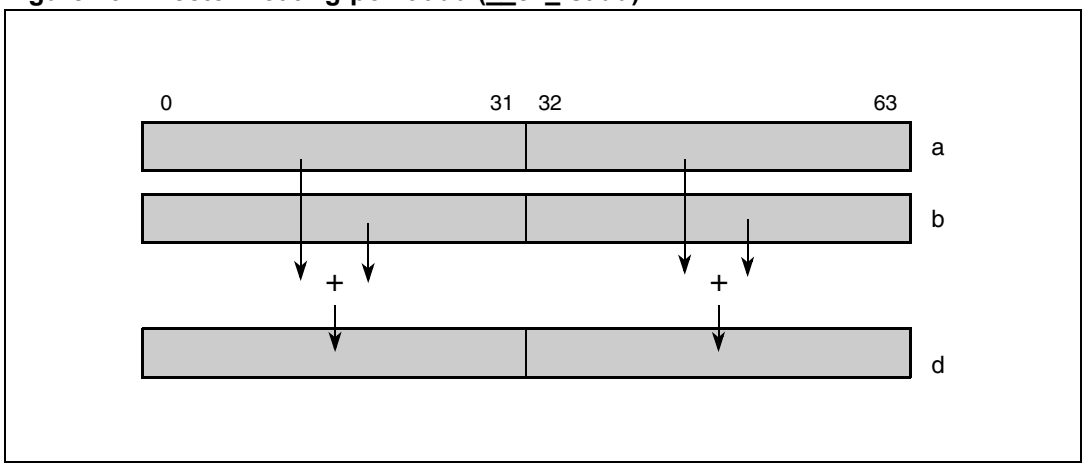


Table 49. __ev_fsadd (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfsadd d,a,b**

__ev_fscfsf

Vector Convert Floating-Point from Signed Fraction

d = __ev_fscfsf (a)

$d_{0:31} \leftarrow$ `CnvtI32ToFP32Sat(`$a_{0:31}$`, SIGN, UPPER, F)`

$d_{32:63} \leftarrow$ `CnvtI32ToFP32Sat(`$a_{32:63}$`, SIGN, LOWER, F)`

The signed fractional values in each element of parameter a are converted to the nearest single-precision floating-point value using the current rounding mode and placed in parameter d.

The following status bits are set in the SPEFSCR:

- FINXS, FG, FGH, FX, FXH if the result is inexact

Figure 44. Vector convert floating-point from signed fraction (__ev_fscfsf)

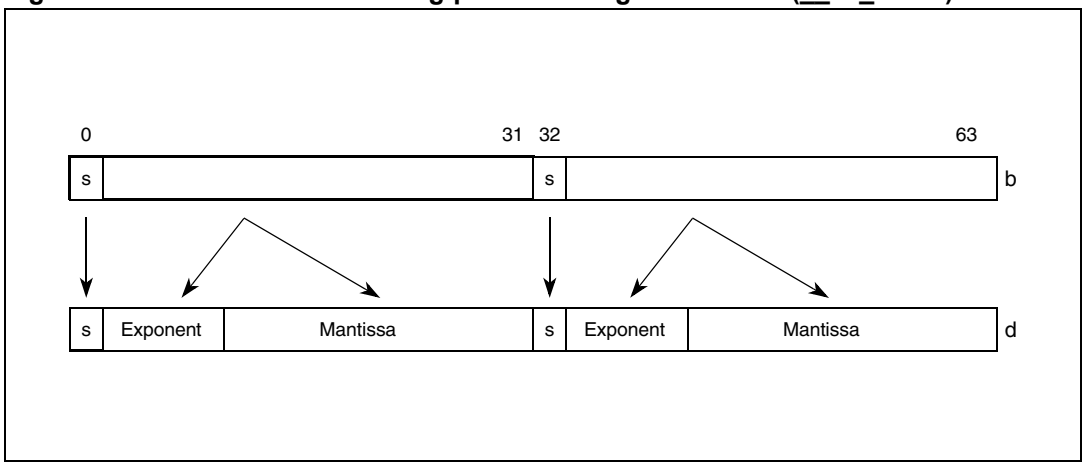


Table 50. __ev_fscfsf (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtscfsf d,a**

__ev_fscfsi

Vector Convert Floating-Point from Signed Integer
d = __ev_fscfsi (a)

$d_{0:31} \leftarrow$ `CnvtSI32ToFP32Sat(`$a_{0:31}$`, SIGN, UPPER, I)`
$d_{32:63} \leftarrow$ `CnvtSI32ToFP32Sat(`$a_{32:63}$`, SIGN, LOWER, I)`

The signed integer values in each element in parameter a are converted to the nearest single-precision floating-point value using the current rounding mode and placed in parameter d.

The following status bits are set in the SPEFSCR:

FINXS, FG, FGH, FX, FXH if the result is inexact

Figure 45. Vector convert floating-point from signed integer (__ev_fscfsi)

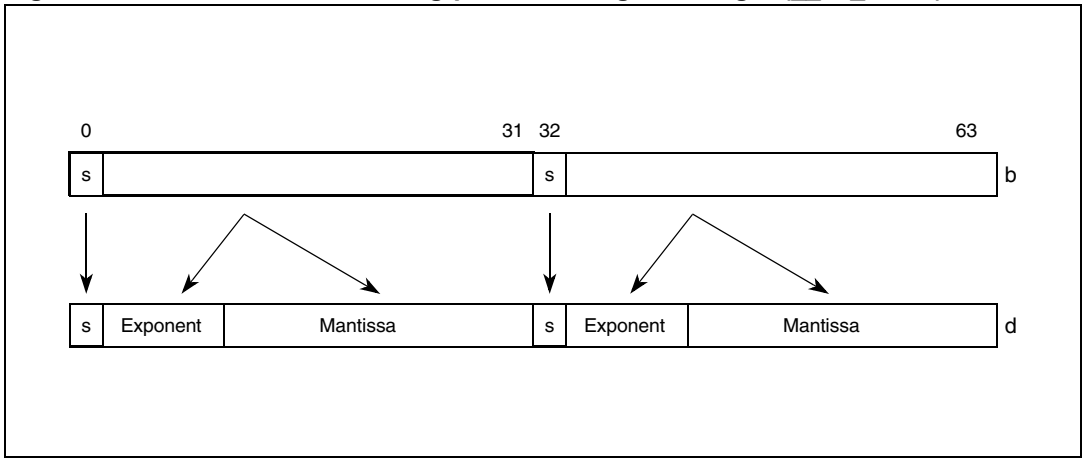


Table 51. __ev_fscfsi (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtscfsi d,a**

__ev_fscfuf

Vector Convert Floating-Point from Unsigned Fraction

d = __ev_fscfuf (a)

$d_{0:31} \leftarrow$ `CnvtI32ToFP32Sat(`$a_{0:31}$`, UNSIGN, UPPER, F)`
$d_{32:63} \leftarrow$ `CnvtI32ToFP32Sat(`$a_{32:63}$`, UNSIGN, LOWER, F)`

The unsigned fractional values in each element of parameter a are converted to the nearest single-precision floating-point value using the current rounding mode and placed in parameter d.

The following status bits are set in the SPEFSCR:

- FINXS, FG, FX if the result is inexact

Figure 46. Vector convert floating-point from unsigned fraction (__ev_fscfuf)

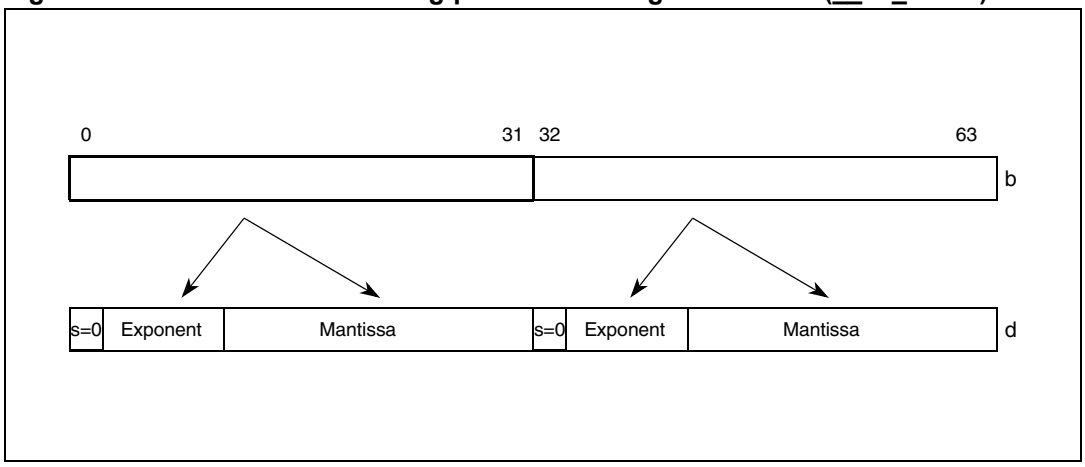


Table 52. __ev_fscfuf (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtscfuf d,a**

__ev_fscfui

Vector Convert Floating-Point from Unsigned Integer

d = __ev_fscfui (a)

$d_{0:31} \leftarrow$ `CnvtI32ToFP32Sat(`a_{031}`, UNSIGN, UPPER, I)`

$d_{32:63} \leftarrow$ `CnvtI32ToFP32Sat(`$a_{32:63}$`, UNSIGN, LOWER, I)`

The unsigned integer value in each element of parameter a are converted to the nearest single-precision floating-point value using the current rounding mode and placed in parameter d.

The following status bits are set in the SPEFSCR:

- FINXS, FG, FGH, FX, FXH if the result is inexact

Figure 47. Vector convert floating-point from unsigned integer (__ev_fscfui)

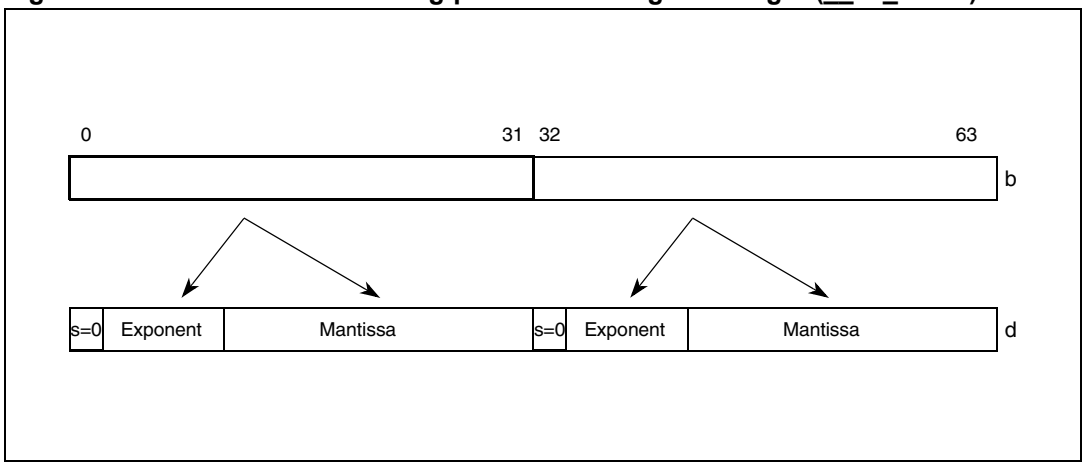


Table 53. __ev_fscfui (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtscfui d,a**

__ev_fsctsf

Vector Convert Floating-Point to Signed Fraction

d = __ev_fsctsf (a)

$d_{0:31} \leftarrow$ `CnvtFP32ToISat(`$a_{0:31}$`, SIGN, UPPER, ROUND, F)`
$d_{32:63} \leftarrow$ `CnvtFP32ToISat(`$a_{32:63}$`, SIGN, LOWER, ROUND, F)`

The single-precision floating-point value in each element of parameter a is converted to a signed fraction using the current rounding mode and the results are placed in parameter d. The result saturates if it cannot be represented in a 32-bit fraction. NaNs are converted to 0.

The following status bits are set in the SPEFSCR:

- FINV, FINVH if the contents of parameter a are +inf, -inf, Denorm, or NaN or parameter a cannot be represented in the target format
- FINXS, FG, FGH, FX, FXH if the result is inexact

Figure 48. Vector convert floating-point to signed fraction (__ev_x)

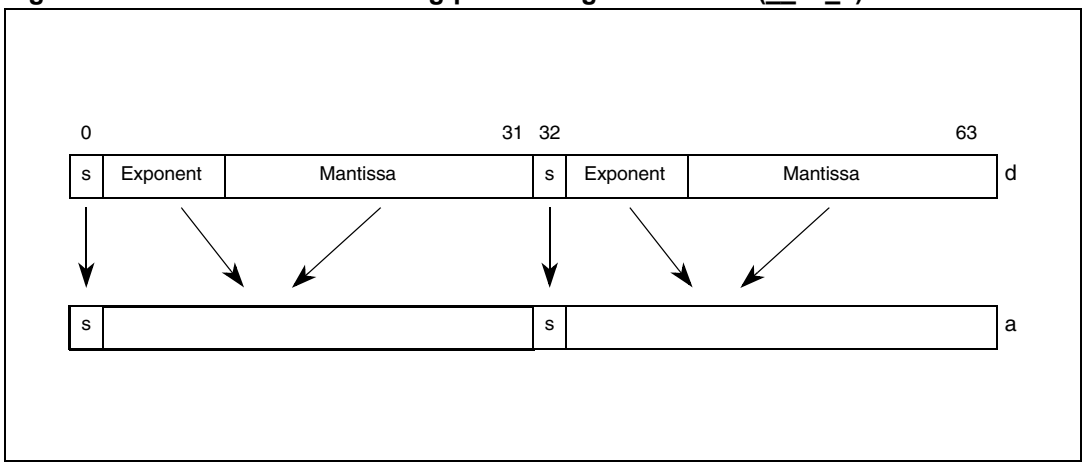


Table 54. __ev_fsctsf (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsctsf d,a**

__ev_fsctsi

Vector Convert Floating-Point to Signed Integer

d = __ev_fsctsi (a)

$d_{0:31} \leftarrow$ `CnvtFP32ToISat(`$a_{0:31}$`, SIGN, UPPER, ROUND, I)`
$d_{32:63} \leftarrow$ `CnvtFP32ToISat(`$a_{32:63}$`, SIGN, LOWER, ROUND, I)`

The single-precision floating-point value in each element of parameter a is converted to a signed integer using the current rounding mode, and the results are placed in parameter d. The result saturates if it cannot be represented in a 32-bit integer. NaNs are converted to 0.

The following status bits are set in the SPEFSCR:

- FINV, FINVH if the contents of parameter a are +inf, -inf, Denorm or NaN or parameter a cannot be represented in the target format
- FINXS, FG, FGH, FX, FXH if the result is inexact

Figure 49. Vector convert floating-point to signed integer (__ev_fsctsi)

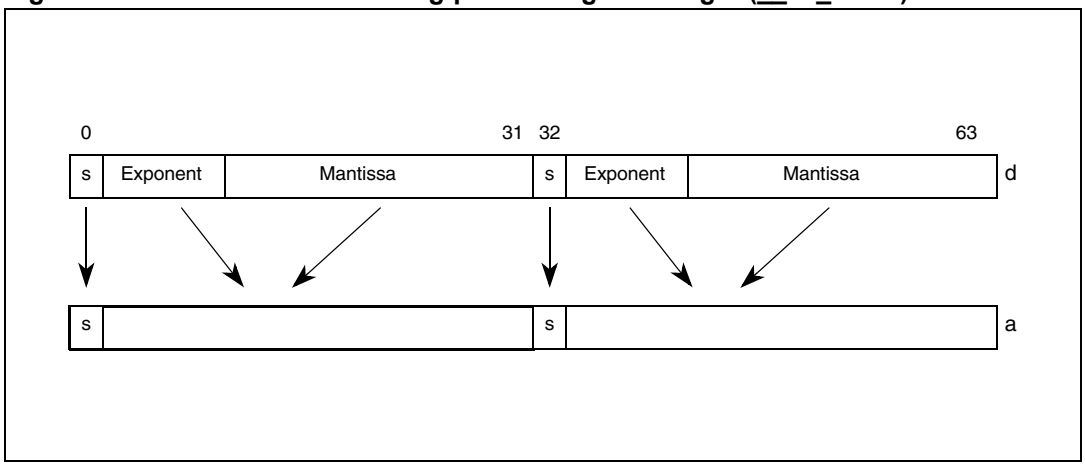


Table 55. __ev_fsctsi (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsctsi d,a**

__ev_fsctsiz

Vector Convert Floating-Point to Signed Integer with Round Toward Zero

d = __ev_fsctsiz (a)

$d_{0:31} \leftarrow$ `CnvtFP32ToISat(`$a_{0:31}$`, SIGN, UPPER, TRUNC, I)`
$d_{32:63} \leftarrow$ `CnvtFP32ToISat(`$a_{32:63}$`, SIGN, LOWER, TRUNC, I)`

The single-precision floating-point value in each element of parameter a is converted to a signed integer using the rounding mode Round Towards Zero, and the results are placed in parameter d. The result saturates if it cannot be represented in a 32-bit integer. NaNs are converted to 0.

The following status bits are set in the SPEFSCR:

- FINV, FINVH if the contents of parameter a are +inf, -inf, Denorm, or NaN or if parameter a cannot be represented in the target format
- FINXS, FG, FGH, FX, FXH if the result is inexact

Figure 50. Vector convert floating-point to signed integer with roundtoward zero (__ev_fsctsiz)

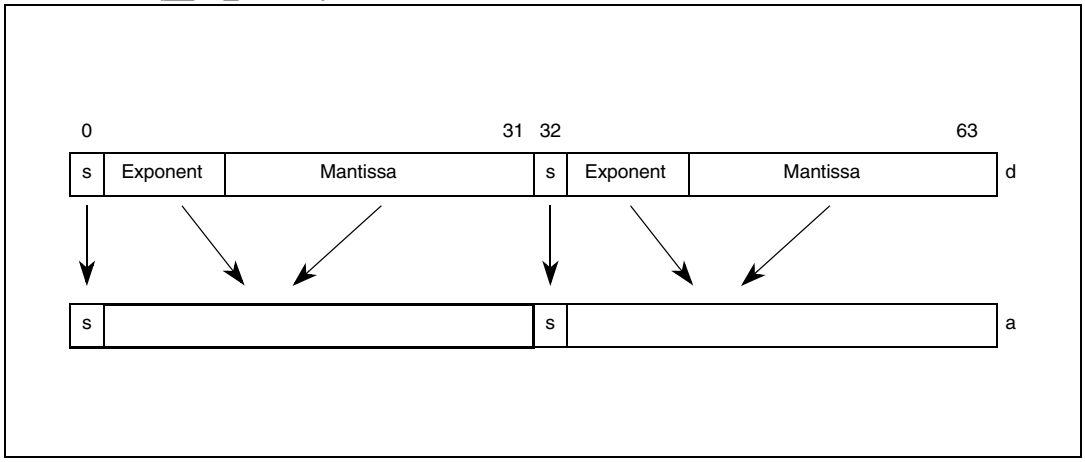


Table 56. __ev_fsctsiz (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsctsiz d,a**

__ev_fsctuf

Vector Convert Floating-Point to Unsigned Fraction

d = __ev_fsctuf (a)

$d_{0:31} \leftarrow$ `CnvtFP32ToISat(`$a_{0:31}$`, UNSIGN, UPPER, ROUND, F)`

$d_{32:63} \leftarrow$ `CnvtFP32ToISat(`$a_{32:63}$`, UNSIGN, LOWER, ROUND, F)`

The single-precision floating-point value in each element of parameter a is converted to an unsigned fraction using the current rounding mode, and the results are placed in parameter d. The result saturates if it cannot be represented in a 32-bit unsigned fraction. NaNs are converted to 0.

The following status bits are set in the SPEFSCR:

- FINV, FINVH if the contents of parameter a are +inf, -inf, Denorm, or NaN or if parameter a cannot be represented in the target format
- FINXS, FG, FGH, FX, FXH if the result is inexact

Figure 51. Vector convert floating-point to unsigned fraction (__ev_fsctuf)

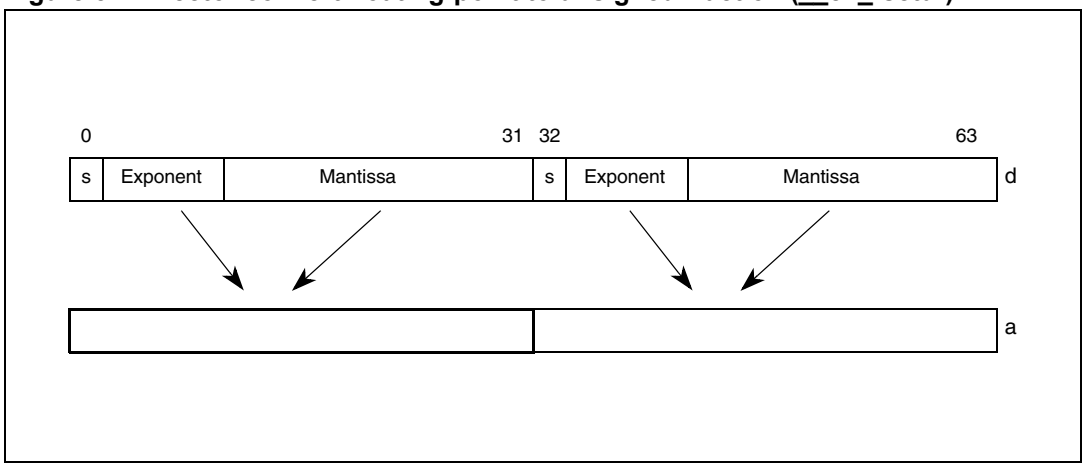


Table 57. __ev_fsctuf (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsctuf d,a**

__ev_fsctui

Vector Convert Floating-Point to Unsigned Integer

d = __ev_fsctui (a)

$d_{0:31} \leftarrow$ `CnvtFP32ToISat(`$a_{0:31}$`, UNSIGN, UPPER, ROUND, I)`
$d_{32:63} \leftarrow$ `CnvtFP32ToISat(`$a_{32:63}$`, UNSIGN, LOWER, ROUND, I)`

The single-precision floating-point value in each element of parameter a is converted to an unsigned integer using the current rounding mode, and the results are placed in parameter d. The result saturates if it cannot be represented in a 32-bit unsigned integer. NaNs are converted to 0.

The following status bits are set in the SPEFSCR:

- FINV, FINVH if the contents of parameter a are +inf, -inf, Denorm or NaN or parameter a cannot be represented in the target format
- FINXS, FG, FGH, FX, FXH if the result is inexact

Figure 52. Vector convert floating-point to unsigned integer (__ev_fsctui)

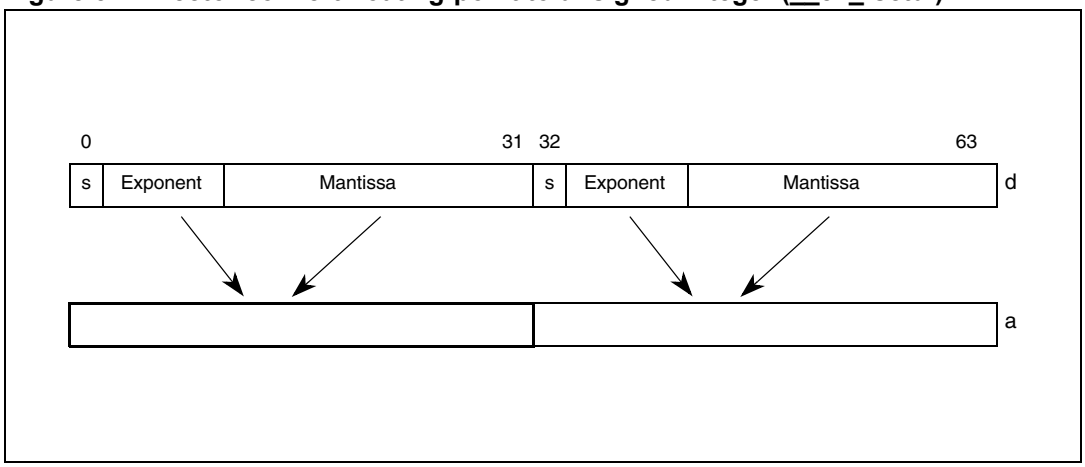


Table 58. __ev_fsctui (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsctui d,a**

__ev_fsctuiz

Vector Convert Floating-Point to Unsigned Integer with Round toward Zero

d = __ev_fsctuiz (a)

$d_{0:31} \leftarrow$ `CnvtFP32ToISat(`$a_{0:31}$`, UNSIGN, UPPER, TRUNC, I)`
$d_{32:63} \leftarrow$ `CnvtFP32ToISat(`$a_{32:63}$`, UNSIGN, LOWER, TRUNC, I)`

The single-precision floating-point value in each element of parameter a is converted to an unsigned integer using the rounding mode Round towards Zero, and the results are placed in parameter d. The result saturates if it cannot be represented in a 32-bit unsigned integer. NaNs are converted to 0.

The following status bits are set in the SPEFSCR:

- FINV, FINVH if the contents of parameter a are +inf, -inf, Denorm, or NaN or parameter a cannot be represented in the target format
- FINXS, FG, FGH, FX, FXH if the result is inexact

Figure 53. Vector convert floating-point to unsigned integer with roundtoward zero (__ev_fsctuiz)

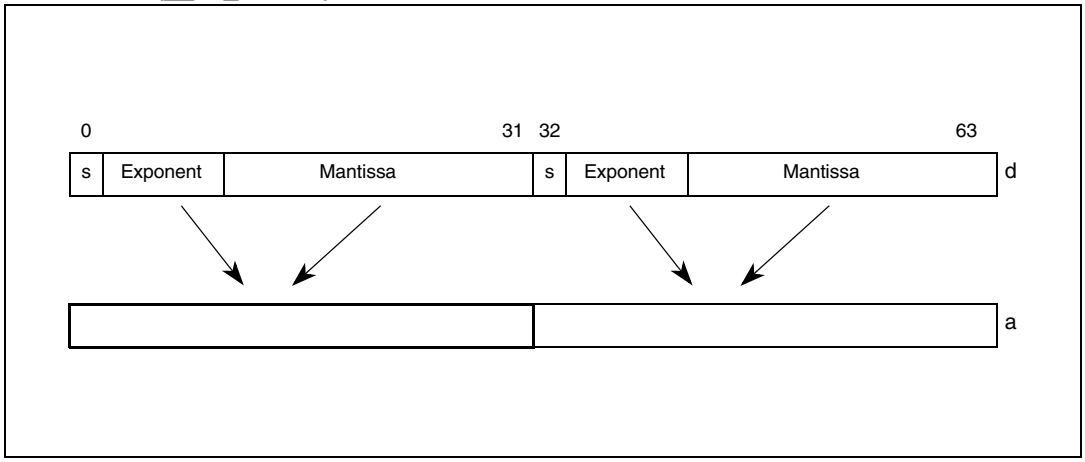


Table 59. __ev_fsctuiz (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsctuiz d,a**

__ev_fsdiv

Vector Floating-Point Divide

d = __ev_fsdiv(a,b)

$d_{0:31} \leftarrow a_{0:31} \div_{sp} b_{0:31}$
$d_{32:63} \leftarrow a_{32:63} \div_{sp} d_{32:63}$

The single-precision floating-point value in each element of parameter a is divided by the corresponding elements in parameter b, and the results are placed in parameter d.

If an overflow is detected, parameter b is a Denorm (or 0 value), or parameter a is a NaN or Infinity and parameter b is a normalized number, the result is an appropriately signed maximum floating-point value.

If an underflow is detected or parameter b is a NaN or Infinity, the result is an appropriately signed floating-point 0.

The following status bits are set in the SPEFSCR:

- FINV, FINVH if the contents of parameter a or b are +inf, -inf, Denorm, or NaN
- FOFV, FOFVH if an overflow occurs
- FUNV, FUNVH if an underflow occurs
- FDBZS, FDBZ, FDBZH if a divide by zero occurs
- FINXS, FG, FGH, FX, FXH if the result is inexact or overflow occurred and overflow exceptions are disabled

Figure 54. Vector floating-point divide (__ev_fsdiv)

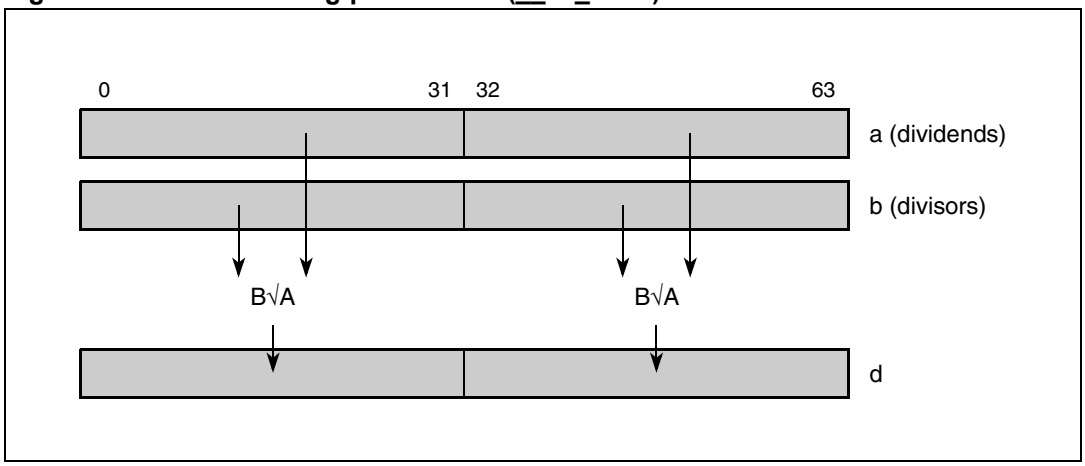


Table 60. __ev_fsdiv (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfsdiv d,a,b**

__ev_fsmul

Vector Floating-Point Multiply

d = __ev_fsmul (a,b)

$d_{0:31} \leftarrow a_{0:31} \times_{sp} b_{0:31}$
$d_{32:63} \leftarrow a_{32:63} \times_{sp} b_{32:63}$

Each single-precision floating-point element of parameter a is multiplied with the corresponding element of parameter b, and the result is stored in parameter d. If an overflow is likely, pmax or nmax is stored in parameter d. If an underflow is likely, +0, or –0 is stored in parameter d. The following condition defines when an overflow is likely and the corresponding result for each element of the vector:

```
ei = (ea - 127) + (eb - 127) + 127
if (sa = sb) then
   if (ei ≥ 127) then r = pmax
   else if (ei < -126) then r = +0
else
   if (ei ≥ 127) then r = nmax
   else if (ei < -126) then r = -0
```

- If the contents of parameter a or b are +inf, –inf, Denorm, QNaN, or SNaN, at least one of the SPEFSCR[FINVH] or SPEFSCR[FINV] bits is set.
- If an overflow occurs or is likely, at least one of the SPEFSCR[FOVFH] or SPEFSCR[FOVF] bits is set.
- If an underflow occurs or is likely, at least one of the SPEFSCR[FUNFH] or SPEFSCR[FUNF] bits is set.
- If the exception is enabled for the high or low element in which the error occurs, the exception is taken.

Figure 55. Vector floating-point multiply (__ev_fsmul)

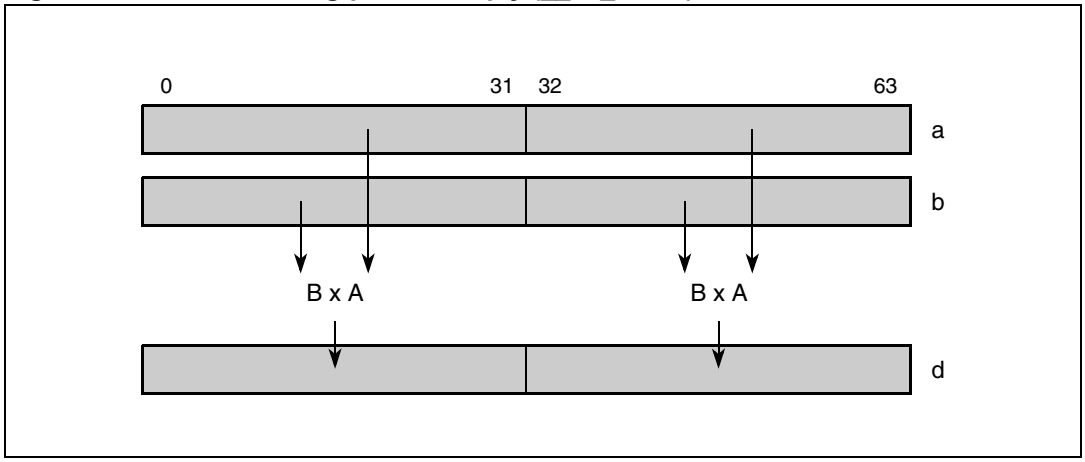


Table 61. __ev_fsmul (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfsmul d,a,b**

__ev_fsnabs

Vector Floating-Point Negative Absolute Value

d = __ev_fsnabs (a)

$d_{0:31} \leftarrow 0b1 \;||\; a_{1:31}$
$d_{32:63} \leftarrow 0b1 \;||\; a_{33:63}$

The signed bits of each element of parameter a are all set and the result is placed into parameter d. No exceptions are taken during the execution of this instruction.

Figure 56. Vector floating-point negative absolute value (__ev_fsnabs)

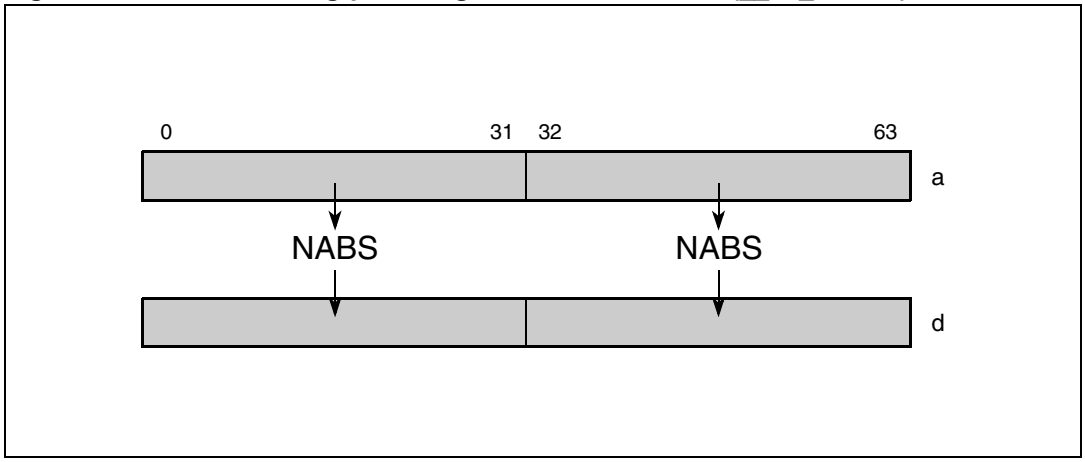


Table 62. __ev_fsnabs (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsnabs d,a**

__ev_fsneg

Vector Floating-Point Negate

d = __ev_fsneg (a)

$d_{0:31} \leftarrow \neg a_0 \;||\; a_{1:31}$
$d_{32:63} \leftarrow \neg a_{32} \;||\; a_{33:63}$

The signed bits of each element of parameter a are complemented and the result is placed into parameter d. No exceptions are taken during the execution of this instruction.

Figure 57. Vector floating-point negate (__ev_fsneg)

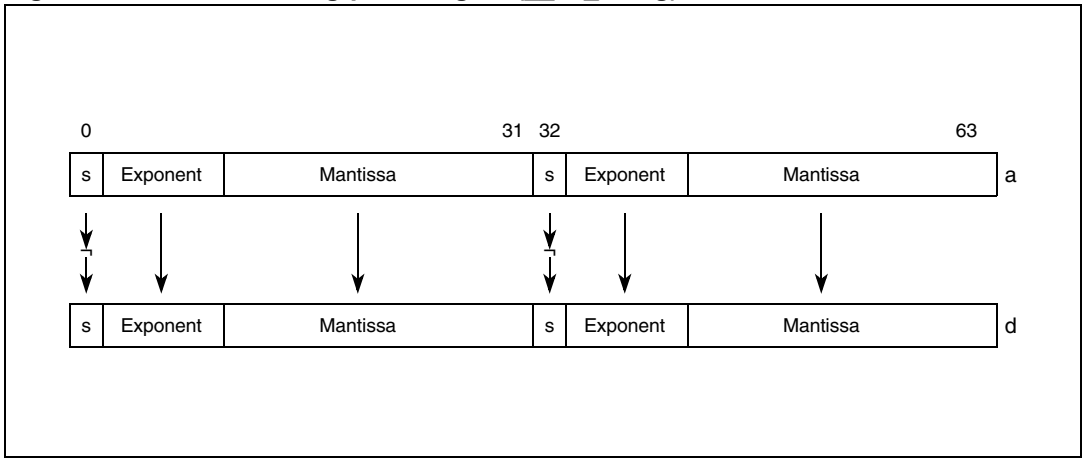


Table 63. __ev_fsneg (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsneg d,a**

__ev_fssub

Vector Floating-Point Subtract

d = __ev_fssub (a,b)

$d_{0:31} \leftarrow a_{0:31} -_{sp} b_{0:31}$
$d_{32:63} \leftarrow a_{32:63} -_{sp} b_{32:63}$

Each single-precision floating-point element of parameter b is subtracted from the corresponding element of parameter a and the result is stored in parameter d. If an overflow is likely, pmax or nmax is stored in parameter d. If an underflow is likely, +0 or –0 is stored in parameter d. The following condition defines how boundary cases of inputs (+inf, –inf, Denorm, QNaN, SNaN) are treated, when an overflow is likely, and the corresponding result for each element of the vector:

```
if ((sa = 0) & (sb = 1)) then
  if (max(ea, eb) ≥ 127) then r = pmax
else if ((sa = 1) & (sb = 0)) then
  if (max(ea, eb) ≥ 127) then r = nmax
else if (sa = sb) then
            // Boundary case to be defined later
```

- If the contents of parameter a or b are +inf, –inf, Denorm, QNaN, or SNaN, at least one of the SPEFSCR[FINVH] or SPEFSCR[FINV] bits is set.
- If an overflow occurs or is likely, the SPEFSCR[FOVFH] or SPEFSCR[FOVF] bits is set.
- If an underflow occurs or is likely, at least one of the SPEFSCR[FUNFH] or SPEFSCR[FUNF] bits is set.
- If the exception is enabled for the high or low element in which the error occurs, the exception is taken.

Figure 58. Vector Floating-Point subtract (__ev_fssub)

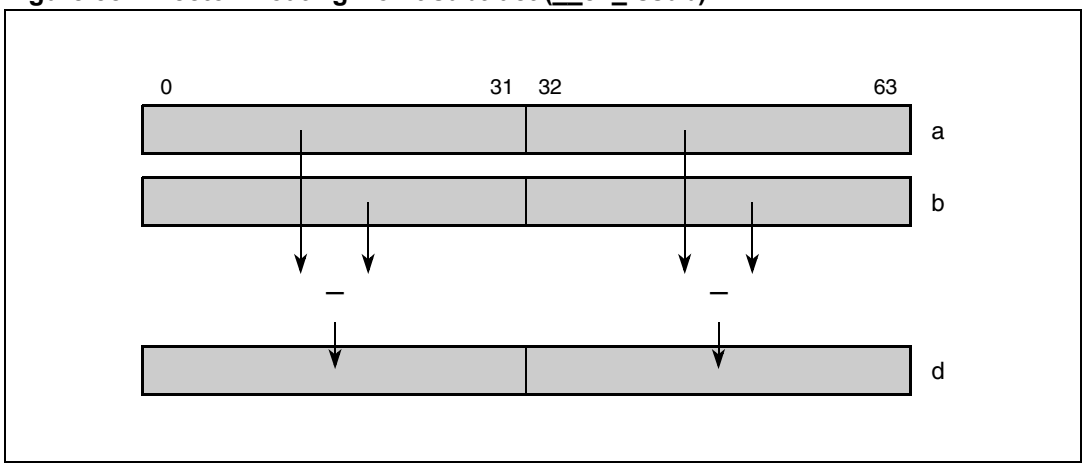


Table 64. __ev_fssub (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfssub d,a,b**

__ev_ldd

Vector Load Double Word into Double Word

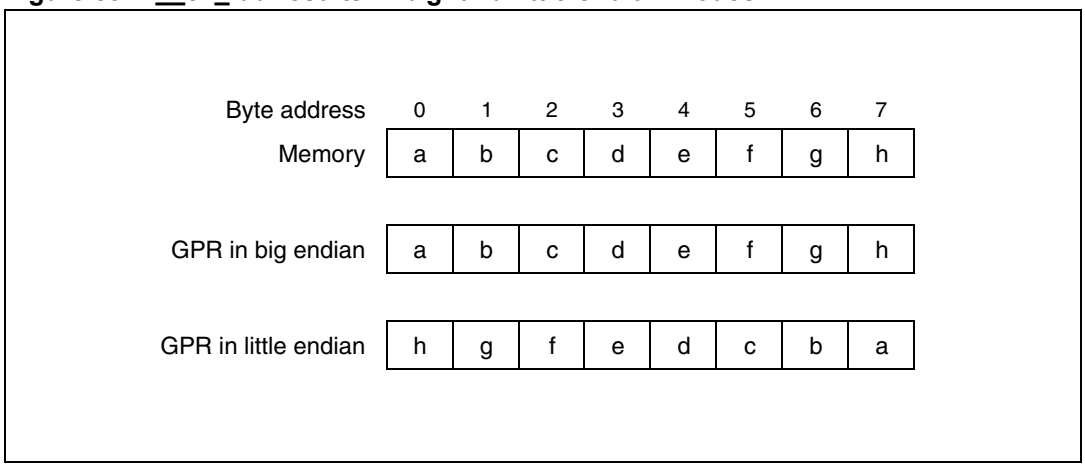
d = __ev_ldd (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*8)
d ← MEM(EA, 8)
```

The double word addressed by EA is loaded from memory and placed in parameter d.

Figure 59 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 59. __ev_ldd results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the effective address (EA) is not double-word aligned.

Table 65. __ev_ldd (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	5-bit unsigned	**evldd d,a,b**

__ev_lddx__ev_lddx

Vector Load Double Word into Double Word Indexed

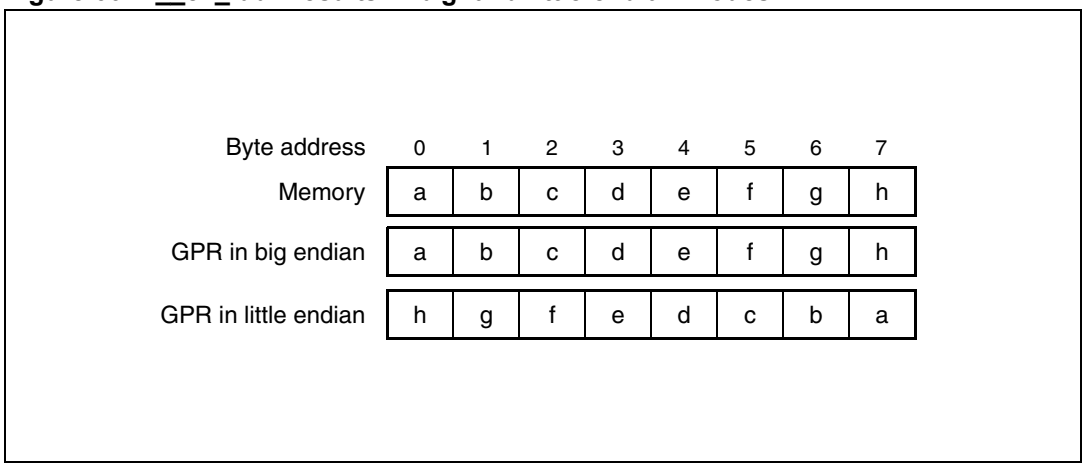
d = __ev_lddx **(a,b)**

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d ← MEM(EA, 8)
```

The double word addressed by EA is loaded from memory and placed in parameter d.

Figure 60 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 60. __ev_lddx results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not double-word aligned.

Table 66. __ev_lddx__ev_lddx (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	int32_t	**evlddx d,a,b**

__ev_ldh

Vector Load Double into Four Half Words

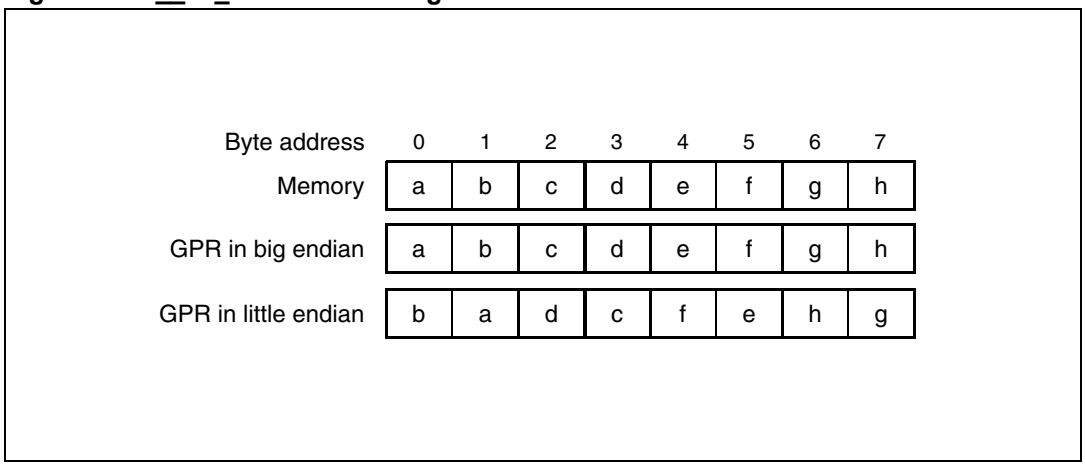
d = __ev_ldh (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*8)
d0:15 ← MEM(EA, 2)
d16:31 ← MEM(EA+2,2)
d32:47 ← MEM(EA+4,2)
d48:63 ← MEM(EA+6,4)
```

The double word addressed by EA is loaded from memory and placed in parameter d.

Figure 61 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 61. __ev_ldh results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not double-word aligned.

Table 67. __ev_ldh (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	5-bit unsigned	**evldh d,a,b**

__ev_ldhx

Vector Load Double into Four Half Words Indexed

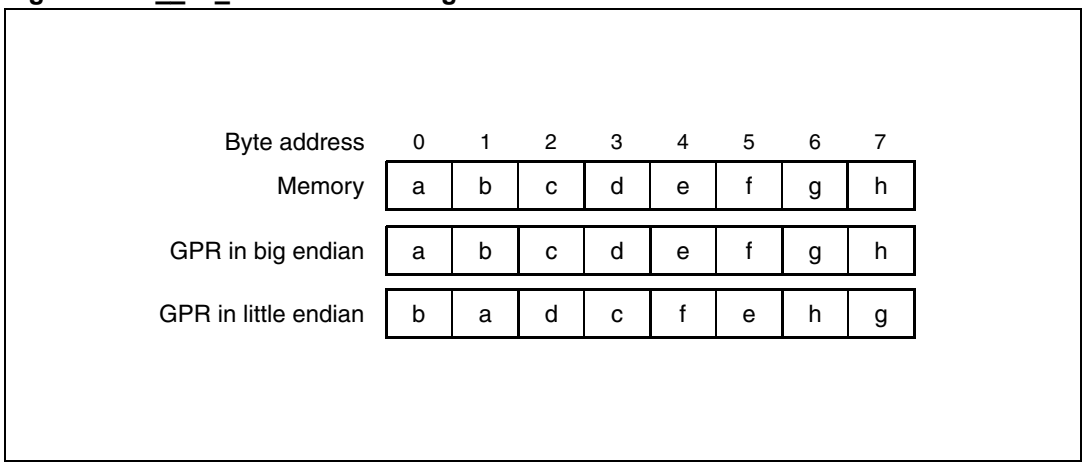
d = __ev_ldhx (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:15 ← MEM(EA, 2)
d16:31 ← MEM(EA+2,2)
d32:47 ← MEM(EA+4,2)
d48:63 ← MEM(EA+6,4)
```

The double word addressed by EA is loaded from memory and placed in parameter d.

Figure 62 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 62. __ev_ldhx results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not double-word aligned.

Table 68. __ev_ldhx (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	int32_t	**evldhx d,a,b**

__ev_ldw

Vector Load Double into Two Words

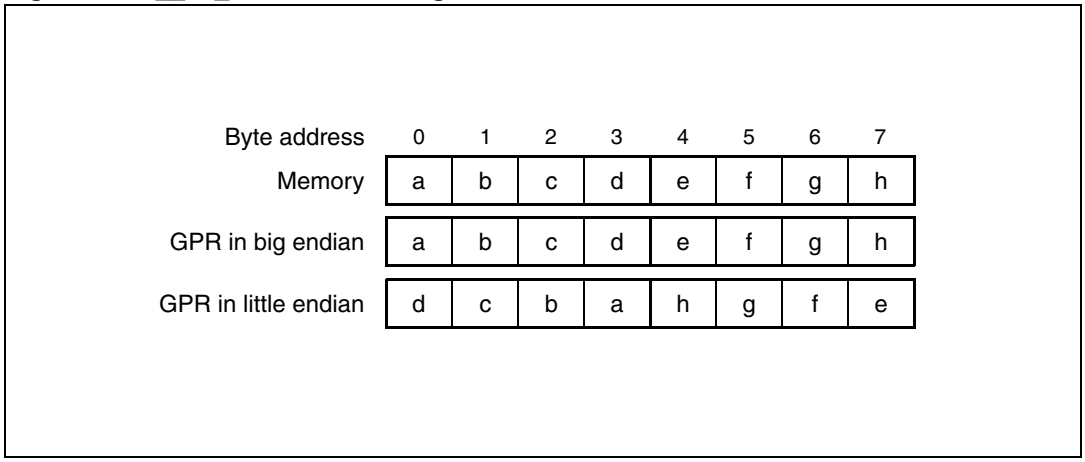
d = __ev_ldw (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*8)
d0:31 ← MEM(EA, 4)
d32:63 ← MEM(EA+4, 4)
```

The double word addressed by EA is loaded from memory and placed in parameter d.

Figure 63 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 63. __ev_ldw results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not double-word aligned.

Table 69. __ev_ldw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	5-bit unsigned	**evldw d,a,b**

__ev_ldwx

Vector Load Double into Two Words Indexed

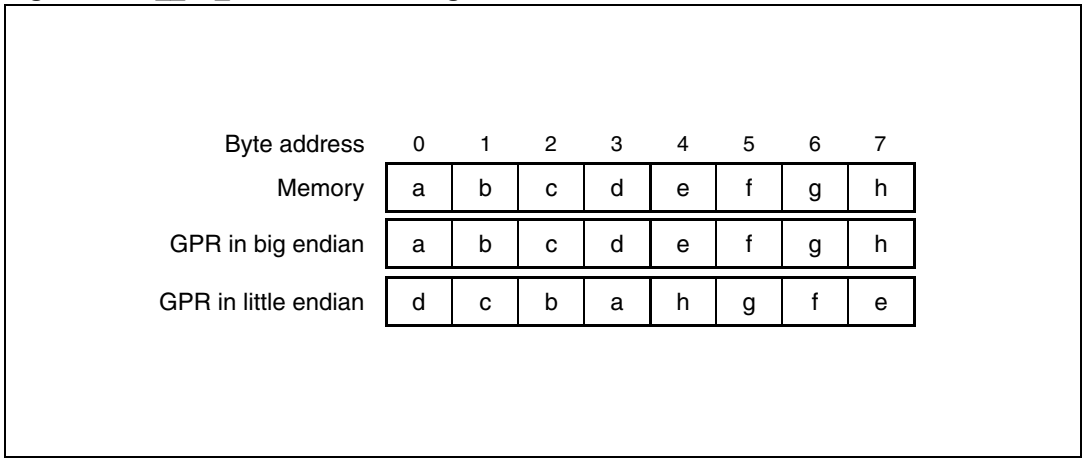
d = __ev_ldwx (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:31 ← MEM(EA, 4)
d32:63 ← MEM(EA+4, 4)
```

The double word addressed by EA is loaded from memory and placed in parameter d.

Figure 64 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 64. __ev_ldwx results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not double-word aligned.

Table 70. __ev_ldwx (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	int32_t	**evldwx d,a,b**

__ev_lhhesplat

Vector Load Half Word into Half Words Even and Splat

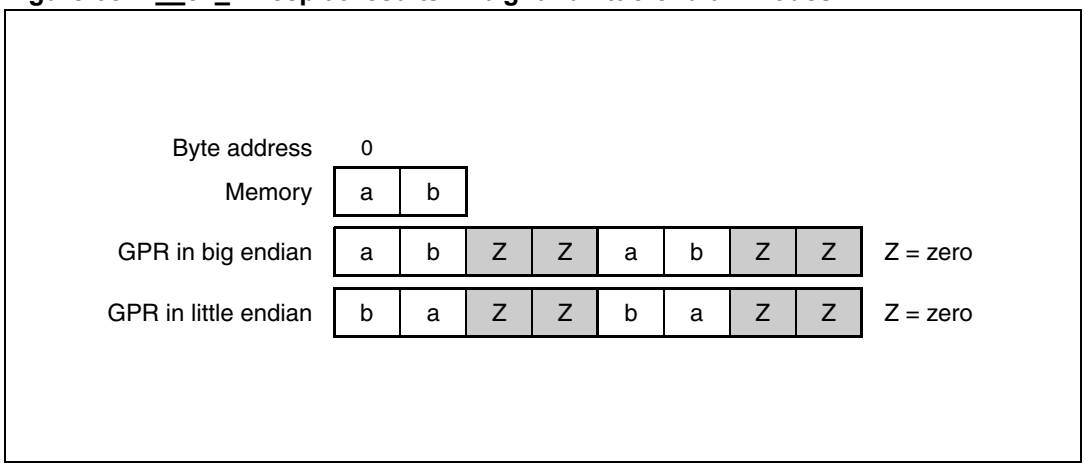
d = __ev_lhhesplat **(a,b)**

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*2)
d0:15 ← MEM(EA,2)
d16:31 ← 0x0000
d32:47 ← MEM(EA,2)
d48:63 ← 0x0000
```

The half word addressed by EA is loaded from memory and placed in the even half words of each element of parameter d.

Figure 65 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 65. __ev_lhhesplat results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not half word-aligned.

Table 71. __ev_lhhesplat (registers altered by).

d	a	b	Maps to
__ev64_opaque	uint16_t	5-bit unsigned	**evlhhesplat d,a,b**

__ev_lhhesplatx

Vector Load Half Word into Half Words Even and Splat-Indexed

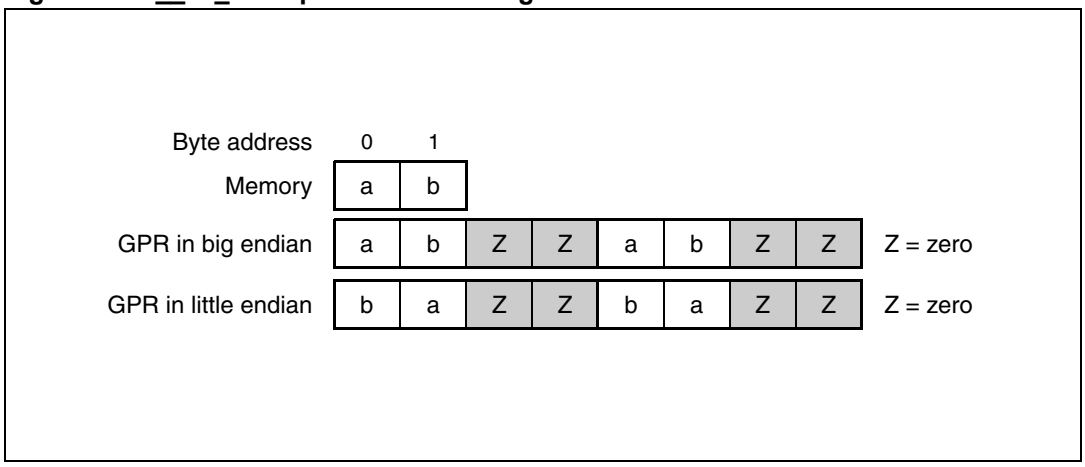
d = __ev_lhhesplatx (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:15 ← MEM(EA,2)
d16:31 ← 0x0000
d32:47 ← MEM(EA,2)
d48:63 ← 0x0000
```

The half word addressed by EA is loaded from memory and placed in the even half words of each element of parameter d.

Figure 66 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 66. __ev_lhhesplatx results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not half word-aligned.

Table 72. __ev_lhhesplatx (registers altered by).

d	a	b	Maps to
__ev64_opaque	uint16_t	int32_t	**evlhhesplatx d,a,b**

__ev_lhhossplat

Vector Load Half Word into Half Word Odd Signed and Splat

d = __ev_lhhossplat (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*2)
d0:31 ← EXTS(MEM(EA,2))
d32:63 ← EXTS(MEM(EA,2))
```

The half word addressed by EA is loaded from memory and placed in the odd half words sign extended in each element of parameter d.

Figure 67 shows how bytes are loaded into parameter d as determined by the endian mode.

- In big-endian mode, the msb of parameter a is sign-extended.
- In little-endian mode, the msb of parameter b is sign-extended.

Note: During implementation, an alignment exception occurs if the EA is not half word-aligned.

Figure 67. __ev_lhhossplat results in big- and little-endian modes

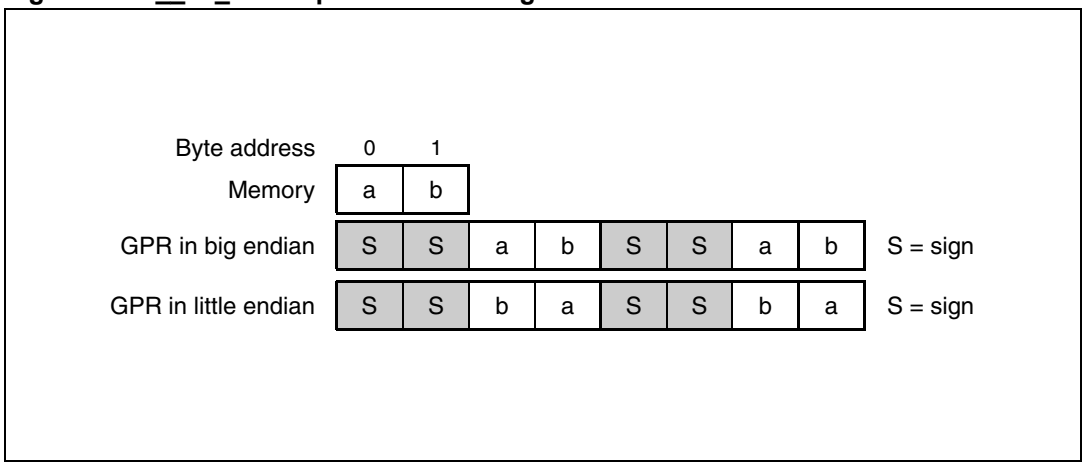


Table 73. __ev_lhhossplat (registers altered by).

d	a	b	Maps to
__ev64_opaque	uint16_t	5-bit unsigned	**evlhhosplat d,a,b**

__ev_lhhossplatx

Vector Load Half Word into Half Word Odd Signed and Splat-Indexed

d = __ev_lhhossplatx (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:31 ← EXTS(MEM(EA,2))
d32:63 ← EXTS(MEM(EA,2))
```

The half-word addressed by EA is loaded from memory and placed in the odd half-words sign extended in each element of parameter d.

Figure 68 shows how bytes are loaded into parameter d as determined by the endian mode.

- In big-endian mode, the msb of parameter a is sign-extended.
- In little-endian mode, the msb of parameter b is sign-extended.

Note: During implementation, an alignment exception occurs if the EA is not half word-aligned.

Figure 68. __ev_lhhossplatx results in big- and little-endian modes

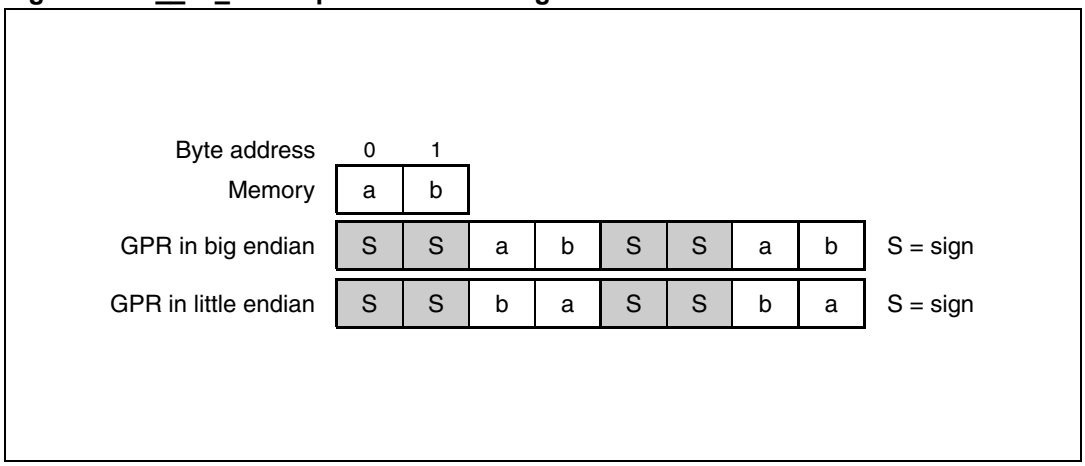


Table 74. __ev_lhhossplatx (registers altered by).

d	a	b	Maps to
__ev64_opaque	uint16_t	int32_t	**evlhhosplatx d,a,b**

__ev_lhhousplat

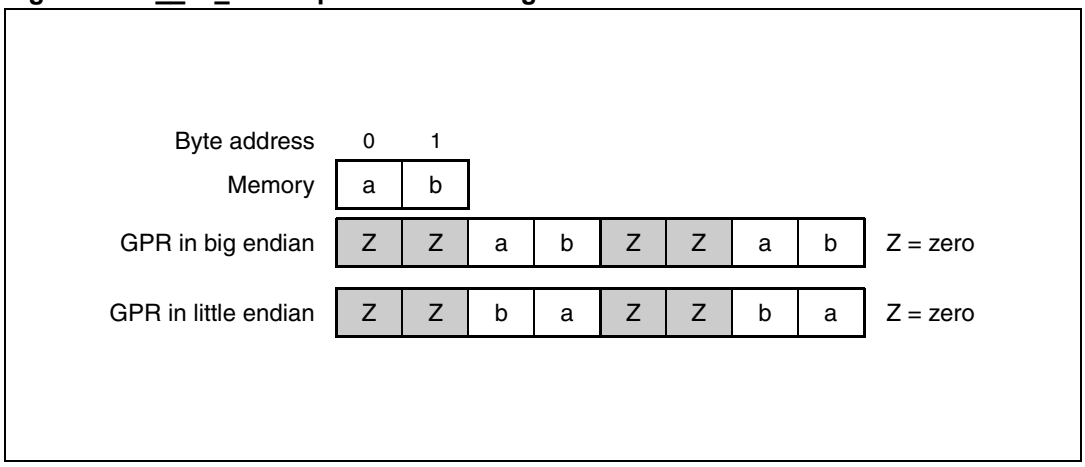
Vector Load Half Word into Half Word Odd Unsigned and Splat

d = __ev_lhhousplat (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*2)
d0:15 ← 0x0000
d16:31 ← MEM(EA,2)
d32:47 ← 0x0000
d48:63 ← MEM(EA,2)
```

The half word addressed by EA is loaded from memory and placed in the odd half words zero extended in each element of parameter d. The following diagram shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 69. __ev_lhhousplat results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not half word-aligned.

Table 75. __ev_lhhousplat (registers altered by).

d	a	b	Maps to
__ev64_opaque	uint16_t	5-bit unsigned	**evlhhousplat d,a,b**

__ev_lhhousplatx

Vector Load Half Word into Half Word Odd Unsigned and Splat-Indexed

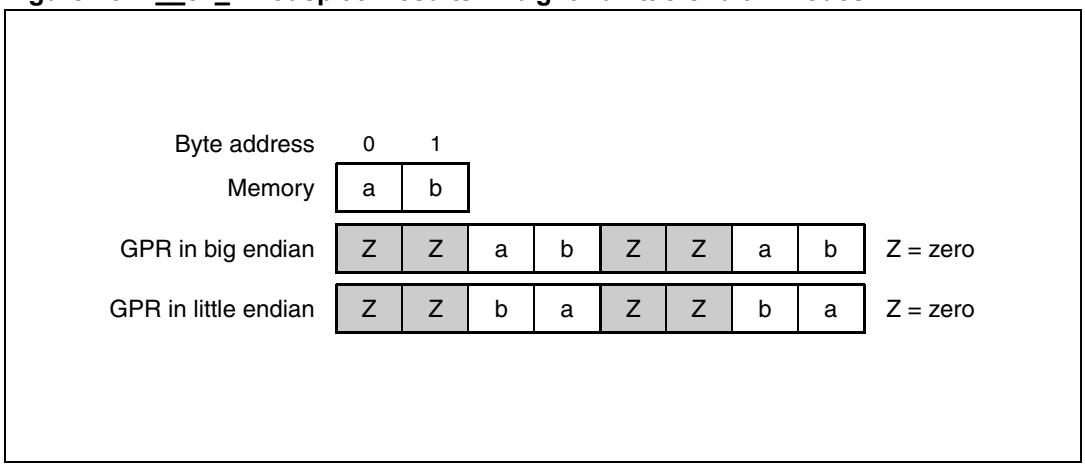
d = __ev_lhhousplatx (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:15 ← 0x0000
d16:31 ← MEM(EA,2)
d32:47 ← 0x0000
d48:63 ← MEM(EA,2)
```

The half-word addressed by EA is loaded from memory and placed in the odd half words zero extended in each element of parameter d.

Figure 70 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 70. __ev_lhhousplatx results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not half word-aligned.

Table 76. __ev_lhhousplatx (registers altered by).

d	a	b	Maps to
__ev64_opaque	uint16_t	int32_t	**evlhhousplatx d,a,b**

__ev_lower_eq

Vector Lower Bits Equal

d = __ev_lower_eq(a,b)

```
if (a32:63 = b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b.

Figure 71. Vector lower equal (__ev_lower_eq)

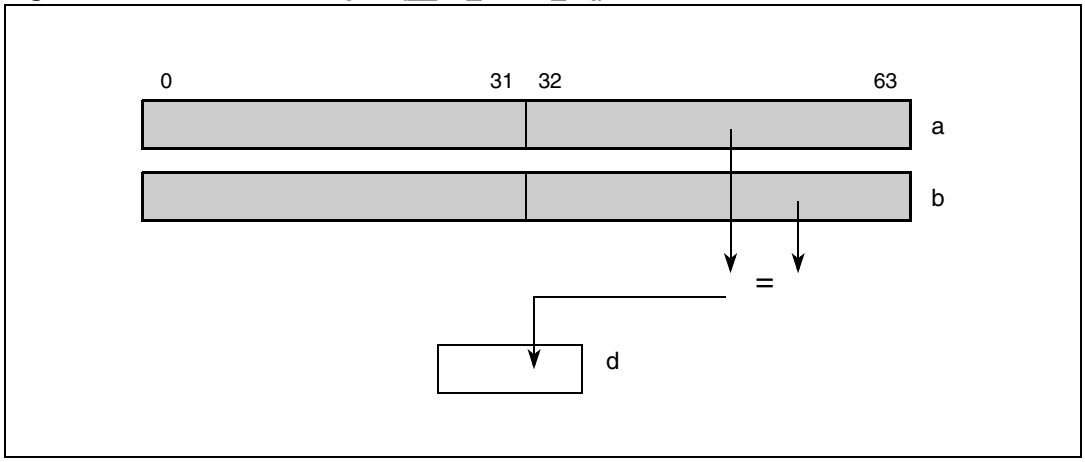


Table 77. __ev_lower_eq (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpeq x,a,b**

__ev_lower_fs_eq

Vector Lower Bits Floating-Point Equal

d = __ev_lower_fs_eq(a,b)

```
if (a32:63 = b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b.

