

RISC-V "P" Extension Proposal

Version 0.5.4-draft-20200302

Table of Contents

Revision History	1
1. Introduction	2
2. DSP ISA Extension Instruction Summary	3
2.1. Shorthand Definitions	3
2.2. SIMD Data Processing Instructions	4
2.2.1. 16-bit Addition & Subtraction Instructions	4
2.2.2. 8-bit Addition & Subtraction Instructions	8
2.2.3. 16-bit Shift Instructions	10
2.2.4. 8-bit Shift Instructions	12
2.2.5. 16-bit Compare Instructions	14
2.2.6. 8-bit Compare Instructions	15
2.2.7. 16-bit Multiply Instructions	16
2.2.8. 8-bit Multiply Instructions	18
2.2.9. 16-bit Misc Instructions	20
2.2.10. 8-bit Misc Instructions	21
2.2.11. 8-bit Unpacking Instructions	22
2.3. Partial-SIMD Data Processing Instructions	24
2.3.1. 16-bit Packing Instructions	24
2.3.2. Most Significant Word “32x32” Multiply & Add Instructions	25
2.3.3. Most Significant Word “32x16” Multiply & Add Instructions	27
2.3.4. Signed 16-bit Multiply with 32-bit Add/Subtract Instructions	31
2.3.5. Signed 16-bit Multiply with 64-bit Add/Subtract Instructions	34
2.3.6. Miscellaneous Instructions	35
2.3.7. 8-bit Multiply with 32-bit Add Instructions	36
2.4. 64-bit Profile Instructions	37
2.4.1. 64-bit Addition & Subtraction Instructions	37
2.4.2. 32-bit Multiply with 64-bit Add/Subtract Instructions	39
2.4.3. Signed 16-bit Multiply with 64-bit Add/Subtract Instructions	40
2.5. Non-SIMD Instructions	42
2.5.1. Q15 saturation instructions	42
2.5.2. Q31 saturation Instructions	43
2.5.3. 32-bit Computation Instructions	46
2.5.4. Overflow status manipulation instructions	48
2.5.5. Miscellaneous Instructions	49
2.6. RV64 Only Instructions	51
3. P Extension Subsets	65
3.1. Zp64 (for RV32 Only)	65
4. Detailed Instruction Descriptions (both RV32 & RV64)	67

4.1. ADD8 (SIMD 8-bit Addition)	68
4.2. ADD16 (SIMD 16-bit Addition)	69
4.3. ADD64 (64-bit Addition)	70
4.4. AVE (Average with Rounding)	72
4.5. BITREV (Bit Reverse)	73
4.6. BITREVI (Bit Reverse Immediate)	74
4.7. BPICK (Bit-wise Pick)	75
4.8. CLROV (Clear OV flag)	76
4.9. CLRS8 (SIMD 8-bit Count Leading Redundant Sign)	77
4.10. CLRS16 (SIMD 16-bit Count Leading Redundant Sign)	79
4.11. CLRS32 (SIMD 32-bit Count Leading Redundant Sign)	81
4.12. CLO8 (SIMD 8-bit Count Leading One)	83
4.13. CLO16 (SIMD 16-bit Count Leading One)	85
4.14. CLO32 (SIMD 32-bit Count Leading One)	87
4.15. CLZ8 (SIMD 8-bit Count Leading Zero)	89
4.16. CLZ16 (SIMD 16-bit Count Leading Zero)	91
4.17. CLZ32 (SIMD 32-bit Count Leading Zero)	93
4.18. CMPEQ8 (SIMD 8-bit Integer Compare Equal)	95
4.19. CMPEQ16 (SIMD 16-bit Integer Compare Equal)	96
4.20. CRAS16 (SIMD 16-bit Cross Addition & Subtraction)	97
4.21. CRSA16 (SIMD 16-bit Cross Subtraction & Addition)	99
4.22. INSB (Insert Byte)	101
4.23. KABS8 (SIMD 8-bit Saturating Absolute)	102
4.24. KABS16 (SIMD 16-bit Saturating Absolute)	104
4.25. KABSW (Scalar 32-bit Absolute Value with Saturation)	106
4.26. KADD8 (SIMD 8-bit Signed Saturating Addition)	107
4.27. KADD16 (SIMD 16-bit Signed Saturating Addition)	109
4.28. KADD64 (64-bit Signed Saturating Addition)	111
4.29. KADDH (Signed Addition with Q15 Saturation)	113
4.30. KADDW (Signed Addition with Q31 Saturation)	114
4.31. KCRAS16 (SIMD 16-bit Signed Saturating Cross Addition & Subtraction)	115
4.32. KCRSA16 (SIMD 16-bit Signed Saturating Cross Subtraction & Addition)	117
4.33. KDMBB, KDMBT, KDMTT	119
4.33.1. KDMBB (Signed Saturating Double Multiply B16 x B16)	119
4.33.2. KDMBT (Signed Saturating Double Multiply B16 x T16)	119
4.33.3. KDMTT (Signed Saturating Double Multiply T16 x T16)	119
4.34. KDMABB, KDMABT, KDMATT	122
4.34.1. KDMABB (Signed Saturating Double Multiply Addition B16 x B16)	122
4.34.2. KDMABT (Signed Saturating Double Multiply Addition B16 x T16)	122
4.34.3. KDMATT (Signed Saturating Double Multiply Addition T16 x T16)	122
4.35. KHM8, KHMX8	125

4.35.1. KHM8 (SIMD Signed Saturating Q7 Multiply)	125
4.35.2. KHXM8 (SIMD Signed Saturating Crossed Q7 Multiply)	125
4.36. KHM16, KHXM16	127
4.36.1. KHM16 (SIMD Signed Saturating Q15 Multiply)	127
4.36.2. KHXM16 (SIMD Signed Saturating Crossed Q15 Multiply)	127
4.37. KHMBB, KHMBT, KHM TT	129
4.37.1. KHMBB (Signed Saturating Half Multiply B16 x B16)	129
4.37.2. KHMBT (Signed Saturating Half Multiply B16 x T16)	129
4.37.3. KHM TT (Signed Saturating Half Multiply T16 x T16)	129
4.38. KMABB, KMABT, KMATT	132
4.38.1. KMABB (SIMD Saturating Signed Multiply Bottom Halfs & Add)	132
4.38.2. KMABT (SIMD Saturating Signed Multiply Bottom & Top Halfs & Add)	132
4.38.3. KMATT (SIMD Saturating Signed Multiply Top Halfs & Add)	132
4.39. KMADA, KMAXDA	135
4.39.1. KMADA (SIMD Saturating Signed Multiply Two Halfs and Two Adds)	135
4.39.2. KMAXDA (SIMD Saturating Signed Crossed Multiply Two Halfs and Two Adds)	135
4.40. KMADS, KMADRS, KMAXDS	138
4.40.1. KMADS (SIMD Saturating Signed Multiply Two Halfs & Subtract & Add)	138
4.40.2. KMADRS (SIMD Saturating Signed Multiply Two Halfs & Reverse Subtract & Add)	138
4.40.3. KMAXDS (SIMD Saturating Signed Crossed Multiply Two Halfs & Subtract & Add)	138
4.41. KMAR64 (Signed Multiply and Saturating Add to 64-Bit Data)	141
4.42. KMDA, KMXDA	143
4.42.1. KMDA (SIMD Signed Multiply Two Halfs and Add)	143
4.42.2. KMXDA (SIMD Signed Crossed Multiply Two Halfs and Add)	143
4.43. KMMAC, KMMAC.u	145
4.43.1. KMMAC (SIMD Saturating MSW Signed Multiply Word and Add)	145
4.43.2. KMMAC.u (SIMD Saturating MSW Signed Multiply Word and Add with Rounding)	145
4.44. KMMAWB, KMMAWB.u	147
4.44.1. KMMAWB (SIMD Saturating MSW Signed Multiply Word and Bottom Half and Add)	147
4.44.2. KMMAWB.u (SIMD Saturating MSW Signed Multiply Word and Bottom Half and Add with Rounding)	147
4.45. KMMAWB2, KMMAWB2.u	149
4.45.1. KMMAWB2 (SIMD Saturating MSW Signed Multiply Word and Bottom Half & 2 and Add)	149
4.45.2. KMMAWB2.u (SIMD Saturating MSW Signed Multiply Word and Bottom Half & 2 and Add with Rounding)	149
4.46. KMMAWT, KMMAWT.u	152
4.46.1. KMMAWT (SIMD Saturating MSW Signed Multiply Word and Top Half and Add)	152
4.46.2. KMMAWT.u (SIMD Saturating MSW Signed Multiply Word and Top Half and Add with Rounding)	152
4.47. KMMAWT2, KMMAWT2.u	154

4.47.1. KMMAWT2 (SIMD Saturating MSW Signed Multiply Word and Top Half & 2 and Add)	154
4.47.2. KMMAWT2.u (SIMD Saturating MSW Signed Multiply Word and Top Half & 2 and Add with Rounding)	154
4.48. KMMSB, KMMSB.u	157
4.48.1. KMMSB (SIMD Saturating MSW Signed Multiply Word and Subtract)	157
4.48.2. KMMSB.u (SIMD Saturating MSW Signed Multiply Word and Subtraction with Rounding)	157
4.49. KMMWB2, KMMWB2.u	159
4.49.1. KMMWB2 (SIMD Saturating MSW Signed Multiply Word and Bottom Half & 2)	159
4.49.2. KMMWB2.u (SIMD Saturating MSW Signed Multiply Word and Bottom Half & 2 with Rounding)	159
4.50. KMMWT2, KMMWT2.u	161
4.50.1. KMMWT2 (SIMD Saturating MSW Signed Multiply Word and Top Half & 2)	161
4.50.2. KMMWT2.u (SIMD Saturating MSW Signed Multiply Word and Top Half & 2 with Rounding)	161
4.51. KMSDA, KMSXDA	163
4.51.1. KMSDA (SIMD Saturating Signed Multiply Two Halfs & Add & Subtract)	163
4.51.2. KMSXDA (SIMD Saturating Signed Crossed Multiply Two Halfs & Add & Subtract)	163
4.52. KMSR64 (Signed Multiply and Saturating Subtract from 64-Bit Data)	166
4.53. KSLLW (Saturating Shift Left Logical for Word)	168
4.54. KSLLIW (Saturating Shift Left Logical Immediate for Word)	169
4.55. KSLL8 (SIMD 8-bit Saturating Shift Left Logical)	170
4.56. KSLLI8 (SIMD 8-bit Saturating Shift Left Logical Immediate)	172
4.57. KSLL16 (SIMD 16-bit Saturating Shift Left Logical)	174
4.58. KSLLI16 (SIMD 16-bit Saturating Shift Left Logical Immediate)	176
4.59. KSLRA8, KSLRA8.u	178
4.59.1. KSLRA8 (SIMD 8-bit Shift Left Logical with Saturation or Shift Right Arithmetic)	178
4.59.2. KSLRA8.u (SIMD 8-bit Shift Left Logical with Saturation or Rounding Shift Right Arithmetic)	178
4.60. KSLRA16, KSLRA16.u	181
4.60.1. KSLRA16 (SIMD 16-bit Shift Left Logical with Saturation or Shift Right Arithmetic)	181
4.60.2. KSLRA16.u (SIMD 16-bit Shift Left Logical with Saturation or Rounding Shift Right Arithmetic)	181
4.61. KSLRAW (Shift Left Logical with Q31 Saturation or Shift Right Arithmetic)	184
4.62. KSLRAW.u (Shift Left Logical with Q31 Saturation or Rounding Shift Right Arithmetic)	186
4.63. KSTAS16 (SIMD 16-bit Signed Saturating Straight Addition & Subtraction)	188
4.64. KSTSA16 (SIMD 16-bit Signed Saturating Straight Subtraction & Addition)	190
4.65. KSUB8 (SIMD 8-bit Signed Saturating Subtraction)	192
4.66. KSUB16 (SIMD 16-bit Signed Saturating Subtraction)	194
4.67. KSUB64 (64-bit Signed Saturating Subtraction)	196
4.68. KSUBH (Signed Subtraction with Q15 Saturation)	198

4.69. KSUBW (Signed Subtraction with Q31 Saturation)	199
4.70. KWMMUL, KWMMUL.u	200
4.70.1. KWMMUL (SIMD Saturating MSW Signed Multiply Word & Double)	200
4.70.2. KWMMUL.u (SIMD Saturating MSW Signed Multiply Word & Double with Rounding)	200
4.71. MADDR32 (Multiply and Add to 32-Bit Word)	202
4.72. MAXW (32-bit Signed Word Maximum)	203
4.73. MINW (32-bit Signed Word Minimum)	204
4.74. MSUBR32 (Multiply and Subtract from 32-Bit Word)	205
4.75. MULR64 (Multiply Word Unsigned to 64-bit Data)	206
4.76. MULSR64 (Multiply Word Signed to 64-bit Data)	208
4.77. PBSAD (Parallel Byte Sum of Absolute Difference)	210
4.78. PBSADA (Parallel Byte Sum of Absolute Difference Accum)	211
4.79. PKBB16, PKBT16, PKTT16, PKTB16	212
4.79.1. PKBB16 (Pack Two 16-bit Data from Both Bottom Half)	212
4.79.2. PKBT16 (Pack Two 16-bit Data from Bottom and Top Half)	212
4.79.3. PKTT16 (Pack Two 16-bit Data from Both Top Half)	212
4.79.4. PKTB16 (Pack Two 16-bit Data from Top and Bottom Half)	212
4.80. RADD8 (SIMD 8-bit Signed Halving Addition)	215
4.81. RADD16 (SIMD 16-bit Signed Halving Addition)	216
4.82. RADD64 (64-bit Signed Halving Addition)	217
4.83. RADDW (32-bit Signed Halving Addition)	219
4.84. RCRAS16 (SIMD 16-bit Signed Halving Cross Addition & Subtraction)	220
4.85. RCRSA16 (SIMD 16-bit Signed Halving Cross Subtraction & Addition)	222
4.86. RDOV (Read OV flag)	224
4.87. RSTAS16 (SIMD 16-bit Signed Halving Straight Addition & Subtraction)	225
4.88. RSTSA16 (SIMD 16-bit Signed Halving Straight Subtraction & Addition)	227
4.89. RSUB8 (SIMD 8-bit Signed Halving Subtraction)	229
4.90. RSUB16 (SIMD 16-bit Signed Halving Subtraction)	231
4.91. RSUB64 (64-bit Signed Halving Subtraction)	233
4.92. RSUBW (32-bit Signed Halving Subtraction)	235
4.93. SCLIP8 (SIMD 8-bit Signed Clip Value)	236
4.94. SCLIP16 (SIMD 16-bit Signed Clip Value)	238
4.95. SCLIP32 (SIMD 32-bit Signed Clip Value)	240
4.96. SCMPLE8 (SIMD 8-bit Signed Compare Less Than & Equal)	242
4.97. SCMPLE16 (SIMD 16-bit Signed Compare Less Than & Equal)	243
4.98. SCMPLT8 (SIMD 8-bit Signed Compare Less Than)	244
4.99. SCMPLT16 (SIMD 16-bit Signed Compare Less Than)	245
4.100. SLL8 (SIMD 8-bit Shift Left Logical)	246
4.101. SLLI8 (SIMD 8-bit Shift Left Logical Immediate)	247
4.102. SLL16 (SIMD 16-bit Shift Left Logical)	248
4.103. SLLI16 (SIMD 16-bit Shift Left Logical Immediate)	249

4.104. SMAL (Signed Multiply Halfs & Add 64-bit).....	250
4.105. SMALBB, SMALBT, SMALTT.....	252
4.105.1. SMALBB (Signed Multiply Bottom Halfs & Add 64-bit).....	252
4.105.2. SMALBT (Signed Multiply Bottom Half & Top Half & Add 64-bit).....	252
4.105.3. SMALTT (Signed Multiply Top Halfs & Add 64-bit).....	252
4.106. SMALDA, SMALXDA	256
4.106.1. SMALDA (Signed Multiply Two Halfs and Two Adds 64-bit).....	256
4.106.2. SMALXDA (Signed Crossed Multiply Two Halfs and Two Adds 64-bit)	256
4.107. SMALDS, SMALDRS, SMALXDS.....	259
4.107.1. SMALDS (Signed Multiply Two Halfs & Subtract & Add 64-bit).....	259
4.107.2. SMALDRS (Signed Multiply Two Halfs & Reverse Subtract & Add 64-bit)	259
4.107.3. SMALXDS (Signed Crossed Multiply Two Halfs & Subtract & Add 64-bit)	259
4.108. SMAR64 (Signed Multiply and Add to 64-Bit Data)	263
4.109. SMAQA (Signed Multiply Four Bytes with 32-bit Adds).....	265
4.110. SMAQA.SU (Signed and Unsigned Multiply Four Bytes with 32-bit Adds)	267
4.111. SMAX8 (SIMD 8-bit Signed Maximum)	269
4.112. SMAX16 (SIMD 16-bit Signed Maximum).....	270
4.113. SMBB16, SMBT16, SMTT16.....	271
4.113.1. SMBB16 (SIMD Signed Multiply Bottom Half & Bottom Half)	271
4.113.2. SMBT16 (SIMD Signed Multiply Bottom Half & Top Half)	271
4.113.3. SMTT16 (SIMD Signed Multiply Top Half & Top Half)	271
4.114. SMDS, SMDRS, SMXDS	274
4.114.1. SMDS (SIMD Signed Multiply Two Halfs and Subtract).....	274
4.114.2. SMDRS (SIMD Signed Multiply Two Halfs and Reverse Subtract)	274
4.114.3. SMXDS (SIMD Signed Crossed Multiply Two Halfs and Subtract)	274
4.115. SMIN8 (SIMD 8-bit Signed Minimum).....	277
4.116. SMIN16 (SIMD 16-bit Signed Minimum).....	278
4.117. SMMUL, SMMUL.u.....	279
4.117.1. SMMUL (SIMD MSW Signed Multiply Word)	279
4.117.2. SMMUL.u (SIMD MSW Signed Multiply Word with Rounding)	279
4.118. SMMWB, SMMWB.u	281
4.118.1. SMMWB (SIMD MSW Signed Multiply Word and Bottom Half)	281
4.118.2. SMMWB.u (SIMD MSW Signed Multiply Word and Bottom Half with Rounding)	281
4.119. SMMWT, SMMWT.u	283
4.119.1. SMMWT (SIMD MSW Signed Multiply Word and Top Half)	283
4.119.2. SMMWT.u (SIMD MSW Signed Multiply Word and Top Half with Rounding)	283
4.120. SMSLDA, SMSLXDA	285
4.120.1. SMSLDA (Signed Multiply Two Halfs & Add & Subtract 64-bit).....	285
4.120.2. SMSLXDA (Signed Crossed Multiply Two Halfs & Add & Subtract 64-bit)	285
4.121. SMSR64 (Signed Multiply and Subtract from 64-Bit Data)	288
4.122. SMUL8, SMULX8.....	290

4.122.1. SMUL8 (SIMD Signed 8-bit Multiply).....	290
4.122.2. SMULX8 (SIMD Signed Crossed 8-bit Multiply).....	290
4.123. SMUL16, SMULX16.....	293
4.123.1. SMUL16 (SIMD Signed 16-bit Multiply).....	293
4.123.2. SMULX16 (SIMD Signed Crossed 16-bit Multiply).....	293
4.124. SRA.u (Rounding Shift Right Arithmetic).....	296
4.125. SRAI.u (Rounding Shift Right Arithmetic Immediate).....	298
4.126. SRA8, SRA8.u	300
4.126.1. SRA8 (SIMD 8-bit Shift Right Arithmetic).....	300
4.126.2. SRA8.u (SIMD 8-bit Rounding Shift Right Arithmetic).....	300
4.127. SRAI8, SRAI8.u	302
4.127.1. SRAI8 (SIMD 8-bit Shift Right Arithmetic Immediate).....	302
4.127.2. SRAI8.u (SIMD 8-bit Rounding Shift Right Arithmetic Immediate)	302
4.128. SRA16, SRA16.u	304
4.128.1. SRA16 (SIMD 16-bit Shift Right Arithmetic)	304
4.128.2. SRA16.u (SIMD 16-bit Rounding Shift Right Arithmetic)	304
4.129. SRAI16, SRAI16.u	306
4.129.1. SRAI16 (SIMD 16-bit Shift Right Arithmetic Immediate).....	306
4.129.2. SRAI16.u (SIMD 16-bit Rounding Shift Right Arithmetic Immediate)	306
4.130. SRL8, SRL8.u	308
4.130.1. SRL8 (SIMD 8-bit Shift Right Logical).....	308
4.130.2. SRL8.u (SIMD 8-bit Rounding Shift Right Logical)	308
4.131. SRLI8, SRLI8.u	310
4.131.1. SRLI8 (SIMD 8-bit Shift Right Logical Immediate)	310
4.131.2. SRLI8.u (SIMD 8-bit Rounding Shift Right Logical Immediate)	310
4.132. SRL16, SRL16.u	312
4.132.1. SRL16 (SIMD 16-bit Shift Right Logical)	312
4.132.2. SRL16.u (SIMD 16-bit Rounding Shift Right Logical)	312
4.133. SRLI16, SRLI16.u	314
4.133.1. SRLI16 (SIMD 16-bit Shift Right Logical Immediate)	314
4.133.2. SRLI16.u (SIMD 16-bit Rounding Shift Right Logical Immediate)	314
4.134. STAS16 (SIMD 16-bit Straight Addition & Subtraction)	316
4.135. STSA16 (SIMD 16-bit Straight Subtraction & Addition)	318
4.136. SUB8 (SIMD 8-bit Subtraction)	320
4.137. SUB16 (SIMD 16-bit Subtraction)	321
4.138. SUB64 (64-bit Subtraction)	322
4.139. SUNPKD810, SUNPKD820, SUNPKD830, SUNPKD831, SUNPKD832	324
4.139.1. SUNPKD810 (Signed Unpacking Bytes 1 & 0)	324
4.139.2. SUNPKD820 (Signed Unpacking Bytes 2 & 0)	324
4.139.3. SUNPKD830 (Signed Unpacking Bytes 3 & 0)	324
4.139.4. SUNPKD831 (Signed Unpacking Bytes 3 & 1)	324

4.139.5. SUNPKD832 (Signed Unpacking Bytes 3 & 2)	324
4.140. SWAP8 (Swap Byte within Halfword)	327
4.141. SWAP16 (Swap Halfword within Word)	328
4.142. UCLIP8 (SIMD 8-bit Unsigned Clip Value)	329
4.143. UCLIP16 (SIMD 16-bit Unsigned Clip Value)	331
4.144. UCLIP32 (SIMD 32-bit Unsigned Clip Value)	333
4.145. UCMPLE8 (SIMD 8-bit Unsigned Compare Less Than & Equal)	335
4.146. UCMPLE16 (SIMD 16-bit Unsigned Compare Less Than & Equal)	336
4.147. UCMPLT8 (SIMD 8-bit Unsigned Compare Less Than)	337
4.148. UCMPLT16 (SIMD 16-bit Unsigned Compare Less Than)	338
4.149. UKADD8 (SIMD 8-bit Unsigned Saturating Addition)	339
4.150. UKADD16 (SIMD 16-bit Unsigned Saturating Addition)	341
4.151. UKADD64 (64-bit Unsigned Saturating Addition)	343
4.152. UKADDH (Unsigned Addition with U16 Saturation)	345
4.153. UKADDW (Unsigned Addition with U32 Saturation)	346
4.154. UKCRAS16 (SIMD 16-bit Unsigned Saturating Cross Addition & Subtraction)	347
4.155. UKCRSA16 (SIMD 16-bit Unsigned Saturating Cross Subtraction & Addition)	349
4.156. UKMAR64 (Unsigned Multiply and Saturating Add to 64-Bit Data)	351
4.157. UKMSR64 (Unsigned Multiply and Saturating Subtract from 64-Bit Data)	353
4.158. UKSTAS16 (SIMD 16-bit Unsigned Saturating Straight Addition & Subtraction)	355
4.159. UKSTSA16 (SIMD 16-bit Unsigned Saturating Straight Subtraction & Addition)	357
4.160. UKSUB8 (SIMD 8-bit Unsigned Saturating Subtraction)	359
4.161. UKSUB16 (SIMD 16-bit Unsigned Saturating Subtraction)	361
4.162. UKSUB64 (64-bit Unsigned Saturating Subtraction)	363
4.163. UKSUBH (Unsigned Subtraction with U16 Saturation)	365
4.164. UKSUBW (Unsigned Subtraction with U32 Saturation)	366
4.165. UMAR64 (Unsigned Multiply and Add to 64-Bit Data)	367
4.166. UMAQA (Unsigned Multiply Four Bytes with 32-bit Adds)	369
4.167. UMAX8 (SIMD 8-bit Unsigned Maximum)	371
4.168. UMAX16 (SIMD 16-bit Unsigned Maximum)	372
4.169. UMIN8 (SIMD 8-bit Unsigned Minimum)	373
4.170. UMIN16 (SIMD 16-bit Unsigned Minimum)	374
4.171. UMSR64 (Unsigned Multiply and Subtract from 64-Bit Data)	375
4.172. UMUL8, UMULX8	377
4.172.1. UMUL8 (SIMD Unsigned 8-bit Multiply)	377
4.172.2. UMULX8 (SIMD Unsigned Crossed 8-bit Multiply)	377
4.173. UMUL16, UMULX16	380
4.173.1. UMUL16 (SIMD Unsigned 16-bit Multiply)	380
4.173.2. UMULX16 (SIMD Unsigned Crossed 16-bit Multiply)	380
4.174. URADD8 (SIMD 8-bit Unsigned Halving Addition)	383
4.175. URADD16 (SIMD 16-bit Unsigned Halving Addition)	385

4.176. URADD64 (64-bit Unsigned Halving Addition)	387
4.177. URADDW (32-bit Unsigned Halving Addition)	389
4.178. URCRAS16 (SIMD 16-bit Unsigned Halving Cross Addition & Subtraction)	390
4.179. URCRSA16 (SIMD 16-bit Unsigned Halving Cross Subtraction & Addition)	392
4.180. URSTAS16 (SIMD 16-bit Unsigned Halving Straight Addition & Subtraction)	394
4.181. URSTSA16 (SIMD 16-bit Unsigned Halving Straight Subtraction & Addition)	396
4.182. URSUB8 (SIMD 8-bit Unsigned Halving Subtraction)	398
4.183. URSUB16 (SIMD 16-bit Unsigned Halving Subtraction)	400
4.184. URSUB64 (64-bit Unsigned Halving Subtraction)	402
4.185. URSUBW (32-bit Unsigned Halving Subtraction)	404
4.186. WEXTI (Extract Word from 64-bit Immediate)	405
4.187. WEXT (Extract Word from 64-bit)	407
4.188. ZUNPKD810, ZUNPKD820, ZUNPKD830, ZUNPKD831, ZUNPKD832	409
4.188.1. ZUNPKD810 (Unigned Unpacking Bytes 1 & 0)	409
4.188.2. ZUNPKD820 (Unigned Unpacking Bytes 2 & 0)	409
4.188.3. ZUNPKD830 (Unigned Unpacking Bytes 3 & 0)	409
4.188.4. ZUNPKD831 (Unigned Unpacking Bytes 3 & 1)	409
4.188.5. ZUNPKD832 (Unigned Unpacking Bytes 3 & 2)	409
5. Detailed Instruction Descriptions (RV64 Only)	412
5.1. ADD32 (SIMD 32-bit Addition)	413
5.2. CRAS32 (SIMD 32-bit Cross Addition & Subtraction)	414
5.3. CRSA32 (SIMD 32-bit Cross Subtraction & Addition)	415
5.4. KABS32 (Scalar 32-bit Absolute Value with Saturation)	416
5.5. KADD32 (SIMD 32-bit Signed Saturating Addition)	418
5.6. KCRAS32 (SIMD 32-bit Signed Saturating Cross Addition & Subtraction)	419
5.7. KCRSA32 (SIMD 32-bit Signed Saturating Cross Subtraction & Addition)	421
5.8. KDMBB16, KDMBT16, KDMTT16	423
5.8.1. KDMBB16 (SIMD Signed Saturating Double Multiply B16 x B16)	423
5.8.2. KDMBT16 (SIMD Signed Saturating Double Multiply B16 x T16)	423
5.8.3. KDMTT16 (SIMD Signed Saturating Double Multiply T16 x T16)	423
5.9. KDMABB16, KDMABT16, KDMATT16	425
5.9.1. KDMABB16 (SIMD Signed Saturating Double Multiply Addition B16 x B16)	425
5.9.2. KDMABT16 (SIMD Signed Saturating Double Multiply Addition B16 x T16)	425
5.9.3. KDMATT16 (SIMD Signed Saturating Double Multiply Addition T16 x T16)	425
5.10. KHMBB16, KHMBT16, KHMTT16	428
5.10.1. KHMBB16 (SIMD Signed Saturating Half Multiply B16 x B16)	428
5.10.2. KHMBT16 (SIMD Signed Saturating Half Multiply B16 x T16)	428
5.10.3. KHMTT16 (SIMD Signed Saturating Half Multiply T16 x T16)	428
5.11. KMABB32, KMABT32, KMATT32	430
5.11.1. KMABB32 (Saturating Signed Multiply Bottom Words & Add)	430
5.11.2. KMABT32 (Saturating Signed Multiply Bottom & Top Words & Add)	430

5.11.3. KMATT32 (Saturating Signed Multiply Top Words & Add)	430
5.12. KMADA32, KMAXDA32	433
5.12.1. KMADA32 (Saturating Signed Multiply Two Words and Two Adds)	433
5.12.2. KMAXDA32 (Saturating Signed Crossed Multiply Two Words and Two Adds)	433
5.13. KMDA32, KMXDA32	435
5.13.1. KMDA32 (Signed Multiply Two Words and Add)	435
5.13.2. KMXDA32 (Signed Crossed Multiply Two Words and Add)	435
5.14. KMADS32, KMADRS32, KMAXDS32	437
5.14.1. KMADS32 (Saturating Signed Multiply Two Words & Subtract & Add)	437
5.14.2. KMADRS32 (Saturating Signed Multiply Two Words & Reverse Subtract & Add)	437
5.14.3. KMAXDS32 (Saturating Signed Crossed Multiply Two Words & Subtract & Add)	437
5.15. KMSDA32, KMSXDA32	440
5.15.1. KMSDA32 (Saturating Signed Multiply Two Words & Add & Subtract)	440
5.15.2. KMSXDA32 (Saturating Signed Crossed Multiply Two Words & Add & Subtract)	440
5.16. KSLL32 (SIMD 32-bit Saturating Shift Left Logical)	442
5.17. KSLLI32 (SIMD 32-bit Saturating Shift Left Logical Immediate)	444
5.18. KSLRA32, KSLRA32.u	446
5.18.1. KSLRA32 (SIMD 32-bit Shift Left Logical with Saturation or Shift Right Arithmetic) ..	446
5.18.2. KSLRA32.u (SIMD 32-bit Shift Left Logical with Saturation or Rounding Shift Right Arithmetic)	446
5.19. KSTAS32 (SIMD 32-bit Signed Saturating Straight Addition & Subtraction)	448
5.20. KSTSA32 (SIMD 32-bit Signed Saturating Straight Subtraction & Addition)	450
5.21. KSUB32 (SIMD 32-bit Signed Saturating Subtraction)	452
5.22. PKBB32, PKBT32, PKTT32, PKTB32	453
5.22.1. PKBB32 (Pack Two 32-bit Data from Both Bottom Half)	453
5.22.2. PKBT32 (Pack Two 32-bit Data from Bottom and Top Half)	453
5.22.3. PKTT32 (Pack Two 32-bit Data from Both Top Half)	453
5.22.4. PKTB32 (Pack Two 32-bit Data from Top and Bottom Half)	453
5.23. RADD32 (SIMD 32-bit Signed Halving Addition)	455
5.24. RCRAS32 (SIMD 32-bit Signed Halving Cross Addition & Subtraction)	456
5.25. RCRSA32 (SIMD 32-bit Signed Halving Cross Subtraction & Addition)	457
5.26. RSTAS32 (SIMD 32-bit Signed Halving Straight Addition & Subtraction)	458
5.27. RSTSA32 (SIMD 32-bit Signed Halving Straight Subtraction & Addition)	459
5.28. RSUB32 (SIMD 32-bit Signed Halving Subtraction)	460
5.29. SLL32 (SIMD 32-bit Shift Left Logical)	461
5.30. SLLI32 (SIMD 32-bit Shift Left Logical Immediate)	462
5.31. SMAX32 (SIMD 32-bit Signed Maximum)	463
5.32. SMBB32, SMBT32, SMTT32	464
5.32.1. SMBB32 (Signed Multiply Bottom Word & Bottom Word)	464
5.32.2. SMBT32 (Signed Multiply Bottom Word & Top Word)	464
5.32.3. SMTT32 (Signed Multiply Top Word & Top Word)	464

5.33. SMDS32, SMDRS32, SMXDS32	466
5.33.1. SMDS32 (Signed Multiply Two Words and Subtract)	466
5.33.2. SMDRS32 (Signed Multiply Two Words and Reverse Subtract)	466
5.33.3. SMXDS32 (Signed Crossed Multiply Two Words and Subtract)	466
5.34. SMIN32 (SIMD 32-bit Signed Minimum)	468
5.35. SRA32, SRA32.u	469
5.35.1. SRA32 (SIMD 32-bit Shift Right Arithmetic)	469
5.35.2. SRA32.u (SIMD 32-bit Rounding Shift Right Arithmetic)	469
5.36. SRAI32, SRAI32.u	471
5.36.1. SRAI32 (SIMD 32-bit Shift Right Arithmetic Immediate)	471
5.36.2. SRAI32.u (SIMD 32-bit Rounding Shift Right Arithmetic Immediate)	471
5.37. SRAIW.u (Rounding Shift Right Arithmetic Immediate Word)	473
5.38. SRL32, SRL32.u	474
5.38.1. SRL32 (SIMD 32-bit Shift Right Logical)	474
5.38.2. SRL32.u (SIMD 32-bit Rounding Shift Right Logical)	474
5.39. SRLI32, SRLI32.u	476
5.39.1. SRLI32 (SIMD 32-bit Shift Right Logical Immediate)	476
5.39.2. SRLI32.u (SIMD 32-bit Rounding Shift Right Logical Immediate)	476
5.40. STAS32 (SIMD 32-bit Straight Addition & Subtraction)	478
5.41. STSA32 (SIMD 32-bit Straight Subtraction & Addition)	479
5.42. SUB32 (SIMD 32-bit Subtraction)	480
5.43. UKADD32 (SIMD 32-bit Unsigned Saturating Addition)	481
5.44. UKCRAS32 (SIMD 32-bit Unsigned Saturating Cross Addition & Subtraction)	482
5.45. UKRSA32 (SIMD 32-bit Unsigned Saturating Cross Subtraction & Addition)	484
5.46. UKSTAS32 (SIMD 32-bit Unsigned Saturating Straight Addition & Subtraction)	486
5.47. UKSTA32 (SIMD 32-bit Unsigned Saturating Straight Subtraction & Addition)	488
5.48. UKSUB32 (SIMD 32-bit Unsigned Saturating Subtraction)	490
5.49. UMAX32 (SIMD 32-bit Unsigned Maximum)	491
5.50. UMIN32 (SIMD 32-bit Unsigned Minimum)	492
5.51. URADD32 (SIMD 32-bit Unsigned Halving Addition)	493
5.52. URCRAS32 (SIMD 32-bit Unsigned Halving Cross Addition & Subtraction)	494
5.53. URCRSA32 (SIMD 32-bit Unsigned Halving Cross Subtraction & Addition)	495
5.54. URSTAS32 (SIMD 32-bit Unsigned Halving Straight Addition & Subtraction)	496
5.55. URSTA32 (SIMD 32-bit Unsigned Halving Straight Subtraction & Addition)	497
5.56. URSUB32 (SIMD 32-bit Unsigned Halving Subtraction)	498
6. New User Control & Status Registers	499
6.1. Code Register	500
7. Instruction Encoding Table	501
Appendix A: Instruction Latency and Throughput	505
A.1. Example RV32 and RV64 Cores with 5 Stages of Pipeline	505
A.1.1. Listed by Individual Instruction	505

Revision History

Rev.	Revision Date	Author	Revised Content
v0.5.4	2020/03/02	Chuanhua Chang	Added P subset extensions (Chapter 3)
v0.5.3	2019/11/8	Chuanhua Chang	Adjusted BPICK encoding along with the following 20 instructions: STAS16, RSTAS16, KSTAS16, URSTAS16, UKSTAS16, STSA16, RSTSA16, KSTSA16, URSTSA16, UKSTSA16, STAS32, RSTAS32, KSTAS32, URSTAS32, UKSTAS32, STSA32, RSTSA32, KSTSA32, URSTSA32, UKSTSA32. (Chapter 7)
v0.5.2	2019/10/17	Chuanhua Chang	Fixed SRAIW.u operation typo. (Section 5.37)
v0.5.1	2019/10/8	Chuanhua Chang	Fixed SLLI32 encoding. (Section 5.30)
v0.5.0	2019/4/26	Chuanhua Chang	Initial Release.

Chapter 1. Introduction

Digital Signal Processing (DSP), has emerged as an important technology for modern electronic systems. A wide range of modern applications employ DSP algorithms to solve problems in their particular domains, including sensor fusion, servo motor control, audio decode/encode, speech synthesis and coding, MPEG4 decode, medical imaging, computer vision, embedded control, robotics, human interface, etc.

The proposed P instruction set extension increases the DSP algorithm processing capabilities of the RISC-V CPU IP products. With the addition of the RISC-V P instruction set extension, the RISC-V CPUs can now run these various DSP applications with lower power and higher performance.

Chapter 2. DSP ISA Extension Instruction Summary

2.1. Shorthand Definitions

- $r.H == rH1: r[31:16], r.L == r.H0: r[15:0]$
- $r.B3: r[31:24], r.B2: r[23:16], r.B1: r[15:8], r.B0: r[7:0]$
- $r.B[x]: r[(x*8+7):(x*8+0)]$
- $r.H[x]: r[(x*16+7):(x*16+0)]$
- $r.W[x]: r[(x*32+31):(x*32+0)]$
- $r[xU]:$ the upper 32-bit of a 64-bit number; xU represents the GPR number that contains this upper part 32-bit value.
- $r[xL]:$ the lower 32-bit of a 64-bit number; xL represents the GPR number that contains this lower part 32-bit value.
- $r[xU].r[xL]:$ a 64-bit number that is formed from a pair of GPRs.
- $s>>:$ signed arithmetic right shift:
- $u>>:$ unsigned logical right shift
- $SAT.Qn():$ Saturate to the range of $[-2^n, 2^n-1]$, if saturation happens, set PSW.OV.
- $SAT.Um():$ Saturate to the range of $[0, 2^m-1]$, if saturation happens, set PSW.OV.
- $RUND():$ Indicate “rounding”, i.e., add 1 to the most significant discarded bit for right shift or MSW-type multiplication instructions.
- Sign or Zero Extending functions:
 - $SEm(data):$ Sign-Extend data to m-bit.
 - $ZEm(data):$ Zero-Extend data to m-bit.
- $ABS(x):$ Calculate the absolute value of “ x ”.
- $CONCAT(x,y):$ Concatinate “ x ” and “ y ” to form a value.
- $u<:$ Unsigned less than comparison.
- $u\leq:$ Unsigned less than & equal comparison.
- $u>:$ Unsigned greater than comparison.
- $s^*:$ Signed multiplication.
- $u^*:$ Unsigned multiplication.

2.2. SIMD Data Processing Instructions

2.2.1. 16-bit Addition & Subtraction Instructions

Based on the combination of the types of the two 16-bit arithmetic operations, the SIMD 16-bit add/subtract instructions can be classified into 6 main categories: Addition (two 16-bit addition), Subtraction (two 16-bit subtraction), Crossed Add & Sub (one addition and one subtraction), and Crossed Sub & Add (one subtraction and one addition), Straight Add & Sub (one addition and one subtraction), and Straight Sub & Add (one subtraction and one addition).

Based on the way of how an overflow condition is handled, the SIMD 16-bit add/subtract instructions can be classified into 5 groups: Wrap-around (dropping overflow), Signed Halving (keeping overflow by dropping 1 LSB bit), Unsigned Halving, Signed Saturation (clipping overflow), and Unsigned Saturation.

Together, there are 30 SIMD 16-bit add/subtract instructions.