Figure 72. Vector lower floating-point equal (__ev_lower_fs_eq)

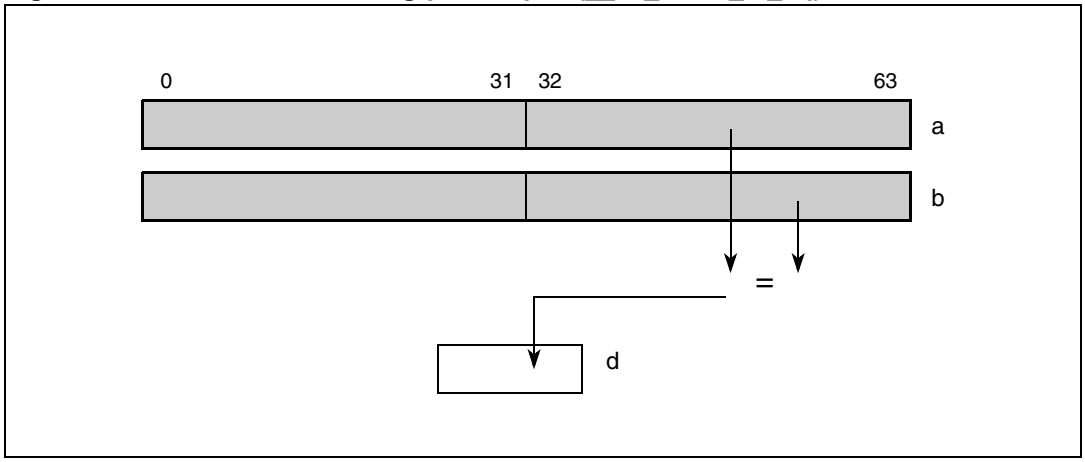


Table 78. __ev_lower_fs_eq (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmpeq x,a,b**

__ev_lower_fs_gt

Vector Lower Bits Floating-Point Greater Than

d = __ev_lower_fs_gt(a,b)

```
if (a32:63 > b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

Figure 73. Vector lower floating-point greater than (__ev_lower_fs_gt)

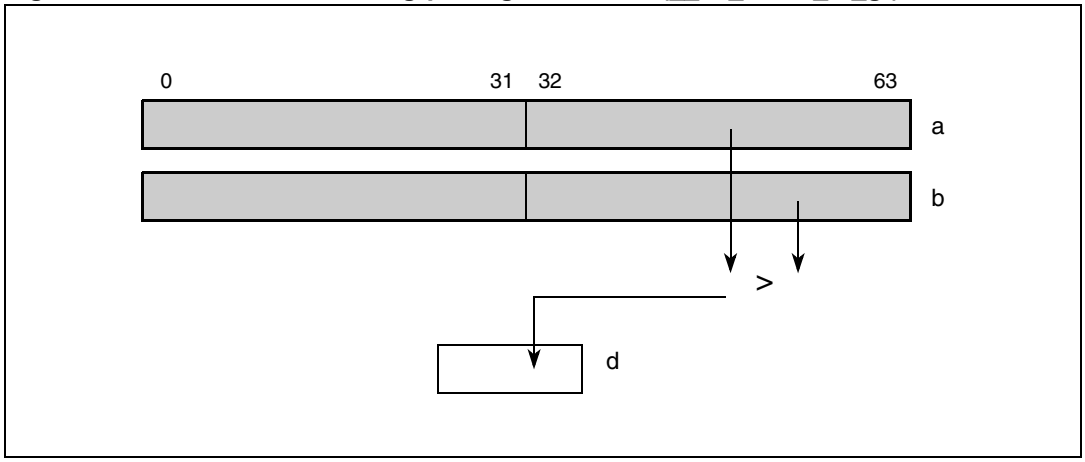


Table 79. __ev_lower_fs_gt (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmpgt x,a,b**

__ev_lower_fs_lt

Vector Lower Bits Floating-Point Less Than

d = __ev_lower_fs_lt(a,b)

```
if (a32:63 < b32:63) then d← true
else d←false
```

This intrinsic returns true if the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

Figure 74. Vector lower floating-point less than (__ev_lower_fs_lt)

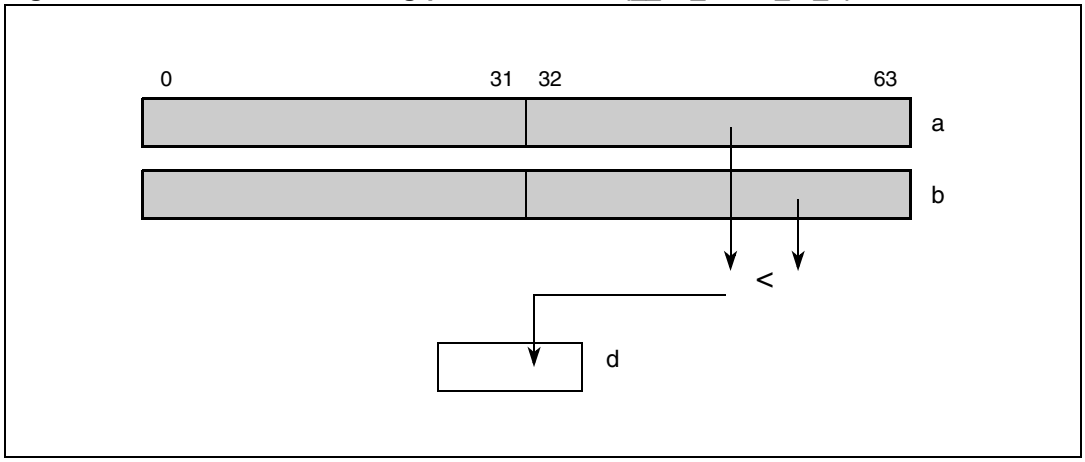


Table 80. __ev_lower_fs_lt (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmplt x,a,b**

__ev_lower_fs_tst_eq

Vector Lower Bits Floating-Point TestEqual

d = __ev_lower_fs_tst_eq(a,b)

```
if (a32:63 = b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b. This intrinsic differs from __ev_lower_fs_eq because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_lower_fs_eq instead.

Figure 75. Vector lower floating-point test equal (__ev_lower_fs_tst_eq)

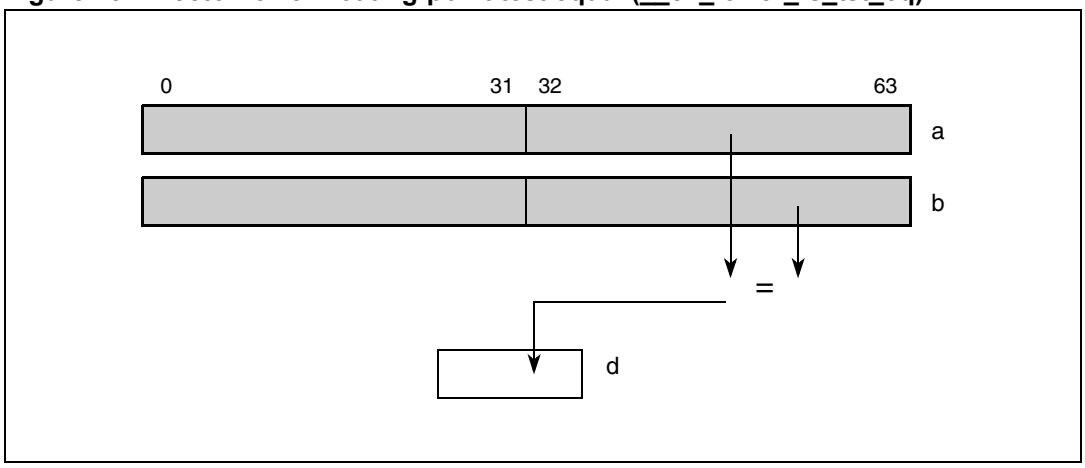


Table 81. __ev_lower_fs_tst_eq (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststeq x,a,b**

__ev_lower_fs_tst_gt

Vector Lower Bits Floating-Point Test Greater Than

d = __ev_lower_fs_tst_gt(a,b)

```
if (a32:63 > b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b. This intrinsic differs from __ev_lower_fs_gt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_lower_fs_gt instead.

Figure 76. Vector lower floating-point test greater than (__ev_lower_fs_tst_gt)

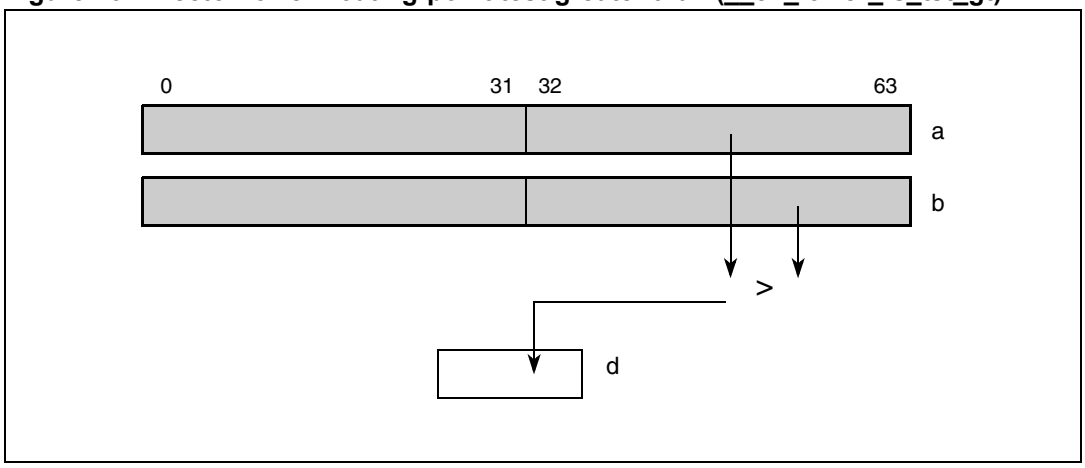


Table 82. __ev_lower_fs_tst_gt (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststgt x,a,b**

__ev_lower_fs_tst_lt

Vector Lower Bits Floating-Point Test Less Than

d = __ev_lower_fs_tst_lt(a,b)

```
if (a32:63 < b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are less than the lower 32 bits of parameter b. This intrinsic differs from __ev_lower_fs_lt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_lower_fs_lt instead.

Figure 77. Vector lower floating-point test less than (__ev_lower_fs_tst_lt)

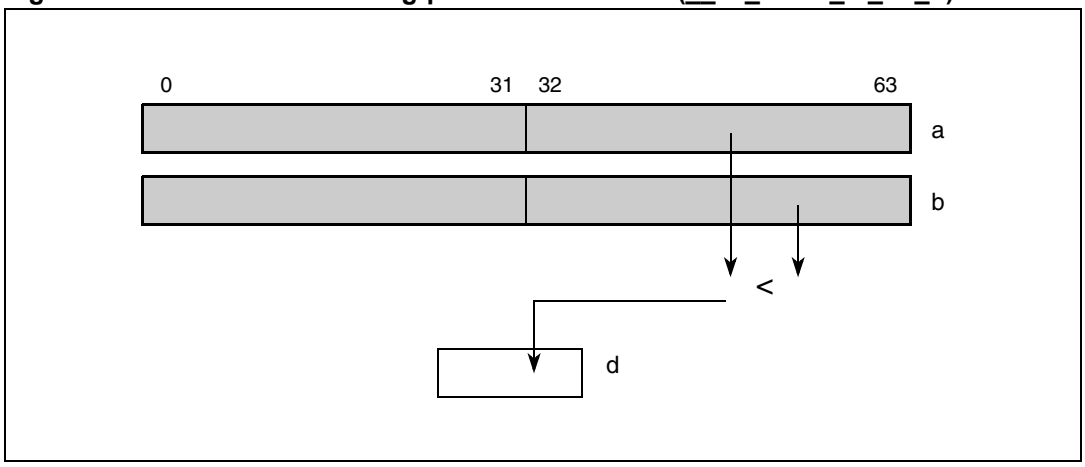


Table 83. __ev_lower_fs_tst_lt (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststlt x,a,b**

__ev_lower__gts

Vector Lower Bits Greater Than Signed

d = __ev_lower_gts(a,b)

```
if (a32:63 >signed b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

Figure 78. Vector lower greater than signed (__ev_lower_gts)

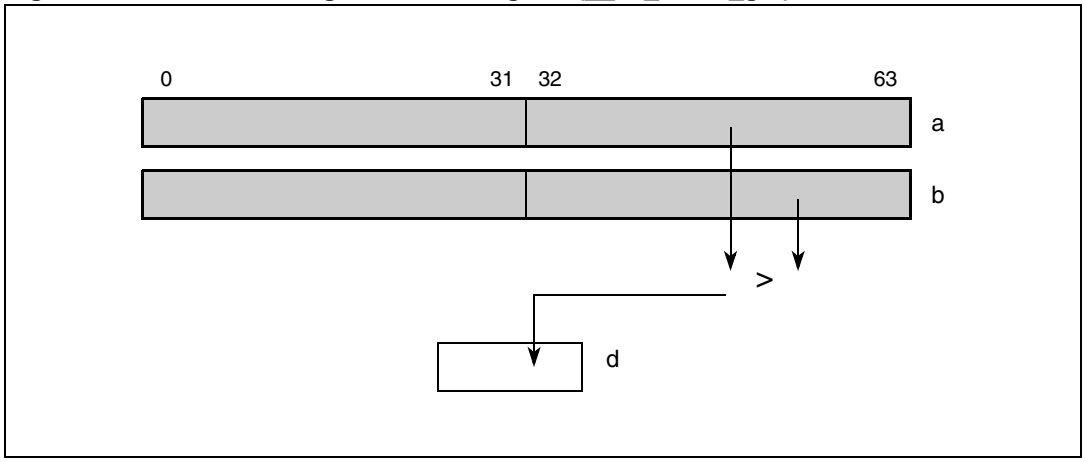


Table 84. __ev_lower__gts (registers altered by).

d	a	b	Maps to
Bool	__ev64_opaque	__ev64_opaque	**evcmpgts x,a,b**

__ev_lower_gtu

Vector Lower Bits Greater Than Unsigned

d = __ev_lower_gtu(a,b)

```
if (a32:63 > unsigned b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

Figure 79. Vector lower greater than unsigned (__ev_lower_gtu)

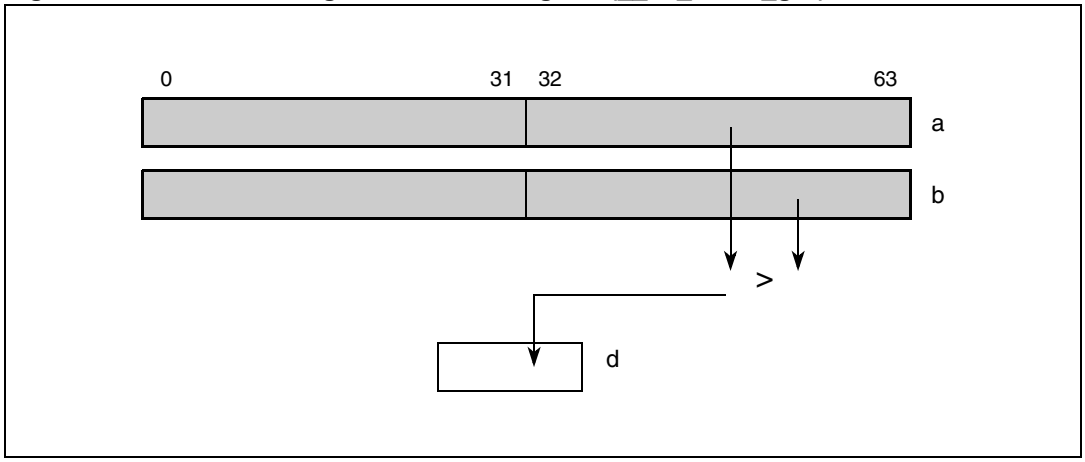


Table 85. __ev_lower_gtu (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpgtu x,a,b**

__ev_lower_lts

Vector Lower Bits Less Than Signed

d = __ev_lower_lts(a,b)

```
if (a32:63 <signed b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

Figure 80. Vector lower less than signed (__ev_lower_lts)

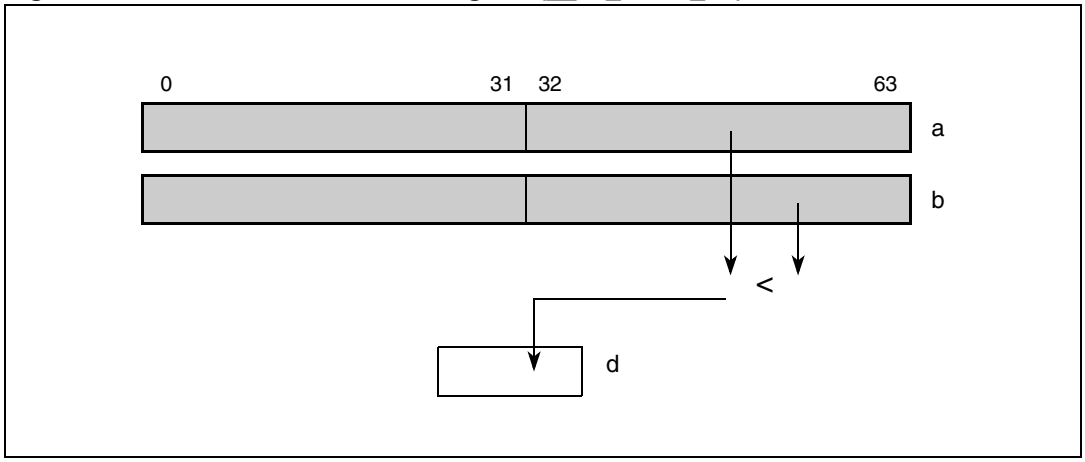


Table 86. __ev_lower_lts (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmplts x,a,b**

__ev_lower_ltu

Vector Lower Bits Less Than Unsigned

d = __ev_lower_ltu(a,b)

```
if (a32:63 <unsigned b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

Figure 81. Vector lower less than unsigned (__ev_lower_ltu)

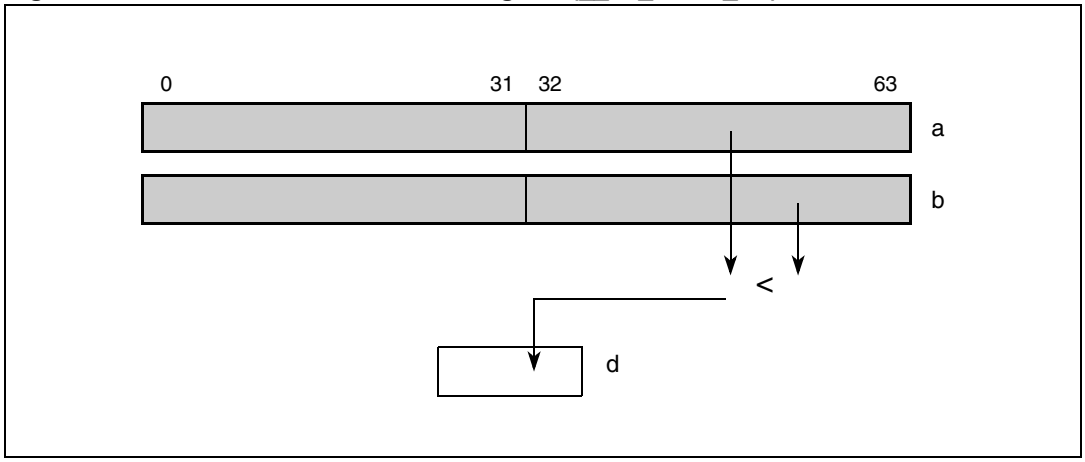


Table 87. __ev_lower_ltu (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpltu x,a,b**

__ev_lwhe

Vector Load Word into Two Half Words Even

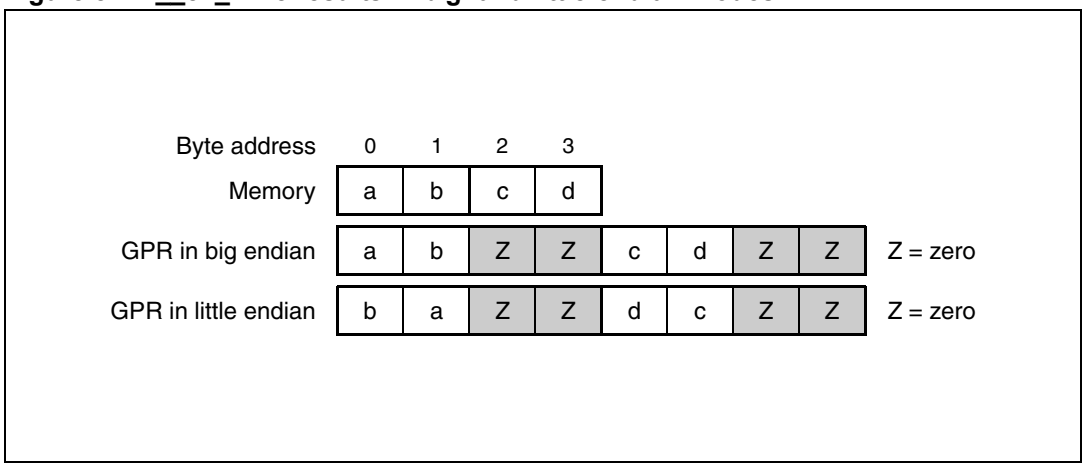
d = __ev_lwhe (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*4)
d0:15 ← MEM(EA,2)
d16:31 ← 0x0000
d32:47 ← MEM(EA+2,2)
d48:63 ← 0x0000
```

The word addressed by EA is loaded from memory and placed in the even half words in each element of parameter d.

Figure 82 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 82. __ev_lwhe results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 88. __ev_lwhe (registers altered by).

d	a	b	Maps to
__ev64_opaque	uint32_t	5-bit unsigned	**evlwhe d,a,b**

__ev_lwhex

Vector Load Word into Two Half Words Even Indexed

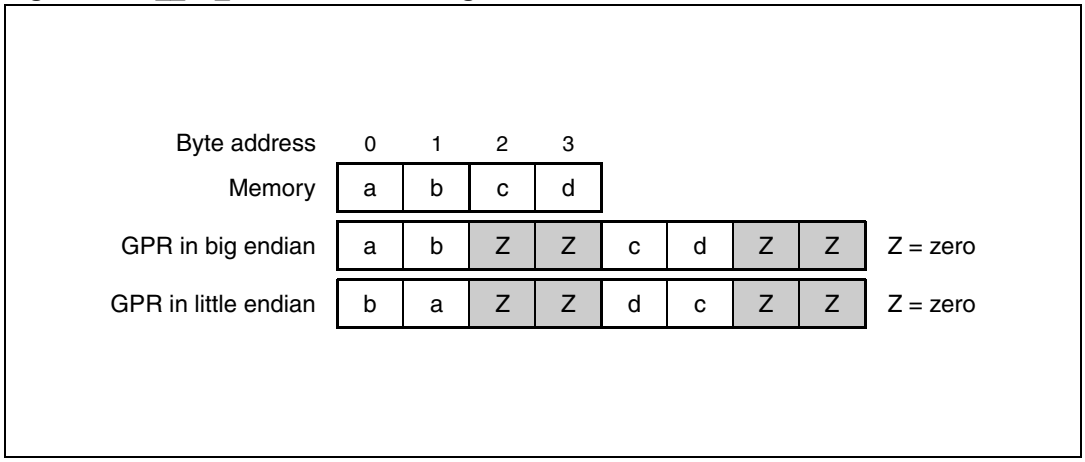
d = __ev_lwhex (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:15 ← MEM(EA,2)
d16:31 ← 0x0000
d32:47 ← MEM(EA+2,2)
d48:63 ← 0x0000
```

The word addressed by EA is loaded from memory and placed in the even half words in each element of parameter d.

Figure 83 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 83. __ev_lwhex results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 89. __ev_lwhex (registers altered by).

d	a	b	Maps to
__ev64_opaque	uint32_t	int32_t	**evlwhex d,a,b**

__ev_lwhos

Vector Load Word into Two Half Words Odd Signed (with sign extension)

d = __ev_lwhos (a,b)

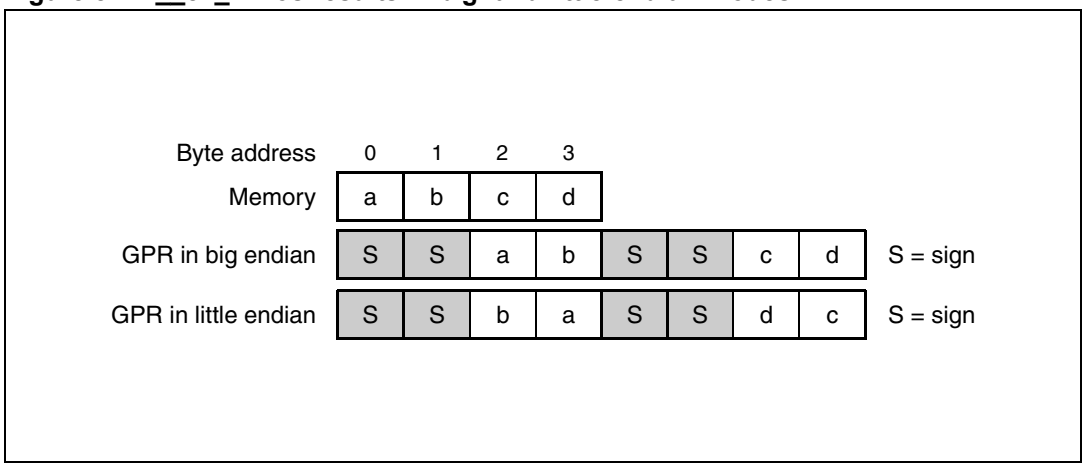
```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*4)
d0:31 ← EXTS(MEM(EA,2))
d32:63 ← EXTS(MEM(EA+2,2))
```

The word addressed by EA is loaded from memory and placed in the odd half words sign extended in each element of parameter d.

Figure 84 shows how bytes are loaded into parameter d as determined by the endian mode.

- In big-endian memory, the msbs of parameters a and c are sign-extended.
- In little-endian memory, the msbs of parameters b and d are sign-extended.

Figure 84. __ev_lwhos results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 90. __ev_lwhos (registers altered by).

d	a	b	Maps to
__ev64_opaque	uint32_t	5-bit unsigned	**evlwhos d,a,b**

__ev_lwhosx

Vector Load Word into Two Half Words Odd Signed Indexed (with sign extension)

d = __ev_lwhosx (a,b)

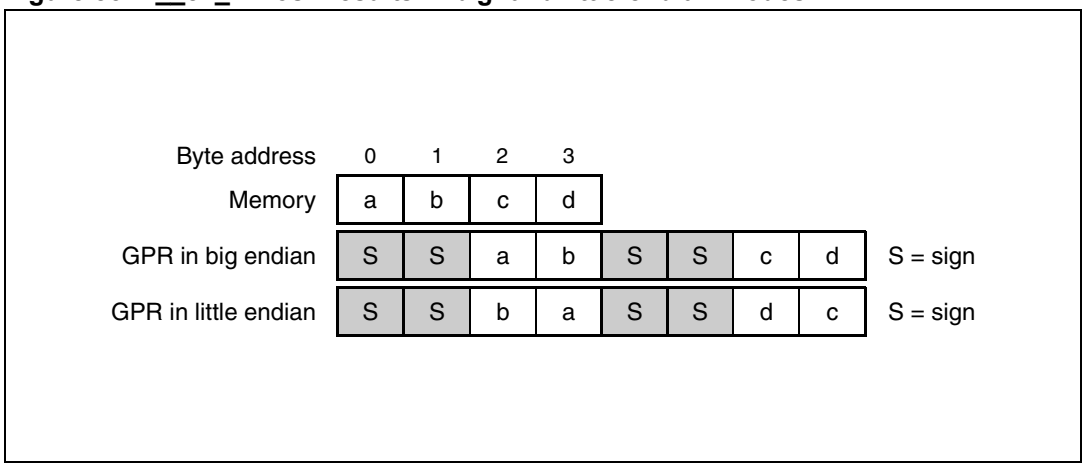
```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:31 ← EXTS(MEM(EA,2))
d32:63 ← EXTS(MEM(EA+2,2))
```

The word addressed by EA is loaded from memory and placed in the odd half words sign extended in each element of parameter d.

Figure 85 shows how bytes are loaded into parameter d as determined by the endian mode.

- In big-endian memory, the msbs of parameters a and c are sign-extended.
- In little-endian memory, the msbs of parameters b and d are sign-extended.

Figure 85. __ev_lwhosx results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 91. __ev_lwhosx (registers altered by).

d	a	b	Maps to
__ev64_opaque	uint32_t	int32_t	**evlwhosx d,a,b**

__ev_lwhou

Vector Load Word into Two Half Words Odd Unsigned (zero-extended)

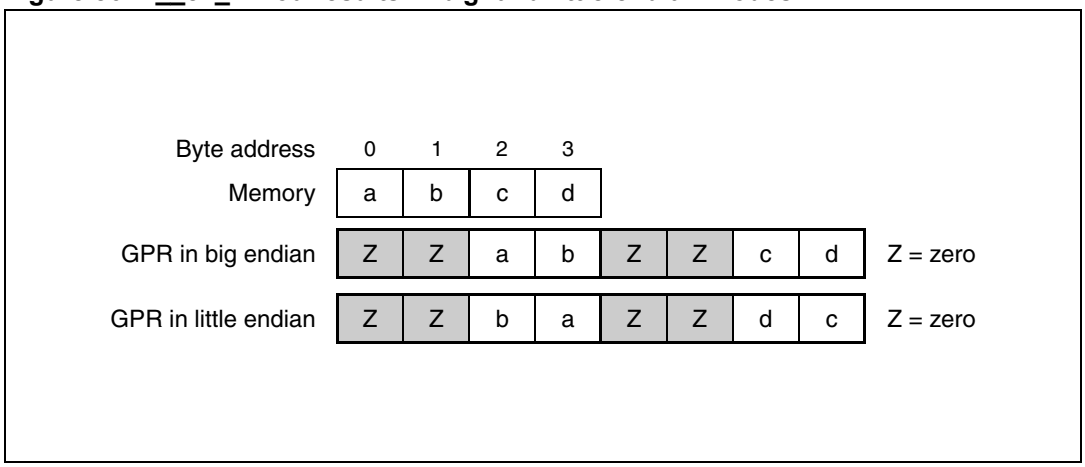
d = __ev_lwhou (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*4)
d0:15 ← 0x0000
d16:31 ← MEM(EA,2)
d32:47 ← 0x0000
d48:63 ← MEM(EA+2,2)
```

The word addressed by EA is loaded from memory and placed in the odd half words zero extended in each element of parameter d.

Figure 86 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 86. __ev_lwhou results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 92. __ev_lwhou (registers altered by).

d	a	b	Maps to
__ev64_opaque	uint32_t	5-bit unsigned	**evlwhou d,a,b**

__ev_lwhoux

Vector Load Word into Two Half Words Odd Unsigned Indexed (zero-extended)

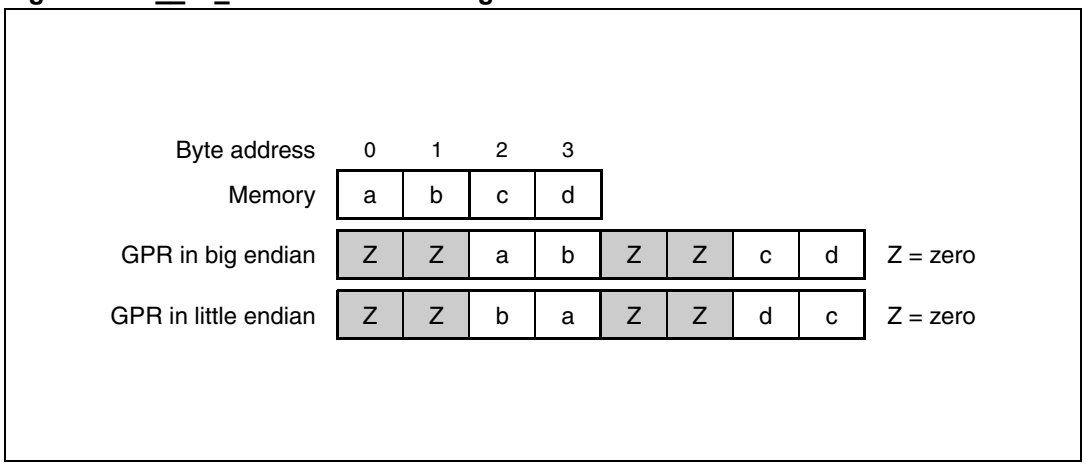
d = __ev_lwhoux (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:15 ← 0x0000
d16:31 ← MEM(EA,2)
d32:47 ← 0x0000
d48:63 ← MEM(EA+2,2)
```

The word addressed by EA is loaded from memory and placed in the odd half words zero extended in each element of parameter d.

Figure 87 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 87. __ev_lwhoux results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 93. __ev_lwhoux (registers altered by).

d	a	b	Maps to
__ev64_opaque	uint32_t	int32_t	**evlwhoux d,a,b**

__ev_lwhsplat

Vector Load Word into Two Half Words and Splat

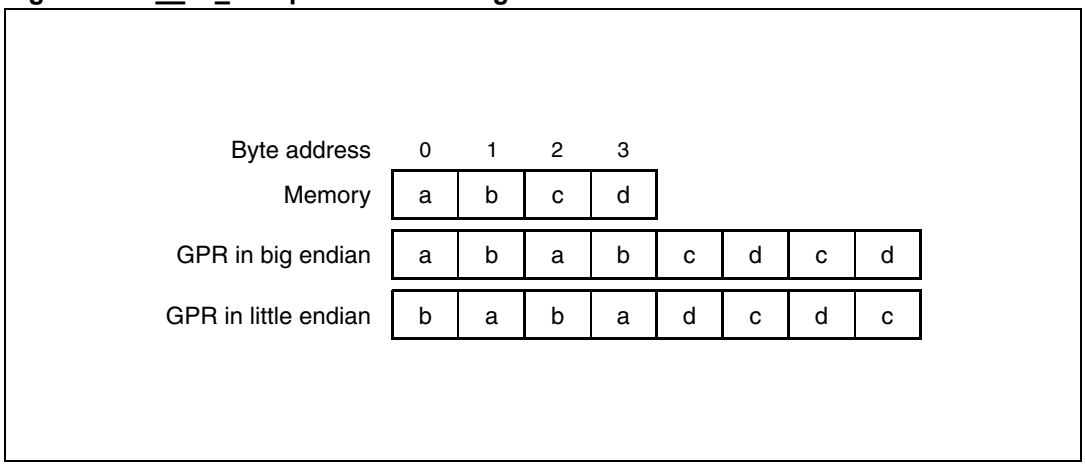
d = __ev_lwhsplat (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*4)
d0:15 ← MEM(EA,2)
d16:31 ← MEM(EA,2)
d32:47 ← MEM(EA+2,2)
d48:63 ← MEM(EA+2,2)
```

The word addressed by EA is loaded from memory and placed in both the even and odd half words in each element of parameter d.

Figure 88 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 88. __ev_lwhsplat results in big- and little-endian modes



During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 94. __ev_lwhsplat (registers altered by).

d	a	b	Maps to
__ev64_opaque	uint32_t	5-bit unsigned	**evlwhsplat d,a,b**

__ev_lwhsplatx

Vector Load Word into Two Half Words and Splat-Indexed

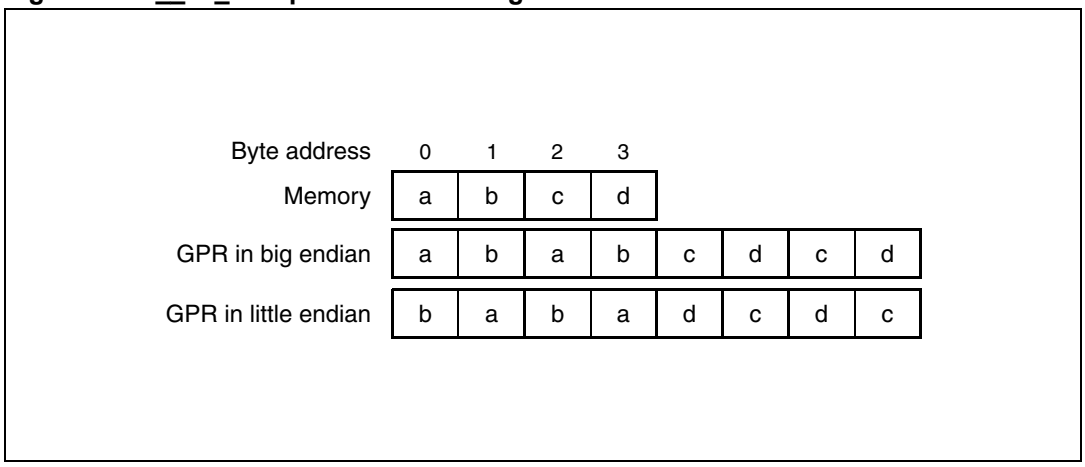
d = __ev_lwhsplatx (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:15 ← MEM(EA,2)
d16:31 ← MEM(EA,2)
d32:47 ← MEM(EA+2,2)
d48:63 ← MEM(EA+2,2)
```

The word addressed by EA is loaded from memory and placed in both the even and odd half words in each element of parameter d.

Figure 89 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 89. __ev_lwhsplatx results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 95. __ev_lwhsplatx (registers altered by).

d	a	b	Maps to
__ev64_opaque	uint32_t	int32_t	**evlwhsplatx d,a,b**

__ev_lwwsplat

Vector Load Word into Word and Splat

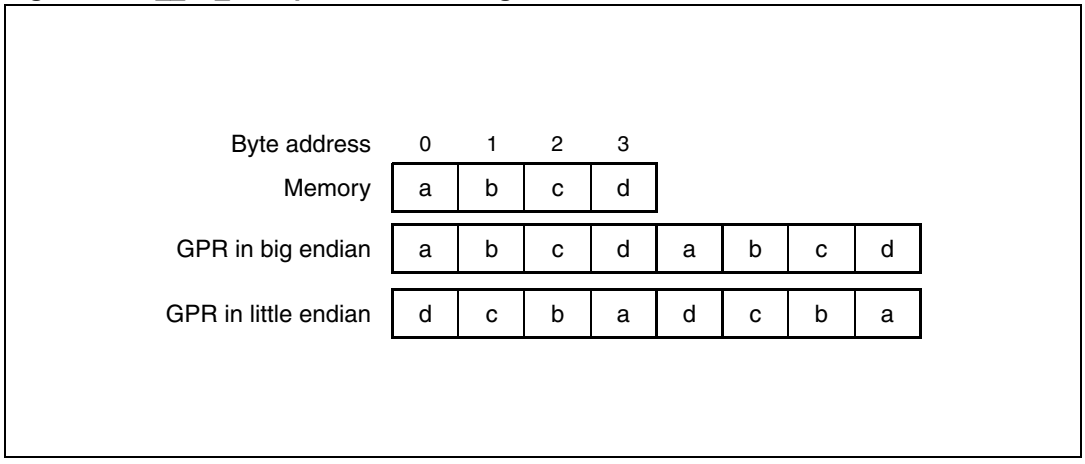
d = __ev_lwwsplat (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*4)
d0:31 ← MEM(EA,4)
d32:63 ← MEM(EA,4)
```

The word addressed by EA is loaded from memory and placed in both elements of parameter d.

Figure 90 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 90. __ev_lwwsplat results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 96. __ev_lwwsplat (registers altered by).

d	a	b	Maps to
__ev64_opaque	uint32_t	5-bit unsigned	**evlwwsplat d,a,b**

__ev_lwwsplatx

Vector Load Word into Word and Splat-Indexed

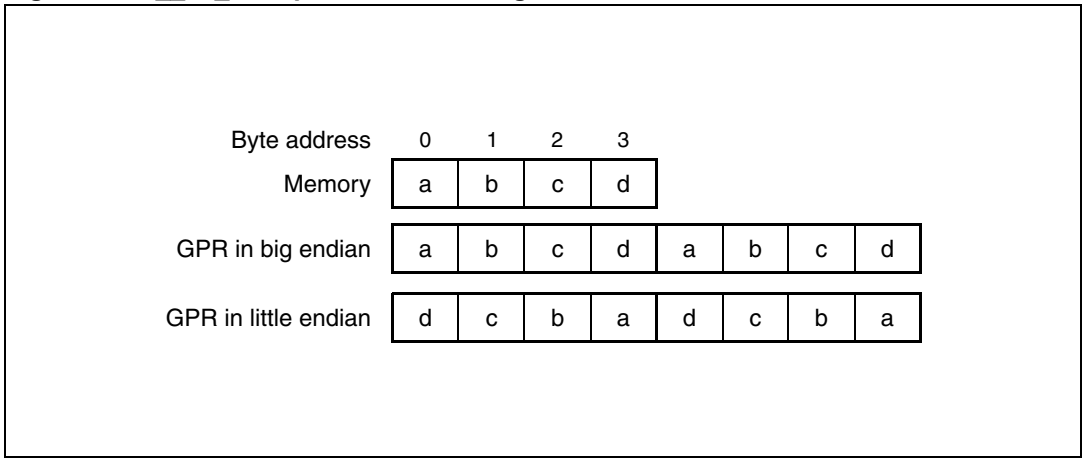
d = __ev_lwwsplatx (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:31 ← MEM(EA,4)
d32:63 ← MEM(EA,4)
```

The word addressed by EA is loaded from memory and placed in both elements of parameter d.

Figure 91 shows how bytes are loaded into parameter d as determined by the endian mode.

Figure 91. __ev_lwwsplatx results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 97. __ev_lwwsplatx (registers altered by).

d	a	b	Maps to
__ev64_opaque	uint32_t	int32_t	**evlwwsplatx d,a,b**

__ev_mergehi

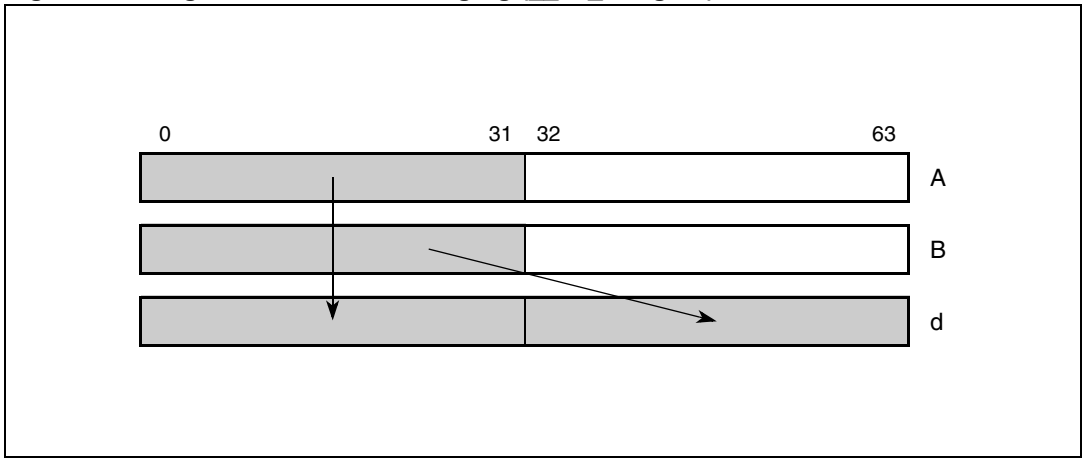
Vector Merge High

d = __ev_mergehi (a,b)

$d_{0:31} \leftarrow a_{0:31}$
$d_{32:63} \leftarrow b_{0:31}$

The high-order elements of parameters a and b are merged and placed into parameter d, are shown below:

Figure 92. High-order element merging (__ev_mergehi)



Note: To perform a vector splat high, specify the same register in parameters a and b.

Table 98. __ev_mergehi (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmergehi d,a,b**

__ev_mergehilo

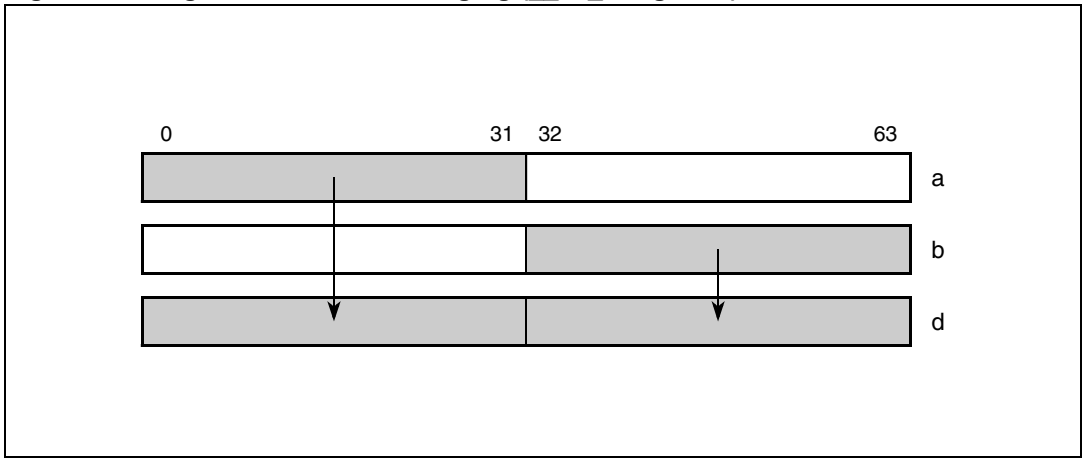
Vector Merge High/Low

d = __ev_mergehilo (a,b)

$d_{0:31} \leftarrow a_{0:31}$
$d_{32:63} \leftarrow b_{32:63}$

The high-order element of parameter a and the low-order element of parameter b are merged and placed into parameter d, as shown below:

Figure 93. High-order element merging (__ev_mergehilo)



*Note: Application note: With appropriate specification of parameter a and b, **evmergehi**, **evmergelo**, **evmergehilo**, and **evmergelohi** provide a full 32-bit permute of two source parameters.*

Table 99. __ev_mergehilo (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmergehilo d,a,b**

__ev_mergelo

Vector Merge Low

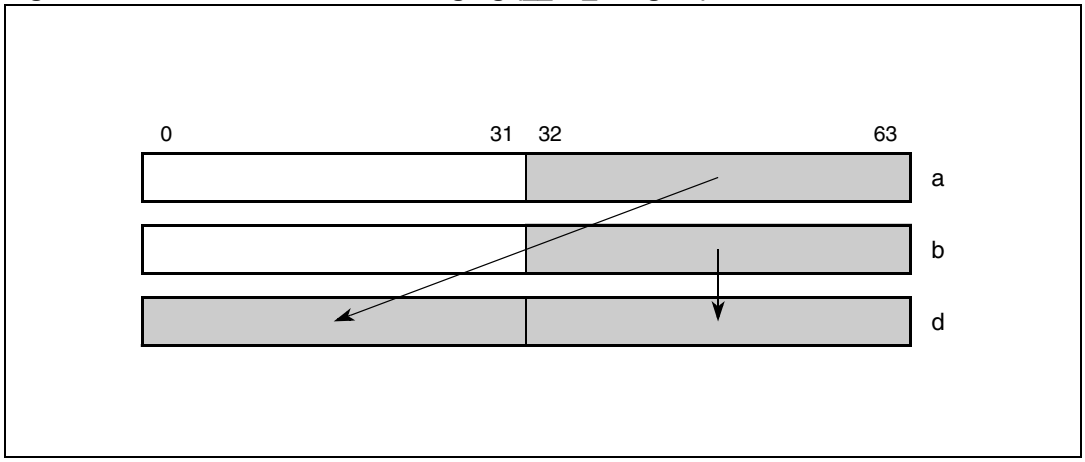
d = __ev_mergelo (a,b)

$d_{0:31} \leftarrow a_{32:63}$

$d_{32:63} \leftarrow b_{32:63}$

The low-order elements of parameters a and b are merged and placed in parameter d, as shown below:

Figure 94. Low-order element merging (__ev_mergelo)



Note: To perform a vector splat low, specify the same register in parameters a and b.

Table 100. __ev_mergelo (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmergelo d,a,b**

__ev_mergelohi

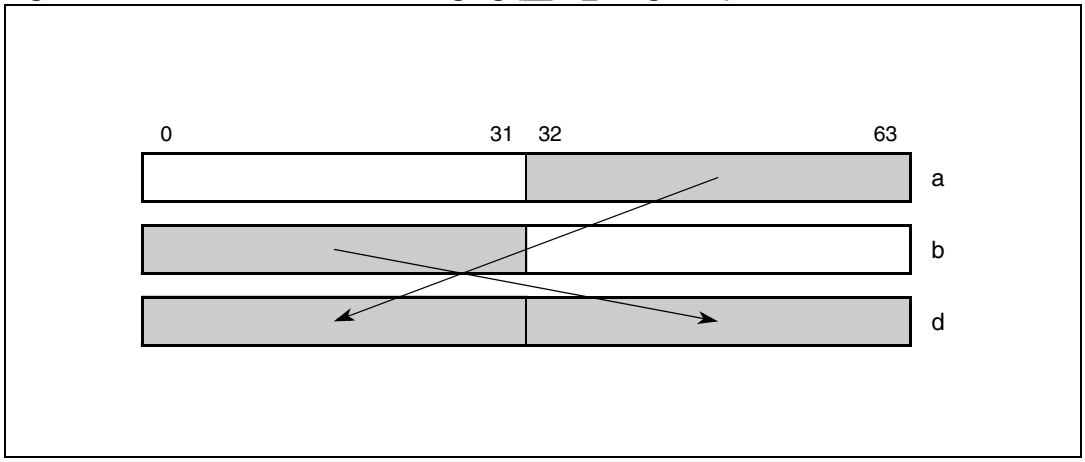
Vector Merge Low/High

d = __ev_mergelohi (a,b)

$d_{0:31} \leftarrow a_{32:63}$
$d_{32:63} \leftarrow b_{0:31}$

The low-order element of parameter a and the high-order element of parameter b are merged and placed into parameter d, as shown below:

Figure 95. Low-order element merging (__ev_mergelohi)



Note: To perform a vector swap, specify the same register in parameters a and b.

Table 101. __ev_mergelohi (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmergelohi d,a,b**

__ev_mhegsmfaa

Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate

d = __ev_mhegsmfaa (a,b)

$temp_{0:31} \leftarrow a_{32:47} \times_{sf} b_{32:47}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low even-numbered, half-word signed fractional elements in parameters a and b are multiplied. The product is added to the contents of the 64-bit accumulator, and the result is placed into parameter d and the accumulator.

Note: This sum is a modulo sum. Neither overflow check nor saturation is performed. Any overflow of the 64-bit sum is not recorded into the SPEFSCR.

Figure 96. __ev_mhegsmfaa (even form)

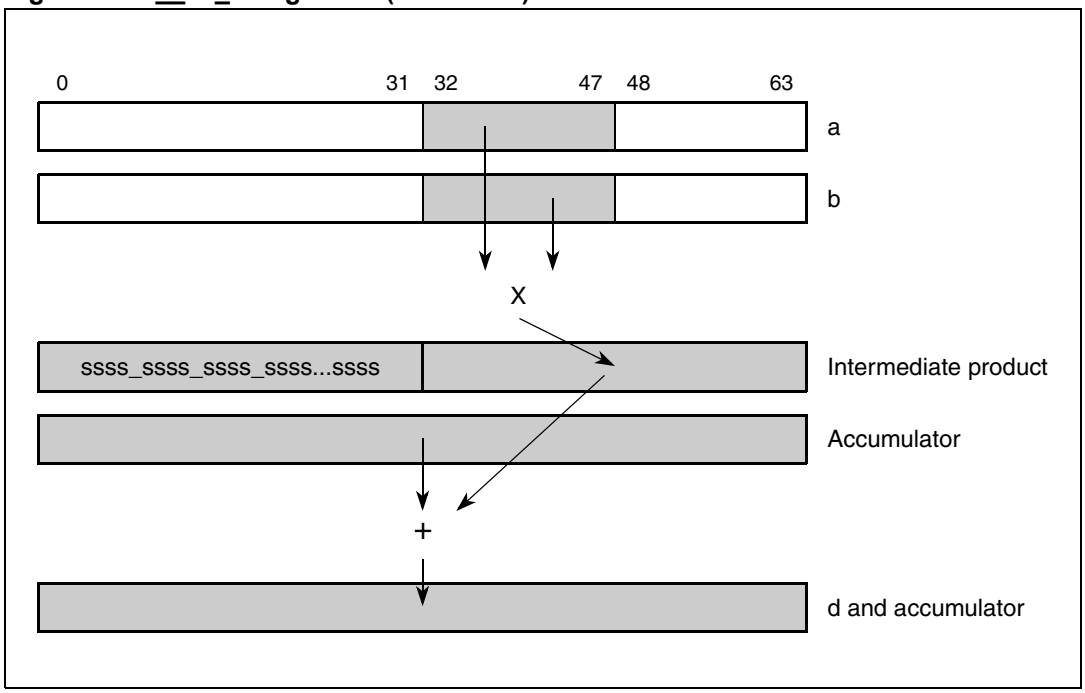


Table 102. __ev_mhegsmfaa (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhegsmfaa d,a,b**

__ev_mhegsmfan

Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative

d = __ev_mhegsmfan (a,b)

$temp_{0:31} \leftarrow a_{32:47} \times_{sf} b_{32:47}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low even-numbered, half-word signed fractional elements in parameters a and b are multiplied. The product is subtracted from the contents of the 64-bit accumulator, and the result is placed into parameter d and the accumulator.

Note: This difference is a modulo difference. Neither overflow check nor saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

Figure 97. __ev_mhegsmfan (even form)

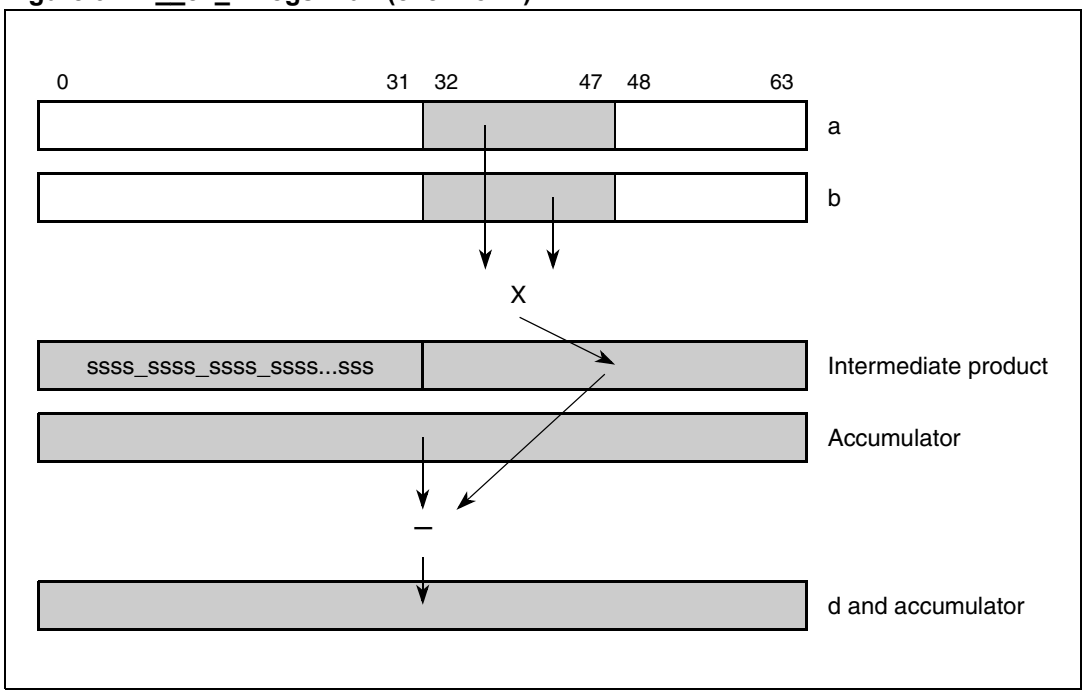


Table 103. __ev_mhegsmfan (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhegsmfan d,a,b**

__ev_mhegsmiaa

Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate

d = __ev_mhegsmiaa (a,b)

$temp_{0:31} \leftarrow a_{32:47} \times_{si} b_{32:47}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$

```
// update accumulator
```
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low even-numbered half-word signed integer elements in parameters a and b are multiplied. The intermediate product is sign-extended and added to the contents of the 64-bit accumulator, and the resulting sum is placed into parameter d and the accumulator.

Note: This sum is a modulo sum. Neither overflow check nor saturation is performed. Any overflow of the 64-bit sum is not recorded into the SPEFSCR.

Figure 98. __ev_mhegsmiaa (even form)

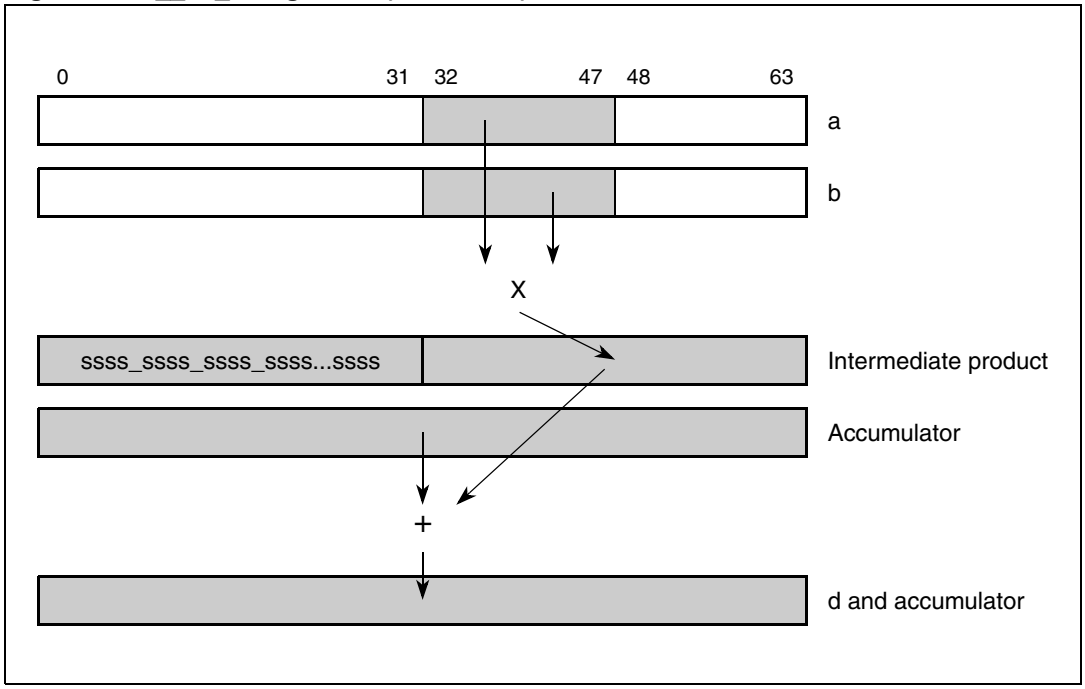


Table 104. __ev_mhegsmiaa (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhegsmiaa d,a,b**

__ev_mhegsmian

Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative

d = __ev_mhegsmian (a,b)

$temp_{0:31} \leftarrow a_{32:47} \times_{si} b_{32:47}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$The corresponding low even-numbered half-word signed integer elements in parameters a and b are multiplied. The intermediate product is sign-extended and subtracted from the contents of the 64-bit accumulator, and the result is placed into parameter d and into the accumulator.

Note: This difference is a modulo difference. Neither overflow check nor saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

Figure 99. __ev_mhegsmian (even form)

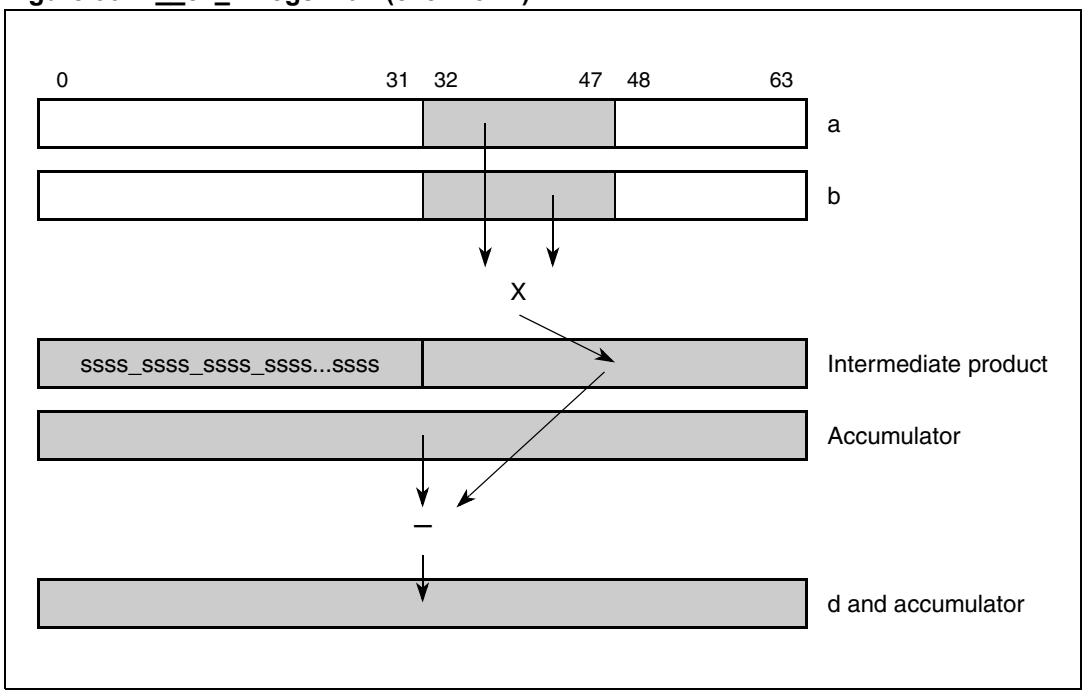


Table 105. __ev_mhegsmian (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhegsmian d,a,b**

__ev_mhegumfaa

Vector Multiply Half Words, Even, Guarded, Unsigned, Modulo, Fractional and Accumulate

d = __ev_mhegumfaa (a,b)

$temp_{0:31} \leftarrow a_{32:47} \times_{ui} b_{32:47}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low even-numbered elements in parameters a and b are multiplied. The intermediate product is zero-extended and added to the contents of the 64-bit accumulator. The resulting sum is placed into parameter d and into the accumulator.

Note: This sum is a modulo sum. Neither overflow check nor saturation is performed. Overflow of the 64-bit sum is not recorded into the SPEFSCR.

Figure 100. __ev_mhegumfaa (even form)

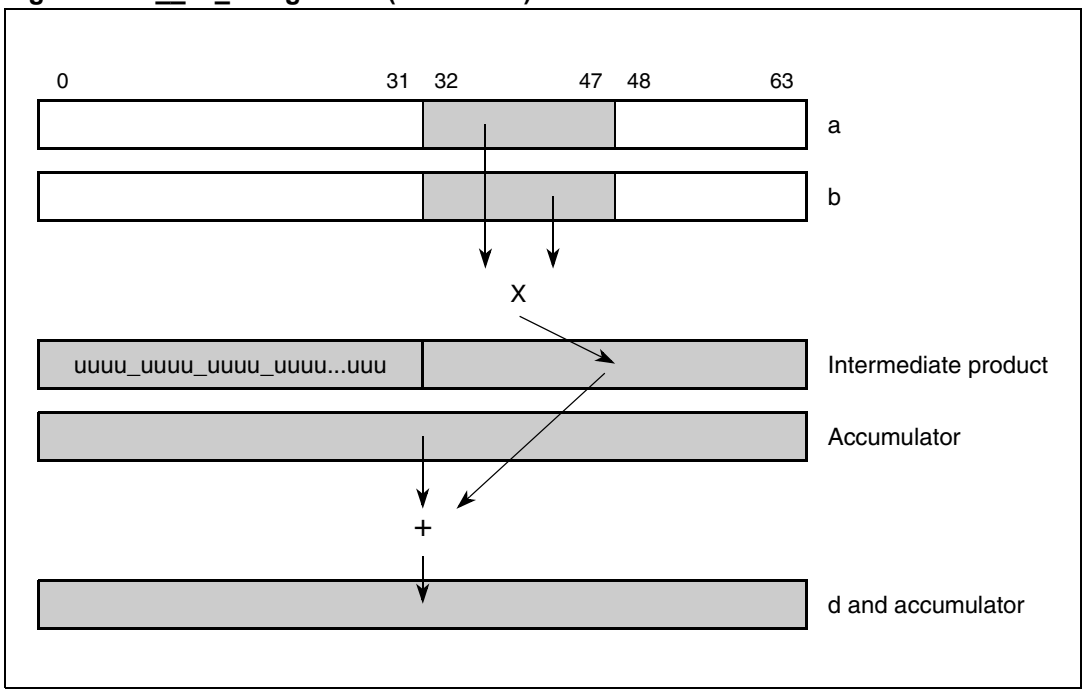


Table 106. __ev_mhegumfaa (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhegumiaa d,a,b**

__ev_mhegumiaa

Vector Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate

d = __ev_mhegumiaa (a,b)

$temp_{0:31} \leftarrow a_{32:47} \times_{ui} b_{32:47}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low even-numbered half-word unsigned integer elements in parameters a and b are multiplied. The intermediate product is zero-extended and added to the contents of the 64-bit accumulator. The resulting sum is placed into parameter d and into the accumulator.

This sum is a modulo sum. Neither overflow check nor saturation is performed. Any overflow of the 64-bit sum is not recorded into the SPEFSCR.

Figure 101. __ev_mhegumiaa (even form)

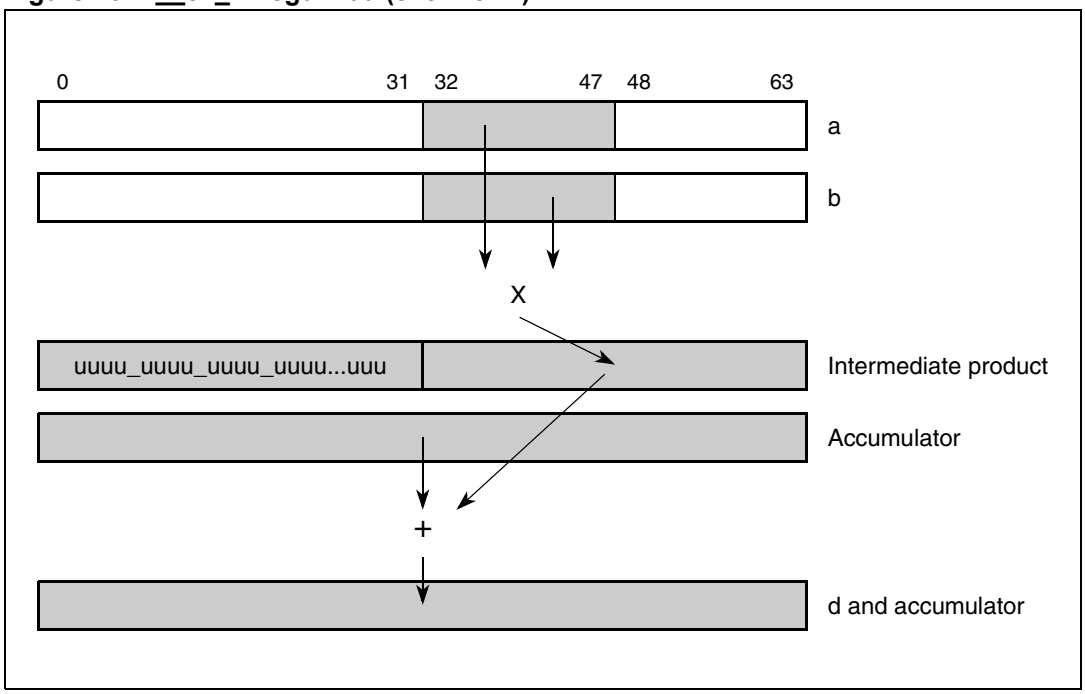


Table 107. __ev_mhegumiaa (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhegumiaa d,a,b**

__ev_mhegumfan

Vector Multiply Half Words, Even, Guarded, Unsigned, Modulo, Fractional and Accumulate Negative

d = __ev_mhegumfan (a,b)

$temp_{0:31} \leftarrow a_{32:47} \times_{ui} b_{32:47}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low even-numbered elements in parameters a and b are multiplied. The intermediate product is zero-extended and subtracted from the contents of the 64-bit accumulator. The result is placed into parameter d and into the accumulator.

Note: This difference is a modulo difference. Neither overflow check nor saturation is performed. Overflow of the 64-bit difference is not recorded into the SPEFSCR.

Figure 102. __ev_mhegumfan (even form)

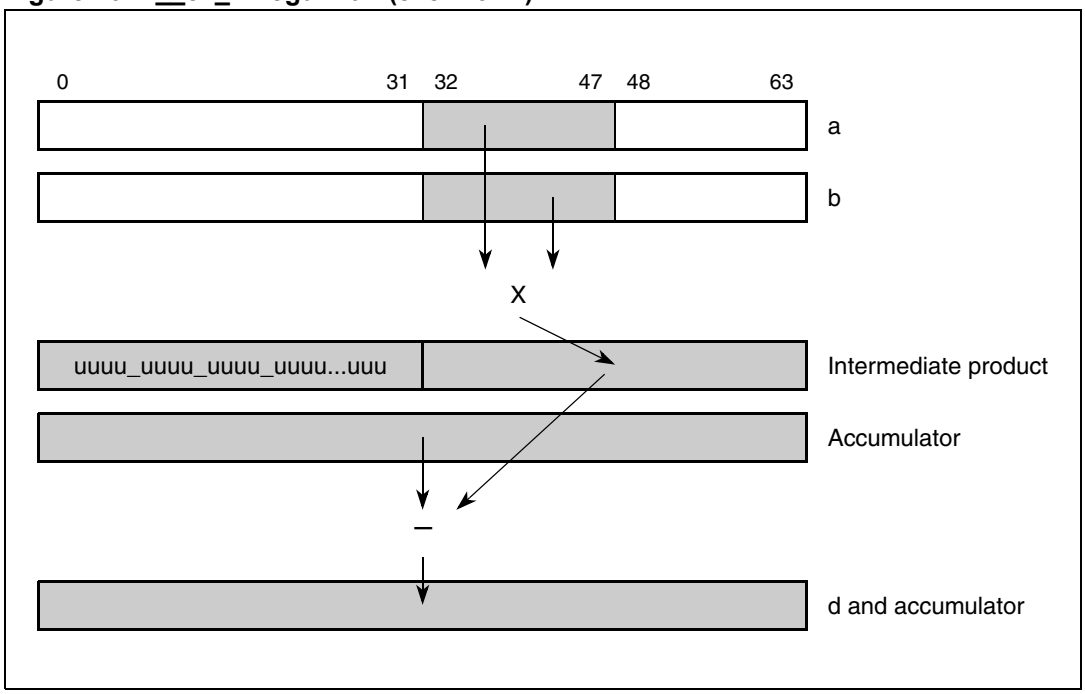


Table 108. __ev_mhegumfan (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhegumian d,a,b**

__ev_mhegumian

Vector Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative

d = __ev_mhegumian (a,b)

$temp_{0:31} \leftarrow a_{32:47} \times_{ui} b_{32:47}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low even-numbered unsigned integer elements in parameter a and b are multiplied. The intermediate product is zero-extended and subtracted from the contents of the 64-bit accumulator. The result is placed into parameter d and into the accumulator.

Note: This difference is a modulo difference. Neither overflow check nor saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

Figure 103. __ev_mhegumian (even form)

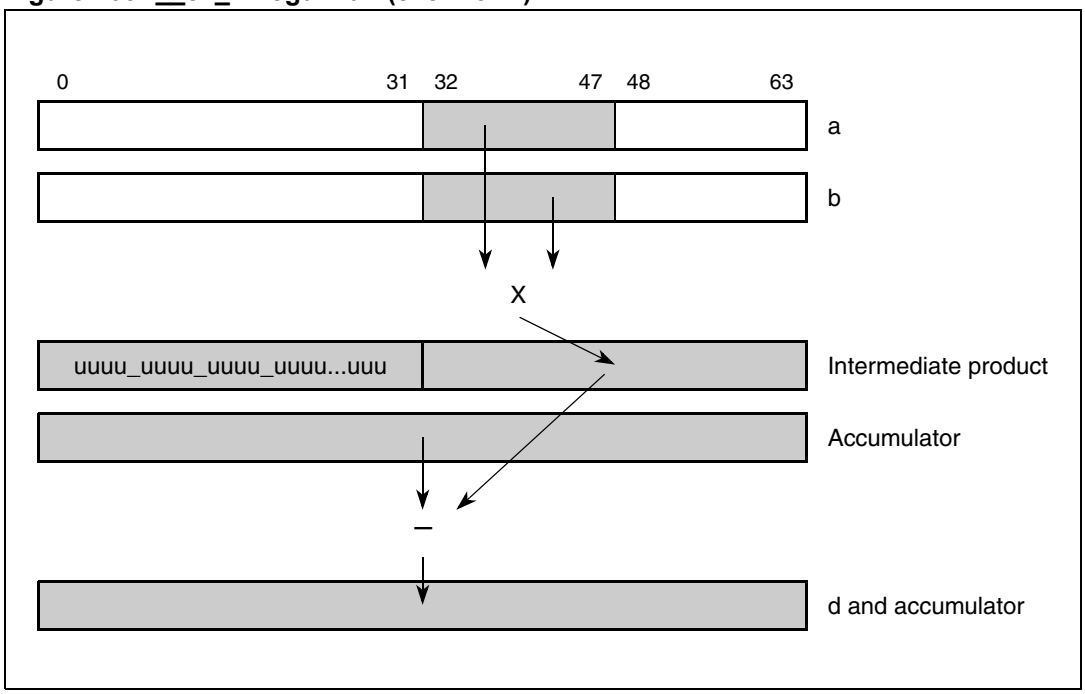


Table 109. __ev_mhegumian (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhegumian d,a,b**

__ev_mhesmf

Vector Multiply Half Words, Even, Signed, Modulo, Fractional (to Accumulator)

d = __ev_mhesmf (a,b) (A = 0)

d = __ev_mhesmfa (a,b) (A = 1)

// high
$d_{0:31} \leftarrow (a_{0:15} \times_{sf} b_{0:15})$

// low
$d_{32:63} \leftarrow (a_{32:47} \times_{sf} b_{32:47})$

// update accumulator
if A = 1 then $ACC_{0:63} \leftarrow d_{0:63}$

The corresponding even-numbered half-word signed fractional elements in parameters a and b are multiplied, and the 32 bits of each product are placed into the corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

Figure 104. Even multiply of two signed modulo fractional elements (to accumulator) (__ev_mhesmf)

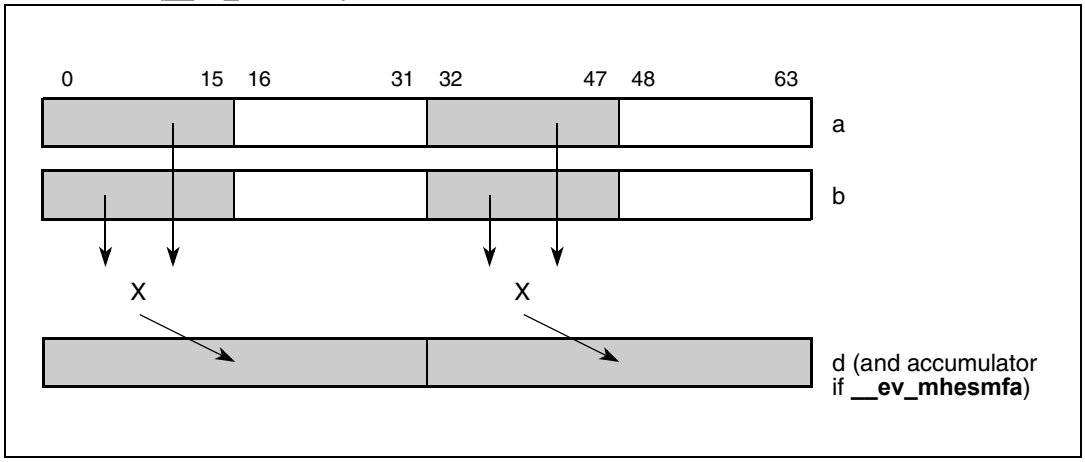


Table 110. __ev_mhesmf (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhesmf d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhesmfa d,a,b**

__ev_mhesmfaaw

Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate into Words

d = __ev_mhesmfaaw (a,b)

```
// high
temp0:31 ← (a0:15 ×sf b0:15)
d0:31 ← ACC0:31 + temp0:31

// low
temp0:31 ← (a32:47 ×sf b32:47)
d32:63 ← ACC32:63 + temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding even-numbered half-word signed fractional elements in parameters a and b are multiplied. The 32 bits of each intermediate product are added to the contents of the accumulator words to form intermediate sums, which are placed into the corresponding parameter d words and into the accumulator.

Other registers altered: ACC

Figure 105. Even form of vector half-word multiply (__ev_mhesmfaaw)

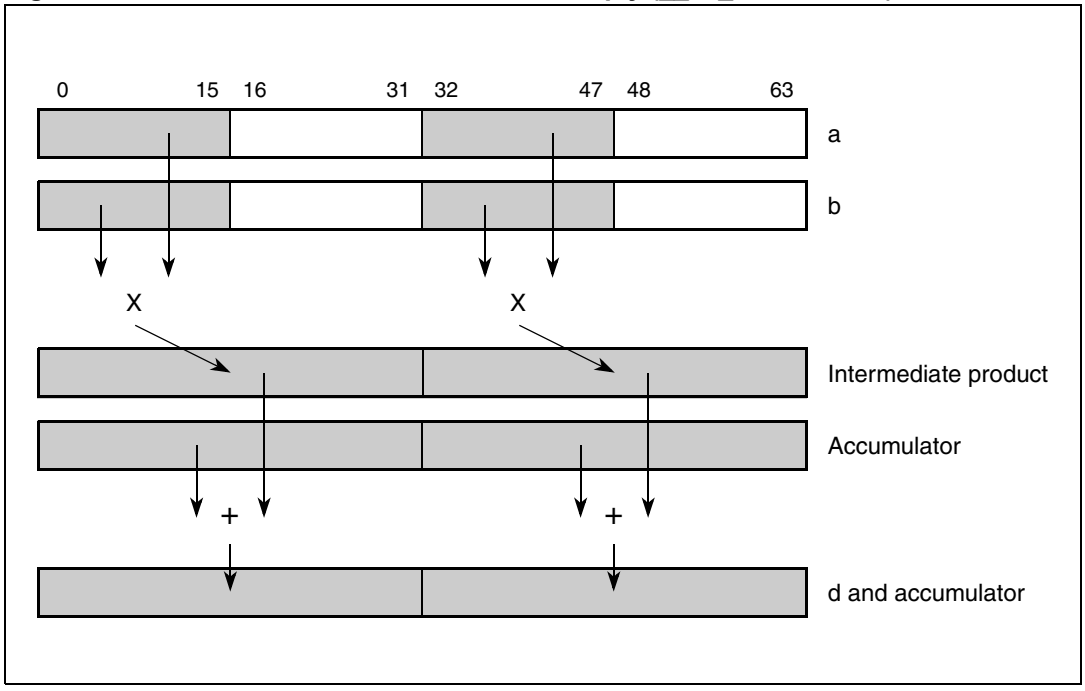


Table 111. __ev_mhesmfaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhesmfaaw d,a,b**

__ev_mhesmfanw

Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate Negative into Words

d = __ev_mhesmfanw (a,b)

```
// high
temp0:31 ← a0:15 ×sf b0:15
d0:31 ← ACC0:31 - temp0:31

// low
temp0:31 ← a32:47 ×sf b32:47
d32:63← ACC32:63 - temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding even-numbered half-word signed fractional elements in parameters a and b are multiplied. The 32-bit intermediate products are subtracted from the contents of the accumulator words to form intermediate differences, which are placed into the corresponding parameter d words and into the accumulator.

Other registers altered: ACC

Figure 106. Even form of vector half-word multiply (__ev_mhesmfanw)

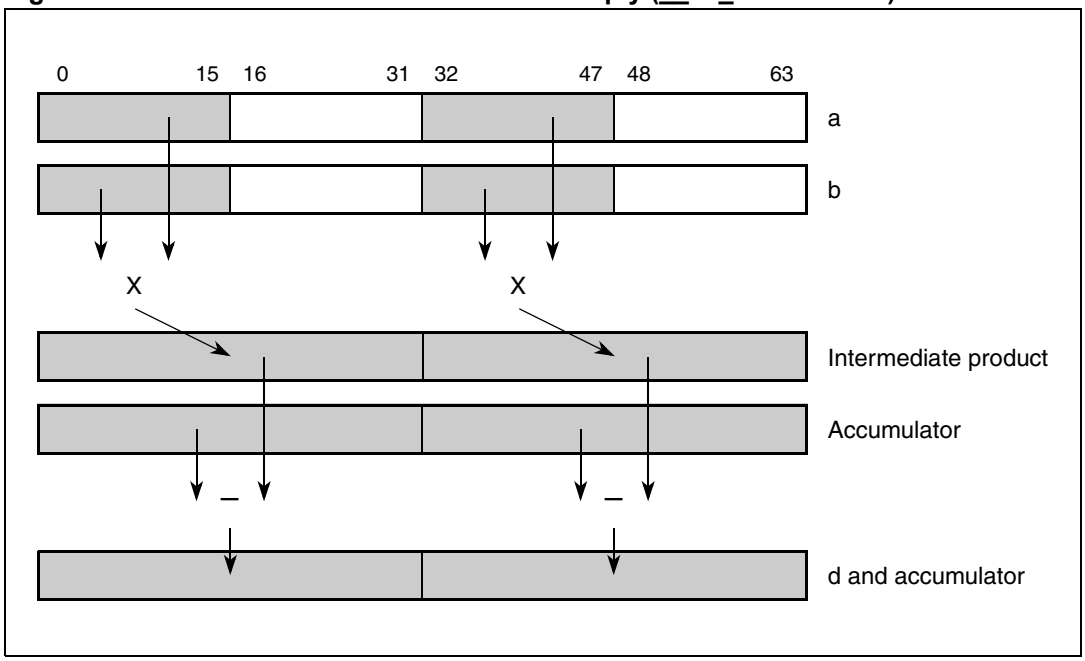


Table 112. __ev_mhesmfanw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhesmfanw d,a,b**

__ev_mhesmi

Vector Multiply Half Words, Even, Signed, Modulo, Integer (to Accumulator)

d = __ev_mhesmi (a,b) (A = 0)

d = __ev_mhesmia (a,b) (A = 1)

```
// high
d0:31 ← a0:15 ×si b0:15

// low
d32:63 ← a32:47 ×si b32:47

// update accumulator
if A = 1, then ACC0:63 ← d0:63
```

The corresponding even-numbered half-word signed integer elements in parameters a and b are multiplied. The two 32-bit products are placed into the corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

Figure 107. Even form for vector multiply (to accumulator) (__ev_mhesmi)

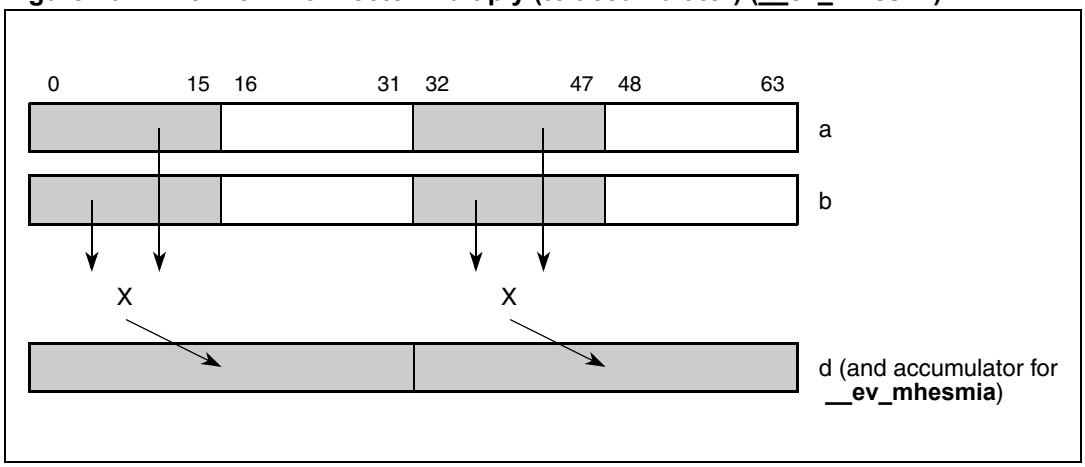


Table 113. __ev_mhesmi (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhesmi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhesmia d,a,b**

__ev_mhesmiaaw

Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate into Words

d = __ev_mhesmiaaw (a,b)

```
// high
temp0:31 ← a0:15 ×si b0:15
d0:31 ← ACC0:31 + temp0:31

// low
temp0:31 ← a32:47 ×si b32:47
d32:63 ← ACC32:63 + temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding even-numbered half-word signed integer elements in parameters a and b are multiplied. Each intermediate 32-bit product is added to the contents of the accumulator words to form intermediate sums, which are placed into the corresponding parameter d words and into the accumulator.

Other registers altered: ACC

Figure 108. Even form of vector half-word multiply (__ev_mhesmiaaw)

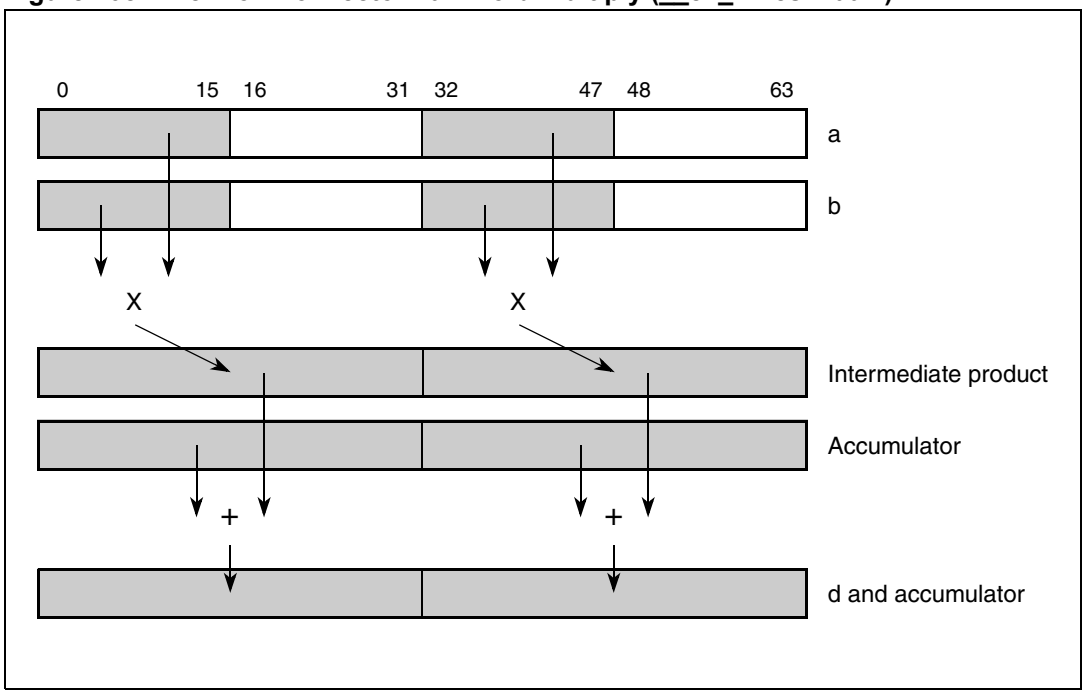


Table 114. __ev_mhesmiaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhesmiaaw d,a,b**

__ev_mhesmianw

Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate Negative into Words

d = __ev_mhesmianw (a,b)

```
// high
temp0 0:31 ←a 0:15 ×si b 0:15
d 0:31 ← ACC 0:31 - temp0 0:31
// low
temp1 0:31 ← a 32:47 ×si b 32:47
d 32:63 ← ACC 32:63 - temp1 0:31
// update accumulator
ACC 0:63 ← d 0:63
```

For each word element in the accumulator, the corresponding even-numbered half-word signed integer elements in parameters a and b are multiplied. Each intermediate 32-bit product is subtracted from the contents of the accumulator words to form intermediate differences, which are placed into the corresponding parameter d words and into the accumulator.

Other registers altered: ACC

Figure 109. Even form of vector half-word multiply (__ev_mhesmianw)

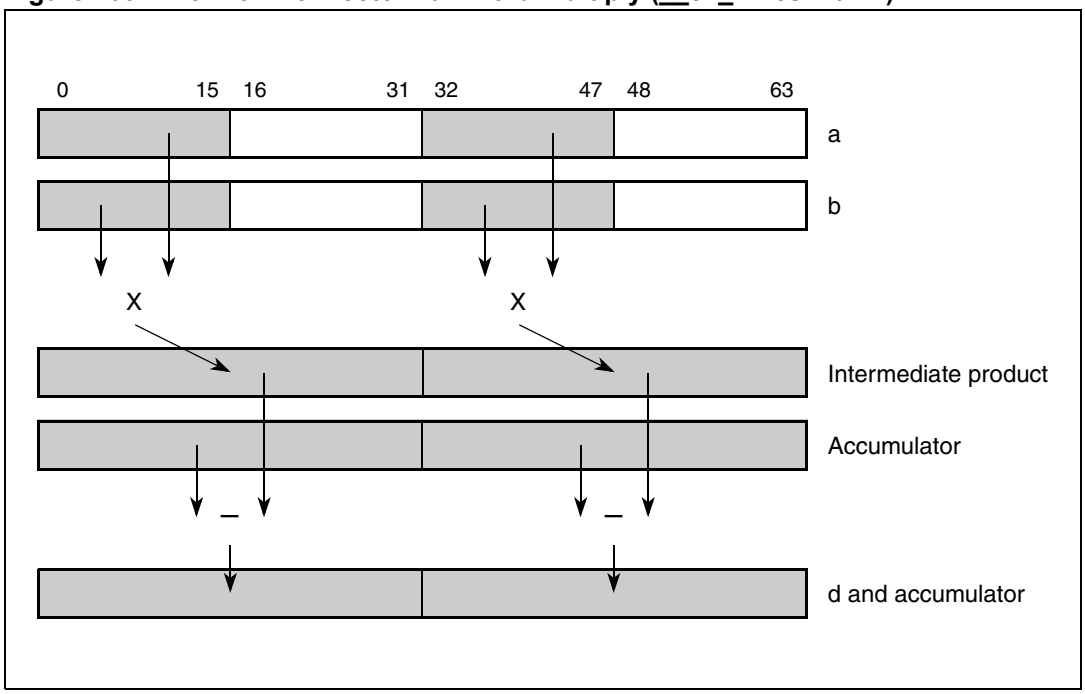


Table 115. __ev_mhesmianw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhesmianw d,a,b**

__ev_mhessf

Vector Multiply Half Words, Even, Signed, Saturate, Fractional (to Accumulator)

d = __ev_mhessf (a,b) (A = 0)

d = __ev_mhessfa (a,b) (A = 1)

```
// high
temp0:31 ← a0:15 ×sf b0:15
if (a0:15 = 0x8000) & (b0:15 = 0x8000) then
   d0:31 ← 0x7FFF_FFFF //saturate
   movh ← 1
else
   d0:31 ← temp0:31
   movh ← 0

// low
temp0:31 ← a32:47 ×sf b32:47
if (a32:47 = 0x8000) & (b32:47 = 0x8000) then
   d32:63 ← 0x7FFF_FFFF //saturate
   movl ← 1
else
   d32:63 ← temp0:31
   movl ← 0

// update accumulator
if A = 1 then ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding even-numbered half-word signed fractional elements in parameters a and b are multiplied. The 32 bits of each product are placed into the corresponding words of parameter d. If both inputs are -1.0, the result saturates to the largest positive signed fraction and the overflow and summary overflow bits are recorded in the SPEFSCR.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: SPEFSCR
ACC (if A = 1)

Figure 110. Even multiply of two signed saturate fractional elements (to accumulator) (__ev_mhessf)

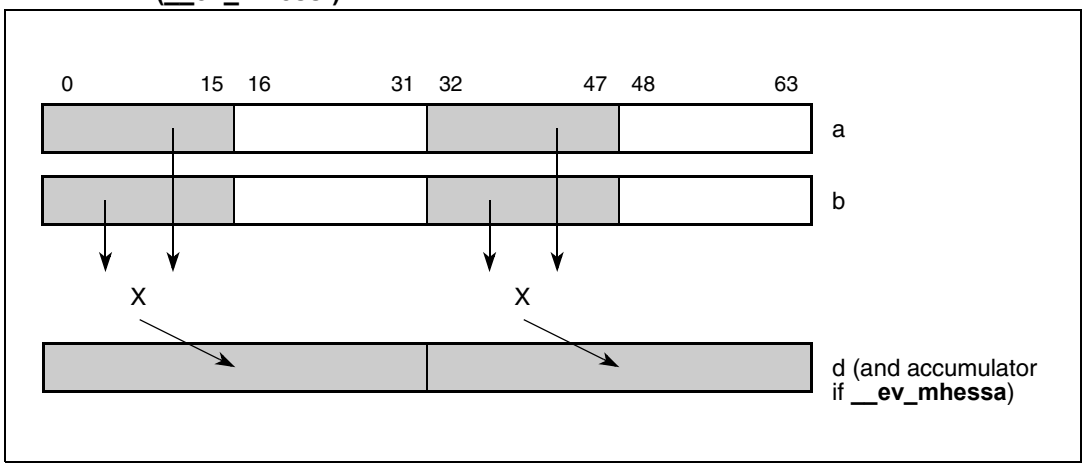


Table 116. __ev_mhessf (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhessf d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhessfa d,a,b**

__ev_mhessfaaw

Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate into Words

d = __ev_mhessfaaw (a,b)

```
// high
temp0:31 ← a0:15 ×sf b0:15
if (a0:15 = 0x8000) & (b0:15 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF //saturate
   movh ← 1
else
   movh ← 0
temp0:63 ← EXTS(ACC0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← a32:47 ×sf b32:47
if (a32:47 = 0x8000) & (b32:47 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF //saturate
   movl ← 1
else
   movl ← 0
temp0:63 ← EXTS(ACC32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding even-numbered half-word signed fractional elements in parameters a and b are multiplied, producing a 32-bit product. If both inputs are –1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then added to the corresponding word in the accumulator, saturating if overflow or underflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 111. Even form of vector half-word multiply (__ev_mhessfaaw)

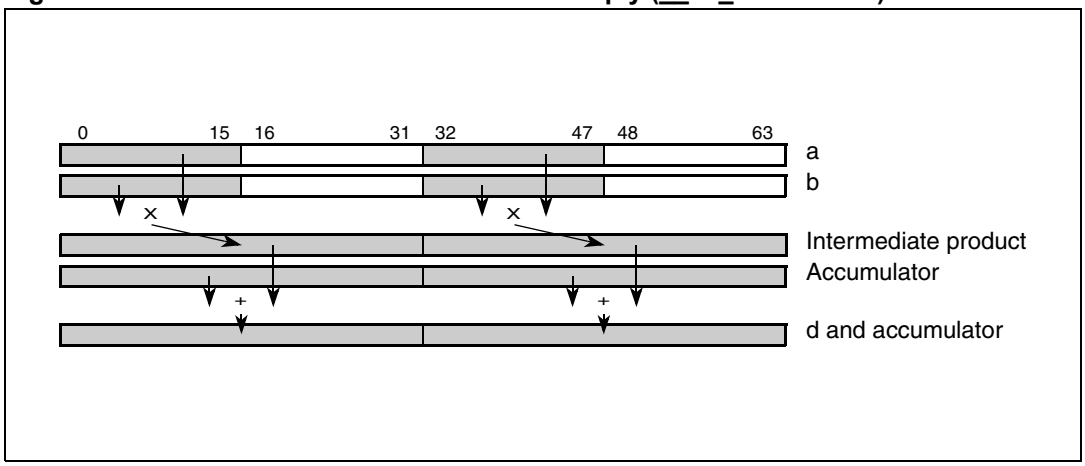


Table 117. __ev_mhessfaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhessfaaw d,a,b**

__ev_mhessfanw

Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate Negative into Words

d = __ev_mhessfanw (a,b)

```
// high
temp0:31 ← a0:15 ×sf b0:15
if (a0:15 = 0x8000) & (b0:15 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF //saturate
   movh ← 1
else
   movh ← 0
temp0:63 ← EXTS(ACC0:31) - EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← a32:47 ×sf b32:47
if (a32:47 = 0x8000) & (b32:47 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF //saturate
   movl ← 1
else
   movl ← 0
temp0:63 ← EXTS(ACC32:63) - EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding even-numbered half-word signed fractional elements in parameters a and b are multiplied, producing a 32-bit product. If both inputs are –1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then subtracted from the corresponding word in the accumulator, saturating if overflow or underflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 112. Even form of vector half-word multiply (__ev_mhessfanw)

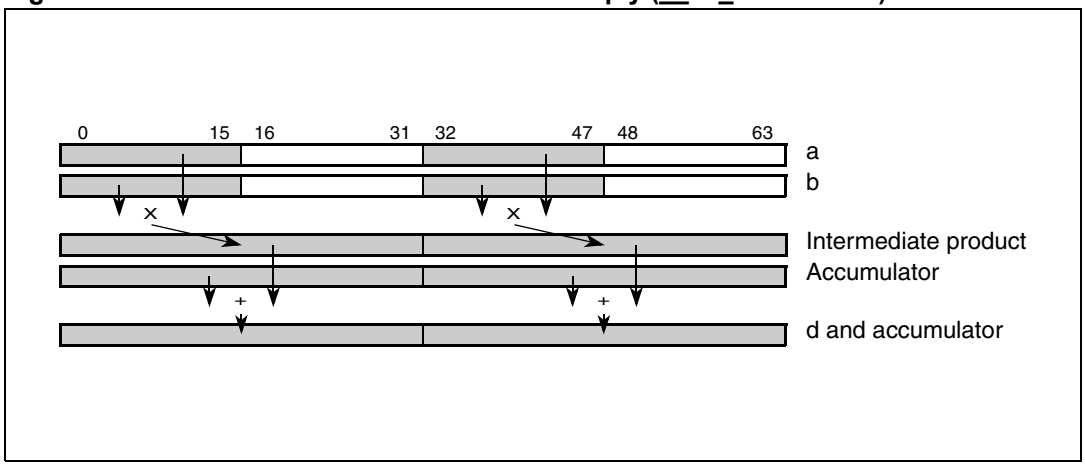


Table 118. __ev_mhessfanw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhessfanw d,a,b**

__ev_mhessiaaw

Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate into Words

d = __ev_mhessiaaw (a,b)

```
// high
temp0:31 ← a0:15 ×si b0:15
temp0:63 ← EXTS(ACC0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← a32:47 ×si b32:47
temp0:63 ← EXTS(ACC32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The corresponding even-numbered half-word signed integer elements in parameters a and b are multiplied, producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator, saturating if overflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 113. Even form of vector half-word multiply (__ev_mhessiaaw)

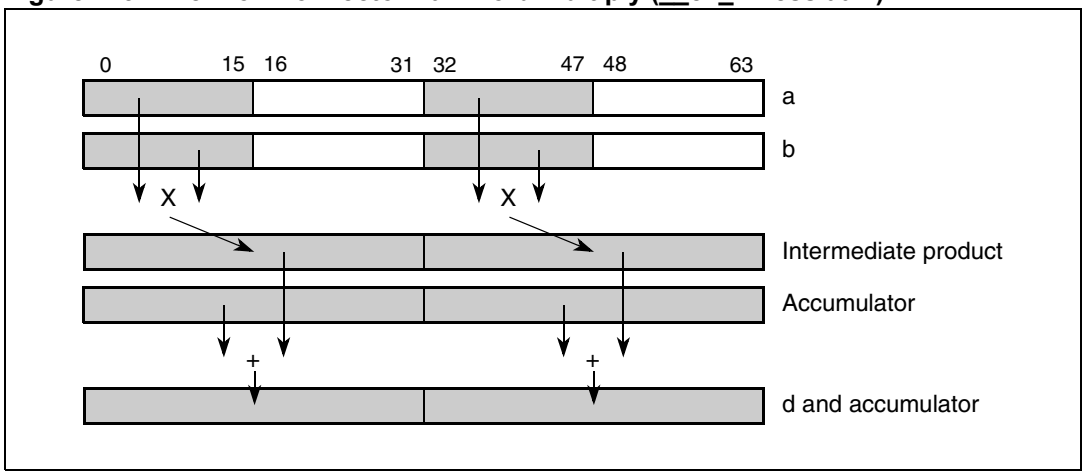


Table 119. __ev_mhessiaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhessiaaw d,a,b**

__ev_mhessianw

Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate Negative into Words

d = __ev_mhessianw (a,b)

```
// high
temp0:31 ← a0:15 ×si b0:15
temp0:63 ← EXTS(ACC0:31) - EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← a32:47 ×si b32:47
temp0:63 ← EXTS(ACC32:63) - EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, the corresponding even-numbered half-word signed integer elements in parameters a and b are multiplied, producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator, saturating if overflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 114. Even form of vector half-word multiply (__ev_mhessianw)

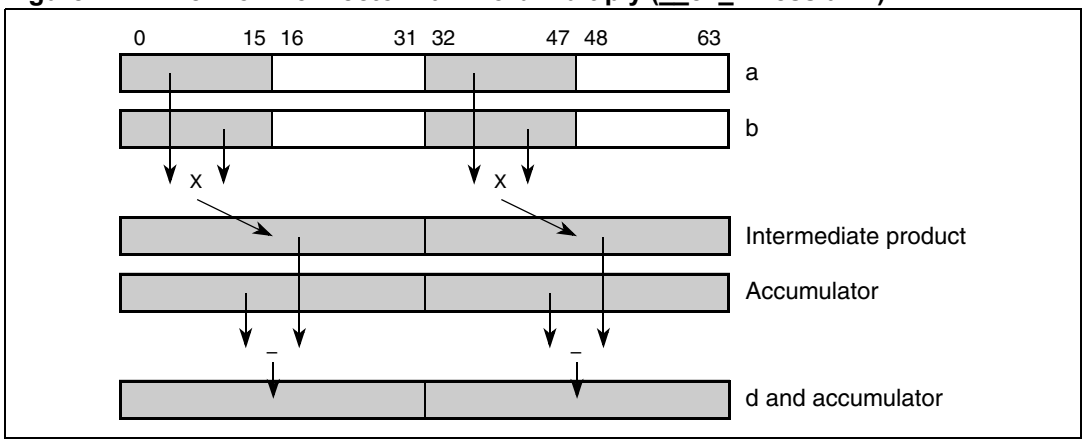


Table 120. __ev_mhessianw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhessianw d,a,b**

__ev_mheumf

Vector Multiply Half Words, Even, Unsigned, Modulo, Fractional (to Accumulator)

d = __ev_mheumf (a,b) (A = 0)

d = __ev_mheumfa (a,b) (A = 1)

```
// high
d0:31 ← a0:15 ×ui b0:15
// low
d32:63 ← a32:47 ×ui b32:47
// update accumulator
if A = 1, ACC0:63 ← d0:63
```

The corresponding even-numbered half word elements in parameters a and b are multiplied. The two 32-bit products are placed into the corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Figure 115. Vector multiply half words, even, unsigned, modulo, fractional (to accumulator) (__ev_mheumf)

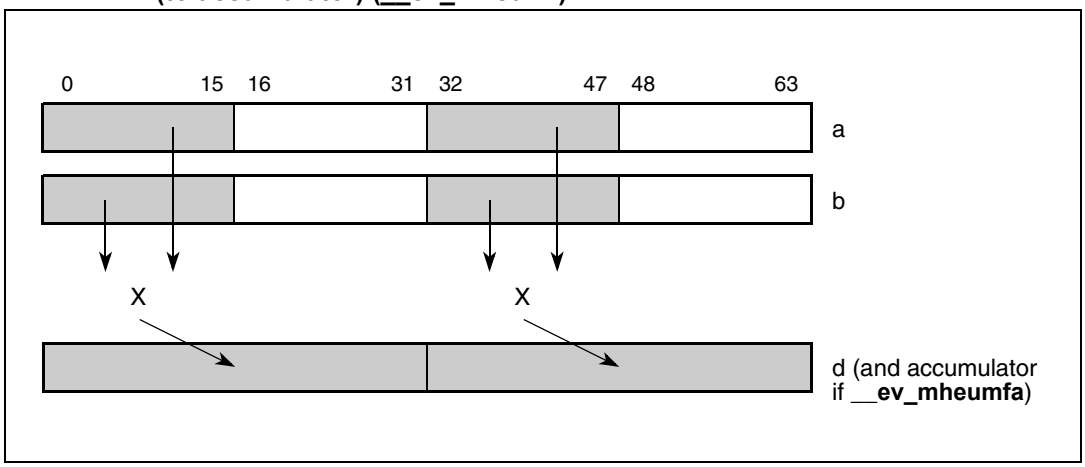


Table 121. __ev_mheumf (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheumi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheumia d,a,b**

__ev_mheumi

Vector Multiply Half Words, Even, Unsigned, Modulo, Integer (to Accumulator)

d = __ev_mheumi (a,b) (A = 0)

d = __ev_mheumia (a,b) (A = 1)

```
// high
d0:31 ← a0:15 ×ui b0:15

// low
d32:63 ← a32:47 ×ui b32:47

// update accumulator
if A = 1 then ACC0:63 ← d0:63
```

The corresponding even-numbered half-word unsigned integer elements in parameters a and b are multiplied. The two 32-bit products are placed into the corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Figure 116. Vector multiply half words, even, unsigned, modulo, integer (to accumulator) (__ev_mheumi)

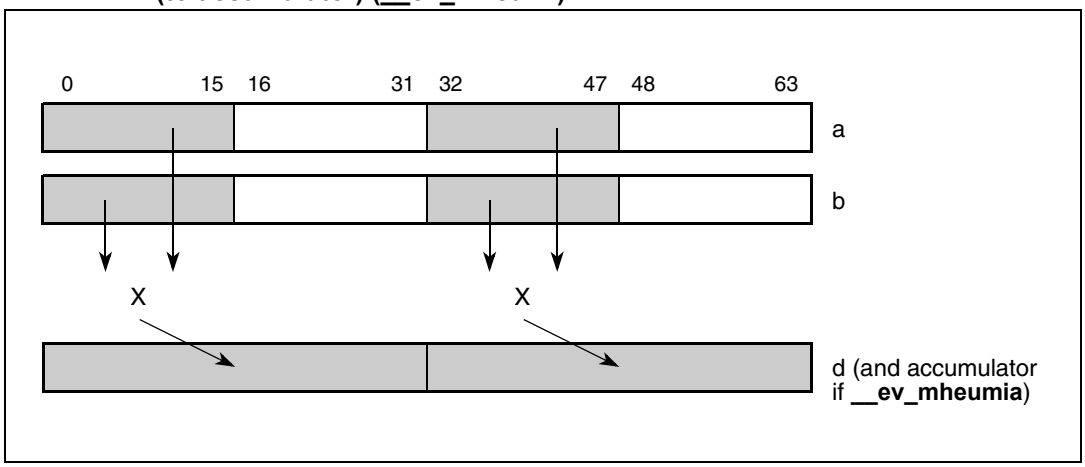


Table 122. __ev_mheumi (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheumi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheumia d,a,b**

__ev_mheumfaaw

Vector Multiply Half Words, Even, Unsigned, Modulo, Fractional and Accumulate into Words

d = __ev_mheumfaaw (a,b)

```
// high
temp0 0:31 ← a 0:15 ×ui b 0:15
d 0:31 ← ACC 0:31 + temp0 0:31
// low
temp1 0:31 ← a 32:47 ×ui b 32:47
d 32:63 ← ACC 32:63 + temp1 0:31
// update accumulator
ACC 0:63 ← d 0:63
```

For each word element in the accumulator, the corresponding even-numbered half word elements in parameters a and b are multiplied. Each intermediate product is added to the contents of the corresponding accumulator words, and the sums are placed into the corresponding parameter d and accumulator words.

Other registers altered: ACC

Figure 117. Even form of vector half-word multiply (__ev_mheumfaaw)

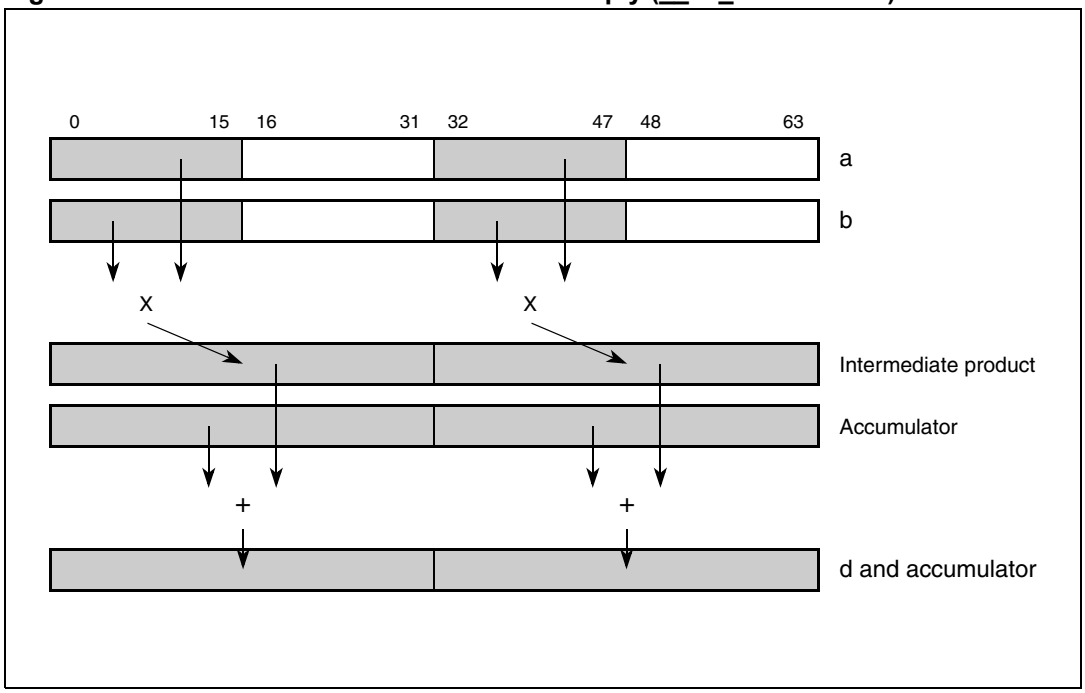


Table 123. __ev_mheumfaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheumiaaw d,a,b**

__ev_mheumiaaw

Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate into Words

d = __ev_mheumiaaw (a,b)

```
// high
temp0:31 ← a0:15 ×ui b0:15
d0:31 ← ACC0:31 + temp0:31

// low
temp0:31 ← a32:47 ×ui b32:47
d32:63 ← ACC32:63 + temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding even-numbered half-word unsigned integer elements in parameters a and b are multiplied. Each intermediate product is added to the contents of the corresponding accumulator words, and the sums are placed into the corresponding parameter d and accumulator words.

Other registers altered: ACC

Figure 118. Even form of vector half-word multiply (__ev_mheumiaaw)

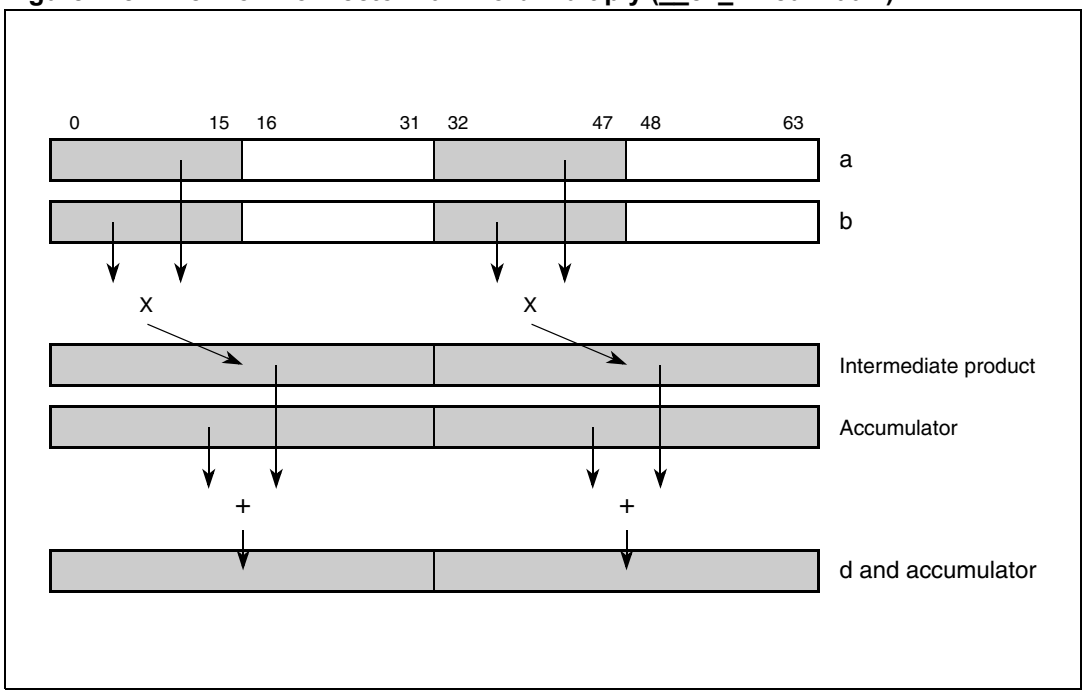


Table 124. __ev_mheumiaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheumiaaw d,a,b**

__ev_mheumfanw

Vector Multiply Half Words, Even, Unsigned, Modulo, Fractional and Accumulate Negative into Words

d = __ev_mheumfanw (a,b)

```
// high
temp0 0:31 ← a 0:15 ×ui b 0:15
d 0:31 ← ACC 0:31 - temp0 0:31
// low
temp1 0:31 ← a 32:47 ×ui b 32:47
d 32:63 ← ACC 32:63 - temp1 0:31
// update accumulator
ACC 0:63 ← d 0:63
```

For each word element in the accumulator, the corresponding even-numbered half word elements in parameters a and b are multiplied. Each intermediate product is subtracted from the contents of the corresponding accumulator words. The differences are placed into the corresponding parameter d and accumulator words.

Other registers altered: ACC

Figure 119. Even form of vector half-word multiply (__ev_mheumfanw)

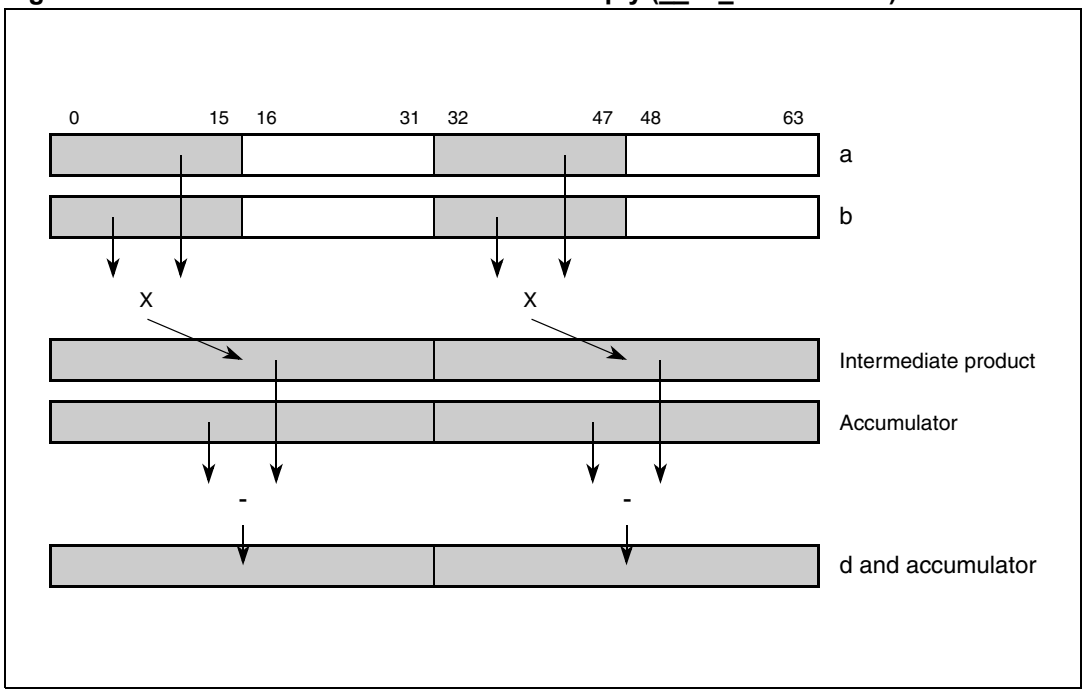


Table 125. __ev_mheumfanw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheumianw d,a,b**

__ev_mheumianw

Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words

d = __ev_mheumianw (a,b)

```
// high
temp0:31 ← a0:15 ×ui b0:15
d0:31 ← ACC0:31 - temp0:31

// low
temp0:31 ← a32:47 ×ui b32:47
d32:63 ← ACC32:63 - temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding even-numbered half-word unsigned integer elements in parameters a and b are multiplied. Each intermediate product is subtracted from the contents of the corresponding accumulator words. The differences are placed into the corresponding parameter d and accumulator words.

Other registers altered: ACC

Figure 120. Even form of vector half-word multiply (__ev_mheumianw)

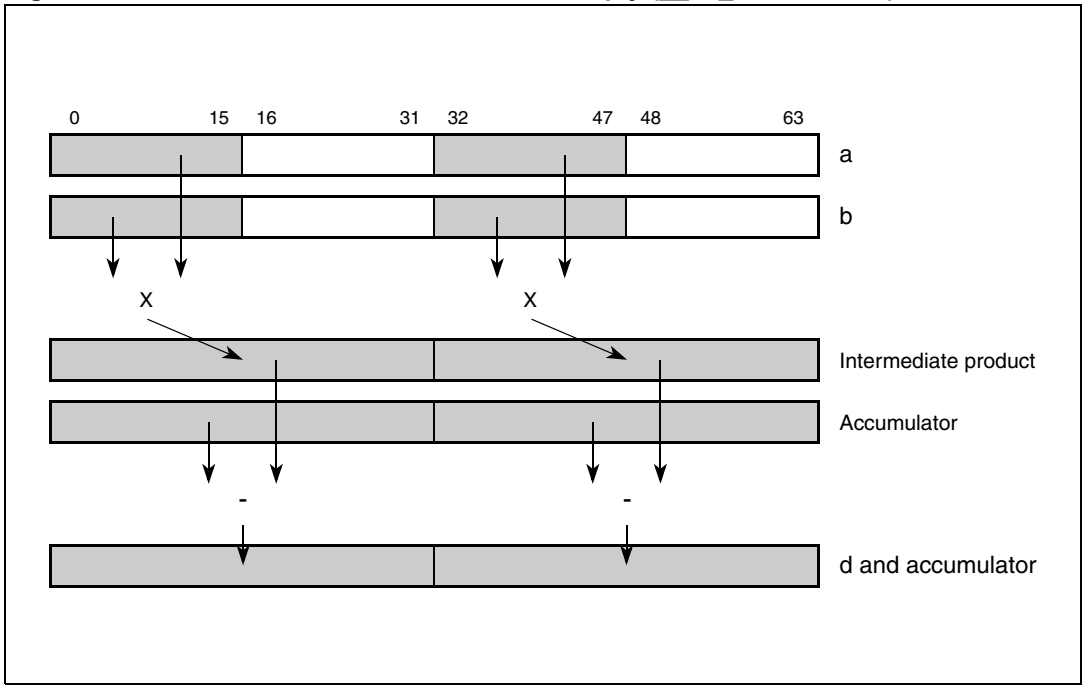


Table 126. __ev_mheumianw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheumianw d,a,b**

__ev_mheusfaaw

Vector Multiply Half Words, Even, Unsigned, Saturate, Fractional and Accumulate into Words

d = __ev_mheusfaaw (a,b)

```
// high
temp0[0:31] ← a[0:15] ×ui b[0:15]
temp0[0:63] ← EXTZ(ACC[0:31]) + EXTZ(temp0[0:31])
if temp0[31] = 1
   d[0:31] ← 0xFFFF_FFFF //overflow
   ovh ← 1
else
   d[0:31] ← temp0[32:63]
   ovh ← 0
//low
temp1[0:31] ← a[32:47] ×ui b[32:47]
temp1[0:63] ← EXTZ(ACC[32:63]) + EXTZ(temp1[0:31])
if temp1[31] = 1
   d[32:63] ← 0xFFFF_FFFF //overflow
   ovl ← 1
else
   d[32:63] ← temp1[32:63]
   ovl ← 0
// update accumulator
ACC[0:63] ← d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ← ovh
SPEFSCR[OV] ← ovl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | ovh
SPEFSCR[SOV] ← SPEFSCR[SOV] | ovl
```

For each word element in the accumulator, corresponding even-numbered half word elements in parameters a and b are multiplied. Each product is added to the contents of the corresponding accumulator words. If a sum overflows, 0xFFFF_FFFF is placed into the corresponding parameter d and accumulator words. Otherwise, the intermediate sums are placed there.

Overflow information is recorded in SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

Figure 121. Even form of vector half-word multiply (__ev_mheusfaaw)

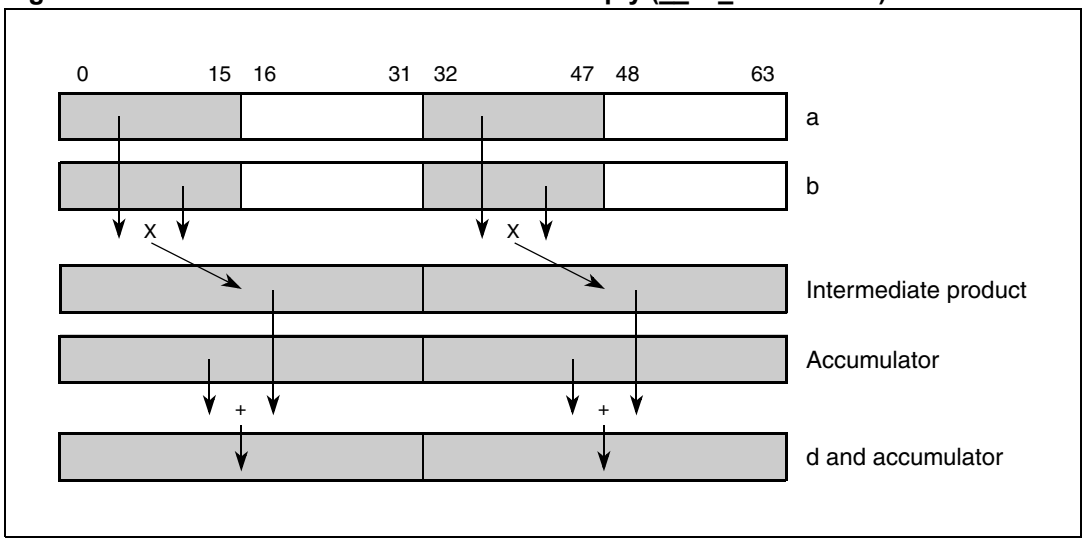


Table 127. __ev_mheusfaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheusiaaw d,a,b**

__ev_mheusiaaw

Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate into Words

d = __ev_mheusiaaw (a,b)

```
// high
temp0:31 ← a0:15 ×ui b0:15
temp0:63 ← EXTZ(ACC0:31) + EXTZ(temp0:31)
ovh ← temp31
d0:31 ← SATURATE(ovh, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temp32:63)

//low
temp0:31 ← a32:47 ×ui b32:47
temp0:63 ← EXTZ(ACC32:63) + EXTZ(temp0:31)
ovl ← temp31
d32:63 ← SATURATE(ovl, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding even-numbered half-word unsigned integer elements in parameters a and b are multiplied, producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator, saturating if overflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow from the addition, the overflow and summary overflow bits are recorded in the SPEFSCR. Other registers altered: SPEFSCR ACC

Figure 122. Even form of vector half-word multiply (__ev_mheusiaaw)

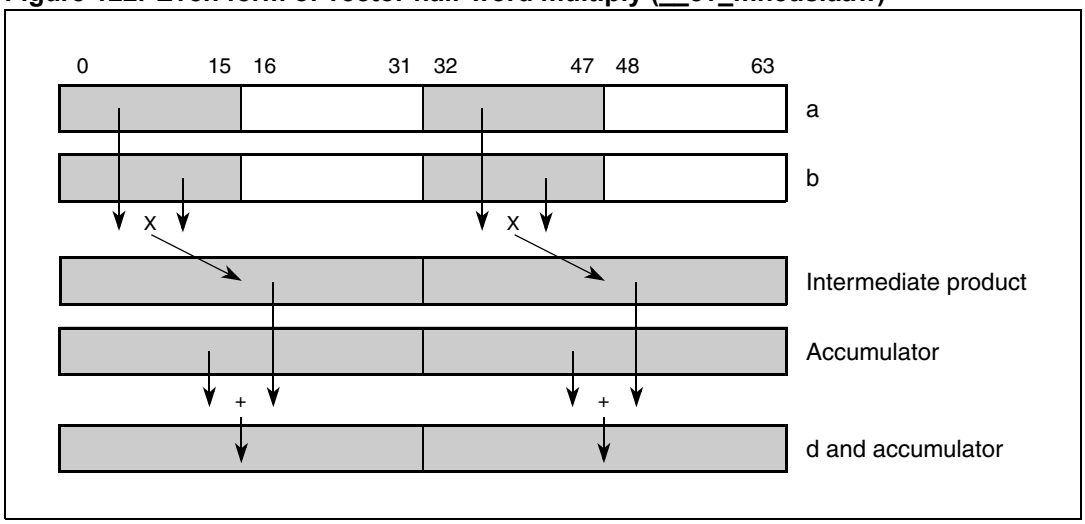


Table 128. __ev_mheusiaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheusiaaw d,a,b**

__ev_mheusfanw

Vector Multiply Half Words, Even, Unsigned, Saturate, Fractional and Accumulate Negative into Words

d = __ev_mheusfanw (a,b)

```
// high
temp0[0:31] ← a[0:15] ×ui b[0:15]
temp0[0:63] ← EXTZ(ACC[0:31]) - EXTZ(temp0[0:31])
if temp0[31] = 1
   d[0:31] ← 0xFFFF_FFFF //overflow
   ovh ← 1
else
   d[0:31] ← temp0[32:63]
   ovh ← 0
//low
temp1[0:31] ← a[32:47] ×ui b[32:47]
temp1[0:63] ← EXTZ(ACC[32:63]) - EXTZ(temp1[0:31])
if temp1[31] = 1
   d[32:63] ← 0xFFFF_FFFF //overflow
   ovl ← 1
else
   d[32:63] ← temp1[32:63]
   ovl ← 0
// update accumulator
ACC[0:63] ← d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ← ovh
SPEFSCR[OV] ← ovl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | ovh
SPEFSCR[SOV] ← SPEFSCR[SOV] | ovl
```

For each word element in the accumulator, corresponding even-numbered half word elements in parameters a and b are multiplied. Each product is subtracted from the contents of the corresponding accumulator words. If a result overflows, 0xFFFF_FFFF is placed into the corresponding parameter d and accumulator words. Otherwise, the intermediate results are placed there.

Overflow information is recorded in SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

Figure 123. Even form of vector half-word multiply (__ev_mheusfanw)

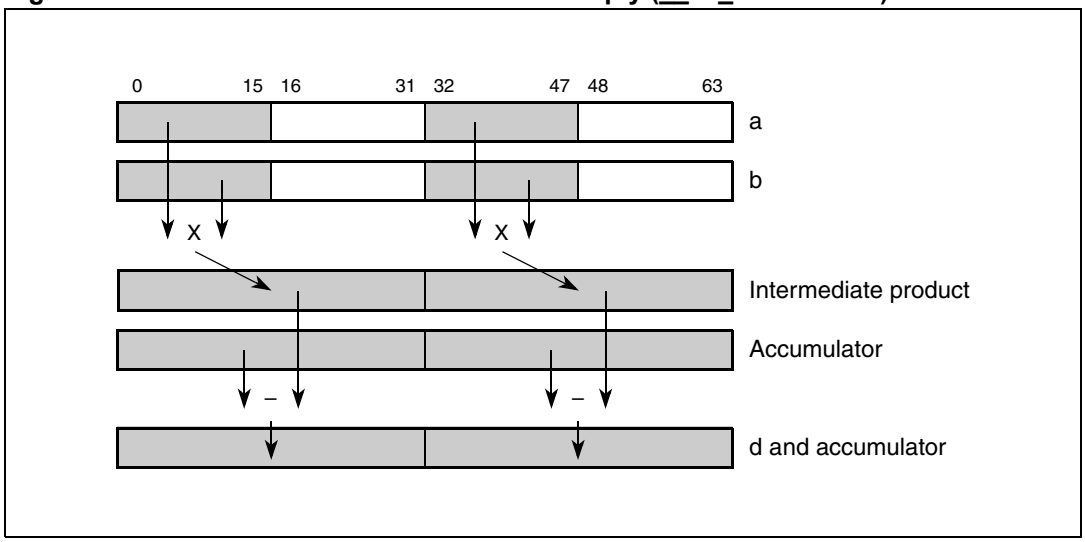


Table 129. __ev_mheusfanw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheusianw d,a,b**

__ev_mheusianw

Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words

d = __ev_mheusianw (a,b)

```
// high
temp0:31 ← a0:15 ×ui b0:15
temp0:63 ← EXTZ(ACC0:31) - EXTZ(temp0:31)
ovh ← temp31
d0:31 ← SATURATE(ovh, 0, 0x0000_0000, 0x0000_0000, temp32:63)

//low
temp0:31 ← a32:47 ×ui b32:47
temp0:63 ← EXTZ(ACC32:63) - EXTZ(temp0:31)
ovl ← temp31
d32:63 ← SATURATE(ovl, 0, 0x0000_0000, 0x0000_0000, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding even-numbered half-word unsigned integer elements in parameters a and b are multiplied, producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator, saturating if underflow occurs, and the result is placed in parameter d and the accumulator.

If there is an underflow from the subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 124. Even form of vector half-word multiply (__ev_mheusianw)

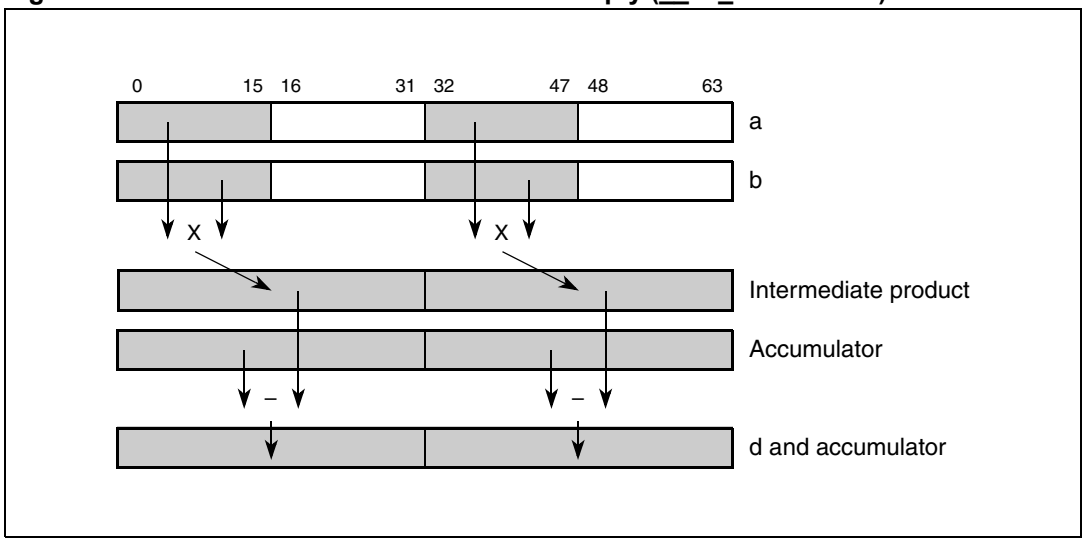


Table 130. __ev_mheusianw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheusianw d,a,b**

__ev_mhogsmfaa

Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate

d = __ev_mhogsmfaa (a,b)

$temp_{0:31} \leftarrow a_{48:63} \times_{sf} b_{48:63}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low odd-numbered half-word signed fractional elements in parameters a and b are multiplied. The intermediate product is sign-extended to 64 bits and added to the contents of the 64-bit accumulator. This result is placed into parameter d and into the accumulator.

Note: This sum is a modulo sum. Neither overflow check nor saturation is performed. If an overflow from the 64-bit sum occurs, it is not recorded into the SPEFSCR.