Table 1. SIMD 16-bit Add/Subtract Instructions

No.	Mnemonic	Instruction	Operation
1	ADD16 rt, ra, rb	16-bit Addition	$rt.H[x] = ra.H[x] + rb.H[x];$ (RV32: $x=1..0$, RV64: $x=3..0$)
2	RADD16 rt, ra, rb	16-bit Signed Halving Addition	$rt.H[x] = (ra.H[x] + rb.H[x]) s\gg 1;$ (RV32: $x=1..0$, RV64: $x=3..0$)
3	URADD16 rt, ra, rb	16-bit Unsigned Halving Addition	$rt.H[x] = (ra.H[x] + rb.H[x]) u\gg 1;$ (RV32: $x=1..0$, RV64: $x=3..0$)
4	KADD16 rt, ra, rb	16-bit Signed Saturating Addition	$rt.H[x] = SAT.Q15(ra.H[x] + rb.H[x]);$ (RV32: $x=1..0$, RV64: $x=3..0$)
5	UKADD16 rt, ra, rb	16-bit Unsigned Saturating Addition	$rt.H[x] = SAT.U16(ra.H[x] + rb.H[x]);$ (RV32: $x=1..0$, RV64: $x=3..0$)
6	SUB16 rt, ra, rb	16-bit Subtraction	$rt.H[x] = ra.H[x] - rb.H[x];$ (RV32: $x=1..0$, RV64: $x=3..0$)
7	RSUB16 rt, ra, rb	16-bit Signed Halving Subtraction	$rt.H[x] = (ra.H[x] - rb.H[x]) s\gg 1;$ (RV32: $x=1..0$, RV64: $x=3..0$)

No.	Mnemonic	Instruction	Operation
8	URSUB16 rt, ra, rb	16-bit Unsigned Halving Subtraction	$rt.H[x] = (ra.H[x] - rb.H[x]) \text{ u}>> 1;$ (RV32: $x=1..0$, RV64: $x=3..0$)
9	KSUB16 rt, ra, rb	16-bit Signed Saturating Subtraction	$rt.H[x] = \text{SAT.Q15}(ra.H[x] - rb.H[x]);$ (RV32: $x=1..0$, RV64: $x=3..0$)
10	UKSUB16 rt, ra, rb	16-bit Unsigned Saturating Subtraction	$rt.H[x] = \text{SAT.U16}(ra.H[x] - rb.H[x]);$ (RV32: $x=1..0$, RV64: $x=3..0$)
11	CRAS16 rt, ra, rb	16-bit Cross Add & Sub	$rt.H[x] = ra.H[x] + rb.H[x-1];$ $rt.H[x-1] = ra.H[x-1] - rb.H[x];$ (RV32: $x=1$, RV64: $x=1,3$)
12	RCRAS16 rt, ra, rb	16-bit Signed Halving Cross Add & Sub	$rt.H[x] = (ra.H[x] + rb.H[x-1]) \text{ s}>> 1;$ $rt.H[x-1] = (ra.H[x-1] - rb.H[x]) \text{ s}>> 1;$ (RV32: $x=1$, RV64: $x=1,3$)
13	URCRAS16 rt, ra, rb	16-bit Unsigned Halving Cross Add & Sub	$rt.H[x] = (ra.H[x] + rb.H[x-1]) \text{ u}>> 1;$ $rt.H[x-1] = (ra.H[x-1] - rb.H[x]) \text{ u}>> 1;$ (RV32: $x=1$, RV64: $x=1,3$)
14	KCRAS16 rt, ra, rb	16-bit Signed Saturating Cross Add & Sub	$rt.H[x] = \text{SAT.Q15}(ra.H[x] + rb.H[x-1]);$ $rt.H[x-1] = \text{SAT.Q15}(ra.H[x-1] - rb.H[x]);$ (RV32: $x=1$, RV64: $x=1,3$)
15	UKCRAS16 rt, ra, rb	16-bit Unsigned Saturating Cross Add & Sub	$rt.H[x] = \text{SAT.U16}(ra.H[x] + rb.H[x-1]);$ $rt.H[x-1] = \text{SAT.U16}(ra.H[x-1] - rb.H[x]);$ (RV32: $x=1$, RV64: $x=1,3$)
16	CRSA16 rt, ra, rb	16-bit Cross Sub & Add	$rt.H[x] = ra.H[x] - rb.H[x-1];$ $rt.H[x-1] = ra.H[x-1] + rb.H[x];$ (RV32: $x=1$, RV64: $x=1,3$)
17	RCRSA16 rt, ra, rb	16-bit Signed Halving Cross Sub & Add	$rt.H[x] = (ra.H[x] - rb.H[x-1]) \text{ s}>> 1;$ $rt.H[x-1] = (ra.H[x-1] + rb.H[x]) \text{ s}>> 1;$ (RV32: $x=1$, RV64: $x=1,3$)

No.	Mnemonic	Instruction	Operation
18	URCRSA16 rt, ra, rb	16-bit Unsigned Halving Cross Sub & Add	$rt.H[x] = (ra.H[x] - rb.H[x-1]) \text{ u}>> 1;$ $rt.H[x-1] = (ra.H[x-1] + rb.H[x]) \text{ u}>> 1;$ (RV32: $x=1$, RV64: $x=1,3$)
19	KCRSA16 rt, ra, rb	16-bit Signed Saturating Cross Sub & Add	$rt.H[x] = \text{SAT.Q15}(ra.H[x] - rb.H[x-1]);$ $rt.H[x-1] = \text{SAT.Q15}(ra.H[x-1] + rb.H[x]);$ (RV32: $x=1$, RV64: $x=1,3$)
20	UKCRSA16 rt, ra, rb	16-bit Unsigned Saturating Cross Sub & Add	$rt.H[x] = \text{SAT.U16}(ra.H[x] - rb.H[x-1]);$ $rt.H[x-1] = \text{SAT.U16}(ra.H[x-1] + rb.H[x]);$ (RV32: $x=1$, RV64: $x=1,3$)
21	STAS16 rt, ra, rb	16-bit Straight Add & Sub	$rt.H[x] = ra.H[x] + rb.H[x];$ $rt.H[x-1] = ra.H[x-1] - rb.H[x-1];$ (RV32: $x=1$, RV64: $x=1,3$)
22	RSTAS16 rt, ra, rb	16-bit Signed Halving Straight Add & Sub	$rt.H[x] = (ra.H[x] + rb.H[x]) \text{ s}>> 1;$ $rt.H[x-1] = (ra.H[x-1] - rb.H[x-1]) \text{ s}>> 1;$ (RV32: $x=1$, RV64: $x=1,3$)
23	URSTAS16 rt, ra, rb	16-bit Unsigned Halving Straight Add & Sub	$rt.H[x] = (ra.H[x] + rb.H[x]) \text{ u}>> 1;$ $rt.H[x-1] = (ra.H[x-1] - rb.H[x-1]) \text{ u}>> 1;$ (RV32: $x=1$, RV64: $x=1,3$)
24	KSTAS16 rt, ra, rb	16-bit Signed Saturating Straight Add & Sub	$rt.H[x] = \text{SAT.Q15}(ra.H[x] + rb.H[x]);$ $rt.H[x-1] = \text{SAT.Q15}(ra.H[x-1] - rb.H[x-1]);$ (RV32: $x=1$, RV64: $x=1,3$)
25	UKSTAS16 rt, ra, rb	16-bit Unsigned Saturating Straight Add & Sub	$rt.H[x] = \text{SAT.U16}(ra.H[x] + rb.H[x]);$ $rt.H[x-1] = \text{SAT.U16}(ra.H[x-1] - rb.H[x-1]);$ (RV32: $x=1$, RV64: $x=1,3$)
26	STSA16 rt, ra, rb	16-bit Straight Sub & Add	$rt.H[x] = ra.H[x] - rb.H[x];$ $rt.H[x-1] = ra.H[x-1] + rb.H[x-1];$ (RV32: $x=1$, RV64: $x=1,3$)

No.	Mnemonic	Instruction	Operation
27	RSTSA16 rt, ra, rb	16-bit Signed Halving Straight Sub & Add	$rt.H[x] = (ra.H[x] - rb.H[x]) s\gg 1;$ $rt.H[x-1] = (ra.H[x-1] + rb.H[x-1]) s\gg 1;$ (RV32: $x=1$, RV64: $x=1,3$)
28	URSTSA16 rt, ra, rb	16-bit Unsigned Halving Straight Sub & Add	$rt.H[x] = (ra.H[x] - rb.H[x]) u\gg 1;$ $rt.H[x-1] = (ra.H[x-1] + rb.H[x-1]) u\gg 1;$ (RV32: $x=1$, RV64: $x=1,3$)
29	KSTSA16 rt, ra, rb	16-bit Signed Saturating Straight Sub & Add	$rt.H[x] = SAT.Q15(ra.H[x] - rb.H[x]);$ $rt.H[x-1] = SAT.Q15(ra.H[x-1] + rb.H[x-1]);$ (RV32: $x=1$, RV64: $x=1,3$)
30	UKSTSA16 rt, ra, rb	16-bit Unsigned Saturating Straight Sub & Add	$rt.H[x] = SAT.U16(ra.H[x] - rb.H[x]);$ $rt.H[x-1] = SAT.U16(ra.H[x-1] + rb.H[x-1]);$ (RV32: $x=1$, RV64: $x=1,3$)

2.2.2. 8-bit Addition & Subtraction Instructions

Based on the types of the four 8-bit arithmetic operations, the SIMD 8-bit add/subtract instructions can be classified into 2 main categories: Addition (four 8-bit addition), and Subtraction (four 8-bit subtraction).

Based on the way of how an overflow condition is handled for singed or unsigned operation, the SIMD 8-bit add/subtract instructions can be classified into 5 groups: Wrap-around (dropping overflow), Signed Halving (keeping overflow by dropping 1 LSB bit), Unsigned Halving, Signed Saturation (clipping overflow), and Unsigned Saturation.

Together, there are 10 SIMD 8-bit add/subtract instructions.

Table 2. SIMD 8-bit Add/Subtract Instructions

No.	Mnemonic	Instruction	Operation
1	ADD8 rt, ra, rb	8-bit Addition	$rt.B[x] = ra.B[x] + rb.B[x];$ (RV32: $x=3..0$, RV64: $x=7..0$)
2	RADD8 rt, ra, rb	8-bit Signed Halving Addition	$rt.B[x] = (ra.B[x] + rb.B[x]) s\gg 1;$ (RV32: $x=3..0$, RV64: $x=7..0$)
3	URADD8 rt, ra, rb	8-bit Unsigned Halving Addition	$rt.B[x] = (ra.B[x] + rb.B[x]) u\gg 1;$ (RV32: $x=3..0$, RV64: $x=7..0$)
4	KADD8 rt, ra, rb	8-bit Signed Saturating Addition	$rt.B[x] = SAT.Q7(ra.B[x] + rb.B[x]);$ (RV32: $x=3..0$, RV64: $x=7..0$)
5	UKADD8 rt, ra, rb	8-bit Unsigned Saturating Addition	$rt.B[x] = SAT.U8(ra.B[x] + rb.B[x]);$ (RV32: $x=3..0$, RV64: $x=7..0$)
6	SUB8 rt, ra, rb	8-bit Subtraction	$rt.B[x] = ra.B[x] - rb.B[x];$ (RV32: $x=3..0$, RV64: $x=7..0$)
7	RSUB8 rt, ra, rb	8-bit Signed Halving Subtraction	$rt.B[x] = (ra.B[x] - rb.B[x]) s\gg 1;$ (RV32: $x=3..0$, RV64: $x=7..0$)
8	URSUB8 rt, ra, rb	8-bit Unsigned Halving Subtraction	$rt.B[x] = (ra.B[x] - rb.B[x]) u\gg 1;$ (RV32: $x=3..0$, RV64: $x=7..0$)

No.	Mnemonic	Instruction	Operation
9	KSUB8 rt, ra, rb	8-bit Signed Saturating Subtraction	$rt.B[x] = SAT.Q7(ra.B[x] - rb.B[x]);$ (RV32: $x=3..0$, RV64: $x=7..0$)
10	UKSUB8 rt, ra, rb	8-bit Unsigned Saturating Subtraction	$rt.B[x] = SAT.U8(ra.B[x] - rb.B[x]);$ (RV32: $x=3..0$, RV64: $x=7..0$)

2.2.3. 16-bit Shift Instructions

There are 14 instructions here.

Table 3. SIMD 16-bit Shift Instructions

No.	Mnemonic	Instruction	Operation
1	SRA16 rt, ra, rb	16-bit Shift Right Arithmetic	$rt.H[x] = ra.H[x] \text{ s}>> rb[3:0];$ (RV32: $x=1..0$, RV64: $x=3..0$)
2	SRAI16 rt, ra, im4u	16-bit Shift Right Arithmetic Immediate	$rt.H[x] = ra.H[x] \text{ s}>> im4u;$ (RV32: $x=1..0$, RV64: $x=3..0$)
3	SRA16.u rt, ra, rb	16-bit Rounding Shift Right Arithmetic	$rt.H[x] = \text{RUND}(ra.H[x] \text{ s}>> rb[3:0]);$ (RV32: $x=1..0$, RV64: $x=3..0$)
4	SRAI16.u rt, ra, im4u	16-bit Rounding Shift Right Arithmetic Immediate	$rt.H[x] = \text{RUND}(ra.H[x] \text{ s}>> im4u);$ (RV32: $x=1..0$, RV64: $x=3..0$)
5	SRL16 rt, ra, rb	16-bit Shift Right Logical	$rt.H[x] = ra.H[x] \text{ u}>> rb[3:0];$ (RV32: $x=1..0$, RV64: $x=3..0$)
6	SRLI16 rt, ra, im4u	16-bit Shift Right Logical Immediate	$rt.H[x] = ra.H[x] \text{ u}>> im4u;$ (RV32: $x=1..0$, RV64: $x=3..0$)
7	SRL16.u rt, ra, rb	16-bit Rounding Shift Right Logical	$rt.H[x] = \text{RUND}(ra.H[x] \text{ u}>> rb[3:0]);$ (RV32: $x=1..0$, RV64: $x=3..0$)
8	SRLI16.u rt, ra, im4u	16-bit Rounding Shift Right Logical Immediate	$rt.H[x] = \text{RUND}(ra.H[x] \text{ u}>> im4u);$ (RV32: $x=1..0$, RV64: $x=3..0$)
9	SLL16 rt, ra, rb	16-bit Shift Left Logical	$rt.H[x] = ra.Hx \ll rb[3:0];$ (RV32: $x=1..0$, RV64: $x=3..0$)
10	SLLI16 rt, ra, im4u	16-bit Shift Left Logical Immediate	$rt.H[x] = ra.H[x] \ll im4u;$ (RV32: $x=1..0$, RV64: $x=3..0$)

No.	Mnemonic	Instruction	Operation
11	KSLL16 rt, ra, rb	16-bit Saturating Shift Left Logical	$rt.H[x] = SAT.Q15(ra.H[x] \ll im4u);$ (RV32: $x=1..0$, RV64: $x=3..0$)
12	KSLLI16 rt, ra, im4u	16-bit Saturating Shift Left Logical Immediate	$rt.H[x] = SAT.Q15(ra.H[x] \ll rb[3:0]);$ (RV32: $x=1..0$, RV64: $x=3..0$)
13	KSLRA16 rt, ra, rb	16-bit Shift Left Logical with Saturation & Shift Right Arithmetic	$if (rb[4:0] < 0)$ $rt.H[x] = ra.H[x] s\gg -rb[4:0];$ $if (rb[4:0] > 0)$ $rt.H[x] = SAT.Q15(ra.H[x] \ll rb[4:0]);$ (RV32: $x=1..0$, RV64: $x=3..0$)
14	KSLRA16.u rt, ra, rb	16-bit Shift Left Logical with Saturation & Rounding Shift Right Arithmetic	$if (rb[4:0] < 0)$ $rt.H[x] = RUND(ra.H[x] s\gg -rb[4:0]);$ $if (rb[4:0] > 0)$ $rt.H[x] = SAT.Q15(ra.H[x] \ll rb[4:0]);$ (RV32: $x=1..0$, RV64: $x=3..0$)

2.2.4. 8-bit Shift Instructions

There are 14 instructions here.

Table 4. SIMD 8-bit Shift Instructions

No.	Mnemonic	Instruction	Operation
1	SRA8 rt, ra, rb	8-bit Shift Right Arithmetic	$rt.B[x] = ra.B[x] \text{ s} >> rb[2:0];$ (RV32: $x=3..0$, RV64: $x=7..0$)
2	SRAI8 rt, ra, im3u	8-bit Shift Right Arithmetic Immediate	$rt.B[x] = ra.B[x] \text{ s} >> im3u;$ (RV32: $x=3..0$, RV64: $x=7..0$)
3	SRA8.u rt, ra, rb	8-bit Rounding Shift Right Arithmetic	$rt.B[x] = \text{RUND}(ra.B[x] \text{ s} >> rb[2:0]);$ (RV32: $x=3..0$, RV64: $x=7..0$)
4	SRAI8.u rt, ra, im4u	8-bit Rounding Shift Right Arithmetic Immediate	$rt.B[x] = \text{RUND}(ra.B[x] \text{ s} >> im3u);$ (RV32: $x=3..0$, RV64: $x=7..0$)
5	SRL8 rt, ra, rb	8-bit Shift Right Logical	$rt.B[x] = ra.B[x] \text{ u} >> rb[2:0];$ (RV32: $x=3..0$, RV64: $x=7..0$)
6	SRLI8 rt, ra, im3u	8-bit Shift Right Logical Immediate	$rt.B[x] = ra.B[x] \text{ u} >> im3u;$ (RV32: $x=3..0$, RV64: $x=7..0$)
7	SRL8.u rt, ra, rb	8-bit Rounding Shift Right Logical	$rt.B[x] = \text{RUND}(ra.B[x] \text{ u} >> rb[2:0]);$ (RV32: $x=3..0$, RV64: $x=7..0$)
8	SRLI8.u rt, ra, im4u	8-bit Rounding Shift Right Logical Immediate	$rt.B[x] = \text{RUND}(ra.B[x] \text{ u} >> im3u);$ (RV32: $x=3..0$, RV64: $x=7..0$)
9	SLL8 rt, ra, rb	8-bit Shift Left Logical	$rt.B[x] = ra.B[x] \ll rb[2:0];$ (RV32: $x=3..0$, RV64: $x=7..0$)
10	SLLI8 rt, ra, im3u	8-bit Shift Left Logical Immediate	$rt.B[x] = ra.B[x] \ll im3u;$ (RV32: $x=3..0$, RV64: $x=7..0$)

No.	Mnemonic	Instruction	Operation
11	KSLI8 rt, ra, rb	8-bit Saturating Shift Left Logical	$rt.B[x] = SAT.Q7(ra.B[x] \ll rb[2:0]);$ (RV32: $x=3..0$, RV64: $x=7..0$)
12	KSLLI8 rt, ra, im3u	8-bit Saturating Shift Left Logical Immediate	$rt.B[x] = SAT.Q7(ra.B[x] \ll im3u);$ (RV32: $x=3..0$, RV64: $x=7..0$)
13	KSLRA8 rt, ra, rb	8-bit Shift Left Logical with Saturation & Shift Right Arithmetic	if ($rb[3:0] < 0$) $rt.B[x] = ra.B[x] s\gg -rb[3:0];$ if ($rb[3:0] > 0$) $rt.B[x] = SAT.Q7(ra.B[x] \ll rb[3:0]);$ (RV32: $x=3..0$, RV64: $x=7..0$)
14	KSLRA8.u rt, ra, rb	8-bit Shift Left Logical with Saturation & Rounding Shift Right Arithmetic	if ($rb[3:0] < 0$) $rt.B[x] = RUND(ra.B[x] s\gg -rb[3:0]);$ if ($rb[3:0] > 0$) $rt.B[x] = SAT.Q7(ra.B[x] \ll rb[3:0]);$ (RV32: $x=3..0$, RV64: $x=7..0$)

2.2.5. 16-bit Compare Instructions

There are 5 instructions here.

Table 5. SIMD 16-bit Compare Instructions

No.	Mnemonic	Instruction	Operation
1	CMPEQ16 rt, ra, rb	16-bit Compare Equal	$rt.H[x] = (ra.H[x] == rb.H[x])? 0xffff : 0;$ (RV32: $x=1..0$, RV64: $x=3..0$)
2	SCMPLT16 rt, ra, rb	16-bit Signed Compare Less Than	$rt.H[x] = (ra.H[x] < rb.H[x])? 0xffff : 0;$ (RV32: $x=1..0$, RV64: $x=3..0$)
3	SCMPLE16 rt, ra, rb	16-bit Signed Compare Less Than & Equal	$rt.H[x] = (ra.H[x] \leq rb.H[x])? 0xffff : 0;$ (RV32: $x=1..0$, RV64: $x=3..0$)
4	UCMPLT16 rt, ra, rb	16-bit Unsigned Compare Less Than	$rt.H[x] = (ra.H[x] u< rb.H[x])? 0xffff : 0;$ (RV32: $x=1..0$, RV64: $x=3..0$)
5	UCMPLE16 rt, ra, rb	16-bit Unsigned Compare Less Than & Equal	$rt.H[x] = (ra.H[x] u\leq rb.H[x])? 0xffff : 0;$ (RV32: $x=1..0$, RV64: $x=3..0$)

2.2.6. 8-bit Compare Instructions

There are 5 instructions here.

Table 6. SIMD 8-bit Compare Instructions

No.	Mnemonic	Instruction	Operation
1	CMPEQ8 rt, ra, rb	8-bit Compare Equal	$rt.B[x] = (ra.B[x] == rb.B[x])? 0xff : 0;$ (RV32: $x=3..0$, RV64: $x=7..0$)
2	SCMPLT8 rt, ra, rb	8-bit Signed Compare Less Than	$rt.B[x] = (ra.B[x] < rb.B[x])? 0xff : 0;$ (RV32: $x=3..0$, RV64: $x=7..0$)
3	SCMPLE8 rt, ra, rb	8-bit Signed Compare Less Than & Equal	$rt.B[x] = (ra.B[x] \leq rb.B[x])? 0xff : 0;$ (RV32: $x=3..0$, RV64: $x=7..0$)
4	UCMPLT8 rt, ra, rb	8-bit Unsigned Compare Less Than	$rt.B[x] = (ra.B[x] u< rb.B[x])? 0xff : 0;$ (RV32: $x=3..0$, RV64: $x=7..0$)
5	UCMPLE8 rt, ra, rb	8-bit Unsigned Compare Less Than & Equal	$rt.B[x] = (ra.B[x] u\leq rb.B[x])? 0xff : 0;$ (RV32: $x=3..0$, RV64: $x=7..0$)

2.2.7. 16-bit Multiply Instructions

There are 6 instructions here.

Table 7. SIMD 16-bit Multiply Instructions

No.	Mnemonic	Instruction	Operation
1	SMUL16 rt, ra, rb	16-bit Signed Multiply	RV32: $r[tL] = ra.H[0] s^* rb.H[0];$ $r[tH] = ra.H[1] s^* rb.H[1];$ RV64: $rt.W[0] = ra.H[0] s^* rb.H[0];$ $rt.W[1] = ra.H[1] s^* rb.H[1];$
2	SMULX16 rt, ra, rb	16-bit Signed Crossed Multiply	RV32: $r[tL] = ra.H[0] s^* rb.H[1];$ $r[tH] = ra.H[1] s^* rb.H[0];$ RV64: $rt.W[0] = ra.H[0] s^* rb.H[1];$ $rt.W[1] = ra.H[1] s^* rb.H[0];$
3	UMUL16 rt, ra, rb	16-bit Unsigned Multiply	RV32: $r[tL] = ra.H[0] u^* rb.H[0];$ $r[tH] = ra.H[1] u^* rb.H[1];$ RV64: $rt.W[0] = ra.H[0] u^* rb.H[0];$ $rt.W[1] = ra.H[1] u^* rb.H[1];$

No.	Mnemonic	Instruction	Operation
4	UMULX16 rt, ra, rb	16-bit Unsigned Crossed Multiply	<p>RV32:</p> $r[tL] = ra.H[0] \text{ u}^* rb.H[1];$ $r[tH] = ra.H[1] \text{ u}^* rb.H[0];$ <p>RV64:</p> $rt.W[0] = ra.H[0] \text{ u}^* rb.H[1];$ $rt.W[1] = ra.H[1] \text{ u}^* rb.H[0];$
5	KHM16 rt, ra, rb	Q15 Signed Saturating Multiply	$rt.H[x] = \text{SAT.Q15}((ra.H[x] \text{ s}^* rb.H[x]) \gg 15);$ <p>(RV32: $x=1..0$, RV64: $x=3..0$)</p>
6	KHMX16 rt, ra, rb	Q15 Signed Saturating Crossed Multiply	$rt.H[x] = \text{SAT.Q15}((ra.H[x] \text{ s}^* rb.H[y]) \gg 15);$ <p>(RV32: $(x,y)=(1,0), (0,1)$, RV64: $(x,y)=(3,2), (2,3), (1,0), (0,1)$)</p>

2.2.8. 8-bit Multiply Instructions

There are 6 instructions here.

Table 8. SIMD 8-bit Multiply Instructions

No.	Mnemonic	Instruction	Operation
1	SMUL8 rt, ra, rb	8-bit Signed Multiply	<p>RV32:</p> $r[tL].H[0] = ra.B[0] s^* rb.B[0];$ $r[tL].H[1] = ra.B[1] s^* rb.B[1];$ $r[tH].H[0] = ra.B[2] s^* rb.B[2];$ $r[tH].H[1] = ra.B[3] s^* rb.B[3];$ <p>RV64:</p> $rt.H[0] = ra.B[0] s^* rb.B[0];$ $rt.H[1] = ra.B[1] s^* rb.B[1];$ $rt.H[2] = ra.B[2] s^* rb.B[2];$ $rt.H[3] = ra.B[3] s^* rb.B[3];$
2	SMULX8 rt, ra, rb	8-bit Signed Crossed Multiply	<p>RV32:</p> $r[tL].H[0] = ra.B[0] s^* rb.B[1];$ $r[tL].H[1] = ra.B[1] s^* rb.B[0];$ $r[tH].H[0] = ra.B[2] s^* rb.B[3];$ $r[tH].H[1] = ra.B[3] s^* rb.B[2];$ <p>RV64:</p> $rt.H[0] = ra.B[0] s^* rb.B[1];$ $rt.H[1] = ra.B[1] s^* rb.B[0];$ $rt.H[2] = ra.B[2] s^* rb.B[3];$ $rt.H[3] = ra.B[3] s^* rb.B[2];$

No.	Mnemonic	Instruction	Operation
3	UMUL8 rt, ra, rb	8-bit Unsigned Multiply	<p>RV32:</p> $r[tL].H[0] = ra.B[0] u^* rb.B[0];$ $r[tL].H[1] = ra.B[1] u^* rb.B[1];$ $r[tH].H[0] = ra.B[2] u^* rb.B[2];$ $r[tH].H[1] = ra.B[3] u^* rb.B[3];$ <p>RV64:</p> $rt.H[0] = ra.B[0] u^* rb.B[0];$ $rt.H[1] = ra.B[1] u^* rb.B[1];$ $rt.H[2] = ra.B[2] u^* rb.B[2];$ $rt.H[3] = ra.B[3] u^* rb.B[3];$
4	UMULX8 rt, ra, rb	8-bit Unsigned Crossed Multiply	<p>RV32:</p> $r[tL].H[0] = ra.B[0] u^* rb.B[1];$ $r[tL].H[1] = ra.B[1] u^* rb.B[0];$ $r[tH].H[0] = ra.B[2] u^* rb.B[3];$ $r[tH].H[1] = ra.B[3] u^* rb.B[2];$ <p>RV64:</p> $rt.H[0] = ra.B[0] u^* rb.B[1];$ $rt.H[1] = ra.B[1] u^* rb.B[0];$ $rt.H[2] = ra.B[2] u^* rb.B[3];$ $rt.H[3] = ra.B[3] u^* rb.B[2];$
5	KHM8 rt, ra, rb	Q7 Signed Saturating Multiply	$rt.B[x] = SAT.Q7((ra.B[x] s^* rb.B[x]) >> 7);$ <p>(RV32: $x=3..0$, RV64: $x=7..0$)</p>
6	KHMX8 rt, ra, rb	Q7 Signed Saturating Crossed Multiply	$rt.B[x] = SAT.Q7((ra.B[x] s^* rb.B[y]) >> 7);$ <p>(RV32: $(x,y)=(3,2),(2,3),(1,0), (0,1)$, RV64: $(x,y)=(7,6),(6,7),(5,4), (4,5), (3,2), (2,3), (1,0), (0,1)$)</p>

2.2.9. 16-bit Misc Instructions

There are 11 instructions here.

Table 9. SIMD 16-bit Miscellaneous Instructions

No.	Mnemonic	Instruction	Operation
1	SMIN16 rt, ra, rb	16-bit Signed Minimum	$rt.H[x] = (ra.H[x] < rb.H[x])? ra.H[x] : rb.H[x];$ (RV32: $x=1..0$, RV64: $x=3..0$)
2	UMIN16 rt, ra, rb	16-bit Unsigned Minimum	$rt.H[x] = (ra.H[x] u< rb.H[x])? ra.H[x] : rb.H[x];$ (RV32: $x=1..0$, RV64: $x=3..0$)
3	SMAX16 rt, ra, rb	16-bit Signed Maximum	$rt.H[x] = (ra.H[x] > rb.H[x])? ra.H[x] : rb.H[x];$ (RV32: $x=1..0$, RV64: $x=3..0$)
4	UMAX16 rt, ra, rb	16-bit Unsigned Maximum	$rt.H[x] = (ra.H[x] u> rb.H[x])? ra.H[x] : rb.H[x];$ (RV32: $x=1..0$, RV64: $x=3..0$)
5	SCLIP16 rt, ra, imm4u	16-bit Signed Clip Value	$n = imm4u;$ $rt.H[x] = SAT.Qn(ra.H[x]);$ (RV32: $x=1..0$, RV64: $x=3..0$)
6	UCLIP16 rt, ra, imm4u	16-bit Unsigned Clip Value	$m = imm4u;$ $rt.H[x] = SAT.Um(ra.H[x]);$ (RV32: $x=1..0$, RV64: $x=3..0$)
7	KABS16 rt, ra	16-bit Absolute Value	$rt.H[x] = SAT.Q15(ABS(ra.H[x]));$ (RV32: $x=1..0$, RV64: $x=3..0$)
8	CLRS16 rt, ra	16-bit Count Leading Redundant Sign	
9	CLZ16 rt, ra	16-bit Count Leading Zero	
10	CLO16 rt, ra	16-bit Count Leading One	

2.2.10. 8-bit Misc Instructions

There are 11 instructions here.

Table 10. SIMD 8-bit Miscellaneous Instructions

No.	Mnemonic	Instruction	Operation
1	SMIN8 rt, ra, rb	8-bit Signed Minimum	$rt.B[x] = (ra.B[x] < rb.B[x])? ra.B[x] : rb.B[x];$ (RV32: $x=3..0$, RV64: $x=7..0$)
2	UMIN8 rt, ra, rb	8-bit Unsigned Minimum	$rt.B[x] = (ra.B[x] u< rb.B[x])? ra.B[x] : rb.B[x];$ (RV32: $x=3..0$, RV64: $x=7..0$)
3	SMAX8 rt, ra, rb	8-bit Signed Maximum	$rt.B[x] = (ra.B[x] > rb.B[x])? ra.B[x] : rb.B[x];$ (RV32: $x=3..0$, RV64: $x=7..0$)
4	UMAX8 rt, ra, rb	8-bit Unsigned Maximum	$rt.B[x] = (ra.B[x] u> rb.B[x])? ra.B[x] : rb.B[x];$ (RV32: $x=3..0$, RV64: $x=7..0$)
5	KABS8 rt, ra	8-bit Absolute Value	$rt.B[x] = SAT.Q7(ABS(ra.B[x]));$ (RV32: $x=3..0$, RV64: $x=7..0$)
6	SCLIP8 rt, ra, imm3u	8-bit Signed Clip Value	$n = imm3u;$ $rt.B[x] = SAT.Qn(ra.B[x]);$ (RV32: $x=3..0$, RV64: $x=7..0$)
7	UCLIP8 rt, ra, imm3u	8-bit Unsigned Clip Value	$m = imm3u;$ $rt.B[x] = SAT.Um(ra.B[x]);$ (RV32: $x=3..0$, RV64: $x=7..0$)
8	CLRS8 rt, ra	8-bit Count Leading Redundant Sign	
9	CLZ8 rt, ra	8-bit Count Leading Zero	
10	CLO8 rt, ra	8-bit Count Leading One	

2.2.11. 8-bit Unpacking Instructions

There are 8 instructions here.

Table 11. 8-bit Unpacking Instructions

No.	Mnemonic	Instruction	Operation
1	SUNPKD810 rt, ra	Signed Unpacking Bytes 1 & 0	$rt.H[x] = SE16(ra.B[y]);$ RV32: $(x,y) = (1,1), (0,0)$ RV64: $(x,y) = (3,5),(2,4),(1,1), (0,0)$
2	SUNPKD820 rt, ra	Signed Unpacking Bytes 2 & 0	$rt.H[x] = SE16(ra.B[y]);$ RV32: $(x,y) = (1,2), (0,0)$ RV64: $(x,y) = (3,6),(2,4),(1,2), (0,0)$
3	SUNPKD830 rt, ra	Signed Unpacking Bytes 3 & 0	$rt.H[x] = SE16(ra.B[y]);$ RV32: $(x,y) = (1,3), (0,0)$ RV64: $(x,y) = (3,7),(2,4),(1,3), (0,0)$
4	SUNPKD831 rt, ra	Signed Unpacking Bytes 3 & 1	$rt.H[x] = SE16(ra.B[y]);$ RV32: $(x,y) = (1,3), (0,1)$ RV64: $(x,y) = (3,7),(2,5),(1,3), (0,1)$
5	SUNPKD832 rt, ra	Signed Unpacking Bytes 3 & 2	$rt.H[x] = SE16(ra.B[y]);$ RV32: $(x,y) = (1,3), (0,2)$ RV64: $(x,y) = (3,7),(2,6),(1,3), (0,2)$
6	ZUNPKD810 rt, ra	Unsigned Unpacking Bytes 1 & 0	$rt.H[x] = ZE16(ra.B[y]);$ RV32: $(x,y) = (1,1), (0,0)$ RV64: $(x,y) = (3,5),(2,4),(1,1), (0,0)$
7	ZUNPKD820 rt, ra	Unsigned Unpacking Bytes 2 & 0	$rt.H[x] = ZE16(ra.B[y]);$ RV32: $(x,y) = (1,2), (0,0)$ RV64: $(x,y) = (3,6),(2,4),(1,2), (0,0)$

No.	Mnemonic	Instruction	Operation
8	ZUNPKD830 rt, ra	Unsigned Unpacking Bytes 3 & 0	rt.H[x] = ZE16(ra.B[y]); RV32: (x,y) = (1,3), (0,0) RV64: (x,y) = (3,7),(2,4),(1,3), (0,0)
9	ZUNPKD831 rt, ra	Unsigned Unpacking Bytes 3 & 1	rt.H[x] = ZE16(ra.B[y]); RV32: (x,y) = (1,3), (0,1) RV64: (x,y) = (3,7),(2,5),(1,3), (0,1)
10	ZUNPKD832 rt, ra	Unsigned Unpacking Bytes 3 & 2	rt.H[x] = ZE16(ra.B[y]); RV32: (x,y) = (1,3), (0,2) RV64: (x,y) = (3,7),(2,6),(1,3), (0,2)

2.3. Partial-SIMD Data Processing Instructions

2.3.1. 16-bit Packing Instructions

There are 4 instructions here.

Table 12. 16-bit Packing Instructions

No.	Mnemonic	Instruction	Operation
1	PKBB16 rt, ra, rb	Pack two 16-bit data from Bottoms	$rt.W[x] = \text{CONCAT}(ra.W[x].H[0], rb.W[x].H[0]);$ (RV32: $x=0$, RV64: $x=1..0$)
2	PKBT16 rt, ra, rb	Pack two 16-bit data Bottom & Top	$rt.W[x] = \text{CONCAT}(ra.W[x].H[0], rb.W[x].H[1]);$ (RV32: $x=0$, RV64: $x=1..0$)
3	PKTB16 rt, ra, rb	Pack two 16-bit data Top & Bottom	$rt.W[x] = \text{CONCAT}(ra.W[x].H[1], rb.W[x].H[0]);$ (RV32: $x=0$, RV64: $x=1..0$)
4	PKTT16 rt, ra, rb	Pack two 16-bit data from Tops	$rt.W[x] = \text{CONCAT}(ra.W[x].H[1], rb.W[x].H[0]);$ (RV32: $x=0$, RV64: $x=1..0$)

2.3.2. Most Significant Word “32x32” Multiply & Add Instructions

There are 8 instructions here.

Table 13. Signed MSW 32x32 Multiply and Add Instructions

No.	Mnemonic	Instruction	Operation
1	SMMUL rt, ra, rb	MSW “32 x 32” Signed Multiplication (MSW 32 = 32x32)	$rt.W[x] = (ra.W[x]*rb.W[x])[63:32];$ (RV32: $x=0$, RV64: $x=1..0$)
2	SMMUL.u rt, ra, rb	MSW “32 x 32” Signed Multiplication with Rounding (MSW 32 = 32x32)	$rt.W[x] = \text{RUND}(ra.W[x]*rb.W[x])[63:32];$ (RV32: $x=0$, RV64: $x=1..0$)
3	KMMAC rt, ra, rb	MSW “32 x 32” Signed Multiplication and Saturating Addition (MSW 32 = 32 + 32x32)	$rt.W[x] = \text{SAT.Q31}(rt.W[x] + (ra.W[x]*rb.W[x])[63:32]);$ (RV32: $x=0$, RV64: $x=1..0$)
4	KMMAC.u rt, ra, rb	MSW “32 x 32” Signed Multiplication and Saturating Addition with Rounding (MSW 32 = 32 + 32x32)	$rt.W[x] = \text{SAT.Q31}(rt.W[x] + \text{RUND}(ra.W[x]*rb.W[x])[63:32]);$ (RV32: $x=0$, RV64: $x=1..0$)
5	KMMSB rt, ra, rb	MSW “32 x 32” Signed Multiplication and Saturating Subtraction (MSW 32 = 32 - 32x32)	$rt.W[x] = \text{SAT.Q31}(rt.W[x] - (ra.W[x]*rb.W[x])[63:32]);$ (RV32: $x=0$, RV64: $x=1..0$)
6	KMMSB.u rt, ra, rb	MSW “32 x 32” Signed Multiplication and Saturating Subtraction with Rounding (MSW 32 = 32 - 32x32)	$rt.W[x] = \text{SAT.Q31}(rt.W[x] - \text{RUND}(ra.W[x]*rb.W[x])[63:32]);$ (RV32: $x=0$, RV64: $x=1..0$)

No.	Mnemonic	Instruction	Operation
7	KWMMUL rt, ra, rb	MSW “32 x 32” Signed Multiplication & Double (MSW 32 = 32x32 << 1)	rt.W[x] = SAT.Q31((ra.W[x]*rb.W[x] << 1)[63:32]); (RV32: x=0, RV64: x=1..0)
8	KWMMUL.u rt, ra, rb	MSW “32 x 32” Signed Multiplication & Double with Rounding (MSW 32 = 32x32 << 1)	rt.W[x] = SAT.Q31(RUND(ra.W[x]*rb.W[x] << 1)[63:32]); (RV32: x=0, RV64: x=1..0)

2.3.3. Most Significant Word “32x16” Multiply & Add Instructions

There are 16 instructions here.

Table 14. Signed MSW 32x16 Multiply and Add Instructions

No.	Mnemonic	Instruction	Operation
1	SMMWB rt, ra, rb	MSW “32 x Bottom 16” Signed Multiplication (MSW 32 = 32x16)	$rt.W[x] = (ra.W[x]*rb.W[x].H[0])[47:16];$ (RV32: $x=0$, RV64: $x=1..0$)
2	SMMWB.u rt, ra, rb	MSW “32 x Bottom 16” Signed Multiplication with Rounding (MSW 32 = 32x16)	$rt.W[x] = RUND(ra.W[x]*rb.W[x].H[0])[47:16];$ (RV32: $x=0$, RV64: $x=1..0$)
3	SMMWT rt, ra, rb	MSW “32 x Top 16” Signed Multiplication (MSW 32 = 32x16)	$rt.W[x] = (ra.W[x]*rb.W[x].H[1])[47:16];$ (RV32: $x=0$, RV64: $x=1..0$)
4	SMMWT.u rt, ra, rb	MSW “32 x Top 16” Signed Multiplication with Rounding (MSW 32 = 32x16)	$rt.W[x] = RUND(ra.W[x]*rb.W[x].H[1])[47:16];$ (RV32: $x=0$, RV64: $x=1..0$)
5	KMMAWB rt, ra, rb	MSW “32 x Bottom 16” Signed Multiplication and Saturating Addition (MSW 32 = 32 + 32x16)	$rt.W[x] = SAT.Q31(rt.W[x] + (ra.W[x]*rb.W[x].H[0])[47:16]);$ (RV32: $x=0$, RV64: $x=1..0$)
6	KMMAWB.u rt, ra, rb	MSW “32 x Bottom 16” Signed Multiplication and Saturating Addition with Rounding (MSW 32 = 32 + 32x16)	$rt.W[x] = SAT.Q31(rt.W[x] + RUND(ra.W[x]*rb.W[x].H[0])[47:16]);$ (RV32: $x=0$, RV64: $x=1..0$)
7	KMMAWT rt, ra, rb	MSW “32 x Top 16” Signed Multiplication and Saturating Addition (MSW 32 = 32 + 32x16)	$rt.W[x] = SAT.Q31(rt.W[x] + (ra.W[x]*rb.W[x].H[1])[47:16]);$ (RV32: $x=0$, RV64: $x=1..0$)

No.	Mnemonic	Instruction	Operation
8	KMMAWT.u rt, ra, rb	MSW “32 x Top 16” Signed Multiplication and Saturating Addition with Rounding (MSW 32 = 32 + 32x16)	$\text{rt.W}[x] = \text{SAT.Q31}(\text{rt.W}[x]) + \text{RUND}(\text{ra.W}[x]*\text{rb.W}[x].H[1])[47:16];$ (RV32: $x=0$, RV64: $x=1..0$)
9	KMMWB2 rt, ra, rb	MSW “32 x Bottom 16” Saturating Signed Multiplication and double (MSW 32 = (32x16) $\ll 1$)	$\begin{aligned} \text{a}[x] &= \text{ra.W}[x]; \text{b}[x] = \text{rb.W}[x]; \\ \text{if } \text{a}[x] == 0x80000000 \& (\text{b}[x].L == 0x8000) \{ \\ \text{t}[x] &= 0x7fffffff; \text{OV} = 1; \} \text{ else } \{ \\ \text{t}[x] &= ((\text{a}[x]*\text{b}[x].L)\ll 1)[47:16]; \\ \} \\ \text{rt.W}[x] &= \text{t}[x]; \\ \end{aligned}$ (RV32: $x=0$, RV64: $x=1..0$)
10	KMMWB2.u rt, ra, rb	MSW “32 x Bottom 16” Saturating Signed Multiplication and double with Rounding (MSW 32 = (32x16) $\ll 1$)	$\begin{aligned} \text{a}[x] &= \text{ra.W}[x]; \text{b}[x] = \text{rb.W}[x]; \\ \text{if } \text{a}[x] == 0x80000000 \& (\text{b}[x].L == 0x8000) \{ \\ \text{t}[x] &= 0x7fffffff; \text{OV} = 1; \} \text{ else } \{ \\ \text{t}[x] &= \text{RUND}((\text{a}[x]*\text{b}[x].L)\ll 1)[47:16]; \\ \} \\ \text{rt.W}[x] &= \text{t}[x]; \\ \end{aligned}$ (RV32: $x=0$, RV64: $x=1..0$)
11	KMMWT2 rt, ra, rb	MSW “32 x Top 16” Saturating Signed Multiplication and double (MSW 32 = (32x16) $\ll 1$)	$\begin{aligned} \text{a}[x] &= \text{ra.W}[x]; \text{b}[x] = \text{rb.W}[x]; \\ \text{if } \text{a}[x] == 0x80000000 \& (\text{b}[x].H == 0x8000) \{ \\ \text{t}[x] &= 0x7fffffff; \text{OV} = 1; \} \text{ else } \{ \\ \text{t}[x] &= ((\text{a}[x]*\text{b}[x].H)\ll 1)[47:16]; \\ \} \\ \text{rt.W}[x] &= \text{t}[x]; \\ \end{aligned}$ (RV32: $x=0$, RV64: $x=1..0$)

No.	Mnemonic	Instruction	Operation
12	KMMWT2.u rt, ra, rb	MSW “32 x Top 16” Saturating Signed Multiplication and double with Rounding (MSW 32 = (32x16) << 1)	$a[x] = ra.W[x]; b[x] = rb.W[x];$ $\text{if } a[x] == 0x80000000 \& (b[x].H == 0x8000 \{$ $t[x] = 0x7fffffff; OV = 1; \} \text{ else } \{$ $t[x] = \text{RUND}((a[x]*b[x].H)<<1)[47:16];$ $\}$ $rt.W[x] = t[x];$ $(\text{RV32: } x=0, \text{ RV64: } x=1..0)$
13	KMMAWB2 rt, ra, rb	MSW “32 x Bottom 16” Signed Multiplication & double and Saturating Addition (MSW 32 = 32 + (32x16)<<1)	$a[x] = ra.W[x]; b[x] = rb.W[x];$ $\text{if } a[x] == 0x80000000 \& (b[x].L == 0x8000 \{$ $t[x] = 0x7fffffff; OV = 1; \} \text{ else } \{$ $t[x] = ((a[x]*b[x].L)<<1)[47:16];$ $\}$ $rt.W[x] = \text{SAT.Q31}(rt.W[x] + t[x]);$ $(\text{RV32: } x=0, \text{ RV64: } x=1..0)$
14	KMMAWB2.u rt, ra, rb	MSW “32 x Bottom 16” Signed Multiplication & double and Saturating Addition with Rounding (MSW 32 = 32 + (32x16)<<1)	$a[x] = ra.W[x]; b[x] = rb.W[x];$ $\text{if } a[x] == 0x80000000 \& (b[x].L == 0x8000 \{$ $t[x] = 0x7fffffff; OV = 1; \} \text{ else } \{$ $t[x] = \text{RUND}((a[x]*b[x].L)<<1)[47:16];$ $\}$ $rt.W[x] = \text{SAT.Q31}(rt.W[x] + t[x]);$ $(\text{RV32: } x=0, \text{ RV64: } x=1..0)$

No.	Mnemonic	Instruction	Operation
15	KMMAWT2 rt, ra, rb	MSW “32 x Top 16” Signed Multiplication & double and Saturating Addition (MSW 32 = 32 + (32x16)<<1)	a[x]=ra.W[x]; b[x]=rb.W[x]; if a[x]==0x80000000) & (b[x].H==0x8000 \{ t[x]=0x7fffffff; OV=1; } else \{ t[x]= ((a[x]*b[x].H)<<1)[47:16]; } rt.W[x] = SAT.Q31(rt.W[x] + t[x]); (RV32: x=0, RV64: x=1..0)
	KMMAWT2.u rt, ra, rb	MSW “32 x Top 16” Signed Multiplication & double and Saturating Addition with Rounding (MSW 32 = 32 + (32x16)<<1)	a[x]=ra.W[x]; b[x]=rb.W[x]; if a[x]==0x80000000) & (b[x].H==0x8000 \{ t[x]=0x7fffffff; OV=1; } else \{ t[x]= RUND((a[x]*b[x].H)<<1)[47:16]; } rt.W[x] = SAT.Q31(rt.W[x] + t[x]); (RV32: x=0, RV64: x=1..0)

2.3.4. Signed 16-bit Multiply with 32-bit Add/Subtract Instructions

There are 18 instructions here.