Figure 125. __ev_mhogsmfaa (odd form)

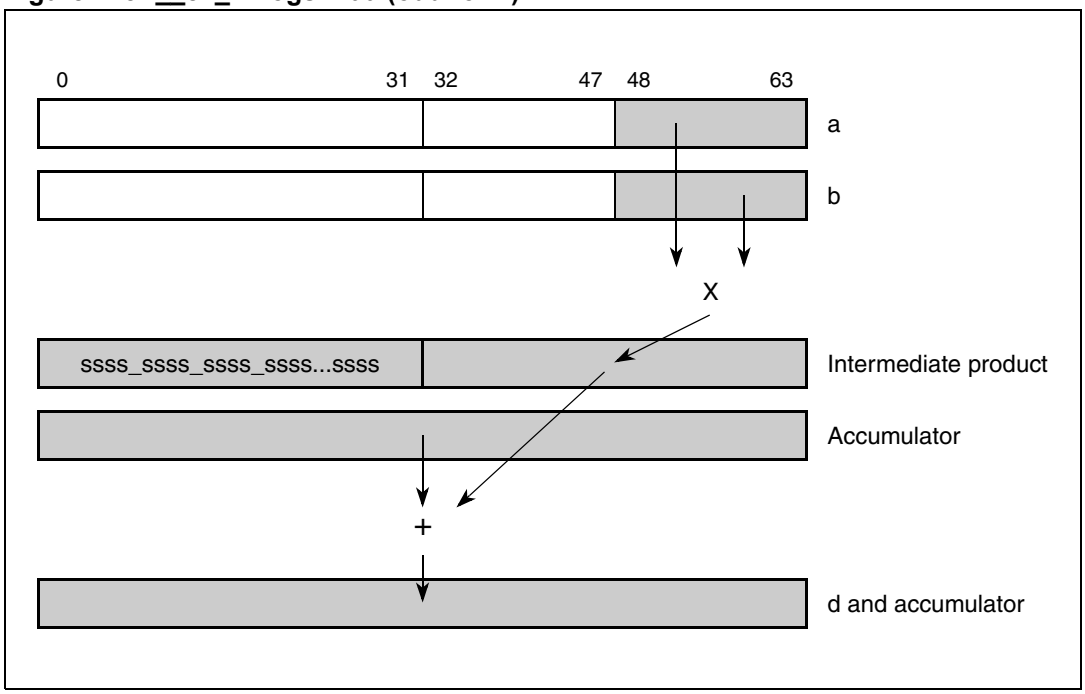


Table 131. __ev_mhogsmfaa (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhogsmfaa d,a,b**

__ev_mhogsmfan

Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative

d = __ev_mhogsmfan (a,b)

$temp_{0:31} \leftarrow a_{48:63} \times_{sf} b_{48:63}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low odd-numbered half-word signed fractional elements in parameters a and b are multiplied. The intermediate product is sign-extended to 64 bits and subtracted from the contents of the 64-bit accumulator. This result is placed into parameter d and into the accumulator.

Note: This difference is a modulo difference. Neither overflow check nor saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

Figure 126. __ev_mhogsmfan (odd form)

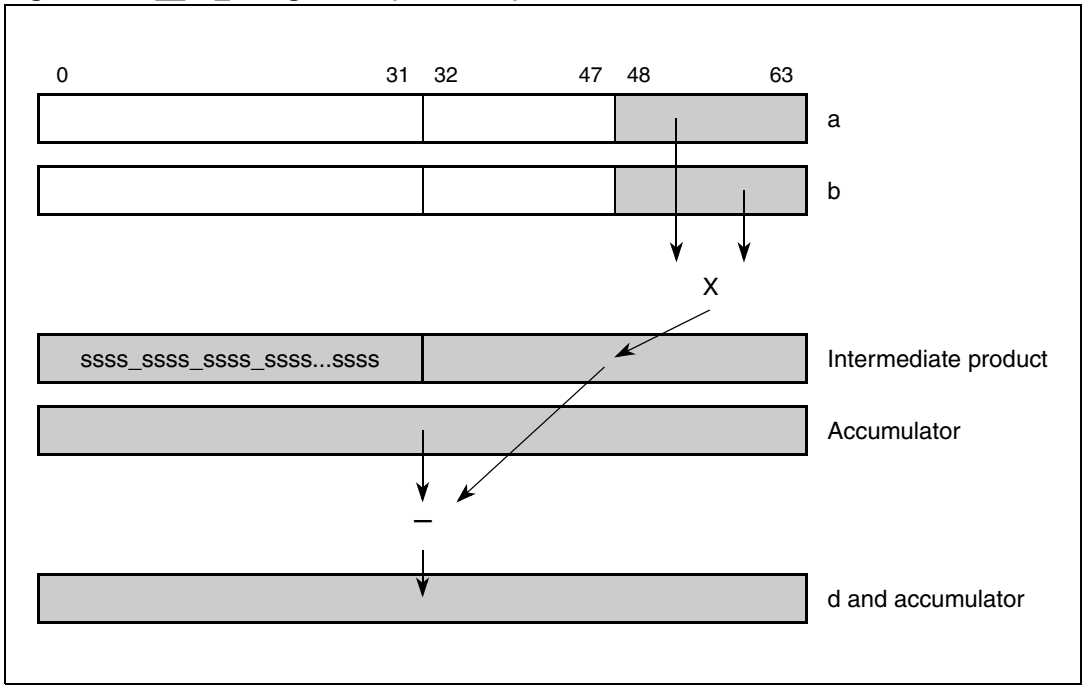


Table 132. __ev_mhogsmfan (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhogsmfan d,a,b**

__ev_mhogsmiaa

Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Intege and Accumulate

d = __ev_mhogsmiaa (a,b)

$temp_{0:31} \leftarrow a_{48:63} \times_{si} b_{48:63}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low odd-numbered half-word signed integer elements in parameters a and b are multiplied. The intermediate product is sign-extended to 64 bits and added to the contents of the 64-bit accumulator. This sum is placed into parameter d and into the accumulator.

Note: This sum is a modulo sum. Neither overflow check nor saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into the SPEFSCR.

Figure 127. __ev_mhogsmiaa (odd form)

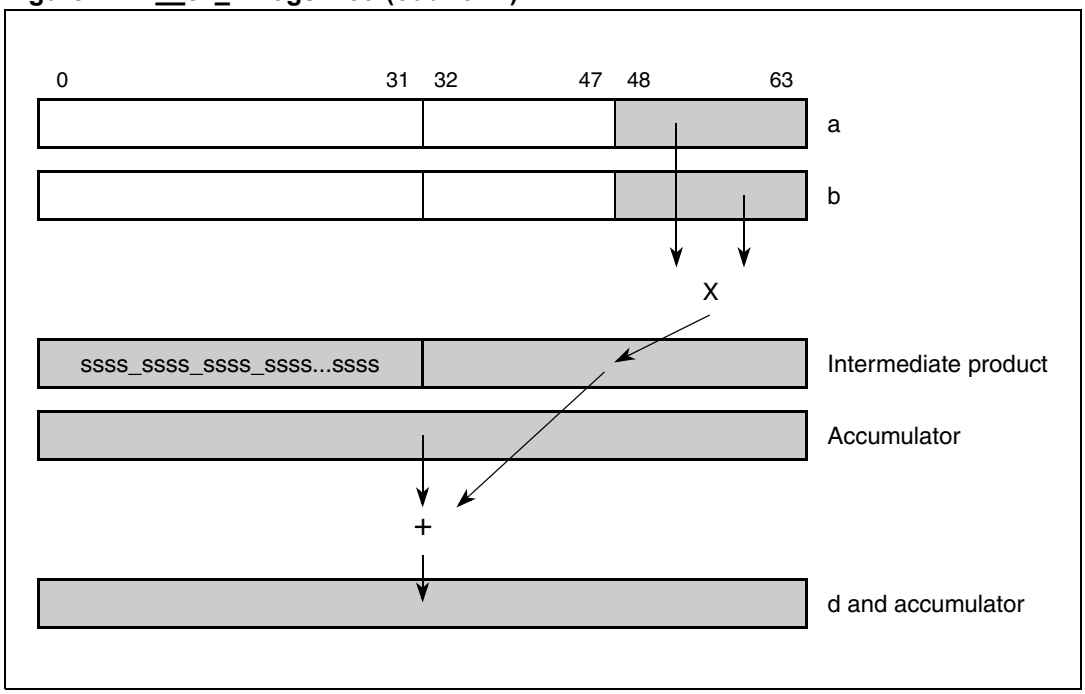


Table 133. __ev_mhogsmiaa (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhogsmiaa d,a,b**

__ev_mhogsmian

Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative

d = __ev_mhogsmian (a,b)

$temp_{0:31} \leftarrow a_{48:63} \times_{si} b_{48:63}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low odd-numbered half-word signed integer elements in parameters a and b are multiplied. The intermediate product is sign-extended to 64 bits and subtracted from the contents of the 64-bit accumulator. This result is placed into parameter d and into the accumulator.

Note: This difference is a modulo difference. Neither overflow check nor saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

Figure 128. __ev_mhogsmian (odd form)

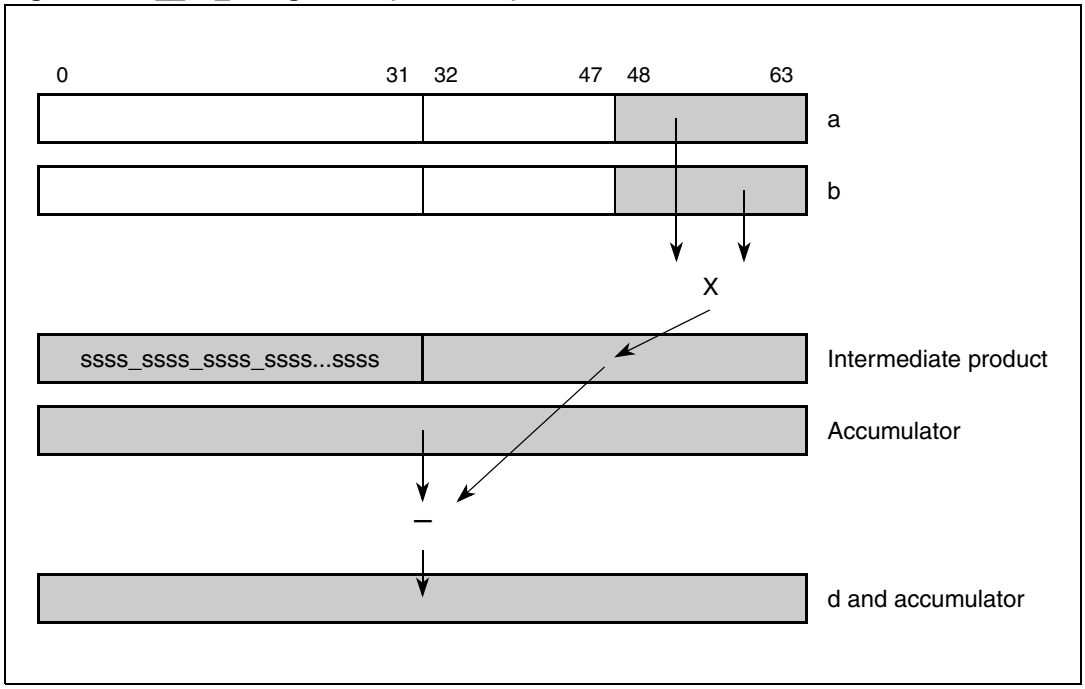


Table 134. __ev_mhogsmian (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhogsmian d,a,b**

__ev_mhogumfaa

Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Fractional and Accumulate

d = __ev_mhogumfaa (a,b)

$temp_{0:31} \leftarrow a_{48:63} \times_{ui} b_{48:63}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low odd-numbered half word elements in parameters a and b are multiplied. The intermediate product is zero-extended to 64 bits and added to the contents of the 64-bit accumulator. This sum is placed into parameter d and into the accumulator.

Note: This sum is a modulo sum. Neither overflow check nor saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into the SPEFSCR.

Figure 129. __ev_mhogumfaa (odd form)

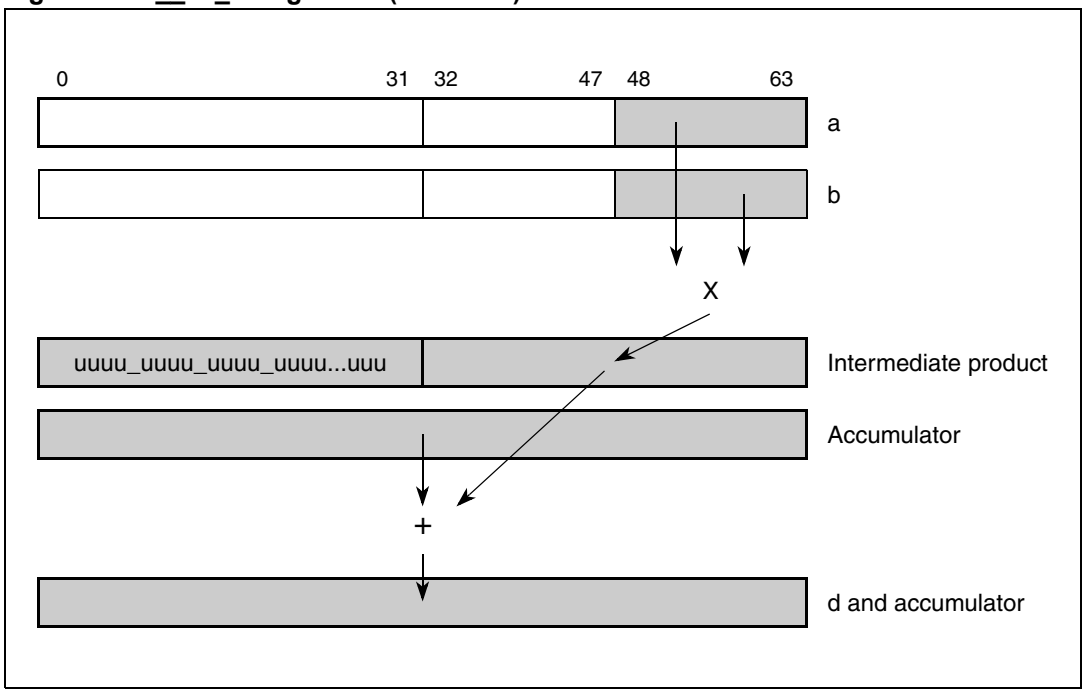


Table 135. __ev_mhogumfaa (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhogumiaa d,a,b**

__ev_mhogumiaa

Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate

d = __ev_mhogumiaa (a,b)

$temp_{0:31} \leftarrow a_{48:63} \times_{ui} b_{48:63}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low odd-numbered half-word unsigned integer elements in parameters a and b are multiplied. The intermediate product is zero-extended to 64 bits and added to the contents of the 64-bit accumulator. This sum is placed into parameter d and into the accumulator.

Note: This sum is a modulo sum. Neither overflow check nor saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into the SPEFSCR.

Figure 130. __ev_mhogumiaa (odd form)

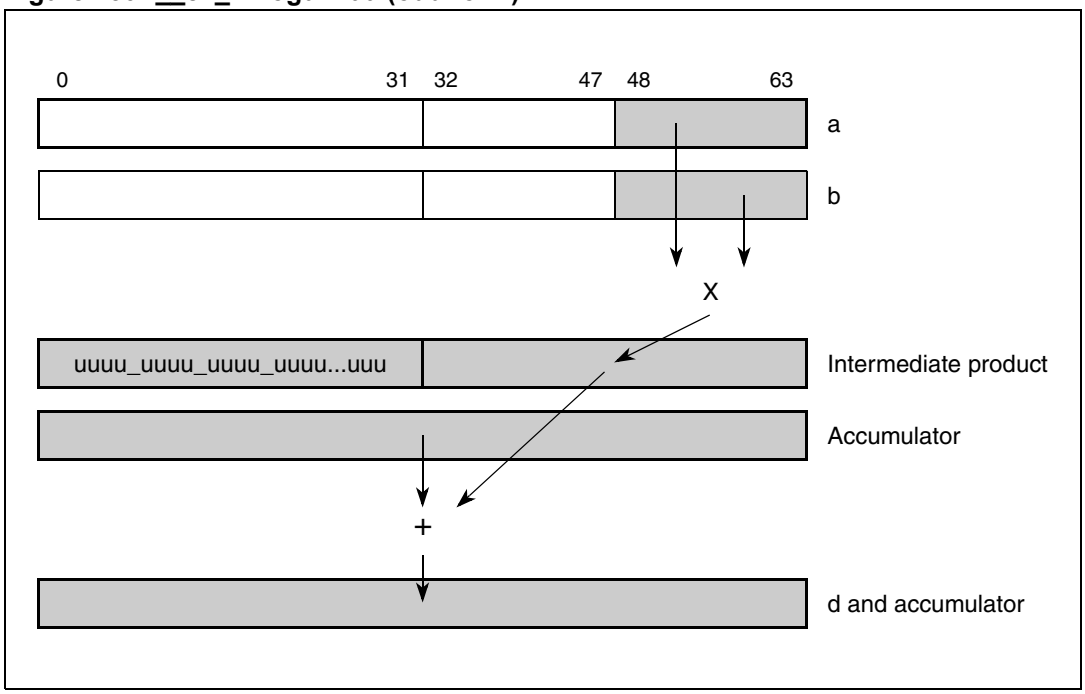


Table 136. __ev_mhogumiaa (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhogumiaa d,a,b**

__ev_mhogumfan

Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Fractional and Accumulate Negative

d = __ev_mhogumfan (a,b)

$temp_{0:31} \leftarrow a_{48:63} \times_{ui} b_{48:63}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low odd-numbered half word elements in parameters a and b are multiplied. The intermediate product is zero-extended to 64 bits and subtracted from the contents of the 64-bit accumulator. This result is placed into parameter d and into the accumulator.

Note: This difference is a modulo difference. Neither overflow check nor saturation is performed. Overflow of the 64-bit difference is not recorded into the SPEFSCR.

Figure 131. __ev_mhogumfan (odd form)

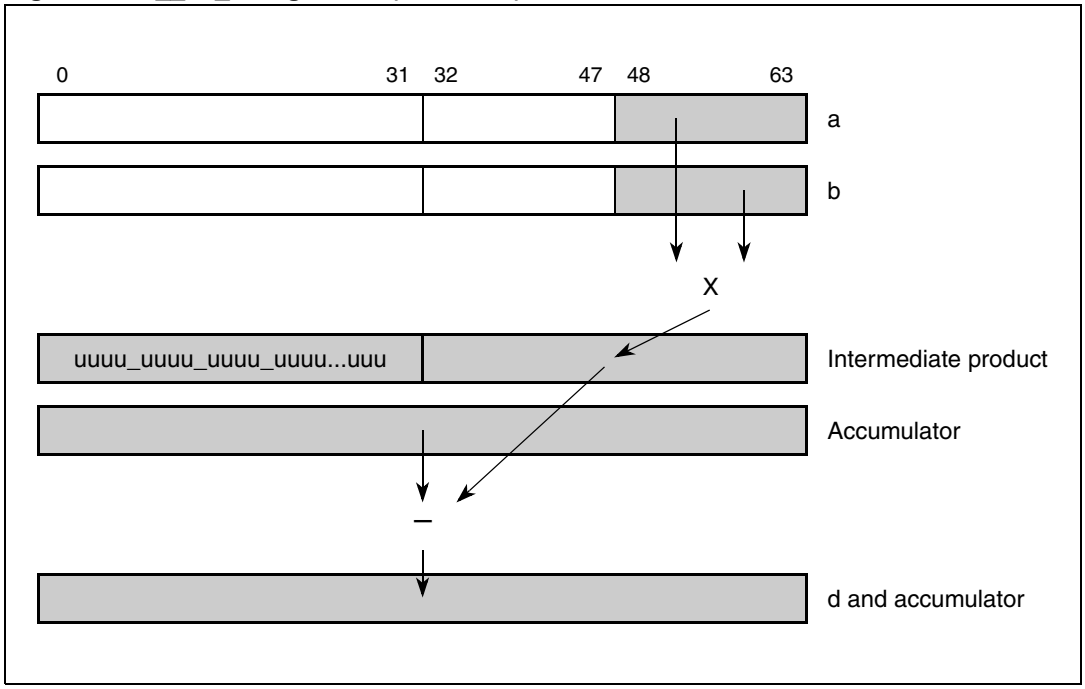


Table 137. __ev_mhogumfan (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhogumian d,a,b**

__ev_mhogumian

Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative

d = __ev_mhogumian (a,b)

$temp_{0:31} \leftarrow a_{48:63} \times_{ui} b_{48:63}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low odd-numbered half-word unsigned integer elements in parameters a and b are multiplied. The intermediate product is zero-extended to 64 bits and subtracted from the contents of the 64-bit accumulator. This result is placed into parameter d and into the accumulator.

Note: This difference is a modulo difference. Neither overflow check nor saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

Figure 132. __ev_mhogumian (odd form)

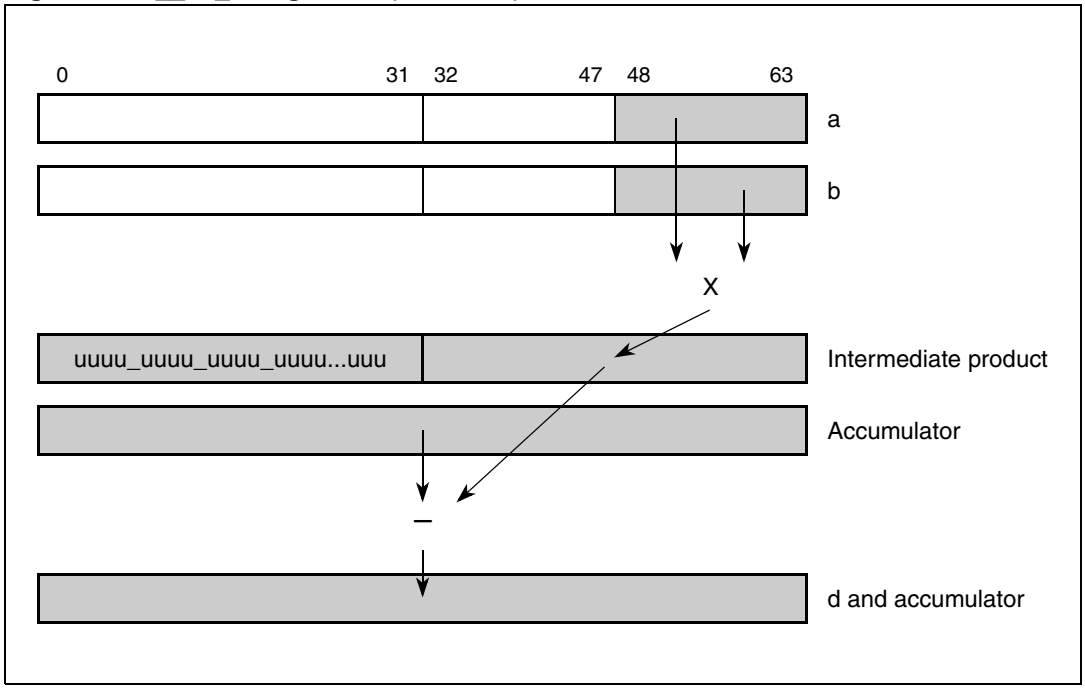


Table 138. __ev_mhogumian (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhogumian d,a,b**

__ev_mhosmf

Vector Multiply Half Words, Odd, Signed, Modulo, Fractional (to Accumulator)

d = __ev_mhosmf (a,b) (A = 0)

d = __ev_mhosmfa (a,b) (A = 1)

```
// high
d0:31 ← a16:31 ×sf b16:31
// low
d32:63 ← a48:63 ×sf b48:63
// update accumulator
if A = 1, then ACC0:63 ← d0:63
```

The corresponding odd-numbered, half-word signed fractional elements in parameters a and b are multiplied. Each product is placed into the corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

Figure 133. Vector multiply half words, odd, signed, modulo, fractional (to accumulator) (__ev_mhosmf)

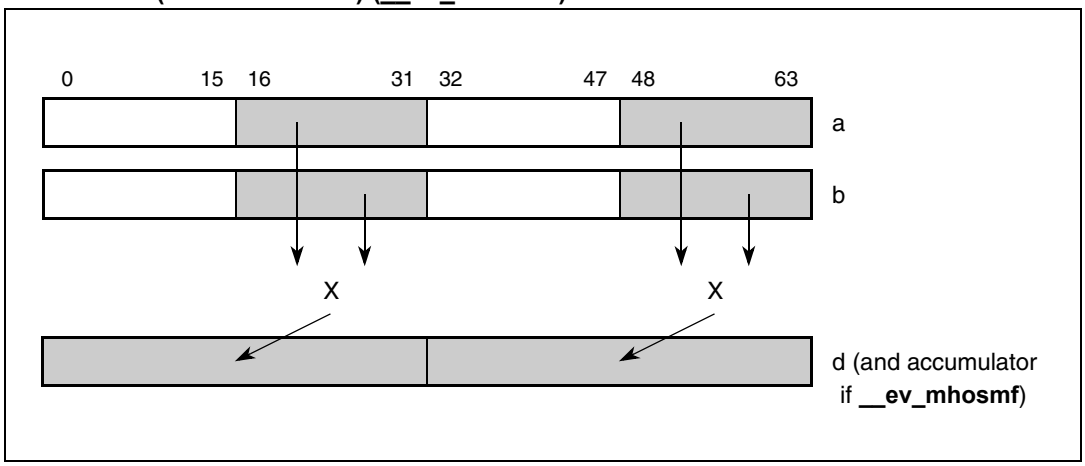


Table 139. __ev_mhosmf (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhosmf d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhosmfa d,a,b**

__ev_mhosmfaaw

Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate into Words

d = __ev_mhosmfaaw (a,b)

```
// high
temp0:31 ← a16:31 ×sf b16:31
d0:31 ← ACC0:31 + temp0:31

// low
temp0:31 ← a48:63 ×sf b48:63
d32:63 ← ACC32:63 + temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding odd-numbered half-word signed fractional elements in parameters a and b are multiplied. The 32 bits of each intermediate product is added to the contents of the corresponding accumulator word, and the results are placed into the corresponding parameter d words and into the accumulator

Other registers altered: ACC

Figure 134. Odd form of vector half-word multiply (__ev_mhosmfaaw)

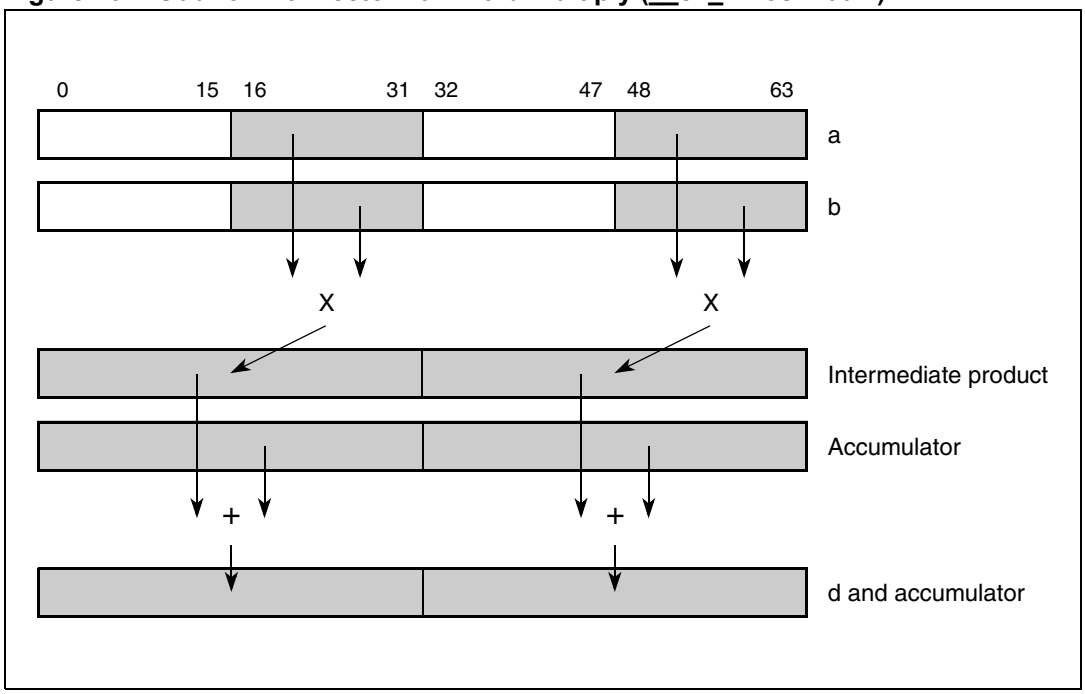


Table 140. __ev_mhosmfaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhosmfaaw d,a,b**

__ev_mhosmfanw

Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words

d = __ev_mhosmfanw (a,b)

```
// high
temp0:31 ← a16:31 ×sf b16:31
d0:31 ← ACC0:31 - temp0:31

// low
temp0:31 ← a48:63 ×sf b48:63
d32:63 ← ACC32:63 - temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding odd-numbered half-word signed fractional elements in parameters a and b are multiplied. The 32 bits of each intermediate product is subtracted from the contents of the corresponding accumulator word. The word and the results are placed into the corresponding parameter d word and into the accumulator.

Other registers altered: ACC

Figure 135. Odd form of vector half-word multiply (__ev_mhosmfanw)

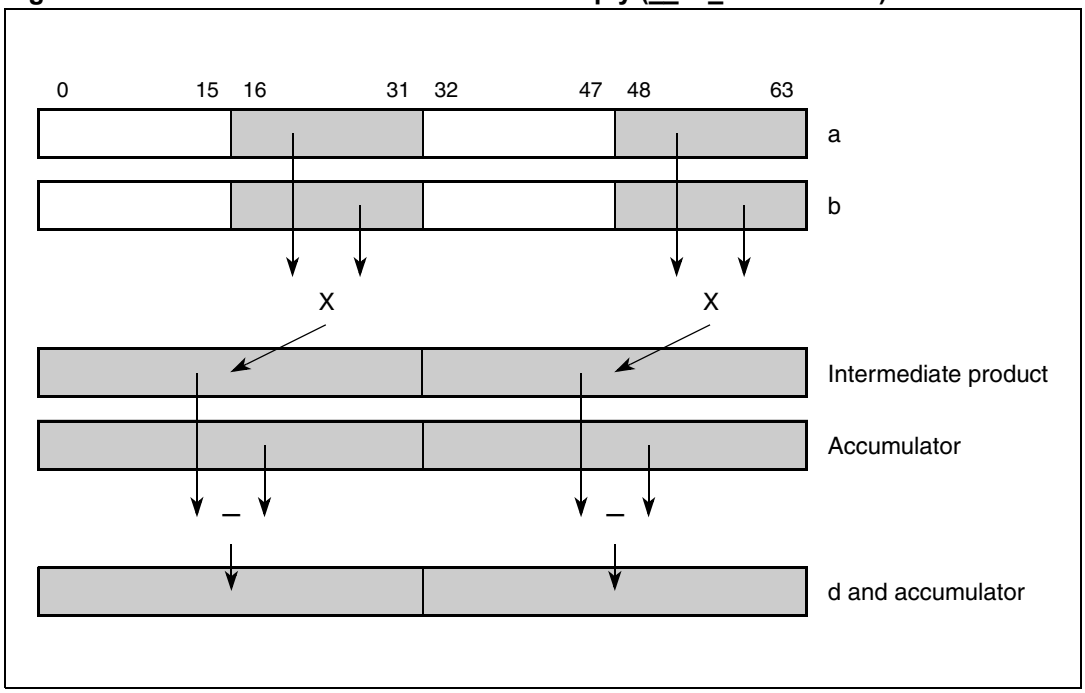


Table 141. __ev_mhosmfanw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhosmfanw d,a,b**

__ev_mhosmi

Vector Multiply Half Words, Odd, Signed, Modulo, Integer (to Accumulator)

d = __ev_mhosmi (a,b) (A = 0)

d = __ev_mhosmia (a,b) (A = 1)

```
// high
d0:31 ← a16:31 ×si b16:31
// low
d32:63 ← a48:63 ×si b48:63
// update accumulator
if A = 1, then ACC0:63 ← d0:63
```

The corresponding odd-numbered half-word signed integer elements in parameters a and b are multiplied. The two 32-bit products are placed into the corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

Figure 136. Vector multiply half words, odd, signed, modulo, integer (to accumulator) (__ev_mhosmi)

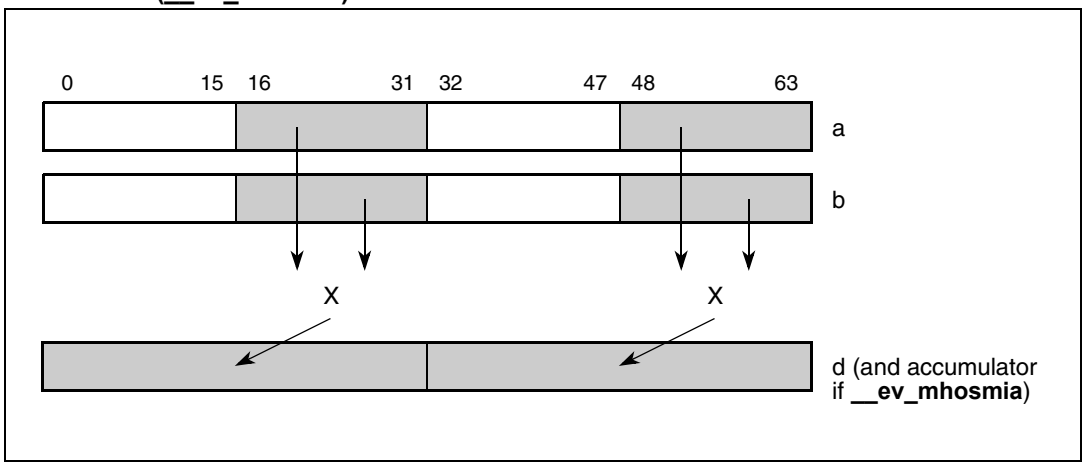


Table 142. __ev_mhosmi (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhosmi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhosmia d,a,b**

__ev_mhosmiaaw

Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate into Words

d = __ev_mhosmiaaw (a,b)

```
// high
temp0:31 ← a16:31 ×si b16:31
d0:31 ← ACC0:31 + temp0:31

// low
temp0:31 ← a48:63 ×si b48:63
d32:63 ← ACC32:63 + temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding odd-numbered half-word signed integer elements in parameters a and b are multiplied. Each intermediate 32-bit product is added to the contents of the corresponding accumulator word and the results are placed into the corresponding parameter d words and into the accumulator.

Other registers altered: ACC

Figure 137. Odd form of vector half-word multiply (__ev_mhosmiaaw)

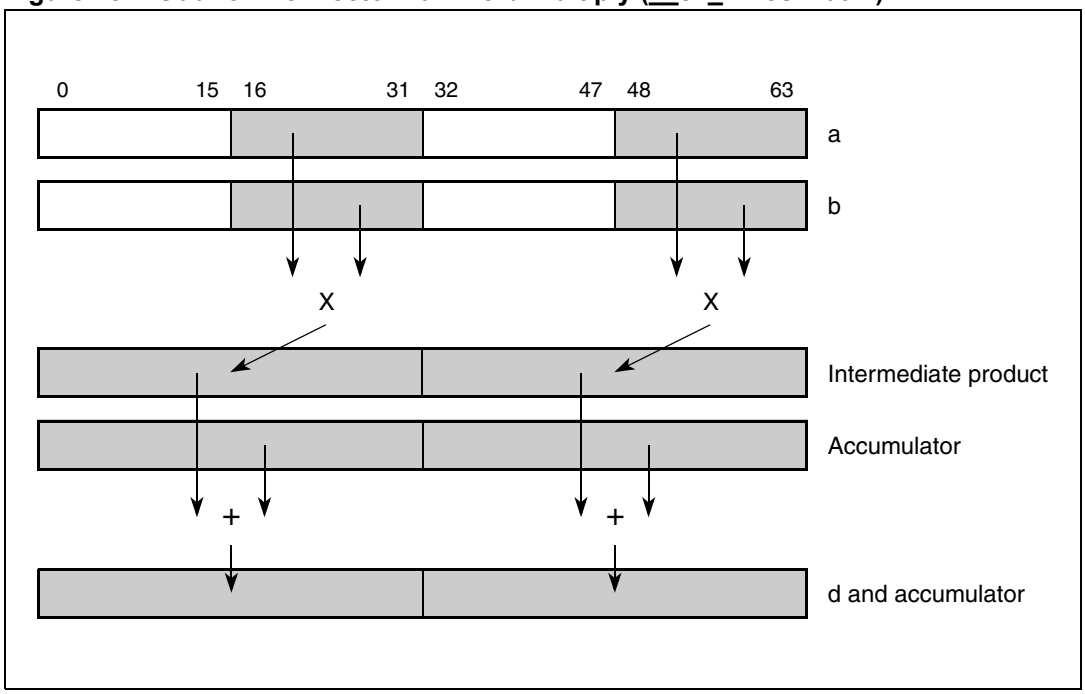


Table 143. __ev_mhosmiaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhosmiaaw d,a,b**

__ev_mhosmianw

Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate Negative into Words

d = __ev_mhosmianw (a,b)

```
// high
temp0:31 ←a16:31 ×si b16:31
d0:31 ← ACC0:31 - temp0:31

// low
temp0:31 ← a48:63 ×si b48:63
d32:63 ← ACC32:63 - temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding odd-numbered half-word signed integer elements in parameters a and b are multiplied. Each intermediate 32-bit product is subtracted from the contents of the corresponding accumulator word and the results are placed into the corresponding parameter d words and into the accumulator.

Other registers altered: ACC

Figure 138. Odd form of vector half-word multiply (__ev_mhosmianw)

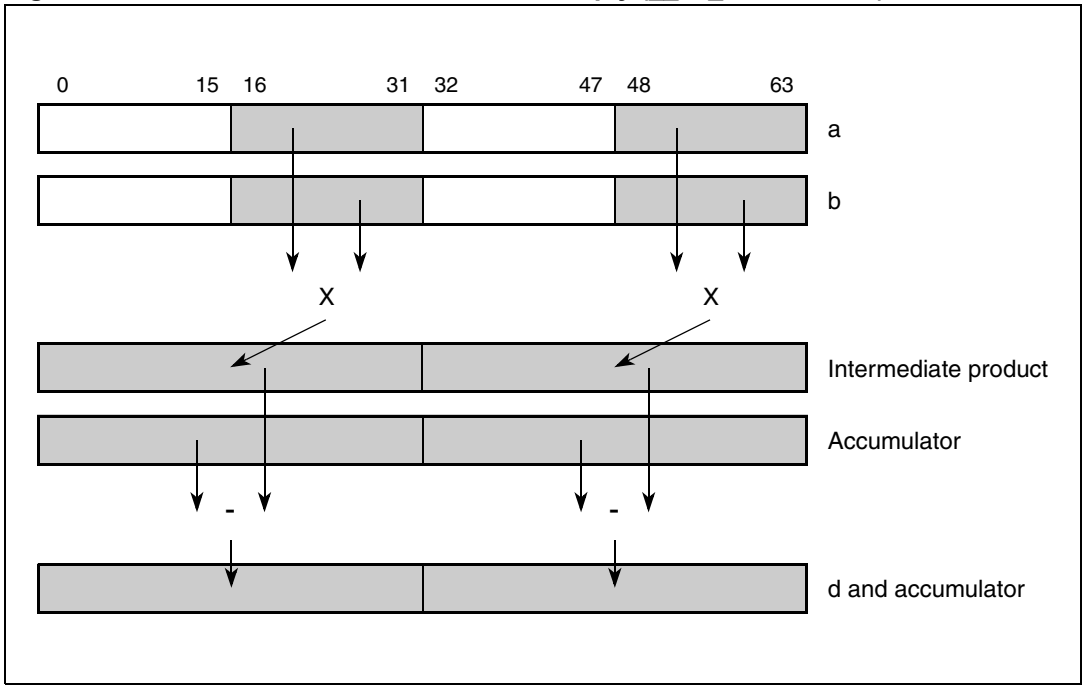


Table 144. __ev_mhosmianw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhosmianw d,a,b**

__ev_mhossf

Vector Multiply Half Words, Odd, Signed, Saturate, Fractional (to Accumulator)

d = __ev_mhossf (a,b) (A = 0)

d = __ev_mhossfa (a,b) (A = 1)

```
// high
temp0:31 ← a16:31 ×sf b16:31
if (a16:31 = 0x8000) & (b16:31 = 0x8000) then
   d0:31 ← 0x7FFF_FFFF //saturate
   movh ← 1
else
   d0:31 ← temp0:31
   movh ← 0

// low
temp0:31 ← a48:63 ×sf b48:63
if (a48:63 = 0x8000) & (b48:63 = 0x8000) then
   d32:63 ← 0x7FFF_FFFF //saturate
   movl ← 1
else
   d32:63 ← temp0:31
   movl ← 0

// update accumulator
if A = 1 then ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding odd-numbered half-word signed fractional elements in parameters a and b are multiplied. The 32 bits of each product are placed into the corresponding words of parameter d. If both inputs are -1.0, the result saturates to the largest positive signed fraction and the overflow and summary overflow bits are recorded in the SPEFSCR.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: SPEFSCR
ACC (if A = 1)

Figure 139. Vector multiply half words, odd, signed, saturate, fractional (to Accumulator) (__ev_mhossf)

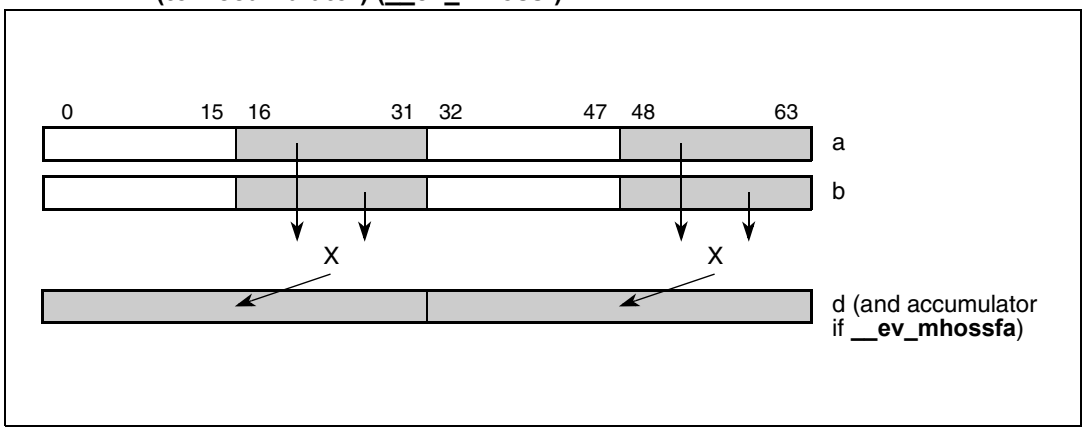


Table 145. __ev_mhossf (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhossf d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhossfa d,a,b**

__ev_mhossfaaw

Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate into Words

d = __ev_mhossfaaw (a,b)

```
// high
temp0:31 ← a16:31 ×sf b16:31
if (a16:31 = 0x8000) & (b16:31 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF //saturate
   movh ← 1
else
   movh ← 0
temp0:63 ← EXTS(ACC0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← a48:63 ×sf b48:63
if (a48:63 = 0x8000) & (b48:63 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF //saturate
   movl ← 1
else
   movl ← 0
temp0:63 ← EXTS(ACC32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding odd-numbered half-word signed fractional elements in parameters a and b are multiplied, producing a 32-bit product. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then added to the corresponding word in the accumulator, saturating if overflow or underflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 140. Odd form of vector half-word multiply (__ev_mhossfaaw)

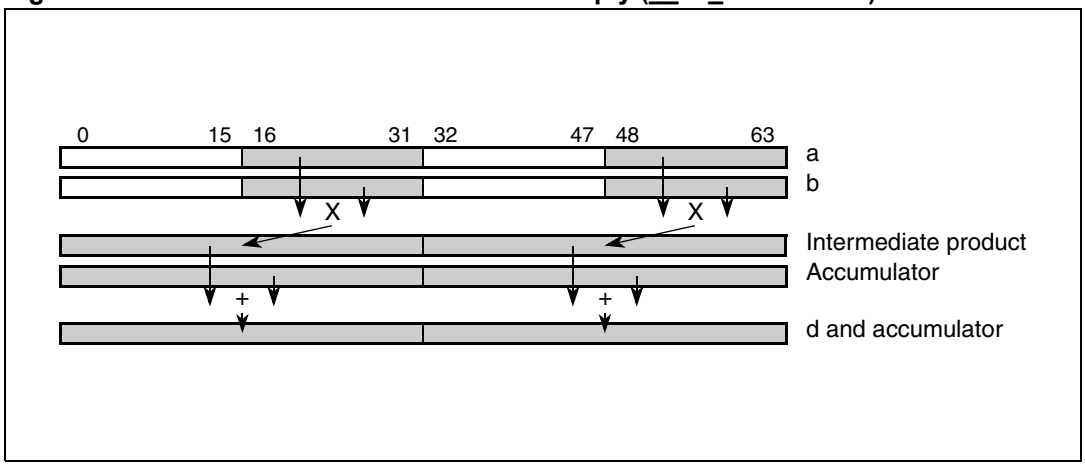


Table 146. __ev_mhossfaaw (registers altered by)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhossfaaw d,a,b**

__ev_mhossfanw

Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words

d = __ev_mhossfanw (a,b)

```
// high
temp0:31 ← a16:31 ×sf b16:31
if (a16:31 = 0x8000) & (b16:31 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF //saturate
   movh ← 1
else
   movh ← 0
temp0:63 ← EXTS(ACC0:31) - EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← a48:63 ×sf b48:63
if (a48:63 = 0x8000) & (b48:63 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF //saturate
   movl ← 1
else
   movl ← 0
temp0:63 ← EXTS(ACC32:63) - EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding odd-numbered half-word signed fractional elements in parameters a and b are multiplied, producing a 32-bit product. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then subtracted from the corresponding word in the accumulator, saturating if overflow or underflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from either the multiply or the subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 141. Odd form of vector half-word multiply (__ev_mhossfanw)

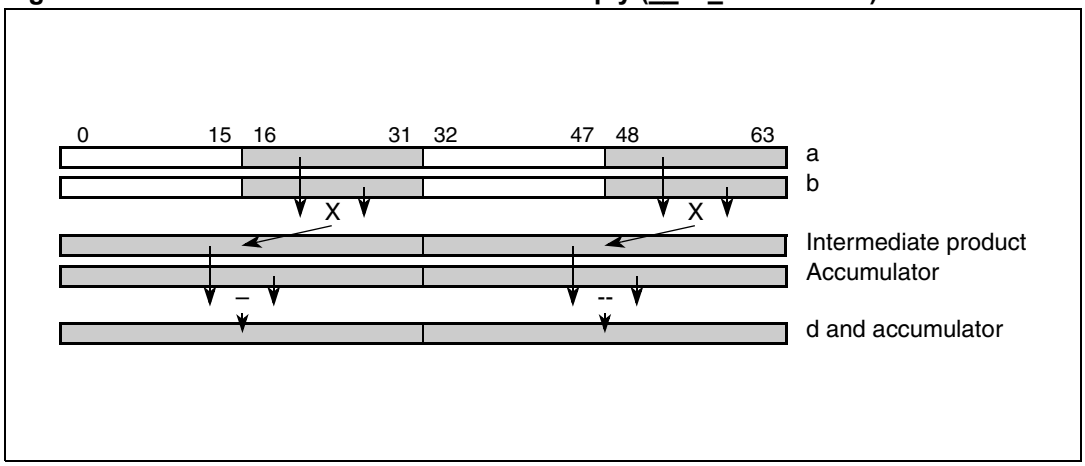


Table 147. __ev_mhossfanw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhossfanw d,a,b**

__ev_mhossiaaw

Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate into Words

d = __ev_mhossiaaw (a,b)

```
// high
temp0:31 ← a16:31 ×si b16:31
temp0:63 ← EXTS(ACC0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← a48:63 ×si b48:63
temp0:63 ← EXTS(ACC32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The corresponding odd-numbered half-word signed integer elements in parameters a and b are multiplied, producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator, saturating if overflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 142. Odd form of vector half-word multiply (__ev_mhossiaaw)

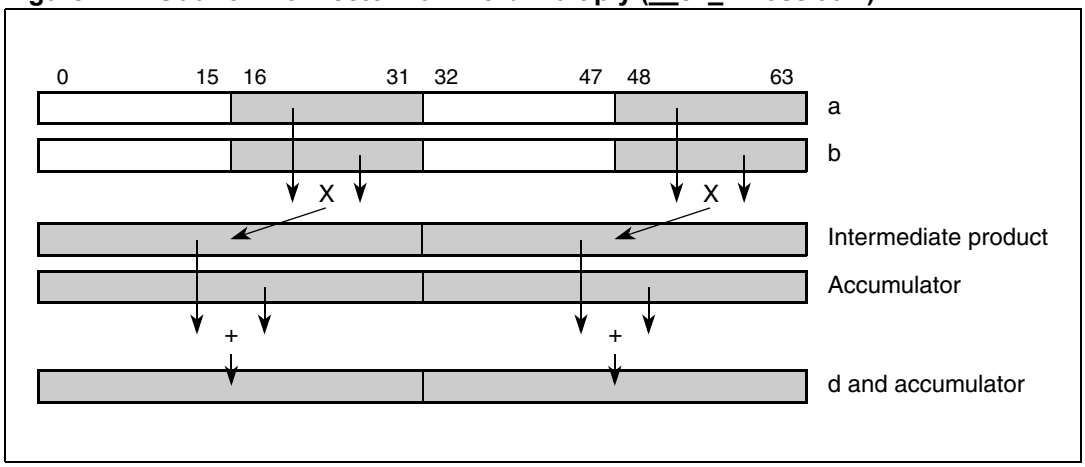


Table 148. __ev_mhossiaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhossiaaw d,a,b**

__ev_mhossianw

Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate Negative into Words

d = __ev_mhossianw (a,b)

```
// high
temp0:31 ← a16:31 ×si b16:31
temp0:63 ← EXTS(ACC0:31) - EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← a48:63 ×si b48:63
temp0:63 ← EXTS(ACC32:63) - EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The corresponding odd-numbered half-word signed integer elements in parameter a and b are multiplied, producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator, saturating if overflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from the subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 143. Odd form of vector half-word multiply (__ev_mhossianw)

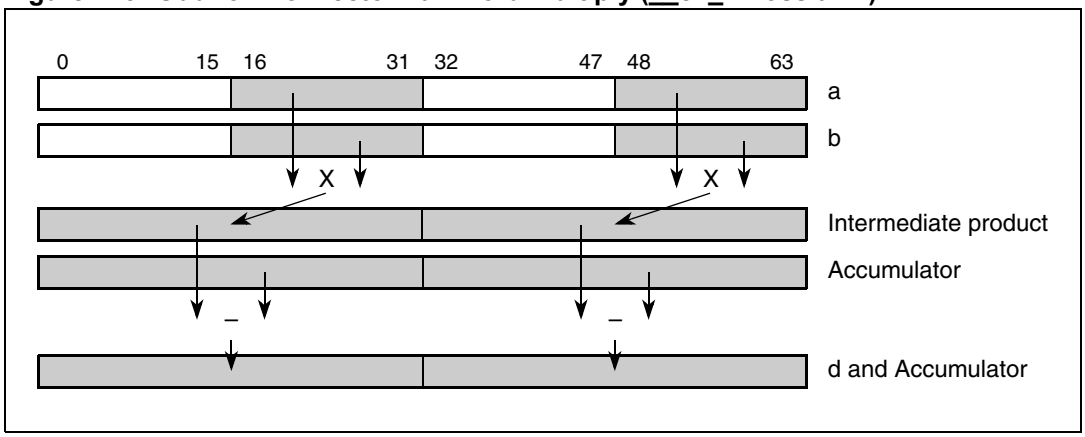


Table 149. __ev_mhossianw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhossianw d,a,b**

__ev_mhoumf

Vector Multiply Half Words, Odd, Unsigned, Modulo, Fractional (to Accumulator)

d = __ev_mhoumf (a,b) (A = 0)

d = __ev_mhoumfa (a,b) (A = 1)

// high
$d_{0:31} \leftarrow a_{16:31} \times_{ui} b_{16:31}$
// low
$d_{32:63} \leftarrow a_{48:63} \times_{ui} b_{48:63}$
// update accumulator
if A = 1, $ACC_{0:63} \leftarrow d_{0:63}$

The corresponding odd-numbered half-word elements in parameters a and b are multiplied. The two 32-bit products are placed into the corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

Figure 144. Vector multiply half words, odd, unsigned, modulo, fractional (to accumulator) (__ev_mhoumf)

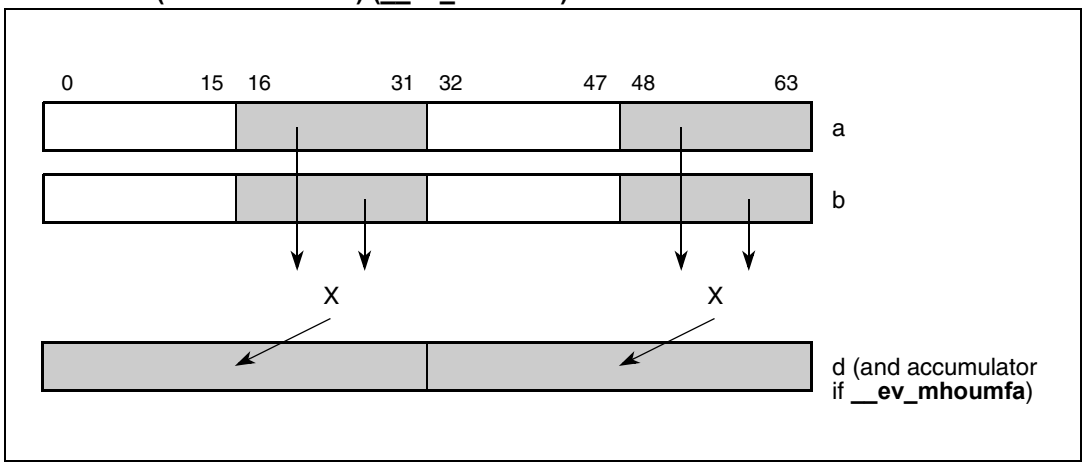


Table 150. __ev_mhoumf (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhoumi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhoumia d,a,b**

__ev_mhoumi

Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer (to Accumulator)

d = __ev_mhoumi (a,b) (A = 0)

d = __ev_mhoumia (a,b))A = 1)

```
// high
d0:31 ← a16:31 ×ui b16:31
// low
d32:63 ← a48:63 ×ui b48:63
// update accumulator
if A = 1, then ACC0:63 ← d0:63
```

The corresponding odd-numbered half-word unsigned integer elements in parameters a and b are multiplied. The two 32-bit products are placed into the corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

Figure 145. Vector multiply half words, odd, unsigned, modulo, integer (to Accumulator) (__ev_mhoumi)

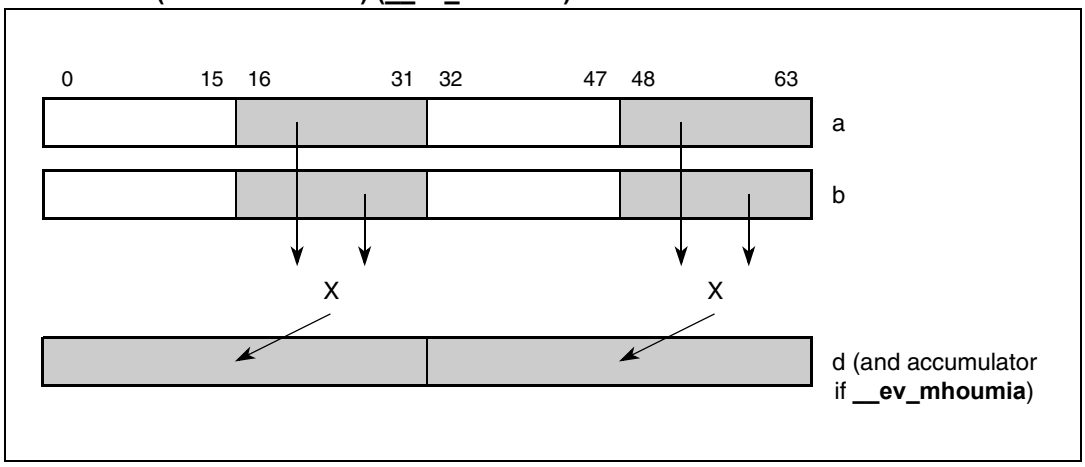


Table 151. __ev_mhoumi (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhoumi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhoumia d,a,b**

__ev_mhoumfaaw

Vector Multiply Half Words, Odd, Unsigned, Modulo, Fractional and Accumulate into Words

d = __ev_mhoumfaaw (a,b)

```
// high
temp0[0:31] ← a[16:31] ×ui b[16:31]
d[0:31] ← ACC[0:31] + temp0[0:31]
// low
temp1[0:31] ← a[48:63] ×ui b[48:63]
d[32:63] ← ACC[32:63] + temp1[0:31]
// update accumulator
ACC[0:63] ← d[0:63]
```

For each word element in the accumulator, the corresponding odd-numbered half-word elements in parameters a and b are multiplied. Each intermediate product is added to the contents of the corresponding accumulator word. The sums are placed into the corresponding parameter d and accumulator words.

Other registers altered: ACC

Figure 146. Odd form of vector half-word multiply (__ev_mhoumfaaw)

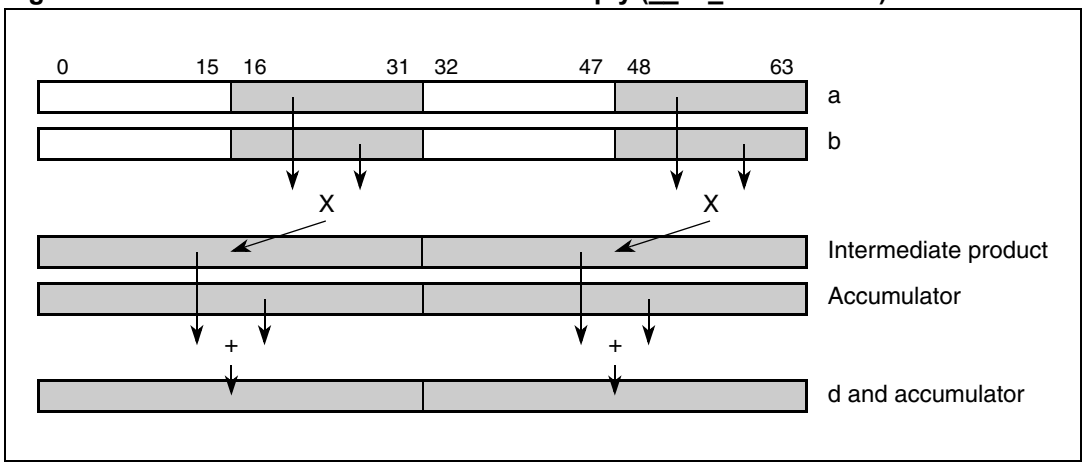


Table 152. __ev_mhoumfaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhoumiaaw d,a,b**

__ev_mhoumiaaw

Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate into Words

d = __ev_mhoumiaaw (a,b)

```
// high
temp0:31 ← a16:31 ×ui b16:31
d0:31 ← ACC0:31 + temp0:31

// low
temp0:31 ← a48:63 ×ui b48:63
d32:63 ← ACC32:63 + temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding odd-numbered half-word unsigned integer elements in parameters a and b are multiplied. Each intermediate product is added to the contents of the corresponding accumulator word. The sums are placed into the corresponding parameter d and accumulator words.

Other registers altered: ACC

Figure 147. Odd form of vector half-Word multiply (__ev_mhoumiaaw)

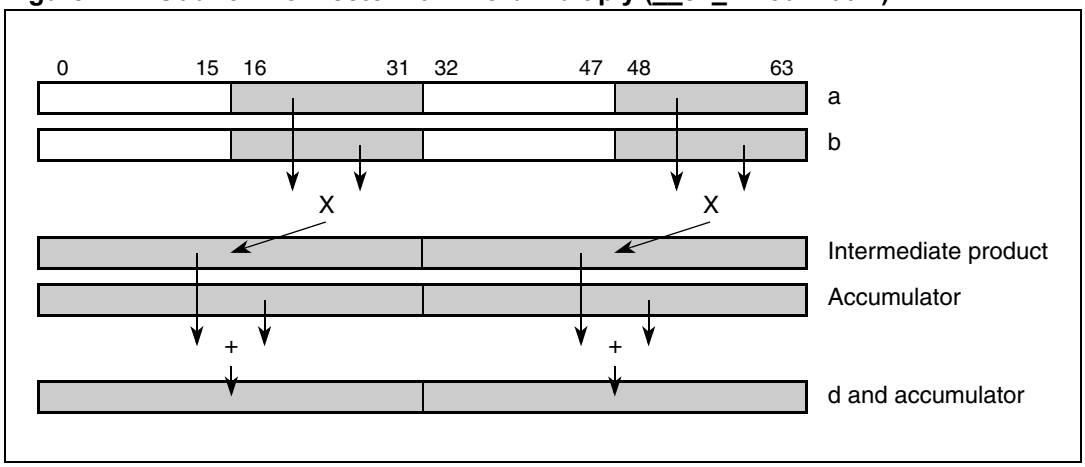


Table 153. __ev_mhoumiaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhoumiaaw d,a,b**

__ev_mhoumfanw

Vector Multiply Half Words, Odd, Unsigned, Modulo, Fractional and Accumulate Negative into Words

d = __ev_mhoumfanw (a,b)

```
// high
temp0 0:31 ← a 0:15 ×ui b 0:15
d 0:31 ← ACC 0:31 - temp0 0:31
// low
temp1 0:31 ← a 32:47 ×ui b 32:47
d 32:63 ← ACC 32:63 - temp1 0:31
// update accumulator
ACC 0:63 ← d 0:63
```

For each word element in the accumulator, the corresponding odd-numbered half word elements in parameters a and b are multiplied. Each intermediate product is subtracted from the contents of the corresponding accumulator word. The results are placed into the corresponding parameter d and accumulator words.

Other registers altered: ACC

Figure 148. Odd form of vector half-word multiply (__ev_mhoumfanw)

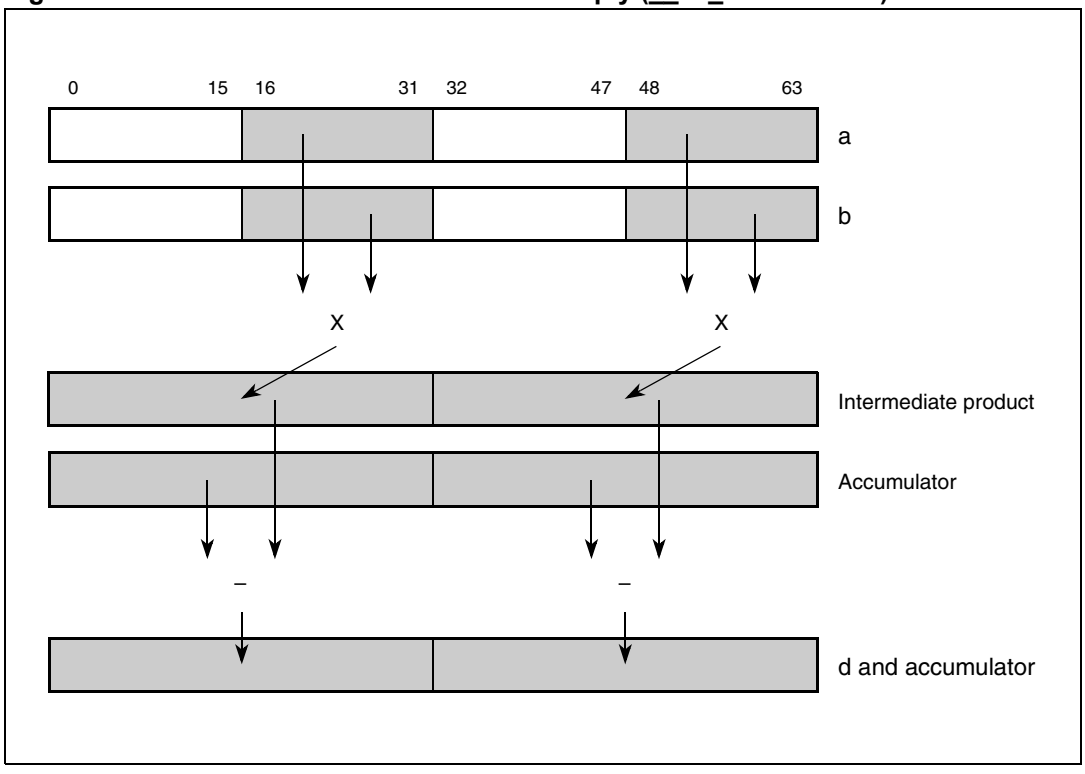


Table 154. __ev_mhoumfanw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhoumianw d,a,b**

__ev_mhoumianw

Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words

d = __ev_mhoumianw (a,b)

```
// high
temp0:31 ← a0:15 ×ui b0:15
d0:31 ← ACC0:31 - temp0:31
/
/ low
temp0:31 ← a32:47 ×ui b32:47
d32:63 ← ACC32:63 - temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding odd-numbered half-word unsigned integer elements in parameters a and b are multiplied. Each intermediate product is subtracted from the contents of the corresponding accumulator word. The results are placed into the corresponding parameter d and accumulator words.

Other registers altered: ACC

Figure 149. Odd form of vector half-word multiply (__ev_mhoumianw)

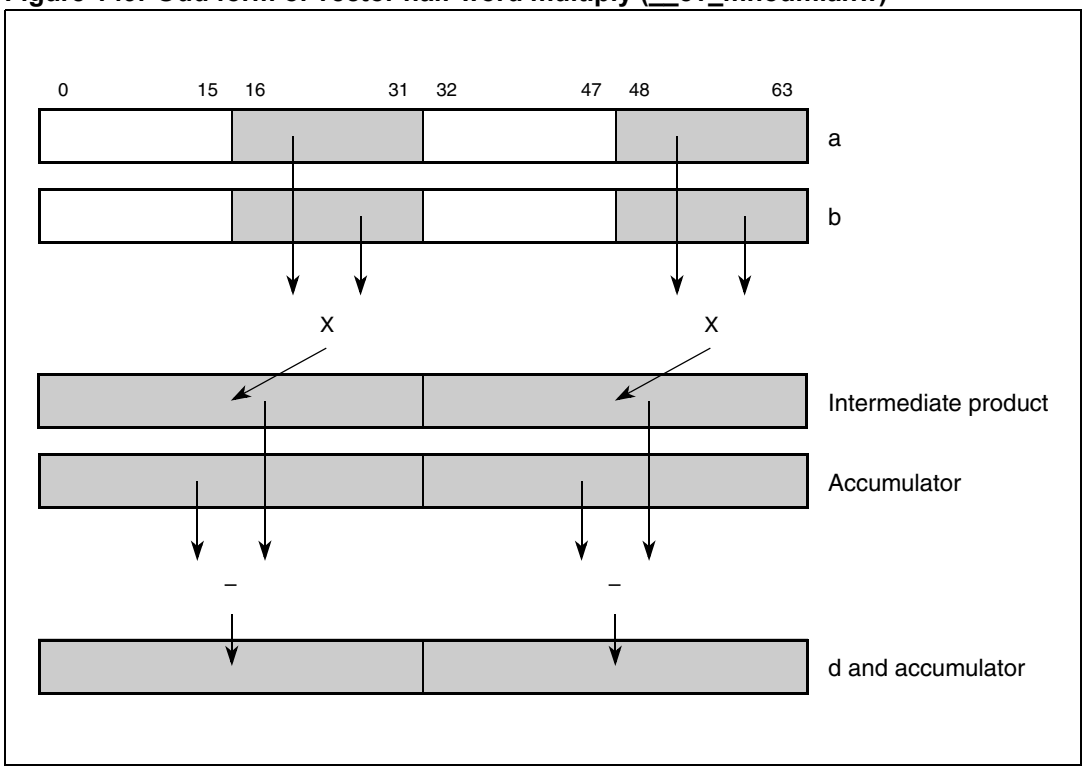


Table 155. __ev_mhoumianw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhoumianw d,a,b**

__ev_mhousfaaw

Vector Multiply Half Words, Odd, Unsigned, Saturate, Fractional and Accumulate into Words

d = __ev_mhousfaaw (a,b)

```
// high
temp0[0:31] ← a[16:31] ×ui b[16:31]
temp0[0:63] ← EXTZ(ACC[0:31]) + EXTZ(temp0[0:31])
if temp0[31] = 1
   d[0:31] ← 0xFFFF_FFFF //overflow
   ovh ← 1
else
   d[0:31] ← temp0[32:63]
   ovh ← 0
//low
temp1[0:31] ← a[48:63] ×ui b[48:63]
temp1[0:63] ← EXTZ(ACC[32:63]) + EXTZ(temp1[0:31])
if temp1[31] = 1
   d[32:63] ← 0xFFFF_FFFF //overflow
   ovl ← 1
else
   d[32:63] ← temp1[32:63]
   ovl ← 0
// update accumulator
ACC[0:63] ← d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ← ovh
SPEFSCR[OV] ← ovl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | ovh
SPEFSCR[SOV] ← SPEFSCR[SOV] | ovl
```

For each word element in the accumulator, the corresponding odd-numbered half word elements in parameters a and b are multiplied. Each product is added to the corresponding accumulator word contents. If a sum overflows, the appropriate saturation value is placed into the corresponding parameter d and accumulator words. Otherwise, the sums are placed there. The SPEFSCR records overflow or summary overflow information.

Other registers altered: SPEFSCR ACC

Figure 150. Odd form of vector half word multiply (__ev_mhousfaaw)

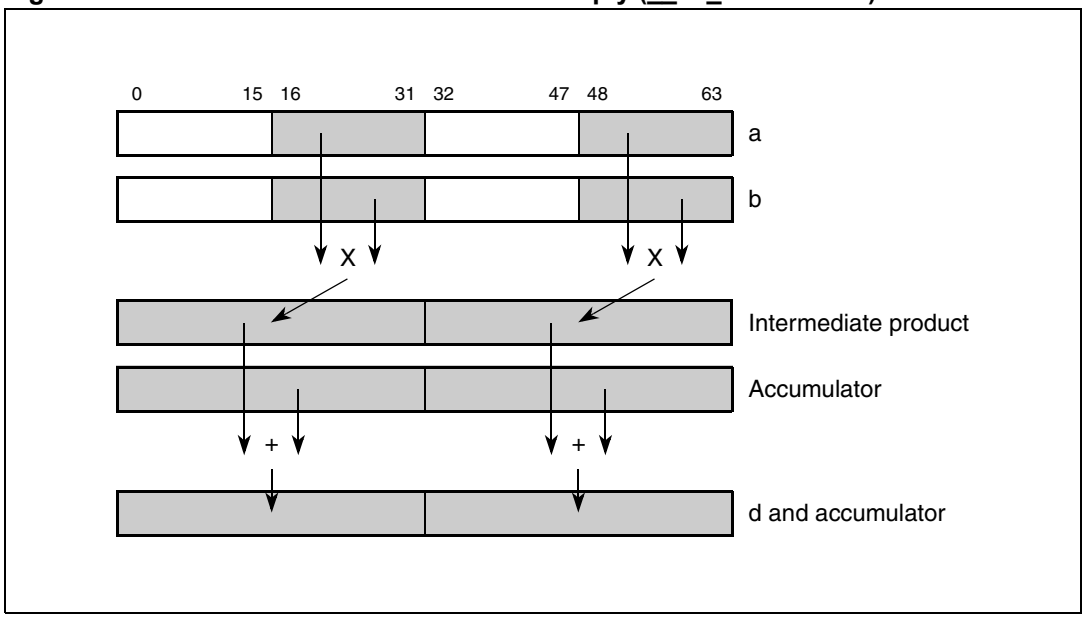


Table 156. __ev_mhousfaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhousiaaw d,a,b**

__ev_mhousiaaw

Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate into Words

d = __ev_mhousiaaw (a,b)

```
// high
temp0:31 ← a16:31 ×ui b16:31
temp0:63 ← EXTZ(ACC0:31) + EXTZ(temp0:31)
ovh ← temp31
d0:31 ← SATURATE(ovh, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temp32:63)

//low
temp0:31 ← a48:63 ×ui b48:63
temp0:63 ← EXTZ(ACC32:63) + EXTZ(temp0:31)
ovl ← temp31
d32:63 ← SATURATE(ovl, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding odd-numbered half-word unsigned integer elements in parameters a and b are multiplied, producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator, saturating if overflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow from the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 151. Odd form of vector half word multiply (__ev_mhousiaaw)

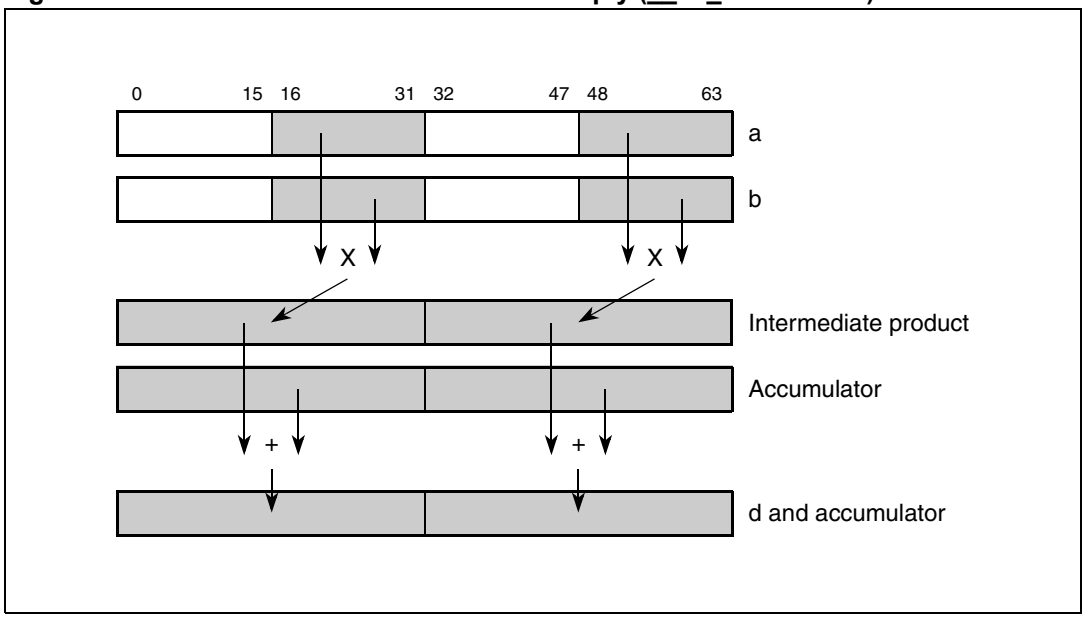


Table 157. __ev_mhousiaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhousiaaw d,a,b**

__ev_mhousfanw

Vector Multiply Half Words, Odd, Unsigned, Saturate, Fractional and Accumulate Negative into Words

d = __ev_mhousfanw (a,b)

```
// high
temp0[0:31] ← a[16:31] ×ui b[16:31]
temp0[0:63] ← EXTZ(ACC[0:31]) - EXTZ(temp0[0:31])
if temp0[31] = 1
   d[0:31] ← 0xFFFF_FFFF //overflow
   ovh ← 1
else
   d[0:31] ← temp0[32:63]
   ovh ← 0
//low
temp1[0:31] ← a[48:63] ×ui b[48:63]
temp1[0:63] ← EXTZ(ACC[32:63]) - EXTZ(temp1[0:31])
if temp1[31] = 1
   d[32:63] ← 0xFFFF_FFFF //overflow
   ovl ← 1
else
   d[32:63] ← temp1[32:63]
   ovl ← 0
// update accumulator
ACC[0:63] ← d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ← ovh
SPEFSCR[OV] ← ovl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | ovh
SPEFSCR[SOV] ← SPEFSCR[SOV] | ovl
```

For each word element in the accumulator, the corresponding odd-numbered half word elements in parameters a and b are multiplied. Each product is subtracted from the accumulator word contents. If a result overflows, the appropriate saturation value is placed into the corresponding parameter d and accumulator words. Otherwise, the sums are placed there. The SPEFSCR records overflow or summary overflow information.

Other registers altered: SPEFSCR ACC

Figure 152. Odd form of vector half word multiply (__ev_mhousfanw)

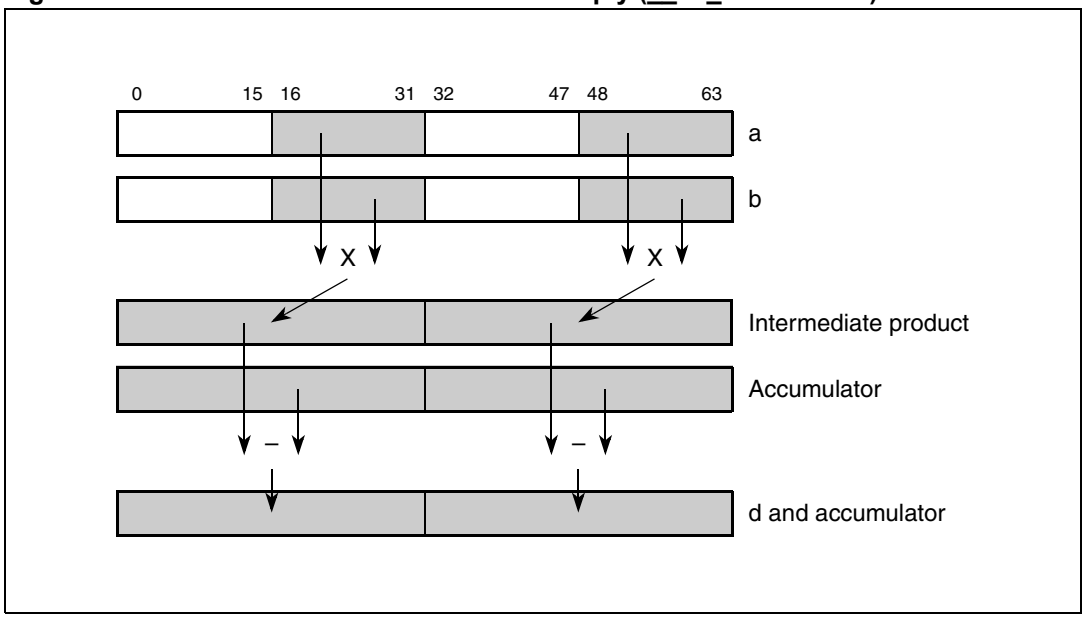


Table 158. __ev_mhousfanw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhousianw d,a,b**

__ev_mhousianw

Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words

d = __ev_mhousianw (a,b)

```
// high
temp0:31 ← a16:31 ×ui b16:31
temp0:63 ← EXTZ(ACC0:31) - EXTZ(temp0:31)
ovh ← temp31
d0:31 ← SATURATE(ovh, 0, 0, 0, temp32:63)

//low
temp0:31 ← a48:63 ×ui b48:63
temp0:63 ← EXTZ(ACC32:63) - EXTZ(temp0:31)
ovl ← temp31
d32:63 ← SATURATE(ovl, 0, 0, 0, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding odd-numbered half-word unsigned integer elements in parameters a and b are multiplied, producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator, saturating if overflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow from the subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 153. Odd form of vector half word multiply (__ev_mhousianw)

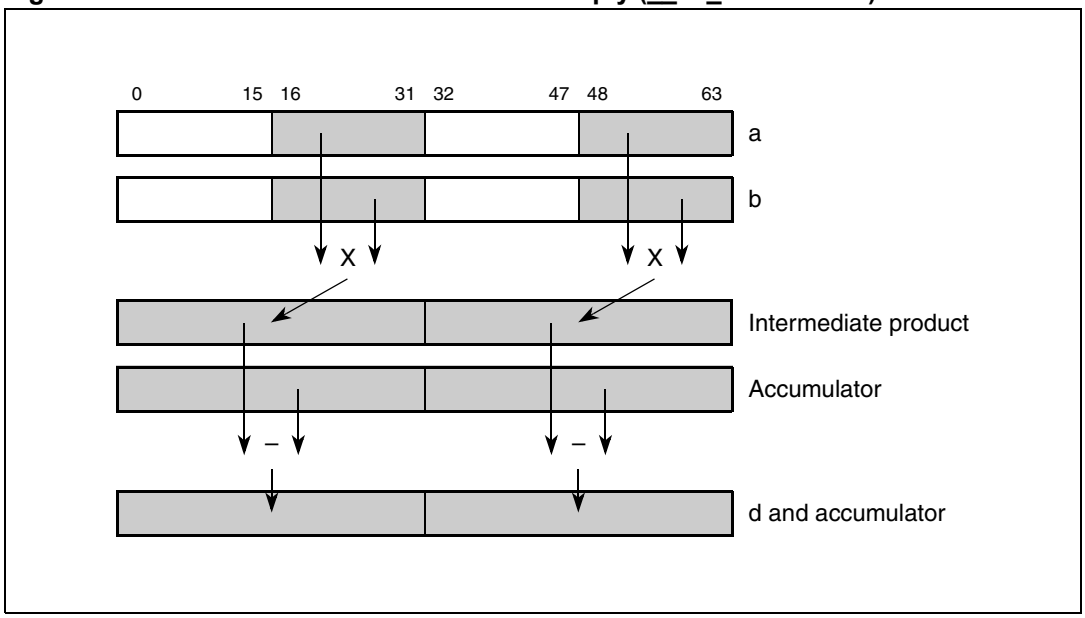


Table 159. __ev_mhousianw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhousianw d,a,b**

__ev_mra

Initialize Accumulator

d = __ev_mra (a)

$ACC_{0:63} \leftarrow a_{0:63}$

$d_{0:63} \leftarrow a_{0:63}$

The contents of parameter a are written into the accumulator and copied into parameter d. This is the method for initializing the accumulator.

Other registers altered: ACC

Figure 154. Initialize accumulator (__ev_mra)

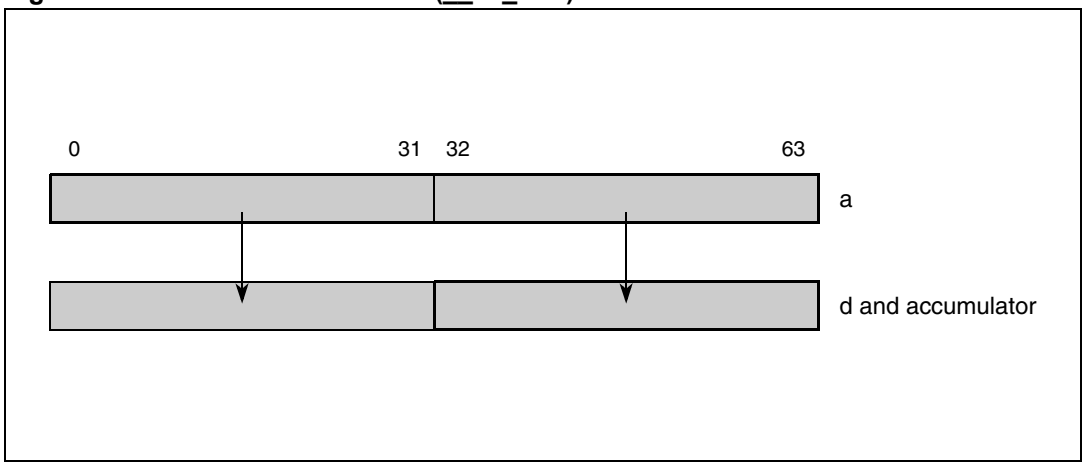


Table 160. __ev_mra (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evmra d,a**

__ev_mwhsmf

Vector Multiply Word High Signed, Modulo, Fractional (to Accumulator)

d = __ev_mwhsnf (a,b) (A = 0)

d = __ev_mwhsmfa (a,b))A = 1)

```
// high
temp0:63 ← a0:31 ×sf b0:31
d0:31 ← temp0:31

// low
temp0:63 ← a32:63 ×sf b32:63
d32:63 ← temp0:31

// update accumulator
if A = 1 then ACC0:63 ← d0:63
```

The corresponding word signed fractional elements in parameters a and b are multiplied, and bits 0–31 of the two products are placed into the two corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A =1)

Figure 155. Vector multiply word high signed, modulo, fractional (to accumulator) (__ev_mwhsmf)

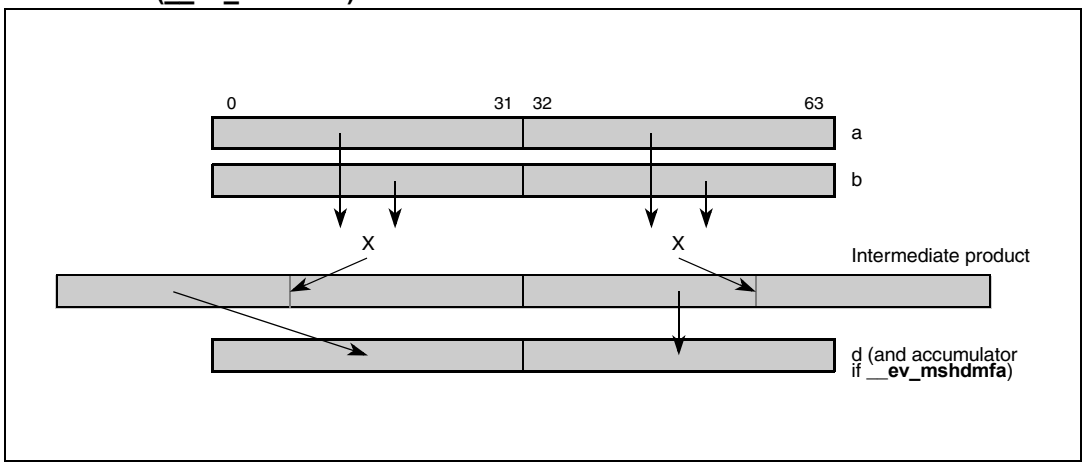


Table 161. __ev_mwhsmf (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhsmf d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhsmfa d,a,b**

__ev_mwhsmi

Vector Multiply Word High Signed, Modulo, Integer (to Accumulator)

d = __ev_mwhsmi (a,b) (A = 0)

d = __ev_mwhsmia (a,b))A = 1)

```
// high
temp0:63 ← a0:31 ×si b0:31
d0:31 ← temp0:31

// low
temp0:63 ← a32:63 ×si b32:63
d32:63 ← temp0:31

// update accumulator
if A = 1 then ACC0:63 ← d0:63
```

The corresponding word signed integer elements in parameters a and b are multiplied. Bits 0–31 of the two 64-bit products are placed into the two corresponding words of parameter d.

If A = 1,The result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

Figure 156. Vector multiply word high signed, modulo, integer (to Accumulator) (__ev_mwhsmi)

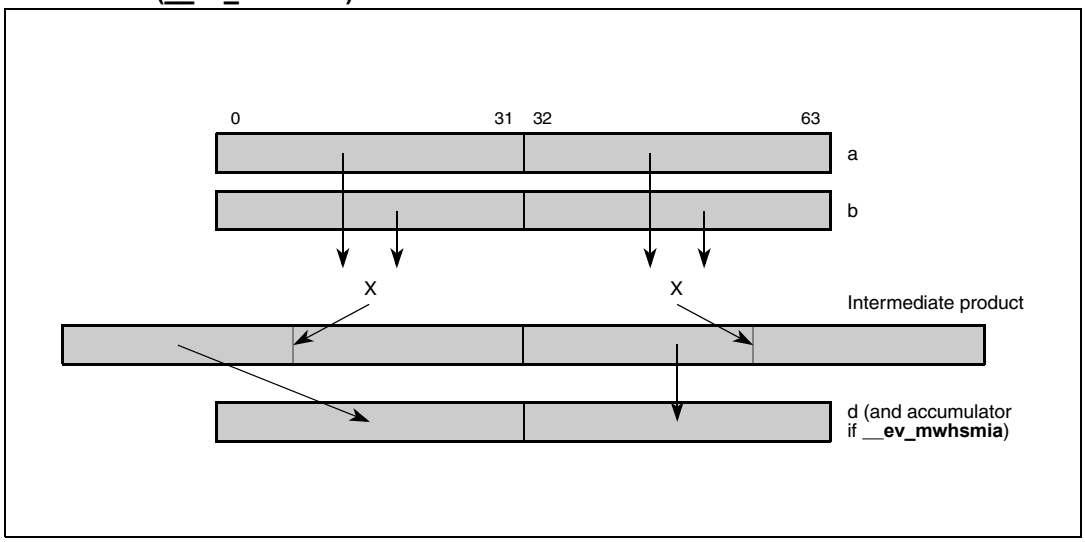


Table 162. __ev_mwhsmi (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhsmi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhsmia d,a,b**

__ev_mwhssf

Vector Multiply Word High Signed, Saturate, Fractional (to Accumulator)

d = __ev_mwhssf (a,b) (A = 0)

d = __ev_mwhssfa (a,b))A = 1)

```
// high
temp0:63 ← a0:31 ×sf b0:31
if (a0:31 = 0x8000_0000) & (b0:31 = 0x8000_0000) then
   d0:31 ← 0x7FFF_FFFF //saturate
   movh ← 1
else
   d0:31 ← temp0:31
   movh ← 0

// low
temp0:63 ← a32:63 ×sf b32:63
if (a32:63 = 0x8000_0000) & (b32:63 = 0x8000_0000) then
   d32:63 ← 0x7FFF_FFFF //saturate
   movl ← 1
else
   d32:63 ← temp0:31
   movl ← 0

// update accumulator
if A = 1 then ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding word signed fractional elements in parameters a and b are multiplied. Bits 0–31 of each product are placed into the corresponding words of parameter d. If both inputs are -1.0, the result saturates to the largest positive signed fraction and the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC (if A = 1)

Figure 157. Vector multiply word high signed, saturate, fractional (to Accumulator)(__ev_mwhssf)

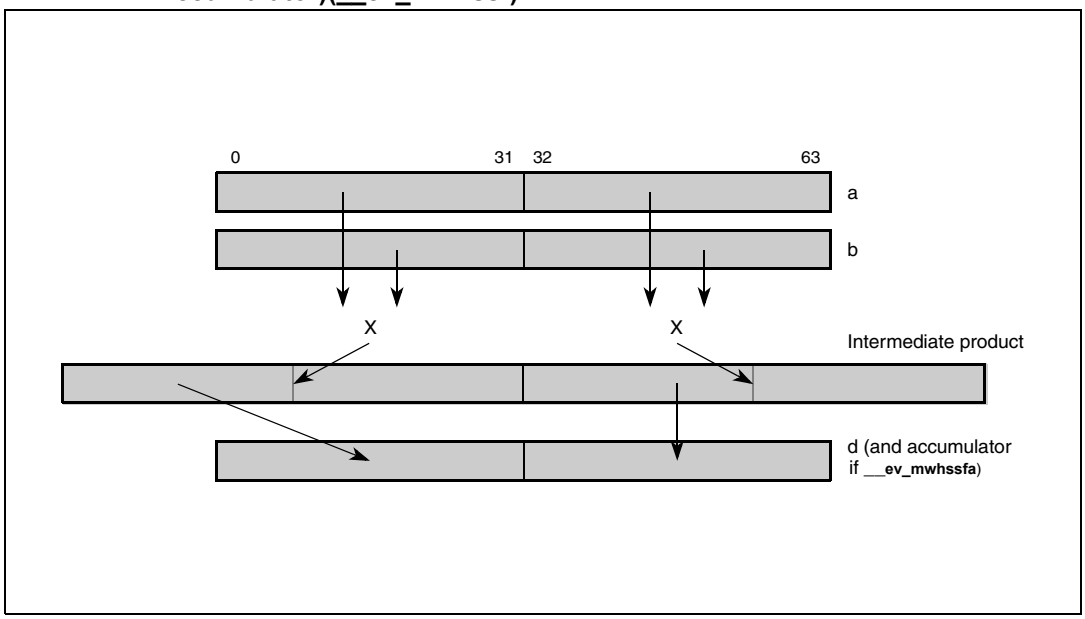


Table 163. __ev_mwhssf (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhssf d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhssfa d,a,b**

__ev_mwhumf

Vector Multiply Word High Unsigned, Modulo, Fractional (to Accumulator)

d = __ev_mwhumf (a,b) (A = 0)

d = __ev_mwhumfa (a,b))A = 1)

```
// high
temp0 0:63 ← a 0:31 ×ui b 0:31
d 0:31 ← temp0 0:31
// low
temp1 0:63 ← a 32:63 ×ui b 32:63
d 32:63 ← temp1 0:31
// update accumulator
if A = 1, ACC 0:63 ← d 0:63
```

The corresponding word unsigned integer elements in parameters a and b are multiplied. Bits 0–31 of the two products are placed into the two corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

Figure 158. Vector multiply word high unsigned, modulo, integer (to accumulator) (__ev_mwhumi)

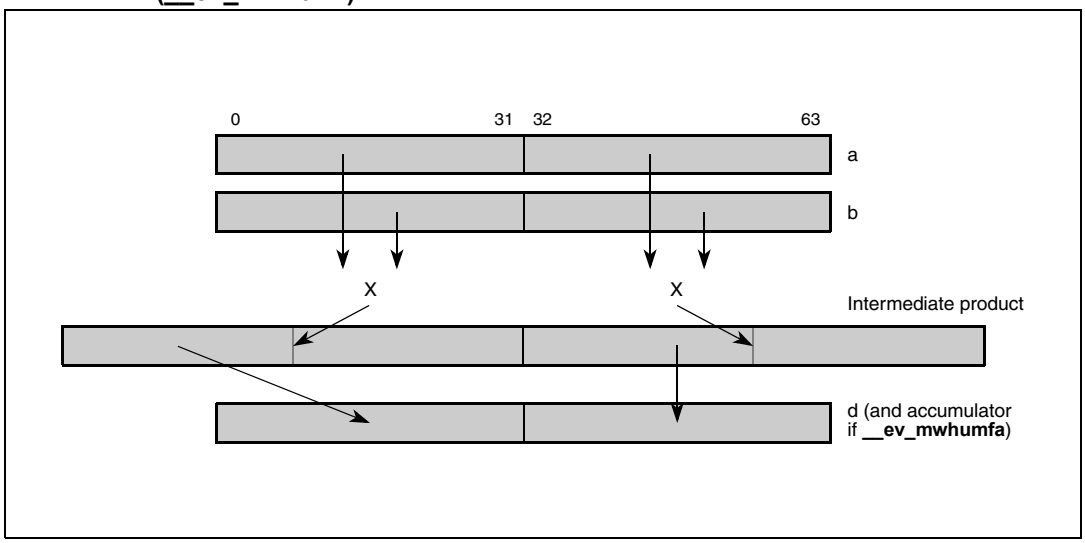


Table 164. __ev_mwhumf (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhumi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhumia d,a,b**

__ev_mwhumi

Vector Multiply Word High Unsigned, Modulo, Integer (to Accumulator)

d = __ev_mwhumi (a,b) (A = 0)

d = __ev_mwhumia (a,b) (A = 1)

```
// high
temp0:63 ← a0:31 ×ui b0:31
d0:31 ← temp0:31

// low
temp0:63 ← a32:63 ×ui b32:63
d32:63 ← temp0:31

// update accumulator
if A = 1, ACC0:63 ← d0:63
```

The corresponding word unsigned integer elements in parameters a and b are multiplied. Bits 0–31 of the two products are placed into the two corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

Figure 159. Vector multiply word high unsigned, modulo, integer (to accumulator) (__ev_mwhumi)

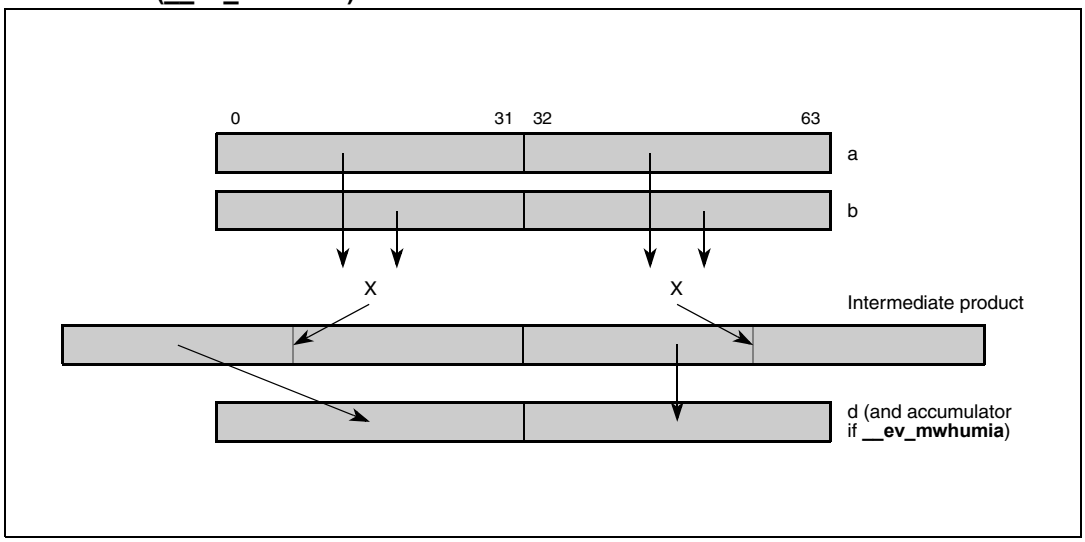


Table 165. __ev_mwhumi (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhumi d,a,b**

__ev_mwlsmiaaw

Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words

d = __ev_mwlsmiaaw (a,b)

```
// high
temp0:63 ← a0:31 ×si b0:31
d0:31 ← ACC0:31 + temp32:63

// low
temp0:63 ← a32:63 ×si b32:63
d32:63 ← ACC32:63 + temp32:63

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding word signed integer elements in parameters a and b are multiplied. The least significant 32 bits of each intermediate product is added to the contents of the corresponding accumulator words, and the result is placed into parameter d and the accumulator.

Other registers altered: ACC

Figure 160. Vector multiply word low signed, modulo, integer and accumulate in words (__ev_mwlsmiaaw)

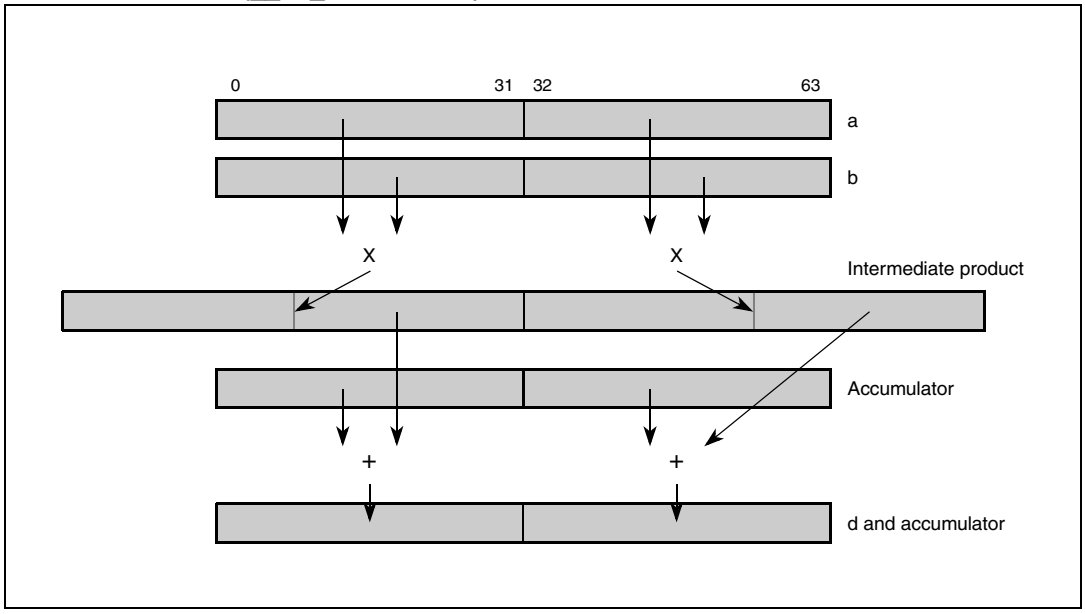


Table 166. __ev_mwlsmiaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlsmiaaw d,a,b**

__ev_mwlsmianw

Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words

d = __ev_mwlsmianw (a,b)

```
// high
temp0:63 ← a0:31 ×si b0:31
d0:31 ← ACC0:31 - temp32:63

// low
temp0:63 ← a32:63 ×si b32:63
d32:63 ← ACC32:63 - temp32:63

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding word elements in parameters a and b are multiplied. The least significant 32 bits of each intermediate product is subtracted from the contents of the corresponding accumulator words, and the result is placed in parameter d and the accumulator.

Other registers altered: ACC

Figure 161. Vector multiply word low signed, modulo, integer and accumulate negative in words (__ev_mwlsmianw)

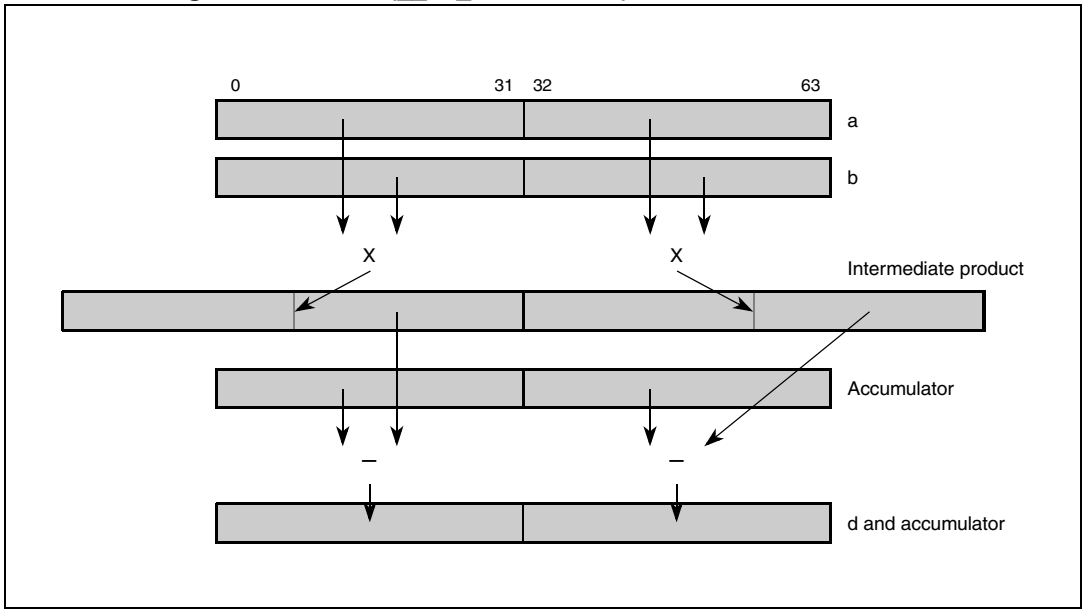


Table 167. __ev_mwlsmianw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlsmfanw d,a,b**

__ev_mwlssiaaw

Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words

d = __ev_mwlssiaaw (a,b)

```
// high
temp0:63 ← a0:31 ×si b0:31
temp0:63 ← EXTS(ACC0:31) + EXTS(temp32:63)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:63 ← a32:63 ×si b32:63
temp0:63 ← EXTS(ACC32:63) + EXTS(temp32:63)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The corresponding word signed integer elements in parameters a and b are multiplied, producing a 64-bit product. The least significant 32 bits of each product are then added to the corresponding word in the accumulator, saturating if overflow or underflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 162. Vector multiply word low signed, saturate, integer and accumulate in words (__ev_mwlssiaaw)

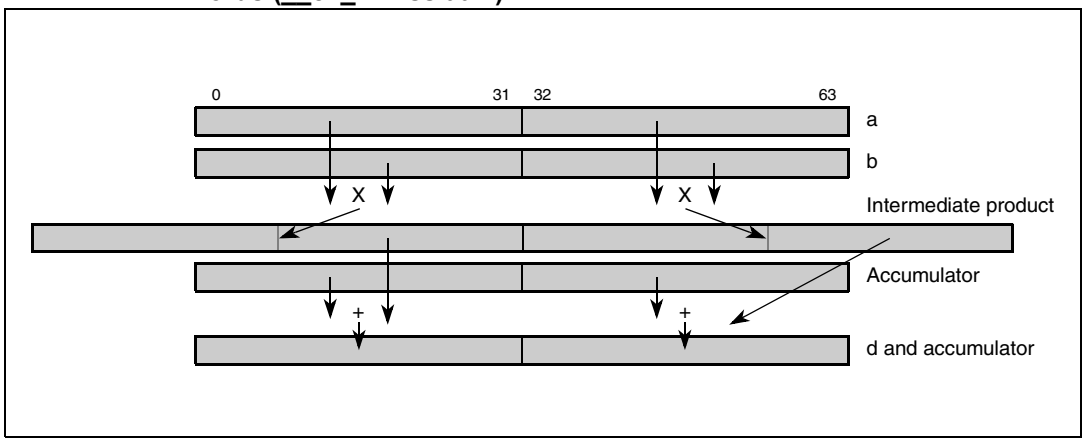


Table 168. __ev_mwlssiaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlssiaaw d,a,b**

__ev_mwlssianw

Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words

d = __ev_mwlssianw (a,b)

```
// high
temp0:63 ← a0:31 ×si b0:31
temp0:63 ← EXTS(ACC0:31) - EXTS(temp32:63)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:63 ← a32:63 ×si b32:63
temp0:63 ← EXTS(ACC32:63) - EXTS(temp32:63)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The corresponding word signed integer elements in parameters a and b are multiplied, producing a 64-bit product. The least significant 32 bits of each product are then subtracted from the corresponding word in the accumulator, saturating if overflow or underflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 163. Vector multiply word low signed, saturate, integer and accumulate negative in words (__ev_mwlssianw)

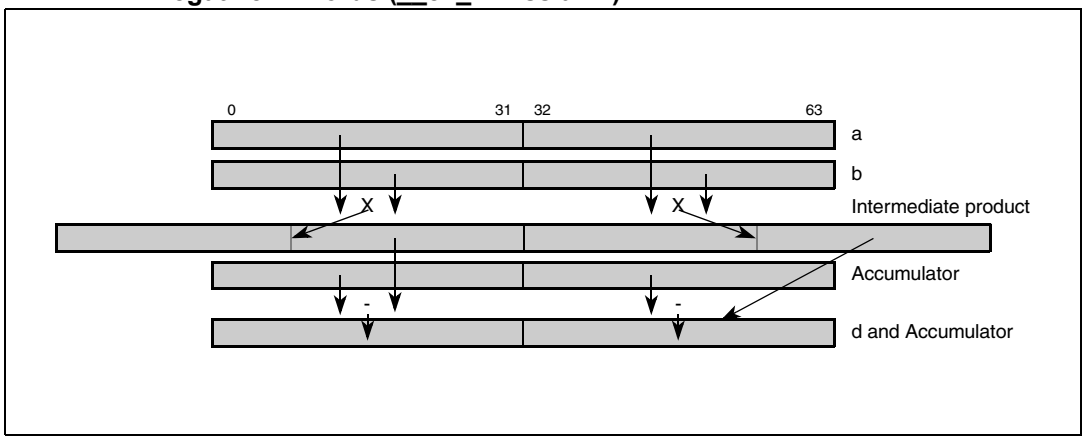


Table 169. __ev_mwlssianw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlssianw d,a,b**

__ev_mwlumi

Vector Multiply Word Low Unsigned, Modulo, Integer

d = __ev_mwlumi (a,b)

d = __ev_mwlumia (a,b)

```
// high
temp0:63 ← a0:31 ×ui b0:31
d0:31 ← temp32:63

// low
temp0:63 ← a32:63 ×ui b32:63
d32:63 ← temp32:63

// update accumulator
If A = 1 then ACC0:63 ← d0:63
```

The corresponding word unsigned integer elements in parameters a and b are multiplied. The least significant 32 bits of each product are placed into the two corresponding words of parameter d.

Note: The least significant 32 bits of the product are independent of whether the word elements in parameters a and b are treated as signed or unsigned 32-bit integers.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

*Note: The **evmwlumi** and **evmwlumia** can be used for signed or unsigned integers.*

Figure 164. Vector multiply word low unsigned, modulo, integer (__ev_mwlumi)

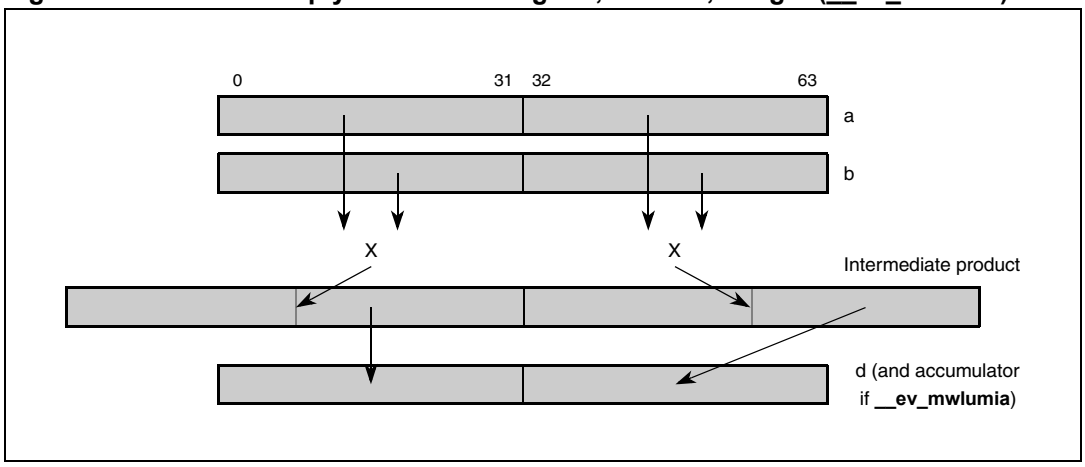


Table 170. __ev_mwlumi (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlumi d,a,b**
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlumia d,a,b**

__ev_mwlumiaaw

Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words

d = __ev_mwlumiaaw (a,b)

```
// high
temp0:63 ← a0:31 ×ui b0:31
d0:31 ← ACC0:31 + temp32:63

// low
temp0:63 ← a32:63 ×ui b32:63
d32:63 ← ACC32:63 + temp32:63

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding word unsigned integer elements in parameters a and b are multiplied. The least significant 32 bits of each product are added to the contents of the corresponding accumulator word, and the result is placed into the corresponding parameter d and accumulator word.

Other registers altered: ACC

Figure 165. Vector multiply word low unsigned, modulo, integer and accumulate in words (__ev_mwlumiaaw)

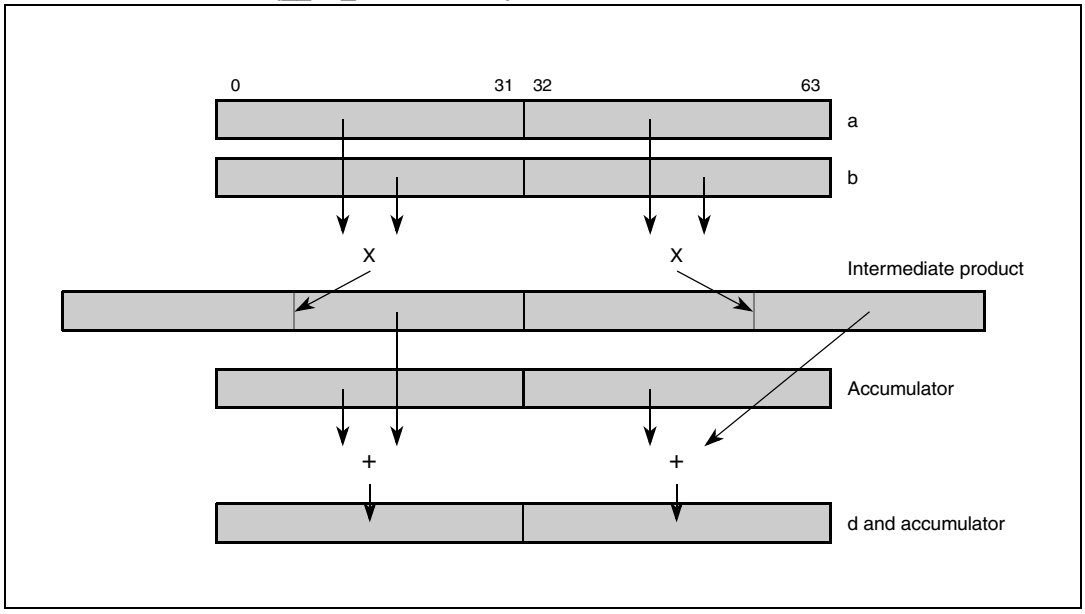


Table 171. __ev_mwlumiaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlumiaaw d,a,b**

__ev_mwlumianw

Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words

d = __ev_mwlumianw (a,b)

```
// high
temp0:63 ← a0:31 ×ui b0:31
d0:31 ← ACC0:31 - temp32:63

// low
temp0:63 ← a32:63 ×ui b32:63
d32:63 ← ACC32:63 - temp32:63

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding word unsigned integer elements in parameters a and b are multiplied. The least significant 32 bits of each product are subtracted from the contents of the corresponding accumulator word, and the result is placed into parameter d and the accumulator.

Other registers altered: ACC

Figure 166. Vector multiply word low unsigned, modulo, integer and accumulate negative in words (__ev_mwlumianw)

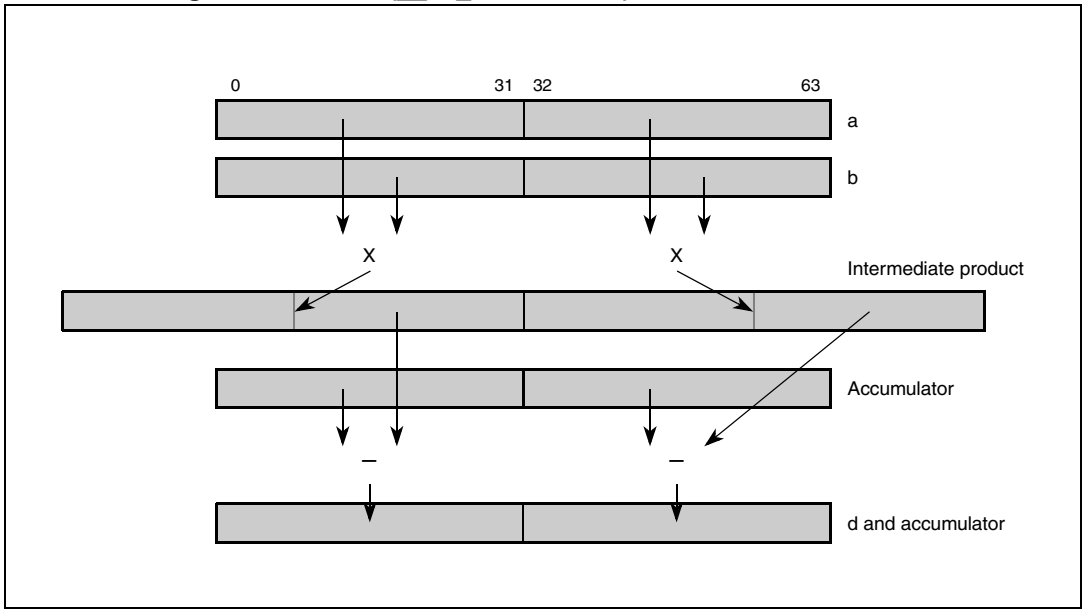


Table 172. __ev_mwlumianw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlumianw d,a,b**

__ev_mwlusiaaw

Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words

d = __ev_mwlusiaaw (a,b)

```
// high
temp0:63 ← a0:31 ×ui b0:31
temp0:63 ← EXTZ(ACC0:31) + EXTZ(temp32:63)
ovh ← temp31
d0:31 ← SATURATE(ovh, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temp32:63)

//low
temp0:63 ← a32:63 ×ui b32:63
temp0:63 ← EXTZ(ACC32:63) + EXTZ(temp32:63)
ovl ← temp31
d32:63 ← SATURATE(ovl, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding word unsigned integer elements in parameters a and b are multiplied, producing a 64-bit product. The least significant 32 bits of each product are then added to the corresponding word in the accumulator, saturating if overflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow from the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 167. Vector multiply word low unsigned, saturate, integer and accumulate in words (__ev_mwlusiaaw)

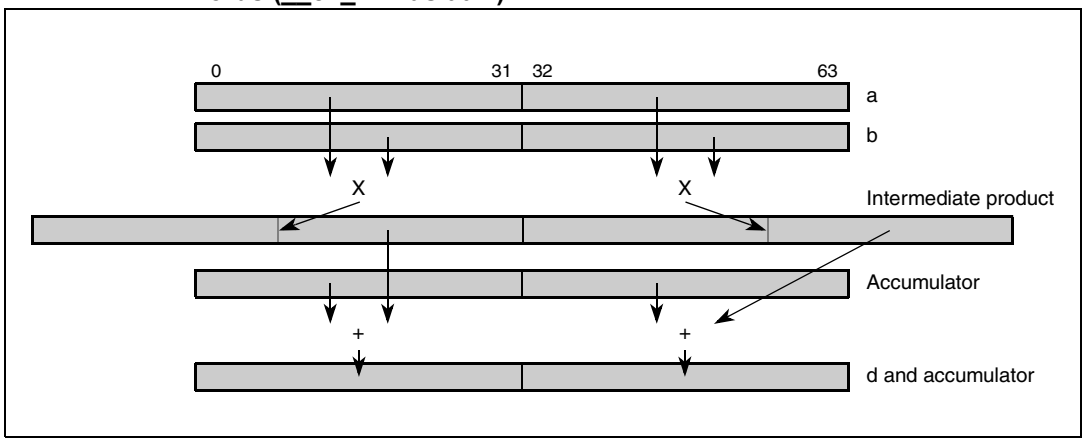


Table 173. __ev_mwlusiaaw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlusiaaw d,a,b**

__ev_mwlusianw

Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words

d = __ev_mwlusianw (a,b)

```
// high
temp0:63 ← a0:31 ×ui b0:31
temp0:63 ← EXTZ(ACC0:31) - EXTZ(temp32:63)
ovh ← temp31
d0:31 ← SATURATE(ovh, 0, 0x0000_0000, 0x0000_0000, temp32:63)

//low
temp0:63 ← a32:63 ×ui b32:63
temp0:63 ← EXTZ(ACC32:63) - EXTZ(temp32:63)
ovl ← temp31
d32:63 ← SATURATE(ovl, 0, 0x0000_0000, 0x0000_0000, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding word unsigned integer elements in parameters a and b are multiplied, producing a 64-bit product. The least significant 32 bits of each product are then subtracted from the corresponding word in the accumulator, saturating if underflow occurs, and the result is placed in parameter d and the accumulator.

If there is an underflow from the subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

Figure 168. Vector multiply word low unsigned, saturate, integer and accumulate negative in words (__ev_mwlusianw)

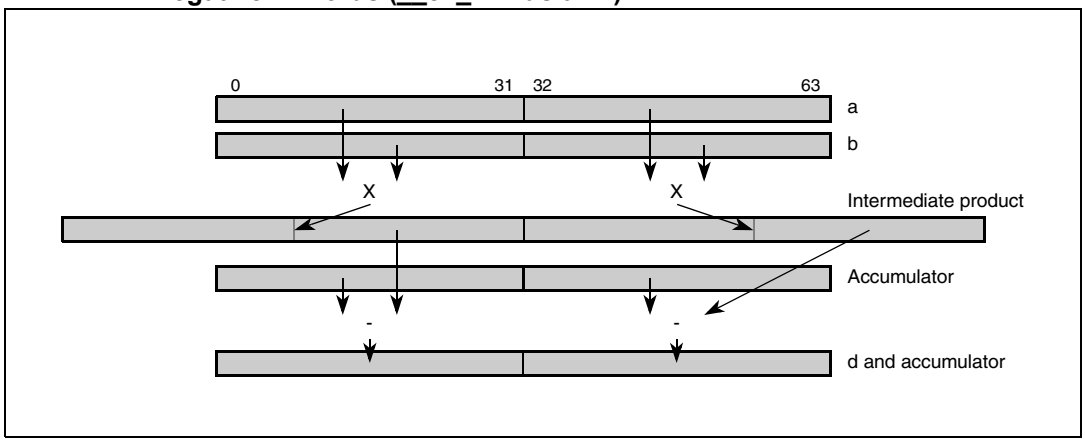


Table 174. __ev_mwlusianw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlusianw d,a,b**

__ev_mwsmf

Vector Multiply Word Signed, Modulo, Fractional (to Accumulator)

Table 175. __ev_mwsmf (registers altered by).

d = __ev_mwsmf (a,b)	(A = 0)
d = __ev_mwsmfa (a,b)	(A = 1)

$d_{0:63} \leftarrow a_{32:63} \times_{sf} b_{32:63}$

```
// update accumulator
if A = 1 then ACC0:63 ← d0:63
```

The corresponding low word signed fractional elements in parameters a and b are multiplied. The product is placed into parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

Figure 169. Vector multiply word signed, modulo, fractional (to Accumulator) (__ev_mwsmf)

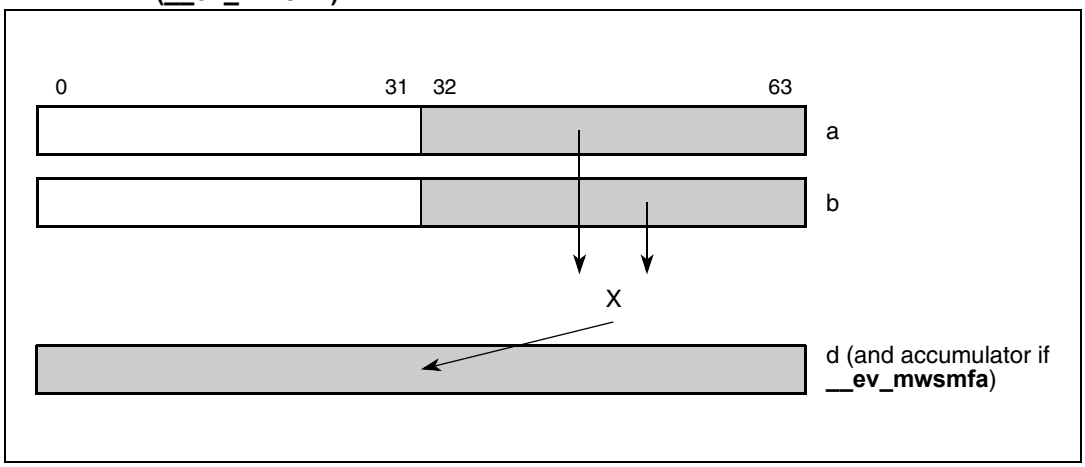


Table 176. __ev_mwsmf (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwsmf d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwsmfa d,a,b**

__ev_mwsmfaa

Vector Multiply Word Signed, Modulo, Fractional and Accumulate

d = __ev_mwsmfaa (a,b)

$temp_{0:63} \leftarrow a_{32:63} \times_{sf} b_{32:63}$
$d_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low word signed fractional elements in parameters a and b are multiplied. The intermediate product is added to the contents of the 64-bit accumulator and the result is placed in parameter d and the accumulator.

Other registers altered: ACC

Figure 170. Vector multiply word signed, modulo, fractional and Accumulate (__ev_mwsmfaa)

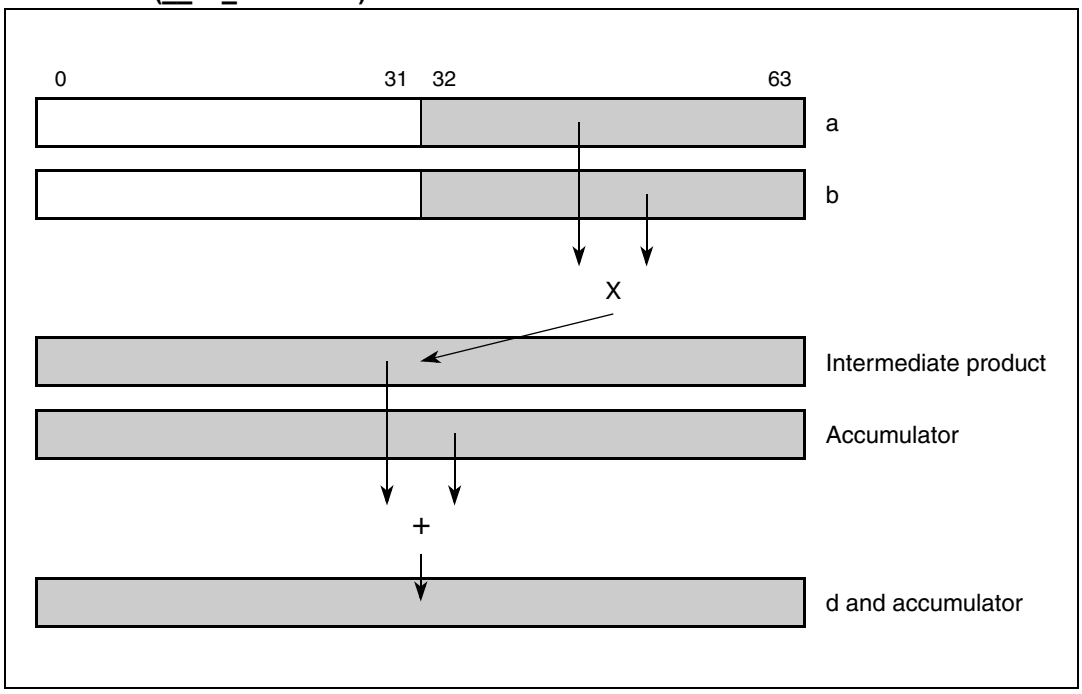


Table 177. __ev_mwsmfaa (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwsmfaa d,a,b**

__ev_mwsmfan

Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative

d = __ev_mwsmfan (a,b)

$temp_{0:63} \leftarrow a_{32:63} \times_{sf} b_{32:63}$
$d_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$

```
// update accumulator
```
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low word signed fractional elements in parameters a and b are multiplied. The intermediate product is subtracted from the contents of the accumulator, and the result is placed in parameter d and the accumulator.

Other registers altered: ACC

Figure 171. Vector multiply word signed, modulo, fractional, and accumulate Negative (__ev_mwsmfan)

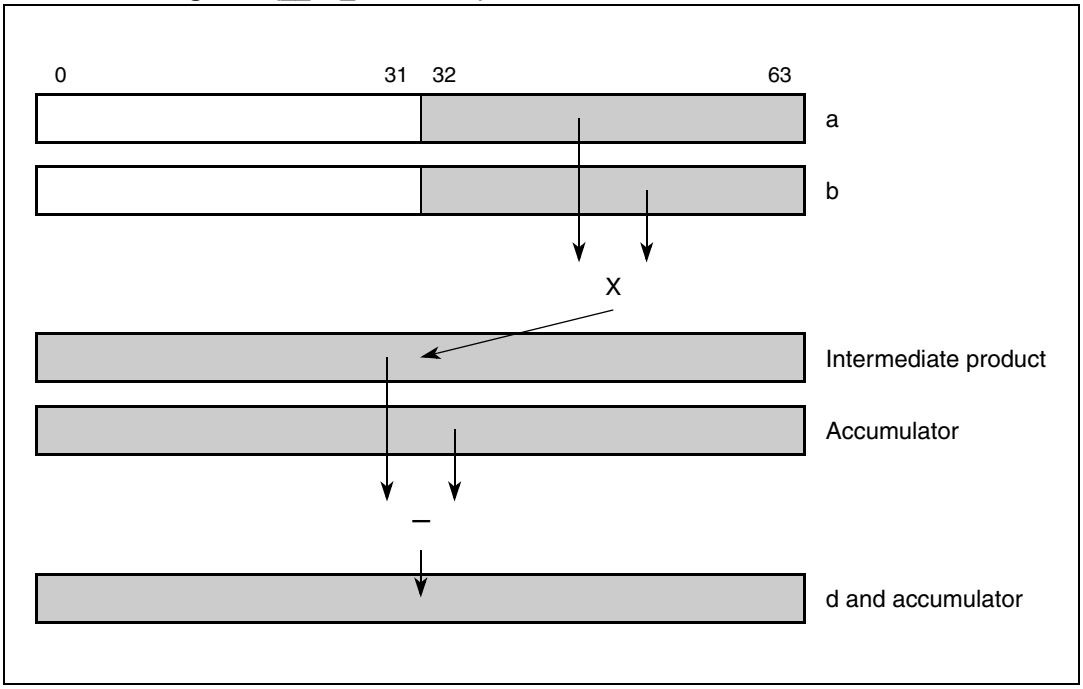


Table 178. __ev_mwsmfan (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwsmfan d,a,b**

__ev_mwsmi

Vector Multiply Word Signed, Modulo, Integer (to Accumulator)

d = __ev_mwsmi (a,b) (A = 0)

d = __ev_mwsmia (a,b))A = 1)

$d_{0:63} \leftarrow a_{32:63} \times_{si} b_{32:63}$

```
// update accumulator
if A = 1 then ACC0:63 ← d0:63
```

The low word signed integer elements in parameters a and b are multiplied. The product is placed into the parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

Figure 172. Vector multiply word signed, modulo, integer (to Accumulator) (__ev_mwsmi)

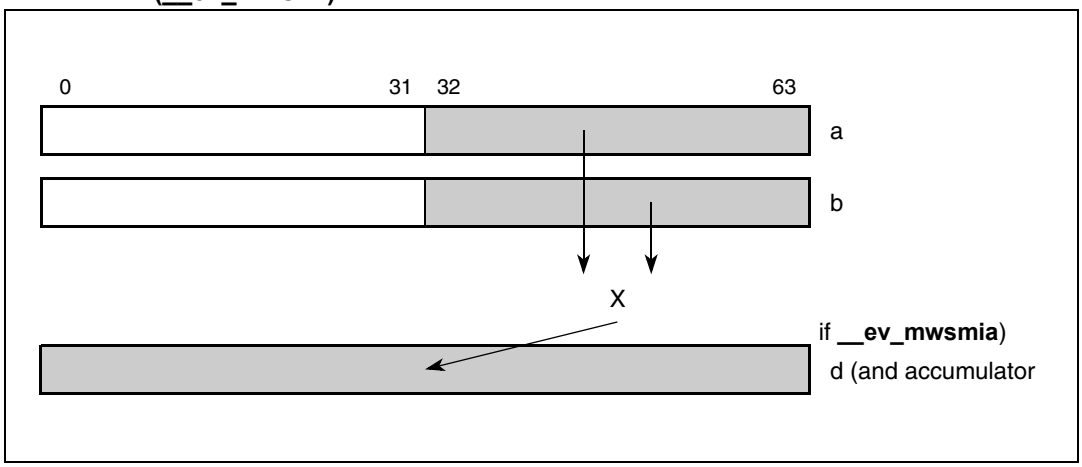


Table 179. __ev_mwsmi (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwsmi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwsmia d,a,b**

__ev_mwsmiaa

Vector Multiply Word Signed, Modulo, Integer and Accumulate

d = __ev_mwsmiaa (a,b)

$temp_{0:63} \leftarrow a_{32:63} \times_{si} b_{32:63}$
$d_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The low word signed integer elements in parameters a and b are multiplied. The intermediate product is added to the contents of the 64-bit accumulator, and the result is placed into parameter d and the accumulator.

Other registers altered: ACC

Figure 173. Vector multiply word signed, modulo, integer and accumulate (__ev_mwsmiaa)

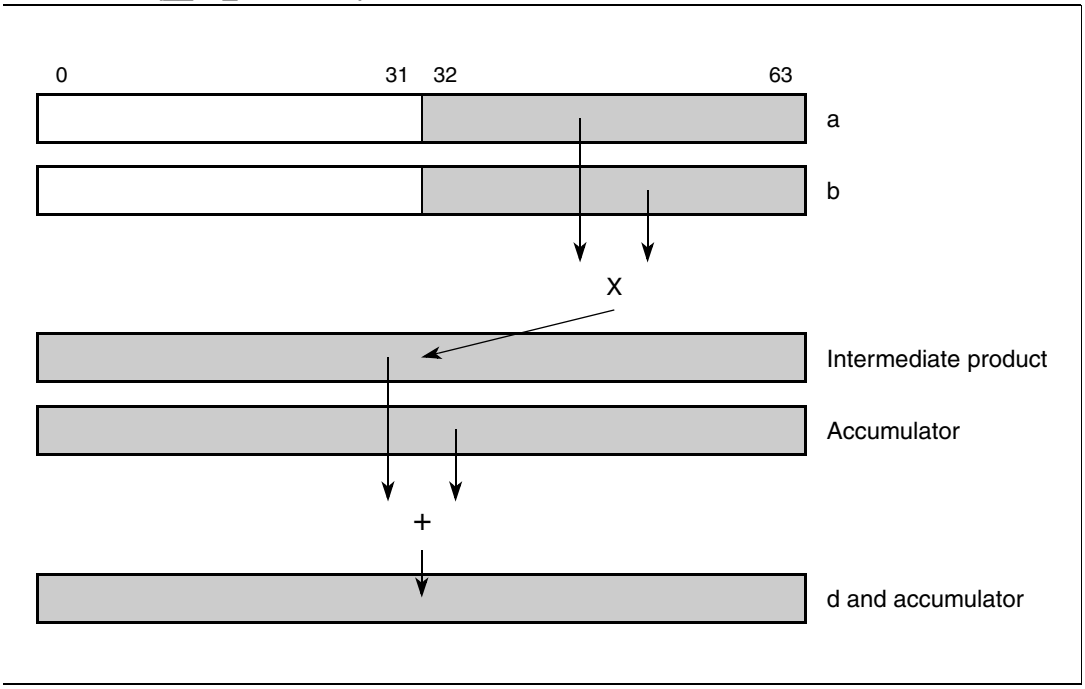


Table 180. __ev_mwsmiaa (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwsmiaa d,a,b**

__ev_mwsmian

Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative

d = __ev_mwsmian (a,b)

$temp_{0:63} \leftarrow a_{32:63} \times_{si} b_{32:63}$
$d_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The corresponding low word signed integer elements in parameters a and b are multiplied. The intermediate product is subtracted from the contents of the 64-bit accumulator and the result is placed into parameter d and the accumulator.