Table 15. Signed 16-bit Multiply 32-bit Add/Subtract Instructions

No.	Mnemonic	Instruction	Operation
1	SMBB16 rt, ra, rb	Signed Multiply Bottom 16 & Bottom 16 (32 = 16x16)	$rt.W[x] = ra.W[x].H[0]*rb.W[x].H[0];$ (RV32: $x=0$, RV64: $x=1..0$)
2	SMBT16 rt, ra, rb	Signed Multiply Bottom 16 & Top 16 (32 = 16x16)	$rt.W[x] = ra.W[x].H[0]*rb.W[x].H[1];$ (RV32: $x=0$, RV64: $x=1..0$)
3	SMTT16 rt, ra, rb	Signed Multiply Top 16 & Top 16 (32 = 16x16)	$rt.W[x] = ra.W[x].H[1]*rb.W[x].H[1];$ (RV32: $x=0$, RV64: $x=1..0$)
4	KMDA rt, ra, rb	Two “16x16” and Signed Addition (32 = 16x16 + 16x16)	$rt.W[x] = SAT.Q31(ra.W[x].H[1]*rb.W[x].H[1] + ra.W[x].H[0]*rb.W[x].H[0]);$ (RV32: $x=0$, RV64: $x=1..0$)
5	KMXDA rt, ra, rb	Two Crossed “16x16” and Signed Addition (32 = 16x16 + 16x16)	$rt.W[x] = SAT.Q31(ra.W[x].H[1]*rb.W[x].H[0] + ra.W[x].H[0]*rb.W[x].H[1]);$ (RV32: $x=0$, RV64: $x=1..0$)
6	SMDS rt, ra, rb	Two “16x16” and Signed Subtraction (32 = 16x16 - 16x16)	$rt.W[x] = (ra.W[x].H[1]*rb.W[x].H[1]) - (ra.W[x].H[0]*rb.W[x].H[0]);$ (RV32: $x=0$, RV64: $x=1..0$)
7	SMDRS rt, ra, rb	Two “16x16” and Signed Reversed Subtraction (32 = 16x16 - 16x16)	$rt.W[x] = (ra.W[x].H[0]*rb.W[x].H[0]) - (ra.W[x].H[1]*rb.W[x].H[1]);$ (RV32: $x=0$, RV64: $x=1..0$)
8	SMXDS rt, ra, rb	Two Crossed “16x16” and Signed Subtraction (32 = 16x16 - 16x16)	$rt.W[x] = (ra.W[x].H[1]*rb.W[x].H[0]) - (ra.W[x].H[0]*rb.W[x].H[1]);$ (RV32: $x=0$, RV64: $x=1..0$)
9	KMABB rt, ra, rb	“Bottom 16 x Bottom 16” with 32-bit Signed Addition (32 = 32 + 16x16)	$rt.W[x] = SAT.Q31(rt.W[x] + ra.W[x].H[0]*rb.W[x].H[0]);$ (RV32: $x=0$, RV64: $x=1..0$)

No.	Mnemonic	Instruction	Operation
10	KMABT rt, ra, rb	“Bottom 16 x Top 16” with 32-bit Signed Addition ($32 = 32 + 16 \times 16$)	$rt.W[x] = SAT.Q31(rt.W[x] + ra.W[x].H[0]*rb.W[x].H[1]);$ (RV32: $x=0$, RV64: $x=1..0$)
11	KMATT rt, ra, rb	“Top 16 x Top 16” with 32-bit Signed Addition ($32 = 32 + 16 \times 16$)	$rt.W[x] = SAT.Q31(rt.W[x] + ra.W[x].H[1]*rb.W[x].H[1]);$ (RV32: $x=0$, RV64: $x=1..0$)
12	KMADA rt, ra, rb	Two “16x16” with 32-bit Signed Double Addition ($32 = 32 + 16 \times 16 + 16 \times 16$)	$rt.W[x] = SAT.Q31(rt.W[x] + ra.W[x].H[1]*rb.W[x].H[1] + ra.W[x].H[0]*rb.W[x].H[0]);$ (RV32: $x=0$, RV64: $x=1..0$)
13	KMAXDA rt, ra, rb	Two Crossed “16x16” with 32-bit Signed Double Addition ($32 = 32 + 16 \times 16 + 16 \times 16$)	$rt.W[x] = SAT.Q31(rt.W[x] + ra.W[x].H[1]*rb.W[x].H[0] + ra.W[x].H[0]*rb.W[x].H[1]);$ (RV32: $x=0$, RV64: $x=1..0$)
14	KMADS rt, ra, rb	Two “16x16” with 32-bit Signed Addition and Subtraction ($32 = 32 + 16 \times 16 - 16 \times 16$)	$rt.W[x] = SAT.Q31(rt.W[x] + ra.W[x].H[1]*rb.W[x].H[1] - ra.W[x].H[0]*rb.W[x].H[0]);$ (RV32: $x=0$, RV64: $x=1..0$)
15	KMADRS rt, ra, rb	Two “16x16” with 32-bit Signed Addition and Reversed Subtraction ($32 = 32 + 16 \times 16 - 16 \times 16$)	$rt.W[x] = SAT.Q31(rt.W[x] + ra.W[x].H[0]*rb.W[x].H[0] - ra.W[x].H[1]*rb.W[x].H[1]);$ (RV32: $x=0$, RV64: $x=1..0$)
16	KMAXDS rt, ra, rb	Two Crossed “16x16” with 32-bit Signed Addition and Subtraction ($32 = 32 + 16 \times 16 - 16 \times 16$)	$rt.W[x] = SAT.Q31(rt.W[x] + ra.W[x].H[1]*rb.W[x].H[0] - ra.W[x].H[0]*rb.W[x].H[1]);$ (RV32: $x=0$, RV64: $x=1..0$)

No.	Mnemonic	Instruction	Operation
17	KMSDA rt, ra, rb	Two “16x16” with 32-bit Signed Double Subtraction $(32 = 32 - 16x16 - 16x16)$	$rt.W[x] = SAT.Q31(rt.W[x] - ra.W[x].H[1]*rb.W[x].H[1] - ra.W[x].H[0]*rb.W[x].H[0]);$ (RV32: $x=0$, RV64: $x=1..0$)
18	KMSXDA rt, ra, rb	Two Crossed “16x16” with 32-bit Signed Double Subtraction $(32 = 32 - 16x16 - 16x16)$	$rt.W[x] = SAT.Q31(rt.W[x] - ra.W[x].H[1]*rb.W[x].H[0] - ra.W[x].H[0]*rb.W[x].H[1]);$ (RV32: $x=0$, RV64: $x=1..0$)

2.3.5. Signed 16-bit Multiply with 64-bit Add/Subtract Instructions

Table 16. Signed 16-bit Multiply 64-bit Add/Subtract Instructions

No.	Mnemonic	Instruction	Operation
1	SMAL rt, ra, rb	“16 x 16” with 64-bit Signed Addition ($64 = 64 + 16 \times 16$)	<p>RV32:</p> $a64 = r[aU].r[aL];$ $t64 = a64 + rb.W[0].H[1]*rb.W[0].H[0];$ $r[tU].r[tL] = t64;$ <p>RV64:</p> $a64 = ra;$ $rt = a64 + rb.W[1].H[1]*rb.W[1].H[0] +$ $rb.W[0].H[1]*rb.W[0].H[0];$

2.3.6. Miscellaneous Instructions

There are 7 instructions here.

Table 17. Partial-SIMD Miscellaneous Instructions

No.	Mnemonic	Instruction	Operation
1	SCLIP32 rt, ra, imm5u	Signed Clip Value	$n = \text{imm5u};$ $rt = \text{SAT.Qn}(\text{ra.W}[x]);$ (RV32: $x=0$, RV64: $x=1..0$)
2	UCLIP32 rt, ra, imm5u	Unsigned Clip Value	$m = \text{imm5u};$ $rt = \text{SAT.Um}(\text{ra.W}[x]);$ (RV32: $x=0$, RV64: $x=1..0$)
3	CLRS32 rt, ra	32-bit Count Leading Redundant Sign	$\text{rt.W}[x] = \text{CLRS}(\text{ra.W}[x])$ (RV32: $x=0$, RV64: $x=1..0$)
4	CLZ32 rt, ra	32-bit Count Leading Zero	$\text{rt.W}[x] = \text{CLZ}(\text{ra.W}[x])$ (RV32: $x=0$, RV64: $x=1..0$)
5	CLO32 rt, ra	32-bit Count Leading One	$\text{rt.W}[x] = \text{CLO}(\text{ra.W}[x])$ (RV32: $x=0$, RV64: $x=1..0$)
6	PBSAD rt, ra, rb	Parallel Byte Sum of Absolute Difference	$\text{absdiff}[x] = \text{ABS}(\text{ra.B}[x] - \text{rb.B}[x]);$ $rt = \text{SUM}(\text{absdiff}[x]);$ (RV32: $x=3..0$, RV64: $x=7..0$)
7	PBSADA rt, ra, rb	Parallel Byte Sum of Absolute Difference Accumulation	$\text{absdiff}[x] = \text{ABS}(\text{ra.B}[x] - \text{rb.B}[x]);$ $rt = rt + \text{SUM}(\text{absdiff}[x]);$ (RV32: $x=3..0$, RV64: $x=7..0$)

2.3.7. 8-bit Multiply with 32-bit Add Instructions

There are 3 instructions here.

Table 18. 8-bit Multiply with 32-bit Add Instructions

No.	Mnemonic	Instruction	Operation
1	SMAQA rt, ra, rb	Four signed “8x8” with 32-bit Signed Addition $(32 = 32 + 8x8 + 8x8 + 8x8 + 8x8)$	$rt.W[x] = rt.W[x] + ra.W[x].B[3]*rb.W[x].B[3] + ra.W[x].B[2]*rb.W[x].B[2] + ra.W[x].B[1]*rb.W[x].B[1] + ra.W[x].B[0]*rb.W[x].B[0]);$ (RV32: x=0, RV64: x=1..0) Elements of ra and rb are signed numbers.
2	UMAQQA rt, ra, rb	Four unsigned “8x8” with 32-bit Unsigned Addition $(32 = 32 + 8x8 + 8x8 + 8x8 + 8x8)$	$rt.W[x] = rt.W[x] + ra.W[x].B[3]*rb.W[x].B[3] + ra.W[x].B[2]*rb.W[x].B[2] + ra.W[x].B[1]*rb.W[x].B[1] + ra.W[x].B[0]*rb.W[x].B[0]);$ (RV32: x=0, RV64: x=1..0) Elements of ra and rb are unsigned numbers.
3	SMAQA.SU rt, ra, rb	Four “signed 8 x unsigned 8” with 32-bit Signed Addition $(32 = 32 + 8x8 + 8x8 + 8x8 + 8x8)$	$rt.W[x] = rt.W[x] + ra.W[x].B[3]*rb.W[x].B[3] + ra.W[x].B[2]*rb.W[x].B[2] + ra.W[x].B[1]*rb.W[x].B[1] + ra.W[x].B[0]*rb.W[x].B[0]);$ (RV32: x=0, RV64: x=1..0) Elements of ra are signed numbers. Elements of rb are unsigned numbers.

2.4. 64-bit Profile Instructions

2.4.1. 64-bit Addition & Subtraction Instructions

Table 19. 64-bit Add/Subtract Instructions

No.	Mnemonic	Instruction	Operation
1	ADD64 rt, ra, rb	64-bit Addition	$a64 = r[aU].r[aL]; b64 = r[bU].r[bL];$ $t64 = a64 + b64;$ $r[tU].r[tL] = t64;$
2	RADD64 rt, ra, rb	64-bit Signed Halving Addition	$a64 = r[aU].r[aL]; b64 = r[bU].r[bL];$ $t64 = (a64 + b64) s>>1;$ $r[tU].r[tL] = t64;$
3	URADD64 rt, ra, rb	64-bit Unsigned Halving Addition	$a64 = r[aU].r[aL]; b64 = r[bU].r[bL];$ $t64 = (a64 + b64) u>>1;$ $r[tU].r[tL] = t64;$
4	KADD64 rt, ra, rb	64-bit Signed Saturating Addition	$a64 = r[aU].r[aL]; b64 = r[bU].r[bL];$ $t64 = SAT.Q63(a64 + b64);$ $r[tU].r[tL] = t64;$
5	UKADD64 rt, ra, rb	64-bit Unsigned Saturating Addition	$a64 = r[aU].r[aL]; b64 = r[bU].r[bL];$ $t64 = SAT.U64(a64 + b64);$ $r[tU].r[tL] = t64;$
6	SUB64 rt, ra, rb	64-bit Subtraction	$a64 = r[aU].r[aL]; b64 = r[bU].r[bL];$ $t64 = a64 - b64;$ $r[tU].r[tL] = t64;$
7	RSUB64 rt, ra, rb	64-bit Signed Halving Subtraction	$a64 = r[aU].r[aL]; b64 = r[bU].r[bL];$ $t64 = (a64 - b64) s>>1;$ $r[tU].r[tL] = t64;$

No.	Mnemonic	Instruction	Operation
8	URSUB64 rt, ra, rb	64-bit Unsigned Halving Subtraction	$a64 = r[aU].r[aL]; b64 = r[bU].r[bL];$ $t64 = (a64 - b64) \text{ u}>>1;$ $r[tU].r[tL] = t64;$
9	KSUB64 rt, ra, rb	64-bit Signed Saturating Subtraction	$a64 = r[aU].r[aL]; b64 = r[bU].r[bL];$ $t64 = \text{SAT.Q63}(a64 - b64);$ $r[tU].r[tL] = t64;$
10	UKSUB64 rt, ra, rb	64-bit Unsigned Saturating Subtraction	$a64 = r[aU].r[aL]; b64 = r[bU].r[bL];$ $t64 = \text{SAT.U64}(a64 - b64);$ $r[tU].r[tL] = t64;$

2.4.2. 32-bit Multiply with 64-bit Add/Subtract Instructions

Table 20. 32-bit Multiply 64-bit Add/Subtract Instructions

No.	Mnemonic	Instruction	Operation
1	SMAR64 rt, ra, rb	32x32 with 64-bit Signed Addition	$c64 = r[tU].r[tL];$ $t64 = c64 + ra*rb; // signed$ $r[tU].r[tL] = t64;$
2	SMSR64 rt, ra, rb	32x32 with 64-bit Signed Subtraction	$c64 = r[tU].r[tL];$ $t64 = c64 - ra*rb; // signed$ $r[tU].r[tL] = t64;$
3	UMAR64 rt, ra, rb	32x32 with 64-bit Unsigned Addition	$c64 = r[tU].r[tL];$ $t64 = c64 + ra*rb; // unsigned$ $r[tU].r[tL] = t64;$
4	UMSR64 rt, ra, rb	32x32 with 64-bit Unsigned Subtraction	$c64 = r[tU].r[tL];$ $t64 = c64 - ra*rb; // unsigned$ $r[tU].r[tL] = t64;$
5	KMAR64 rt, ra, rb	32x32 with Saturating 64-bit Signed Addition	$c64 = r[tU].r[tL];$ $t64 = SAT.Q63(c64 + ra*rb);$ $r[tU].r[tL] = t64;$
6	KMSR64 rt, ra, rb	32x32 with Saturating 64-bit Signed Subtraction	$c64 = r[tU].r[tL];$ $t64 = SAT.Q63(c64 - ra*rb);$ $r[tU].r[tL] = t64;$
7	UKMAR64 rt, ra, rb	32x32 with Saturating 64-bit Unsigned Addition	$c64 = r[tU].r[tL];$ $t64 = SAT.U64(c64 + ra*rb);$ $r[tU].r[tL] = t64;$
8	UKMSR64 rt, ra, rb	32x32 with Saturating 64-bit Unsigned Subtraction	$c64 = r[tU].r[tL];$ $t64 = SAT.U64(c64 - ra*rb);$ $r[tU].r[tL] = t64;$

2.4.3. Signed 16-bit Multiply with 64-bit Add/Subtract Instructions

Table 21. Signed 16-bit Multiply 64-bit Add/Subtract Instructions

No.	Mnemonic	Instruction	Operation
1	SMALBB rt, ra, rb	“Bottom 16 x Bottom 16” with 64-bit Signed Addition ($64 = 64 + 16 \times 16$)	$c64 = r[tU].r[tL];$ $t64 = c64 + ra.L * rb.L;$ $r[tU].r[tL] = t64;$
2	SMALBT rt, ra, rb	“Bottom 16 x Top 16” with 64-bit Signed Addition ($64 = 64 + 16 \times 16$)	$c64 = r[tU].r[tL];$ $t64 = c64 + ra.L * rb.H;$ $r[tU].r[tL] = t64;$
3	SMALTT rt, ra, rb	“Top 16 x Top 16” with 64-bit Signed Addition ($64 = 64 + 16 \times 16$)	$c64 = r[tU].r[tL];$ $t64 = c64 + ra.H * rb.H;$ $r[tU].r[tL] = t64;$
4	SMALDA rt, ra, rb	Two “16x16” with 64-bit Signed Double Addition ($64 = 64 + 16 \times 16 + 16 \times 16$)	$c64 = r[tU].r[tL];$ $t64 = c64 + ra.H * rb.H + ra.L * rb.L;$ $r[tU].r[tL] = t64;$
5	SMALXDA rt, ra, rb	Two Crossed “16x16” with 64-bit Signed Double Addition ($64 = 64 + 16 \times 16 + 16 \times 16$)	$c64 = r[tU].r[tL];$ $t64 = c64 + ra.H * rb.L + ra.L * rb.H;$ $r[tU].r[tL] = t64;$
6	SMALDS rt, ra, rb	Two “16x16” with 64-bit Signed Addition and Subtraction ($64 = 64 + 16 \times 16 - 16 \times 16$)	$c64 = r[tU].r[tL];$ $t64 = c64 + ra.H * rb.H - ra.L * rb.L;$ $r[tU].r[tL] = t64;$
7	SMALDRS rt, ra, rb	Two “16x16” with 64-bit Signed Addition and Reversed Subtraction ($64 = 64 + 16 \times 16 - 16 \times 16$)	$c64 = r[tU].r[tL];$ $t64 = c64 + ra.L * rb.L - ra.H * rb.H;$ $r[tU].r[tL] = t64;$

No.	Mnemonic	Instruction	Operation
8	SMALXDS rt, ra, rb	Two Crossed “16x16” with 64-bit Signed Addition and Subtraction $(64 = 64 + 16x16 - 16x16)$	$c64 = r[tU].r[tL];$ $t64 = c64 + ra.H*rb.L - ra.L*rb.H;$ $r[tU].r[tL] = t64;$
9	SMSLDA rt, ra, rb	Two “16x16” with 64-bit Signed Double Subtraction $(64 = 64 - 16x16 - 16x16)$	$c64 = r[tU].r[tL];$ $t64 = c64 - ra.H*rb.H - ra.L*rb.L;$ $r[tU].r[tL] = t64;$
10	SMSLXDA rt, ra, rb	Two Crossed “16x16” with 64-bit Signed Double Subtraction $(64 = 64 - 16x16 - 16x16)$	$c64 = r[tU].r[tL];$ $t64 = c64 - ra.H*rb.L - ra.L*rb.H;$ $r[tU].r[tL] = t64;$

2.5. Non-SIMD Instructions

2.5.1. Q15 saturation instructions

The following table lists non-SIMD instructions related to Q15 arithmetic.

Table 22. Non-SIMD Q15 saturation ALU Instructions

No.	Mnemonic	Instruction	Operation
1	KADDH Rt, Ra, Rb	Add with Q15 saturation	$Rt = SE(SAT.Q15(Ra.W[0] + Rb.W[0]))$
2	KSUBH Rt, Ra, Rb	Subtract with Q15 saturation	$Rt = SE(SAT.Q15(Ra.W[0] - Rb.W[0]))$
3	KHMBB Rt, Ra, Rb	Multiply the first 16-bit Q15 elements of two registers and transform the Q30 result into a saturated Q15 number.	$Rt = SAT.Q15((Ra.H0 * Rb.H0) s>> 15)$
4	KHMBT Rt, Ra, Rb	Multiply the first 16-bit Q15 element of one register with the second 16-bit Q15 element of another register and transform the Q30 result into a saturated Q15 number.	$Rt = SAT.Q15((Ra.H0 * Rb.H1) s>> 15)$
5	KHM TT Rt, Ra, Rb	Multiply the second 16-bit Q15 elements of two registers and transform the Q30 result into a saturated Q15 number.	$Rt = SAT.Q15((Ra.H1 * Rb.H1) s>> 15)$
6	UKADDH Rt, Ra, Rb	Add with I16 saturation	$Rt = ZE(SAT.U16(Ra.W[0] + Rb.W[0]))$
7	UKSUBH Rt, Ra, Rb	Subtract with I16 saturation	$Rt = ZE(SAT.U16(Ra.W[0] - Rb.W[0]))$

2.5.2. Q31 saturation Instructions

The following table lists non-SIMD instructions related to Q31 arithmetic.

Table 23. Non-SIMD Q31 saturation ALU Instructions

No.	Mnemonic	Instruction	Operation
1	KADDW Rt, Ra, Rb	Add with Q31 saturation	$Rt = SAT.Q31(Ra + Rb) // RV32$ $Rt = SE(SAT.Q31(Ra.W[0] + Rb.W[0])) // RV64$
2	UKADDW Rt, Ra, Rb	Unsigned Add with I32 saturation	$Rt = SAT.I32(Ra + Rb) // RV32$ $Rt = ZE(SAT.U32(Ra.W[0] + Rb.W[0])) // RV64$
3	KSUBW Rt, Ra, Rb	Subtract with Q31 saturation	$Rt = SAT.Q31(Ra - Rb) // RV32$ $Rt = SE(SAT.Q31(Ra.W[0] - Rb.W[0])) // RV64$
4	UKSUBW Rt, Ra, Rb	Unsigned Subtract with I32 saturation	$Rt = SAT.I32(Ra - Rb) // RV32$ $Rt = ZE(SAT.U32(Ra.W[0] - Rb.W[0])) // RV64$
5	KDMBB Rt, Ra, Rb	Multiply the first 16-bit Q15 elements of two registers and transform the Q30 result into a saturated Q31 number.	$Rt = SAT.Q31((Ra.H0 * Rb.H0) << 1)$
6	KDMBT Rt, Ra, Rb	Multiply the first 16-bit Q15 element of one register with the second 16-bit Q15 element of another register and transform the Q30 result into a saturated Q31 number.	$Rt = SAT.Q31((Ra.H0 * Rb.H1) << 1)$
7	KDMTT Rt, Ra, Rb	Multiply the second 16-bit Q15 elements of two registers and transform the Q30 result into a saturated Q31 number.	$Rt = SAT.Q31((Ra.H1 * Rb.H1) << 1)$

No.	Mnemonic	Instruction	Operation
8	KSLRAW Rt, Ra, Rb	Shift Left Logical with Q31 Saturation or Shift Right Arithmetic	(Rb[5:0] >=0) ? Rt = SAT.Q31(Ra << Rb[5:0]): Rt = (Ra s>> -Rb[5:0])
9	KSLRAW.u Rt, Ra, Rb	Shift Left Logical with Q31 Saturation or Rounding Shift Right Arithmetic	if (rb[5:0] < 0) rt = RUND(ra s>> -rb[5:0]); if (rb[5:0] > 0) rt = SAT.Q31(ra << rb[5:0]);
10	KSLLW rt, ra, rb	Saturating Shift Left Logical for 32-bit Word	rt = SE(SAT.Q31(ra.W[0] << rb[4:0]));
11	KSLLIW rt, ra, imm5u	Saturating Shift Left Logical Immediate for 32-bit Word	rt = SE(SAT.Q31(ra.W[0] << imm5u));
12	KDMABB Rt, Ra, Rb	Multiply the first 16-bit Q15 elements of two registers and transform the Q30 result into a saturated Q31 number. Add the Q31 number with a 32-bit accumulator.	res = Rt.W[0] + SAT.Q31((Ra.H0 * Rb.H0) << 1) Rt = SE(SAT.Q31(res));
13	KDMABT Rt, Ra, Rb	Multiply the first 16-bit Q15 element of one register with the second 16-bit Q15 element of another register and transform the Q30 result into a saturated Q31 number. Add the Q31 number with a 32-bit accumulator.	res = Rt.W[0] + SAT.Q31((Ra.H0 * Rb.H1) << 1) Rt = SE(SAT.Q31(res));

No.	Mnemonic	Instruction	Operation
14	KDMATT Rt, Ra, Rb	Multiply the second 16-bit Q15 elements of two registers and transform the Q30 result into a saturated Q31 number. Add the Q31 number with a 32-bit accumulator.	$\text{res} = \text{Rt.W[0]} + \text{SAT.Q31}((\text{Ra.H1} * \text{Rb.H1}) \ll 1)$ $\text{Rt} = \text{SE}(\text{SAT.Q31}(\text{res}));$
15	KABSW rt, ra	32-bit Absolute Value (scalar version)	$\text{rt} = \text{SAT.Q31}(\text{ABS}(\text{ra})); // \text{RV32}$ $\text{rt} = \text{SE}(\text{SAT.Q31}(\text{ABS}(\text{ra.W[0]}))); // \text{RV64}$

2.5.3. 32-bit Computation Instructions

There are 8 instructions here.

Table 24. 32-bit Computation Instructions

No.	Mnemonic	Instruction	Operation
1	RADDW rt, ra, rb	32-bit Signed Halving Addition	$res = (ra.W[0] + rb.W[0]) \text{ s} \gg 1;$ $rt = SE(res);$
2	URADDW rt, ra, rb	32-bit Unsigned Halving Addition	$res = (ra.W[0] + rb.W[0]) \text{ u} \gg 1;$ $rt = SE(res);$
3	RSUBW rt, ra, rb	32-bit Signed Halving Subtraction	$res = (ra.W[0] - rb.W[0]) \text{ s} \gg 1;$ $rt = SE(res);$
4	URSUBW rt, ra, rb	32-bit Unsigned Halving Subtraction	$res = (ra.W[0] - rb.W[0]) \text{ u} \gg 1;$ $rt = SE(res);$
5	MAXW rt, ra, rb	32-bit Signed Word Maximum	if ($ra.W[0] \geq rb.W[0]$) { $rt = SE(ra.W[0]);$ } else { $rt = SE(rb.W[0]);$ }
6	MINW rt, ra, rb	32-bit Signed Word Minimum	if ($ra.W[0] \geq rb.W[0]$) { $rt = SE(rb.W[0]);$ } else { $rt = SE(ra.W[0]);$ }

No.	Mnemonic	Instruction	Operation
7	MULR64 rt, ra, rb	Multiply Word Unsigned to 64-bit data	<p>RV32:</p> $mres[63:0] = ra \cdot rb;$ $r[tU] = mres.W[1];$ $r[tL] = mres.W[0];$ <p>RV64:</p> $rt = ra.W[0] \cdot rb.W[0];$
8	MULSR64 rt, ra, rb	Multiply Word Signed to 64-bit data	<p>RV32:</p> $mres[63:0] = ra \cdot rb;$ $r[tU] = mres.W[1];$ $r[tL] = mres.W[0];$ <p>RV64:</p> $rt = ra.W[0] \cdot rb.W[0];$

2.5.4. Overflow status manipulation instructions

The following table lists the user instructions related to Overflow (OV) flag manipulation.

Table 25. OV (Overflow) flag Set/Clear Instructions

No.	Mnemonic	Instruction	Operation
1	RDOV Rt	Read ucode.OV to Rt.	Rt = ZE(ucode.OV)
2	CLROV	Clear ucode.OV flag	ucode.OV = 0

2.5.5. Miscellaneous Instructions

There are 11 instructions here.

Table 26. Non-SIMD Miscellaneous Instructions

No.	Mnemonic	Instruction	Operation
1	AVE rt, ra, rb	Average with rounding	
2	SRA.u rt, ra, rb	Rounding Shift Right Arithmetic	$rt = \text{RUND}(ra \ s\gg rb[4:0]); // RV32$ $rt = \text{RUND}(ra \ s\gg rb[5:0]); // RV64$
3	SRAI.u rt, ra, imm5u	Rounding Shift Right Arithmetic Immediate	$rt = \text{RUND}(ra \ s\gg imm5u); // RV32$ $rt = \text{RUND}(ra \ s\gg imm6u); // RV64$
4	BITREV rt, ra, rb	Bit Reverse	$\text{msb} = rb[4:0]; // RV32$ $\text{msb} = rb[5:0]; // RV64$ $\text{rev}[0:\text{msb}] = ra[\text{msb}:0];$ $rt = \text{ZE}(\text{rev}[\text{msb}:0]);$
5	BITREVI rt, ra, imm5u	Bit Reverse Immediate	$\text{msb} = imm5u; // RV32$ $\text{msb} = imm6u; // RV64$ $\text{rev}[0:\text{msb}] = ra[\text{msb}:0];$ $rt = \text{ZE}(\text{rev}[\text{msb}:0]);$
6	WEXT rt, ra, rb	Extract 32-bit from a 64-bit value	$a64 = r[aU].r[aL]; // RV32$ $a64 = ra; // RV64$ $\text{lsb} = rb[4:0];$ $\text{exword} = a64[(31+\text{lsb}):\text{lsb}];$ $rt = \text{SE}(\text{exword});$

No.	Mnemonic	Instruction	Operation
7	WEXTI rt, ra, imm5u	Extract 32-bit from a 64-bit value Immediate	$a64 = r[aU].r[aL]; // RV32$ $a64 = ra; // RV64$ $lsb = imm5u;$ $exword = a64[(31+lsb):lsb];$ $rt = SE(exword);$
8	BPICK rt, ra, rb, rc	Bit-wise Pick	$rt[i] = rc[i]? ra[i] : rb[i];$ (RV32: $i=31..0$, RV64: $i=63..0$)
9	INSB rt, ra, imm3u	Insert Byte	$byte_idx = imm2u; // RV32$ $byte_idx = imm3u; // RV64$ $rt.B[byte_idx] = ra.B[0];$
10	MADDR32 rt, ra, tb	<p>Multiply and Add to 32-bit WoRd RV32:</p> $Mresult = Ra * Rb;$ $Rd = Rd + Mresult.W[0];$ <p>RV64:</p> $Mresult = Ra.W[0] * Rb.W[0];$ $tresult[31:0] = Rd.W[0] + Mresult.W[0];$ $Rd = SE64(tresult[31:0]);$	11

2.6. RV64 Only Instructions

The following tables list instructions that are only present in RV64.

There are 30 SIMD 32-bit addition or subtraction instructions.

Table 27. (RV64 Only) SIMD 32-bit Add/Subtract Instructions

No.	Mnemonic	Instruction	Operation
1	ADD32 rt, ra, rb	SIMD 32-bit Addition	$rt.W[x] = ra.W[x] + rb.W[x];$ (RV64: $x=1..0$)
2	RADD32 rt, ra, rb	SIMD 32-bit Signed Halving Addition	$rt.W[x] = (ra.W[x] + rb.W[x]) s\gg 1;$ (RV64: $x=1..0$)
3	URADD32 rt, ra, rb	SIMD 32-bit Unsigned Halving Addition	$rt.W[x] = (ra.W[x] + rb.W[x]) u\gg 1;$ (RV64: $x=1..0$)
4	KADD32 rt, ra, rb	SIMD 32-bit Signed Saturating Addition	$rt.W[x] = SAT.Q31(ra.W[x] + rb.W[x]);$ (RV64: $x=1..0$)
5	UKADD32 rt, ra, rb	SIMD 32-bit Unsigned Saturating Addition	$rt.W[x] = SAT.U32(ra.W[x] + rb.W[x]);$ (RV64: $x=1..0$)
6	SUB32 rt, ra, rb	SIMD 32-bit Subtraction	$rt.W[x] = ra.W[x] - rb.W[x];$ (RV64: $x=1..0$)
7	RSUB32 rt, ra, rb	SIMD 32-bit Signed Halving Subtraction	$rt.W[x] = (ra.W[x] - rb.W[x]) s\gg 1;$ (RV64: $x=1..0$)
8	URSUB32 rt, ra, rb	SIMD 32-bit Unsigned Halving Subtraction	$rt.W[x] = (ra.W[x] - rb.W[x]) u\gg 1;$ (RV64: $x=1..0$)
9	KSUB32 rt, ra, rb	SIMD 32-bit Signed Saturating Subtraction	$rt.W[x] = SAT.Q31(ra.W[x] - rb.W[x]);$ (RV64: $x=1..0$)
10	UKSUB32 rt, ra, rb	SIMD 32-bit Unsigned Saturating Subtraction	$rt.W[x] = SAT.U32(ra.W[x] - rb.W[x]);$ (RV64: $x=1..0$)

No.	Mnemonic	Instruction	Operation
11	CRAS32 rt, ra, rb	SIMD 32-bit Cross Add & Sub	$rt.W[x] = ra.W[x] + rb.W[x-1];$ $rt.W[x-1] = ra.W[x-1] - rb.W[x];$ (RV64: $x=1$)
12	RCRAS32 rt, ra, rb	SIMD 32-bit Signed Halving Cross Add & Sub	$rt.W[x] = (ra.W[x] + rb.W[x-1]) s\gg 1;$ $rt.W[x-1] = (ra.W[x-1] - rb.W[x]) s\gg 1;$ (RV64: $x=1$)
13	URCRAS32 rt, ra, rb	SIMD 32-bit Unsigned Halving Cross Add & Sub	$rt.W[x] = (ra.W[x] + rb.W[x-1]) u\gg 1;$ $rt.W[x-1] = (ra.W[x-1] - rb.W[x]) u\gg 1;$ (RV64: $x=1$)
14	KCRAS32 rt, ra, rb	SIMD 32-bit Signed Saturating Cross Add & Sub	$rt.W[x] = SAT.Q31(ra.W[x] + rb.W[x-1]);$ $rt.W[x-1] = SAT.Q31(ra.W[x-1] - rb.W[x]);$ (RV64: $x=1$)
15	UKCRAS32 rt, ra, rb	SIMD 32-bit Unsigned Saturating Cross Add & Sub	$rt.W[x] = SAT.U32(ra.W[x] + rb.W[x-1]);$ $rt.W[x-1] = SAT.U32(ra.W[x-1] - rb.W[x]);$ (RV64: $x=1$)
16	CRSA32 rt, ra, rb	SIMD 32-bit Cross Sub & Add	$rt.W[x] = ra.W[x] - rb.W[x-1];$ $rt.W[x-1] = ra.W[x-1] + rb.W[x];$ (RV64: $x=1$)
17	RCRSA32 rt, ra, rb	SIMD 32-bit Signed Halving Cross Sub & Add	$rt.W[x] = (ra.W[x] - rb.W[x-1]) s\gg 1;$ $rt.W[x-1] = (ra.W[x-1] + rb.W[x]) s\gg 1;$ (RV64: $x=1$)
18	URCRSA32 rt, ra, rb	SIMD 32-bit Unsigned Halving Cross Sub & Add	$rt.W[x] = (ra.W[x] - rb.W[x-1]) u\gg 1;$ $rt.W[x-1] = (ra.W[x-1] + rb.W[x]) u\gg 1;$ (RV64: $x=1$)
19	KCRSA32 rt, ra, rb	SIMD 32-bit Signed Saturating Cross Sub & Add	$rt.W[x] = SAT.Q31(ra.W[x] - rb.W[x-1]);$ $rt.W[x-1] = SAT.Q31(ra.W[x-1] + rb.W[x]);$ (RV64: $x=1$)

No.	Mnemonic	Instruction	Operation
20	UKCRSA32 rt, ra, rb	SIMD 32-bit Unsigned Saturating Cross Sub & Add	$rt.W[x] = SAT.U32(ra.W[x] - rb.W[x-1]);$ $rt.W[x-1] = SAT.U32(ra.W[x-1] + rb.W[x]);$ (RV64: $x=1$)
21	STAS32 rt, ra, rb	SIMD 32-bit Straight Add & Sub	$rt.W[x] = ra.W[x] + rb.W[x];$ $rt.W[x-1] = ra.W[x-1] - rb.W[x-1];$ (RV64: $x=1$)
22	RSTAS32 rt, ra, rb	SIMD 32-bit Signed Halving Straight Add & Sub	$rt.W[x] = (ra.W[x] + rb.W[x]) s\gg 1;$ $rt.W[x-1] = (ra.W[x-1] - rb.W[x-1]) s\gg 1;$ (RV64: $x=1$)
23	URSTAS32 rt, ra, rb	SIMD 32-bit Unsigned Halving Straight Add & Sub	$rt.W[x] = (ra.W[x] + rb.W[x]) u\gg 1;$ $rt.W[x-1] = (ra.W[x-1] - rb.W[x-1]) u\gg 1;$ (RV64: $x=1$)
24	KSTAS32 rt, ra, rb	SIMD 32-bit Signed Saturating Straight Add & Sub	$rt.W[x] = SAT.Q31(ra.W[x] + rb.W[x]);$ $rt.W[x-1] = SAT.Q31(ra.W[x-1] - rb.W[x-1]);$ (RV64: $x=1$)
25	UKSTAS32 rt, ra, rb	SIMD 32-bit Unsigned Saturating Straight Add & Sub	$rt.W[x] = SAT.U32(ra.W[x] + rb.W[x]);$ $rt.W[x-1] = SAT.U32(ra.W[x-1] - rb.W[x-1]);$ (RV64: $x=1$)
26	STSA32 rt, ra, rb	SIMD 32-bit Straight Sub & Add	$rt.W[x] = ra.W[x] - rb.W[x];$ $rt.W[x-1] = ra.W[x-1] + rb.W[x-1];$ (RV64: $x=1$)
27	RSTSA32 rt, ra, rb	SIMD 32-bit Signed Halving Straight Sub & Add	$rt.W[x] = (ra.W[x] - rb.W[x]) s\gg 1;$ $rt.W[x-1] = (ra.W[x-1] + rb.W[x-1]) s\gg 1;$ (RV64: $x=1$)
28	URSTSA32 rt, ra, rb	SIMD 32-bit Unsigned Halving Straight Sub & Add	$rt.W[x] = (ra.W[x] - rb.W[x]) u\gg 1;$ $rt.W[x-1] = (ra.W[x-1] + rb.W[x-1]) u\gg 1;$ (RV64: $x=1$)

No.	Mnemonic	Instruction	Operation
29	KSTSA32 rt, ra, rb	SIMD 32-bit Signed Saturating Straight Sub & Add	$rt.W[x] = SAT.Q31(ra.W[x] - rb.W[x]);$ $rt.W[x-1] = SAT.Q31(ra.W[x-1] + rb.W[x-1]);$ (RV64: $x=1$)
30	UKSTSA32 rt, ra, rb	SIMD 32-bit Unsigned Saturating Straight Sub & Add	$rt.W[x] = SAT.U32(ra.W[x] - rb.W[x]);$ $rt.W[x-1] = SAT.U32(ra.W[x-1] + rb.W[x-1]);$ (RV64: $x=1$)

There are 14 SIMD 32-bit shift instructions.

Table 28. (RV64 Only) SIMD 32-bit Shift Instructions

No.	Mnemonic	Instruction	Operation
1	SRA32 rt, ra, rb	SIMD 32-bit Shift Right Arithmetic	$rt.W[x] = ra.W[x] s\gg rb[4:0];$ (RV64: $x=1..0$)
2	SRAI32 rt, ra, im5u	SIMD 32-bit Shift Right Arithmetic Immediate	$rt.W[x] = ra.W[x] s\gg im5u;$ (RV64: $x=1..0$)
3	SRA32.u rt, ra, rb	SIMD 32-bit Rounding Shift Right Arithmetic	$rt.W[x] = RUND(ra.W[x] s\gg rb[4:0]);$ (RV64: $x=1..0$)
4	SRAI32.u rt, ra, im5u	SIMD 32-bit Rounding Shift Right Arithmetic Immediate	$rt.W[x] = RUND(ra.W[x] s\gg im5u);$ (RV64: $x=1..0$)
5	SRL32 rt, ra, rb	SIMD 32-bit Shift Right Logical	$rt.W[x] = ra.W[x] u\gg rb[4:0];$ (RV64: $x=1..0$)
6	SRLI32 rt, ra, im5u	SIMD 32-bit Shift Right Logical Immediate	$rt.W[x] = ra.W[x] u\gg im5u;$ (RV64: $x=1..0$)
7	SRL32.u rt, ra, rb	SIMD 32-bit Rounding Shift Right Logical	$rt.W[x] = RUND(ra.W[x] u\gg rb[4:0]);$ (RV64: $x=1..0$)
8	SRLI32.u rt, ra, im5u	SIMD 32-bit Rounding Shift Right Logical Immediate	$rt.W[x] = RUND(ra.W[x] u\gg im5u);$ (RV64: $x=1..0$)

No.	Mnemonic	Instruction	Operation
9	SLL32 rt, ra, rb	SIMD 32-bit Shift Left Logical	$rt.W[x] = ra.W[x] \ll rb[4:0];$ (RV64: $x=1..0$)
10	SLLI32 rt, ra, im5u	SIMD 32-bit Shift Left Logical Immediate	$rt.W[x] = ra.W[x] \ll im5u;$ (RV64: $x=1..0$)
11	KSLL32 rt, ra, rb	SIMD 32-bit Saturating Shift Left Logical	$rt.W[x] = SAT.Q31(ra.W[x] \ll rb[4:0]);$ (RV64: $x=1..0$)
12	KSLLI32 rt, ra, im5u	SIMD 32-bit Saturating Shift Left Logical Immediate	$rt.W[x] = SAT.Q31(ra.W[x] \ll im5u);$ (RV64: $x=1..0$)
13	KSLRA32 rt, ra, rb	SIMD 32-bit Shift Left Logical with Saturation & Shift Right Arithmetic	if ($rb[5:0] < 0$) $rt.W[x] = ra.W[x] s\gg -rb[5:0];$ if ($rb[5:0] > 0$) $rt.W[x] = SAT.Q31(ra.W[x] \ll rb[5:0]);$ (RV64: $x=1..0$)
14	KSLRA32.u rt, ra, rb	SIMD 32-bit Shift Left Logical with Saturation & Rounding Shift Right Arithmetic	if ($rb[5:0] < 0$) $rt.W[x] = RUND(ra.W[x] s\gg -rb[5:0]);$ if ($rb[5:0] > 0$) $rt.W[x] = SAT.Q31(ra.W[x] \ll rb[5:0]);$ (RV64: $x=1..0$)

Table 29. (RV64 Only) SIMD 32-bit Miscellaneous Instructions

No.	Mnemonic	Instruction	Operation
1	SMIN32 rt, ra, rb	SIMD 32-bit Signed Minimum	$rt.W[x] = (ra.W[x] < rb.W[x])? ra.W[x] : rb.W[x];$ (RV64: $x=1..0$)
2	UMIN32 rt, ra, rb	SIMD 32-bit Unsigned Minimum	$rt.W[x] = (ra.W[x] u< rb.W[x])? ra.W[x] : rb.W[x];$ (RV64: $x=1..0$)

No.	Mnemonic	Instruction	Operation
3	SMAX32 rt, ra, rb	SIMD 32-bit Signed Maximum	$rt.W[x] = (ra.W[x] > rb.W[x])? ra.W[x] : rb.W[x];$ (RV64: $x=1..0$)
4	UMAX32 rt, ra, rb	SIMD 32-bit Unsigned Maximum	$rt.W[x] = (ra.W[x] u> rb.W[x])? ra.W[x] : rb.W[x];$ (RV64: $x=1..0$)
5	KABS32 rt, ra	SIMD 32-bit Absolute Value	$rt.W[x] = SAT.Q31(ABS(ra.W[x]));$ (RV64: $x=1..0$)

Table 30. (RV64 Only) SIMD Q15 saturating Multiply Instructions

No.	Mnemonic	Instruction	Operation
1	KHMBB16 Rt, Ra, Rb	Multiply the first 16-bit Q15 elements of 32-bit chunks in two registers and transform the Q30 results into saturated Q15 numbers.	$Rt.W[x] = SAT.Q15((Ra.W[x].H0 * Rb.W[x].H0) s>> 15)$ (RV64: $x=1..0$)
2	KHMBT16 Rt, Ra, Rb	Multiply the first 16-bit Q15 element of 32-bit chunks in one register with the second 16-bit Q15 element of 32-bit chunks in another register and transform the Q30 results into saturated Q15 numbers.	$Rt.W[x] = SAT.Q15((Ra.W[x].H0 * Rb.W[x].H1) s>> 15)$ (RV64: $x=1..0$)
3	KHM TT16 Rt, Ra, Rb	Multiply the second 16-bit Q15 elements of 32-bit chunks in two registers and transform the Q30 results into saturated Q15 numbers.	$Rt.W[x] = SAT.Q15((Ra.W[x].H1 * Rb.W[x].H1) s>> 15)$ (RV64: $x=1..0$)

No.	Mnemonic	Instruction	Operation
4	KDMBB16 Rt, Ra, Rb	Multiply the first 16-bit Q15 elements of 32-bit chunks in two registers and transform the Q30 results into saturated Q31 numbers.	$Rt.W[x] = SAT.Q31((Ra.W[x].H0 * Rb.W[x].H0) \ll 1)$ (RV64: $x=1..0$)
5	KDMBT16 Rt, Ra, Rb	Multiply the first 16-bit Q15 element of 32-bit chunks in one register with the second 16-bit Q15 element of 32-bit chunks in another register and transform the Q30 results into saturated Q31 numbers.	$Rt.W[x] = SAT.Q31((Ra.W[x].H0 * Rb.W[x].H1) \ll 1)$ (RV64: $x=1..0$)
6	KDMTT16 Rt, Ra, Rb	Multiply the second 16-bit Q15 elements of 32-bit chunks in two registers and transform the Q30 results into saturated Q31 numbers.	$Rt.W[x] = SAT.Q31((Ra.W[x].H1 * Rb.W[x].H1) \ll 1)$ (RV64: $x=1..0$)
7	KDMABB16 Rt, Ra, Rb	Multiply the first 16-bit Q15 elements of 32-bit chunks in two registers and transform the Q30 results into saturated Q31 numbers. Add the Q31 numbers with 32-bit accumulator values.	$res = SAT.Q31((Ra.W[x].H0 * Rb.W[x].H0) \ll 1)$ $Rt.W[x] = SAT.Q31(Rt.W[x] + res)$ (RV64: $x=1..0$)

No.	Mnemonic	Instruction	Operation
8	KDMABT16 Rt, Ra, Rb	Multiply the first 16-bit Q15 element of 32-bit chunks in one register with the second 16-bit Q15 element of 32-bit chunks in another register and transform the Q30 results into saturated Q31 numbers. Add the Q31 numbers with 32-bit accumulator values.	$\text{res} = \text{SAT.Q31}((\text{Ra.W}[x].H0 * \text{Rb.W}[x].H1) \ll 1)$ $\text{Rt.W}[x] = \text{SAT.Q31}(\text{Rt.W}[x] + \text{res})$ (RV64: $x=1..0$)
9	KDMATT16 Rt, Ra, Rb	Multiply the second 16-bit Q15 elements of 32-bit chunks in two registers and transform the Q30 results into saturated Q31 numbers. Add the Q31 numbers with 32-bit accumulator values.	$\text{res} = \text{SAT.Q31}((\text{Ra.W}[x].H1 * \text{Rb.W}[x].H1) \ll 1)$ $\text{Rt.W}[x] = \text{SAT.Q31}(\text{Rt.W}[x] + \text{res})$ (RV64: $x=1..0$)

Table 31. (RV64 Only) 32-bit Multiply Instructions

No.	Mnemonic	Instruction	Operation
1	SMBB32 Rt, Ra, Rb	Multiply the first 32-bit elements of two registers.	$\text{Rt} = \text{Ra.W}[0] * \text{Rb.W}[0];$
2	SMBT32 Rt, Ra, Rb	Multiply the first 32-bit element of one register with the second 32-bit element of another register.	$\text{Rt} = \text{Ra.W}[0] * \text{Rb.W}[1];$
3	SMTT32 Rt, Ra, Rb	Multiply the second 32-bit elements of two registers.	$\text{Rt} = \text{Ra.W}[1] * \text{Rb.W}[1];$

Table 32. (RV64 Only) 32-bit Multiply & Add Instructions

No.	Mnemonic	Instruction	Operation
1	KMABB32 Rt, Ra, Rb	Multiply the first 32-bit elements of two registers and the signed multiplication result is added to a third register with Q63 saturation. (64 = 64 + 32x32)	Rt = SAT.Q63(Rt + Ra.W[0] * Rb.W[0]);
2	KMABT32 Rt, Ra, Rb	Multiply the first 32-bit element of one register with the second 32-bit element of another register and the signed multiplication result is added to a third register with Q63 saturation. (64 = 64 + 32x32)	Rt = SAT.Q63(Rt + Ra.W[0] * Rb.W[1]);
3	KMATT32 Rt, Ra, Rb	Multiply the second 32-bit elements of two registers and the signed multiplication result is added to a third register with Q63 saturation. (64 = 64 + 32x32)	Rt = SAT.Q63(Rt + Ra.W[1] * Rb.W[1]);

Table 33. (RV64 Only) 32-bit Parallel Multiply & Add Instructions

No.	Mnemonic	Instruction	Operation
1	KMDA32 Rt, Ra, Rb	<p>Multiply the corresponding 32-bit elements of two registers and add the signed multiplication results with Q63 saturation.</p> <p>(64 = 32x32 + 32x32)</p>	$Rt = SAT.Q63(Ra.W[1] * Rb.W[1] + Ra.W[0] * Rb.W[0]);$
2	KMXDA32 Rt, Ra, Rb	<p>Multiply the cross-positioned 32-bit elements of two registers and add the signed multiplication results with Q63 saturation.</p> <p>(64 = 32x32 + 32x32)</p>	$Rt = SAT.Q63(Ra.W[1] * Rb.W[0] + Ra.W[0] * Rb.W[1]);$
3	KMADA32 Rt, Ra, Rb	<p>Multiply the corresponding 32-bit elements of two registers and add the signed multiplication results and a third register with Q63 saturation.</p> <p>(64 = 64+ 32x32 + 32x32)</p>	$Rt = SAT.Q63(Rt + Ra.W[1] * Rb.W[1] + Ra.W[0] * Rb.W[0]);$

No.	Mnemonic	Instruction	Operation
4	KMAXDA32 Rt, Ra, Rb	<p>Multiply the cross-positioned 32-bit elements of two registers and add the signed multiplication results and a third register with Q63 saturation.</p> $(64 = 64 + 32 \times 32 + 32 \times 32)$	$Rt = SAT.Q63(Rt + Ra.W[1] * Rb.W[0] + Ra.W[0] * Rb.W[1]);$
5	KMADS32 Rt, Ra, Rb	<p>Multiply the corresponding 32-bit elements of two registers and add the top signed multiplication result with a third register and subtract the bottom signed multiplication result with Q63 saturation.</p> $(64 = 64 + 32 \times 32 - 32 \times 32)$	$Rt = SAT.Q63(Rt + Ra.W[1] * Rb.W[1] - Ra.W[0] * Rb.W[0]);$
6	KMADRS32 Rt, Ra, Rb	<p>Multiply the corresponding 32-bit elements of two registers and add the bottom signed multiplication result with a third register and subtract the top signed multiplication result with Q63 saturation.</p> $(64 = 64 + 32 \times 32 - 32 \times 32)$	$Rt = SAT.Q63(Rt + Ra.W[0] * Rb.W[0] - Ra.W[1] * Rb.W[1]);$

No.	Mnemonic	Instruction	Operation
7	KMAXDS32 Rt, Ra, Rb	<p>Multiply the cross-positioned 32-bit elements of two registers and add the top signed multiplication result with a third register and subtract the bottom signed multiplication result with Q63 saturation.</p> $(64 = 64 + 32x32 - 32x32)$	$Rt = SAT.Q63(Rt + Ra.W[1] * Rb.W[0] - Ra.W[0] * Rb.W[1]);$
8	KMSDA32 Rt, Ra, Rb	<p>Multiply the corresponding 32-bit elements of two registers and subtract the signed multiplication results from a third register with Q63 saturation.</p> $(64 = 64 - 32x32 - 32x32)$	$Rt = SAT.Q63(Rt - Ra.W[1] * Rb.W[1] - Ra.W[0] * Rb.W[0]);$
9	KMSXDA32 Rt, Ra, Rb	<p>Multiply the cross-positioned 32-bit elements of two registers and subtract the signed multiplication results from a third register with Q63 saturation.</p> $(64 = 64 - 32x32 - 32x32)$	$Rt = SAT.Q63(Rt - Ra.W[1] * Rb.W[0] - Ra.W[0] * Rb.W[1]);$

No.	Mnemonic	Instruction	Operation
10	SMD32 Rt, Ra, Rb	Multiply the corresponding 32-bit elements of two registers and subtract the bottom signed multiplication result from the top result. (64 = 32x32 - 32x32)	Rt = Ra.W[1] * Rb.W[1] - Ra.W[0] * Rb.W[0];
11	SMDRS32 Rt, Ra, Rb	Multiply the corresponding 32-bit elements of two registers and subtract the top signed multiplication result from the bottom result. (64 = 32x32 - 32x32)	Rt = Ra.W[0] * Rb.W[0] - Ra.W[1] * Rb.W[1];
12	SMXDS32 Rt, Ra, Rb	Multiply the cross-positioned 32-bit elements of two registers and subtract the bottom signed multiplication result from the top result. (64 = 32x32 - 32x32)	Rt = Ra.W[1] * Rb.W[0] - Ra.W[0] * Rb.W[1];

Table 34. (RV64 Only) Non-SIMD 32-bit Shift Instructions

No.	Mnemonic	Instruction	Operation
1	SRAIW.u Rt, Ra, imm5u	32-bit Rounding arithmetic shift right immediate	Rt = SE64(RUND(Ra.W[0] s>> imm5u));

There are four 32-bit packing instructions here.

Table 35. 32-bit Packing Instructions

No.	Mnemonic	Instruction	Operation
1	PKBB32 rt, ra, rb	Pack two 32-bit data from Bottoms	$rt = \text{CONCAT}(ra.W[0], rb.W[0]);$
2	PKBT32 rt, ra, rb	Pack two 32-bit data Bottom & Top	$rt = \text{CONCAT}(ra.W[0], rb.W[1]);$
3	PKTB32 rt, ra, rb	Pack two 32-bit data Top & Bottom	$rt = \text{CONCAT}(ra.W[1], rb.W[0]);$
4	PKTT32 rt, ra, rb	Pack two 32-bit data from Tops	$rt = \text{CONCAT}(ra.W[1], rb.W[1]);$

Chapter 3. P Extension Subsets

The P extension instructions will be divided into several Z-extension subsets to facilitate the trade-off between implementation complexity and application performance.

3.1. Zp64 (for RV32 Only)

The instructions in Zp64 extension will read or write 64-bit operands using register-pairs in RV32. For RV64, the Zp64 instructions will be included in the baseline P extension.

The instructions in Zp64 are listed in the following table.

No.	Instruction
1	SMAL
2	ADD64
3	RADD64
4	URADD64
5	KADD64
6	UKADD64
7	SUB64
8	RSUB64
9	URSUB64
10	KSUB64
11	UKSUB64
12	SMAR64
13	SMSR64
14	UMAR64
15	UMSR64
16	KMAR64
17	KMSR64
18	UKMAR64
19	UKMSR64
20	SMALBB
21	SMALBT
22	SMALTT
23	SMALDA
24	SMALXDA

No.	Instruction
25	SMALDS
26	SMALDRS
27	SMALXDS
28	SMSLDA
29	SMSLXDA

Chapter 4. Detailed Instruction Descriptions (both RV32 & RV64)

The sections in this chapter describe the detailed operations of the P extension instructions for both RV32 and RV64 in alphabetical order.

Some RV32 instructions reading or writing paired 32-bit registers and the pairing is constrained to a pair of { R_n, R_{n+1} } registers with “n” being an even number. Based on this constraint, since R₀ is X₀, i.e., 0, a constant, the pair of { R₀, R₁ } cannot be used in these instructions. Also, although the instruction encodings of these instructions allow a software developer to specify the pair number of an operand using either an even or an odd number of register index, an assembler or instruction generator may choose to block the use of an odd number as a register index for the paired-register operand.

4.1. ADD8 (SIMD 8-bit Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ADD8 0100100	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
ADD8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit integer element additions simultaneously.

Description: This instruction adds the 8-bit integer elements in Rs1 with the 8-bit integer elements in Rs2, and then writes the 8-bit element results to Rd.

Operations:

```
Rd.B[x] = Rs1.B[x] + Rs2.B[x];  
for RV32: x=3…0,  
for RV64: x=7…0
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned addition.

Intrinsic functions:

```
unsigned long __nds__add8(unsigned long a, unsigned long b);
```

```
RV32:  
uint8x4_t __nds__v_uadd8(uint8x4_t a, uint8x4_t b);  
int8x4_t __nds__v_sadd8(int8x4_t a, int8x4_t b);  
RV64:  
uint8x8_t __nds__v_uadd8(uint8x8_t a, uint8x8_t b);  
int8x8_t __nds__v_sadd8(int8x8_t a, int8x8_t b);
```

4.2. ADD16 (SIMD 16-bit Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ADD16 0100000	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
ADD16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit integer element additions simultaneously.

Description: This instruction adds the 16-bit integer elements in Rs1 with the 16-bit integer elements in Rs2, and then writes the 16-bit element results to Rd.

Operations:

```
Rd.H[x] = Rs1.H[x] + Rs2.H[x];  
for RV32: x=1…0,  
for RV64: x=3…0
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned addition.

Intrinsic functions:

```
unsigned long __nds__add16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_uadd16(uint16x2_t a, uint16x2_t b);  
int16x2_t __nds__v_sadd16(int16x2_t a, int16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_uadd16(uint16x4_t a, uint16x4_t b);  
int16x4_t __nds__v_sadd16(int16x4_t a, int16x4_t b);
```

4.3. ADD64 (64-bit Addition)

Type: 64-bit Profile

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ADD64 1100000	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
ADD64 Rd, Rs1, Rs2
```

Purpose: Add two 64-bit signed or unsigned integers.

RV32 Description: This instruction adds the 64-bit integer of an even/odd pair of registers specified by Rs1(4,1) with the 64-bit integer of an even/odd pair of registers specified by Rs2(4,1), and then writes the 64-bit result to an even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., value d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction has the same behavior as the ADD instruction in RV64I.

Operations:

RV32:

```
t_L = CONCAT(Rd(4,1),1\b0); t_H = CONCAT(Rd(4,1),1\b1);
a_L = CONCAT(Rs1(4,1),1\b0); a_H = CONCAT(Rs1(4,1),1\b1);
b_L = CONCAT(Rs2(4,1),1\b0); b_H = CONCAT(Rs2(4,1),1\b1);
R[t_H].R[t_L] = R[a_H].R[a_L] + R[b_H].R[b_L];
```

RV64:

```
Rd = Rs1 + Rs2;
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned addition.

Intrinsic functions:

```
long long __nds__sadd64(long long a, long long b);  
unsigned long long __nds__uadd64(unsigned long long a, unsigned long long b);
```

4.4. AVE (Average with Rounding)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
AVE 1110000	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
AVE Rd, Rs1, Rs2
```

Purpose: Calculate the average of the contents of two general registers.

Description: This instruction calculates the average value of two signed integers stored in Rs1 and Rs2, rounds up a half-integer result to the nearest integer, and writes the result to Rd.

Operations:

```
Sum = CONCAT(Rs1[MSB],Rs1[MSB:0]) + CONCAT(Rs2[MSB],Rs2[MSB:0]) + 1;  
Rd = Sum[(MSB+1):1];  
for RV32: MSB=31,  
for RV64: MSB=63
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long __nds_ave(long a, long b);
```

4.5. BITREV (Bit Reverse)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
BITREV 1110011	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
BITREV Rd, Rs1, Rs2
```

Purpose: Reverse the bit positions of the source operand within a specified width starting from bit 0. The reversed width is a variable from a GPR.

Description: This instruction reverses the bit positions of the content of Rs1. The reversed bit width is calculated as $Rs2[4:0]+1$ (RV32) or $Rs2[5:0]+1$ (RV64). The upper bits beyond the reversed width are filled with zeros. After the bit reverse operation, the result is written to Rd.

Operations:

```
msb = Rs2[4:0]; (for RV32)
msb = Rs2[5:0]; (for RV64)
rev[0:msb] = Rs1[msb:0];
Rd = ZE(rev[msb:0]);
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_bitrev(unsigned long a, unsigned long b);
```

4.6. BITREVI (Bit Reverse Immediate)

Type: DSP

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
BITREVI 111010	imm[5]	imm[4:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
(RV32) BITREVI Rd, Rs1, imm[4:0]
(RV64) BITREVI Rd, Rs1, imm[5:0]
```

Purpose: Reverse the bit positions of the source operand within a specified width starting from bit 0. The reversed width is an immediate value.

Description: This instruction reverses the bit positions of the content of Rs1. The reversed bit width is calculated as imm[4:0]+1 (RV32) or imm[5:0]+1 (RV64). The upper bits beyond the reversed width are filled with zeros. After the bit reverse operation, the result is written to Rd.

Operations:

```
msb = imm[4:0]; (RV32)
msb = imm[5:0]; (RV64)
rev[0:msb] = Rs1[msb:0];
Rd = ZE(rev[msb:0]);
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__bitrev(unsigned long a, unsigned long b);
```

4.7. BPICK (Bit-wise Pick)

Type: DSP

Format:

31 27	26 25	24 20	19 15	14 12	11 7	6 0
BPICK Rc	00	Rs2	Rs1	011	Rd	GE80B 1111111

Syntax:

```
BPICK Rd, Rs1, Rs2, Rc
```

Purpose: Select from two source operands based on a bit mask in the third operand.

Description: This instruction selects individual bits from Rs1 or Rs2, based on the bit mask value in Rc. If a bit in Rc is 1, the corresponding bit is from Rs1; otherwise, the corresponding bit is from Rs2. The selection results are written to Rd.

Operations:

```
Rd[x] = Rc[x]? Rs1[x] : Rs2[x];  
for RV32, x=31...0  
for RV64, x=63...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_bpick(unsigned long a, unsigned long b, unsigned long c);
```

4.8. CLROV (Clear OV flag)

Type: DSP

Format:

31 20	19 15	14 12	11 7	6 0
ucode (0x801) 100000000001	00001	111	Rd	SYSTEM 1110011

Syntax:

```
CLROV # pseudo mnemonic
```

Purpose: This pseudo instruction is an alias to “CSRRCI x0, ucode, 1” instruction.

Intrinsic functions:

```
void __nds__clrov(void);
```

4.9. CLRS8 (SIMD 8-bit Count Leading Redundant Sign)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP2 1010111	CLRS8 00000	Rs1	000	Rd	GE80B 1111111

Syntax:

```
CLRS8 Rd, Rs1
```

Purpose: Count the number of redundant sign bits of the 8-bit elements of a general register.

Description: Starting from the bits next to the sign bits of the 8-bit elements of Rs1, this instruction counts the number of redundant sign bits and writes the result to the corresponding 8-bit elements of Rd.

Operations:

```
snum[x] = Rs1.B[x];
cnt[x] = 0;
for (i = 6 to 0) {
    if (snum[x](i) == snum[x](7)) {
        cnt[x] = cnt[x] + 1;
    } else {
        break;
    }
}
Rd.B[x] = cnt[x];
for RV32: x=3...0
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__clrs8(unsigned long a);
```

RV32:

```
uint8x4_t __nds__v_clrs8(int8x4_t a);
```

RV64:

```
uint8x8_t __nds__v_clrs8(int8x8_t a);
```

4.10. CLRS16 (SIMD 16-bit Count Leading Redundant Sign)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP2 1010111	CLRS16 01000	Rs1	000	Rd	GE80B 1111111

Syntax:

```
CLRS16 Rd, Rs1
```

Purpose: Count the number of redundant sign bits of the 16-bit elements of a general register.

Description: Starting from the bits next to the sign bits of the 16-bit elements of Rs1, this instruction counts the number of redundant sign bits and writes the result to the corresponding 16-bit elements of Rd.

Operations:

```
snum[x] = Rs1.H[x];
cnt[x] = 0;
for (i = 14 to 0) {
    if (snum[x](i) == snum[x](15)) {
        cnt[x] = cnt[x] + 1;
    } else {
        break;
    }
}
Rd.H[x] = cnt[x];
for RV32: x=1...0
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_clrs16(unsigned long a);
```

RV32:

```
uint16x2_t __nds__v_clrs16(int16x2_t a);
```

RV64:

```
uint16x4_t __nds__v_clrs16(int16x4_t a);
```

4.11. CLRS32 (SIMD 32-bit Count Leading Redundant Sign)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP2 1010111	CLRS32 11000	Rs1	000	Rd	GE80B 1111111

Syntax:

```
CLRS32 Rd, Rs1
```

Purpose: Count the number of redundant sign bits of the 32-bit elements of a general register.

Description: Starting from the bits next to the sign bits of the 32-bit elements of Rs1, this instruction counts the number of redundant sign bits and writes the result to the corresponding 32-bit elements of Rd.

Operations:

```
snum[x] = Rs1.W[x];
cnt[x] = 0;
for (i = 30 to 0) {
    if (snum[x](i) == snum[x](31)) {
        cnt[x] = cnt[x] + 1;
    } else {
        break;
    }
}
Rd.W[x] = cnt[x];
for RV32: x=0
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_clrs32(unsigned long a);
```

RV64:

```
uint32x2_t __nds_v_clrs32(int32x2_t a);
```

4.12. CLO8 (SIMD 8-bit Count Leading One)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP2 1010111	CLO8 00011	Rs1	000	Rd	GE80B 1111111

Syntax:

```
CLO8 Rd, Rs1
```

Purpose: Count the number of leading one bits of the 8-bit elements of a general register.

Description: Starting from the most significant bits of the 8-bit elements of Rs1, this instruction counts the number of leading one bits and writes the results to the corresponding 8-bit elements of Rd.

Operations:

```
snum[x] = Rs1.B[x];
cnt[x] = 0;
for (i = 7 to 0) {
    if (snum[x](i) == 1) {
        cnt[x] = cnt[x] + 1;
    } else {
        break;
    }
}
Rd.B[x] = cnt[x];
for RV32: x=3...0
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__clo8(unsigned long a);
```

RV32:

```
uint8x4_t __nds__v_clo8(uint8x4_t a);
```

RV64:

```
uint8x8_t __nds__v_clo8(uint8x8_t a);
```

4.13. CLO16 (SIMD 16-bit Count Leading One)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP2 1010111	CLO16 01011	Rs1	000	Rd	GE80B 1111111

Syntax:

```
CLO16 Rd, Rs1
```

Purpose: Count the number of leading one bits of the 16-bit elements of a general register.

Description: Starting from the most significant bits of the 16-bit elements of Rs1, this instruction counts the number of leading one bits and writes the results to the corresponding 16-bit elements of Rd.

Operations:

```
snum[x] = Rs1.H[x];
cnt[x] = 0;
for (i = 15 to 0) {
    if (snum[x](i) == 1) {
        cnt[x] = cnt[x] + 1;
    } else {
        break;
    }
}
Rd.H[x] = cnt[x];
for RV32: x=1...0
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__clo16(unsigned long a);
```

RV32:

```
uint16x2_t __nds__v_clo16(uint16x2_t a);
```

RV64:

```
uint16x4_t __nds__v_clo16(uint16x4_t a);
```

4.14. CLO32 (SIMD 32-bit Count Leading One)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP2 1010111	CLO32 11011	Rs1	000	Rd	GE80B 1111111

Syntax:

```
CLO32 Rd, Rs1
```

Purpose: Count the number of leading one bits of the 32-bit elements of a general register.

Description: Starting from the most significant bits of the 32-bit elements of Rs1, this instruction counts the number of leading one bits and writes the results to the corresponding 32-bit elements of Rd.

Operations:

```
snum[x] = Rs1.W[x];
cnt[x] = 0;
for (i = 31 to 0) {
    if (snum[x](i) == 1) {
        cnt[x] = cnt[x] + 1;
    } else {
        break;
    }
}
Rd.W[x] = cnt[x];
for RV32: x=0
for RV64: x=1…0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__clo32(unsigned long a);
```

RV64:

```
uint32x2_t __nds__v_clo32(uint32x2_t a);
```

4.15. CLZ8 (SIMD 8-bit Count Leading Zero)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP2 1010111	CLZ8 00001	Rs1	000	Rd	GE80B 1111111

Syntax:

```
CLZ8 Rd, Rs1
```

Purpose: Count the number of leading zero bits of the 8-bit elements of a general register.

Description: Starting from the most significant bits of the 8-bit elements of Rs1, this instruction counts the number of leading zero bits and writes the results to the corresponding 8-bit elements of Rd.

Operations:

```
snum[x] = Rs1.B[x];
cnt[x] = 0;
for (i = 7 to 0) {
    if (snum[x](i) == 0) {
        cnt[x] = cnt[x] + 1;
    } else {
        break;
    }
}
Rd.B[x] = cnt[x];
for RV32: x=3...0
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__clz8(unsigned long a);
```

RV32:

```
uint8x4_t __nds__v_clz8(uint8x4_t a);
```

RV64:

```
uint8x8_t __nds__v_clz8(uint8x8_t a);
```

4.16. CLZ16 (SIMD 16-bit Count Leading Zero)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP2 1010111	CLZ16 01001	Rs1	000	Rd	GE80B 1111111

Syntax:

```
CLZ16 Rd, Rs1
```

Purpose: Count the number of leading zero bits of the 16-bit elements of a general register.

Description: Starting from the most significant bits of the 16-bit elements of Rs1, this instruction counts the number of leading zero bits and writes the results to the corresponding 16-bit elements of Rd.

Operations:

```
snum[x] = Rs1.H[x];
cnt[x] = 0;
for (i = 15 to 0) {
    if (snum[x](i) == 0) {
        cnt[x] = cnt[x] + 1;
    } else {
        break;
    }
}
Rd.H[x] = cnt[x];
for RV32: x=1...0
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__clz16(unsigned long a);
```

RV32:

```
uint16x2_t __nds__v_clz16(uint16x2_t a);
```

RV64:

```
uint16x4_t __nds__v_clz16(uint16x4_t a);
```

4.17. CLZ32 (SIMD 32-bit Count Leading Zero)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP2 1010111	CLZ32 11001	Rs1	000	Rd	GE80B 1111111

Syntax:

```
CLZ32 Rd, Rs1
```

Purpose: Count the number of leading zero bits of the 32-bit elements of a general register.

Description: Starting from the most significant bits of the 32-bit elements of Rs1, this instruction counts the number of leading zero bits and writes the results to the corresponding 32-bit elements of Rd.

Operations:

```
snum[x] = Rs1.W[x];
cnt[x] = 0;
for (i = 31 to 0) {
    if (snum[x](i) == 0) {
        cnt[x] = cnt[x] + 1;
    } else {
        break;
    }
}
Rd.W[x] = cnt[x];
for RV32: x=0
for RV64: x=1…0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__clz32(unsigned long a);
```

RV64:

```
uint32x2_t __nds__v_clz32(uint32x2_t a);
```

4.18. CMPEQ8 (SIMD 8-bit Integer Compare Equal)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
CMPEQ8 0100111	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
CMPEQ8 Rs, Rs1, Rs2
```

Purpose: Do 8-bit integer elements equal comparisons simultaneously.

Description: This instruction compares the 8-bit integer elements in Rs1 with the 8-bit integer elements in Rs2 to see if they are equal. If they are equal, the result is 0xFF; otherwise, the result is 0x0. The 8-bit element comparison results are written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] == Rs2.B[x])? 0xff : 0x0;  
for RV32: x=3...0,  
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned numbers.

Intrinsic functions:

```
unsigned long __nds_cmpeq8(unsigned long a, unsigned long b);
```

RV32:

```
uint8x4_t __nds_v_scmpeq8(int8x4_t a, int8x4_t b);  
uint8x4_t __nds_v_umpeq8(uint8x4_t a, uint8x4_t b);
```

RV64:

```
uint8x8_t __nds_v_scmpeq8(int8x8_t a, int8x8_t b);  
uint8x8_t __nds_v_umpeq8(uint8x8_t a, uint8x8_t b);
```

4.19. CMPEQ16 (SIMD 16-bit Integer Compare Equal)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
CMPEQ16 0100110	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
CMPEQ16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit integer elements equal comparisons simultaneously.

Description: This instruction compares the 16-bit integer elements in Rs1 with the 16-bit integer elements in Rs2 to see if they are equal. If they are equal, the result is 0xFFFF; otherwise, the result is 0x0. The 16-bit element comparison results are written to Rt.

Operations:

```
Rd.H[x] = (Rs1.H[x] == Rs2.H[x])? 0xffff : 0x0;  
for RV32: x=1…0,  
for RV64: x=3…0
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned numbers.

Intrinsic functions:

```
unsigned long __nds_cmpeq16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds_v_scmpeq16(int16x2_t a, int16x2_t b);  
uint16x2_t __nds_v_ucmpeq16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds_v_scmpeq16(int16x4_t a, int16x2_t b);  
uint16x4_t __nds_v_ucmpeq16(uint16x4_t a, uint16x2_t b);
```

4.20. CRAS16 (SIMD 16-bit Cross Addition & Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
CRAS16 0100010	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
CRAS16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit integer element addition and 16-bit integer element subtraction in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks.

Description: This instruction adds the 16-bit integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit integer element in [15:0] of 32-bit chunks in Rs2, and writes the result to [31:16] of 32-bit chunks in Rd; at the same time, it subtracts the 16-bit integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit integer element in [15:0] of 32-bit chunks, and writes the result to [15:0] of 32-bit chunks in Rd.

Operations:

```
Rd.W[x][31:16] = Rs1.W[x][31:16] + Rs2.W[x][15:0];  
Rd.W[x][15:0] = Rs1.W[x][15:0] - Rs2.W[x][31:16];  
for RV32, x=0  
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned operations.

Intrinsic functions:

```
unsigned long __nds_cras16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_uclas16(uint16x2_t a, uint16x2_t b);  
int16x2_t __nds__v_sclas16(int16x2_t a, int16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_uclas16(uint16x4_t a, uint16x4_t b);  
int16x4_t __nds__v_sclas16(int16x4_t a, int16x4_t b);
```

4.21. CRSA16 (SIMD 16-bit Cross Subtraction & Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
CRSA16 0100011	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
CRSA16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit integer element subtraction and 16-bit integer element addition in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks.

Description: This instruction subtracts the 16-bit integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit integer element in [31:16] of 32-bit chunks in Rs1, and writes the result to [31:16] of 32-bit chunks in Rd; at the same time, it adds the 16-bit integer element in [31:16] of 32-bit chunks in Rs2 with the 16-bit integer element in [15:0] of 32-bit chunks in Rs1, and writes the result to [15:0] of 32-bit chunks in Rd.

Operations:

```
Rd.W[x][31:16] = Rs1.W[x][31:16] - Rs2.W[x][15:0];  
Rd.W[x][15:0] = Rs1.W[x][15:0] + Rs2.W[x][31:16];  
for RV32, x=0  
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned operations.

Intrinsic functions:

```
unsigned long __nds__crsa16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_ursa16(uint16x2_t a, uint16x2_t b);  
int16x2_t __nds__v_srsa16(int16x2_t a, int16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_ursa16(uint16x4_t a, uint16x4_t b);  
int16x4_t __nds__v_srsa16(int16x4_t a, int16x4_t b);
```

4.22. INSB (Insert Byte)

Type: DSP

Format:

31 25	24 23	22	21 20	19 15	14 12	11 7	6 0
ONEOP 1010110	INSB 00	imm[2]	imm[1:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
(RV32) INSB Rd, Rs1, imm[1:0]  
(RV64) INSB Rd, Rs1, imm[2:0]
```

Purpose: Insert byte 0 of a 32-bit or 64-bit register into one of the byte elements of another register.

Description: This instruction inserts byte 0 of Rs1 into byte “imm[1:0]” (RV32) or “imm[2:0]” (RV64) of Rd.

Operations:

```
bpos = imm[1:0]; (RV32)  
bpos = imm[2:0]; (RV64)  
Rd.B[bpos] = Rs1.B[0]
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__insb(unsigned long t, unsigned long a, unsigned long b);
```

4.23. KABS8 (SIMD 8-bit Saturating Absolute)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP 1010110	KABS8 10000	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KABS8 Rd, Rs1
```

Purpose: Get the absolute value of 8-bit signed integer elements simultaneously.

Description: This instruction calculates the absolute value of 8-bit signed integer elements stored in Rs1 and writes the element results to Rd. If the input number is 0x80, this instruction generates 0x7f as the output and sets the OV bit to 1.

Operations:

```
src = Rs1.B[x];
if (src == 0x80) {
    src = 0x7f;
    OV = 1;
} else if (src[7] == 1)
    src = -src;
}
Rd.B[x] = src;
for RV32: x=3...0,
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__kabs8(unsigned long a);
```

RV32:

```
int8x4_t __nds__v_kabs8(int8x4_t a);
```

RV64:

```
int8x8_t __nds__v_kabs8(int8x8_t a);
```

4.24. KABS16 (SIMD 16-bit Saturating Absolute)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP 1010110	KABS16 10001	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KABS16 Rd, Rs1
```

Purpose: Get the absolute value of 16-bit signed integer elements simultaneously.

Description: This instruction calculates the absolute value of 16-bit signed integer elements stored in Rs1 and writes the element results to Rd. If the input number is 0x8000, this instruction generates 0x7fff as the output and sets the OV bit to 1.

Operations:

```
src = Rs1.H[x];
if (src == 0x8000) {
    src = 0x7fff;
    OV = 1;
} else if (src[15] == 1)
    src = -src;
}
Rd.H[x] = src;
for RV32: x=1...0,
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__kabs16(unsigned long a);
```

RV32:

```
int16x2_t __nds__v_kabs16(int16x2_t a);
```

RV64:

```
int16x4_t __nds__v_kabs16(int16x4_t a);
```

4.25. KABSW (Scalar 32-bit Absolute Value with Saturation)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP 1010110	KABSW 10100	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KABSW Rd, Rs1
```

Purpose: Get the absolute value of a signed 32-bit integer in a general register.

Description: This instruction calculates the absolute value of a signed 32-bit integer stored in Rs1. The result is sign-extended (for RV64) and written to Rd. This instruction with the minimum negative integer input of 0x80000000 will produce a saturated output of maximum positive integer of 0x7fffffff and the OV flag will be set to 1.

Operations:

```
if (Rs1.W[0] >= 0) {  
    res = Rs1.W[0];  
} else {  
    If (Rs1.W[0] == 0x80000000) {  
        res = 0x7fffffff;  
        OV = 1;  
    } else {  
        res = -Rs1.W[0];  
    }  
}  
Rd = SE32(res);
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__kabsw(signed long a);
```

4.26. KADD8 (SIMD 8-bit Signed Saturating Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KADD8 0001100	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KADD8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit signed integer element saturating additions simultaneously.

Description: This instruction adds the 8-bit signed integer elements in Rs1 with the 8-bit signed integer elements in Rs2. If any of the results are beyond the Q7 number range ($-2^7 \leq Q7 \leq 2^7 - 1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.B[x] + Rs2.B[x];
if (res[x] > 127) {
    res[x] = 127;
    OV = 1;
} else if (res[x] < -128) {
    res[x] = -128;
    OV = 1;
}
Rd.B[x] = res[x];
for RV32: x=3...0,
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__kadd8(unsigned long a, unsigned long b);
```

RV32:

```
int8x4_t __nds__v_kadd8(int8x4_t a, int8x4_t b);
```

RV64:

```
int8x8_t __nds__v_kadd8(int8x8_t a, int8x8_t b);
```

4.27. KADD16 (SIMD 16-bit Signed Saturating Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KADD16 0001000	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KADD16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element saturating additions simultaneously.