Other registers altered: ACC

Figure 174. Vector multiply word signed, modulo, integer and accumulate Negative (__ev_mwsmian)

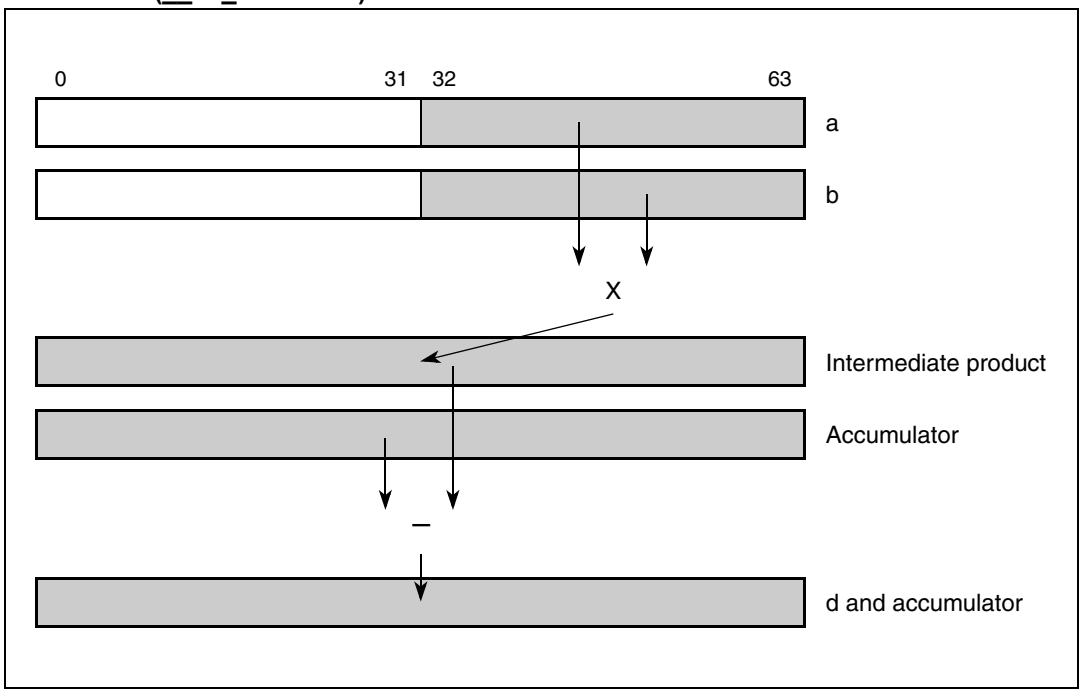


Table 181. __ev_mwsmian (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwsmian d,a,b**

__ev_mwssf

Vector Multiply Word Signed, Saturate, Fractional (to Accumulator)

d = __ev_mwssf (a,b) (A = 0)

d = __ev_mwssfa (a,b))A = 1)

```
temp0:63 ← a32:63 ×sf b32:63
if (a32:63 = 0x8000_0000) & (b32:63 = 0x8000_0000) then
   d0:63 ← 0x7FFF_FFFF_FFFF_FFFF //saturate
   mov ← 1
else
   d0:63 ← temp0:63
   mov ← 0

// update accumulator
if A = 1 then ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← 0
SPEFSCROV ← mov
SPEFSCRSOV ← SPEFSCRSOV | mov
```

The low word signed fractional elements in parameters a and b are multiplied. The 64-bit product is placed into parameter d. If both inputs are -1.0, the result saturates to the largest positive signed fraction, and the overflow and summary overflow bits are recorded in the SPEFSCR.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: SPEFSCR ACC (if A = 1)

Figure 175. Vector multiply word signed, saturate, fractional (to Accumulator) (__ev_mwssf)

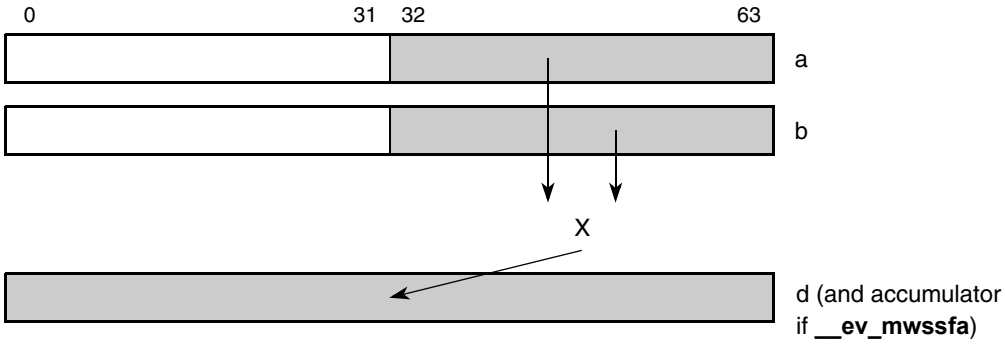


Table 182. __ev_mwssf (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwssf d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwssfa d,a,b**

__ev_mwssfaa

Vector Multiply Word Signed, Saturate, Fractional and Accumulate

d = __ev_mwssfaa (a,b)

```
temp0:63 ← a32:63 ×sf b32:63
if (a32:63 = 0x8000_0000) & (b32:63 = 0x8000_0000) then
   temp0:63 ← 0x7FFF_FFFF_FFFF_FFFF //saturate
   mov ← 1
else
   mov ← 0
temp0:64 ← EXTS(ACC0:63) + EXTS(temp0:63)
ov ← (temp0 ⊕ temp1)
d0:63 ← temp1:64)
// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← 0
SPEFSCROV ← mov
SPEFSCRSOV ← SPEFSCRSOV | ov | mov
```

The low word signed fractional elements in parameters a and b are multiplied, producing a 64-bit product. If both inputs are -1.0, the product saturates to the largest positive signed fraction. The 64-bit product is added to the accumulator, and the result is placed in parameter d and in the accumulator.

If there is an overflow from the multiply, the overflow and summary overflow bits are recorded in the SPEFSCR.

Note: There is no saturation on the addition with the accumulator.

Other registers altered: SPEFSCR ACC

Figure 176. Vector multiply word signed, saturate, fractional and accumulate (__ev_mwssfaa)

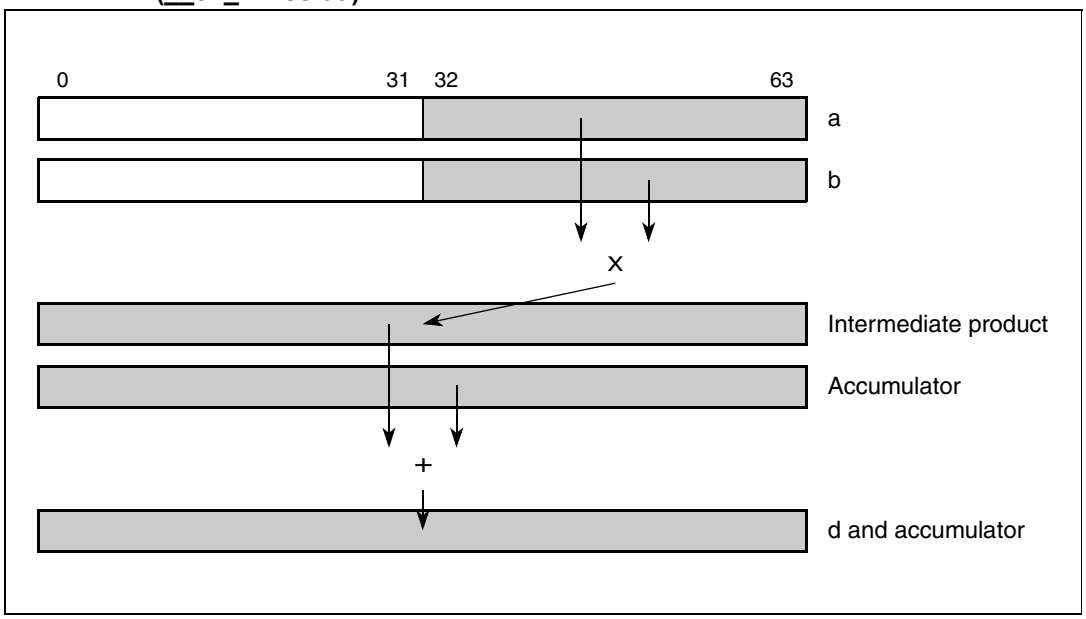


Table 183. __ev_mwssfaa (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwssfaa d,a,b**

__ev_mwssfan

Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative

d = __ev_mwssfan (a,b)

```
temp0:63 ← a32:63 ×sf b32:63
if (a32:63 = 0x8000_0000) & (b32:63 = 0x8000_0000) then
   temp0:63 ← 0x7FFF_FFFF_FFFF_FFFF //saturate
   mov ← 1
else
   mov ← 0
temp0:64 ← EXTS(ACC0:63) - EXTS(temp 0:63)
ov ← (temp0 ⊕ temp1)
d0:63 ← temp1:64


// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← 0
SPEFSCROV ← mov
SPEFSCRSOV ← SPEFSCRSOV | ov | mov
```

The low word signed fractional elements in parameters a and b are multiplied producing a 64-bit product. If both inputs are -1.0, the product saturates to the largest positive signed fraction. The 64-bit product is then subtracted from the accumulator and the result is placed in parameter d and the accumulator.

If there is an overflow from the multiply, the overflow and summary overflow bits are recorded in the SPEFSCR.

Note: There is no saturation on the subtraction with the accumulator.

Other registers altered: SPEFSCR ACC

Figure 177. Vector multiply word signed, saturate, fractional and accumulate Negative (__ev_mwssfan)

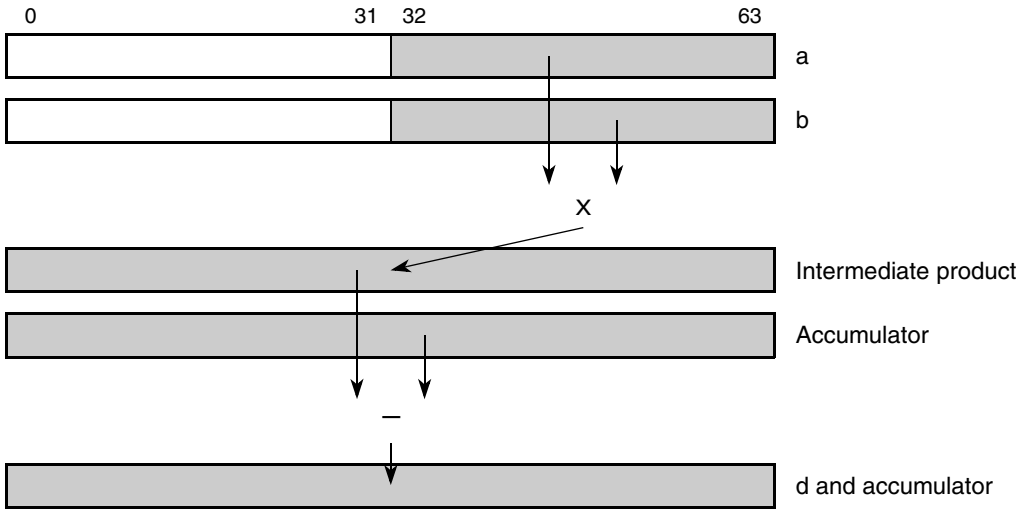


Table 184. __ev_mwssfan (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwssfan d,a,b**

__ev_mwumi

Vector Multiply Word Unsigned, Modulo, Integer (to Accumulator)

d = __ev_mwumi (a,b) (A = 0)

d = __ev_mwumia (a,b))A = 1)

$d_{0:63} \leftarrow a_{32:63} \times_{ui} b_{32:63}$

```
// update accumulator
if A = 1 then ACC0:63 ← d0:63
```

The low word unsigned integer elements in parameters a and b are multiplied to form a 64-bit product that is placed into parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

Figure 178. Vector multiply word unsigned, modulo, integer (to Accumulator) (__ev_mwumi)

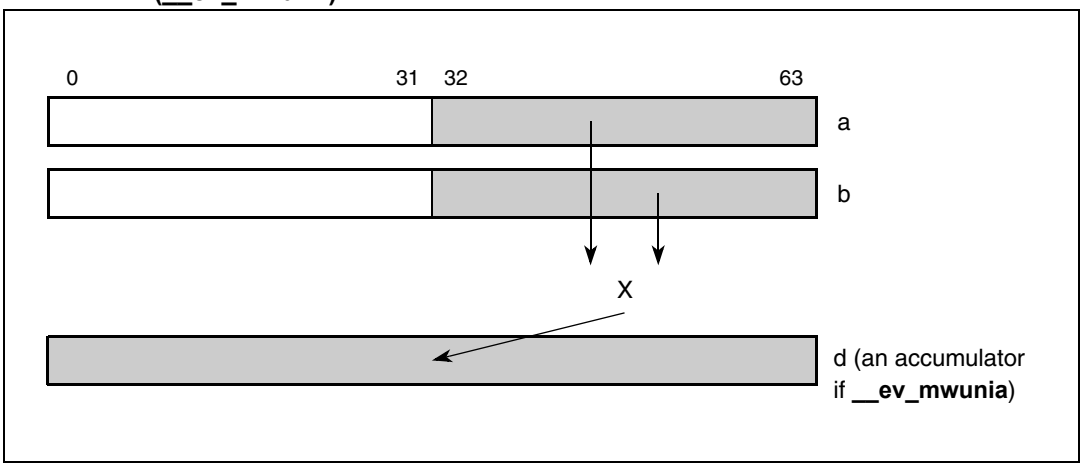


Table 185. __ev_mwumi (registers altered by).

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwumi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwumia d,a,b**

__ev_mwumiaa

Vector Multiply Word Unsigned, Modulo, Integer and Accumulate

d = __ev_mwumiaa (a,b)

$temp_{0:63} \leftarrow a_{32:63} \times_{ui} b_{32:63}$
$d_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The low word unsigned integer elements in parameters a and b are multiplied. The intermediate product is added to the contents of the 64-bit accumulator, and the resulting value is placed into the accumulator and into parameter d.

Other registers altered: ACC

Figure 179. Vector multiply word unsigned, modulo, integer and accumulate (__ev_mwumiaa)

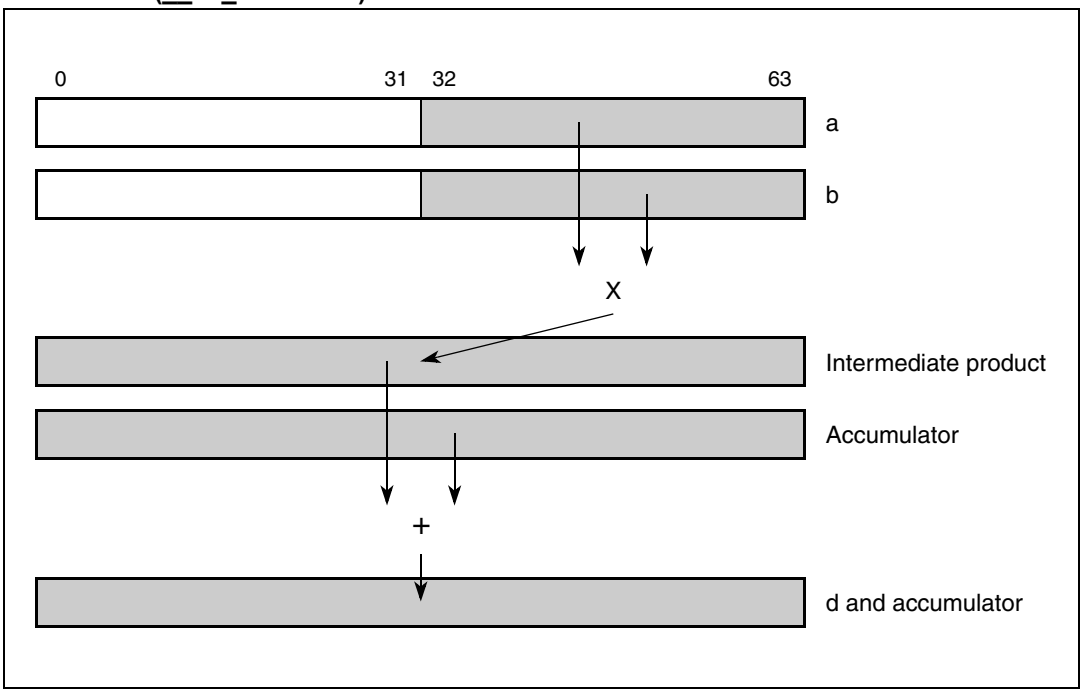


Table 186. __ev_mwumiaa (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwumiaa d,a,b**

__ev_mwumian

Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative

d = __ev_mwumian (a,b)

$temp_{0:63} \leftarrow a_{32:63} \times_{ui} b_{32:63}$
$d_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

The low word unsigned integer elements in parameters a and b are multiplied. The intermediate product is subtracted from the contents of the 64-bit accumulator, and the resulting value is placed into the accumulator and into parameter d.

Other registers altered: ACC

Figure 180. Vector multiply word unsigned, modulo, integer and accumulate Negative (__ev_mwumian)

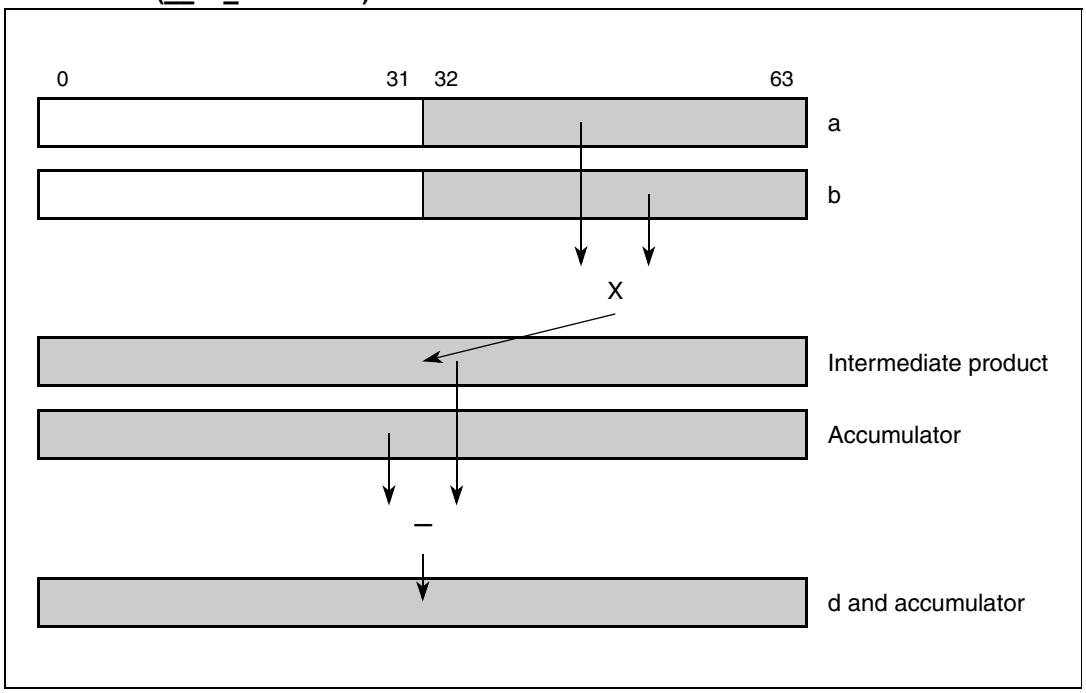


Table 187. __ev_mwumian (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwumian d,a,b**

__ev_nand

Vector NAND

d = __ev_nand (a,b)

$d_{0:31} \leftarrow \neg(a_{0:31}$ & $b_{0:31})$// Bitwise NAND

$d_{32:63} \leftarrow \neg(a_{32:63}$ & $b_{32:63})$ // Bitwise NAND

Each element of parameters a and b are bitwise NANDed. The result is placed in the corresponding element of parameter d.

Figure 181. Vector NAND (__ev_nand)

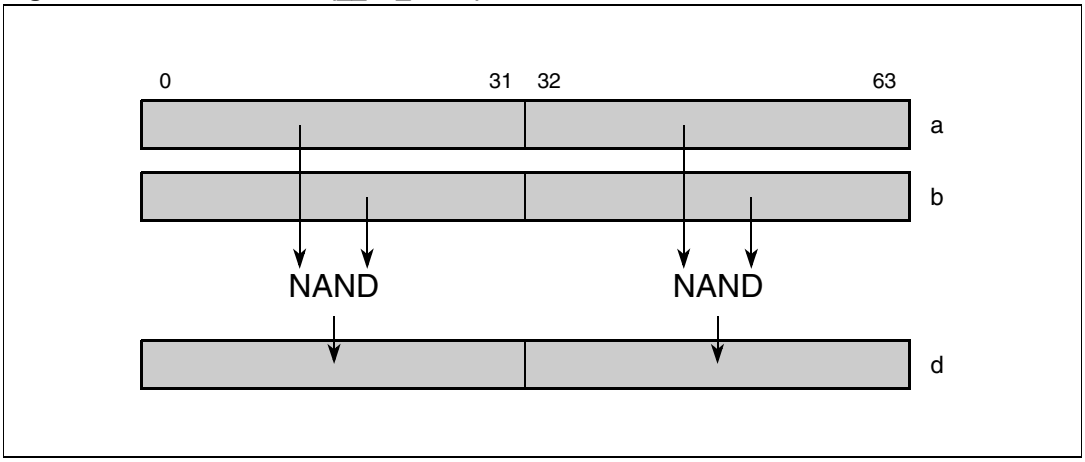


Table 188. __ev_nand (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evnand d,a,b**

__ev_neg

Vector Negate

d = __ev_neg(a)

$d_{0:31} \leftarrow \mathbf{NEG}(a_{0:31})$

$d_{32:63} \leftarrow \mathbf{NEG}(a_{32:63})$

The negative of each element of parameter a is placed in parameter d. The negative of 0x8000_0000 (most negative number) returns 0x8000_0000. No overflow is detected.

Figure 182. Vector negate (__ev_neg)

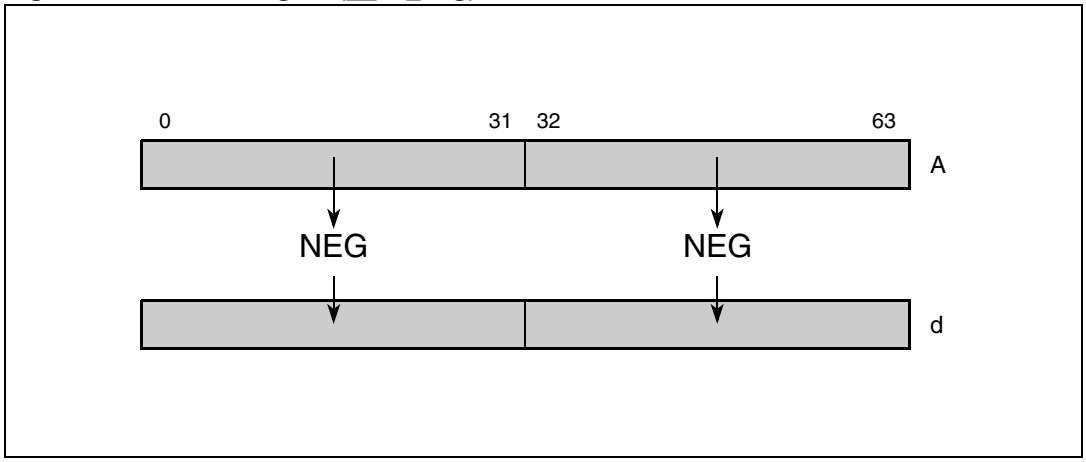


Table 189. __ev_neg (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evneg d,a,b**

__ev_nor

Vector NOR

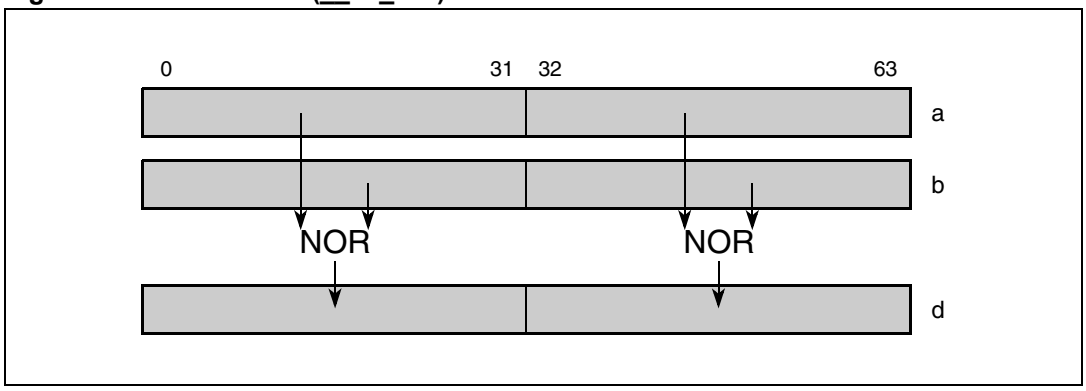
d = __ev_nor (a,b)

$d_{0:31} \leftarrow \neg(a_{0:31} \mid b_{0:31})$ `// Bitwise NOR`
$d_{32:63} \leftarrow \neg(a_{32:63} \mid b_{32:63})$ `// Bitwise NOR`

Each element of parameters a and b is bitwise NORed. The result is placed in the corresponding element of parameter d.

*Note: Use **evnand** or **evnor** for **evnot**.*

Figure 183. Vector NOR (__ev_nor)



Simplified mnemonic: **evnot** d,a performs a complement register.

evnot d,a equivalent to **evnor** d,a,a

Table 190. __ev_nor (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evnor d,a,b**

__ev_or

Vector OR

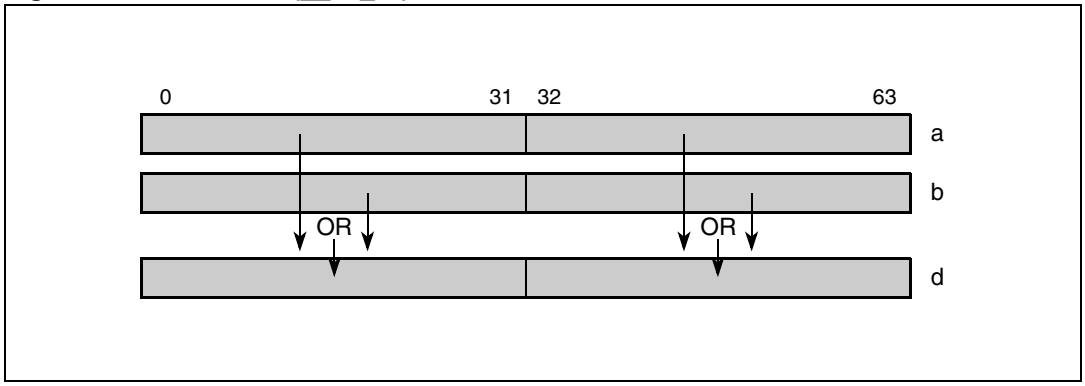
d = __ev_or (a,b)

$d_{0:31} \leftarrow a_{0:31}$ | $b_{0:31}$ //Bitwise OR

$d_{32:63} \leftarrow a_{32:63}$ | $b_{32:63}$// Bitwise OR

Each element of parameters a and b is bitwise ORed. The result is placed in the corresponding element of parameter d.

Figure 184. Vector OR (__ev_or)



Simplified mnemonic: **evmr** d,a handles moving of the full 64-bit SPE register.

evmr d,a equivalent to **evor** d,a,a

Table 191. __ev_or (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evor d,a,b**

__ev_orc

Vector OR with Complement

d = __ev_orc (a,b)

$d_{0:31} \leftarrow a_{0:31}$ | ($\neg b_{0:31}$) // Bitwise ORC

$d_{32:63} \leftarrow a_{32:63}$ | ($\neg b_{32:63}$) // Bitwise ORC

Each element of parameter a is bitwise ORed with the complement of parameter b. The result is placed in the corresponding element of parameter d.

Figure 185. Vector OR with complement (__ev_orc)

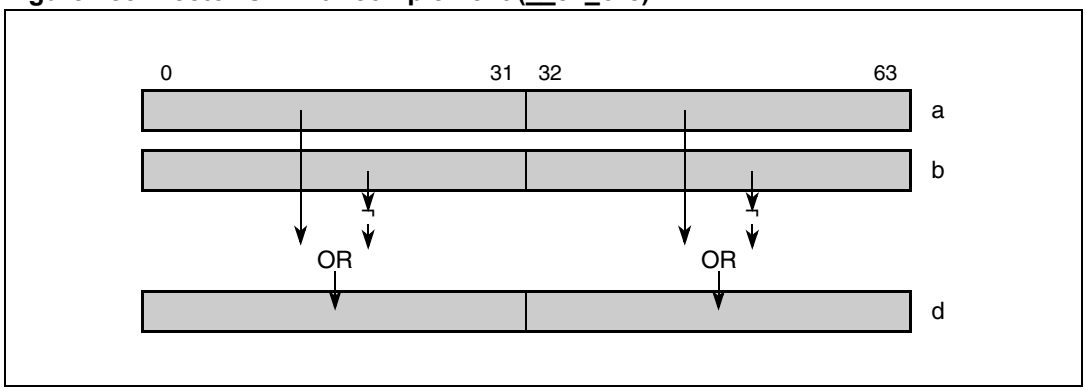


Table 192. __ev_orc (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evorc d,a,b**

__ev_rlw

Vector Rotate Left Word

d = __ev_rlw(a,b)

$nh \leftarrow b_{27:31}$
$nl \leftarrow b_{59:63}$
$d_{0:31} \leftarrow ROTL(a_{0:31}, nh)$
$d_{32:63} \leftarrow ROTL(a_{32:63}, nl)$

Each of the high and low elements of parameter a is rotated left by an amount specified in parameter b. The result is placed into parameter d. Rotate values for each element of parameter a are found in bit positions b[27–31] and b[59–63].

Figure 186. Vector rotate left word (__ev_rlw)

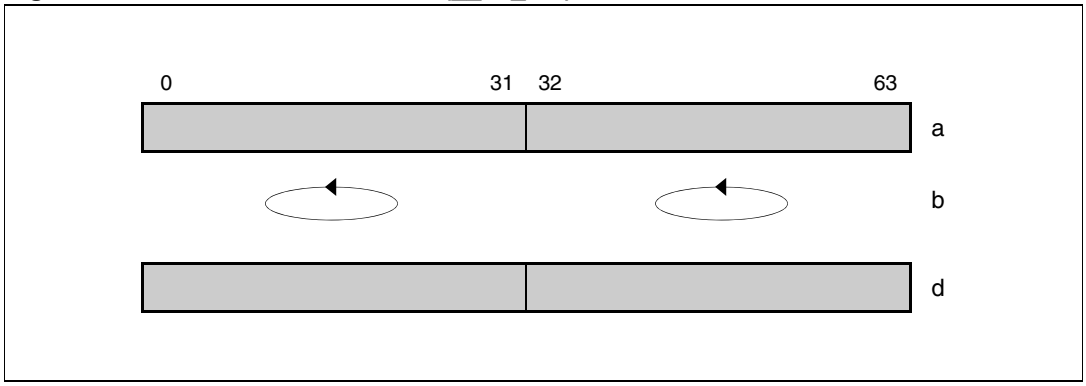


Table 193. __ev_rlw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evrlw d,a,b**

__ev_rlwi

Vector Rotate Left Word Immediate

d = __ev_rlwi (a,b)

$n \leftarrow UIMM$

$d_{0:31} \leftarrow ROTL(a_{0:31}, n)$

$d_{32:63} \leftarrow ROTL(a_{32:63}, n)$

Both the high and low elements of parameter a are rotated left by an amount specified by a 5-bit immediate value.

Figure 187. Vector rotate left word immediate (__ev_rlwi)

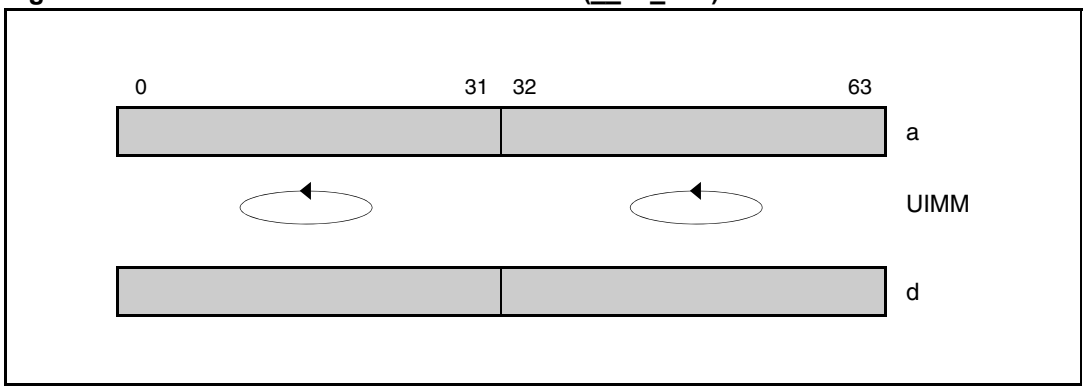


Table 194. __ev_rlwi (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	5-bit unsigned	**evrlwi d,a,b**

__ev_rndw

Vector Round Word

d = __ev_rndw(a)

$d_{0:31} \leftarrow (a_{0:31}$+0x00008000) & 0xFFFF0000 // Modulo sum
$d_{32:63} \leftarrow (a_{32:63}$+0x00008000) & 0xFFFF0000 // Modulo sum

The 32-bit elements of parameter a are rounded into 16 bits. The result is placed into parameter d. The resulting 16 bits are placed in the most significant 16 bits of each element of parameter d, zeroing out the low order 16 bits of each element.

Figure 188. Vector round word (__ev_rndw)

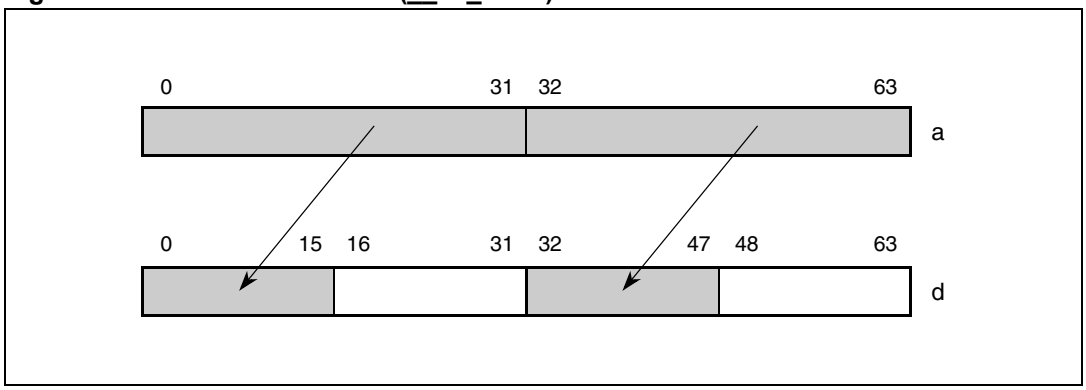


Table 195. __ev_rndw (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evrndw d,a**

__ev_select_eq

Vector Select Equal

e = __ev_select_eq(a,b,c,d)

```
if  (a0:31 = b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31
if (a32:63 = b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameters c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a = b? c : d.

Figure 189. Vector select equal (__ev_select_eq)

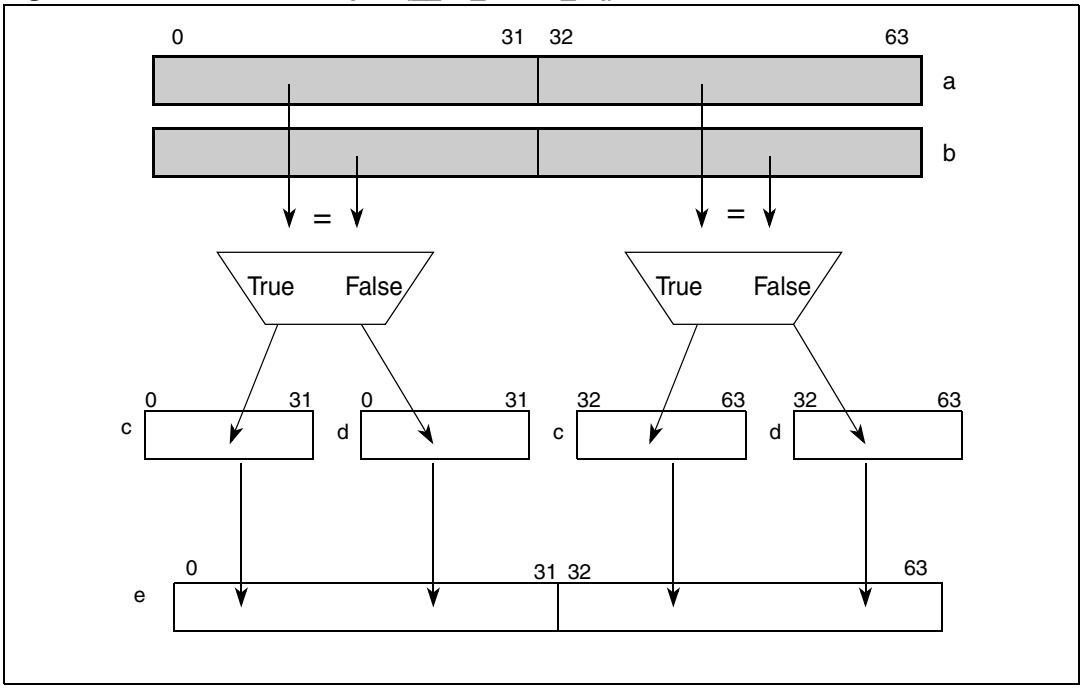


Table 196. __ev_select_eq (registers altered by).

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evcmpeq x,a,b** **evsel e,c,d,x**

__ev_select_fs_eq

Vector Select Floating-Point Equal

e = __ev_select_fs_eq(a,b,c,d)

```
if  (a0:31 = b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31
if (a32:63 = b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in the C programming language. For example, the aforementioned intrinsic maps to the following logical expression: a = b? c : d.

Figure 190. Vector select Floating-Point equal (__ev_select_fs_eq)

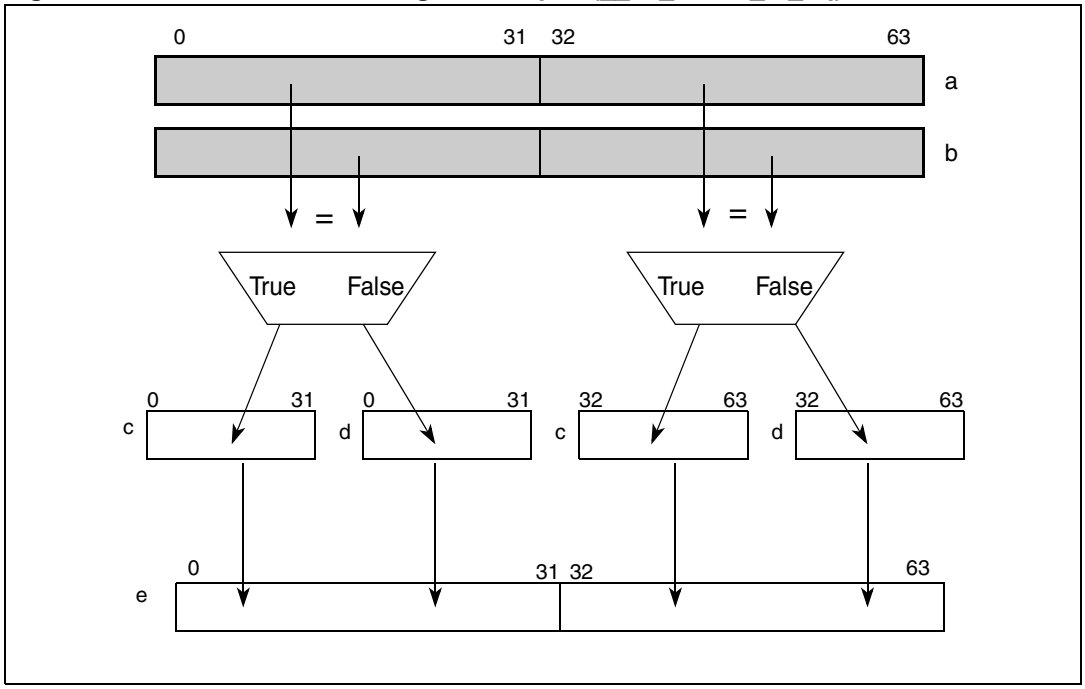


Table 197. __ev_select_fs_eq (registers altered by).

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfscmpeq x,a,b** **evsel e,c,d,x**

__ev_select_fs_gt

Vector Select Floating-Point Greater Than

e = ___ev_select_fs_gt(a,b,c,d)

```
if  (a0:31 > b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31
if (a32:63 > b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a > b ? c : d.

Figure 191. Vector select Floating-Point greater than (__ev_select_fs_gt)

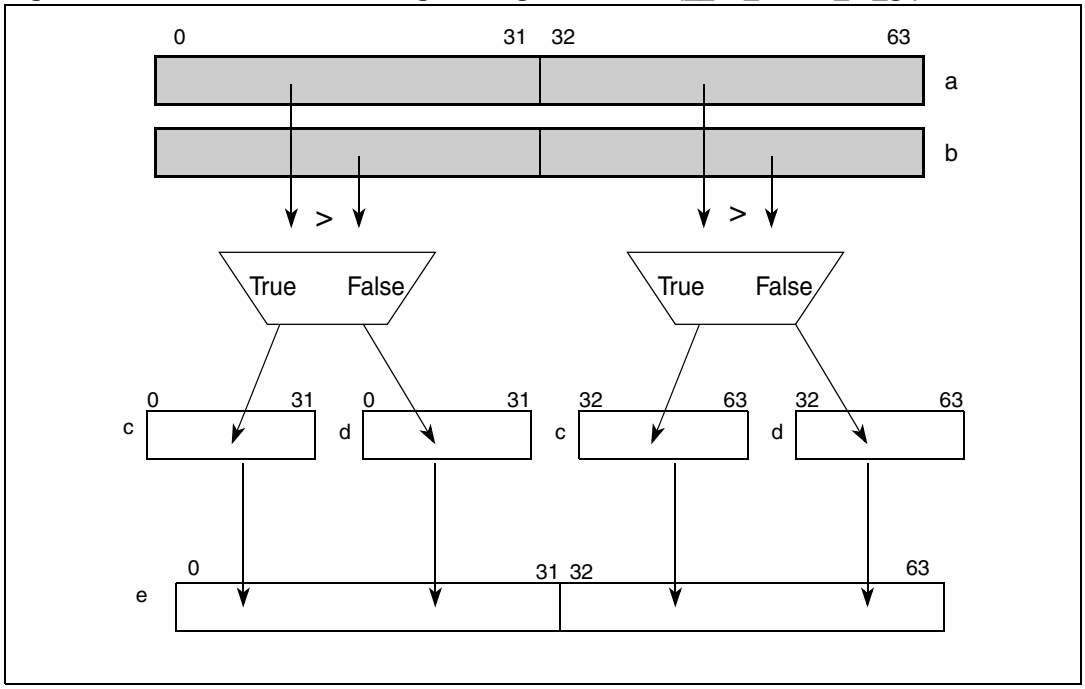


Table 198. __ev_select_fs_gt (registers altered by).

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfscmpgt x,a,b** **evsel e,c,d,x**

__ev_select_fs_lt

Vector Select Floating-Point Less Than

e = __ev_select_fs_lt(a,b,c,d)

```
if  (a0:31 < b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31
if (a32:63 < b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a < b? c : d.

Figure 192. Vector select Floating-Point less than (__ev_select_fs_lt)

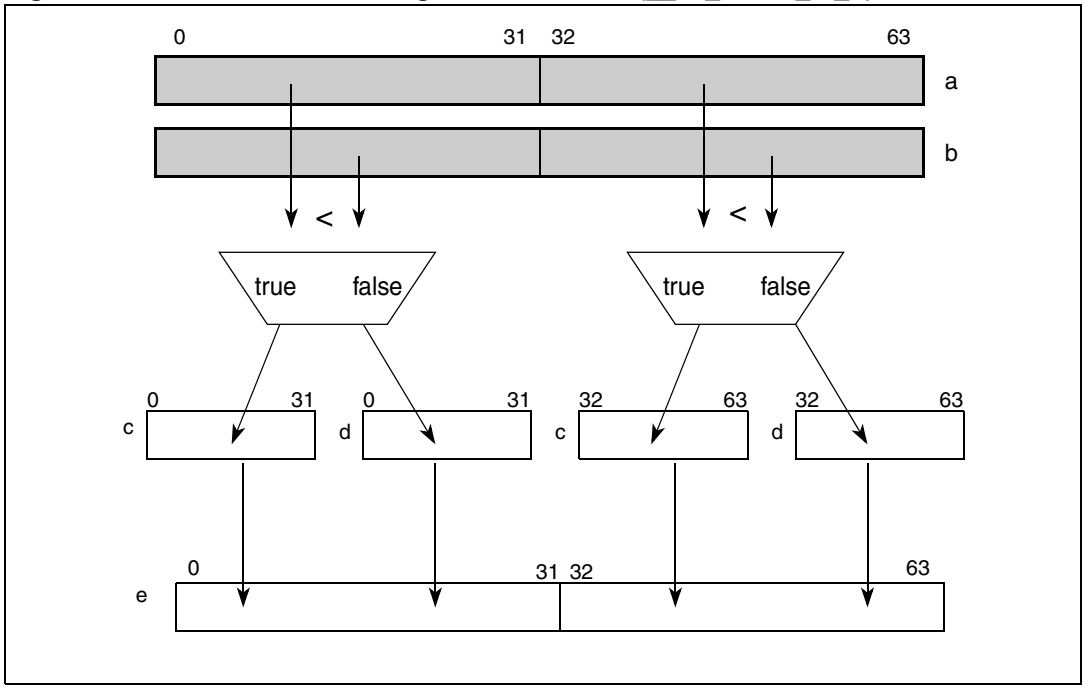


Table 199. __ev_select_fs_lt (registers altered by).

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfscmplt x,a,b** **evsel e,c,d,x**

__ev_select_fs_tst_eq

Vector Select Floating-Point Test Equal

e = __ev_select_fs_tst_eq(a,b,c,d)

```
if  (a0:31 = b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31
if (a32:63 = b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a =b? c : d. This intrinsic differs from __ev_select_fs_eq because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_select_fs_eq instead.

Figure 193. Vector select Floating-Point test equal (__ev_select_fs_tst_eq)

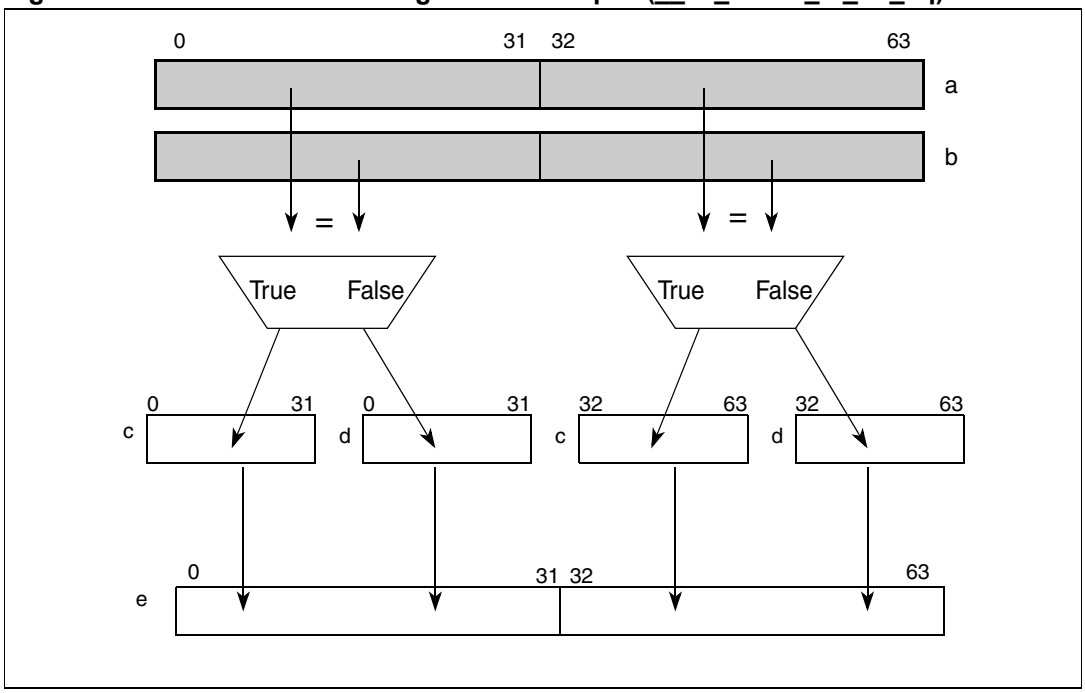


Table 200. __ev_select_fs_tst_eq (registers altered by).

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfststeq x,a,b** **evsel e,c,d,x**

__ev_select_fs_tst_gt

Vector Select Floating-Point Test Greater Than

e = ___ev_select_fs_tst_gt(a,b,c,d)

```
if  (a0:31 > b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31
if (a32:63 > b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a > b ? c : d. This intrinsic differs from __ev_select_fs_gt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_select_fs_gt instead.

Figure 194. Vector select Floating-Point test greater than (__ev_select_fs_tst_gt)

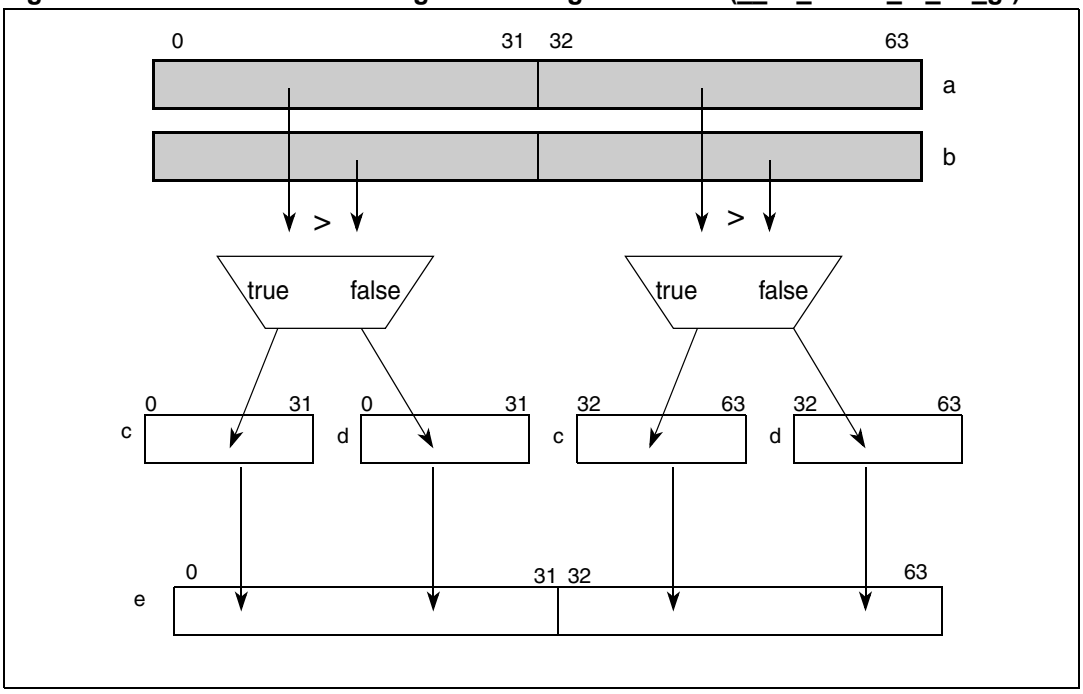


Table 201. __ev_select_fs_tst_gt (registers altered by).

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfststgt x,a,b** **evsel e,c,d,x**

__ev_select_fs_tst_lt

Vector Select Floating-Point Test Less Than

e = __ev_select_fs_tst_lt(a,b,c,d)

```
if  (a0:31 < b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31
if (a32:63 < b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a < b? c : d. This intrinsic differs from __ev_select_fs_lt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_select_fs_lt instead.

Figure 195. Vector select Floating-Point test less than (__ev_select_fs_tst_lt)

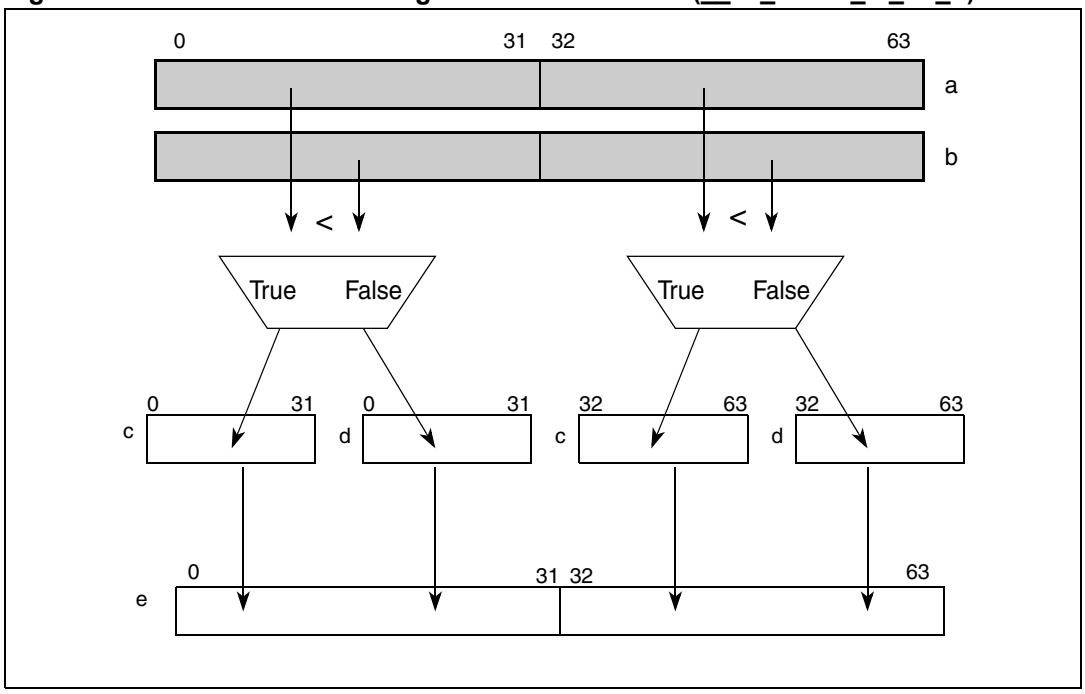


Table 202. __ev_select_fs_tst_lt (registers altered by).

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfststlt x,a,b** **evsel e,c,d,x**

__ev_select_gts

Vector Select Greater Than Signed

e = __ev_select_gts(a,b,c,d)

```
if  (a0:31 >signed b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31
if (a32:63 >signed b32:63) then e32:63← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a > b ? c : d.

Figure 196. Vector select greater than signed (__ev_select_gts)

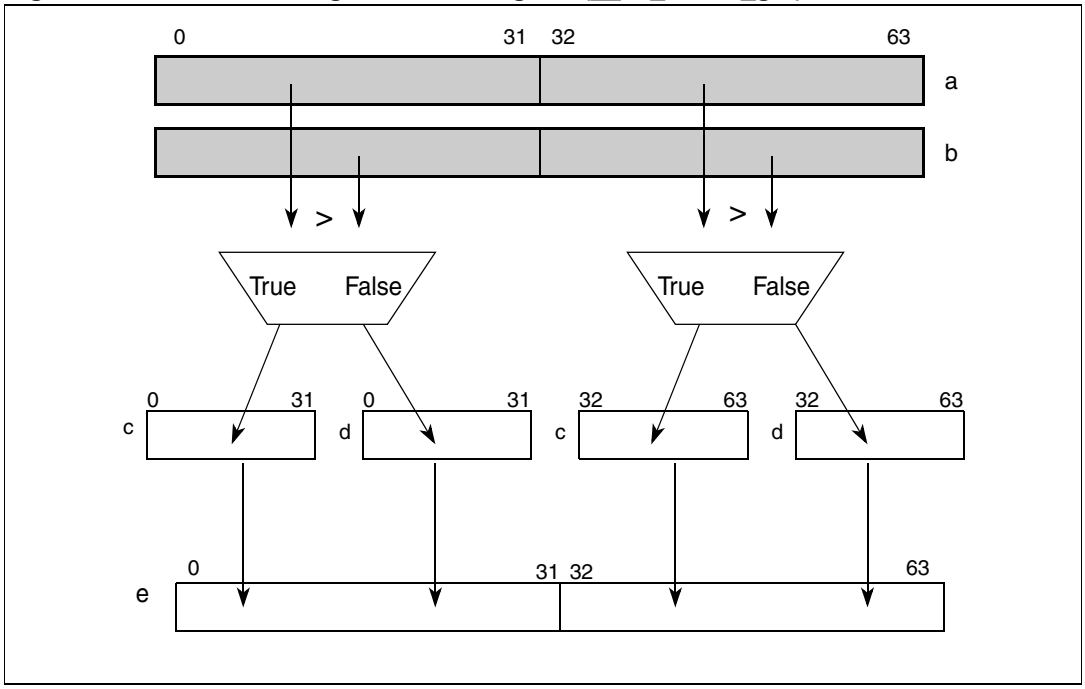


Table 203. __ev_select_gts (registers altered by).

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evcmpgts x,a,b** **evsel e,c,d,x**

__ev_select_gtu

Vector Select Greater Than Unsigned

e = __ev_select_gtu(a,b,c,d)

```
if  (a0:31 > unsigned c0:31) then e0:31 ← c0:31
else e0:31 ← d0:31
if (a32:63 > unsigned b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a > b? c : d.

Figure 197. Vector select greater than unsigned (__ev_select_gtu)

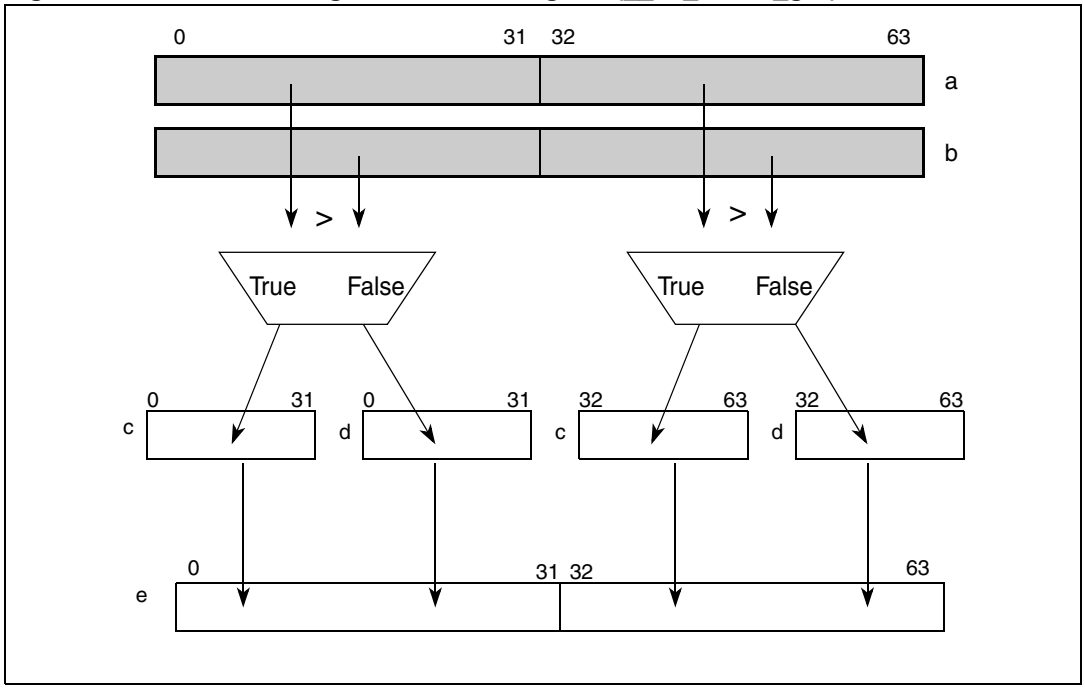


Table 204. __ev_select_gtu (registers altered by).

e	A	B	C	D	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evcmpgtu x,a,b** **evsel e,c,d,x**

__ev_select_lts

Vector Select Less Than Signed

e = __ev_select_lts(a,b,c,d)

```
if  (a0:31 <signed b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31
if (a32:63 <signed b32:63) then e ← c32:63
else e ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a < b? c : d.

Figure 198. Vector select less than signed (__ev_select_lts)

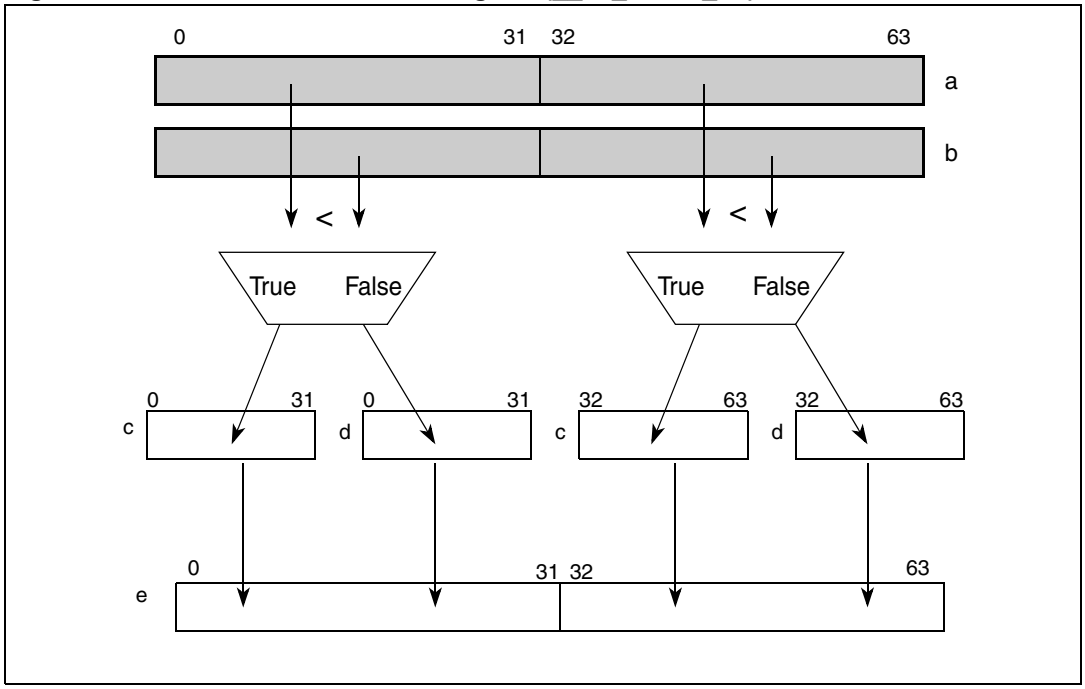


Table 205. __ev_select_lts (registers altered by).