Description: This instruction adds the 16-bit signed integer elements in Rs1 with the 16-bit signed integer elements in Rs2. If any of the results are beyond the Q15 number range ($-2^{15} \leq Q15 \leq 2^{15}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.H[x] + Rs2.H[x];
if (res[x] > 32767) {
    res[x] = 32767;
    OV = 1;
} else if (res[x] < -32768) {
    res[x] = -32768;
    OV = 1;
}
Rd.H[x] = res[x];
for RV32: x=1…0,
for RV64: x=3…0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__kadd16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds__v_kadd16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __nds__v_kadd16(int16x4_t a, int16x4_t b);
```

4.28. KADD64 (64-bit Signed Saturating Addition)

Type: DSP (64-bit Profile)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
KADD64 1001000		Rs2		Rs1		001		Rd		GE80B 1111111	

Syntax:

```
KADD64 Rd, Rs1, Rs2
```

Purpose: Add two 64-bit signed integers. The result is saturated to the Q63 range.

RV32 Description: This instruction adds the 64-bit signed integer of an even/odd pair of registers specified by Rs1(4,1) with the 64-bit signed integer of an even/odd pair of registers specified by Rs2(4,1). If the 64-bit result is beyond the Q63 number range ($-2^{63} \leq Q63 \leq 2^{63}-1$), it is saturated to the range and the OV bit is set to 1. The saturated result is written to an even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., value d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction adds the 64-bit signed integer in Rs1 with the 64-bit signed integer in Rs2. If the result is beyond the Q63 number range ($-2^{63} \leq Q63 \leq 2^{63}-1$), it is saturated to the range and the OV bit is set to 1. The saturated result is written to Rd.

Operations:

```
RV32:  
t_L = CONCAT(Rd(4,1),1\b0); t_H = CONCAT(Rd(4,1),1\b1);  
a_L = CONCAT(Rs1(4,1),1\b0); a_H = CONCAT(Rs1(4,1),1\b1);  
b_L = CONCAT(Rs2(4,1),1\b0); b_H = CONCAT(Rs2(4,1),1\b1);  
result = R[a_H].R[a_L] + R[b_H].R[b_L];  
if (result > (2^63)-1) {  
    result = (2^63)-1; OV = 1;  
} else if (result < -2^63) {  
    result = -2^63; OV = 1;  
}  
R[t_H].R[t_L] = result;
```

RV64:

```
result = Rs1 + Rs2;
if (result > (2^63)-1) {
    result = (2^63)-1; OV = 1;
} else if (result < -2^63) {
    result = -2^63; OV = 1;
}
Rd = result;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long long __nds__kadd64(long long a, long long b);
```

4.29. KADDH (Signed Addition with Q15 Saturation)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KADDH 0000010	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KADDH Rd, Rs1, Rs2
```

Purpose: Add the signed lower 32-bit content of two registers with Q15 saturation.

Description: The signed lower 32-bit content of Rs1 is added with the signed lower 32-bit content of Rs2. And the result is saturated to the 16-bit signed integer range of $[-2^{15}, 2^{15}-1]$ and then sign-extended and written to Rd. If saturation happens, this instruction sets the OV flag.

Operations:

```
tmp = Rs1.W[0] + Rs2.W[0];
if (tmp > 32767) {
    res = 32767;
    OV = 1;
} else if (tmp < -32768) {
    res = -32768;
    OV = 1
} else {
    res = tmp;
}
Rd = SE(tmp[15:0]);
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long __nds__kaddh(int a, int b);
```

4.30. KADDW (Signed Addition with Q31 Saturation)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KADDW 0000000	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KADDW Rd, Rs1, Rs2
```

Purpose: Add the lower 32-bit signed content of two registers with Q31 saturation.

Description: The lower 32-bit signed content of Rs1 is added with the lower 32-bit signed content of Rs2. And the result is saturated to the 32-bit signed integer range of $[-2^{31}, 2^{31}-1]$ and then sign-extended and written to Rd. If saturation happens, this instruction sets the OV flag.

Operations:

```
tmp = Rs1.W[0] + Rs2.W[0];
if (tmp > (2^31)-1) {
    res = (2^31)-1;
    OV = 1;
} else if (tmp < -2^31) {
    res = -2^31;
    OV = 1
} else {
    res = tmp;
}
Rd = res[31:0]; // RV32
Rd = SE(res[31:0]) // RV64
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long __nds__kaddw(int a, int b);
```

4.31. KCRAS16 (SIMD 16-bit Signed Saturating Cross Addition & Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KCRAS16 0001010	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KCRAS16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element saturating addition and 16-bit signed integer element saturating subtraction in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks.

Description: This instruction adds the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2; at the same time, it subtracts the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs1. If any of the results are beyond the Q15 number range ($-2^{15} \leq Q15 \leq 2^{15}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for addition and [15:0] of 32-bit chunks in Rd for subtraction.

Operations:

```
res1 = Rs1.W[x][31:16] + Rs2.W[x][15:0];
res2 = Rs1.W[x][15:0] - Rs2.W[x][31:16];
for (res in [res1, res2]) {
    if (res > (2^15)-1) {
        res = (2^15)-1;
        OV = 1;
    } else if (res < -2^15) {
        res = -2^15;
        OV = 1;
    }
}
Rd.W[x][31:16] = res1;
Rd.W[x][15:0] = res2;
for RV32, x=0
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__kcras16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds__v_kcras16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __nds__v_kcras16(int16x4_t a, int16x4_t b);
```

4.32. KCRSA16 (SIMD 16-bit Signed Saturating Cross Subtraction & Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KCRSA16 0001011	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KCRSA16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element saturating subtraction and 16-bit signed integer element saturating addition in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks.

Description: This instruction subtracts the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1; at the same time, it adds the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2 with the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs1. If any of the results are beyond the Q15 number range ($-2^{15} \leq Q15 \leq 2^{15}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for subtraction and [15:0] of 32-bit chunks in Rd for addition.

Operations:

```
res1 = Rs1.W[x][31:16] - Rs2.W[x][15:0];
res2 = Rs1.W[x][15:0] + Rs2.W[x][31:16];
for (res in [res1, res2]) {
    if (res > (2^15)-1) {
        res = (2^15)-1;
        OV = 1;
    } else if (res < -2^15) {
        res = -2^15;
        OV = 1;
    }
}
Rd.W[x][31:16] = res1;
Rd.W[x][15:0] = res2;
for RV32, x=0
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__kcrsa16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds__v_kcrsa16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __nds__v_kcrsa16(int16x4_t a, int16x4_t b);
```

4.33. KDMBB, KDMBT, KDMTT

4.33.1. KDMBB (Signed Saturating Double Multiply B16 x B16)

4.33.2. KDMBT (Signed Saturating Double Multiply B16 x T16)

4.33.3. KDMTT (Signed Saturating Double Multiply T16 x T16)

Type: DSP

Format:

KDMBB

31 25	24 20	19 15	14 12	11 7	6 0
KDMBB 0000101	Rs2	Rs1	001	Rd	GE80B 1111111

KDMBT

31 25	24 20	19 15	14 12	11 7	6 0
KDMBT 0001101	Rs2	Rs1	001	Rd	GE80B 1111111

KDMTT

31 25	24 20	19 15	14 12	11 7	6 0
KDMTT 0010101	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

KDMxy Rd, Rs1, Rs2 (xy = BB, BT, TT)

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the lower 32-bit chunk in registers and then double and saturate the Q31 result. The result is written into the destination register for RV32 or sign-extended to 64-bits and written into the destination register for RV64. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs1 with the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs2. The Q30 result is then doubled and saturated into a Q31 value. The Q31 value is then written into Rd (sign-extended in RV64). When both the two Q15 inputs are 0x8000, saturation will happen. The result will be saturated to 0x7FFFFFFF and the overflow flag OV will be set.

Operations:

```

aop = Rs1.H[0]; bop = Rs2.H[0]; // KDMBB
aop = Rs1.H[0]; bop = Rs2.H[1]; // KDMBT
aop = Rs1.H[1]; bop = Rs2.H[1]; // KDMTT
If (0x8000 != aop | 0x8000 != bop) {
    Mresult = aop * bop;
    resQ31 = Mresult << 1;
    Rd = resQ31; // RV32
    Rd = SE(resQ31); // RV64
} else {
    resQ31 = 0x7FFFFFFF;
    Rd = resQ31; // RV32
    Rd = SE(resQ31); // RV64
    OV = 1;
}

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KDMBB

```
long __nds__kdmrb(unsiged int a, unsiged int b);
```

RV32:

```
long __nds__v_kdmrb(int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds__v_kdmrb(int16x4_t a, int16x4_t b);
```

- KDMBT

```
long __nds__kdmbt(unsiged int a, unsiged int b);
```

RV32:

```
long __nds__v_kdmbt(int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds__v_kdmbt(int16x4_t a, int16x4_t b);
```

- KDMTT

```
long __nds__kdmrr(unsiged int a, unsiged int b);
```

RV32:

```
long __nds__v_kdmtt(int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds__v_kdmtt(int16x4_t a, int16x4_t b);
```

4.34. KDMABB, KDMABT, KDMATT

4.34.1. KDMABB (Signed Saturating Double Multiply Addition B16 x B16)

4.34.2. KDMABT (Signed Saturating Double Multiply Addition B16 x T16)

4.34.3. KDMATT (Signed Saturating Double Multiply Addition T16 x T16)

Type: DSP

Format:

KDMABB

31 25	24 20	19 15	14 12	11 7	6 0
KDMABB 1101001	Rs2	Rs1	001	Rd	GE80B 1111111

KDMABT

31 25	24 20	19 15	14 12	11 7	6 0
KDMABT 1110001	Rs2	Rs1	001	Rd	GE80B 1111111

KDMATT

31 25	24 20	19 15	14 12	11 7	6 0
KDMATT 1111001	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

KDMAxy Rd, Rs1, Rs2 (xy = BB, BT, TT)

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the lower 32-bit chunk in registers and then double and saturate the Q31 result, add the result with the sign-extended lower 32-bit chunk destination register and write the saturated addition result into the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs1 with the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs2. The Q30 result is then doubled and saturated into a Q31 value. The Q31 value is then added with the content of Rd. If the addition result is beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), it is saturated to the range and the OV flag is set to 1. The result after saturation is written to Rd.

When both the two Q15 inputs are 0x8000, saturation will happen and the overflow flag OV will be

set.

Operations:

```
aop = Rs1.H[0]; bop = Rs2.H[0]; // KDMABB
aop = Rs1.H[0]; bop = Rs2.H[1]; // KDMABT
aop = Rs1.H[1]; bop = Rs2.H[1]; // KDMATT
If (0x8000 != aop | 0x8000 != bop) {
    Mresult = aop * bop;
    resQ31 = Mresult << 1;
} else {
    resQ31 = 0x7FFFFFFF;
    OV = 1;
}
resadd = Rd + resQ31; // RV32
resadd = Rd.W[0] + resQ31; // RV64
if (resadd > (2^31)-1) {
    resadd = (2^31)-1;
    OV = 1;
} else if (resadd < -2^31) {
    resadd = -2^31;
    OV = 1;
}
Rd = resadd; // RV32
Rd = SE(resadd); // RV64
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KDMABB

```
long __nds__kdmabb(long t, unsigned int a, unsigned int b);
```

RV32:

```
long __nds__v_kdmabb(long t, int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds__v_kdmabb(long t, int16x4_t a, int16x4_t b);
```

- KDMABT

```
long __nds__kdmabt(long t, unsigned int a, unsigned int b);
```

RV32:

```
long __nds__v_kdmabt(long t, int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds__v_kdmabt(long t, int16x4_t a, int16x4_t b);
```

- KDMATT

```
long __nds__kdmatt(long t, unsigned int a, unsigned int b);
```

RV32:

```
long __nds__v_kdmatt(long t, int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds__v_kdmatt(long t, int16x4_t a, int16x4_t b);
```

4.35. KHM8, KHMX8

4.35.1. KHM8 (SIMD Signed Saturating Q7 Multiply)

4.35.2. KHMX8 (SIMD Signed Saturating Crossed Q7 Multiply)

Type: SIMD

Format:

KHM8

31 25	24 20	19 15	14 12	11 7	6 0
KHM8 1000111	Rs2	Rs1	000	Rd	GE80B 1111111

KHMX8

31 25	24 20	19 15	14 12	11 7	6 0
KHM8 1001111	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KHM8 Rd, Rs1, Rs2
KHMX8 Rd, Rs1, Rs2
```

Purpose: Do Q7xQ7 element multiplications simultaneously. The Q14 results are then reduced to Q7 numbers again.

Description: For the “KHM8” instruction, multiply the top 8-bit Q7 content of 16-bit chunks in Rs1 with the top 8-bit Q7 content of 16-bit chunks in Rs2. At the same time, multiply the bottom 8-bit Q7 content of 16-bit chunks in Rs1 with the bottom 8-bit Q7 content of 16-bit chunks in Rs2.

For the “KHMX16” instruction, multiply the top 8-bit Q7 content of 16-bit chunks in Rs1 with the bottom 8-bit Q7 content of 16-bit chunks in Rs2. At the same time, multiply the bottom 8-bit Q7 content of 16-bit chunks in Rs1 with the top 8-bit Q7 content of 16-bit chunks in Rs2.

The Q14 results are then right-shifted 7-bits and saturated into Q7 values. The Q7 results are then written into Rd. When both the two Q7 inputs of a multiplication are 0x80, saturation will happen. The result will be saturated to 0x7F and the overflow flag OV will be set.

Operations:

```

if (is_KHM8) {
    op1t = Rs1.B[x+1]; op2t = Rs2.B[x+1]; // top
    op1b = Rs1.B[x]; op2b = Rs2.B[x]; // bottom
} else if (is_KHMX8) {
    op1t = Rs1.H[x+1]; op2t = Rs2.H[x]; // Rs1 top
    op1b = Rs1.H[x]; op2b = Rs2.H[x+1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
    if (0x80 != aop | 0x80 != bop) {
        res = (aop s* bop) >> 7;
    } else {
        res= 0x7F;
        OV = 1;
    }
}
Rd.H[x/2] = concat(rest, resb);
for RV32, x=0,2
for RV64, x=0,2,4,6

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KHM8

```
unsigned long __nds__khm8(unsigned long a, unsigned long b);
```

RV32:

```
int8x4_t __nds__v_khm8(int8x4_t a, int8x4_t b);
```

RV64:

```
int8x8_t __nds__v_khm8(int8x8_t a, int8x8_t b);
```

- KHMX8

```
unsigned long __nds__khmx8(unsigned long a, unsigned long b);
```

RV32:

```
int8x4_t __nds__v_khmx8(int8x4_t a, int8x4_t b);
```

RV64:

```
int8x8_t __nds__v_khmx8(int8x8_t a, int8x8_t b);
```

4.36. KHM16, KHMX16

4.36.1. KHM16 (SIMD Signed Saturating Q15 Multiply)

4.36.2. KHMX16 (SIMD Signed Saturating Crossed Q15 Multiply)

Type: SIMD

Format:

KHM16

31 25	24 20	19 15	14 12	11 7	6 0
KHM16 1000011	Rs2	Rs1	000	Rd	GE80B 1111111

KHMX16

31 25	24 20	19 15	14 12	11 7	6 0
KHM16 1001011	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KHM16 Rd, Rs1, Rs2
KHMX16 Rd, Rs1, Rs2
```

Purpose: Do Q15xQ15 element multiplications simultaneously. The Q30 results are then reduced to Q15 numbers again.

Description: For the “KHM16” instruction, multiply the top 16-bit Q15 content of 32-bit chunks in Rs1 with the top 16-bit Q15 content of 32-bit chunks in Rs2. At the same time, multiply the bottom 16-bit Q15 content of 32-bit chunks in Rs1 with the bottom 16-bit Q15 content of 32-bit chunks in Rs2.

For the “KHMX16” instruction, multiply the top 16-bit Q15 content of 32-bit chunks in Rs1 with the bottom 16-bit Q15 content of 32-bit chunks in Rs2. At the same time, multiply the bottom 16-bit Q15 content of 32-bit chunks in Rs1 with the top 16-bit Q15 content of 32-bit chunks in Rs2.

The Q30 results are then right-shifted 15-bits and saturated into Q15 values. The Q15 results are then written into Rd. When both the two Q15 inputs of a multiplication are 0x8000, saturation will happen. The result will be saturated to 0x7FFF and the overflow flag OV will be set.

Operations:

```

if (is KHM16) {
    op1t = Rs1.H[x+1]; op2t = Rs2.H[x+1]; // top
    op1b = Rs1.H[x]; op2b = Rs2.H[x]; // bottom
} else if (is KHMX16) {
    op1t = Rs1.H[x+1]; op2t = Rs2.H[x]; // Rs1 top
    op1b = Rs1.H[x]; op2b = Rs2.H[x+1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
    if (0x8000 != aop | 0x8000 != bop) {
        res = (aop s* bop) >> 15;
    } else {
        res= 0xFFFF;
        OV = 1;
    }
}
Rd.W[x/2] = concat(rest, resb);
for RV32: x=0
for RV64: x=0,2

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KHM16

```
unsigned long __nds__khm6(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds__v_khm16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __nds__v_khm16(int16x4_t a, int16x4_t b);
```

- KHMX16

```
unsigned long __nds__khmx16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds__v_khmx16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __nds__v_khmx16(int16x4_t a, int16x4_t b);
```

4.37. KHMBB, KHMBT, KHMTT

4.37.1. KHMBB (Signed Saturating Half Multiply B16 x B16)

4.37.2. KHMBT (Signed Saturating Half Multiply B16 x T16)

4.37.3. KHMTT (Signed Saturating Half Multiply T16 x T16)

Type: DSP

Format:

KHMBB

31 25	24 20	19 15	14 12	11 7	6 0
KHMBB 0000110	Rs2	Rs1	001	Rd	GE80B 1111111

KHMBT

31 25	24 20	19 15	14 12	11 7	6 0
KHMBT 0001110	Rs2	Rs1	001	Rd	GE80B 1111111

KHMTT

31 25	24 20	19 15	14 12	11 7	6 0
KHMTT 0010110	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

KHMxy Rd, Rs1, Rs2 (xy = BB, BT, TT)

Purpose: Multiply the signed Q15 number contents of two 16-bit data in the corresponding portion of the lower 32-bit chunk in registers and then right-shift 15 bits to turn the Q30 result into a Q15 number again and saturate the Q15 result into the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs1 with the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs2. The Q30 result is then right-shifted 15-bits and saturated into a Q15 value. The Q15 value is then sign-extended and written into Rd. When both the two Q15 inputs are 0x8000, saturation will happen. The result will be saturated to 0x7FFF and the overflow flag OV will be set.

Operations:

```

aop = Rs1.H[0]; bop = Rs2.H[0]; // KHMKB
aop = Rs1.H[0]; bop = Rs2.H[1]; // KHMKT
aop = Rs1.H[1]; bop = Rs2.H[1]; // KHMTR
If (0x8000 != aop | 0x8000 != bop) {
    Mresult[31:0] = aop * bop;
    res[15:0] = Mresult[30:15];
} else {
    res[15:0] = 0x7FFF;
    OV = 1;
}
Rd = SE32(res[15:0]); // Rv32
Rd = SE64(res[15:0]); // RV64

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KHMKB

```
long __nds__khmbb(unsigned int a, unsigned int b);
```

RV32:

```
long __nds__v_khmbb(int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds__v_khmbb(int16x4_t a, int16x4_t b);
```

- KHMKT

```
long __nds__khmbt(unsigned int a, unsigned int b);
```

RV32:

```
long __nds__v_khmbt(int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds__v_khmbt(int16x4_t a, int16x4_t b);
```

- KHMTR

```
long __nds__khmtt(unsigned int a, unsigned int b);
```

RV32:

```
long __nds__v_khmtt(int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds__v_khmtt(int16x4_t a, int16x4_t b);
```

4.38. KMABB, KMABT, KMATT

4.38.1. KMABB (SIMD Saturating Signed Multiply Bottom Halfs & Add)

4.38.2. KMABT (SIMD Saturating Signed Multiply Bottom & Top Halfs & Add)

4.38.3. KMATT (SIMD Saturating Signed Multiply Top Halfs & Add)

Type: SIMD

Format:

KMABB

31 25	24 20	19 15	14 12	11 7	6 0
KMABB 0101101	Rs2	Rs1	001	Rd	GE80B 1111111

KMABT

31 25	24 20	19 15	14 12	11 7	6 0
KMABT 0110101	Rs2	Rs1	001	Rd	GE80B 1111111

KMATT

31 25	24 20	19 15	14 12	11 7	6 0
KMATT 0111101	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

KMABB Rd, Rs1, Rs2

KMABT Rd, Rs1, Rs2

KMATT Rd, Rs1, Rs2

Purpose: Multiply the signed 16-bit content of 32-bit elements in a register with the 16-bit content of 32-bit elements in another register and add the result to the content of 32-bit elements in the third register. The addition result may be saturated and is written to the third register.

- KMABB: rd.W[x] + bottom*bottom (per 32-bit element)
- KMABT rd.W[x] + bottom*top (per 32-bit element)
- KMATT rd.W[x] + top*top (per 32-bit element)

Description:

For the “KMABB” instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2.

For the “KMABT” instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2.

For the “KMATT” instruction, it multiplies the top 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2.

The multiplication result is added to the content of 32-bit elements in Rd. If the addition result is beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res[x] = Rd.W[x] + (Rs1.W[x].H[0] * Rs2.W[x].H[0]); // KMABB
res[x] = Rd.W[x] + (Rs1.W[x].H[0] * Rs2.W[x].H[1]); // KMABT
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[1]); // KMATT
if (res[x] > (2^31)-1) {
    res[x] = (2^31)-1;
    OV = 1;
} else if (res[x] < -2^31) {
    res[x] = -2^31;
    OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMABB

```
long __nds__kmabb(long t, unsigned long a, unsigned long b);
```

RV32:

```
long __nds__v_kmabb(long t, int16x2_t a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmabb(int32x2_t t, int16x4_t a, int16x4_t b);
```

- KMABT

```
long __nds__kmabt(long t, unsigned long a, unsigned long b);
```

RV32:

```
long __nds__v_kmabt(long t, int16x2_t a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmabt(int32x2_t t, int16x4_t a, int16x4_t b);
```

- KMATT

```
long ndskmatt(long t, unsigned long a, unsigned long b);
```

RV32:

```
long __nds__v_kmatt(long t, int16x2_t a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmatt(int32x2_t t, int16x4_t a, int16x4_t b);
```

4.39. KMADA, KMAXDA

4.39.1. KMADA (SIMD Saturating Signed Multiply Two Halfs and Two Adds)

4.39.2. KMAXDA (SIMD Saturating Signed Crossed Multiply Two Halfs and Two Adds)

Type: SIMD

Format:

KMADA

31 25	24 20	19 15	14 12	11 7	6 0
KMADA 0100100	Rs2	Rs1	001	Rd	GE80B 1111111

KMAXDA

31 25	24 20	19 15	14 12	11 7	6 0
KMAXDA 0100101	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KMADA Rd, Rs1, Rs2
KMAXDA Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from 32-bit elements in two registers; and then adds the two 32-bit results and 32-bit elements in a third register together. The addition result may be saturated.

- KMADA: rd.W[x] + top*top + bottom*bottom (per 32-bit element)
- KMAXDA: rd.W[x] + top*bottom + bottom*top (per 32-bit element)

Description:

For the “KMADA instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2 and then adds the result to the result of multiplying the top 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2.

For the “KMAXDA” instruction, it multiplies the top 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2 and then adds the result to the result of multiplying the bottom 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2.

The result is added to the content of 32-bit elements in Rd. If the addition result is beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), it is saturated to the range and the OV bit is set to 1. The 32-bit results after saturation are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
// KMADA
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[1]) + (Rs1.W[x].H[0] * Rs2.W[x].H[0]);
// KMAXDA
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[0]) + (Rs1.W[x].H[0] * Rs2.W[x].H[1]);
if (res[x] > (2^31)-1) {
    res[x] = (2^31)-1;
    OV = 1;
} else if (res[x] < -2^31) {
    res[x] = -2^31;
OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMADA

```
long __nds__kmada(long t, unsigned long a, unsigned long b);
```

RV32:

```
long __nds__v_kmada(long t, int16x2_t a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmada(int32x2_t t, int16x4_t a, int16x4_t b);
```

- KMAXDA

```
long __nds__kmaxda(long t, unsigned long a, unsigned long b);
```

RV32:

```
long __nds__v_kmaxda(long t, int16x2_t a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmaxda(int32x2_t t, int16x4_t a, int16x4_t b);
```

4.40. KMADS, KMADRS, KMAXDS

4.40.1. KMADS (SIMD Saturating Signed Multiply Two Halfs & Subtract & Add)

4.40.2. KMADRS (SIMD Saturating Signed Multiply Two Halfs & Reverse Subtract & Add)

4.40.3. KMAXDS (SIMD Saturating Signed Crossed Multiply Two Halfs & Subtract & Add)

Type: SIMD

Format:

KMADS

31 25	24 20	19 15	14 12	11 7	6 0
KMADS 0101110	Rs2	Rs1	001	Rd	GE80B 1111111

KMADRS

31 25	24 20	19 15	14 12	11 7	6 0
KMADRS 0110110	Rs2	Rs1	001	Rd	GE80B 1111111

KMAXDS

31 25	24 20	19 15	14 12	11 7	6 0
KMAXDS 0111110	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KMADS Rd, Rs1, Rs2
KMADRS Rd, Rs1, Rs2
KMAXDS Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from 32-bit elements in two registers; and then perform a subtraction operation between the two 32-bit results. Then add the subtraction result to the corresponding 32-bit elements in a third register. The addition result may be saturated.

- KMADS: rd.W[x] + (top*top - bottom*bottom) (per 32-bit element)
- KMADRS: rd.W[x] + (bottom*bottom - top*top) (per 32-bit element)

- KMAXDS: rd.W[x] + (top*bottom - bottom*top) (per 32-bit element)

Description:

For the “KMADS” instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2.

For the “KMADRS” instruction, it multiplies the top 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2.

For the “KMAXDS” instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2.

The subtraction result is then added to the content of the corresponding 32-bit elements in Rd. If the addition result is beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), it is saturated to the range and the OV bit is set to 1. The 32-bit results after saturation are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
// KMADS
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[1]) - (Rs1.W[x].H[0] * Rs2.W[x].H[0]);
// KMADRS
res[x] = Rd.W[x] + (Rs1.W[x].H[0] * Rs2.W[x].H[0]) - (Rs1.W[x].H[1] * Rs2.W[x].H[1]);
// KMAXDS
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[0]) - (Rs1.W[x].H[0] * Rs2.W[x].H[1]);
if (res[x] > (2^31)-1) {
    res[x] = (2^31)-1;
    OV = 1;
} else if (res[x] < -2^31) {
    res[x] = -2^31;
    OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1…0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMADS

```
long __nds__kmads(long t, unsigned long a, unsigned long b);
```

RV32:

```
long __nds__v_kmads(long t, int16x2_t a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmads(int32x2_t t, int16x4_t a, int16x4_t b);
```

- KMADRS

```
long __nds__kmadrs(long t, unsigned long a, unsigned long b);
```

RV32:

```
long __nds__v_kmadrs(long t, int16x2_t a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmadrs(int32x2_t t, int16x4_t a, int16x4_t b);
```

- KMAXDS

```
long __nds__kmaxds(long t, unsigned long a, unsigned long b);
```

RV32:

```
long __nds__v_kmaxds(long t, int16x2_t a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmaxds(int32x2_t t, int16x4_t a, int16x4_t b);
```

4.41. KMAR64 (Signed Multiply and Saturating Add to 64-Bit Data)

Type: DSP (64-bit Profile)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KMAR64 1001010	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KMAR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit signed elements in two registers and add the 64-bit multiplication results to the 64-bit signed data of a pair of registers (RV32) or a register (RV64). The result is saturated to the Q63 range and written back to the pair of registers (RV32) or the register (RV64).

RV32 Description: This instruction multiplies the 32-bit signed data of Rs1 with that of Rs2. It adds the 64-bit multiplication result to the 64-bit signed data of an even/odd pair of registers specified by Rd(4,1) with unlimited precision. If the 64-bit addition result is beyond the Q63 number range ($-2^{63} \leq Q63 \leq 2^{63}-1$), it is saturated to the range and the OV bit is set to 1. The saturated result is written back to the even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., value d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction multiplies the 32-bit signed elements of Rs1 with that of Rs2. It adds the 64-bit multiplication results to the 64-bit signed data of Rd with unlimited precision. If the 64-bit addition result is beyond the Q63 number range ($-2^{63} \leq Q63 \leq 2^{63}-1$), it is saturated to the range and the OV bit is set to 1. The saturated result is written back to Rd.

Operations:

RV32:

```

t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
result = R[t_H].R[t_L] + (Rs1 * Rs2);
if (result > (2^63)-1) {
    result = (2^63)-1; OV = 1;
} else if (result < -2^63) {
    result = -2^63; OV = 1;
}
R[t_H].R[t_L] = result;

```

RV64:

```

// result has unlimited precision
result = Rd + (Rs1.W[0] * Rs2.W[0]) + (Rs1.W[1] * Rs2.W[1]);
if (result > (2^63)-1) {
    result = (2^63)-1; OV = 1;
} else if (result < -2^63) {
    result = -2^63; OV = 1;
}
Rd = result;

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long long __nds__kmar64(long long t, long a, long b);
```

RV64:

```
long long __nds__v_kmar64(long long t, int32x2_t a, int32x2_t b);
```

4.42. KMDA, KMXDA

4.42.1. KMDA (SIMD Signed Multiply Two Halfs and Add)

4.42.2. KMXDA (SIMD Signed Crossed Multiply Two Halfs and Add)

Type: SIMD

Format:

KMDA

31 25	24 20	19 15	14 12	11 7	6 0
KMDA 0011100	Rs2	Rs1	001	Rd	GE80B 1111111

KMXDA

31 25	24 20	19 15	14 12	11 7	6 0
KMXDA 0011101	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KMDA Rd, Rs1, Rs2
KMXDA Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then adds the two 32-bit results together. The addition result may be saturated.

- KMDA: top*top + bottom*bottom (per 32-bit element)
- KMXDA: top*bottom + bottom*top (per 32-bit element)

Description:

For the “KMDA” instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then adds the result to the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2.

For the “KMXDA” instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then adds the result to the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2.

The addition result is checked for saturation. If saturation happens, the result is saturated to $2^{31}-1$. The final results are written to Rd. The 16-bit contents are treated as signed integers.

Operations:

```
if Rs1.W[x] != 0x80008000) or (Rs2.W[x] != 0x80008000 { // KMDA Rd.W[x] = Rs1.W[x].H[1] *  
Rs2.W[x].H[1]) + (Rs1.W[x].H[0] * Rs2.W[x].H[0]; // KMXDA Rd.W[x] = Rs1.W[x].H[1] * Rs2.W[x].H[0])  
+ (Rs1.W[x].H[0] * Rs2.W[x].H[1]; } else { Rd.W[x] = 0x7fffffff; OV = 1; } for RV32: x=0 for RV64:  
x=1...0
```

Exceptions: None

Privilege level: All

Note:

- Usage domain: Complex, Statistics, Transform

Intrinsic functions:

- KMDA

```
long __nds__kmda(unsigned long a, unsigned long b);
```

RV32:

```
long __nds__v_kmda(int16x2_t a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmda(int16x4_t a, int16x4_t b);
```

- KMXDA

```
long __nds__kmxda(unsigned long a, unsigned long b);
```

RV32:

```
long __nds__v_kmxda(int16x2_t a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmxda(int16x4_t a, int16x4_t b);
```

4.43. KMMAC, KMMAC.u

4.43.1. KMMAC (SIMD Saturating MSW Signed Multiply Word and Add)

4.43.2. KMMAC.u (SIMD Saturating MSW Signed Multiply Word and Add with Rounding)

Type: SIMD

Format:

KMMAC

31 25	24 20	19 15	14 12	11 7	6 0
KMMAC 0110000	Rs2	Rs1	001	Rd	GE80B 1111111

KMMAC.u

31 25	24 20	19 15	14 12	11 7	6 0
KMMAC.u 0111000	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KMMAC Rd, Rs1, Rs2
KMMAC.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of two registers and add the most significant 32-bit results with the signed 32-bit integer elements of a third register. The addition results are saturated first and then written back to the third register. The “.u” form performs an additional rounding up operation on the multiplication results before adding the most significant 32-bit part of the results.

Description:

This instruction multiplies the signed 32-bit elements of Rs1 with the signed 32-bit elements of Rs2 and adds the most significant 32-bit multiplication results with the signed 32-bit elements of Rd. If the addition result is beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to Rd. The “.u” form of the instruction additionally rounds up the most significant 32-bit of the 64-bit multiplication results by adding a 1 to bit 31 of the results.

Operations:

```

Mres[x][63:0] = Rs1.W[x] * Rs2.W[x];
if (!.u form) {
    Round[x][32:0] = Mres[x][63:31] + 1;
    res[x] = Rd.W[x] + Round[x][32:1];
} else {
    res[x] = Rd.W[x] + Mres[x][63:32];
}
if (res[x] > (2^31)-1) {
    res[x] = (2^31)-1;
    OV = 1;
} else if (res[x] < -2^31) {
    res[x] = -2^31;
    OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1…0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMMAC

```
long __nds__kmmac(long t, long a, long b);
```

RV64:

```
int32x2_t __nds__v_kmmac(int32x2_t t, int32x2_t a, int32x2_t b);
```

- KMMAC.u

```
long __nds__kmmac_u(long t, long a, long b);
```

RV64:

```
int32x2_t __nds__v_kmmac_u(int32x2_t t, int32x2_t a, int32x2_t b);
```

4.44. KMMAWB, KMMAWB.u

4.44.1. KMMAWB (SIMD Saturating MSW Signed Multiply Word and Bottom Half and Add)

4.44.2. KMMAWB.u (SIMD Saturating MSW Signed Multiply Word and Bottom Half and Add with Rounding)

Type: SIMD

Format:

KMMAWB

31 25	24 20	19 15	14 12	11 7	6 0
KMMAWB 0100011	Rs2	Rs1	001	Rd	GE80B 1111111

KMMAWB.u

31 25	24 20	19 15	14 12	11 7	6 0
KMMAWB.u 0101011	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KMMAWB Rd, Rs1, Rs2
KMMAWB.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of one register and the bottom 16-bit of the corresponding 32-bit elements of another register and add the most significant 32-bit results with the corresponding signed 32-bit elements of a third register. The addition result is written to the corresponding 32-bit elements of the third register. The “.u” form rounds up the multiplication results from the most significant discarded bit before the addition operations.

Description:

This instruction multiplies the signed 32-bit elements of Rs1 with the signed bottom 16-bit content of the corresponding 32-bit elements of Rs2 and adds the most significant 32-bit multiplication results with the corresponding signed 32-bit elements of Rd. If the addition result is beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to the corresponding 32-bit elements of Rd. The “.u” form of the instruction rounds up the most significant 32-bit of the 48-bit multiplication results by adding a 1 to bit 15 of the result before the addition operations.

Operations:

```

Mres[x][47:0] = Rs1.W[x] * Rs2.W[x].H[0];
if (!.u[] form) {
    Round[x][32:0] = Mres[x][47:15] + 1;
    res[x] = Rd.W[x] + Round[x][32:1];
} else {
    res[x] = Rd.W[x] + Mres[x][47:16];
}
if (res[x] > (2^31)-1) {
    res[x] = (2^31)-1;
    OV = 1;
} else if (res[x] < -2^31) {
    res[x] = -2^31;
    OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1…0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMMAWB

```
long __nds__kmmawb(long t, unsigned long a, unsigned long b);
```

RV32:

```
int __nds__v_kmmawb(int t, int a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmmawb(int32x2_t t, int32x2_t a, int16x4_t b);
```

- KMMAWB.u

```
long __nds__kmmawb_u(long t, unsigned long a, unsigned long b);
```

RV32:

```
int __nds__v_kmmawb_u(int t, int a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmmawb_u(int32x2_t t, int32x2_t a, int16x4_t b);
```

4.45. KMMAWB2, KMMAWB2.u

4.45.1. KMMAWB2 (SIMD Saturating MSW Signed Multiply Word and Bottom Half & 2 and Add)

4.45.2. KMMAWB2.u (SIMD Saturating MSW Signed Multiply Word and Bottom Half & 2 and Add with Rounding)

Type: SIMD

Format:

KMMAWB2

31 25	24 20	19 15	14 12	11 7	6 0
KMMAWB2 1100111	Rs2	Rs1	001	Rd	GE80B 1111111

KMMAWB2.u

31 25	24 20	19 15	14 12	11 7	6 0
KMMAWB2.u 1101111	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KMMAWB2 Rd, Rs1, Rs2
KMMAWB2.u Rd, Rs1, Rs2
```

Purpose: **Multiply** the signed 32-bit elements of one register and the bottom 16-bit of the corresponding 32-bit elements of another register, **double** the multiplication results and add the **saturated** most significant 32-bit results with the corresponding signed 32-bit elements of a third register. The **saturated** addition result is written to the corresponding 32-bit elements of the third register. The “.u” form rounds up the multiplication results from the most significant discarded bit before the addition operations.

Description:

This instruction multiplies the signed 32-bit Q31 elements of Rs1 with the signed bottom 16-bit Q15 content of the corresponding 32-bit elements of Rs2, doubles the Q46 results to Q47 numbers and adds the saturated most significant 32-bit Q31 multiplication results with the corresponding signed 32-bit elements of Rd. If the addition result is beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to the corresponding 32-bit elements of Rd. The “.u” form of the instruction rounds up the most significant 32-bit of the 48-bit Q47 multiplication results by adding a 1 to bit 15 (i.e., bit 14 before doubling) of the result before the addition operations.

Operations:

```
if ((Rs1.W[x] == 0x80000000) & (Rs2.W[x].H[0] == 0x8000)) {  
    addop.W[x] = 0x7fffffff;  
    OV = 1;  
} else {  
    Mres[x][47:0] = Rs1.W[x] s* Rs2.W[x].H[0];  
    if (!.u form) {  
        Mres[x][47:14] = Mres[x][47:14] + 1;  
    }  
    addop.W[x] = Mres[x][46:15]; // doubling  
}  
res[x] = Rd.W[x] + addop.W[x];  
if (res[x] > (2^31)-1) {  
    res[x] = (2^31)-1;  
    OV = 1;  
} else if (res[x] < -2^31) {  
    res[x] = -2^31;  
    OV = 1;  
}  
Rd.W[x] = res[x];  
for RV32: x=0  
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMMAWB2

```
long __nds__kmmawb2(long t, unsigned long a, unsigned long b);
```

RV32:

```
int __nds_v_kmmawb2(int t, int a, int16x2_t b);
```

RV64:

```
int32x2_t __nds_v_kmmawb2(int32x2_t t, int32x2_t a, int16x4_t b);
```

- KMMAWB2.u

```
long __nds__kmmawb2_u(long t, unsigned long a, unsigned long b);
```

RV32:

```
int __nds__v_kmmawb2_u(int t, int a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmmawb2_u(in32x2_t t, int32x2_t a, int16x4_t b);
```

4.46. KMMAWT, KMMAWT.u

4.46.1. KMMAWT (SIMD Saturating MSW Signed Multiply Word and Top Half and Add)

4.46.2. KMMAWT.u (SIMD Saturating MSW Signed Multiply Word and Top Half and Add with Rounding)

Type: SIMD

Format:

KMMAWT

31 25	24 20	19 15	14 12	11 7	6 0
KMMAWT 0110011	Rs2	Rs1	001	Rd	GE80B 1111111

KMMAWT.u

31 25	24 20	19 15	14 12	11 7	6 0
KMMAWT.u 0111011	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KMMAWT Rd, Rs1, Rs2
KMMAWT.u Rd Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of one register and the signed top 16-bit of the corresponding 32-bit elements of another register and add the most significant 32-bit results with the corresponding signed 32-bit elements of a third register. The addition results are written to the corresponding 32-bit elements of the third register. The “.u” form rounds up the multiplication results from the most significant discarded bit before the addition operations.

Description:

This instruction multiplies the signed 32-bit elements of Rs1 with the signed top 16-bit of the corresponding 32-bit elements of Rs2 and adds the most significant 32-bit multiplication results with the corresponding signed 32-bit elements of Rd. If the addition result is beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to the corresponding 32-bit elements of Rd. The “.u” form of the instruction rounds up the most significant 32-bit of the 48-bit multiplication results by adding a 1 to bit 15 of the result before the addition operations.

Operations:

```

Mres[x][47:0] = Rs1.W[x] * Rs2.W[x].H[1];
if (!.u[] form) {
    Round[x][32:0] = Mres[x][47:15] + 1;
    res[x] = Rd.W[x] + Round[x][32:1];
} else {
    res[x] = Rd.W[x] + Mres[x][47:16];
}
if (res[x] > (2^31)-1) {
    res[x] = (2^31)-1;
    OV = 1;
} else if (res[x] < -2^31) {
    res[x] = -2^31;
    OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1…0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMMAWT

```
long __nds__kmmawt(long t, unsigned long a, unsigned long b);
```

RV32:

```
int __nds__v_kmmawt(int t, int a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmmawt(int32x2_t t, int32x2_t a, int16x4_t b);
```

- KMMAWT.u

```
long __nds__kmmawt_u(long t, unsigned long a, unsigned long b);
```

RV32:

```
int __nds__v_kmmawt_u(int t, int a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmmawt_u(int32x2_t t, int32x2_t a, int16x4_t b);
```

4.47. KMMAWT2, KMMAWT2.u

4.47.1. KMMAWT2 (SIMD Saturating MSW Signed Multiply Word and Top Half & 2 and Add)

4.47.2. KMMAWT2.u (SIMD Saturating MSW Signed Multiply Word and Top Half & 2 and Add with Rounding)

Type: SIMD

Format:

KMMAWT2

31 25	24 20	19 15	14 12	11 7	6 0
KMMAWT2 1110111	Rs2	Rs1	001	Rd	GE80B 1111111

KMMAWT2.u

31 25	24 20	19 15	14 12	11 7	6 0
KMMAWT2.u 1111111	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KMMAWT2 Rd, Rs1, Rs2
KMMAWT2.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit elements of one register and the top 16-bit of the corresponding 32-bit elements of another register, double the multiplication results and add the saturated most significant 32-bit results with the corresponding signed 32-bit elements of a third register. The saturated addition result is written to the corresponding 32-bit elements of the third register. The “.u” form rounds up the multiplication results from the most significant discarded bit before the addition operations.

Description:

This instruction multiplies the signed 32-bit Q31 elements of Rs1 with the signed top 16-bit Q15 content of the corresponding 32-bit elements of Rs2, doubles the Q46 results to Q47 numbers and adds the saturated most significant 32-bit Q31 multiplication results with the corresponding signed 32-bit elements of Rd. If the addition result is beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to the corresponding 32-bit elements of Rd. The “.u” form of the instruction rounds up the most significant 32-bit of the 48-bit Q47 multiplication results by adding a 1 to bit 15 (i.e., bit 14 before doubling) of the result before the addition operations.

Operations:

```
if ((Rs1.W[x] == 0x80000000) & (Rs2.W[x].H[1] == 0x8000)) {
    addop.W[x] = 0x7fffffff;
    OV = 1;
} else {
    Mres[x][47:0] = Rs1.W[x] s* Rs2.W[x].H[1];
    if (!.u form) {
        Mres[x][47:14] = Mres[x][47:14] + 1;
    }
    addop.W[x] = Mres[x][46:15]; // doubling
}
res[x] = Rd.W[x] + addop.W[x];
if (res[x] > (2^31)-1) {
    res[x] = (2^31)-1;
    OV = 1;
} else if (res[x] < -2^31) {
    res[x] = -2^31;
    OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMMAWT2

```
long __nds__kmmawt2(long t, unsigned long a, unsigned long b);
```

RV32:

```
int __nds_v_kmmawt2(int t, int a, int16x2_t b);
```

RV64:

```
int32x2_t __nds_v_kmmawt2(int32x2_t t, int32x2_t a, int16x4_t b);
```

- KMMAWT2.u

```
long __nds__kmmawt2_u(long t, unsigned long a, unsigned long b);
```

RV32:

```
int __nds__v_kmmawt2_u(int t, int a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmmawt2_u(in32x2_t t, int32x2_t a, int16x4_t b);
```

4.48. KMMSB, KMMSB.u

4.48.1. KMMSB (SIMD Saturating MSW Signed Multiply Word and Subtract)

4.48.2. KMMSB.u (SIMD Saturating MSW Signed Multiply Word and Subtraction with Rounding)

Type: SIMD

Format:

KMMSB

31 25	24 20	19 15	14 12	11 7	6 0
KMMSB 0100001	Rs2	Rs1	001	Rd	GE80B 1111111

KMMSB.u

31 25	24 20	19 15	14 12	11 7	6 0
KMMSB.u 0101001	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KMMSB Rd, Rs1, Rs2
KMMSB.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of two registers and subtract the most significant 32-bit results from the signed 32-bit elements of a third register. The subtraction results are written to the third register. The “.u” form performs an additional rounding up operation on the multiplication results before subtracting the most significant 32-bit part of the results.

Description:

This instruction multiplies the signed 32-bit elements of Rs1 with the signed 32-bit elements of Rs2 and subtracts the most significant 32-bit multiplication results from the signed 32-bit elements of Rd. If the subtraction result is beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to Rd. The “.u” form of the instruction additionally rounds up the most significant 32-bit of the 64-bit multiplication results by adding a 1 to bit 31 of the results.

Operations:

```

Mres[x][63:0] = Rs1.W[x] * Rs2.W[x];
if (!.u[] form) {
    Round[x][32:0] = Mres[x][63:31] + 1;
    res[x] = Rd.W[x] - Round[x][32:1];
} else {
    res[x] = Rd.W[x] - Mres[x][63:32];
}
if (res[x] > (2^31)-1) {
    res[x] = (2^31)-1;
    OV = 1;
} else if (res[x] < -2^31) {
    res[x] = -2^31;
    OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMMSB

```
long __nds__kmmsb(long t, long a, long b);
```

RV64:

```
int32x2_t __nds__v_kmmsb(int32x2_t t, int32x2_t a, int32x2_t b);
```

- KMMSB.u

```
long __nds__kmmsb_u(long t, long a, long b);
```

RV64:

```
int32x2_t __nds__v_kmmsb_u(int32x2_t t, int32x2_t a, int32x2_t b);
```

4.49. KMMWB2, KMMWB2.u

4.49.1. KMMWB2 (SIMD Saturating MSW Signed Multiply Word and Bottom Half & 2)

4.49.2. KMMWB2.u (SIMD Saturating MSW Signed Multiply Word and Bottom Half & 2 with Rounding)

Type: SIMD

Format:

KMMWB2

31 25	24 20	19 15	14 12	11 7	6 0
KMMWB2 1000111	Rs2	Rs1	001	Rd	GE80B 1111111

KMMWB2.u

31 25	24 20	19 15	14 12	11 7	6 0
KMMWB2.u 1001111	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KMMWB2 Rd, Rs1, Rs2
KMMWB2.u Rd, Rs1, Rs2
```

Purpose: **Multiply** the signed 32-bit integer elements of one register and the bottom 16-bit of the corresponding 32-bit elements of another register, **double** the multiplication results and write the **saturated** most significant 32-bit results to the corresponding 32-bit elements of a register. The “.u” form rounds up the results from the most significant discarded bit.

Description:

This instruction multiplies the signed 32-bit Q31 elements of Rs1 with the signed bottom 16-bit Q15 content of the corresponding 32-bit elements of Rs2, doubles the Q46 results to Q47 numbers and writes the saturated most significant 32-bit Q31 multiplication results to the corresponding 32-bit elements of Rd. The “.u” form of the instruction rounds up the most significant 32-bit of the 48-bit Q47 multiplication results by adding a 1 to bit 15 (i.e., bit 14 before doubling) of the results.

Operations:

```

if ((Rs1.W[x] == 0x80000000) & (Rs2.W[x].H[0] == 0x8000)) {
    Rd.W[x] = 0x7fffffff;
    OV = 1;
} else {
    Mres[x][47:0] = Rs1.W[x] s* Rs2.W[x].H[0];
    if (0.u form) {
        Round[x][32:0] = Mres[x][46:14] + 1;
        Rd.W[x] = Round[x][32:1];
    } else {
        Rd.W[x] = Mres[x][46:15];
    }
}
for RV32: x=0
for RV64: x=1...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMMWB2

```
long __nds__kmmwb2(long a, unsigned long b);
```

RV32:

```
int __nds__v_kmmwb2(int a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmmwb2(int32x2_t a, int16x4_t b);
```

- KMMWB2.u

```
long __nds__kmmwb2_u(long a, unsigned long b);
```

RV32:

```
int __nds__v_kmmwb2_u(int a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmmwb2_u(int32x2_t a, int16x4_t b);
```

4.50. KMMWT2, KMMWT2.u

4.50.1. KMMWT2 (SIMD Saturating MSW Signed Multiply Word and Top Half & 2)

4.50.2. KMMWT2.u (SIMD Saturating MSW Signed Multiply Word and Top Half & 2 with Rounding)

Type: SIMD

Format:

KMMWT2

31 25	24 20	19 15	14 12	11 7	6 0
KMMWT2 1010111	Rs2	Rs1	001	Rd	GE80B 1111111

KMMWT2.u

31 25	24 20	19 15	14 12	11 7	6 0
KMMWT2.u 1011111	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KMMWT2 Rd, Rs1, Rs2
KMMWT2.u Rd, Rs1, Rs2
```

Purpose: **Multiply** the signed 32-bit integer elements of one register and the top 16-bit of the corresponding 32-bit elements of another register, **double** the multiplication results and write the **saturated** most significant 32-bit results to the corresponding 32-bit elements of a register. The “.u” form rounds up the results from the most significant discarded bit.

Description:

This instruction multiplies the signed 32-bit Q31 elements of Rs1 with the signed top 16-bit Q15 content of the corresponding 32-bit elements of Rs2, doubles the Q46 results to Q47 numbers and writes the saturated most significant 32-bit Q31 multiplication results to the corresponding 32-bit elements of Rd. The “.u” form of the instruction rounds up the most significant 32-bit of the 48-bit Q47 multiplication results by adding a 1 to bit 15 (i.e., bit 14 before doubling) of the results.

Operations:

```

if ((Rs1.W[x] == 0x80000000) & (Rs2.W[x].H[1] == 0x8000)) {
    Rd.W[x] = 0x7fffffff;
    OV = 1;
} else {
    Mres[x][47:0] = Rs1.W[x] s* Rs2.W[x].H[1];
    if (0.u form) {
        Round[x][32:0] = Mres[x][46:14] + 1;
        Rd.W[x] = Round[x][32:1];
    } else {
        Rd.W[x] = Mres[x][46:15];
    }
}
for RV32: x=0
for RV64: x=1...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMMWT2

```
long __nds__kmmwt2(long a, unsigned long b);
```

RV32:

```
int __nds__v_kmmwt2(int a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmmwt2(int32x2_t a, int16x4_t b);
```

- KMMWT2.u

```
long __nds__kmmwt2_u(long a, unsigned long b);
```

RV32:

```
int __nds__v_kmmwt2_u(int a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmmwt2_u(int32x2_t a, int16x4_t b);
```

4.51. KMSDA, KMSXDA

4.51.1. KMSDA (SIMD Saturating Signed Multiply Two Halfs & Add & Subtract)

4.51.2. KMSXDA (SIMD Saturating Signed Crossed Multiply Two Halfs & Add & Subtract)

Type: SIMD

Format:

KMSDA

31 25	24 20	19 15	14 12	11 7	6 0
KMSDA 0100110	Rs2	Rs1	001	Rd	GE80B 1111111

KMSXDA

31 25	24 20	19 15	14 12	11 7	6 0
KMSXDA 0100111	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KMSDA Rd, Rs1, Rs2
KMSXDA Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then subtracts the two 32-bit results from the corresponding 32-bit elements of a third register. The subtraction result may be saturated.

- KMSDA: rd.W[x] - top*top - bottom*bottom (per 32-bit element)
- KMSXDA: rd.W[x] - top*bottom - bottom*top (per 32-bit element)

Description:

For the “KMSDA” instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2.

For the “KMSXDA” instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and multiplies the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2.

The two 32-bit multiplication results are then subtracted from the content of the corresponding 32-

bit elements of Rd. If the subtraction result is beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to Rd. The 16-bit contents are treated as signed integers.

Operations:

```
// KMSDA
res[x] = Rd.W[x] - (Rs1.W[x].H[1] * Rs2.W[x].H[1]) - (Rs1.W[x].H[0] * Rs2.W[x].H[0]);
// KMSXDA
res[x] = Rd.W[x] - (Rs1.W[x].H[1] * Rs2.W[x].H[0]) - (Rs1.W[x].H[0] * Rs2.W[x].H[1]);
if (res[x] > (2^31)-1) {
    res[x] = (2^31)-1;
    OV = 1;
} else if (res[x] < -2^31) {
    res[x] = -2^31;
    OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1…0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMSDA

```
long __nds__kmsda(long t, unsigned long a, unsigned long b);
```

RV32:

```
int __nds__v_kmsda(int t, int16x2_t a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmsda(int32x2_t t, int16x4_t a, int16x4_t b);
```

- KMSXDA

```
long __nds__kmsxda(long t, unsigned long a, unsigned long b);
```

RV32:

```
int __nds__v_kmsxda(int t, int16x2_t a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_kmsxda(int32x2_t t, int16x4_t a, int16x4_t b);
```

4.52. KMSR64 (Signed Multiply and Saturating Subtract from 64-Bit Data)

Type: DSP (64-bit Profile)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
KMSR64 1001011		Rs2		Rs1		001		Rd		GE80B 1111111	

Syntax:

```
KMSR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit signed elements in two registers and subtract the 64-bit multiplication results from the 64-bit signed data of a pair of registers (RV32) or a register (RV64). The result is saturated to the Q63 range and written back to the pair of registers (RV32) or the register (RV64).

RV32 Description: This instruction multiplies the 32-bit signed data of Rs1 with that of Rs2. It subtracts the 64-bit multiplication result from the 64-bit signed data of an even/odd pair of registers specified by Rd(4,1) with unlimited precision. If the 64-bit subtraction result is beyond the Q63 number range ($-2^{63} \leq Q63 \leq 2^{63}-1$), it is saturated to the range and the OV bit is set to 1. The saturated result is written back to the even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction multiplies the 32-bit signed elements of Rs1 with that of Rs2. It subtracts the 64-bit multiplication results from the 64-bit signed data in Rd with unlimited precision. If the 64-bit subtraction result is beyond the Q63 number range ($-2^{63} \leq Q63 \leq 2^{63}-1$), it is saturated to the range and the OV bit is set to 1. The saturated result is written back to Rd.

Operations:

RV32:

```

t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
result = R[t_H].R[t_L] - (Rs1 * Rs2);
if (result > (2^63)-1) {
    result = (2^63)-1; OV = 1;
} else if (result < -2^63) {
    result = -2^63; OV = 1;
}
R[t_H].R[t_L] = result;

```

RV64:

```

// result has unlimited precision
result = Rd - (Rs1.W[0] * Rs2.W[0]) - (Rs1.W[1] * Rs2.W[1]);
if (result > (2^63)-1) {
    result = (2^63)-1; OV = 1;
} else if (result < -2^63) {
    result = -2^63; OV = 1;
}
Rd = result;

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long long __nds__kmsr64(long long t, long a, long b);
```

RV64:

```
long long __nds__v_kmsr64(long long t, int32x2_t a, int32x2_t b);
```

4.53. KSLLW (Saturating Shift Left Logical for Word)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KSLLW 0010011	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KSLLW Rd, Rs1, Rs2
```

Purpose: Do logical left shift operation with saturation on a 32-bit word. The shift amount is a variable from a GPR.

Description: The first word data in Rs1 is left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the low-order 5-bits of the value in the Rs2 register. Any shifted value greater than $2^{31}-1$ is saturated to $2^{31}-1$. Any shifted value smaller than -2^{31} is saturated to -2^{31} . And the saturated result is sign-extended and written to Rd. If any saturation is performed, set OV bit to 1.

Operations:

```
sa = Rs2[4:0];
res[(31+sa):0] = Rs1.W[0] << sa;
if (res > (2^31)-1) {
    res = 0x7fffffff; OV = 1;
} else if (res < -2^31) {
    res = 0x80000000; OV = 1;
}
Rd[31:0] = res[31:0]; // RV32
Rd[63:0] = SE(res[31:0]); // RV64
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long __nds__ksllw(long a, unsigned int b);
```

4.54. KSLLIW (Saturating Shift Left Logical Immediate for Word)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KSLLIW 0011011	imm5u	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KSLLIW Rd, Rs1, imm5u
```

Purpose: Do logical left shift operation with saturation on a 32-bit word. The shift amount is an immediate value.

Description: The first word data in Rs1 is left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the imm5u constant. Any shifted value greater than $2^{31}-1$ is saturated to $2^{31}-1$. Any shifted value smaller than -2^{31} is saturated to -2^{31} . And the saturated result is sign-extended and written to Rd. If any saturation is performed, set OV bit to 1.

Operations:

```
sa = imm5u;
res[(31+sa):0] = Rs1.W[0] << sa;
if (res > (2^31)-1) {
    res = 0x7fffffff; OV = 1;
} else if (res < -2^31) {
    res = 0x80000000; OV = 1;
}
Rd[31:0] = res[31:0]; // RV32
Rd[63:0] = SE(res[31:0]); // RV64
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long ndsksllw(long a, unsigned int b);
```

4.55. KSLL8 (SIMD 8-bit Saturating Shift Left Logical)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KSLL8 0110110	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KSLL8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit elements logical left shift operations with saturation simultaneously. The shift amount is a variable from a GPR.

Description: The 8-bit data elements in Rs1 are left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the low-order 3-bits of the value in the Rs2 register. Any shifted value greater than 2^7-1 is saturated to 2^7-1 . Any shifted value smaller than -2^7 is saturated to -2^7 . And the saturated results are written to Rd. If any saturation is performed, set OV bit to 1.

Operations:

```
sa = Rs2[2:0];
if (sa != 0) {
    res[(7+sa):0] = Rs1.B[x] << sa;
    if (res > (2^7)-1) {
        res = 0x7f; OV = 1;
    } else if (res < -2^7) {
        res = 0x80; OV = 1;
    }
    Rd.B[x] = res[7:0];
} else {
    Rd = Rs1;
}
for RV32: x=3...0,
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ksl18(unsigned long a, unsigned int b);
```

RV32:

```
int8x4_t __nds__v_ksl18(int8x4_t a, unsigned int b);
```

RV64:

```
int8x8_t __nds__v_ksl18(int8x8_t a, unsigned int b);
```

4.56. KSLLI8 (SIMD 8-bit Saturating Shift Left Logical Immediate)

Type: SIMD

Format:

31 25	24 23	22 20	19 15	14 12	11 7	6 0
KSLLI8 0111110	01	imm3u[2:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KSLLI8 Rd, Rs1, imm3u
```

Purpose: Do 8-bit elements logical left shift operations with saturation simultaneously. The shift amount is an immediate value.

Description: The 8-bit data elements in Rs1 are left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the imm3u constant. Any shifted value greater than 2^7-1 is saturated to 2^7-1 . Any shifted value smaller than -2^7 is saturated to -2^7 . And the saturated results are written to Rd. If any saturation is performed, set OV bit to 1.

Operations:

```
sa = imm3u[2:0];
if (sa != 0) {
    res[(7+sa):0] = Rs1.B[x] << sa;
    if (res > (2^7)-1) {
        res = 0x7f; OV = 1;
    } else if (res < -2^7) {
        res = 0x80; OV = 1;
    }
    Rd.B[x] = res[7:0];
} else {
    Rd = Rs1;
}
for RV32: x=3...0,
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ksl18(unsigned long a, unsigned int b);
```

RV32:

```
int8x4_t __nds__v_ksl18(int8x4_t a, unsigned int b);
```

RV64:

```
int8x8_t __nds__v_ksl18(int8x8_t a, unsigned int b);
```

4.57. KSLL16 (SIMD 16-bit Saturating Shift Left Logical)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KSLL16 0110010	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KSLL16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit elements logical left shift operations with saturation simultaneously. The shift amount is a variable from a GPR.

Description: The 16-bit data elements in Rs1 are left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the low-order 4-bits of the value in the Rs2 register. Any shifted value greater than $2^{15}-1$ is saturated to $2^{15}-1$. Any shifted value smaller than -2^{15} is saturated to -2^{15} . And the saturated results are written to Rd. If any saturation is performed, set OV bit to 1.

Operations:

```
sa = Rs2[3:0];
if (sa != 0) {
    res[(15+sa):0] = Rs1.H[x] << sa;
    if (res > (2^15)-1) {
        res = 0x7fff; OV = 1;
    } else if (res < -2^15) {
        res = 0x8000; OV = 1;
    }
    Rd.H[x] = res[15:0];
} else {
    Rd = Rs1;
}
for RV32: x=1...0,
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ksll16(unsigned long a, unsigned int b);
```

RV32:

```
int16x2_t __nds__v_ksll16(int16x2_t a, unsigned int b);
```

RV64:

```
int16x4_t __nds__v_ksll16(int16x4_t a, unsigned int b);
```

4.58. KSLLI16 (SIMD 16-bit Saturating Shift Left Logical Immediate)

Type: SIMD

Format:

31 25	24	23 20	19 15	14 12	11 7	6 0
KSLLI16 0111010	1	imm4u[3:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KSLLI16 Rd, Rs1, imm4u
```

Purpose: Do 16-bit elements logical left shift operations with saturation simultaneously. The shift amount is an immediate value.

Description: The 16-bit data elements in Rs1 are left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the imm4u constant. Any shifted value greater than $2^{15}-1$ is saturated to $2^{15}-1$. Any shifted value smaller than -2^{15} is saturated to -2^{15} . And the saturated results are written to Rd. If any saturation is performed, set OV bit to 1.

Operations:

```
sa = imm4u[3:0];
if (sa != 0) {
    res[(15+sa):0] = Rs1.H[x] << sa;
    if (res > (2^15)-1) {
        res = 0x7fff; OV = 1;
    } else if (res < -2^15) {
        res = 0x8000; OV = 1;
    }
    Rd.H[x] = res[15:0];
} else {
    Rd = Rs1;
}
for RV32: x=1...0,
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ksll16(unsigned long a, unsigned int b);
```

RV32:

```
int16x2_t __nds__v_ksll16(int16x2_t a, unsigned int b);
```

RV64:

```
int16x4_t __nds__v_ksll16(int16x4_t a, unsigned int b);
```

4.59. KSLRA8, KSLRA8.u

4.59.1. KSLRA8 (SIMD 8-bit Shift Left Logical with Saturation or Shift Right Arithmetic)

4.59.2. KSLRA8.u (SIMD 8-bit Shift Left Logical with Saturation or Rounding Shift Right Arithmetic)

Type: SIMD

Format:

KSLRA8

31 25	24 20	19 15	14 12	11 7	6 0
KSLRA8 0101111	Rs2	Rs1	000	Rd	GE80B 1111111

KSLRA8.u

31 25	24 20	19 15	14 12	11 7	6 0
KSLRA8.u 0110111	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KSLRA8 Rd, Rs1, Rs2  
KSLRA8.u Rd, Rs1, Rs2
```

Purpose: Do 8-bit elements logical left (positive) or arithmetic right (negative) shift operation with Q7 saturation for the left shift. The “.u” form performs additional rounding up operations for the right shift.

Description: The 8-bit data elements of Rs1 are left-shifted logically or right-shifted arithmetically based on the value of Rs2[3:0]. Rs2[3:0] is in the signed range of $[-2^3, 2^3-1]$. A positive Rs2[3:0] means logical left shift and a negative Rs2[3:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[3:0]. However, the behavior of “Rs2[3:0]==-2³ (0x8)” is defined to be equivalent to the behavior of “Rs2[3:0]==-(2³-1) (0x9)”.

The left-shifted results are saturated to the 8-bit signed integer range of $[-2^7, 2^7-1]$. For the “.u” form of the instruction, the right-shifted results are added a 1 to the most significant discarded bit position for rounding effect. After the shift, saturation, or rounding, the final results are written to Rd. If any saturation happens, this instruction sets the OV flag. The value of Rs2[31:4] will not affect this instruction.

Operations:

```

if (Rs2[3:0] < 0) {
    sa = -Rs2[3:0];
    sa = (sa == 8)? 7 : sa;
    if (!.u form) {
        res[7:-1] = SE9(Rs1.B[x][7:sa-1]) + 1;
        Rd.B[x] = res[7:0];
    } else {
        Rd.B[x] = SE8(Rs1.B[x][7:sa]);
    }
} else {
    sa = Rs2[2:0];
    res[(7+sa):0] = Rs1.B[x] <<(logic) sa;
    if (res > (2^7)-1) {
        res[7:0] = 0x7f; OV = 1;
    } else if (res < -2^7) {
        res[7:0] = 0x80; OV = 1;
    }
    Rd.B[x] = res[7:0];
}
for RV32: x=3...0,
for RV64: x=7...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KSLRA8

```
unsigned long __nds__kslra8(unsigned long a, int b);
```

RV32:

```
int8x4_t __nds__v_kslra8(int8x4_t a, int b);
```

RV64:

```
int8x8_t __nds__v_kslra8(int8x8_t a, int b);
```

- KSLRA8.u

```
unsigned long __nds__kslra8_u(unsigned long a, int b);
```

RV32:

```
int8x4_t __nds_v_kslra8_u(int8x4_t a, int b);
```

RV64:

```
int8x8_t __nds_v_kslra8_u(int8x8_t a, int b);
```

4.60. KSLRA16, KSLRA16.u

4.60.1. KSLRA16 (SIMD 16-bit Shift Left Logical with Saturation or Shift Right Arithmetic)

4.60.2. KSLRA16.u (SIMD 16-bit Shift Left Logical with Saturation or Rounding Shift Right Arithmetic)

Type: SIMD

Format:

KSLRA16

31 25	24 20	19 15	14 12	11 7	6 0
KSLRA16 0101011	Rs2	Rs1	000	Rd	GE80B 1111111

KSLRA16.u

31 25	24 20	19 15	14 12	11 7	6 0
KSLRA16.u 0110011	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KSLRA16 Rd, Rs1, Rs2
KSLRA16.u Rd, Rs1, Rs2
```

Purpose: Do 16-bit elements logical left (positive) or arithmetic right (negative) shift operation with Q15 saturation for the left shift. The “.u” form performs additional rounding up operations for the right shift.

Description: The 16-bit data elements of Rs1 are left-shifted logically or right-shifted arithmetically based on the value of Rs2[4:0]. Rs2[4:0] is in the signed range of $[-2^4, 2^4-1]$. A positive Rs2[4:0] means logical left shift and a negative Rs2[4:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[4:0]. However, the behavior of “Rs2[4:0]==-2⁴ (0x10)” is defined to be equivalent to the behavior of “Rs2[4:0]==-(2⁴-1) (0x11)”.

The left-shifted results are saturated to the 16-bit signed integer range of $[-2^{15}, 2^{15}-1]$. For the “.u” form of the instruction, the right-shifted results are added a 1 to the most significant discarded bit position for rounding effect. After the shift, saturation, or rounding, the final results are written to Rd. If any saturation happens, this instruction sets the OV flag. The value of Rs2[31:5] will not affect this instruction.

Operations:

```

if (Rs2[4:0] < 0) {
    sa = -Rs2[4:0];
    sa = (sa == 16)? 15 : sa;
    if (!.u form) {
        res[15:-1] = SE17(Rs1.H[x][15:sa-1]) + 1;
        Rd.H[x] = res[15:0];
    } else {
        Rd.H[x] = SE16(Rs1.H[x][15:sa]);
    }
} else {
    sa = Rs2[3:0];
    res[(15+sa):0] = Rs1.H[x] <<(logic) sa;
    if (res > (2^15)-1) {
        res[15:0] = 0x7fff; OV = 1;
    } else if (res < -2^15) {
        res[15:0] = 0x8000; OV = 1;
    }
    d.H[x] = res[15:0];
}
for RV32: x=1...0,
for RV64: x=3...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KSLRA16

```
unsigned long __nds__kslra16(unsigned long a, int b);
```

RV32:

```
int16x2_t __nds__v_kslra16(int16x2_t a, int b);
```

RV64:

```
int16x4_t __nds__v_kslra16(int16x4_t a, int b);
```

- KSLRA16.u

```
unsigned long __nds__kslra16_u(unsigned long a, int b);
```

RV32:

```
int16x2_t __nds__v_kslra16_u(int16x2_t a, int b);
```

RV64:

```
int16x4_t __nds__v_kslra16_u(int16x4_t a, int b);
```

4.61. KSLRAW (Shift Left Logical with Q31 Saturation or Shift Right Arithmetic)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KSLRAW 0110111	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KSLRAW Rd, Rs1, Rs2
```

Purpose: Perform a logical left (positive) or arithmetic right (negative) shift operation with Q31 saturation for the left shift on a 32-bit data.

Description: The lower 32-bit content of Rs1 is left-shifted logically or right-shifted arithmetically based on the value of Rs2[5:0]. Rs2[5:0] is in the signed range of $[-2^5, 2^5-1]$. A positive Rs2[5:0] means logical left shift and a negative Rs2[5:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[5:0] clamped to the actual shift range of [0, 31].

The left-shifted result is saturated to the 32-bit signed integer range of $[-2^{31}, 2^{31}-1]$. After the shift operation, the final result is bit-31 sign-extended and written to Rd. If any saturation happens, this instruction sets the OV flag. The value of Rs2[31:6] will not affect the operation of this instruction.

Operations:

```

if (Rs2[5:0] < 0) {
    sa = -Rs2[5:0];
    sa = (sa == 32)? 31 : sa;
    res[31:0] = Rs1.W[0] >>(arith) sa;
} else {
    sa = Rs2[5:0];
    tmp = Rs1.W[0] <<(logic) sa;
    if (tmp > (2^31)-1) {
        res[31:0] = (2^31)-1;
        OV = 1;
    } else if (tmp < -2^31) {
        res[31:0] = -2^31;
        OV = 1
    } else {
        res[31:0] = tmp[31:0];
    }
}
Rd = res[31:0]; // RV32
Rd = SE64(res[31:0]); // RV64

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long __nds__kslraw(int a, int b);
```

4.62. KSLRAW.u (Shift Left Logical with Q31 Saturation or Rounding Shift Right Arithmetic)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KSLRAW.u 0111111	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KSLRAW.u Rd, Rs1, Rs2
```

Purpose: Perform a logical left (positive) or arithmetic right (negative) shift operation with Q31 saturation for the left shift and a rounding up operation for the right shift on a 32-bit data.

Description: The lower 32-bit content of Rs1 is left-shifted logically or right-shifted arithmetically based on the value of Rs2[5:0]. Rs2[5:0] is in the signed range of $[-2^5, 2^5-1]$. A positive Rs2[5:0] means logical left shift and a negative Rs2[5:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[5:0] clamped to the actual shift range of [0, 31].

The left-shifted result is saturated to the 32-bit signed integer range of $[-2^{31}, 2^{31}-1]$. The right-shifted result is added a 1 to the most significant discarded bit position for rounding effect. After the shift, saturation, or rounding, the final result is bit-31 sign-extended and written to Rd. If any saturation happens, this instruction sets the OV flag. The value of Rs2[31:6] will not affect the operation of this instruction.

Operations:

```

if (Rs2[5:0] < 0) {
    sa = -Rs2[5:0];
    sa = (sa == 32)? 31 : sa;
    res[31:-1] = SE33(Rs1[31:(sa-1)]) + 1;
    rst[31:0] = res[31:0];
} else {
    sa = Rs2[5:0];
    tmp = Rs1.W[0] <<(logic) sa;
    if (tmp > (2^31)-1) {
        rst[31:0] = (2^31)-1;
        OV = 1;
    } else if (tmp < -2^31) {
        rst[31:0] = -2^31;
        OV = 1;
    } else {
        rst[31:0] = tmp[31:0];
    }
}
Rd = rst[31:0]; // RV32
Rd = SE64(rst[31:0]); // RV64

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long __nds_kslraw_u(int a, int b);
```

4.63. KSTAS16 (SIMD 16-bit Signed Saturating Straight Addition & Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KSTAS16 1100010	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
KSTAS16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element saturating addition and 16-bit signed integer element saturating subtraction in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks.

Description: This instruction adds the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2; at the same time, it subtracts the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs1. If any of the results are beyond the Q15 number range ($-2^{15} \leq Q15 \leq 2^{15}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for addition and [15:0] of 32-bit chunks in Rd for subtraction.

Operations:

```
res1 = Rs1.W[x][31:16] + Rs2.W[x][31:16];
res2 = Rs1.W[x][15:0] - Rs2.W[x][15:0];
for (res in [res1, res2]) {
    if (res > (2^15)-1) {
        res = (2^15)-1;
        OV = 1;
    } else if (res < -2^15) {
        res = -2^15;
        OV = 1;
    }
}
Rd.W[x][31:16] = res1;
Rd.W[x][15:0] = res2;
for RV32, x=0
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__kstas16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds__v_kstas16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __nds__v_kstas16(int16x4_t a, int16x4_t b);
```

4.64. KSTSA16 (SIMD 16-bit Signed Saturating Straight Subtraction & Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KSTSA16 1100011	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
KSTSA16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element saturating subtraction and 16-bit signed integer element saturating addition in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks.

Description: This instruction subtracts the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1; at the same time, it adds the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2 with the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs1. If any of the results are beyond the Q15 number range ($-2^{15} \leq Q15 \leq 2^{15}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for subtraction and [15:0] of 32-bit chunks in Rd for addition.

Operations:

```
res1 = Rs1.W[x][31:16] - Rs2.W[x][31:16];
res2 = Rs1.W[x][15:0] + Rs2.W[x][15:0];
for (res in [res1, res2]) {
    if (res > (2^15)-1) {
        res = (2^15)-1;
        OV = 1;
    } else if (res < -2^15) {
        res = -2^15;
        OV = 1;
    }
}
Rd.W[x][31:16] = res1;
Rd.W[x][15:0] = res2;
for RV32, x=0
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__kstsa16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds__v_kstsa16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __nds__v_kstsa16(int16x4_t a, int16x4_t b);
```

4.65. KSUB8 (SIMD 8-bit Signed Saturating Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KSUB8 0001101	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KSUB8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit signed elements saturating subtractions simultaneously.

Description: This instruction subtracts the 8-bit signed integer elements in Rs2 from the 8-bit signed integer elements in Rs1. If any of the results are beyond the Q7 number range ($-2^7 \leq Q7 \leq 2^7 - 1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.B[x] - Rs2.B[x];
if (res[x] > (2^7)-1) {
    res[x] = (2^7)-1;
    OV = 1;
} else if (res[x] < -2^7) {
    res[x] = -2^7;
    OV = 1;
}
Rd.B[x] = res[x];
for RV32: x=3...0,
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ksub8(unsigned long a, unsigned long b);
```

RV32:

```
int8x4_t __nds__v_ksub8(int8x4_t a, int8x4_t b);
```

RV64:

```
int8x8_t __nds__v_ksub8(int8x8_t a, int8x8_t b);
```

4.66. KSUB16 (SIMD 16-bit Signed Saturating Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KSUB16 0001001	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KSUB16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer elements saturating subtractions simultaneously.

Description: This instruction subtracts the 16-bit signed integer elements in Rs2 from the 16-bit signed integer elements in Rs1. If any of the results are beyond the Q15 number range ($-2^{15} \leq Q15 \leq 2^{15}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.H[x] - Rs2.H[x];
if (res[x] > (2^15)-1) {
    res[x] = (2^15)-1;
    OV = 1;
} else if (res[x] < -2^15) {
    res[x] = -2^15;
    OV = 1;
}
Rd.H[x] = res[x];
for RV32: x=1…0,
for RV64: x=3…0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ksub16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds__v_ksub16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __nds__v_ksub16(int16x4_t a, int16x4_t b);
```

4.67. KSUB64 (64-bit Signed Saturating Subtraction)

Type: DSP (64-bit Profile)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
KSUB64 1001001		Rs2		Rs1		001		Rd		GE80B 1111111	

Syntax:

```
KSUB64 Rd, Rs1, Rs2
```

Purpose: Perform a 64-bit signed integer subtraction. The result is saturated to the Q63 range.

RV32 Description: This instruction subtracts the 64-bit signed integer of an even/odd pair of registers specified by Rs2(4,1) from the 64-bit signed integer of an even/odd pair of registers specified by Rs1(4,1). If the 64-bit result is beyond the Q63 number range ($-2^{63} \leq Q63 \leq 2^{63}-1$), it is saturated to the range and the OV bit is set to 1. The saturated result is then written to an even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the operand and the even “ $2d$ ” register of the pair contains the low 32-bit of the operand.

RV64 Description: This instruction subtracts the 64-bit signed integer of Rs2 from the 64-bit signed integer of Rs1. If the 64-bit result is beyond the Q63 number range ($-2^{63} \leq Q63 \leq 2^{63}-1$), it is saturated to the range and the OV bit is set to 1. The saturated result is then written to Rd.

Operations:

RV32:

```
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
a_L = CONCAT(Rs1(4,1),1'b0); a_H = CONCAT(Rs1(4,1),1'b1);
b_L = CONCAT(Rs2(4,1),1'b0); b_H = CONCAT(Rs2(4,1),1'b1);
result = R[a_H].R[a_L] - R[b_H].R[b_L];
if (result > (2^63)-1) {
    result = (2^63)-1; OV = 1;
} else if (result < -2^63) {
    result = -2^63; OV = 1;
}
R[t_H].R[t_L] = result;
```

RV64:

```
result = Rs1 - Rs2;
if (result > (2^63)-1) {
    result = (2^63)-1; OV = 1;
} else if (result < -2^63) {
    result = -2^63; OV = 1;
}
Rd = result;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long long __nds__ksub64(long long a, long long b);
```

4.68. KSUBH (Signed Subtraction with Q15 Saturation)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KSUBH 0000011	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KSUBH Rd, Rs1, Rs2
```

Purpose: Subtract the signed lower 32-bit content of two registers with Q15 saturation.

Description: The signed lower 32-bit content of Rs2 is subtracted from the signed lower 32-bit content of Rs1. And the result is saturated to the 16-bit signed integer range of $[-2^{15}, 2^{15}-1]$ and then sign-extended and written to Rd. If saturation happens, this instruction sets the OV flag.

Operations:

```
tmp = Rs1.W[0] - Rs2.W[0];
if (tmp > (2^15)-1) {
    res = (2^15)-1;
    OV = 1;
} else if (tmp < -2^15) {
    res = -2^15;
    OV = 1
} else {
    res = tmp;
}
Rd = SE(res[15:0]);
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long __nds__ksubh(int a, int b);
```

4.69. KSUBW (Signed Subtraction with Q31 Saturation)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KSUBW 0000001	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KSUBW Rd, Rs1, Rs2
```

Purpose: Subtract the signed lower 32-bit content of two registers with Q31 saturation.

Description: The signed lower 32-bit content of Rs2 is subtracted from the signed lower 32-bit content of Rs1. And the result is saturated to the 32-bit signed integer range of $[-2^{31}, 2^{31}-1]$ and then sign-extended and written to Rd. If saturation happens, this instruction sets the OV flag.

Operations:

```
tmp = Rs1.W[0] - Rs2.W[0];
if (tmp > (2^31)-1) {
    res = (2^31)-1;
    OV = 1;
} else if (tmp < -2^31) {
    res = -2^31;
    OV = 1
} else {
    res = tmp;
}
Rd = res[31:0]; // RV32
Rd = SE(res[31:0]); // RV64
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long __nds__ksubw(int a, int b);
```

4.70. KWMMUL, KWMMUL.u

4.70.1. KWMMUL (SIMD Saturating MSW Signed Multiply Word & Double)

4.70.2. KWMMUL.u (SIMD Saturating MSW Signed Multiply Word & Double with Rounding)

Type: SIMD

Format:

KWMMUL

31 25	24 20	19 15	14 12	11 7	6 0
KWMMUL 0110001	Rs2	Rs1	001	Rd	GE80B 1111111

KWMMUL.u

31 25	24 20	19 15	14 12	11 7	6 0
KWMMUL.u 0111001	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KWMMUL Rd, Rs1, Rs2
KWMMUL.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of two registers, shift the results left 1-bit, saturate, and write the most significant 32-bit results to a register. The “.u” form additionally rounds up the multiplication results from the most significant discarded bit.

Description:

This instruction multiplies the 32-bit elements of Rs1 with the 32-bit elements of Rs2. It then shifts the multiplication results one bit to the left and takes the most significant 32-bit results. If the shifted result is greater than $2^{31}-1$, it is saturated to $2^{31}-1$ and the OV flag is set to 1. The final element result is written to Rd. The 32-bit elements of Rs1 and Rs2 are treated as signed integers. The “.u” form of the instruction additionally rounds up the 64-bit multiplication results by adding a 1 to bit 30 before the shift and saturation operations.

Operations:

```

if ((0x80000000 != Rs1.W[x]) | (0x80000000 != Rs2.W[x])) {
    Mres[x][63:0] = Rs1.W[x] * Rs2.W[x];
    if (!.u) form) {
        Round[x][33:0] = Mres[x][63:30] + 1;
        Rd.W[x] = Round[x][32:1];
    } else {
        Rd.W[x] = Mres[x][62:31];
    }
} else {
    Rd.W[x] = 0x7fffffff;
    OV = 1;
}
for RV32: x=0
for RV64: x=1...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KWMMUL

```
long __nds_kwmmul(long a, long b);
```

RV64:

```
int32x2_t __nds_v_kwmmul(int32x2_t a, int32x2_t b);
```

- KWMMUL.u

```
long __nds_kwmmul_u(long a, long b);
```

RV64:

```
int32x2_t __nds_v_kwmmul_u(int32x2_t a, int32x2_t b);
```

4.71. MADDR32 (Multiply and Add to 32-Bit Word)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
MADDR32 1100010	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
MADDR32 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit contents of two registers and add the lower 32-bit multiplication result to the 32-bit content of a destination register. Write the final result back to the destination register.

Description: This instruction multiplies the lower 32-bit content of Rs1 with that of Rs2. It adds the lower 32-bit multiplication result to the lower 32-bit content of Rd and writes the final result (RV32) or sign-extended result (RV64) back to Rd. The contents of Rs1 and Rs2 can be either signed or unsigned integers.

Operations:

RV32:

```
Mresult = Rs1 * Rs2;  
Rd = Rd + Mresult.W[0];
```

RV64:

```
Mresult = Rs1.W[0] * Rs2.W[0];  
tres[31:0] = Rd.W[0] + Mresult.W[0];  
Rd = SE64(tres[31:0]);
```

Exceptions: None

Privilege level: All

Note:

4.72. MAXW (32-bit Signed Word Maximum)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
MAXW 1111001	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
MAXW Rd, Rs1, Rs2
```

Purpose: Get the larger value from the 32-bit contents of two general registers.

Description: This instruction compares two signed 32-bit integers stored in Rs1 and Rs2, picks the larger value as the result, and writes the result to Rd.

Operations:

```
if (Rs1.W[0] >= Rs2.W[0]) {  
    Rd = SE(Rs1.W[0]);  
} else {  
    Rd = SE(Rs2.W[0]);  
}
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long __nds_maxw(int a, int b);
```

4.73. MINW (32-bit Signed Word Minimum)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
MINW 1111000	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
MINW Rd, Rs1, Rs2
```

Purpose: Get the smaller value from the 32-bit contents of two general registers.

Description: This instruction compares two signed 32-bit integers stored in Rs1 and Rs2, picks the smaller value as the result, and writes the result to Rd.

Operations:

```
if (Rs1.W[0] >= Rs2.W[0]) { Rd = SE(Rs2.W[0]); } else { Rd = SE(Rs1.W[0]); }
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long __nds_minw(int a, int b);
```

4.74. MSUBR32 (Multiply and Subtract from 32-Bit Word)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
MSUBR32 1100011	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
MSUBR32 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit contents of two registers and subtract the lower 32-bit multiplication result from the 32-bit content of a destination register. Write the final result back to the destination register.

Description: This instruction multiplies the lower 32-bit content of Rs1 with that of Rs2, subtracts the lower 32-bit multiplication result from the lower 32-bit content of Rd, then writes the final result (RV32) or sign-extended result (RV64) back to Rd. The contents of Rs1 and Rs2 can be either signed or unsigned integers.