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evcmplts x,a,b** **evsel e,c,d,x**

__ev_select_ltu

Vector Select Less Than Unsigned

e = __ev_select_ltu(a,b,c,d)

```
if  (a0:31 <unsigned b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31
if (a32:63 <unsigned b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a < b? c : d.

Figure 199. Vector select less than unsigned (__ev_select_ltu)

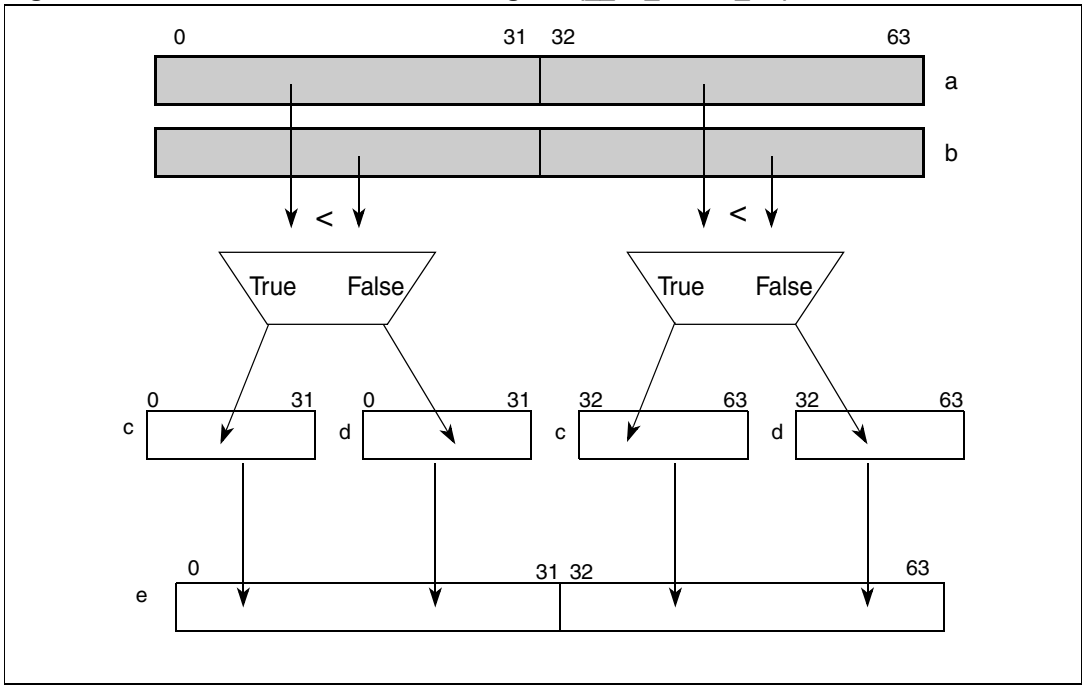


Table 206. __ev_select_ltu (registers altered by).

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evcmpltu x,a,b** **evsel e,c,d,x**

__ev_slw

Vector Shift Left Word

d = __ev_slw (a,b)

$nh \leftarrow b_{26:31}$
$nl \leftarrow b_{58:63}$
$d_{0:31} \leftarrow SL(a_{0:31}, nh)$
$d_{32:63} \leftarrow SL(a_{32:63}, nl)$

Each of the high and low elements of parameter a are shifted left by an amount specified in parameter b. The result is placed into parameter d. The separate shift amounts for each element are specified by 6 bits in parameter b that lie in bit positions 26–31 and 58–63.

Shift amounts from 32 to 63 give a zero result.

Figure 200. Vector shift left word (__ev_slw)

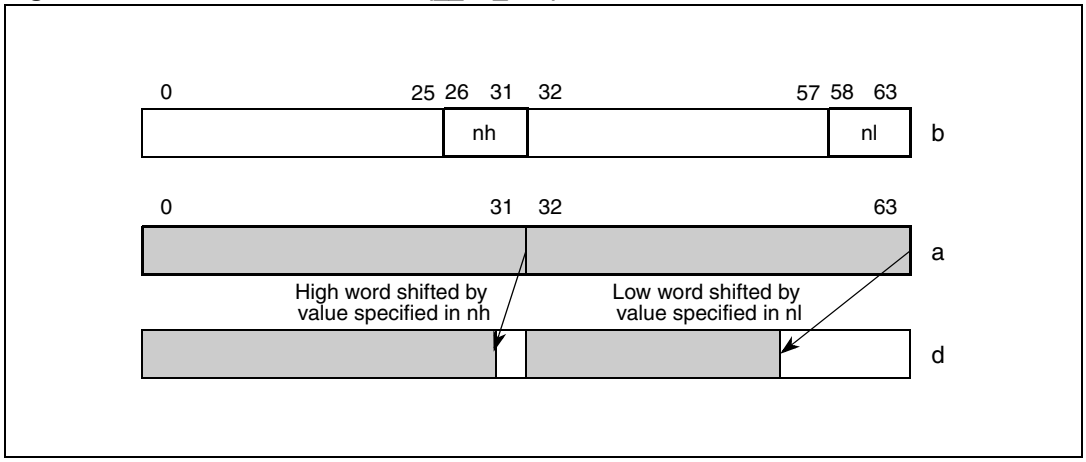


Table 207. __ev_slw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evslw d,a,b**

__ev_slwi

Vector Shift Left Word Immediate

d = __ev_slwi (a,b)

$n \leftarrow UIMM$

$d_{0:31} \leftarrow SL(a_{0:31}, n)$

$d_{32:63} \leftarrow SL(a_{32:63}, n)$

Both high and low elements of parameter a are shifted left by the 5-bit UIMM value, and the results are placed in parameter d.

Figure 201. Vector shift left word immediate (__ev_slwi)

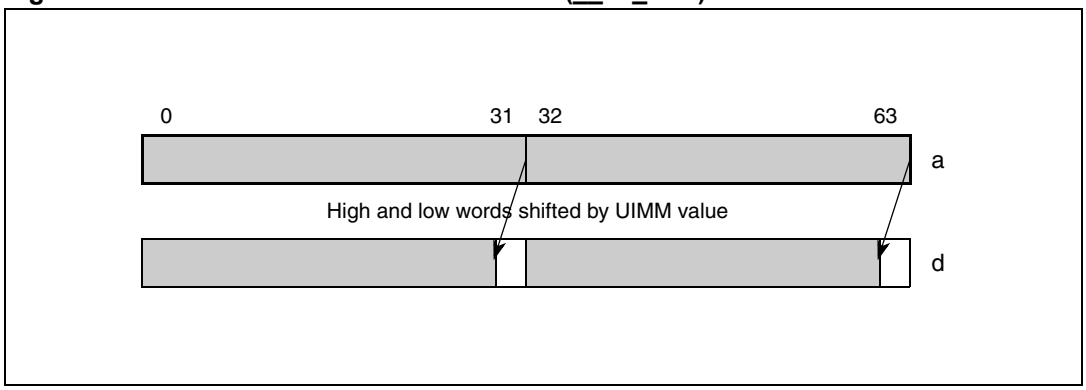


Table 208. __ev_slwi (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	5-bit unsigned	**evslwi d,a,b**

__ev_splatfi

Vector Splat Fractional Immediate

d = __ev_splatfi(a)

$d_{0:31} \leftarrow SIMM \,||\, {}^{27}0$

$d_{32:63} \leftarrow SIMM \,||\, {}^{27}0$

The 5-bit immediate value is padded with trailing zeros and placed in both elements of parameter d, as shown in *Figure 202*. The SIMM ends up in bit positions d[0–4] and d[32–36].

Figure 202. Vector splat fractional immediate (__ev_splatfi)

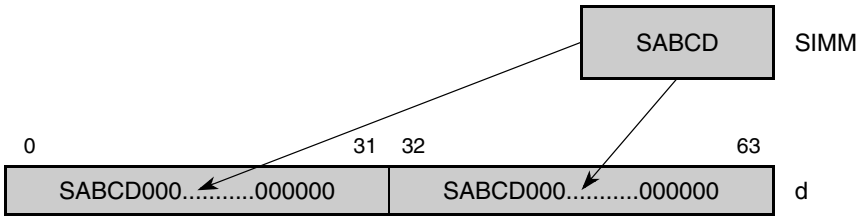


Table 209. __ev_splatfi (registers altered by).

d	a	Maps to
__ev64_opaque	5-bit signed	**evsplatfi d,a**

__ev_splati

Vector Splat Immediate

d = __ev_splati (a)

$d_{0:31} \leftarrow$ **EXTS(SIMM)**

$d_{32:63} \leftarrow$ **EXTS(SIMM)**

The 5-bit immediate value is sign-extended and placed in both elements of parameter d, as shown in *Figure 203*.

Figure 203. __ev_splati sign extend

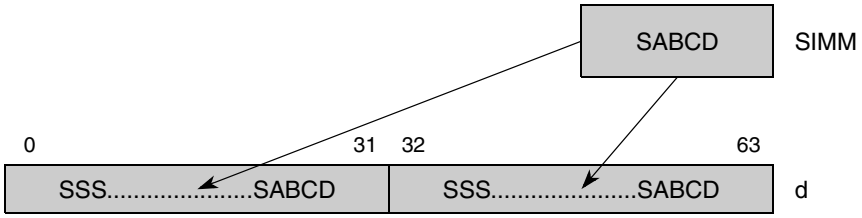


Table 210. __ev_splati (registers altered by).

d	a	Maps to
__ev64_opaque	5-bit signed	**evsplati d,a**

__ev_srwis

Vector Shift Right Word Immediate Signed

d = __ev_srwis**(a,b)**

$n \leftarrow UIMM$

$d_{0:31} \leftarrow EXTS(a_{0:31-n})$

$d_{32:63} \leftarrow EXTS(b_{32:63-n})$

Both high and low elements of parameter a are shifted right by the 5-bit UIMM value. Bits in the most significant positions vacated by the shift are filled with a copy of the sign bit.

Figure 204. Vector shift right word immediate signed (__ev_srwis)

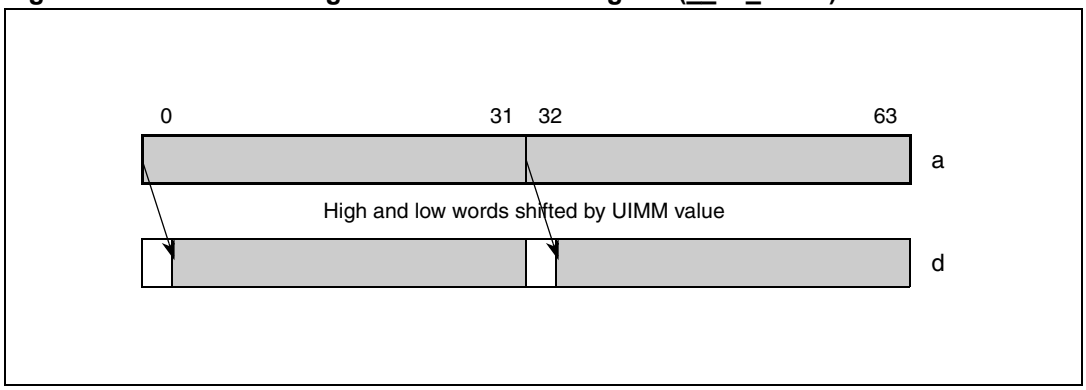


Table 211. __ev_srwis (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	5-bit unsigned	**evsrwis d,a,b**

__ev_srwiu

Vector Shift Right Word Immediate Unsigned

d = __ev_srwiu(a,b)

$n \leftarrow UIMM$

$d_{0:31} \leftarrow EXTZ(a_{0:31-n})$

$d_{32:63} \leftarrow EXTZ(a_{32:63-n})$

Both high and low elements of parameter a are shifted right by the 5-bit UIMM value; 0 bits are shifted in to the most significant position. Bits in the most significant positions vacated by the shift are filled with a zero bit.

Figure 205. Vector shift right word immediate unsigned (__ev_srwiu)

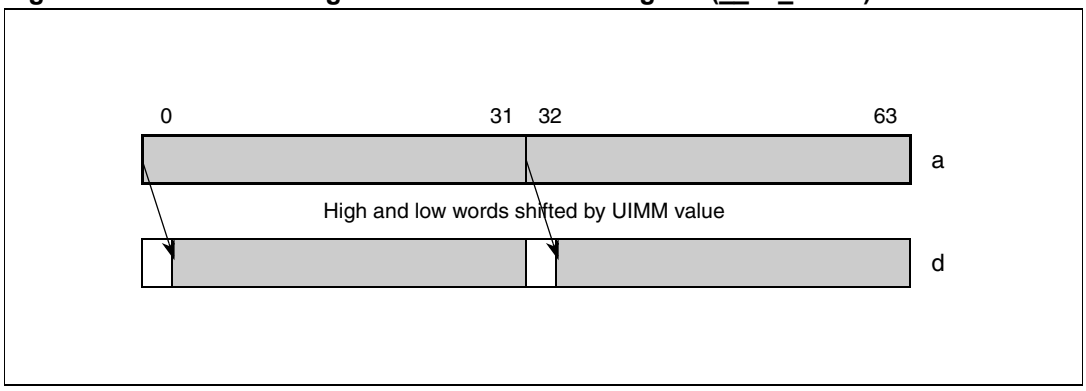


Table 212. __ev_srwiu (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	5-bit unsigned	**evsrwiu d,a,b**

__ev_srws

Vector Shift Right Word Signed

d = __ev_srws **(a,b)**

$nh \leftarrow b_{26:31}$

$nl \leftarrow b_{58:63}$

$d_{0:31} \leftarrow \mathbf{EXTS}(a_{0:31-nh})$

$d_{32:63} \leftarrow \mathbf{EXTS}(a_{32:63-nl})$

Both the high and low elements of parameter a are shifted right by an amount specified in parameter b. The result is placed into parameter d. The separate shift amounts for each element are specified by 6 bits in parameter b that lie in bit positions 26–31 and 58–63. The sign bits are shifted in to the most significant position.

Shift amounts from 32 to 63 give a result of 32 sign bits.

Figure 206. Vector shift right word signed (__ev_srws)

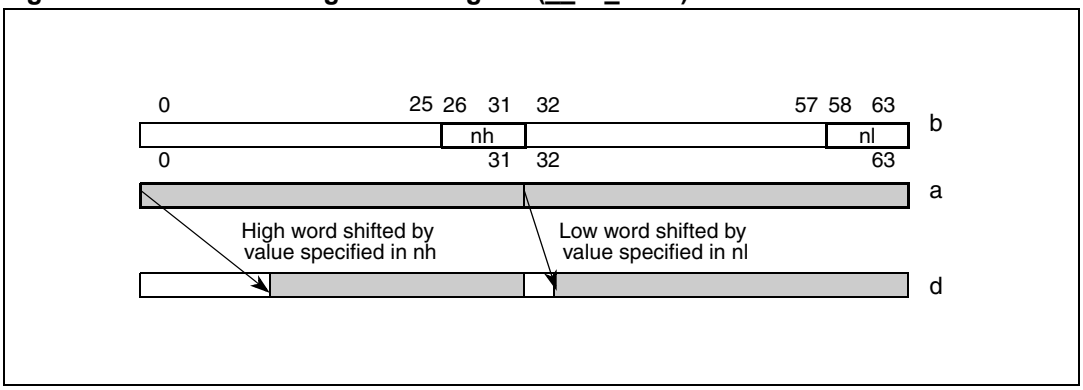


Table 213. __ev_srws (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evsrws d,a,b**

__ev_srwu

Vector Shift Right Word Unsigned

d = __ev_srwu **(a,b)**

$nh \leftarrow b_{26:31}$
$nl \leftarrow b_{58:63}$
$d_{0:31} \leftarrow EXTZ(a_{0:31-nh})$
$d_{32:63} \leftarrow EXTZ(a_{32:63-nl})$

Both the high and low elements of parameter a are shifted right by an amount specified in parameter b. The result is placed into parameter d. The separate shift amounts for each element are specified by 6 bits in parameter b that lie in bit positions 26–31 and 58–63. Zero bits are shifted in to the most significant position.

Shift amounts from 32 to 63 give a zero result.

Figure 207. Vector shift right word unsigned (__ev_srwu)

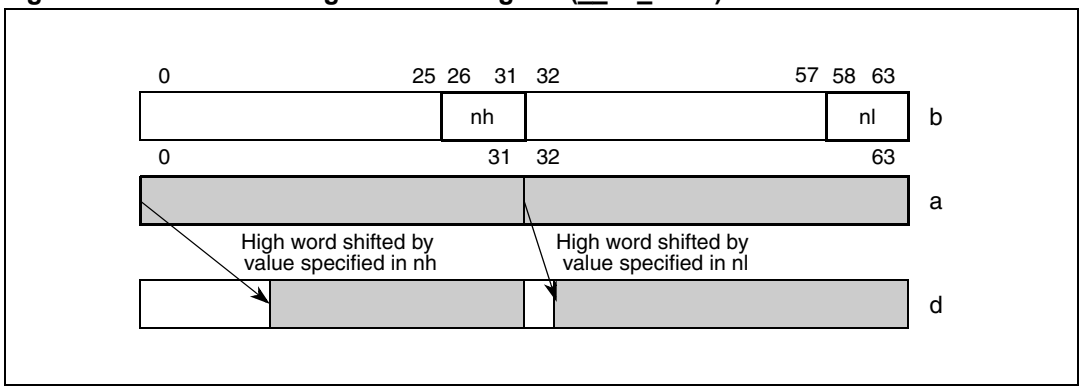


Table 214. __ev_srwu (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evsrwu d,a,b**

__ev_stdd

Vector Store Double of Double

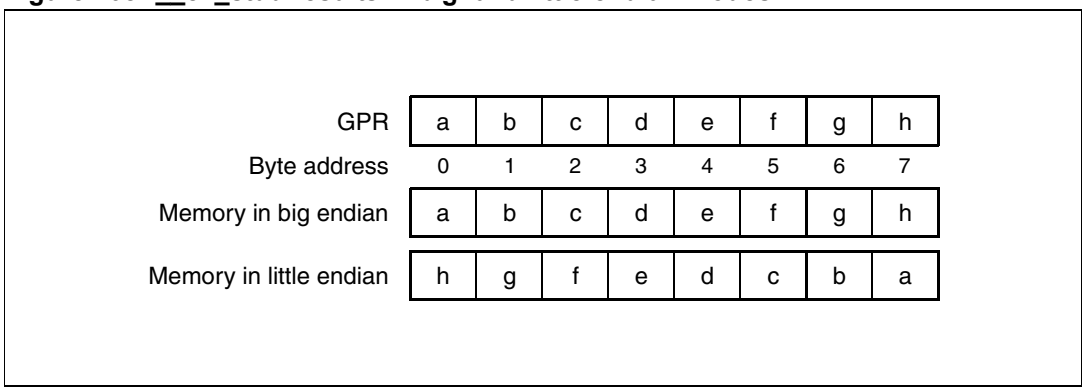
d = __ev_stdd (a,b,c)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*8)
MEM(EA,8) ← RS0:63
```

The contents of **r**S are stored as a double word in storage addressed by EA.

Figure 208 shows how bytes are stored in memory as determined by the endian mode.

Figure 208. __ev_stdd results in big- and little-endian modes



Note: During implementation, an alignment exception occurs if the EA is not double-word aligned.

Table 215. __ev_stdd (registers altered by).

d	a	b	c	Maps to
void	__ev64_opaque	__ev64_opaque	5-bit unsigned	**evstdd d,a,b,c**

__ev_stddx

Vector Store Double of Double Indexed

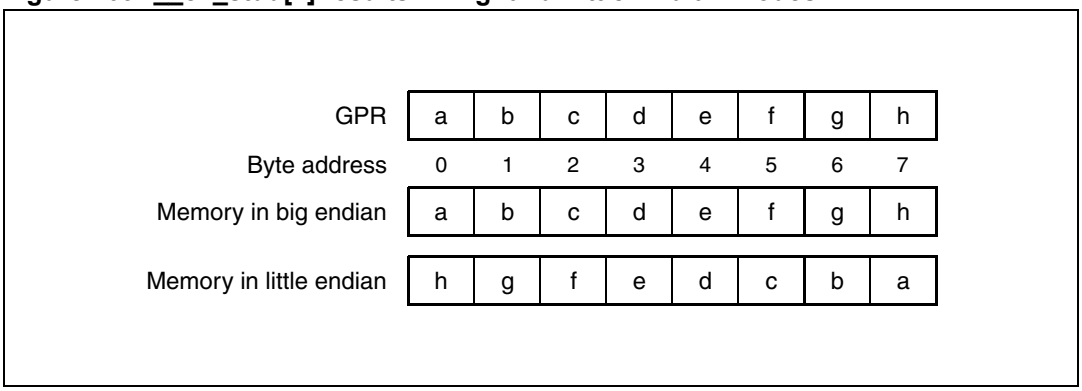
d = __ev_stddx (a,b,c)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
MEM(EA,8) ← RS0:63
```

The contents of **r**S are stored as a double word in storage addressed by EA.

Figure 209 shows how bytes are stored in memory as determined by the endian mode.

Figure 209. __ev_stdd[x] results in Big- and Little-Endian modes



Note: During implementation, an alignment exception occurs if the EA is not double-word aligned.

Table 216. __ev_stddx (registers altered by).

d	a	b	c	Maps to
void	__ev64_opaque	__ev64_opaque	int32_t	**evstddx d,a,b,c**

__ev_stdh

Vector Store Double of Four Half Words

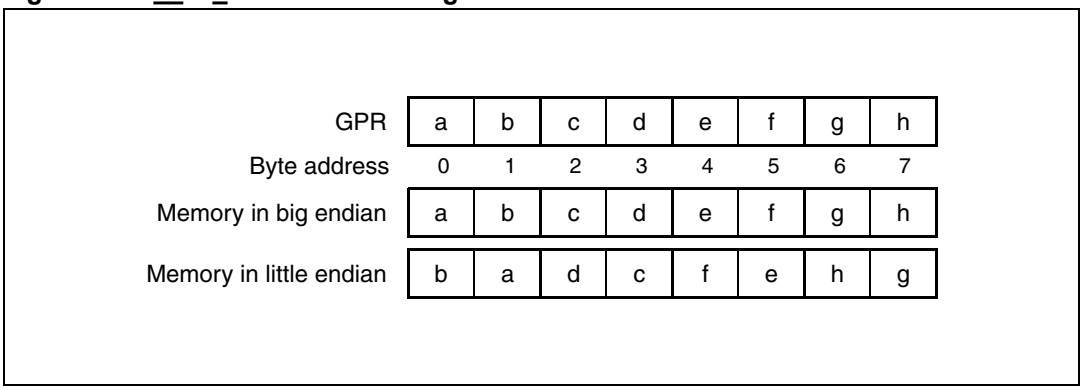
d = __ev_stdh (a,b,c)

```
if (a = 0) then temp ← 0
else temp ← a
EA ← temp + EXTZ(C*8)
MEM(EA,2) ← RS0:15
MEM(EA+2,2) ← RS16:31
MEM(EA+4,2) ← RS32:47
MEM(EA+6,2) ← RS48:63
```

The contents of **r**S are stored as four half words in storage addressed by EA.

Figure 210 shows how bytes are stored in memory as determined by the endian mode.

Figure 210. __ev_stdh results in Big- and Little-Endian modes



Note: During implementation, an alignment exception occurs if the EA is not double-word aligned.

Table 217. __ev_stdh (registers altered by).

d	a	b	c	Maps to
void	__ev64_opaque	__ev64_opaque	5-bit unsigned	**evstdh d,a,b,c**

__ev_stdhx

Vector Store Double of Four Half Words Indexed

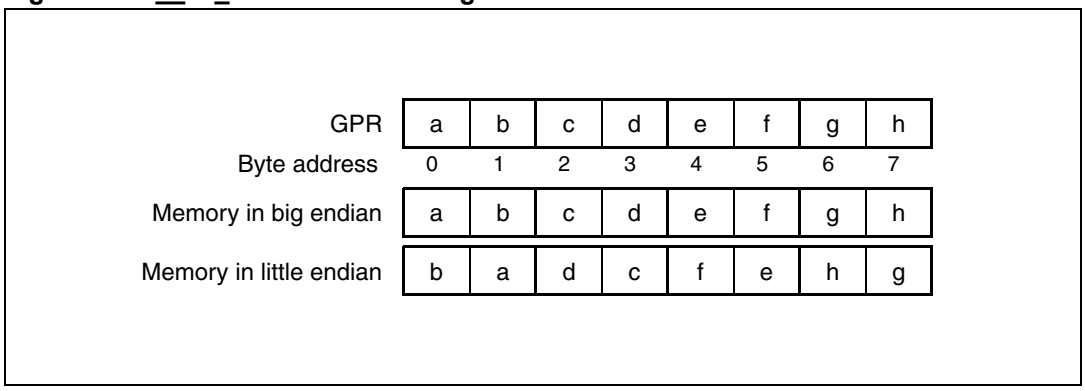
d = __ev_stdhx (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ← temp + (b)
MEM(EA,2) ← RS0:15
MEM(EA+2,2) ← RS16:31
MEM(EA+4,2) ← RS32:47
MEM(EA+6,2) ← RS48:63
```

The contents of **r**S are stored as four half words in storage addressed by EA.

Figure 211 shows how bytes are stored in memory as determined by the endian mode.

Figure 211. __ev_stdhx results in Big- and Little-Endian modes



Note: During implementation, an alignment exception occurs if the EA is not double-word aligned.

Table 218. __ev_stdhx (registers altered by).

d	a	b	c	Maps to
void	__ev64_opaque	__ev64_opaque	int32_t	**evstdhx d,a,b,c**

__ev_stdw

Vector Store Double of Two Words

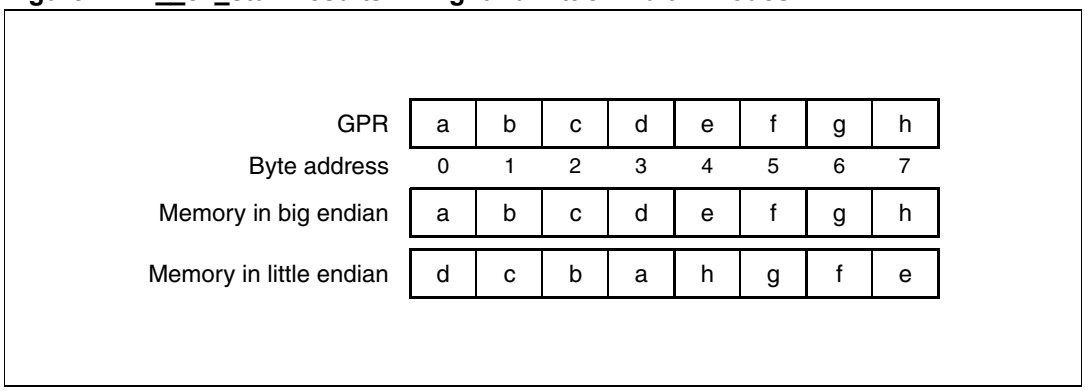
d = __ev_stdw (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ← temp + EXTZ(UIMM*8)
MEM(EA,4) ← RS0:31
MEM(EA+4,4) ← RS32:63
```

The contents of **r**S are stored as two words in storage addressed by EA.

Figure 212 shows how bytes are stored in memory as determined by the endian mode.

Figure 212. __ev_stdw results in Big- and Little-Endian modes



Note: During implementation, an alignment exception occurs if the EA is not double-word aligned.

Table 219. __ev_stdw (registers altered by).

d	a	b	c	Maps to
void	__ev64_opaque	__ev64_opaque	5-bit unsigned	**evstdw d,a,b,c**

__ev_stdwx

Vector Store Double of Two Words Indexed

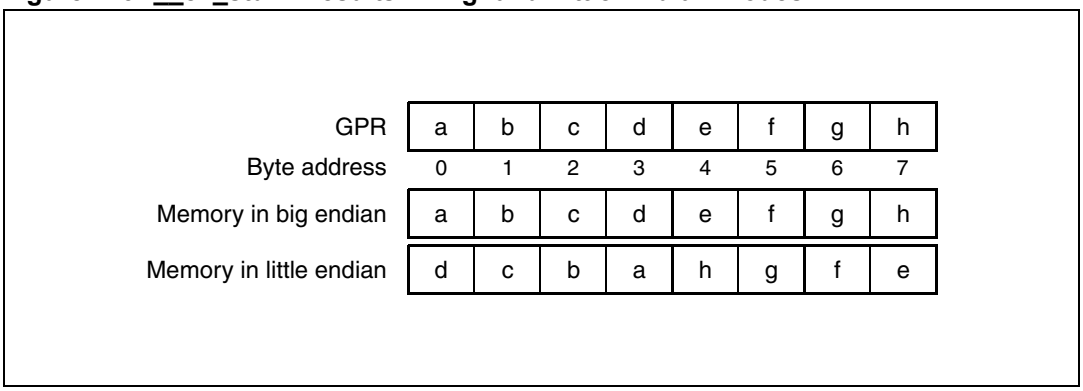
d = __ev_stdwx (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ← temp + (b)
MEM(EA,4) ← RS0:31
MEM(EA+4,4) ← RS32:63
```

The contents of **r**S are stored as two words in storage addressed by EA.

Figure 213 shows how bytes are stored in memory as determined by the endian mode.

Figure 213. __ev_stdwx results in Big- and Little-Endian modes



Note: During implementation, an alignment exception occurs if the EA is not double-word aligned.

Table 220. __ev_stdwx (registers altered by).

d	a	b	c	Maps to
void	__ev64_opaque	__ev64_opaque	int32_t	**evstdwx d,a,b,c**

__ev_stwhe

Vector Store Word of Two Half Words from Even

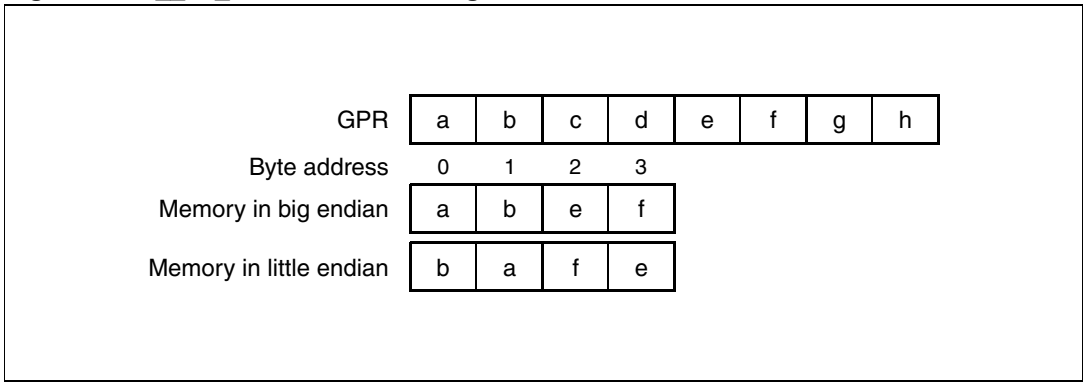
d = __ev_stwhe (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ← temp + EXTZ(UIMM*4)
MEM(EA,2) ← RS0:15
MEM(EA+2,2) ← RS32:47
```

The even half words from each element of **r**S are stored as two half words in storage addressed by EA.

Figure 214 shows how bytes are stored in memory as determined by the endian mode.

Figure 214. __ev_stwhe results in Big- and Little-Endian modes



Note: During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 221. __ev_stwhe (registers altered by).

d	a	b	c	Maps to
void	__ev64_opaque	uint32_t	5-bit unsigned	**evstdwhe d,a,b**

__ev_stwhex

Vector Store Word of Two Half Words from Even Indexed

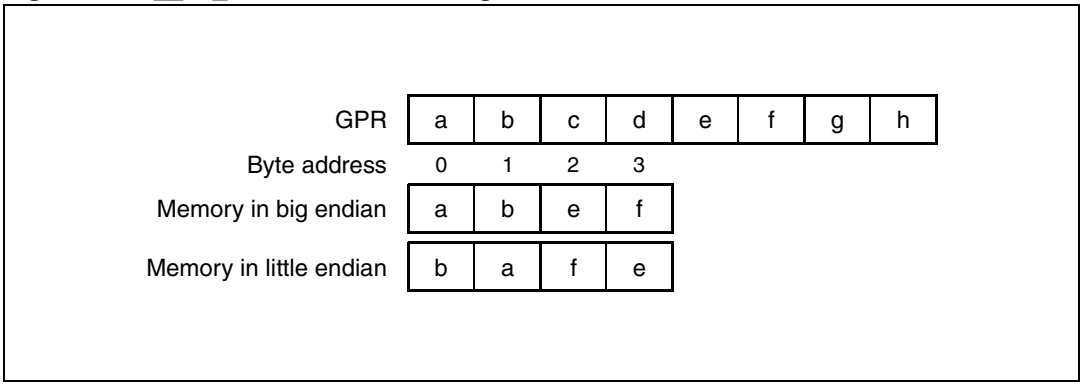
d = __ev_stwhex (a,b,c)

```
if (a = 0) then temp← 0
else temp ←(a)
EA ← temp + (b)
MEM(EA,2) ← RS0:15
MEM(EA+2,2) ← RS32:47
```

The even half words from each element of **r**S are stored as two half words in storage addressed by EA.

Figure 215 shows how bytes are stored in memory as determined by the endian mode.

Figure 215. __ev_stwhex results in Big- and Little-Endian modes



Note: During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 222. __ev_stwhex (registers altered by).

d	a	b	c	Maps to
void	__ev64_opaque	uint32_t	int32_t	**evstwhex d,a,b,c**

__ev_stwho

Vector Store Word of Two Half Words from Odd

d = __ev_stwho (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ← temp + EXTZ(UIMM*4)
MEM(EA,2) ← RS16:31
MEM(EA+2,2) ← RS48:63
```

The odd half words from each element of **r**S are stored as two half words in storage addressed by EA.

Figure 216. __ev_stwho results in Big- and Little-Endian modes

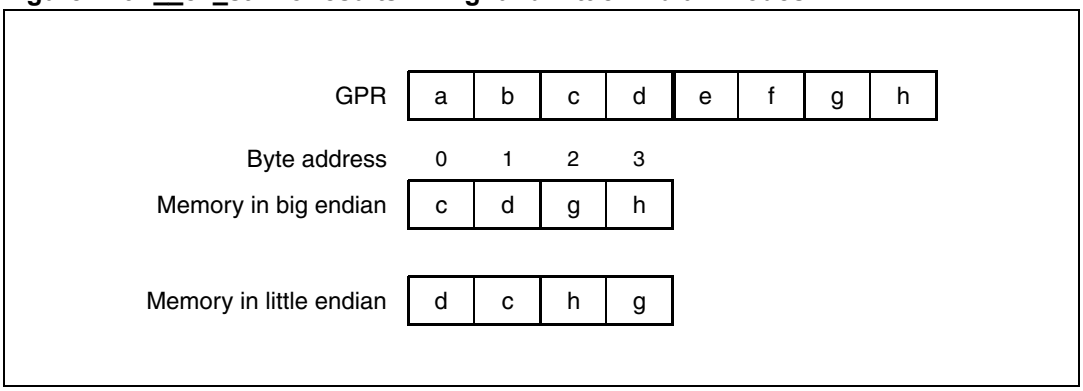


Table 223. __ev_stwho (registers altered by).

d	a	b	c	Maps to
void	__ev64_opaque	uint32_t	5-bit unsigned	**evstwho d,a,b,c**

__ev_stwhox

Vector Store Word of Two Half Words from Odd Indexed

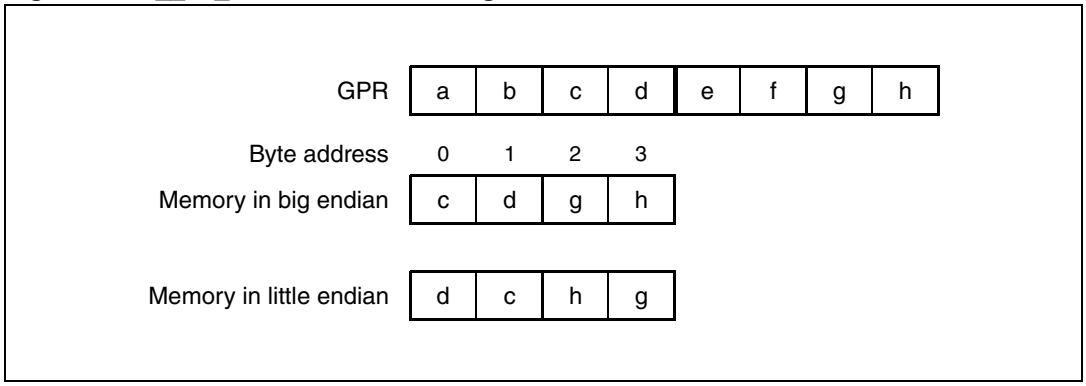
d = __ev_stwhox (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ← temp + (b)
MEM(EA,2) ← RS16:31
MEM(EA+2,2) ← RS48:63
```

The odd half words from each element of **r**S are stored as two half words in storage addressed by EA.

Figure 217 shows how bytes are stored in memory as determined by the endian mode.

Figure 217. __ev_stwhox results in Big- and Little-Endian modes



Note: During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 224. __ev_stwhox (registers altered by).

d	a	b	c	Maps to
void	__ev64_opaque	uint32_t	int32_t	**evstwhox d,a,b,c**

__ev_stwwe

Vector Store Word of Word from Even

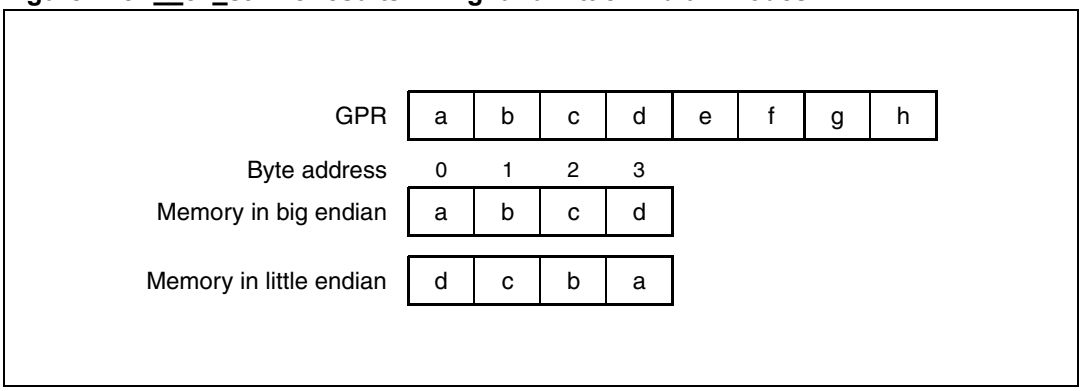
d = __ev_stwwe (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ← temp + EXTZ(UIMM*4)
MEM(EA,4) ← RS0:31
```

The even word of rS is stored in storage addressed by EA.

Figure 218 shows how bytes are stored in memory as determined by the endian mode.

Figure 218. __ev_stwwe results in Big- and Little-Endian modes



Note: During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 225. __ev_stwwe (registers altered by).

d	a	b	c	Maps to
void	__ev64_opaque	uint32_t	5-bit unsigned	**evstwwe d,a,b,c**

__ev_stwwex

Vector Store Word of Word from Even Indexed

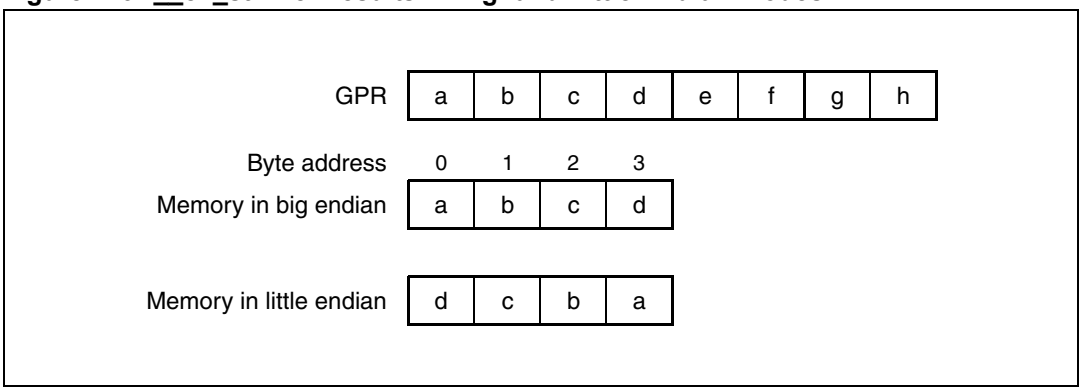
d = __ev_stwwex (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ← temp + (b)
MEM(EA,4) ← RS0:31
```

The even word of **r**S is stored in storage addressed by EA.

Figure 219 shows how bytes are stored in memory as determined by the endian mode.

Figure 219. __ev_stwwex results in Big- and Little-Endian modes



Note: During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 226. __ev_stwwex (registers altered by).

d	a	b	c	Maps to
void	__ev64_opaque	uint32_t	int32_t	**evstwwex d,a,b,c**

__ev_stwwo

Vector Store Word of Word from Odd

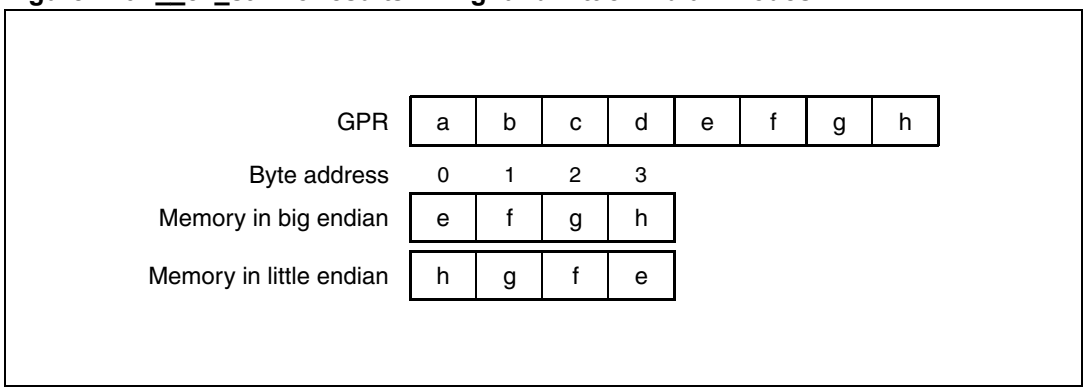
d =__ev_stwwo (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ← temp + EXTZ(UIMM*4)
MEM(EA,4) ← rS32:63
```

The odd word of **r**S is stored in storage addressed by EA.

Figure 220 shows how bytes are stored in memory as determined by the endian mode.

Figure 220. __ev_stwwo results in Big- and Little-Endian modes



Note: During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 227. __ev_stwwo (registers altered by).

d	a	b	c	Maps to
void	__ev64_opaque	uint32_t	5-bit unsigned	**evstwwo d,a,b,c**

__ev_stwwox

Vector Store Word of Word from Odd Indexed

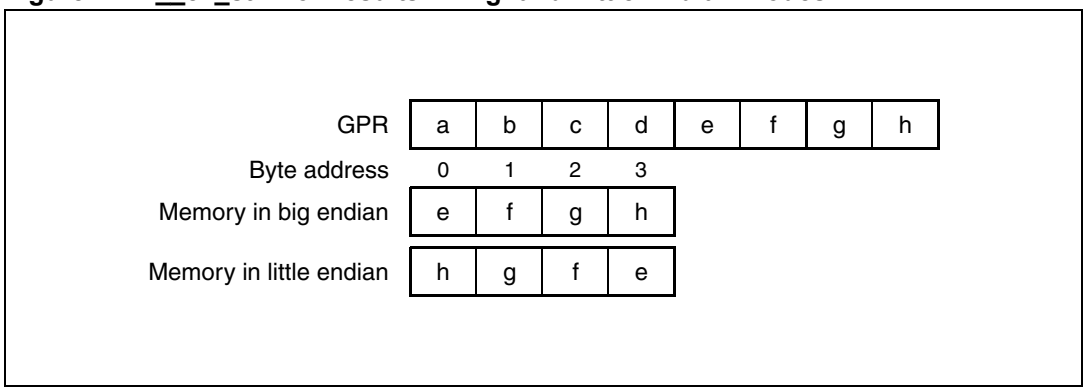
d = __ev_stwwox(a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ← temp + (b)
MEM(EA,4) ← rS32:63
```

The odd word of **r**S is stored in storage addressed by EA.

Figure 221 shows how bytes are stored in memory as determined by the endian mode.

Figure 221. __ev_stwwox results in Big- and Little-Endian modes



Note: During implementation, an alignment exception occurs if the EA is not word-aligned.

Table 228. __ev_stwwox (registers altered by).

d	a	b	c	Maps to
void	__ev64_opaque	uint32_t	int32_t	**evstwwox d,a,b,c**

__ev_subfsmiaaw

Vector Subtract Signed, Modulo, Integer to Accumulator Word

d = __ev_subfsmiaaw(a)

```
// high
d0:31 ← ACC0:31 - a0:31
// low
d32:63 ← ACC32:63 - a32:63
// update accumulator
ACC0:63 ← d0:63
```

Each word element in parameter a is subtracted from the corresponding element in the accumulator and the difference is placed into the corresponding parameter d word and into the accumulator.

Other registers altered: ACC

Figure 222. Vector subtract signed, modulo, integer to accumulator Word (__ev_subfsmiaaw)

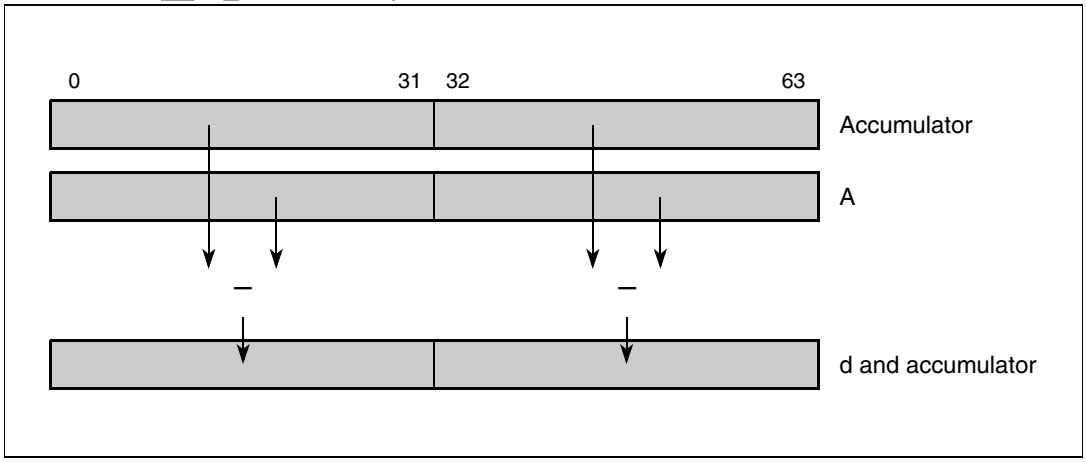


Table 229. __ev_subfsmiaaw (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evsubfsmiaaw d,a**

__ev_subfssiaaw

Vector Subtract Signed, Saturate, Integer to Accumulator Word

d = __ev_subfssiaaw(a)

```
// high
temp0:63 ← EXTS(ACC0:31) - EXTS(a0:31)
ovh ← temp31 ⊕ temp32
d0:31 ← SATURATE(ovh, temp31, 0x80000000, 0x7fffffff, temp32:63)
// low
temp0:63 ← EXTS(ACC32:63) - EXTS(a32:63)
ovl ← temp31 ⊕ temp32
d32:63 ← SATURATE(ovl, temp31, 0x80000000, 0x7fffffff, temp32:63)

// update accumulator
ACC0:63 ← d0:63

SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

Each signed integer word element in parameter a is sign-extended and subtracted from the corresponding sign-extended element in the accumulator, saturating if overflow occurs, and the results are placed in parameter d and the accumulator. Any overflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

Figure 223. Vector subtract signed, saturate, integer to accumulator Word (__ev_subfssiaaw)

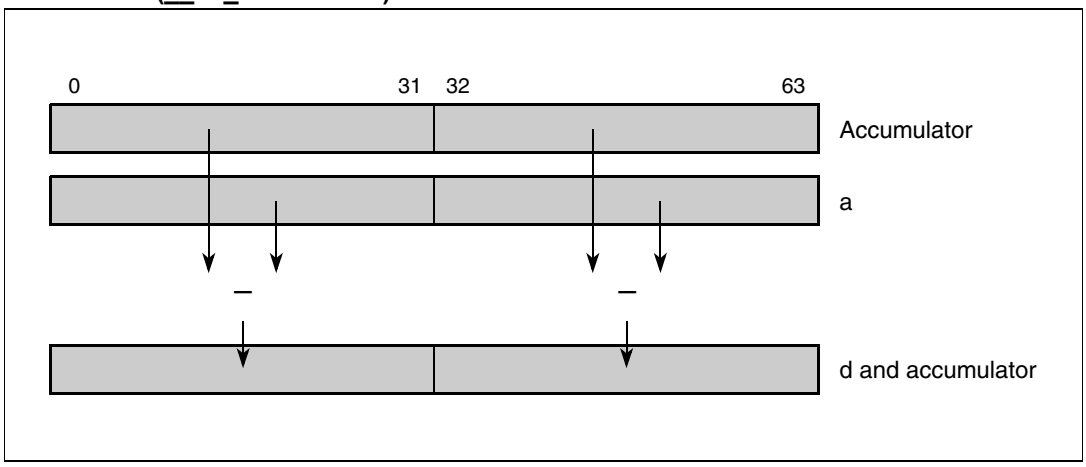


Table 230. __ev_subfssiaaw (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evsubfssiaaw d,a**

__ev_subfumiaaw

Vector Subtract Unsigned, Modulo, Integer to Accumulator Word

d = __ev_subfumiaaw(a)

```
// high
d0:31 ← ACC0:31 - a0:31
// low
d32:63 ← ACC32:63 - a32:63
// update accumulator
ACC0:63 ← d0:63
```

Each unsigned integer word element in parameter a is subtracted from the corresponding element in the accumulator, and the results are placed in the corresponding parameter d and into the accumulator.

Other registers altered: ACC

Figure 224. Vector subtract unsigned, modulo, integer to accumulator Word (__ev_subfumiaaw)

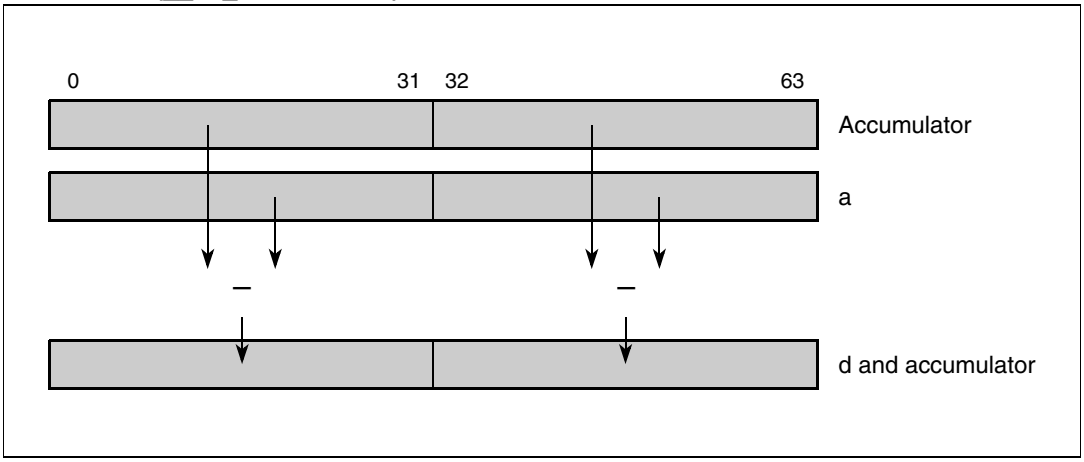


Table 231. __ev_subfumiaaw (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evsubfumiaaw d,a**

__ev_subfusiaaw

Vector Subtract Unsigned, Saturate, Integer to Accumulator Word

d = __ev_subfusiaaw(a)

```
// high
temp0:63 ← EXTZ(ACC0:31) - EXTZ(a0:31)
ovh ← temp31
d0:31 ← SATURATE(ovh, temp31, 0x00000000, 0x00000000, temp32:63)
// low
temp0:63 ← EXTS(ACC32:63) - EXTS(a32:63)
ovl ← temp31
d32:63 ← SATURATE(ovl, temp31, 0x00000000, 0x00000000, temp32:63)

// update accumulator
ACC0:63 ← d0:63

SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

Each unsigned integer word element in parameter a is zero-extended and subtracted from the corresponding zero-extended element in the accumulator, saturating if underflow occurs, and the results are placed in parameter d and the accumulator. Any underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

Figure 225. Vector subtract unsigned, saturate, integer to accumulator Word (__ev_subfusiaaw)

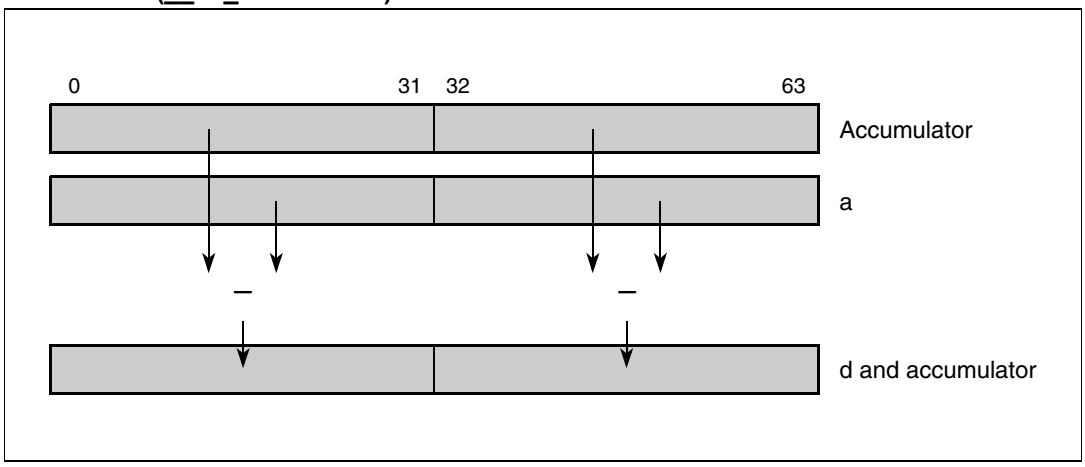


Table 232. __ev_subfusiaaw (registers altered by).

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evsubfusiaaw d,a**

__ev_subfw

Vector Subtract from Word

d = __ev_subfw(a,b)

$d_{0:31} \leftarrow b_{0:31}$ - $a_{0:31}$ // Modulo difference
$d_{32:63} \leftarrow b_{32:63}$ - $a_{32:63}$ // Modulo difference

Each signed integer element of parameter a is subtracted from the corresponding element of parameter b, and the results are placed into parameter d.

Figure 226. Vector subtract from word (__ev_subfw)

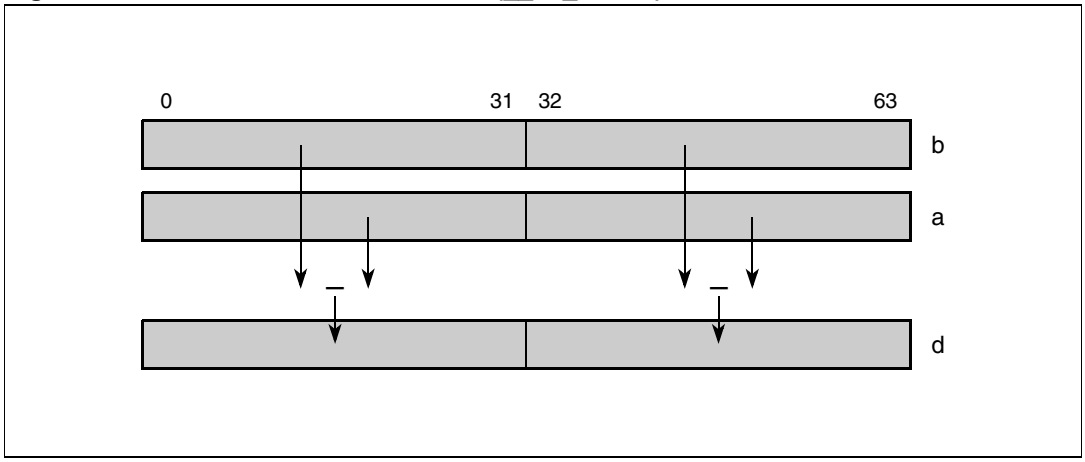


Table 233. __ev_subfw (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evsubfw d,a,b**

__ev_subifw

Vector Subtract Immediate from Word

d = __ev_subifw(a,b)

```
d0:31 ← b0:31  - EXTZ(UIMM) // Modulo difference
d32:63 ← b32:63 - EXTZ(UIMM)// Modulo difference
```

UIMM is zero-extended and subtracted from both the high and low elements of parameter b. Note that the same value is subtracted from both elements of the register. UIMM is 5 bits.

Figure 227. Vector subtract immediate from word (__ev_subifw)

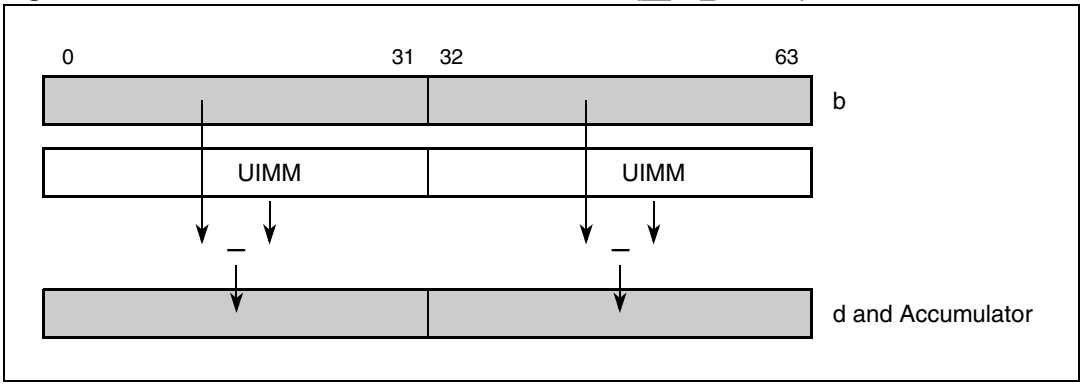


Table 234. __ev_subifw (registers altered by).

d	a	b	Maps to
__ev64_opaque	5-bit unsigned	__ev64_opaque	**evsubifw d,a,b**

__ev_upper_eq

Vector Upper Bits Equal

d = __ev_upper_eq(a,b)

```
if (a0:31 = b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b.

Figure 228. Vector upper Equal(__ev_upper_eq)

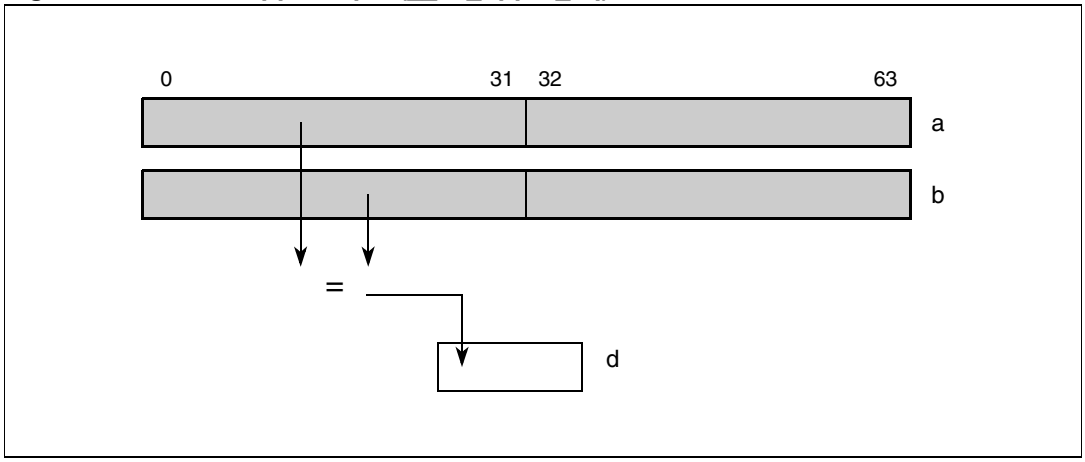


Table 235. __ev_upper_eq (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpeq x,a,b**

__ev_upper_fs_eq

Vector Upper Bits Floating-Point Equal

d = __ev_upper_fs_eq(a,b)

```
if (a0:31 = b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b.

Figure 229. Vector upper Floating-Point Equal(__ev_upper_fs_eq)

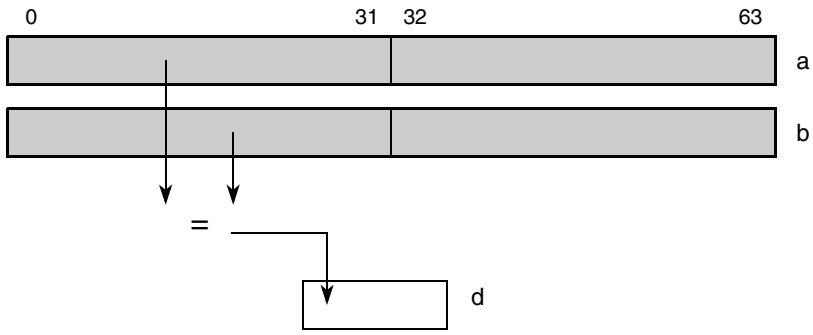


Table 236. __ev_upper_fs_eq (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmpeq x,a,b**

__ev_upper_fs_gt

Vector Upper Bits Floating-Point Greater Than

d = __ev_upper_fs_gt(a,b)

```
if (a0:31 > b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b.

Figure 230. Vector upper Floating-Point greater than (__ev_upper_fs_gt)

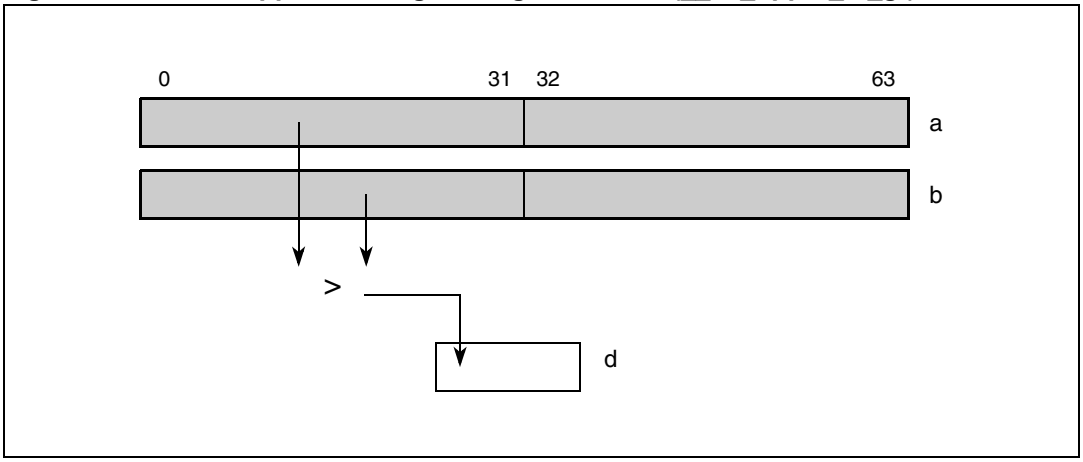


Table 237. __ev_upper_fs_gt (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmpgt x,a,b**

__ev_upper_fs_lt

Vector Upper Bits Floating-Point Less Than

d = __ev_upper_fs_lt(a,b)

```
if (a0:31 < b0:31) then d← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are less than the upper 32 bits of parameter b.

Figure 231. Vector upper Floating-Point less than (__ev_upper_fs_lt)

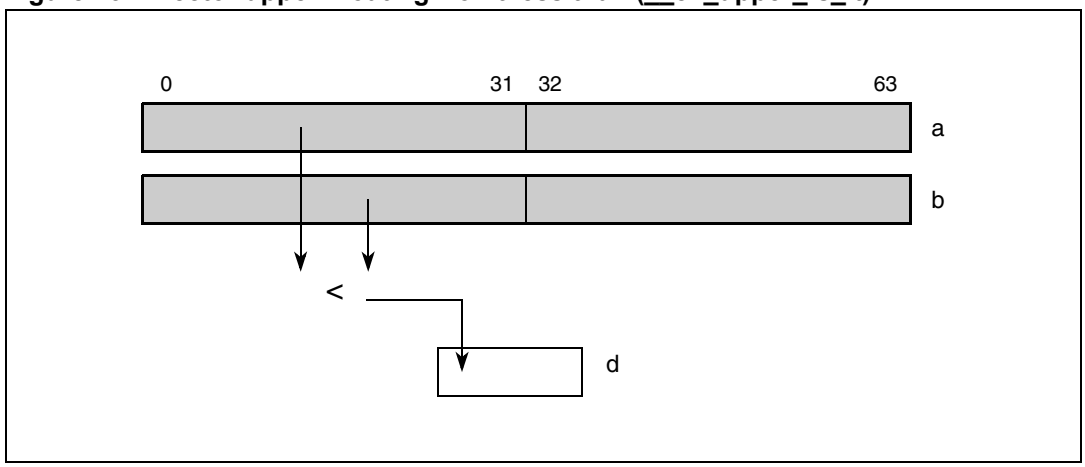


Table 238. __ev_upper_fs_lt (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmplt x,a,b**

__ev_upper_fs_tst_eq

Vector Upper Bits Floating-Point Test Equal

d = __ev_upper_fs_tst_eq(a,b)

```
if (a0:31 = b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b. This intrinsic differs from __ev_upper_fs_eq because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_upper_fs_eq instead.

Figure 232. Vector upper Floating-Point test equal (__ev_upper_fs_tst_eq)

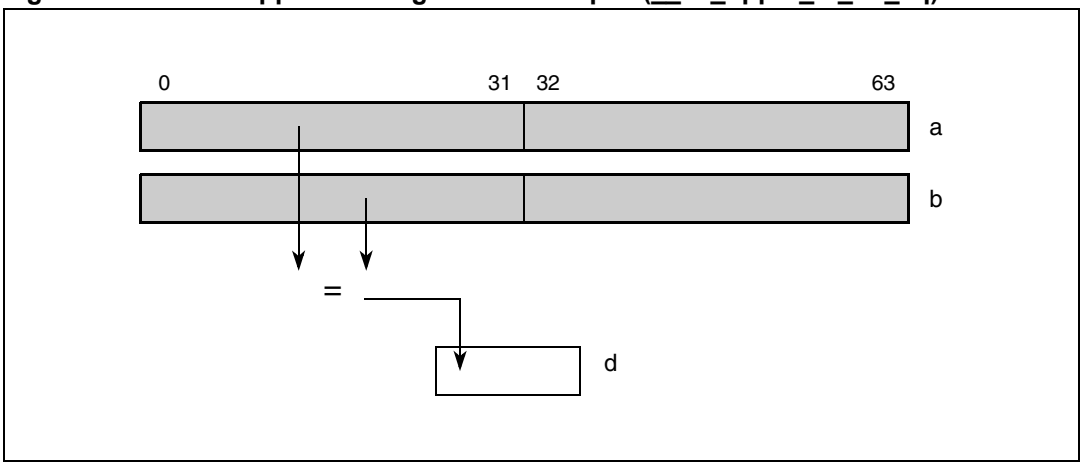


Table 239. __ev_upper_fs_tst_eq (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststeq x,a,b**

__ev_upper_fs_tst_gt

Vector Upper Bits Floating-Point Test Greater Than

d = __ev_upper_fs_tst_gt(a,b)

```
if (a0:31 > b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b. This intrinsic differs from __ev_upper_fs_gt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_upper_fs_gt instead.

Figure 233. Vector upper Floating-Point test greater than (__ev_upper_fs_tst_gt)

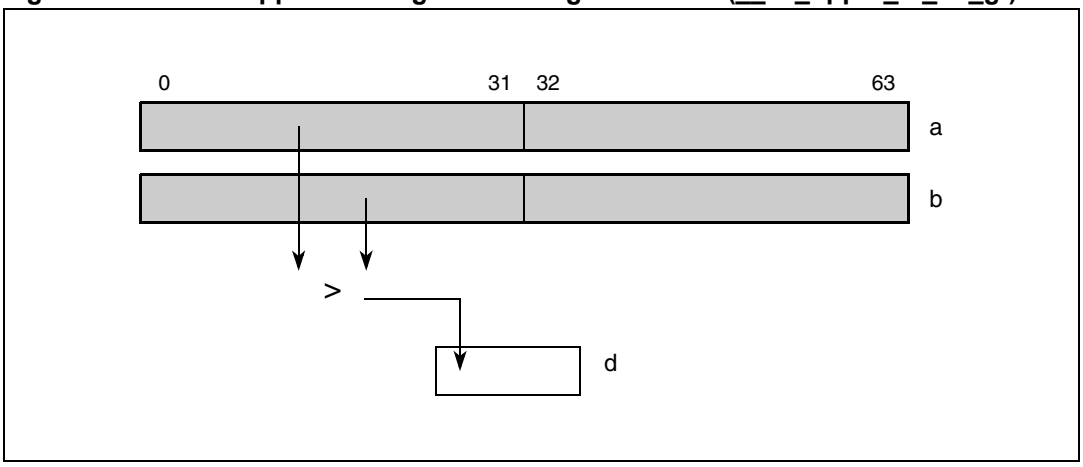


Table 240. __ev_upper_fs_tst_gt (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststgt x,a,b**

__ev_upper_fs_tst_lt

Vector Upper Bits Floating-Point TestLess Than

d = __ev_upper_fs_tst_lt(a,b)

```
if (a0:31 < b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are less than the upper 32 bits of parameter b. This intrinsic differs from __ev_upper_fs_lt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_upper_fs_lt instead.