Operations:

RV32:

```
Mresult = Rs1 * Rs2;  
Rd = Rd - Mresult.W[0];
```

RV64:

```
Mresult = Rs1.W[0] * Rs2.W[0];  
tres[31:0] = Rd.W[0] - Mresult.W[0];  
Rd = SE64(tres[31:0]);
```

Exceptions: None

Privilege level: All

Note:

4.75. MULR64 (Multiply Word Unsigned to 64-bit Data)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
MULR64 1111000	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
MULR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit unsigned integer contents of two registers and write the 64-bit result.

RV32 Description:

This instruction multiplies the 32-bit content of Rs1 with that of Rs2 and writes the 64-bit multiplication result to an even/odd pair of registers containing Rd. Rd(4,1) index d determines the even/odd pair group of the two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

The lower 32-bit contents of Rs1 and Rs2 are treated as unsigned integers.

RV64 Description:

This instruction multiplies the lower 32-bit content of Rs1 with that of Rs2 and writes the 64-bit multiplication result to Rd.

The lower 32-bit contents of Rs1 and Rs2 are treated as unsigned integers.

Operations:

RV32:

```
Mresult = CONCAT(1'b0,Rs1) u* CONCAT(1'b0,Rs2);  
R[Rd(4,1).1(0)][31:0] = Mresult[63:32];  
R[Rd(4,1).0(0)][31:0] = Mresult[31:0];
```

RV64:

```
Rd = Mresult[63:0];  
Mresult = CONCAT(1'b0,Rs1.W[0]) u* CONCAT(1'b0,Rs2.W[0]);
```

Exceptions: None

Privilege level: All

Note:

4.76. MULSR64 (Multiply Word Signed to 64-bit Data)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
MULSR64 1110000	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
MULSR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit signed integer contents of two registers and write the 64-bit result.

RV32 Description:

This instruction multiplies the lower 32-bit content of Rs1 with the lower 32-bit content of Rs2 and writes the 64-bit multiplication result to an even/odd pair of registers containing Rd. Rd(4,1) index d determines the even/odd pair group of the two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

The lower 32-bit contents of Rs1 and Rs2 are treated as signed integers.

RV64 Description:

This instruction multiplies the lower 32-bit content of Rs1 with the lower 32-bit content of Rs2 and writes the 64-bit multiplication result to Rd.

The lower 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

RV32:

```
Mresult = Ra s* Rb;  
R[Rd(4,1).1(0)][31:0] = Mresult[63:32];  
R[Rd(4,1).0(0)][31:0] = Mresult[31:0];
```

RV64:

```
Mresult = Ra.W[0] s* Rb.W[0];  
Rd = Mresult[63:0];
```

Exceptions: None

Privilege level: All

Note:

4.77. PBSAD (Parallel Byte Sum of Absolute Difference)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
PBSAD 1111110	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
PBSAD Rd, Rs1, Rs2
```

Purpose: Calculate the sum of absolute difference of unsigned 8-bit data elements.

Description: This instruction subtracts the un-signed 8-bit elements of Rs2 from those of Rs1. Then it adds the absolute value of each difference together and writes the result to Rd.

Operations:

```
absdiff[x] = ABS(Rs1.B[x] - Rs2.B[x]);  
Rd = SUM(absdiff[x]);  
for RV32: x=3...0,  
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__pbsad(unsigned long a, unsigned long b);
```

RV32:

```
unsigned int __nds__v_pbsad(uint8x4_t a, uint8x4_t b);
```

RV64:

```
unsigned long __nds__v_pbsad(uint8x8_t a, uint8x8_t b);
```

4.78. PBSADA (Parallel Byte Sum of Absolute Difference Accum)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
PBSADA 1111111	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
PBSADA Rd, Rs1, Rs2
```

Purpose: Calculate the sum of absolute difference of four unsigned 8-bit data elements and accumulate it into a register.

Description: This instruction subtracts the un-signed 8-bit elements of Rs2 from those of Rs1. It then adds the absolute value of each difference together along with the content of Rd and writes the accumulated result back to Rd.

Operations:

```
absdiff[x] = ABS(Rs1.B[x] - Rs2.B[x]);  
Rd = Rd + SUM(absdiff[x]);  
for RV32: x=3...0,  
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__pbsada(unsigned long t, unsigned long a, unsigned long b);
```

RV32:

```
unsigned int __nds__v_pbsada(unsigned long t, uint8x4_t a, uint8x4_t b);
```

RV64:

```
unsigned long __nds__v_pbsada(unsigned long t, uint8x8_t a, uint8x8_t b);
```

4.79. PKBB16, PKBT16, PKTT16, PKTB16

4.79.1. PKBB16 (Pack Two 16-bit Data from Both Bottom Half)

4.79.2. PKBT16 (Pack Two 16-bit Data from Bottom and Top Half)

4.79.3. PKTT16 (Pack Two 16-bit Data from Both Top Half)

4.79.4. PKTB16 (Pack Two 16-bit Data from Top and Bottom Half)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
PK <u>xy</u> 16 00 <u>zz</u> 111	Rs2	Rs1	001	Rd	GE80B 1111111

<u>xy</u>	<u>zz</u>
BB	00
BT	01
TT	10
TB	11

Syntax:

```
PKBB16 Rd, Rs1, Rs2  
PKBT16 Rd, Rs1, Rs2  
PKTT16 Rd, Rs1, Rs2  
PKTB16 Rd, Rs1, Rs2
```

Purpose: Pack 16-bit data from 32-bit chunks in two registers.

- PKBB16: bottom.bottom
- PKBT16 bottom.top
- PKTT16 top.top
- PKTB16 top.bottom

Description:

(PKBB16) moves Rs1.W[x][15:0] to Rd.W[x][31:16] and moves Rs2.W[x] [15:0] to Rd.W[x] [15:0].

(PKBT16) moves Rs1.W[x] [15:0] to Rd.W[x] [31:16] and moves Rs2.W[x] [31:16] to Rd.W[x] [15:0].

(PKTT16) moves Rs1.W[x] [31:16] to Rd.W[x] [31:16] and moves Rs2.W[x] [31:16] to Rd.W[x] [15:0].

(PKTB16) moves Rs1.W[x] [31:16] to Rd.W[x] [31:16] and moves Rs2.W[x] [15:0] to Rd.W[x] [15:0].

Operations:

```
Rd.W[x][31:0] = CONCAT(Rs1.W[x][15:0], Rs2.W[x][15:0]); // PKBB16  
Rd.W[x][31:0] = CONCAT(Rs1.W[x][15:0], Rs2.W[x][31:16]); // PKBT16  
Rd.W[x][31:0] = CONCAT(Rs1.W[x][31:16], Rs2.W[x][15:0]); // PKTB16  
Rd.W[x][31:0] = CONCAT(Rs1.W[x][31:16], Rs2.W[x][31:16]); // PKTT16  
for RV32: x=0,  
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- PKBB16

```
unsigned long __nds_pkbb16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds_v_pkbb16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds_v_pkbb16(uint16x4_t a, uint16x4_t b);
```

- PKBT16

```
unsigned long __nds_pkbt16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds_v_pkbt16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds_v_pkbt16(uint16x4_t a, uint16x4_t b);
```

- PKTB16

```
unsigned long __nds_pktb16(unsigned long a, unsigned long b);
```

```
RV32:  
    uint16x2_t __nds__v_pkbt16(uint16x2_t a, uint16x2_t b);  
RV64:  
    uint16x4_t __nds__v_pkbt16(uint16x4_t a, uint16x4_t b);
```

- PKTT16

```
unsigned long __nds__pktt16(unsigned long a, unsigned long b);
```

```
RV32:  
    uint16x2_t __nds__v_pktt16(uint16x2_t a, uint16x2_t b);  
RV64:  
    uint16x4_t __nds__v_pktt16(uint16x4_t a, uint16x4_t b);
```

4.80. RADD8 (SIMD 8-bit Signed Halving Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RADD8 0000100	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
RADD8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit signed integer element additions simultaneously. The element results are halved to avoid overflow or saturation.

Description: This instruction adds the 8-bit signed integer elements in Rs1 with the 8-bit signed integer elements in Rs2. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Operations:

$Rd.B[x] = (Rs1.B[x] + Rs2.B[x]) \text{ s} \gg 1;$ for RV32: $x=3\ldots 0$, for RV64: $x=7\ldots 0$

Exceptions: None

Privilege level: All

Examples:

- $Rs1 = 0x7F, Rs2 = 0x7F, Rd = 0x7F$
- $Rs1 = 0x80, Rs2 = 0x80, Rd = 0x80$
- $Rs1 = 0x40, Rs2 = 0x80, Rd = 0xE0$

Intrinsic functions:

```
unsigned long __nds__radd8(unsigned long a, unsigned long b);
```

RV32:

```
int8x4_t __nds__v_radd8(int8x4_t a, int8x4_t b);
```

RV64:

```
int8x8_t __nds__v_radd8(int8x8_t a, int8x8_t b);
```

4.81. RADD16 (SIMD 16-bit Signed Halving Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RADD16 0000000	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
RADD16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element additions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 16-bit signed integer elements in Rs1 with the 16-bit signed integer elements in Rs2. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Operations:

$Rd.H[x] = (Rs1.H[x] + Rs2.H[x]) \text{ s} \gg 1$; for RV32: $x=1\dots0$, for RV64: $x=3\dots0$

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0x7FFF, Rs2 = 0x7FFF, Rd = 0x7FFF
- Rs1 = 0x8000, Rs2 = 0x8000, Rd = 0x8000
- Rs1 = 0x4000, Rs2 = 0x8000, Rd = 0xE000

Intrinsic functions:

```
unsigned long __nds__radd16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds__v_radd16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __nds__v_radd16(int16x4_t a, int16x4_t b);
```

4.82. RADD64 (64-bit Signed Halving Addition)

Type: DSP (64-bit Profile)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
RADD64 1000000		Rs2		Rs1		001		Rd		GE80B 1111111	

Syntax:

```
RADD64 Rd, Rs1, Rs2
```

Purpose: Add two 64-bit signed integers. The result is halved to avoid overflow or saturation.

RV32 Description: This instruction adds the 64-bit signed integer of an even/odd pair of registers specified by Rs1(4,1) with the 64-bit signed integer of an even/odd pair of registers specified by Rs2(4,1). The 64-bit addition result is first arithmetically right-shifted by 1 bit and then written to an even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., value d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction adds the 64-bit signed integer in Rs1 with the 64-bit signed integer in Rs2. The 64-bit addition result is first arithmetically right-shifted by 1 bit and then written to Rd.

Operations:

RV32:

```
t_L = CONCAT(Rd(4,1),10b0); t_H = CONCAT(Rd(4,1),10b1);
a_L = CONCAT(Rs1(4,1),10b0); a_H = CONCAT(Rs1(4,1),10b1);
b_L = CONCAT(Rs2(4,1),10b0); b_H = CONCAT(Rs2(4,1),10b1);
R[t_H].R[t_L] = (R[a_H].R[a_L] + R[b_H].R[b_L]) s>> 1;
```

RV64:

```
Rd = (Rs1 + Rs2) s>> 1;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long long __nds__radd64(long long a, long long b);
```

4.83. RADDW (32-bit Signed Halving Addition)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RADDW 0010000	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
RADDW Rd, Rs1, Rs2
```

Purpose: Add 32-bit signed integers and the results are halved to avoid overflow or saturation.

Description: This instruction adds the first 32-bit signed integer in Rs1 with the first 32-bit signed integer in Rs2. The result is first arithmetically right-shifted by 1 bit and then sign-extended and written to Rd.

Operations:

RV32:

```
Rd[31:0] = (Rs1[31:0] + Rs2[31:0]) s>> 1;
```

RV64:

```
resw[31:0] = (Rs1[31:0] + Rs2[31:0]) s>> 1;  
Rd[63:0] = SE(resw[31:0]);
```

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0x7FFFFFFF, Rs2 = 0x7FFFFFFF, Rd = 0x7FFFFFFF
- Rs1 = 0x80000000, Rs2 = 0x80000000, Rd = 0x80000000
- Rs1 = 0x40000000, Rs2 = 0x80000000, Rd = 0xE0000000

Intrinsic functions:

```
long __nds__raddw(int a, int b);
```

4.84. RCRAS16 (SIMD 16-bit Signed Halving Cross Addition & Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RCRAS16 0000010	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
RCRAS16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element addition and 16-bit signed integer element subtraction in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2, and subtracts the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs1. The element results are first arithmetically right-shifted by 1 bit and then written to [31:16] of 32-bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] + Rs2.W[x][15:0]) s>> 1;  
Rd.W[x][15:0] = (Rs1.W[x][15:0] - Rs2.W[x][31:16]) s>> 1;  
for RV32, x=0  
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Examples: Please see “RADD16” and “RSUB16” instructions.

Intrinsic functions:

```
unsigned long __nds_rcras16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds__v_rcras16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __nds__v_rcras16(int16x4_t a, int16x4_t b);
```

4.85. RCRSA16 (SIMD 16-bit Signed Halving Cross Subtraction & Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RCRSA16 0000011	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
RCRSA16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element subtraction and 16-bit signed integer element addition in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1, and adds the 16-bit signed element integer in [15:0] of 32-bit chunks in Rs1 with the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2. The two results are first arithmetically right-shifted by 1 bit and then written to [31:16] of 32-bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] - Rs2.W[x][15:0]) s>> 1;  
Rd.W[x][15:0] = (Rs1.W[x][15:0] + Rs2.W[x][31:16]) s>> 1;  
for RV32, x=0  
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Examples: Please see “RADD16” and “RSUB16” instructions.

Intrinsic functions:

```
unsigned long __nds__rcrsa16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds__v_rcrsa16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __nds__v_rcrsa16(int16x4_t a, int16x4_t b);
```

4.86. RDOV (Read OV flag)

Type: DSP

Format:

31 20	19 15	14 12	11 7	6 0
ucode (0x801) 100000000001	00000	010	Rd	SYSTEM 1110011

Syntax:

```
RDOV Rd # pseudo mnemonic
```

Purpose: This pseudo instruction is an alias to “CSRR Rd, ucode” instruction which maps to the real instruction of “CSRRS Rd, ucode, x0”.

Intrinsic functions:

```
unsigned long __nds__rdov(void);
```

4.87. RSTAS16 (SIMD 16-bit Signed Halving Straight Addition & Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RSTAS16 1011010	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
RSTAS16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element addition and 16-bit signed integer element subtraction in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2, and subtracts the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs1. The element results are first arithmetically right-shifted by 1 bit and then written to [31:16] of 32-bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] + Rs2.W[x][31:16]) s>> 1;  
Rd.W[x][15:0] = (Rs1.W[x][15:0] - Rs2.W[x][15:0]) s>> 1;  
for RV32, x=0  
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Examples: Please see “RADD16” and “RSUB16” instructions.

Intrinsic functions:

```
unsigned long __nds__rstas16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds__v_rstas16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __nds__v_rstas16(int16x4_t a, int16x4_t b);
```

4.88. RSTSA16 (SIMD 16-bit Signed Halving Straight Subtraction & Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RSTSA16 1011011	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
RSTSA16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element subtraction and 16-bit signed integer element addition in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1, and adds the 16-bit signed element integer in [15:0] of 32-bit chunks in Rs1 with the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2. The two results are first arithmetically right-shifted by 1 bit and then written to [31:16] of 32-bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] - Rs2.W[x][31:16]) s>> 1;  
Rd.W[x][15:0] = (Rs1.W[x][15:0] + Rs2.W[x][15:0]) s>> 1;  
for RV32, x=0  
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Examples: Please see “RADD16” and “RSUB16” instructions.

Intrinsic functions:

```
unsigned long __nds__rstsa16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds__v_rstsa16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __nds__v_rstsa16(int16x4_t a, int16x4_t b);
```

4.89. RSUB8 (SIMD 8-bit Signed Halving Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RSUB8 0000101	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
RSUB8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit signed integer element subtractions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 8-bit signed integer elements in Rs2 from the 8-bit signed integer elements in Rs1. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] - Rs2.B[x]) s>> 1;  
for RV32: x=3…0,  
for RV64: x=7…0
```

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0x7F, Rs2 = 0x80, Rd = 0x7F
- Rs1 = 0x80, Rs2 = 0x7F, Rd = 0x80
- Rs1 = 0x80, Rs2 = 0x40, Rd = 0xA0

Intrinsic functions:

```
unsigned long __nds__rsub8(unsigned long a, unsigned long b);
```

RV32:

```
int8x4_t __ndS__v_rsub8(int8x4_t a, int8x4_t b);
```

RV64:

```
int8x8_t __ndS__v_rsub8(int8x8_t a, int8x8_t b);
```

4.90. RSUB16 (SIMD 16-bit Signed Halving Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RSUB16 0000001	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
RSUB16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element subtractions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 16-bit signed integer elements in Rs2 from the 16-bit signed integer elements in Rs1. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] - Rs2.H[x]) s>> 1;  
for RV32: x=1…0,  
for RV64: x=3…0
```

Exceptions: None

Privilege level: All

Examples:

- Ra = 0x7FFF, Rb = 0x8000, Rt = 0x7FFF
- Ra = 0x8000, Rb = 0x7FFF, Rt = 0x8000
- Ra = 0x8000, Rb = 0x4000, Rt = 0xA000

Intrinsic functions:

```
unsigned long __nds__rsub16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds__v_rsub16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __nds__v_rsub16(int16x4_t a, int16x4_t b);
```

4.91. RSUB64 (64-bit Signed Halving Subtraction)

Type: DSP (64-bit Profile)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RSUB64 1000001	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
RSUB64 Rd, Rs1, Rs2
```

Purpose: Perform a 64-bit signed integer subtraction. The result is halved to avoid overflow or saturation.

RV32 Description: This instruction subtracts the 64-bit signed integer of an even/odd pair of registers specified by Rb(4,1) from the 64-bit signed integer of an even/odd pair of registers specified by Ra(4,1). The subtraction result is first arithmetically right-shifted by 1 bit and then written to an even/odd pair of registers specified by Rt(4,1).

Rx(4,1), i.e., value d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction subtracts the 64-bit signed integer in Rs2 from the 64-bit signed integer in Rs1. The 64-bit subtraction result is first arithmetically right-shifted by 1 bit and then written to Rd.

Operations:

RV32:

```
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
a_L = CONCAT(Rs1(4,1),1'b0); a_H = CONCAT(Rs1(4,1),1'b1);
b_L = CONCAT(Rs2(4,1),1'b0); b_H = CONCAT(Rs2(4,1),1'b1);
R[t_H].R[t_L] = (R[a_H].R[a_L] - R[b_H].R[b_L]) s>> 1;
```

RV64:

```
Rd = (Rs1 - Rs2) s>> 1;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long long __nds__rsub64(long long a, long long b);
```

4.92. RSUBW (32-bit Signed Halving Subtraction)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RSUBW 0010001	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
RSUBW Rd, Rs1, Rs2
```

Purpose: Subtract 32-bit signed integers and the result is halved to avoid overflow or saturation.

Description: This instruction subtracts the first 32-bit signed integer in Rs2 from the first 32-bit signed integer in Rs1. The result is first arithmetically right-shifted by 1 bit and then sign-extended and written to Rd.

Operations:

RV32:

```
Rd[31:0] = (Rs1[31:0] - Rs2[31:0]) s>> 1;
```

RV64:

```
resw[31:0] = (Rs1[31:0] - Rs2[31:0]) s>> 1;  
Rd[63:0] = SE(resw[31:0]);
```

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0x7FFFFFFF, Rs2 = 0x80000000, Rd = 0x7FFFFFFF
- Rs1 = 0x80000000, Rs2 = 0x7FFFFFFF, Rd = 0x80000000
- Rs1 = 0x80000000, Rs2 = 0x40000000, Rd = 0xA0000000

Intrinsic functions:

```
long __nds__rsubw(int a, int b);
```

4.93. SCLIP8 (SIMD 8-bit Signed Clip Value)

Type: SIMD

Format:

31 25	24 23	22 20	19 15	14 12	11 7	6 0
SCLIP8 1000110	00	imm3u[2:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SCLIP8 Rd, Rs1, imm3u[2:0]
```

Purpose: Limit the 8-bit signed integer elements of a register into a signed range simultaneously.

Description: This instruction limits the 8-bit signed integer elements stored in Rs1 into a signed integer range between $2^{\text{imm3u}}-1$ and -2^{imm3u} , and writes the limited results to Rd. For example, if imm3u is 3, the 8-bit input values should be saturated between 7 and -8. If saturation is performed, set OV bit to 1.

Operations:

```
src = Rs1.B[x];
if (src > (2^imm3u)-1) {
    src = (2^imm3u)-1;
    OV = 1;
} else if (src < -2^imm3u) {
    src = -2^imm3u;
    OV = 1;
}
Rd.B[x] = src
for RV32: x=3...0,
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_sclip8(unsigned long a, unsigned int b);
```

RV32:

```
int8x4_t __nds__v_sclip8(int8x4_t a, unsigned int b);
```

RV64:

```
int8x8_t __nds__v_sclip8(int8x8_t a, unsigned int b);
```

4.94. SCLIP16 (SIMD 16-bit Signed Clip Value)

Type: SIMD

Format:

31 25	24	23 20	19 15	14 12	11 7	6 0
SCLIP16 1000010	0	imm4u[3:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SCLIP16 Rd, Rs1, imm4u[3:0]
```

Purpose: Limit the 16-bit signed integer elements of a register into a signed range simultaneously.

Description: This instruction limits the 16-bit signed integer elements stored in Rs1 into a signed integer range between $2^{\text{imm4u}}-1$ and -2^{imm4u} , and writes the limited results to Rd. For example, if imm4u is 3, the 16-bit input values should be saturated between 7 and -8. If saturation is performed, set OV bit to 1.

Operations:

```
src = Rs1.H[x];
if (src > (2^imm4u)-1) {
    src = (2^imm4u)-1;
    OV = 1;
} else if (src < -2^imm4u) {
    src = -2^imm4u;
    OV = 1;
}
Rd.H[x] = src
for RV32: x=1…0,
for RV64: x=3…0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__sclip16(unsigned long a, unsigned int b);
```

RV32:

```
int16x2_t __nds__v_sclip16(int16x2_t a, unsigned int b);
```

RV64:

```
int16x4_t __nds__v_sclip16(int16x4_t a, unsigned int b);
```

4.95. SCLIP32 (SIMD 32-bit Signed Clip Value)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SCLIP32 1110010	imm5u[4:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SCLIP32 Rd, Rs1, imm5u[4:0]
```

Purpose: Limit the 32-bit signed integer elements of a register into a signed range simultaneously.

Description: This instruction limits the 32-bit signed integer elements stored in Rs1 into a signed integer range between $2^{\text{imm5u}}-1$ and -2^{imm5u} , and writes the limited results to Rd. For example, if imm5u is 3, the 32-bit input values should be saturated between 7 and -8. If saturation is performed, set OV bit to 1.

Operations:

```
src = Rs1.W[x];
if (src > (2^imm5u)-1) {
    src = (2^imm5u)-1;
    OV = 1;
} else if (src < -2^imm5u) {
    src = -2^imm5u;
    OV = 1;
}
Rd.W[x] = src
for RV32: x=0,
for RV64: x=1…0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long __nds_sclip32(long a, unsigned int b);
```

RV64:

```
int32x2_t __nds__v_sclip32(int32x2_t a, unsigned int b);
```

4.96. SCMPLE8 (SIMD 8-bit Signed Compare Less Than & Equal)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SCMPLE8 0001111	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SCMPLE8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit signed integer elements less than & equal comparisons simultaneously.

Description: This instruction compares the 8-bit signed integer elements in Rs1 with the 8-bit signed integer elements in Rs2 to see if the one in Rs1 is less than or equal to the one in Rs2. If it is true, the result is 0xFF; otherwise, the result is 0x0. The element comparison results are written to Rd

Operations:

```
Rd.B[x] = (Rs1.B[x] {le} Rs2.B[x])? 0xff : 0x0;  
for RV32: x=3...0,  
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__scmple8(unsigned long a, unsigned long b);
```

```
RV32:  
    uint8x4_t __nds__v_scmple8(int8x4_t a, int8x4_t b);  
RV64:  
    uint8x8_t __nds__v_scmple8(int8x8_t a, int8x8_t b);
```

4.97. SCMPLE16 (SIMD 16-bit Signed Compare Less Than & Equal)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SCMPLE16 0001110	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SCMPLE16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer elements less than & equal comparisons simultaneously.

Description: This instruction compares the 16-bit signed integer elements in Rs1 with the 16-bit signed integer elements in Rs2 to see if the one in Rs1 is less than or equal to the one in Rs2. If it is true, the result is 0xFFFF; otherwise, the result is 0x0. The element comparison results are written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] {le} Rs2.H[x])? 0xffff : 0x0;  
for RV32: x=1...0,  
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__scmple16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_scmple16(int16x2_t a, int16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_scmple16(int16x4_t a, int16x4_t b);
```

4.98. SCMPLT8 (SIMD 8-bit Signed Compare Less Than)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SCMPLT8 0000111	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SCMPLT8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit signed integer elements less than comparisons simultaneously.

Description: This instruction compares the 8-bit signed integer elements in Rs1 with the 8-bit signed integer elements in Rs2 to see if the one in Rs1 is less than the one in Rs2. If it is true, the result is 0xFF; otherwise, the result is 0x0. The element comparison results are written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] < Rs2.B[x])? 0xff : 0x0;  
for RV32: x=3...0,  
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__scmplt8(unsigned long a, unsigned long b);
```

```
RV32:  
    uint8x4_t __nds__v_scmplt8(int8x4_t a, int8x4_t b);  
RV64:  
    uint8x8_t __nds__v_scmplt8(int8x8_t a, int8x8_t b);
```

4.99. SCMPLT16 (SIMD 16-bit Signed Compare Less Than)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SCMPLT16 0000110	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SCMPLT16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer elements less than comparisons simultaneously.

Description: This instruction compares the 16-bit signed integer elements in Rs1 with the two 16-bit signed integer elements in Rs2 to see if the one in Rs1 is less than the one in Rs2. If it is true, the result is 0xFFFF; otherwise, the result is 0x0. The element comparison results are written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] < Rs2.H[x])? 0xffff : 0x0;  
for RV32: x=1…0,  
for RV64: x=3…0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__scmplt16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_scmplt16(int16x2_t a, int16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_scmplt16(int16x4_t a, int16x4_t b);
```

4.100. SLL8 (SIMD 8-bit Shift Left Logical)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SLL8 0101110	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SLL8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit elements logical left shift operations simultaneously. The shift amount is a variable from a GPR.

Description: The 8-bit elements in Rs1 are left-shifted logically. And the results are written to Rd. The shifted out bits are filled with zero and the shift amount is specified by the low-order 3-bits of the value in the Rs2 register.

Operations:

```
sa = Rs2[2:0];
Rd.B[x] = Rs1.B[x] << sa;
for RV32: x=3...0,
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__sll8(unsigned long a, unsigned int b);
```

```
RV32:
uint8x4_t __nds__v_sll8(uint8x4_t a, unsigned int b);
RV64:
uint8x8_t __nds__v_sll8(uint8x8_t a, unsigned int b);
```

4.101. SLLI8 (SIMD 8-bit Shift Left Logical Immediate)

Type: SIMD

Format:

31 25	24 23	22 20	19 15	14 12	11 7	6 0
SLLI8 0111110	00	imm3u[2:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SLLI8 Rd, Rs1, imm3u
```

Purpose: Do 8-bit elements logical left shift operations simultaneously. The shift amount is an immediate value.

Description: The 8-bit elements in Rs1 are left-shifted logically. And the results are written to Rd. The shifted out bits are filled with zero and the shift amount is specified by the imm3u constant.

Operations:

```
sa = imm3u[2:0];
Rd.B[x] = Rs1.B[x] << sa;
for RV32: x=3...0,
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__sll8(unsigned long a, unsigned int b);
```

RV32:

```
uint8x4_t __nds__v_sll8(uint8x4_t a, unsigned int b);
```

RV64:

```
uint8x8_t __nds__v_sll8(uint8x8_t a, unsigned int b);
```

4.102. SLL16 (SIMD 16-bit Shift Left Logical)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SLL16 0101010	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SLL16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit elements logical left shift operations simultaneously. The shift amount is a variable from a GPR.

Description: The 16-bit elements in Rs1 are left-shifted logically. And the results are written to Rd. The shifted out bits are filled with zero and the shift amount is specified by the low-order 4-bits of the value in the Rs2 register.

Operations:

```
sa = Rs2[3:0];
Rd.H[x] = Rs1.H[x] << sa;
for RV32: x=1...0,
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__sll16(unsigned long a, unsigned int b);
```

RV32:

```
uint16x2_t __nds__v_sll16(uint16x2_t a, unsigned int b);
```

RV64:

```
uint16x4_t __nds__v_sll16(uint16x4_t a, unsigned int b);
```

4.103. SLLI16 (SIMD 16-bit Shift Left Logical Immediate)

Type: SIMD

Format:

31 25	24	23 20	19 15	14 12	11 7	6 0
SLLI16 0111010	0	imm4u[3:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SLLI16 Rd, Rs1, imm4[3:0]
```

Purpose: Do 16-bit element logical left shift operations simultaneously. The shift amount is an immediate value.

Description: The 16-bit elements in Rs1 are left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the imm4[3:0] constant. And the results are written to Rd.

Operations:

```
sa = imm4[3:0];
Rd.H[x] = Rs1.H[x] << sa;
for RV32: x=1...0,
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__sll16(unsigned long a, unsigned int b);
```

RV32:

```
uint16x2_t __nds__v_sll16(uint16x2_t a, unsigned int b);
```

RV64:

```
uint16x4_t __nds__v_sll16(uint16x4_t a, unsigned int b);
```

4.104. SMAL (Signed Multiply Halves & Add 64-bit)

Type: Partial-SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SMAL 0101111	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
SMAL Rd, Rs1, Rs2
```

Purpose: Multiply the signed bottom 16-bit content of the 32-bit elements of a register with the top 16-bit content of the same 32-bit elements of the same register, and add the results with a 64-bit value of an even/odd pair of registers (RV32) or a register (RV64). The addition result is written back to another even/odd pair of registers (RV32) or a register (RV64).

RV32 Description:

This instruction multiplies the bottom 16-bit content of the lower 32-bit of Rs2 with the top 16-bit content of the lower 32-bit of Rs2 and adds the result with the 64-bit value of an even/odd pair of registers specified by Rs1(4,1). The 64-bit addition result is written back to an even/odd pair of registers specified by Rd(4,1). The 16-bit values of Rs2, and the 64-bit value of the Rs1(4,1) register-pair are treated as signed integers.

$Rx(4,1)$, i.e., d , determines the even/odd pair group of the two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the operand and the even “ $2d$ ” register of the pair contains the low 32-bit of the operand.

RV64 Description:

This instruction multiplies the bottom 16-bit content of the 32-bit elements of Rs2 with the top 16-bit content of the same 32-bit elements of Rs2 and adds the results with the 64-bit value of Rs1. The 64-bit addition result is written back to Rd. The 16-bit values of Rs2, and the 64-bit value of Rs1 are treated as signed integers.

Operations:

RV32:

```
Mres[31:0] = Rs2.H[1] * Rs2.H[0];
Idx0 = CONCAT(Rs1(4,1),1'b0); Idx1 = CONCAT(Rs1(4,1),1'b1); +
Idx2 = CONCAT(Rd(4,1),1'b0); Idx3 = CONCAT(Rd(4,1),1'b1);
R[Idx3].R[Idx2] = R[Idx1].R[Idx0] + SE64(Mres[31:0]);
```

RV64:

```
Mres[0][31:0] = Rs2.W[0].H[1] * Rs2.W[0].H[0];
Mres[1][31:0] = Rs2.W[1].H[1] * Rs2.W[1].H[0];
Rd = Rs1 + SE64(Mres[1][31:0]) + SE64(Mres[0][31:0]);
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long long __nds__smal(long long a, unsigned long b);
```

RV32:

```
long long __nds__v_smal(long long a, int16x2_t b);
```

RV64:

```
long __nds__v_smal(long a, int16x4_t b);
```

4.105. SMALBB, SMALBT, SMALTT

4.105.1. SMALBB (Signed Multiply Bottom Halfs & Add 64-bit)

4.105.2. SMALBT (Signed Multiply Bottom Half & Top Half & Add 64-bit)

4.105.3. SMALTT (Signed Multiply Top Halfs & Add 64-bit)

Type: DSP (64-bit Profile)

Format:

SMALBB

31 25	24 20	19 15	14 12	11 7	6 0
SMALBB 1000100	Rs2	Rs1	001	Rd	GE80B 1111111

SMALBT

31 25	24 20	19 15	14 12	11 7	6 0
SMALBT 1001100	Rs2	Rs1	001	Rd	GE80B 1111111

SMALTT

31 25	24 20	19 15	14 12	11 7	6 0
SMALTT 1010100	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
SMALBB Rd, Rs1, Rs2
SMALBT Rd, Rs1, Rs2
SMALTT Rd, Rs1, Rs2
```

Purpose: Multiply the signed 16-bit content of the 32-bit elements of a register with the 16-bit content of the corresponding 32-bit elements of another register and add the results with a 64-bit value of an even/odd pair of registers (RV32) or a register (RV64). The addition result is written back to the register-pair (RV32) or the register (RV64).

- SMALBB: rt pair + bottom*bottom (all 32-bit elements)
- SMALBT rt pair + bottom*top (all 32-bit elements)
- SMALTT rt pair + top*top (all 32-bit elements)

RV32 Description:

For the “SMALBB” instruction, it multiplies the bottom 16-bit content of Rs1 with the bottom 16-bit content of Rs2.

For the “SMALBT” instruction, it multiplies the bottom 16-bit content of Rs1 with the top 16-bit content of Rs2.

For the “SMALTT” instruction, it multiplies the top 16-bit content of Rs1 with the top 16-bit content of Rs2.

The multiplication result is added with the 64-bit value of an even/odd pair of registers specified by Rd(4,1). The 64-bit addition result is written back to the register-pair. The 16-bit values of Rs1 and Rs2, and the 64-bit value of the register-pair are treated as signed integers.

Rd(4,1), i.e., d , determines the even/odd pair group of the two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the operand and the even “ $2d$ ” register of the pair contains the low 32-bit of the operand.

RV64 Description:

For the “SMALBB” instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2.

For the “SMALBT” instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2.

For the “SMALTT” instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2.

The multiplication results are added with the 64-bit value of Rd. The 64-bit addition result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

RV32:

```
Mres[31:0] = Rs1.H[0] * Rs2.H[0]; // SMALBB
Mres[31:0] = Rs1.H[0] * Rs2.H[1]; // SMALBT
Mres[31:0] = Rs1.H[1] * Rs2.H[1]; // SMALTT
Idx0 = CONCAT(Rd(4,1),1'b0); Idx1 = CONCAT(Rd(4,1),1'b1);
R[Idx1].R[Idx0] = R[Idx1].R[Idx0] + SE64(Mres[31:0]);
```

RV64:

```
// SMALBB
Mres[0][31:0] = Rs1.W[0].H[0] * Rs2.W[0].H[0];
Mres[1][31:0] = Rs1.W[1].H[0] * Rs2.W[1].H[0];
```

```
// SMALBT  
Mres[0][31:0] = Rs1.W[0].H[0] * Rs2.W[0].H[1];  
Mres[1][31:0] = Rs1.W[1].H[0] * Rs2.W[1].H[1];
```

```
// SMALTT  
Mres[0][31:0] = Rs1.W[0].H[1] * Rs2.W[0].H[1];  
Mres[1][31:0] = Rs1.W[1].H[1] * Rs2.W[1].H[1];  
Rd = Rd + SE64(Mres[0][31:0]) + SE64(Mres[1][31:0]);
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SMALBB

```
long long __nds__smalbb(long long t, unsigned long a, unsigned long b);
```

RV32:

```
long long __nds__v_smalbb(long long t, int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds__v_smalbb(long t, int16x4_t a, int16x4_t b);
```

- SMALBT

```
long long __nds__smalbt(long long t, unsigned long a, unsigned long b);
```

RV32:

```
long long __nds__v_smalbt(long long t, int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds__v_smalbt(long t, int16x4_t a, int16x4_t b);
```

- SMALTT

```
long long __nds__smaltt(long long t, unsigned long a, unsigned long b);
```

RV32:

```
long long __nds__v_smaltt(long long t, int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds__v_smaltt(long t, int16x4_t a, int16x4_t b);
```

4.106. SMALDA, SMALXDA

4.106.1. SMALDA (Signed Multiply Two Halfs and Two Adds 64-bit)

4.106.2. SMALXDA (Signed Crossed Multiply Two Halfs and Two Adds 64-bit)

Type: DSP (64-bit Profile)

Format:

SMALDA

31 25	24 20	19 15	14 12	11 7	6 0
SMALDA 1000110	Rs2	Rs1	001	Rd	GE80B 1111111

SMALXDA

31 25	24 20	19 15	14 12	11 7	6 0
SMALXDA 1001110	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
SMALDA Rd, Rs1, Rs2
SMALXDA Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then adds the two 32-bit results and the 64-bit value of an even/odd pair of registers together.

- SMALDA: rt pair+ top*top + bottom*bottom (all 32-bit elements)
- SMALXDA: rt pair+ top*bottom + bottom*top (all 32-bit elements)

RV32 Description:

For the “SMALDA” instruction, it multiplies the bottom 16-bit content of Rs1 with the bottom 16-bit content of Rs2 and then adds the result to the result of multiplying the top 16-bit content of Rs1 with the top 16-bit content of Rs2 with unlimited precision.

For the “SMALXDA” instruction, it multiplies the top 16-bit content of Rs1 with the bottom 16-bit content of Rs2 and then adds the result to the result of multiplying the bottom 16-bit content of Rs1 with the top 16-bit content of Rs2 with unlimited precision.

The result is added to the 64-bit value of an even/odd pair of registers specified by Rd(4,1). The 64-bit addition result is written back to the register-pair. The 16-bit values of Rs1 and Rs2, and the 64-bit value of the register-pair are treated as signed integers.

Rd(4,1), i.e., d , determines the even/odd pair group of the two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the operand and the even “ $2d$ ” register of the pair contains the low 32-bit of the operand.

RV64 Description:

For the “SMALDA” instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then adds the result to the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 with unlimited precision.

For the “SMALXDA” instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then adds the result to the result of multiplying the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 with unlimited precision.

The results are added to the 64-bit value of Rd. The 64-bit addition result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

RV32:

```
// SMALDA
Mres0[31:0] = (Rs1.H[0] * Rs2.H[0]);
Mres1[31:0] = (Rs1.H[1] * Rs2.H[1]);
```

```
// SMALXDA
Mres0[31:0] = (Rs1.H[0] * Rs2.H[1]);
Mres1[31:0] = (Rs1.H[1] * Rs2.H[0]);
Idx0 = CONCAT(Rd(4,1),1'b0); Idx1 = CONCAT(Rd(4,1),1'b1);
R[Idx1].R[Idx0] = R[Idx1].R[Idx0] + SE64(Mres0[31:0]) + SE64(Mres1[31:0]);
```

RV64:

```
// SMALDA
Mres0[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[0]);
Mres1[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[1]);
Mres0[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[1].H[0]);
Mres1[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[1].H[1]);
```

```

// SMALXDA
Mres0[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[1]);
Mres1[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[0]);
Mres0[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[1].H[1]);
Mres1[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[1].H[0]);
Rd = Rd + SE64(Mres0[0][31:0]) + SE64(Mres1[0][31:0]) + SE64(Mres0[1][31:0]) +
SE64(Mres1[1][31:0]);

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SMALDA

```
long long __nds__smalda(long long t, unsigned long a, unsigned long b);
```

RV32:

```
long long __nds__v_smalda(long long t, int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds__v_smalda(long t, int16x4_t a, int16x4_t b);
```

- SMALXDA

```
long long __nds__smalxda(long long t, unsigned long a, unsigned long b);
```

RV32:

```
long long __nds__v_smalxda(long long t, int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds__v_smalxda(long t, int16x4_t a, int16x4_t b);
```

4.107. SMALDS, SMALDRS, SMALXDS

4.107.1. SMALDS (Signed Multiply Two Halfs & Subtract & Add 64-bit)

4.107.2. SMALDRS (Signed Multiply Two Halfs & Reverse Subtract & Add 64-bit)

4.107.3. SMALXDS (Signed Crossed Multiply Two Halfs & Subtract & Add 64-bit)

Type: DSP (64-bit Profile)

Format:

SMALDS

31 25	24 20	19 15	14 12	11 7	6 0
SMALDS 1000101	Rs2	Rs1	001	Rd	GE80B 1111111

SMALDRS

31 25	24 20	19 15	14 12	11 7	6 0
SMALDRS 1001101	Rs2	Rs1	001	Rd	GE80B 1111111

SMALXDS

31 25	24 20	19 15	14 12	11 7	6 0
SMALXDS 1010101	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
SMALDS Rd, Rs1, Rs2
SMALDRS Rd, Rs1, Rs2
SMALXDS Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then perform a subtraction operation between the two 32-bit results. Then add the subtraction result to the 64-bit value of an even/odd pair of registers (RV32) or a register (RV64). The addition result is written back to the register-pair.

- SMALDS: rt pair + (top*top - bottom*bottom) (all 32-bit elements)
- SMALDRS: rt pair + (bottom*bottom - top*top) (all 32-bit elements)

- SMALXDS: rt pair + (top*bottom - bottom*top) (all 32-bit elements)

RV32 Description:

For the “SMALDS” instruction, it multiplies the bottom 16-bit content of Rs1 with the bottom 16-bit content of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of Rs1 with the top 16-bit content of Rs2.

For the “SMALDRS” instruction, it multiplies the top 16-bit content of Rs1 with the top 16-bit content of Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of Rs1 with the bottom 16-bit content of Rs2.

For the “SMALXDS” instruction, it multiplies the bottom 16-bit content of Rs1 with the top 16-bit content of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of Rs1 with the bottom 16-bit content of Rs2.

The subtraction result is then added to the 64-bit value of an even/odd pair of registers specified by Rd(4,1). The 64-bit addition result is written back to the register-pair. The 16-bit values of Rs1 and Rs2, and the 64-bit value of the register-pair are treated as signed integers.

Rd(4,1), i.e., d , determines the even/odd pair group of the two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the operand and the even “ $2d$ ” register of the pair contains the low 32-bit of the operand.

RV64 Description:

For the “SMALDS” instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2.