Figure 234. Vector upper Floating-Point test less than (__ev_upper_fs_tst_lt)

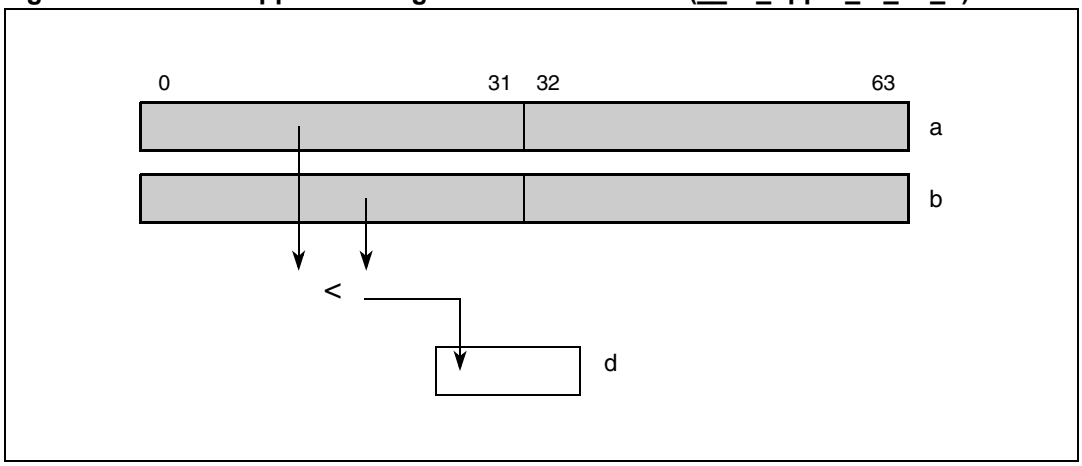


Table 241. __ev_upper_fs_tst_lt (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	evfststlt x,a,b

__ev_upper_gts

Vector Upper Bits Greater Than Signed

d = __ev_upper_gts(a,b)

```
if (a0:31 >signed b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b.

Figure 235. Vector upper greater than signed (__ev_upper_gts)

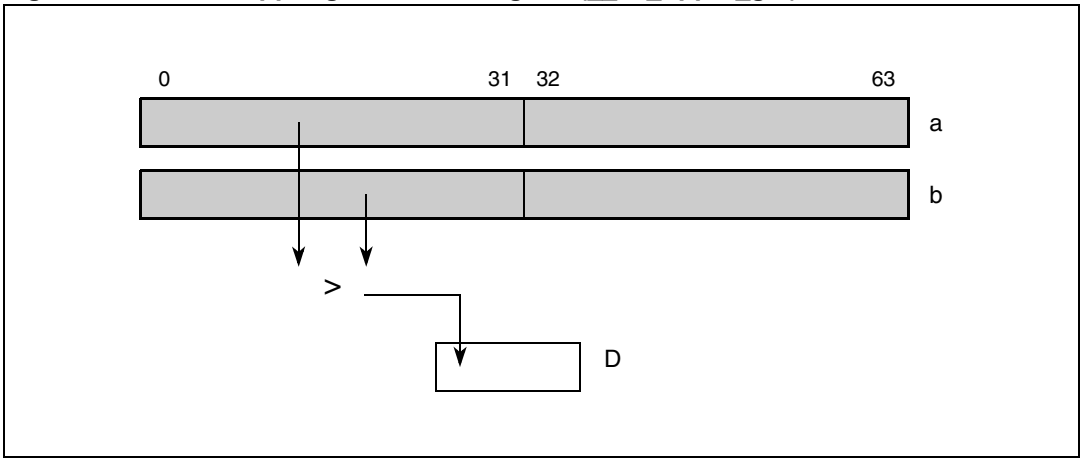


Table 242. __ev_upper_gts (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	evcmpgts x,a,b

__ev_upper_gtu

Vector Upper Bits Greater Than Unsigned

d = __ev_upper_gtu(a,b)

```
if (a0:31 > unsigned b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b.

Figure 236. Vector upper greater than unsigned (__ev_upper_gtu)

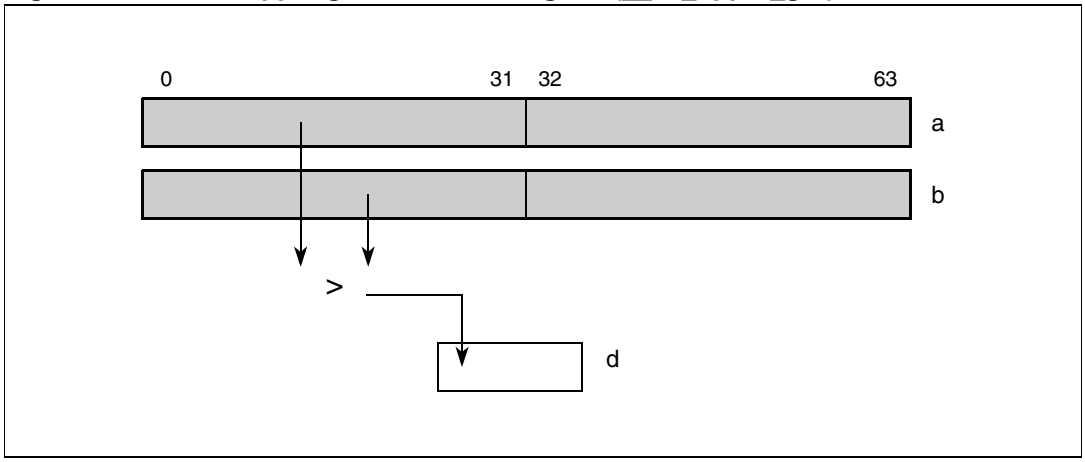


Table 243. __ev_upper_gtu (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	evcmpgtu x,a,b

__ev_upper_lts

Vector Upper Bits Less Than Signed

d = __ev_upper_lts(a,b)

```
if (a0:31 <signed b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are less than the upper 32 bits of parameter b.

Figure 237. Vector upper less than signed (__ev_upper_lts)

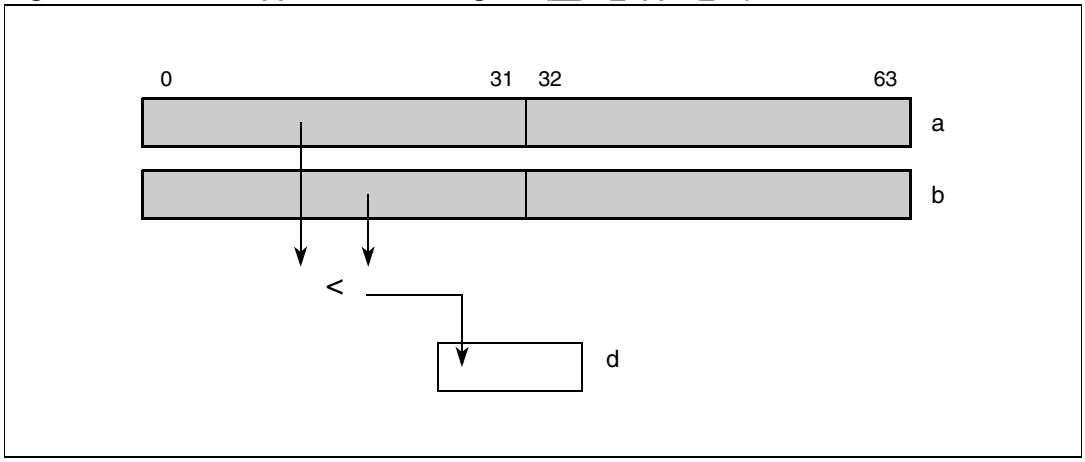


Table 244. __ev_upper_lts (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	evcmplts x,a,b

__ev_upper_ltu

Vector Upper Bits Less Than Unsigned

```
d = __ev_upper_ltu(a,b)
if (a0:31 <unsigned b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are less than the upper 32 bits of parameter b.

Figure 238. Vector upper less than unsigned (__ev_upper_ltu)

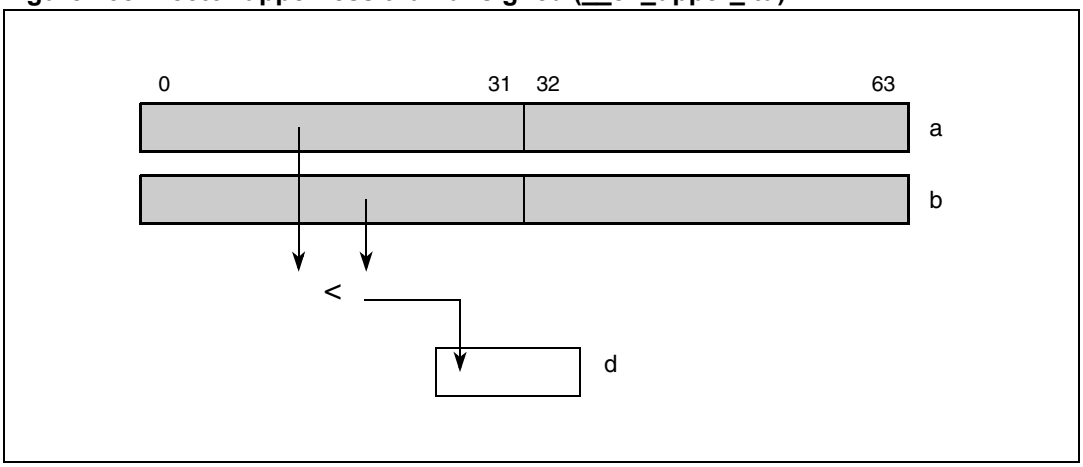


Table 245. __ev_upper_ltu (registers altered by).

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	evcmpltu x,a,b

__ev_xor

Vector XOR

d = __ev_xor (a,b)

$d_{0:31} \leftarrow a_{0:31} \oplus b_{0:31}$ `// Bitwise XOR`
$d_{32:63} \leftarrow a_{32:63} \oplus b_{32:63}$`// Bitwise XOR`

Each element of parameters a and b is exclusive-ORed. The results are placed in parameter d.

Figure 239. Vector XOR (__ev_xor)

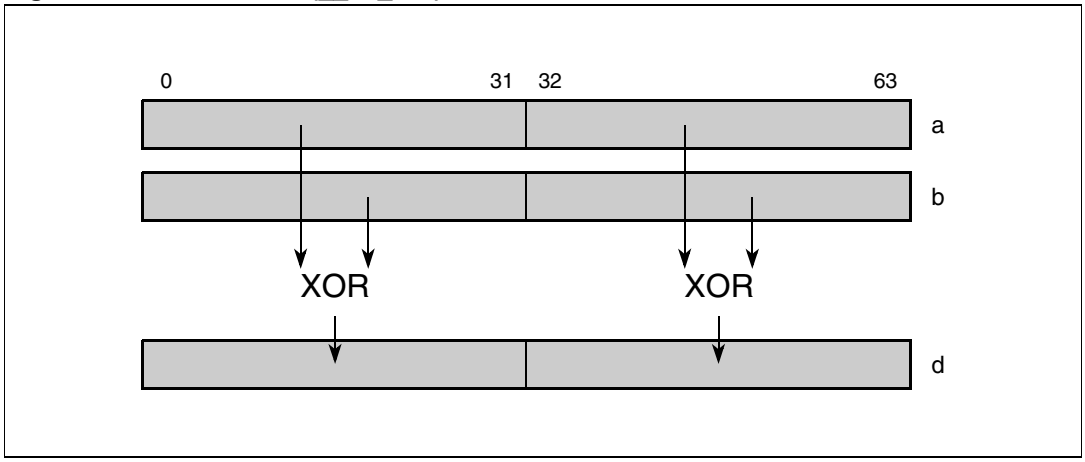


Table 246. __ev_xor (registers altered by).

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evxor d,a,b**

3.6 Basic instruction mapping

```
__ev64_opaque__ __ev_addw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addiw( __ev64_opaque__ a, 5-bit unsigned literal );

// returns ( B - A )
__ev64_opaque__ __ev_subfw( __ev64_opaque__ a, __ev64_opaque__ b );

// returns ( B - UIMM )
__ev64_opaque__ __ev_subifw( 5-bit unsigned literal, __ev64_opaque__ b );

// returns ( A - B )
__ev64_opaque__ __ev_subw( __ev64_opaque__ a, __ev64_opaque__ b );

// returns ( A - UIMM )
__ev64_opaque__ __ev_subiw( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_abs( __ev64_opaque__ a );
__ev64_opaque__ __ev_neg( __ev64_opaque__ a );
__ev64_opaque__ __ev_extsb( __ev64_opaque__ a );
__ev64_opaque__ __ev_extsh( __ev64_opaque__ a );
__ev64_opaque__ __ev_and( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_or( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_xor( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_nand( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_nor( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_eqv( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_andc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_orc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_rlw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_rlwi( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_slw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_slwi( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_srws( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_srwu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_srwis( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_srwiu( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_cntlzw( __ev64_opaque__ a );
__ev64_opaque__ __ev_cntlsw( __ev64_opaque__ a );
__ev64_opaque__ __ev_rndw( __ev64_opaque__ a );
__ev64_opaque__ __ev_mergehi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mergelo( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mergelohi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mergehilo( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_splati( 5-bit signed literal );
__ev64_opaque__ __ev_splatfi( 5-bit signed literal );
__ev64_opaque__ __ev_divws( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_divwu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mra( __ev64_opaque__ a ); uint
32_t __brinc( uint32_t a, uint32_t b );

# COMPARE PREDICATES
```

Note: The __ev_select_* operations work much like the ? : operator does in C. For example: `__ev_select_gts(a,b,c,d) maps to the logical expression a > b ? c : d.`

The following code shows an example of the assembly code:

```
   evcmpgts crfD, A, B
   evsel ret, C, D, crfD
_Bool __ev_any_gts( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_gts( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_gts( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_gts( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_gts( __ev64_opaque__ a, __ev64_opaque__ b,
                      __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_gtu(__ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_gtu( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_gtu( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_gtu( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_gtu( __ev64_opaque__ a, __ev64_opaque__ b,
                      __ev64_opaque__ c, __ev64_opaque__ d);
_Bool __ev_any_lts( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_lts( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_lts( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_lts( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_lts( __ev64_opaque__ a, __ev64_opaque__ b,
                      __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_ltu( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_ltu( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_ltu( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_ltu( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_ltu( __ev64_opaque__ a, __ev64_opaque__ b,
                      __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_eq( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_eq( __ev64_opaque__ a, __ev64_opaque__ b,
                     __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_fs_gt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_fs_gt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_fs_gt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_fs_gt( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_fs_gt( __ev64_opaque__ a, __ev64_opaque__ b,
                        __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_fs_lt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_fs_lt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_fs_lt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_fs_lt( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_fs_lt( __ev64_opaque__ a, __ev64_opaque__ b,
                        __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_fs_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_fs_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_fs_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_fs_eq( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_fs_eq( __ev64_opaque__ a, __ev64_opaque__ b,
                        __ev64_opaque__ c, __ev64_opaque__ d);
```

```
_Bool __ev_any_fs_tst_gt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_fs_tst_gt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_fs_tst_gt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_fs_tst_gt( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_fs_tst_gt( __ev64_opaque__ a, __ev64_opaque__
b,
                              __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_fs_tst_lt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_fs_tst_lt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_fs_tst_lt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_fs_tst_lt( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_fs_tst_lt( __ev64_opaque__ a, __ev64_opaque__
b,
                              __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_fs_tst_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_fs_tst_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_fs_tst_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_fs_tst_eq( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_fs_tst_eq( __ev64_opaque__ a, __ev64_opaque__
b,
                              __ev64_opaque__ c, __ev64_opaque__ d);
# LOAD/STORE
```

Note: The 5-bit unsigned literal in the immediate form is scaled by the size of the load or store to determine how many bytes the pointer 'p' is offset by. The size of the load is determined by the first letter after the 'l': 'd'—double-word (8 bytes), 'w'—word (4 bytes), 'h'—half word (2 bytes). For details, see Chapter 5".

```
__ev64_opaque__ __ev_lddx( __ev64_opaque__ * p, int32_t offset );
__ev64_opaque__ __ev_lddx( __ev64_opaque__ * p, int32_t offset );
__ev64_opaque__ __ev_ldwx( __ev64_opaque__ * p, int32_t offset );
__ev64_opaque__ __ev_ldhx( __ev64_opaque__ * p, int32_t offset );
__ev64_opaque__ __ev_lwhex( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwhoux( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwhosx( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwwsplatx( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwhsplatx( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lhhesplatx( uint16_t * p, int32_t offset );
__ev64_opaque__ __ev_lhhousplatx( uint16_t * p, int32_t offset );
__ev64_opaque__ __ev_lhhossplatx( uint16_t * p, int32_t offset );

__ev64_opaque__ __ev_ldd( __ev64_opaque__ * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_ldw( __ev64_opaque__ * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_ldh( __ev64_opaque__ * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhe( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhou( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhos( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwwsplat( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhsplat( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lhhesplat( uint16_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lhhousplat( uint16_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lhhossplat( uint16_t * p, 5-bit unsigned literal );

void __ev_stddx( __ev64_opaque__ a, __ev64_opaque__ * p, int32_t offset );
```

```
void __ev_stdwx( __ev64_opaque__ a, __ev64_opaque__ * p, int32_t offset );
void __ev_stdhx( __ev64_opaque__ a, __ev64_opaque__ * p, int32_t offset );
void __ev_stwwex( __ev64_opaque__ a, uint32_t * p, int32_t offset );
void __ev_stwwox( __ev64_opaque__ a, uint32_t * p, int32_t offset );
void __ev_stwhex( __ev64_opaque__ a, uint32_t * p, int32_t offset );
void __ev_stwhox( __ev64_opaque__ a, uint32_t * p, int32_t offset );

void __ev_stdd( __ev64_opaque__ a, __ev64_opaque__ * p, 5-bit unsigned
literal );
void __ev_stdw( __ev64_opaque__ a, __ev64_opaque__ * p, 5-bit unsigned
literal );
void __ev_stdh( __ev64_opaque__ a, __ev64_opaque__ * p, 5-bit unsigned
literal );
void __ev_stwwe( __ev64_opaque__ a, uint32_t * p, 5-bit unsigned literal );
void __ev_stwwo( __ev64_opaque__ a, uint32_t * p, 5-bit unsigned literal );
void __ev_stwhe( __ev64_opaque__ a, uint32_t * p, 5-bit unsigned literal );
void __ev_stwho( __ev64_opaque__ a, uint32_t * p, 5-bit unsigned literal );


*** FIXED-POINT COMPLEX ***

__ev64_opaque__ __ev_mhossf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhoumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhossfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhoumia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheumia( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mhoumi
 __ev64_opaque__ __ev_mhoumf( __ev64_opaque__ a, __ev64_opaque__ b );
// maps to __ev_mheumi
 __ev64_opaque__ __ev_mheumf( __ev64_opaque__ a, __ev64_opaque__ b );
// maps to __ev_mhoumia
 __ev64_opaque__ __ev_mhoumfa( __ev64_opaque__ a, __ev64_opaque__ b );
// maps to __ev_mheumia
 __ev64_opaque__ __ev_mheumfa( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mhossfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhossiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhousiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhoumiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
```

```
__ev64_opaque__ __ev_mheusiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheumiaaw( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mhousiaaw
 __ev64_opaque__ __ev_mhousfaaw( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mhoumiaaw
 __ev64_opaque__ __ev_mhoumfaaw( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mheusiaaw
 __ev64_opaque__ __ev_mheusfaaw( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mheumiaaw
 __ev64_opaque__ __ev_mheumfaaw( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mhossfanw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhossianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmfanw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhousianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhoumianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessfanw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmfanw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheusianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheumianw( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mhousianw
 __ev64_opaque__ __ev_mhousfanw( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mhoumianw
 __ev64_opaque__ __ev_mhoumfanw( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mheusianw
 __ev64_opaque__ __ev_mheusfanw( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mheumianw
 __ev64_opaque__ __ev_mheumfanw( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mhogsmfaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhogsmiaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhogumiaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhegsmfaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhegsmiaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhegumiaa( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mhogumiaa
 __ev64_opaque__ __ev_mhogumfaa( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mhegumiaa
 __ev64_opaque__ __ev_mhegumfaa( __ev64_opaque__ a, __ev64_opaque__ b );


__ev64_opaque__ __ev_mhogsmfan( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhogsmian( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhogumian( __ev64_opaque__ a, __ev64_opaque__ b );
```

```
__ev64_opaque__ __ev_mhegsmfan( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhegsmian( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhegumian( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mhogumian
 __ev64_opaque__ __ev_mhogumfan( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mhegumian
 __ev64_opaque__ __ev_mhegumfan( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwhssf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhsmf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhsmi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhssfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhsmfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhsmia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhumia( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mwhumi
 __ev64_opaque__ __ev_mwhumf( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mwhumia
 __ev64_opaque__ __ev_mwhumfa( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwlumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlumia( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwlssiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlsmiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlusiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlumiaaw( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwlssianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlsmianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlusianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlumianw( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwhssfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhssf(a,b);
 __ev_addssiaaw(temp);

__ev64_opaque__ __ev_mwhssiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmi(a,b);
 __ev_addssiaaw(temp);

__ev64_opaque__ __ev_mwhsmfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmf(a,b);
 __ev_addsmiaaw(temp);

__ev64_opaque__ __ev_mwhsmiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmi(a,b);
 __ev_addsmiaaw(temp);

__ev64_opaque__ __ev_mwhusiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhumi(a,b);
 __ev_addusiaaw(temp);
```

```
__ev64_opaque__ __ev_mwhumiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhumi(a,b);
 __ev_addumiaaw(temp);

// maps to __ev_mwhusiaaw
 __ev64_opaque__ __ev_mwhusfaaw( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mwhumiaaw
 __ev64_opaque__ __ev_mwhumfaaw( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwhssfanw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhssf(a,b);
 __ev_subfssiaaw(temp);

__ev64_opaque__ __ev_mwhssianw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmi(a,b);
 __ev_subfssiaaw(temp);

__ev64_opaque__ __ev_mwhsmfanw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmf(a,b);
 __ev_subfsmiaaw(temp);

__ev64_opaque__ __ev_mwhsmianw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmi(a,b);
 __ev_subfsmiaaw(temp);

__ev64_opaque__ __ev_mwhusianw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhumi(a,b);
 __ev_subfusiaaw(temp);

__ev64_opaque__ __ev_mwhumianw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhumi(a,b);
 __ev_subfumiaaw(temp);

**
__ev64_opaque__ __ev_mwhgssfaa( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhssf(a, b);
// Note: the upper 32 bits of the immediate is a do not care.
// Therefore we spec {1, 1} because it can easily be generated by a
// __ev_splati(1)
 __ev_mwsmiaa(temp, (__ev64_u32__){1, 1});

__ev64_opaque__ __ev_mwhgsmfaa( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmf(a, b);
// Note: the upper 32 bits of the immediate is a do not care.
// Therefore we spec {1, 1} because it can easily be generated by a
// __ev_splati(1)
 __ev_mwsmiaa(temp, (__ev64_u32__){1, 1});

__ev64_opaque__ __ev_mwhgsmiaa( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmi(a, b);
// Note: the upper 32 bits of the immediate is a do not care.
// Therefore we spec {1, 1} because it can easily be generated by a
// __ev_splati(1)
```

```
 __ev_mwsmiaa(temp, (__ev64_u32__){1, 1});

__ev64_opaque__ __ev_mwhgumiaa( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhumi(a, b);
// Note: the upper 32 bits of the immediate is a do not care.
// Therefore we spec {1, 1} because it can easily be generated by a
// __ev_splati(1)
 __ev_mwumiaa(temp, (__ev64_u32__){1, 1});

// maps to __ev_mwhgumiaa
 __ev64_opaque__ __ev_mwhgumfaa( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwhgssfan( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhssf(a, b);
// Note: the upper 32 bits of the immediate is a do not care.
// Therefore we spec {1, 1} because it can easily be generated by a
// __ev_splati(1)
 __ev_mwsmian(temp, (__ev64_u32__){1, 1});

__ev64_opaque__ __ev_mwhgsmfan( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmf(a, b);
// Note: the upper 32 bits of the immediate is a do not care.
// Therefore we spec {1, 1} because it can easily be generated by a
// __ev_splati(1)
 __ev_mwsmian(temp, (__ev64_u32__){1, 1});

__ev64_opaque__ __ev_mwhgsmian( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmi(a, b);
// Note: the upper 32 bits of the immediate is a do not care.
// Therefore we spec {1, 1} because it can easily be generated by a
// __ev_splati(1)
 __ev_mwsmian(temp, (__ev64_u32__){1, 1});

__ev64_opaque__ __ev_mwhgumian( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhumi(a, b);
// Note: the upper 32 bits of the immediate is a do not care.
// Therefore we spec {1, 1} because it can easily be generated by a
// __ev_splati(1)
 __ev_mwumian(temp, (__ev64_u32__){1, 1});

// maps to __ev_mwhgumian
 __ev64_opaque__ __ev_mwhgumfan( __ev64_opaque__ a, __ev64_opaque__ b );
```

Note: An optimizing compiler should be able to improve performance by scheduling the instructions implementing an intrinsic, that is, __ev_mwhgumfan.

```
** END OF NOT SUPPORTED **

__ev64_opaque__ __ev_mwssf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwumi( __ev64_opaque__ a, __ev64_opaque__ b );
```

```
__ev64_opaque__ __ev_mwssfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwumia( __ev64_opaque__ a, __ev64_opaque__ b );
// maps to __ev_mwumi
 __ev64_opaque__ __ev_mwumf( __ev64_opaque__ a, __ev64_opaque__ b );
// maps to __ev_mwumia
 __ev64_opaque__ __ev_mwumfa( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwssfaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmfaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmiaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwumiaa( __ev64_opaque__ a, __ev64_opaque__ b );
// maps to __ev_mwumiaa
 __ev64_opaque__ __ev_mwumfaa( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwssfan( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmfan( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmian( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwumian( __ev64_opaque__ a, __ev64_opaque__ b );
// maps to __ev_mwumian
 __ev64_opaque__ __ev_mwumfan( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_addssiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_addsmiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_addusiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_addumiaaw( __ev64_opaque__ a );
// maps to __ev_addusiaaw
 __ev64_opaque__ __ev_addusfaaw( __ev64_opaque__ a );
// maps to __ev_addumiaaw
 __ev64_opaque__ __ev_addumfaaw( __ev64_opaque__ a );
// maps to __ev_addsmiaaw
 __ev64_opaque__ __ev_addsmfaaw( __ev64_opaque__ a );
// maps to __ev_addssiaaw
 __ev64_opaque__ __ev_addssfaaw( __ev64_opaque__ a );

__ev64_opaque__ __ev_subfssiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_subfsmiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_subfusiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_subfumiaaw( __ev64_opaque__ a );
// maps to __ev_subfusiaaw
 __ev64_opaque__ __ev_subfusfaaw( __ev64_opaque__ a );
// maps to __ev_subfumiaaw
 __ev64_opaque__ __ev_subfumfaaw( __ev64_opaque__ a );
// maps to __ev_subfsmiaaw
 __ev64_opaque__ __ev_subfsmfaaw( __ev64_opaque__ a );
// maps to __ev_subfssiaaw
 __ev64_opaque__ __ev_subfssfaaw( __ev64_opaque__ a );


# Floating-Point SIMD Instructions

__ev64_opaque__ __ev_fsabs( __ev64_opaque__ a);
__ev64_opaque__ __ev_fsnabs( __ev64_opaque__ a)
__ev64_opaque__ __ev_fsneg( __ev64_opaque__ a);
__ev64_opaque__ __ev_fsadd( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fssub( __ev64_opaque__ a, __ev64_opaque__ b);
```

```
__ev64_opaque__ __ev_fsmul( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fsdiv( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fscfui( __ev64_opaque__ b);
__ev64_opaque__ __ev_fscfsi( __ev64_opaque__ b);
__ev64_opaque__ __ev_fscfuf( __ev64_opaque__ b);
__ev64_opaque__ __ev_fscfsf( __ev64_opaque__ b);
__ev64_opaque__ __ev_fsctui( __ev64_opaque__ b);
__ev64_opaque__ __ev_fsctsi( __ev64_opaque__ b);
__ev64_opaque__ __ev_fsctuf( __ev64_opaque__ b);
__ev64_opaque__ __ev_fsctsf( __ev64_opaque__ b);
__ev64_opaque__ __ev_fsctuiz( __ev64_opaque__ b);
__ev64_opaque__ __ev_fsctsiz( __ev64_opaque__ b);

# creation/insertion/extraction
```

4 Additional operations

4.1 Data manipulation

The intrinsics in section one act like functions with parameters that are passed by value. *Figure 240* and *Figure 241* show the layout of a __ev64_opaque__ variable in the register with reference to creation, insertion, and extraction routines (regardless of endianess).

Figure 241 shows byte, half-word, and word ordering.

Figure 240. Big-endian word ordering

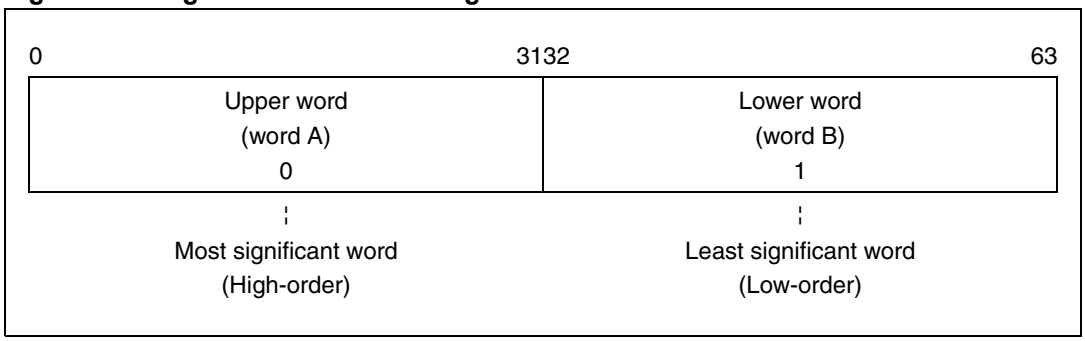
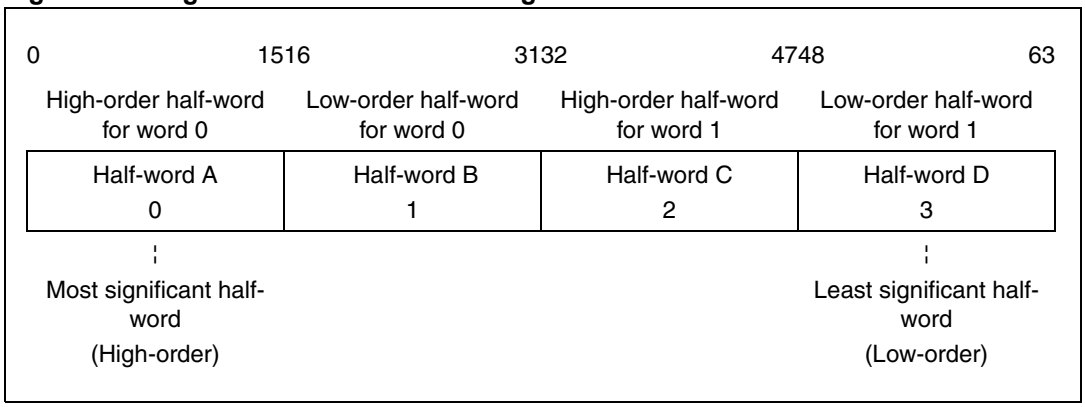


Figure 241. Big-endian half-word ordering



4.1.1 Creation intrinsics

These intrinsics create new generic 64-bit opaque data types from the given inputs passed by value. More specifically, they are created from the following inputs: 1 signed or unsigned 64-bit integer, 2 single-precision floats, 2 signed or unsigned 32-bit integers, or 4 signed or unsigned 16-bit integers.

```
__ev64_opaque__ __ev_create_u64( uint64_t a );
__ev64_opaque__ __ev_create_s64( int64_t a );
__ev64_opaque__ __ev_create_fs( float a, float b );
__ev64_opaque__ __ev_create_u32( uint32_t a, uint32_t b );
__ev64_opaque__ __ev_create_s32( int32_t a, int32_t b );
__ev64_opaque__ __ev_create_u16( uint16_t a, uint16_t b, uint16_t c, uint16_t d );
__ev64_opaque__ __ev_create_s16( int16_t a, int16_t b, int16_t c, int16_t d );
__ev64_opaque__ __ev_create_sfix32_fs( float a, float b );
__ev64_opaque__ __ev_create_ufix32_fs( float a, float b );
```

```
//maps to__ev_create_u32

__ev64_opaque__ __ev_create_ufix32_u32( uint32_t a, uint32_t b );


// maps to __ev_create_s32

__ev64_opaque__ __ev_create_sfix32_s32( int32_t a, int32_t b );
```

4.1.2 Convert intrinsics

These intrinsics convert a generic 64-bit opaque data type to a specific signed or unsigned integral form.

```
uint64_t __ev_convert_u64( __ev64_opaque__ a );
int64_t __ev_convert_s64( __ev64_opaque__ a );
```

4.1.3 Get intrinsics

These intrinsics allow the user to access data from within a specified location of the generic 64-bit opaque data type.

Get_upper/lower

These intrinsics specify whether the upper 32-bits or lower 32-bits of the 64-bit opaque data type are returned. Only signed/unsigned 32-bit integers or single-precision floats are returned.

```
uint32_t __ev_get_upper_u32( __ev64_opaque__ a );
uint32_t __ev_get_lower_u32( __ev64_opaque__ a );
int32_t __ev_get_upper_s32( __ev64_opaque__ a );
int32_t __ev_get_lower_s32( __ev64_opaque__ a );
float __ev_get_upper_fs( __ev64_opaque__ a );
float __ev_get_lower_fs( __ev64_opaque__ a );

// maps to __ev_get_upper_u32
uint32_t __ev_get_upper_ufix32_u32( __ev64_opaque__ a );


// maps to __ev_get_lower_u32
uint32_t __ev_get_lower_ufix32_u32( __ev64_opaque__ a );


// maps to __ev_get_upper_s32
int32_t __ev_get_upper_sfix32_s32( __ev64_opaque__ a );


// maps to __ev_get_lower_s32
int32_t __ev_get_lower_sfix32_s32( __ev64_opaque__ a );


// equivalent to __ev_get_sfix32_fs(a, 0);
float __ev_get_upper_sfix32_fs( __ev64_opaque__ a );


// equivalent to __ev_get_sfix32_fs(a, 1);
float __ev_get_lower_sfix32_fs( __ev64_opaque__ a );


// equivalent to __ev_get_ufix32_fs(a, 0);
float __ev_get_upper_ufix32_fs( __ev64_opaque__ a );
```

```
// equivalent to __ev_get_ufix32_fs(a, 1);
float __ev_get_lower_ufix32_fs( __ev64_opaque__ a );
```

Get explicit position

These intrinsics allow the user to specify the position (pos) in the 64-bit opaque data type where the data is accessed and returned. The position is 0 or 1 for words and either 0, 1, 2, or 3 for half-words.

```
uint32_t __ev_get_u32( __ev64_opaque__ a, uint32_t pos );
int32_t  __ev_get_s32( __ev64_opaque__ a, uint32_t pos );
float  __ev_get_fs( __ev64_opaque__ a, uint32_t pos );
uint16_t __ev_get_u16( __ev64_opaque__ a, uint32_t pos );
int16_t  __ev_get_s16( __ev64_opaque__ a, uint32_t pos );


// maps to __ev_get_u32
uint32_t __ev_get_ufix32_u32( __ev64_opaque__ a, uint32_t pos );

// maps to __ev_get_s32
int32_t  __ev_get_sfix32_s32( __ev64_opaque__ a, uint32_t pos );

float  __ev_get_ufix32_fs( __ev64_opaque__ a, uint32_t pos );
float  __ev_get_sfix32_fs( __ev64_opaque__ a, uint32_t pos );
```

4.1.4 Set intrinsics

These intrinsics provide the capability of setting values in a 64-bit opaque data type that the intrinsic or the user specifies.

Set_upper/lower

These intrinsics specify which word (either upper or lower 32-bits) of the 64-bit opaque data type is set to input value b.

```
__ev64_opaque__ __ev_set_upper_u32( __ev64_opaque__ a, uint32_t b );
__ev64_opaque__ __ev_set_lower_u32( __ev64_opaque__ a, uint32_t b );
__ev64_opaque__ __ev_set_upper_s32( __ev64_opaque__ a, int32_t b );
__ev64_opaque__ __ev_set_lower_s32( __ev64_opaque__ a, int32_t b );
__ev64_opaque__ __ev_set_upper_fs( __ev64_opaque__ a, float b );
__ev64_opaque__ __ev_set_lower_fs( __ev64_opaque__ a, float b );

// maps to __ev_set_upper_u32
__ev64_opaque__ __ev_set_upper_ufix32_u32( __ev64_opaque__ a, uint32_t b );


// maps to __ev_set_lower_u32
__ev64_opaque__ __ev_set_lower_ufix32_u32( __ev64_opaque__ a, uint32_t b );


// maps to __ev_set_upper_s32
__ev64_opaque__ __ev_set_upper_sfix32_s32( __ev64_opaque__ a, int32_t b );


// maps to __ev_set_lower_s32
__ev64_opaque__ __ev_set_lower_sfix32_s32( __ev64_opaque__ a, int32_t b );


// equivalent to __ev_set_sfix32_fs(a, b, 0);
__ev64_opaque__ __ev_set_upper_sfix32_fs( __ev64_opaque__ a, float b );
```

```
// equivalent to __ev_set_sfix32_fs(a, b, 1);
__ev64_opaque__ __ev_set_lower_sfix32_fs( __ev64_opaque__ a, float b );

// equivalent to __ev_set_ufix32_fs(a, b, 0);
__ev64_opaque__ __ev_set_upper_ufix32_fs( __ev64_opaque__ a, float b );

// equivalent to __ev_set_ufix32_fs(a, b, 1);
__ev64_opaque__ __ev_set_lower_ufix32_fs( __ev64_opaque__ a, float b );
```

Set accumulator

These intrinsics initialize the accumulator to the input value a.

```
__ev64_opaque__ __ev_set_acc_u64( uint64_t a );
__ev64_opaque__ __ev_set_acc_s64( int64_t a );
__ev64_opaque__ __ev_set_acc_vec64( __ev64_opaque__ a );
```

Set explicit position

These intrinsics set the 64-bit opaque input value a to the value in b based on the position given in pos. Unlike the intrinsics in 4.1.4.1, the positional value is specified by the user to be either 0 or 1 for words or 0, 1, 2, or 3 for half-words.

```
__ev64_opaque__ __ev_set_u32( __ev64_opaque__ a, uint32_t b, uint32_t pos );
__ev64_opaque__ __ev_set_s32( __ev64_opaque__ a, int32_t b, uint32_t pos );
__ev64_opaque__ __ev_set_fs( __ev64_opaque__ a, float b, uint32_t pos );
__ev64_opaque__ __ev_set_u16( __ev64_opaque__ a, uint16_t b, uint32_t pos );
__ev64_opaque__ __ev_set_s16( __ev64_opaque__ a, int16_t b, uint32_t pos );

// maps to __ev_set_u32
__ev64_opaque__ __ev_set_ufix32_u32( __ev64_opaque__ a, uint32_t b, uint32_t pos);

// maps to __ev_set_s32
__ev64_opaque__ __ev_set_sfix32_s32( __ev64_opaque__ a, int32_t b, uint32_t pos);

__ev64_opaque__ __ev_set_ufix32_fs( __ev64_opaque__ a, float b, uint32_t pos );
__ev64_opaque__ __ev_set_sfix32_fs( __ev64_opaque__ a, float b, uint32_t pos );
```

4.2 Signal processing engine (SPE) APU registers

The SPE includes the following two registers:

- The signal processing and embedded floating-point status and control register (SPEFSCR), described in *Chapter 4.2.1: Signal processing and embedded floating-point status and control register (SPEFSCR) on page 298*."
- A 64-bit accumulator, described in *Chapter 3.1.2: Accumulator (ACC) on page 21*."

4.2.1 Signal processing and embedded floating-point status and control register (SPEFSCR)

The SPEFSCR, which is shown in *Figure 242*, is used for status and control of SPE instructions.

Figure 242. Signal processing and embedded floating-point status and control register (SPEFSCR)

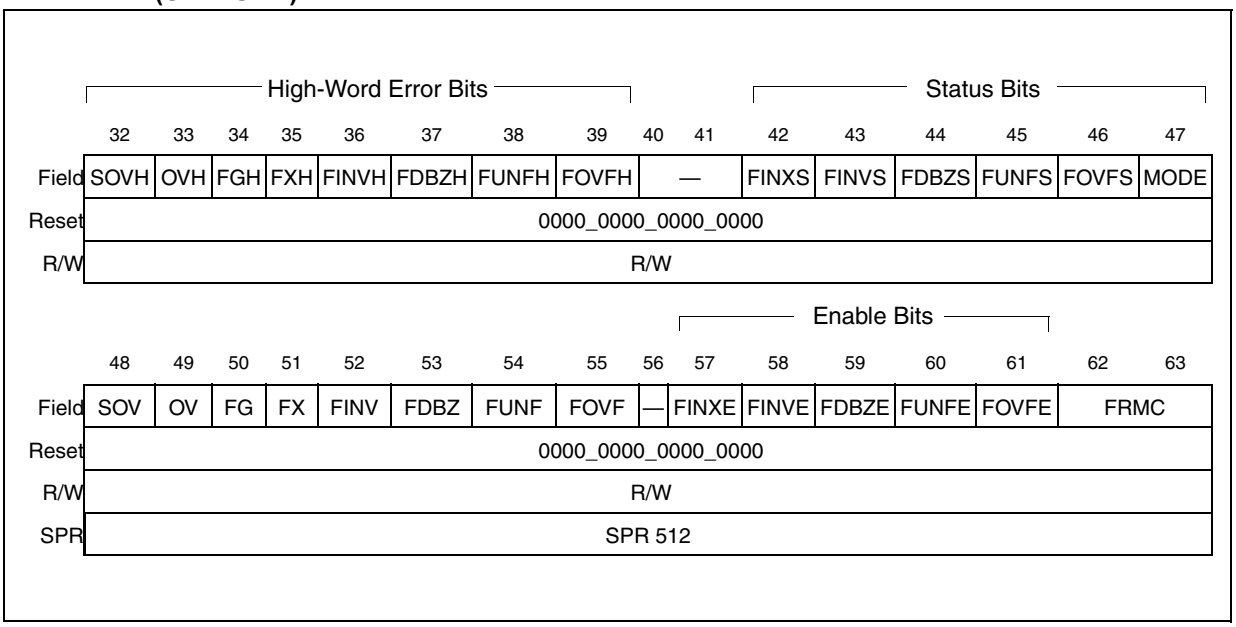


Table 247. SPEFSCR field descriptions

Bits	Name	Function
32	SOVH	Summary integer overflow high. Set whenever an instruction (except **mtspr**) sets OVH. SOVH remains set until it is cleared by an **mtspr[SPEFSCR]**.
33	OVH	Integer overflow high. An overflow occurred in the upper half of the register while executing a SPE integer instruction.
34	FGH	Embedded floating-point guard bit high. Floating-point guard bit from the upper half. The value is undefined if the processor takes a floating-point exception due to input error, floating-point overflow, or floating-point underflow.
35	FXH	Embedded floating-point sticky bit high. Floating bit from the upper half. The value is undefined if the processor takes a floating-point exception due to input error, floating-point overflow, or floating-point underflow.
36	FINVH	Embedded floating-point invalid operation error high. Set when an input value on the high side is a NaN, Inf, or Denorm. Also set on a divide if both the dividend and divisor are zero.
37	FDBZH	Embedded floating-point divide by zero error high. Set if the dividend is non-zero and the divisor is zero.
38	FUNFH	Embedded floating-point underflow error high
39	FOVFH	Embedded floating-point overflow error high
40–41	—	Reserved, and should be cleared
42	FINXS	Embedded floating-point inexact sticky. FINXS = FINXS | FGH | FXH | FG | FX.
43	FINVS	Embedded floating-point invalid operation sticky. Location for software to use when implementing true IEEE floating point.
44	FDBZS	Embedded floating-point divide by zero sticky. FDBZS = FDBZS | FDBZH | FDBZ.
45	FUNFS	Embedded floating-point underflow sticky. Storage location for software to use when implementing true IEEE floating point.

Table 247. SPEFSCR field descriptions (continued)

Bits	Name	Function
46	FOVFS	Embedded floating-point overflow sticky. Storage location for software to use when implementing true IEEE floating point.
47	MODE	Embedded floating-point mode (read-only on e500)
48	SOV	Integer summary overflow. Set whenever an SPE instruction (except **mtspr**) sets OV. SOV remains set until it is cleared by **mtspr[SPEFSCR]**.
49	OV	Integer overflow. An overflow occurred in the lower half of the register while a SPE integer instruction was executed.
50	FG	Embedded floating-point guard bit. Floating-point guard bit from the lower half. The value is undefined if the processor takes a floating-point exception due to input error, floating-point overflow, or floating-point underflow.
51	FX	Embedded floating-point sticky bit. Floating bit from the lower half. The value is undefined if the processor takes a floating-point exception due to input error, floating-point overflow, or floating-point underflow.
52	FINV	Embedded floating-point invalid operation error. Set when an input value on the high side is a NaN, Inf, or Denorm. Also set on a divide if both the dividend and divisor are zero.
53	FDBZ	Embedded floating-point divide by zero error. Set of the dividend is non-zero and the divisor is zero.
54	FUNF	Embedded floating-point underflow error
55	FOVF	Embedded floating-point overflow error
56	—	Reserved, and should be cleared
57	FINXE	Embedded floating-point inexact enable
58	FINVE	Embedded floating-point invalid operation/input error exception enable 0: Exception disabled 1: Exception enabled If the exception is enabled, a floating-point data exception is taken if FINV or FINVH is set by a floating-point instruction.
59	FDBZE	Embedded floating-point divide-by-zero exception enable 0: Exception disabled 1: Exception enabled If the exception is enabled, a floating-point data exception is taken if FDBZ or FDBZH is set by a floating-point instruction.
60	FUNFE	Embedded floating-point underflow exception enable 0: Exception disabled 1: Exception enabled If the exception is enabled, a floating-point data exception is taken if FUNF or FUNFH is set by a floating-point instruction.

Table 247. SPEFSCR field descriptions (continued)

Bits	Name	Function
61	FOVFE	Embedded floating-point overflow exception enable 0: Exception disabled 1: Exception enabled If the exception is enabled, a floating-point data exception is taken if FOVF or FOVFH is set by a floating-point instruction.
62–63	FRMC	Embedded floating-point rounding mode control 00: Round to nearest 01: Round toward zero 10: Round toward +infinity 11: Round toward -infinity

4.2.2 SPEFSCR intrinsics

The following sections discuss SPEFSCR low-level accessors and SPEFSCR clear and set functions.

SPEFSCR low-level accessors

These intrinsics allow the user to access specific bits in the status and control registers.

```
uint32_t __ev_get_spefscr_sovh( );
uint32_t __ev_get_spefscr_ovh( );
uint32_t __ev_get_spefscr_fgh( );
uint32_t __ev_get_spefscr_fxh( );
uint32_t __ev_get_spefscr_finvh( );
uint32_t __ev_get_spefscr_fdbzh( );
uint32_t __ev_get_spefscr_funfh( );
uint32_t __ev_get_spefscr_fovfh( );

uint32_t __ev_get_spefscr_finxs( );
uint32_t __ev_get_spefscr_finvs( );
uint32_t __ev_get_spefscr_fdbzs( );
uint32_t __ev_get_spefscr_funfs( );
uint32_t __ev_get_spefscr_fovfs( );

uint32_t __ev_get_spefscr_mode( );

uint32_t __ev_get_spefscr_sov( );
uint32_t __ev_get_spefscr_ov( );
uint32_t __ev_get_spefscr_fg( );
uint32_t __ev_get_spefscr_fx( );
uint32_t __ev_get_spefscr_finv( );
uint32_t __ev_get_spefscr_fdbz( );
uint32_t __ev_get_spefscr_funf( );
uint32_t __ev_get_spefscr_fovf( );

uint32_t __ev_get_spefscr_finxe( );
uint32_t __ev_get_spefscr_finve( );
uint32_t __ev_get_spefscr_fdbze( );
uint32_t __ev_get_spefscr_funfe( );
uint32_t __ev_get_spefscr_fovfe( );

uint32_t __ev_get_spefscr_frmc( );

SPEFSCR Clear and Set Functions
```

Note: These intrinsics allow the user to clear and set specific bits in the status and control register. The user can set only the rounding mode bits.

```
void __ev_clr_spefscr_sovh( );
void __ev_clr_spefscr_sov( );

void __ev_clr_spefscr_finxs( );
void __ev_clr_spefscr_finvs( );
void __ev_clr_spefscr_fdbzs( );
void __ev_clr_spefscr_funfs( );
void __ev_clr_spefscr_fovfs( );

void __ev_set_spefscr_frmc( uint32_t rnd );
                                  // rnd = 0 (nearest), rnd = 1 (zero),
                                  // rnd = 2 (+inf), rnd = 3 (-inf)
```

4.3 Application binary interface (ABI) extensions

The following sections discuss ABI extensions.

4.3.1 malloc(), realloc(), calloc(), and new

The malloc(), realloc(), and calloc() functions are required to return a pointer with the proper alignment for the object in question. Therefore, to conform to the ABI, these functions must return pointers to memory locations that are at least 8-byte aligned. In the case of the C++ operator new, the implementation of new is required to use the appropriate set of functions based on the alignment requirements of the type.

4.3.2 printf example

The programming model specifies several new conversion format tokens. The programming model expects a combination of existing format tokens, new format tokens, and __ev_get_* intrinsics. *Table 248* lists new tokens specified to handle fixed-point data types.

Table 248. New tokens for fixed-point data types

Token	Data representation
%hr	Signed 16-bit fixed point
%r	Signed 32-bit fixed point
%lr	Signed 64-bit fixed point
%hR	Unsigned 16-bit fixed point
%R	Unsigned 32-bit fixed point
%lR	Unsigned 64-bit fixed point

Example:

```
__ev64_opaque__ a ;

a = __ev_create_s32 ( 2, -3 );

printf ( " %d %d \n", __ev_get_upper_s32(a), __ev_get_lower_s32(a) );

// output:
// 2 -3
```

The default precision for the new tokens is 6 digits. The tokens should be treated like the %f token with respect to floating-point values. The same field width and precision options should be respected for the new tokens, as the following example shows:

```
printf ("%lr", 0x4000);==> "0.500000"
printf ("%r", 0x40000000); ==> "0.500000"
printf ("%hr", 0x4000000000000000ull);==> "0.500000"
printf ("%09.5r",0x40000000);==> "000.50000"
printf ("%09.5f",0.5);==> "000.50000"
```

4.3.3 Additional library routines

The functions atosfix16, atosfix32, atosfix64, atoufix16, atoufix32, and atoufix64 need not affect the value of the integer expression errno on an error. If the value of the result cannot be represented, the behavior is undefined.

```
#include <spe.h>
int16_t atosfix16(const char *str);
int32_t atosfix32(const char *str);
int64_t atosfix64(const char *str);

uint16_t atoufix16(const char *str);
uint32_t atoufix32(const char *str);
uint64_t atoufix64(const char *str);
```

The atosfix16, atosfix32, atosfix64, atoufix16, atoufix32, atoufix64 functions convert the initial portion of the string to which str points to the following numbers:

- 16-bit signed fixed-point number
- 32-bit signed fixed-point number
- 64-bit signed fixed-point number
- 16-bit unsigned fixed-point number
- 32-bit unsigned fixed-point number
- 64-bit unsigned fixed-point number

These numbers are represented as int16_t, int32_t, int64_t, uint16_t, uint32_t, and uint64_t, respectively.

Except for the behavior on error, they are equivalent to the following:

```
atosfix16: strtosfix16(str, (char **)NULL)
atosfix32: strtosfix32(str, (char **)NULL)
atosfix64: strtosfix64(str, (char **)NULL)
atoufix16: strtoufix16(str, (char **)NULL)
atoufix32: strtoufix32(str, (char **)NULL)
atoufix64: strtoufix64(str, (char **)NULL)
```

```
#include <spe.h>
int16_t strtosfix16(const char *str, char **endptr);
int32_t strtosfix32(const char *str, char **endptr);
int64_t strtosfix64(const char *str, char **endptr);

uint16_t strtoufix16(const char *str, char **endptr);
uint32_t strtoufix32(const char *str, char **endptr);
uint64_t strtoufix64(const char *str, char **endptr);
```

The strtosfix16, strtosfix32, strtosfix64, strtoufix16, strtoufix32, strtoufix64 functions convert the initial portion of the string to which str points to the following numbers:

- 16-bit signed fixed-point number
- 32-bit signed fixed-point number
- 64-bit signed fixed-point number
- 16-bit unsigned fixed-point number
- 32-bit unsigned fixed-point number
- 64-bit unsigned fixed-point number

These numbers are represented as int16_t, int32_t, int64_t, uint16_t, uint32_t, and uint64_t, respectively.

The functions support the same string representations for fixed-point numbers that the strtod, strtof, strtold functions support, with the exclusion of NAN and INFINITY support.

For the signed functions, if the input value is greater than or equal to 1.0, positive saturation should occur and errno should be set to ERANGE. If the input value is less than -1.0, negative saturation should occur, and errno should be set to ERANGE.

For the unsigned functions, if the input value is greater than or equal to 1.0, saturation should occur to the upper bound, and errno should be set to ERANGE. If the input value is less than 0.0, saturation should occur to the lower bound and errno should be set to ERANGE.

5 Programming interface examples

5.1 Data type initialization

The following examples show valid and invalid initializations of the SPE data types.

5.1.1 __ev64_opaque__ initialization

The following examples show valid and invalid initializations of __ev64_opaque__:

- Example 1 (Invalid)

```
__ev64_opaque__ x1 = { 0, 1 };
```

  This example is invalid because it lacks qualification for interpreting the array initialization. The compiler is unable to interpret whether the array consists of two unsigned integers, two signed integers, four unsigned integers, four signed integers, or two floats.

- Example 2 (Invalid)

```
__ev64_opaque__ x2 = (__ev64_opaque__) { 0, 1 };
```

  This example is invalid because the qualification provides no additional information for interpreting the array initialization.

- Example 3 (Valid)

```
__ev64_opaque__ x3 = (__ev64_u32__) { 0, 1 };
```

  This example is valid because the array initialization is qualified so that it provides the compiler with a unique interpretation. The array initialization is interpreted as an __ev64_u32__ with an implicit cast from the __ev64_u32__ to __ev64_opaque__.

- Example 4 (Valid)

```
__ev64_opaque__ x4 = (__ev64_opaque__)(__ev64_u32__) { 0, 1 };
```

  Although this example is the same as Example 3, it includes an explicit cast, rather than depending on the implicit casting to __ev64_opaque__ on assignment.

- Example 5 (Valid)

```
__ev64_opaque__ x5 = (__ev64_u16__) (__ev64_opaque__)
(__ev64_u32__) { 0, 1 };
```

  This example shows a series of casts; at the end, the result in x5 is no different from what it would be in Example 3. The example depends on the implicit cast from __ev64_u16__ to __ev64_opaque__.

- Example 6 (Valid)

```
__ev64_opaque__ x6 = (__ev64_opaque__) (__ev64_u16__)
(__ev64_u32__) { 0, 1 };
```

  This example shows a series of casts; at the end, the result in x6 is no different from what it would be in Example 3. The example explicitly casts to __ev64_opaque__ rather than depending on the implicit cast.

- Example 7 (Valid)

```
__ev64_opaque__ x7 = (__ev64_u16__) (__ev64_u32__) { 0, 1 };
```

  This example shows a series of casts; at the end, the result in x6 is no different from what it would be in Example 3. The example depends on the implicit cast from __ev64_u16__ to __ev64_opaque__.

- Example 8 (Valid)

```
__ev64_opaque__ x8 = (__ev64_u16__) { 0, 1, 2, 3 };
```

This example is similar to Example 3. It shows that any SPE data types except __ev64_opaque__ can be used to qualify the array initialization.

5.1.2 Array initialization of SPE data types

The following examples show array initialization of SPE data types:

- Example 1 shows how to initialize an array of four __ev64_u32__.

```
__ev64_u32__ x1[4] = {
   { 0, 1 },
   { 2, 3 },
   { 4, 5 },
   { 6, 7 }
};
```

- Example 2 shows how to initialize an array of four __ev64_u16__.

```
__ev64_u16 x2[4] = {
   { 0, 1, 2, 3 },
   { 4, 5, 6, 7 },
   { 8, 9, 10, 11 },
   { 12, 13, 14, 15 },
};
```

- Example 3 shows how to initialize an array of four __ev64_fs__.

```
__ev64_fs__ x3[4] = {
   { 1.1f, 2.2f },
   { -3.3f, 4.4f },
   { 5.5f, 6.6f },
   { 7.7f, -8.8f }
};
```

- Example 4 shows explicit casting, and is the same as Example 1:

```
__ev64_u32__ x4[4] = {
  (__ev64_u32__){0, 1},
  (__ev64_u32__){2, 3},
  (__ev64_u32__){4, 5},
  (__ev64_u32__){6, 7}
};
```

- Example 5 shows mixed explicit casting. x5[1] is equal to (__ev64_u32__){131075, 262149}.

```
__ev64_u32__ x5[4] = {
  (__ev64_u32__){0, 1},
  (__ev64_u16__){2, 3, 4, 5},
  (__ev64_u32__){6, 7},
  (__ev64_u32__){8, 9}
};
```

5.2 Fixed-point accessors

The following sections discuss fixed-point accessors.

5.2.1 __ev_create_sfix32_fs

The following examples show use of __ev_create_sfix32_fs:

- Example 1

```
__ev64_s32__ x1 = __ev_create_sfix32_fs (0.5, -0.125);
// x1 = {0x40000000, 0xF0000000}
```

The floating-point numbers 0.5 and -0.125 are converted to their fixed-point representations and stored in x1.

- Example 2

```
__ev64_s32__ x2 = __ev_create_sfix32_fs (-1.1, 1.0);
// x2 = {0x80000000, 0x7fffffff}
```

The floating-point numbers are –1.1 and 1.0. Both values are outside of the range that signed fixed-point [–1, 1) supports. Therefore, the results of the conversion are saturated to the most negative number, 0x80000000, and the most positive number, 0x7FFFFFFF.

5.2.2 __ev_create_ufix32_fs

The following examples show use of __ev_create_ufix32_fs:

- Example 1

```
__ev64_u32__ x1 = __ev_create_ufix32_fs(0.5, 0.125);
// x1 = {0x80000000, 0x20000000}
```

The floating-point numbers 0.5 and 0.125 are converted to their unsigned fixed-point representations and stored in x1.

- Example 2

```
__ev64_u32__ x2 = __ev_create_ufix32_fs(-1.1, 1.0);
// x2 = {0x00000000, 0xFFFFFFFF}
```

Both floating-point values, –1.1 and 1.0, are outside of the range that unsigned fixed-point [0, 1) supports. Therefore, the results of the conversion are saturated to the lower bound, 0x00000000, and the upper bound, 0xFFFFFFFF.

5.2.3 __ev_set_ufix32_fs

The following examples show use of __ev_set_ufix32_fs:

- Example 1

```
__ev64_u32__ x1a = { 0x00000000 0xffffffff };
__ev64_u32__ x1b = __ev_set_ufix32_fs (x1a, 0.5, 0);
// x1b = {0x80000000, 0xffffffff}
```

This example shows modification of an element in an SPE variable. The intrinsics work like the create routine in that the floating-point number 0.5 is converted to its unsigned fixed-point representation and placed into element 0.

- Example 2

```
__ev64_u32__ x2a = { 0x00000000 0xffffffff };
__ev64_u32__ x2b = __ev_set_ufix32_fs (x2a, 1.5, 0);
```

```
// x2b = {0xffffffff, 0xffffffff}
```

This example shows modification of an element in an SPE variable. The intrinsics work like the create routine in that the floating-point number 1.5 is saturated to the upper bound for unsigned fixed-point representation and placed into element 0.

5.2.4 __ev_set_sfix32_fs

The following examples show use of __ev_set_sfix32_fs:

- Example 1

```
__ev64_u32__ x1a = { 0x00000000 0xffffffff };
__ev64_u32__ x1b = __ev_set_sfix32_fs (x1a, 0.5, 0);
// x1b = {0x40000000, 0xffffffff}
```

This example shows modification of an element in an SPE variable. The intrinsics work like the create routine in that the floating-point number 0.5 is converted to its signed fixed-point representation and placed into element 0.

- Example 2

```
__ev64_s32__ x2a = { 0x00000000 0xffffffff };
__ev64_s32__ x2b = __ev_set_sfix32_fs (x2a, 1.5, 0);
// x2b = {0x7fffffff, 0xffffffff}
```

This example shows modification of an element in an SPE variable. The intrinsics work like the create routine in that the floating-point number 1.5 is saturated to the upper bound for signed fixed-point representation and placed into element 0.

5.2.5 __ev_get_ufix32_fs

This example shows extraction of a floating-point number from an SPE variable interpreted as an unsigned fixed-point number. The intrinsic extracts element 1 of the variable and converts it from an unsigned fixed-point number to the closest floating-point representation.

```
__ev64_u32__ x1 = { 0x80000000, 0xffffffff };
float f1 = __ev_get_ufix32_fs (x1, 1);
// f1 = 1.0
```

5.2.6 __ev_get_sfix32_fs

This example shows extraction of a floating-point number from an SPE variable interpreted as a signed fixed-point number. The intrinsic extracts element 0 of the variable and converts it from a signed fixed-point number to the closest floating-point value.

```
__ev64_s32__ x1 = { 0xf0000000, 0xffffffff };
float f1 = __ev_get_ufix32_fs (x1, 0);
// f1 = -0.125
```

5.3 Loads

These examples apply to load and store intrinsics. All of the examples reference the same 'ev_table':

```
__ev64_u32__ ev_table[] = {
  (__ev64_u32__){0x01020304, 0x05060708},
  (__ev64_u32__){0x090a0b0c, 0x0d0e0f10},
  (__ev64_u32__){0x11121314, 0x15161718},
  (__ev64_u32__){0x191a1b1c, 0x1d1e1f20},
```

```
  (__ev64_u32__){0x797a7b7c, 0x7d7e7f80},
  (__ev64_u32__){0x81828384, 0x85868788},
  (__ev64_u32__){0x898a8b8c, 0x8d8e8f90},
  (__ev64_u32__){0x91929394, 0x95969798}
};
```

5.3.1 __ev_lddx

This example shows indexing of double-word load. The base pointer is set to the address of ev_table. The intrinsic offsets the base pointer by 2 double-words (16 bytes). This load is equivalent to ev_table[2].

```
__ev64_u32__ x1 = __ev_lddx((__ev64_opaque__ *)(&ev_table[0]), 16);
// x1 = {0x11121314, 0x15161718};
```

5.3.2 __ev_ldd

This example shows an immediate double-word load. The base pointer is set to the address of ev_table. The intrinsic offsets the base pointer by 2 double-words. This load is equivalent to ev_table[2]. The offset in the immediate pointer is scaled by the double-word load size.

```
__ev64_u32__ x1 = __ev_ldd((__ev64_opaque__ *)(&ev_table[0]), 2);
// x1 = {0x11121314, 0x15161718};
```

5.3.3 __ev_lhhesplatx

This example shows an index half-word even splat load. The base pointer is set to the address of ev_table. The intrinsic offsets the base pointer by 4 bytes.

```
__ev64_u32__ x1 = __ev_lhhesplatx((__ev64_opaque__
*)(&ev_table[0]), 4);
// x1 = {0x05060000, 0x05060000}
```

5.3.4 __ev_lhhesplat

This example shows an immediate half-word even splat load. The base pointer is set to the address of ev_table. The intrinsic offsets the base pointer by 4 half-words (8 bytes). Note that the load size, a half-word in this case, scales the offset in the immediate pointer.

```
__ev64_u32__ x1 = __ev_lhhesplat((__ev64_opaque__ *)(&ev_table[0]), 4);
// x1 = {0x090a0000, 0x090a0000}
```

6 Glossary of terms and abbreviations

The glossary contains an alphabetical list of terms, phrases, and abbreviations used in this book. Some of the terms and definitions included in the glossary are reprinted from IEEE Std. 754-1985, IEEE Standard for binary floating-point arithmetic, copyright ©1985 by the Institute of Electrical and Electronics Engineers, Inc. with the permission of the IEEE.

Note that some terms are defined in the context of their usage in this manual.

A

Application binary interface (ABI). A standardized interface that defines calling conventions and stack usage between applications and the operating system.

Architecture. A detailed specification of requirements for a processor or computer system. It does not specify details for implementing the processor or computer system; instead it provides a template for a family of compatible implementations.

B

Biased exponent. An exponent whose range of values is shifted by a constant (bias). Typically a bias is provided to allow a range of positive values to express a range that includes both positive and negative values.

Big-endian. A byte-ordering method in memory where the address n of a word corresponds to the most-significant byte. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 as the most-significant byte. See Little-endian.

C

Cast. A cast expression consists of a left parenthesis, a type name, a right parenthesis, and an operand expression. The cast causes the operand value to be converted to the type name within the parentheses.

D

Denormalized number. A non zero floating-point number whose exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero.

E

Effective address (EA). The 32- or 64-bit address specified for a load, store, or an instruction fetch. This address is then submitted to the MMU for translation to either a physical memory address or an I/O address.

Exponent. In the binary representation of a floating-point number, the exponent is the component that normally signifies the integer power to which the value two is raised in determining the value of the represented number. See also Biased exponent.

F

Fixed-point. (see Fractional)

Fractional. SPE supports 16- and 32-bit signed fractional two's complement data formats. For these two N-bit fractional data types, data is represented using the 1. [N-1] bit format. The MSB is the sign bit (-2^0) and the remaining N-1 bits are fractional bits (2^-1 2^-2 ... 2^-(N-1)).

G

General-purpose register (GPR). Any of the 32 registers in the general-purpose register file. These registers provide the source operands and destination results for all integer data manipulation instructions. Integer load instructions move data from memory to GPRs and store instructions move data from GPRs to memory.

I

IEEE 754. A standard written by the Institute of Electrical and Electronics Engineers that defines operations and representations of binary floating-point arithmetic.

Inexact. Loss of accuracy in an arithmetic operation when the rounded result differs from the infinitely precise value with unbounded range.

L

LSB (Least-significant bit). The bit of least value in an address, register, data element, or instruction encoding.

Little-endian. A byte-ordering method in memory where the address n of a word corresponds to the least-significant byte. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 as the most-significant byte. See Big-endian.

M

Mnemonic. The abbreviated name of an instruction used for coding.

Modulo. A value v that lies outside the range of numbers that an n-bit wide destination type can represent is replaced by the low-order n bits of the two's complement representation of v.

MSB (Most-significant bit). The highest-order bit in an address, registers, data element, or instruction encoding.

N

NaN. An abbreviation for 'Not a Number'; a symbolic entity encoded in floating-point format. The two types of NaNs are signaling NaNs (SNaNs) and quiet NaNs (QNaNs).

Normalization. A process by which a floating-point value is manipulated such that it can be represented in the format for the appropriate precision (single- or double-precision). For a floating-point value to be representable in the single- or double-precision format, the leading implied bit must be a 1.

O

Overflow. An error condition that occurs during arithmetic operations when the result cannot be stored accurately in the destination register(s). For example, if two 32-bit numbers are multiplied, the result may not be representable in 32 bits.

R

Reserved field. In a register, a reserved field is one that is not assigned a function. A reserved field may be a single bit. The handling of reserved bits is implementation-dependent. Software can write any value to such a bit. A subsequent reading of the bit returns 0 if the value last written to the bit was 0 and returns an undefined value (0 or 1) otherwise.

S

Saturate. A value v that lies outside the range of numbers representable by a destination type is replaced by the representable number closest to v.

Significand. The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

SIMD (Single-instruction, multiple-data). An instruction set architecture that performs operations on multiple, parallel values within a single operand.

Splat. To replicate a value in multiple elements of an SIMD target operand.

Sticky bit. A bit that when set must be cleared explicitly.

Supervisor mode. The privileged operation state of a processor. In supervisor mode, software, typically the operating system, can access all control registers and the supervisor memory space, among other privileged operations.

U

Underflow. An error condition that occurs during arithmetic operations when the result cannot be represented accurately in the destination register. For example, underflow can happen if two floating-point fractions are multiplied and the result requires a smaller exponent and/or mantissa than the single-precision format can provide. In other words, the result is too small for accurate representation.

User mode. The unprivileged operating state of a processor used typically by application software. In user mode, software can only access certain control registers and can access only user memory space. No privileged operations can be performed. Also known as problem state.

V

Vector literal. A constant expression with a value that is taken as a vector type.

W

Word. A 32-bit data element.

7 Revision history

Table 249. Document revision history

Date	Revision	Changes
3-Mar-2008	1	Initial release.
22-Oct-2012	2	Added new active RPNs in “Specific properties” of the document. Document reformatted no content change.
17-Sep-2013	3	Updated Disclaimer