For the “SMALDRS” instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2.

For the “SMALXDS” instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2.

The subtraction results are then added to the 64-bit value of Rd. The 64-bit addition result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

- RV32:

```

Mres[31:0] = (Rs1.H[1] * Rs2.H[1]) - (Rs1.H[0] * Rs2.H[0]); // SMALDS
Mres[31:0] = (Rs1.H[0] * Rs2.H[0]) - (Rs1.H[1] * Rs2.H[1]); // SMALDRS
Mres[31:0] = (Rs1.H[1] * Rs2.H[0]) - (Rs1.H[0] * Rs2.H[1]); // SMALXDS
Idx0 = CONCAT(Rd(4,1),1'b0); Idx1 = CONCAT(Rd(4,1),1'b1);
R[Idx1].R[Idx0] = R[Idx1].R[Idx0] + SE64(Mres[31:0]);

```

- RV64:

```

// SMALDS
Mres[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[1]) - (Rs1.W[0].H[0] * Rs2.W[0].H[0]);
Mres[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[0].H[1]) - (Rs1.W[1].H[0] * Rs2.W[1].H[0]);

```

```

// SMALDRS
Mres[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[0]) - (Rs1.W[0].H[1] * Rs2.W[0].H[1]);
Mres[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[0].H[0]) - (Rs1.W[1].H[1] * Rs2.W[1].H[1]);

```

```

// SMALXDS
Mres[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[0]) - (Rs1.W[0].H[0] * Rs2.W[0].H[1]);
Mres[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[0].H[0]) - (Rs1.W[1].H[0] * Rs2.W[1].H[1]);
Rd = Rd + SE64(Mres[0][31:0]) + SE64(Mres[1][31:0]);

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SMALDS

```
long long __nds_smalds(long long t, unsigned long a, unsigned long b);
```

RV32:

```
long long __nds_v_smalds(long long t, int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds_v_smalds(long t, int16x4_t a, int16x4_t b);
```

- SMALDRS

```
long long __nds_smaldrs(long long t, unsigned long a, unsigned long b);
```

RV32:

```
long long __nds__v_smaldrs(long long t, int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds__v_smaldrs(long t, int16x4_t a, int16x4_t b);
```

- SMALXDS

```
long long __nds__smalxds(long long t, unsigned long a, unsigned long b);
```

RV32:

```
long long __nds__v_smalxds(long long t, int16x2_t a, int16x2_t b);
```

RV64:

```
long __nds__v_smalxds(long t, int16x4_t a, int16x4_t b);
```

4.108. SMAR64 (Signed Multiply and Add to 64-Bit Data)

Type: DSP (64-bit Profile)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SMAR64 1000010	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
SMAR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit signed elements in two registers and add the 64-bit multiplication result to the 64-bit signed data of a pair of registers (RV32) or a register (RV64). The result is written back to the pair of registers (RV32) or a register (RV64).

RV32 Description: This instruction multiplies the 32-bit signed data of Rs1 with that of Rs2. It adds the 64-bit multiplication result to the 64-bit signed data of an even/odd pair of registers specified by Rd(4,1). The addition result is written back to the even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction multiplies the 32-bit signed elements of Rs1 with that of Rs2. It adds the 64-bit multiplication results to the 64-bit signed data of Rd. The addition result is written back to Rd.

Operations:

- RV32:

```
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H].R[t_L] = R[t_H].R[t_L] + (Rs1 * Rs2);
```

- RV64:

```
Rd = Rd + (Rs1.W[0] * Rs2.W[0]) + (Rs1.W[1] * Rs2.W[1]);
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long long __nds__smar64(long long t, long a, long b);
```

RV64:

```
long __nds__v_smar64(long t, int32x2_t a, int32x2_t b);
```

4.109. SMAQA (Signed Multiply Four Bytes with 32-bit Adds)

Type: Partial-SIMD (Reduction)

Format:

SMAQA

31 25	24 20	19 15	14 12	11 7	6 0
SMAQA 1100100	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SMAQA Rd, Rs1, Rs2
```

Purpose: Do four signed 8-bit multiplications from 32-bit chunks of two registers; and then adds the four 16-bit results and the content of corresponding 32-bit chunks of a third register together.

Description:

This instruction multiplies the four signed 8-bit elements of 32-bit chunks of Rs1 with the four signed 8-bit elements of 32-bit chunks of Rs2 and then adds the four results together with the signed content of the corresponding 32-bit chunks of Rd. The final results are written back to the corresponding 32-bit chunks in Rd.

Operations:

```
res[x] = Rd.W[x] +  
        (Rs1.W[x].B[3] s* Rs2.W[x].B[3]) + (Rs1.W[x].B[2] s* Rs2.W[x].B[2]) +  
        (Rs1.W[x].B[1] s* Rs2.W[x].B[1]) + (Rs1.W[x].B[0] s* Rs2.W[x].B[0]);  
Rd.W[x] = res[x];  
for RV32: x=0,  
for RV64: x=1,0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long __nds_smaqa(long t, unsigned long a, unsigned long b);
```

RV32:

```
int __nds__v_smaqa(int t, int8x4_t a, int8x4_t b);
```

RV64:

```
int32x2_t __nds__v_smaqa(int32x2_t t, int8x8_t a, int8x8_t b);
```

4.110. SMAQA.SU (Signed and Unsigned Multiply Four Bytes with 32-bit Adds)

Type: Partial-SIMD (Reduction)

Format:

SMAQA.SU

31 25	24 20	19 15	14 12	11 7	6 0
SMAQA.SU 1100101	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SMAQA.SU Rd, Rs1, Rs2
```

Purpose: Do four “signed x unsigned” 8-bit multiplications from 32-bit chunks of two registers; and then adds the four 16-bit results and the content of corresponding 32-bit chunks of a third register together.

Description:

This instruction multiplies the four signed 8-bit elements of 32-bit chunks of Rs1 with the four unsigned 8-bit elements of 32-bit chunks of Rs2 and then adds the four results together with the signed content of the corresponding 32-bit chunks of Rd. The final results are written back to the corresponding 32-bit chunks in Rd.

Operations:

```
res[x] = Rd.W[x] +  
        (Rs1.W[x].B[3] su* Rs2.W[x].B[3]) + (Rs1.W[x].B[2] su* Rs2.W[x].B[2]) +  
        (Rs1.W[x].B[1] su* Rs2.W[x].B[1]) + (Rs1.W[x].B[0] su* Rs2.W[x].B[0]);  
Rd.W[x] = res[x];  
for RV32: x=0,  
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long __nds_smaqa_su(long t, unsigned long a, unsigned long b);
```

RV32:

```
int __nds__v_smaqa_su (int t, int8x4_t a, int8x4_t b);
```

RV64:

```
int32x2_t __nds__v_smaqa_su (int32x2_t t, int8x8_t a, int8x8_t b);
```

4.111. SMAX8 (SIMD 8-bit Signed Maximum)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SMAX8 1000101	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SMAX8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit signed integer elements finding maximum operations simultaneously.

Description: This instruction compares the 8-bit signed integer elements in Rs1 with the 8-bit signed integer elements in Rs2 and selects the numbers that is greater than the other one. The selected results are written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] > Rs2.B[x])? Rs1.B[x] : Rs2.B[x];  
for RV32: x=3...0,  
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_smax8(unsigned long a, unsigned long b);
```

RV32:

```
int8x4_t __nds_v_smax8(int8x4_t a, int8x4_t b);
```

RV64:

```
int8x8_t __nds_v_smax8(int8x8_t a, int8x8_t b);
```

4.112. SMAX16 (SIMD 16-bit Signed Maximum)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SMAX16 1000001	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SMAX16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer elements finding maximum operations simultaneously.

Description: This instruction compares the 16-bit signed integer elements in Rs1 with the 16-bit signed integer elements in Rs2 and selects the numbers that is greater than the other one. The selected results are written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] > Rs2.H[x])? Rs1.H[x] : Rs2.H[x];  
for RV32: x=1…0,  
for RV64: x=3…0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_smax16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds_v_smax16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __nds_v_smax16(int16x4_t a, int16x4_t b);
```

4.113. SMBB16, SMBT16, SMTT16

4.113.1. SMBB16 (SIMD Signed Multiply Bottom Half & Bottom Half)

4.113.2. SMBT16 (SIMD Signed Multiply Bottom Half & Top Half)

4.113.3. SMTT16 (SIMD Signed Multiply Top Half & Top Half)

Type: SIMD

Format:

SMBB16

31 25	24 20	19 15	14 12	11 7	6 0
SMBB16 0000100	Rs2	Rs1	001	Rd	GE80B 1111111

SMBT16

31 25	24 20	19 15	14 12	11 7	6 0
SMBT16 0001100	Rs2	Rs1	001	Rd	GE80B 1111111

SMTT16

31 25	24 20	19 15	14 12	11 7	6 0
SMTT16 0010100	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
SMBB16 Rd, Rs1, Rs2
SMBT16 Rd, Rs1, Rs2
SMTT16 Rd, Rs1, Rs2
```

Purpose: Multiply the signed 16-bit content of the 32-bit elements of a register with the signed 16-bit content of the 32-bit elements of another register and write the result to a third register.

- SMBB16: W[x].bottom * W[x].bottom
- SMBT16: W[x].bottom * W[x].top
- SMTT16: W[x].top * W[x].top

Description:

For the “SMBB16” instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1

with the bottom 16-bit content of the 32-bit elements of Rs2.

For the “SMBT16” instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2.

For the “SMTT16” instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2.

The multiplication results are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
Rd.W[x] = Rs1.W[x].H[0] * Rs2.W[x].H[0]; // SMBB16  
Rd.W[x] = Rs1.W[x].H[0] * Rs2.W[x].H[1]; // SMBT16  
Rd.W[x] = Rs1.W[x].H[1] * Rs2.W[x].H[1]; // SMTT16  
for RV32: x=0,  
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SMBB16

```
long __nds__smbb16(unsigned long a, unsigned long b);
```

RV32:

```
int __nds__v_smbb16(int16x2_t a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_smbb16(int16x4_t a, int16x4_t b);
```

- SMBT16

```
long __nds__smbt16(unsigned long a, unsigned long b);
```

RV32:

```
int __nds__v_smbt16(int16x2_t a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_smbt16(int16x4_t a, int16x4_t b);
```

- SMTT16

```
long __nds__smtt16(unsigned long a, unsigned long b);
```

RV32:

```
int __nds__v_smtt16(int16x2_t a, int16x2_t b);
```

RV64:

```
int32x2_t __nds__v_smtt16(int16x4_t a, int16x4_t b);
```

4.114. SMDS, SMDRS, SMXDS

4.114.1. SMDS (SIMD Signed Multiply Two Halfs and Subtract)

4.114.2. SMDRS (SIMD Signed Multiply Two Halfs and Reverse Subtract)

4.114.3. SMXDS (SIMD Signed Crossed Multiply Two Halfs and Subtract)

Type: SIMD

Format:

SMDS

31 25	24 20	19 15	14 12	11 7	6 0
SMDS 0101100	Rs2	Rs1	001	Rd	GE80B 1111111

SMDRS

31 25	24 20	19 15	14 12	11 7	6 0
SMDRS 0110100	Rs2	Rs1	001	Rd	GE80B 1111111

SMXDS

31 25	24 20	19 15	14 12	11 7	6 0
SMXDS 0111100	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
SMDS Rd, Rs1, Rs2
SMDRS Rd, Rs1, Rs2
SMXDS Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then perform a subtraction operation between the two 32-bit results.

- SMDS: top*top - bottom*bottom (per 32-bit element)
- SMDRS: bottom*bottom - top*top (per 32-bit element)
- SMXDS: top*bottom - bottom*top (per 32-bit element)

Description:

For the “SMDS” instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with

the bottom 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2.

For the “SMDRS” instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2.

For the “SMXDS” instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2.

The subtraction result is written to the corresponding 32-bit element of Rd. The 16-bit contents of multiplication are treated as signed integers.

Operations:

- SMDS:

$$Rd.W[x] = (Rs1.W[x].H[1] * Rs2.W[x].H[1]) - (Rs1.W[x].H[0] * Rs2.W[x].H[0]);$$

- SMDRS:

$$Rd.W[x] = (Rs1.W[x].H[0] * Rs2.W[x].H[0]) - (Rs1.W[x].H[1] * Rs2.W[x].H[1]);$$

- SMXDS:

$$Rd.W[x] = (Rs1.W[x].H[1] * Rs2.W[x].H[0]) - (Rs1.W[x].H[0] * Rs2.W[x].H[1]);$$

Exceptions: None

Privilege level: All

Note:

- Usage domain: Complex, Statistics, Transform

Intrinsic functions:

- SMDS

```
long __nds__smds(unsigned long a, unsigned long b);
```

```
RV32:  
    int __nds__v_smds(int16x2_t a, int16x2_t b);  
RV64:  
    int32x2 __nds__v_smds(int16x4_t a, int16x4_t b);
```

- SMDRS

```
long __nds__smdrs(unsigned long a, unsigned long b);
```

```
RV32:  
    int __nds__v_smdrs(int16x2_t a, int16x2_t b);  
RV64:  
    int32x2 __nds__v_smdrs(int16x4_t a, int16x4_t b);
```

- SMXDS

```
long __nds__smxds(unsigned long a, unsigned long b);
```

```
RV32:  
    int __nds__v_smxds(int16x2_t a, int16x2_t b);  
RV64:  
    int32x2 __nds__v_smxds(int16x4_t a, int16x4_t b);
```

4.115. SMIN8 (SIMD 8-bit Signed Minimum)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SMIN8 1000100	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SMIN8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit signed integer elements finding minimum operations simultaneously.

Description: This instruction compares the 8-bit signed integer elements in Rs1 with the 8-bit signed integer elements in Rs2 and selects the numbers that is less than the other one. The selected results are written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] < Rs2.B[x])? Rs1.B[x] : Rs2.B[x];  
for RV32: x=3...0,  
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_smin8(unsigned long a, unsigned long b);
```

RV32:

```
int8x4_t __nds_v_smin8(int8x4_t a, int8x4_t b);
```

RV64:

```
int8x8_t __nds_v_smin8(int8x8_t a, int8x8_t b);
```

4.116. SMIN16 (SIMD 16-bit Signed Minimum)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SMIN16 1000000	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SMIN16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer elements finding minimum operations simultaneously.

Description: This instruction compares the 16-bit signed integer elements in Rs1 with the 16-bit signed integer elements in Rs2 and selects the numbers that is less than the other one. The selected results are written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] < Rs2.H[x])? Rs1.H[x] : Rs2.H[x];  
for RV32: x=1…0,  
for RV64: x=3…0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_smin16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds_v_smin16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __nds_v_smin16(int16x4_t a, int16x4_t b);
```

4.117. SMMUL, SMMUL.u

4.117.1. SMMUL (SIMD MSW Signed Multiply Word)

4.117.2. SMMUL.u (SIMD MSW Signed Multiply Word with Rounding)

Type: SIMD

Format:

SMMUL

31 25	24 20	19 15	14 12	11 7	6 0
SMMUL 0100000	Rs2	Rs1	001	Rd	GE80B 1111111

SMMUL.u

31 25	24 20	19 15	14 12	11 7	6 0
SMMUL.u 0101000	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
SMMUL Rd, Rs1, Rs2
SMMUL.u Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit signed integer elements of two registers and write the most significant 32-bit results to the corresponding 32-bit elements of a register. The “.u” form performs an additional rounding up operation on the multiplication results before taking the most significant 32-bit part of the results.

Description:

This instruction multiplies the 32-bit elements of Rs1 with the 32-bit elements of Rs2 and writes the most significant 32-bit multiplication results to the corresponding 32-bit elements of Rd. The 32-bit elements of Rs1 and Rs2 are treated as signed integers. The “.u” form of the instruction rounds up the most significant 32-bit of the 64-bit multiplication results by adding a 1 to bit 31 of the results.

- For “smmul/RV32” instruction, it is an alias to “mulh/RV32” instruction.

Operations:

```

Mres[x][63:0] = Rs1.W[x] * Rs2.W[x];
if (x.u form) {
    Round[x][32:0] = Mres[x][63:31] + 1;
    Rd.W[x] = Round[x][32:1];
} else {
    Rd.W[x] = Mres[x][63:32];
}
for RV32: x=0
for RV64: x=1...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SMMUL

```
long __nds__smmul(long a, long b);
```

RV64:

```
int32x2_t __nds__v_smmul(int32x2_t a, int32x2_t b);
```

- SMMUL.u

```
long __nds__smmul_u(long a, long b);
```

RV64:

```
int32x2_t __nds__v_smmul_u(int32x2_t a, int32x2_t b);
```

4.118. SMMWB, SMMWB.u

4.118.1. SMMWB (SIMD MSW Signed Multiply Word and Bottom Half)

4.118.2. SMMWB.u (SIMD MSW Signed Multiply Word and Bottom Half with Rounding)

Type: SIMD

Format:

SMMWB

31 25	24 20	19 15	14 12	11 7	6 0
SMMWB 0100010	Rs2	Rs1	001	Rd	GE80B 1111111

SMMWB.u

31 25	24 20	19 15	14 12	11 7	6 0
SMMWB.u 0101010	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
SMMWB Rd, Rs1, Rs2
SMMWB.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of one register and the bottom 16-bit of the corresponding 32-bit elements of another register, and write the most significant 32-bit results to the corresponding 32-bit elements of a register. The “.u” form rounds up the results from the most significant discarded bit.

Description:

This instruction multiplies the signed 32-bit elements of Rs1 with the signed bottom 16-bit content of the corresponding 32-bit elements of Rs2 and writes the most significant 32-bit multiplication results to the corresponding 32-bit elements of Rd. The “.u” form of the instruction rounds up the most significant 32-bit of the 48-bit multiplication results by adding a 1 to bit 15 of the results.

Operations:

```

Mres[x][47:0] = Rs1.W[x] * Rs2.W[x].H[0];
if (!.u) form {
    Round[x][32:0] = Mres[x][47:15] + 1;
    Rd.W[x] = Round[x][32:1];
} else {
    Rd.W[x] = Mres[x][47:16];
}
for RV32: x=0
for RV64: x=1...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SMMWB

```
long __nds__smmwb(long a, unsigned long b);
```

```

RV32:
int __nds__v_smmwb(int a, int16x2_t b);
RV64:
int32x2_t __nds__v_smmwb(int32x2_t a, int16x4_t b);

```

- SMMWB.u

```
long __nds__smmwb_u(long a, unsigned long b);
```

```

RV32:
int __nds__v_smmwb_u(int a, int16x2_t b);
RV64:
int32x2_t __nds__v_smmwb_u(int32x2_t a, int16x4_t b);

```

4.119. SMMWT, SMMWT.u

4.119.1. SMMWT (SIMD MSW Signed Multiply Word and Top Half)

4.119.2. SMMWT.u (SIMD MSW Signed Multiply Word and Top Half with Rounding)

Type: SIMD

Format:

SMMWT

31 25	24 20	19 15	14 12	11 7	6 0
SMMWT 0110010	Rs2	Rs1	001	Rd	GE80B 1111111

SMMWT.u

31 25	24 20	19 15	14 12	11 7	6 0
SMMWT.u 0111010	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
SMMWT Rd, Rs1, Rs2
SMMWT.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of one register and the top 16-bit of the corresponding 32-bit elements of another register, and write the most significant 32-bit results to the corresponding 32-bit elements of a register. The “.u” form rounds up the results from the most significant discarded bit.

Description:

This instruction multiplies the signed 32-bit elements of Rs1 with the top signed 16-bit content of the corresponding 32-bit elements of Rs2 and writes the most significant 32-bit multiplication results to the corresponding 32-bit elements of Rd. The “.u” form of the instruction rounds up the most significant 32-bit of the 48-bit multiplication results by adding a 1 to bit 15 of the results.

Operations:

```

Mres[x][47:0] = Rs1.W[x] * Rs2.W[x].H[1];
if (!.u) form {
    Round[x][32:0] = Mres[x][47:15] + 1;
    Rd.W[x] = Round[x][32:1];
} else {
    Rd.W[x] = Mres[x][47:16];
}
for RV32: x=0
for RV64: x=1...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SMMWT

```
long __nds__smmwta(long a, unsigned long b);
```

```

RV32:
int __nds__v_smmwt(int a, int16x2_t b);
RV64:
int32x2_t __nds__v_smmwt(int32x2_t a, int16x4_t b);

```

- SMMWT.u

```
long __nds__smmwta_u(long a, unsigned long b);
```

```

RV32:
int __nds__v_smmwt_u(int a, int16x2_t b);
RV64:
int32x2_t __nds__v_smmwt_u(int32x2_t a, int16x4_t b);

```

4.120. SMSLDA, SMSLXDA

4.120.1. SMSLDA (Signed Multiply Two Halfs & Add & Subtract 64-bit)

4.120.2. SMSLXDA (Signed Crossed Multiply Two Halfs & Add & Subtract 64-bit)

Type: DSP (64-bit Profile)

Format:

SMSLDA

31 25	24 20	19 15	14 12	11 7	6 0
SMSLDA 1010110	Rs2	Rs1	001	Rd	GE80B 1111111

SMSLXDA

31 25	24 20	19 15	14 12	11 7	6 0
SMSLXDA 1011110	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
SMSLDA Rd, Rs1, Rs2
SMSLXDA Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then subtracts the two 32-bit results from the 64-bit value of an even/odd pair of registers (RV32) or a register (RV64). The subtraction result is written back to the register-pair.

- SMSLDA: rd pair - top*top - bottom*bottom (all 32-bit elements)
- SMSLXDA: rd pair - top*bottom - bottom*top (all 32-bit elements)

RV32 Description:

For the “SMSLDA” instruction, it multiplies the bottom 16-bit content of Rs1 with the bottom 16-bit content Rs2 and multiplies the top 16-bit content of Rs1 with the top 16-bit content of Rs2.

For the “SMSLXDA” instruction, it multiplies the top 16-bit content of Rs1 with the bottom 16-bit content of Rs2 and multiplies the bottom 16-bit content of Rs1 with the top 16-bit content of Rs2.

The two multiplication results are subtracted from the 64-bit value of an even/odd pair of registers specified by Rd(4,1). The 64-bit subtraction result is written back to the register-pair. The 16-bit values of Rs1 and Rs2, and the 64-bit value of the register-pair are treated as signed integers.

Rd(4,1), i.e., d , determines the even/odd pair group of the two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

RV64 Description:

For the “SMSLDA” instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2.

For the “SMSLXDA” instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2.

The four multiplication results are subtracted from the 64-bit value of Rd. The 64-bit subtraction result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

- RV32:

```
// SMSLDA
Mres0[31:0] = (Rs1.H[0] * Rs2.H[0]);
Mres1[31:0] = (Rs1.H[1] * Rs2.H[1]);
// SMSLXDA
Mres0[31:0] = (Rs1.H[0] * Rs2.H[1]);
Mres1[31:0] = (Rs1.H[1] * Rs2.H[0]);
Idx0 = CONCAT(Rd(4,1),1'b0); Idx1 = CONCAT(Rd(4,1),1'b1);
R[Idx1].R[Idx0] = R[Idx1].R[Idx0] - SE64(Mres0[31:0]) - SE64(Mres1[31:0]);
```

- RV64:

```
// SMSLDA
Mres0[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[0]);
Mres1[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[1]);
Mres0[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[1].H[0]);
Mres1[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[1].H[1]);
// SMSLXDA
Mres0[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[1]);
Mres1[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[0]);
Mres0[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[1].H[1]);
Mres1[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[1].H[0]);
Rd = Rd - SE64(Mres0[0][31:0]) - SE64(Mres1[0][31:0]) - SE64(Mres0[1][31:0]) -
SE64(Mres1[1][31:0]);
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SMSLDA

```
long long __nds__smslda(long long t, unsigned long a, unsigned long b);
```

RV32:

```
long long __nds__v_smslda(long long t, int16x2_t a, int16x2_t b);
```

RV64:

```
long long __nds__v_smslda(long long t, int16x4_t a, int16x4_t b);
```

- SMLSXDA

```
long long __nds__smslxda(long long t, unsigned long a, unsigned long b);
```

RV32:

```
long long __nds__v_smslxda(long long t, int16x2_t a, int16x2_t b);
```

RV64:

```
long long __nds__v_smslxda(long long t, int16x4_t a, int16x4_t b);
```

4.121. SMSR64 (Signed Multiply and Subtract from 64-Bit Data)

Type: DSP (64-bit Profile)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SMSR64 1000011	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
SMSR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit signed elements in two registers and subtract the 64-bit multiplication results from the 64-bit signed data of a pair of registers (RV32) or a register (RV64). The result is written back to the pair of registers (RV32) or a register (RV64).

RV32 Description: This instruction multiplies the 32-bit signed data of Rs1 with that of Rs2. It subtracts the 64-bit multiplication result from the 64-bit signed data of an even/odd pair of registers specified by Rd(4,1). The subtraction result is written back to the even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction multiplies the 32-bit signed elements of Rs1 with that of Rs2. It subtracts the 64-bit multiplication results from the 64-bit signed data of Rd. The subtraction result is written back to Rd.

Operations:

- RV32:

```
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H].R[t_L] = R[t_H].R[t_L] - (Rs1 * Rs2);
```

- RV64:

```
Rd = Rd - (Rs1.W[0] * Rs2.W[0]) - (Rs1.W[1] * Rs2.W[1]);
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long long __nds__smsr64(long long t, long a, long b);
```

RV64:

```
long __nds__v_smsr64(long t, int32x2_t a, int32x2_t b);
```

4.122. SMUL8, SMULX8

4.122.1. SMUL8 (SIMD Signed 8-bit Multiply)

4.122.2. SMULX8 (SIMD Signed Crossed 8-bit Multiply)

Type: SIMD

Format:

SMUL8

31 25	24 20	19 15	14 12	11 7	6 0
SMUL8 1010100	Rs2	Rs1	000	Rd	GE80B 1111111

SMULX8

31 25	24 20	19 15	14 12	11 7	6 0
SMULX8 1010101	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SMUL8 Rd, Rs1, Rs2
SMULX8 Rd, Rs1, Rs2
```

Purpose: Do signed 8-bit multiplications and generate four 16-bit results simultaneously.

RV32 Description: For the “SMUL8” instruction, multiply the 8-bit data elements of Rs1 with the corresponding 8-bit data elements of Rs2.

For the “SMULX8” instruction, multiply the *first* and *second* 8-bit data elements of Rs1 with the *second* and *first* 8-bit data elements of Rs2. At the same time, multiply the *third* and *fourth* 8-bit data elements of Rs1 with the *fourth* and *third* 8-bit data elements of Rs2.

The four 16-bit results are then written into an even/odd pair of registers specified by Rd(4,1). Rd(4,1), i.e., d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the two 16-bit results calculated from the top part of Rs1 and the even “ $2d$ ” register of the pair contains the two 16-bit results calculated from the bottom part of Rs1.

RV64 Description: For the “SMUL8” instruction, multiply the 8-bit data elements of Rs1 with the corresponding 8-bit data elements of Rs2.

For the “SMULX8” instruction, multiply the *first* and *second* 8-bit data elements of Rs1 with the

second and *first* 8-bit data elements of Rs2. At the same time, multiply the *third* and *fourth* 8-bit data elements of Rs1 with the *fourth* and *third* 8-bit data elements of Rs2.

The four 16-bit results are then written into Rd. The Rd.W[1] contains the two 16-bit results calculated from the top part of Rs1 and the Rd.W[0] contains the two 16-bit results calculated from the bottom part of Rs1.

Operations:

- RV32:

```
if (is SMUL8) {
    op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x+1]; // top
    op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x]; // bottom
} else if (is SMULX8) {
    op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x]; // Rs1 top
    op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x+1]; // Rs1 bottom
}
rest[x/2] = op1t[x/2] s* op2t[x/2];
resb[x/2] = op1b[x/2] s* op2b[x/2];
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H].H[1] = rest[1]; R[t_H].H[0] = resb[1];
R[t_L].H[1] = rest[0]; R[t_L].H[0] = resb[0];
x = 0 and 2
```

- RV64:

```
if (is SMUL8) {
    op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x+1]; // top
    op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x]; // bottom
} else if (is SMULX8) {
    op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x]; // Rs1 top
    op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x+1]; // Rs1 bottom
}
rest[x/2] = op1t[x/2] s* op2t[x/2];
resb[x/2] = op1b[x/2] s* op2b[x/2];
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
Rd.W[1].H[1] = rest[1]; Rd.W[1].H[0] = resb[1];
Rd.W[0].H[1] = rest[0]; Rd.W[0].H[0] = resb[0];
x = 0 and 2
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SMUL8

```
unsigned long long __nds__smul8(unsigned int a, unsigned int b);
```

```
int16x4_t __nds__v_smul8(int8x4_t a, int8x4_t b);
```

- SMULX8

```
unsigned long long __nds__smulx8(unsigned int a, unsigned int b);
```

```
int16x4_t __nds__v_smulx8(int8x4_t a, int8x4_t b);
```

4.123. SMUL16, SMULX16

4.123.1. SMUL16 (SIMD Signed 16-bit Multiply)

4.123.2. SMULX16 (SIMD Signed Crossed 16-bit Multiply)

Type: SIMD

Format:

SMUL16

31 25	24 20	19 15	14 12	11 7	6 0
SMUL16 1010000	Rs2	Rs1	000	Rd	GE80B 1111111

SMULX16

31 25	24 20	19 15	14 12	11 7	6 0
SMULX16 1010001	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SMUL16 Rd, Rs1, Rs2
SMULX16 Rd, Rs1, Rs2
```

Purpose: Do signed 16-bit multiplications and generate two 32-bit results simultaneously.

RV32 Description: For the “SMUL16” instruction, multiply the top 16-bit Q15 content of Rs1 with the top 16-bit Q15 content of Rs2. At the same time, multiply the bottom 16-bit Q15 content of Rs1 with the bottom 16-bit Q15 content of Rs2.

For the “SMULX16” instruction, multiply the top 16-bit Q15 content of Rs1 with the bottom 16-bit Q15 content of Rs2. At the same time, multiply the bottom 16-bit Q15 content of Rs1 with the top 16-bit Q15 content of Rs2.

The two Q30 results are then written into an even/odd pair of registers specified by Rd(4,1). Rd(4,1), i.e., d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the 32-bit result calculated from the top part of Rs1 and the even “ $2d$ ” register of the pair contains the 32-bit result calculated from the bottom part of Rs1.

RV64 Description: For the “SMUL16” instruction, multiply the top 16-bit Q15 content of the lower 32-bit word in Rs1 with the top 16-bit Q15 content of the lower 32-bit word in Rs2. At the same time, multiply the bottom 16-bit Q15 content of the lower 32-bit word in Rs1 with the bottom 16-bit Q15 content of the lower 32-bit word in Rs2.

For the “SMULX16” instruction, multiply the top 16-bit Q15 content of the lower 32-bit word in Rs1 with the bottom 16-bit Q15 content of the lower 32-bit word in Rs2. At the same time, multiply the bottom 16-bit Q15 content of the lower 32-bit word in Rs1 with the top 16-bit Q15 content of the lower 32-bit word in Rs2.

The two 32-bit Q30 results are then written into Rd. The result calculated from the top 16-bit of the lower 32-bit word in Rs1 is written to Rd.W[1]. And the result calculated from the bottom 16-bit of the lower 32-bit word in Rs1 is written to Rd.W[0]

Operations:

- RV32:

```

if (is SMUL16) {
    op1t = Rs1.H[1]; op2t = Rs2.H[1]; // top
    op1b = Rs1.H[0]; op2b = Rs2.H[0]; // bottom
} else if (is SMULX16) {
    op1t = Rs1.H[1]; op2t = Rs2.H[0]; // Rs1 top
    op1b = Rs1.H[0]; op2b = Rs2.H[1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
    res = aop s* bop;
}
t_L = CONCAT(Rd(4,1),1b0); t_H = CONCAT(Rd(4,1),1b1);
R[t_H] = rest;
R[t_L] = resb;

```

- RV64:

```

if (is SMUL16) {
    op1t = Rs1.H[1]; op2t = Rs2.H[1]; // top
    op1b = Rs1.H[0]; op2b = Rs2.H[0]; // bottom
} else if (is SMULX16) {
    op1t = Rs1.H[1]; op2t = Rs2.H[0]; // Rs1 top
    op1b = Rs1.H[0]; op2b = Rs2.H[1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
    res = aop s* bop;
}
Rd.W[1] = rest;
Rd.W[0] = resb;

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SMUL16

```
unsigned long long __nds__smul16(unsigned int a, unsigned int b);
```

```
int32x2_t __nds__v_smul16(int16x2_t a, int16x2_t b);
```

- SMULX16

```
unsigned long long __nds__smulx16(unsigned int a, unsigned int b);
```

```
int32x2_t __nds__v_smulx16(int16x2_t a, int16x2_t b);
```

4.124. SRA.u (Rounding Shift Right Arithmetic)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SRA.u 0010010	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
SRA.u Rd, Rs1, Rs2
```

Purpose: Perform an arithmetic right shift operation with rounding. The shift amount is a variable from a GPR.

Description: This instruction right-shifts the content of Rs1 arithmetically. The shifted out bits are filled with the sign-bit and the shift amount is specified by the low-order 5-bits (RV32) or 6-bits (RV64) of the Rs2 register. For the rounding operation, a value of 1 is added to the most significant discarded bit of the data to calculate the final result. And the result is written to Rd.

Operations:

- RV32:

```
sa = Rs2[4:0];
if (sa > 0) {
    res[31:-1] = SE33(Rs1[31:(sa-1)]) + 1;
    Rd = res[31:0];
} else {
    Rd = Rs1;
}
```

- RV64:

```
sa = Rs2[5:0];
if (sa > 0) {
    res[63:-1] = SE65(Rs1[63:(sa-1)]) + 1;
    Rd = res[63:0];
} else {
    Rd = Rs1;
}
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long __nds__sra_u(long a, unsigned int b);
```

4.125. SRAI.u (Rounding Shift Right Arithmetic Immediate)

Type: DSP

Format:

31 26	25 20	19 15	14 12	11 7	6 0
SRAI.u 110101	imm6u[5:0]	Rs1	001	Rd	GE80B 1111111

Syntax:

```
SRAI.u Rd, Rs1, imm6u[4:0] (RV32)
SRAI.u Rd, Rs1, imm6u[5:0] (RV64)
```

Purpose: Perform an arithmetic right shift operation with rounding. The shift amount is an immediate value.

Description: This instruction right-shifts the content of Rs1 arithmetically. The shifted out bits are filled with the sign-bit and the shift amount is specified by the imm6u[4:0] (RV32) or imm6u[5:0] (RV64) constant . For the rounding operation, a value of 1 is added to the most significant discarded bit of the data to calculate the final result. And the result is written to Rd.

Operations:

- RV32:

```
sa = imm6u[4:0];
if (sa > 0) {
    res[31:-1] = SE33(Rs1[31:(sa-1)]) + 1;
    Rd = res[31:0];
} else {
    Rd = Rs1;
}
```

- RV64:

```
sa = imm6u[5:0];
if (sa > 0) {
    res[63:-1] = SE65(Rs1[63:(sa-1)]) + 1;
    Rd = res[63:0];
} else {
    Rd = Rs1;
}
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long __nds__sra_u(long a, unsigned int b);
```

4.126. SRA8, SRA8.u

4.126.1. SRA8 (SIMD 8-bit Shift Right Arithmetic)

4.126.2. SRA8.u (SIMD 8-bit Rounding Shift Right Arithmetic)

Type: SIMD

Format:

SRA8

31 25	24 20	19 15	14 12	11 7	6 0
SRA8 0101100	Rs2	Rs1	000	Rd	GE80B 1111111

SRA8.u

31 25	24 20	19 15	14 12	11 7	6 0
SRA8.u 0110100	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SRA8 Rd, Rs1, Rs2  
SRA8.u Rd, Rs1, Rs2
```

Purpose: Do 8-bit element arithmetic right shift operations simultaneously. The shift amount is a variable from a GPR. The “.u” form performs additional rounding up operations on the shifted results.

Description: The 8-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the data elements. The shift amount is specified by the low-order 3-bits of the value in the Rs2 register. For the rounding operation of the “.u” form, a value of 1 is added to the most significant discarded bit of each 8-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```

sa = Rs2[2:0];
if (sa > 0) {
    if (!.u[] form) { // SRA8.u
        res[7:-1] = SE9(Rs1.B[x][7:sa-1]) + 1;
        Rd.B[x] = res[7:0];
    } else { // SRA8
        Rd.B[x] = SE8(Rd.B[x][7:sa])
    }
} else {
    Rd = Rs1;
}
for RV32: x=3···0,
for RV64: x=7···0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SRA8

```
unsigned long __nds__sra8(unsigned long a, unsigned int b);
```

RV32:

```
int8x4_t __nds__v_sra8(int8x4_t a, unsigned int b);
```

RV64:

```
int8x8_t __nds__v_sra8(int8x8_t a, unsigned int b);
```

- SRA8.u

```
unsigned long __nds__sra8_u(unsigned long a, unsigned int b);
```

RV32:

```
int8x4_t __nds__v_sra8_u(int8x4_t a, unsigned int b);
```

RV64:

```
int8x8_t __nds__v_sra8_u(int8x8_t a, unsigned int b);
```

4.127. SRAI8, SRAI8.u

4.127.1. SRAI8 (SIMD 8-bit Shift Right Arithmetic Immediate)

4.127.2. SRAI8.u (SIMD 8-bit Rounding Shift Right Arithmetic Immediate)

Type: SIMD

Format:

SRAI8

31 25	24 23	22 20	19 15	14 12	11 7	6 0
SRAI8 0111100	00	imm3u[2:0]	Rs1	000	Rd	GE80B 1111111

SRAI8.u

31 25	24 23	22 20	19 15	14 12	11 7	6 0
SRAI8.u 0111100	01	imm3u[2:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SRAI8 Rd, Rs1, imm3u
SRAI8.u Rd, Rs1, imm3u
```

Purpose: Do 8-bit element arithmetic right shift operations simultaneously. The shift amount is an immediate value. The “.u” form performs additional rounding up operations on the shifted results.

Description: The 8-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the data elements. The shift amount is specified by the imm3u constant. For the rounding operation of the “.u” form, a value of 1 is added to the most significant discarded bit of each 8-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```

sa = imm3u[2:0];
if (sa > 0) {
    if (!.u[] form) { // SRA8.u
        res[7:-1] = SE9(Rs1.B[x][7:sa-1]) + 1;
        Rd.B[x] = res[7:0];
    } else { // SRA8
        Rd.B[x] = SE8(Rd.B[x][7:sa])
    }
} else {
    Rd = Rs1;
}
for RV32: x=3...0,
for RV64: x=7...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SRAI8

```
unsigned long __nds_sra8(unsigned long a, unsigned int b);
```

RV32:

```
int8x4_t __nds_v_sra8(int8x4_t a, unsigned int b);
```

RV64:

```
int8x8_t __nds_v_sra8(int8x8_t a, unsigned int b);
```

- SRAI8.u

```
unsigned long __nds_sra8_u(unsigned long a, unsigned int b);
```

RV32:

```
int8x4_t __nds_v_sra8_u(int8x4_t a, unsigned int b);
```

RV64:

```
int8x8_t __nds_v_sra8_u(int8x8_t a, unsigned int b);
```

4.128. SRA16, SRA16.u

4.128.1. SRA16 (SIMD 16-bit Shift Right Arithmetic)

4.128.2. SRA16.u (SIMD 16-bit Rounding Shift Right Arithmetic)

Type: SIMD

Format:

SRA16

31 25	24 20	19 15	14 12	11 7	6 0
SRA16 0101000	Rs2	Rs1	000	Rd	GE80B 1111111

SRA16.u

31 25	24 20	19 15	14 12	11 7	6 0
SRA16.u 0110000	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SRA16 Rd, Rs1, Rs2
SRA16.u Rd, Rs1, Rs2
```

Purpose: Do 16-bit element arithmetic right shift operations simultaneously. The shift amount is a variable from a GPR. The “.u” form performs additional rounding up operations on the shifted results.

Description: The 16-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the data elements. The shift amount is specified by the low-order 4-bits of the value in the Rs2 register. For the rounding operation of the “.u” form, a value of 1 is added to the most significant discarded bit of each 16-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```

sa = Rs2[3:0];
if (sa != 0) {
    if (!.u[] form) { // SRA16.u
        res[15:-1] = SE17(Rs1.H[x][15:sa-1]) + 1;
        Rd.H[x] = res[15:0];
    } else { // SRA16
        Rd.H[x] = SE16(Rs1.H[x][15:sa])
    }
} else {
    Rd = Rs1;
}
for RV32: x=1...0,
for RV64: x=3...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SRA16

```
unsigned long __nds__sra16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds__v_sra16(int16x2_t a, unsigned int b);
```

RV64:

```
int16x4_t __nds__v_sra16(int16x4_t a, unsigned int b);
```

- SRA16.u

```
unsigned long __nds__sra16_u(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds__v_sra16_u(int16x2_t a, unsigned int b);
```

RV64:

```
int16x4_t __nds__v_sra16_u(int16x4_t a, unsigned int b);
```

4.129. SRAI16, SRAI16.u

4.129.1. SRAI16 (SIMD 16-bit Shift Right Arithmetic Immediate)

4.129.2. SRAI16.u (SIMD 16-bit Rounding Shift Right Arithmetic Immediate)

Type: SIMD

Format:

SRAI16

31 25	24	23 20	19 15	14 12	11 7	6 0
SRAI16 0111000	0	imm4[3:0]	Rs1	000	Rd	GE80B 1111111

SRAI16.u

31 25	24	23 20	19 15	14 12	11 7	6 0
SRAI16.u 0111000	1	imm4[3:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SRAI16 Rd, Rs1, imm4u
SRAI16.u Rd, Rs1, imm4u
```

Purpose: Do 16-bit elements arithmetic right shift operations simultaneously. The shift amount is an immediate value. The “.u” form performs additional rounding up operations on the shifted results.

Description: The 16-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the 16-bit data elements. The shift amount is specified by the imm4u constant. For the rounding operation of the “.u” form, a value of 1 is added to the most significant discarded bit of each 16-bit data to calculate the final results. And the results are written to Rd.

Operations:

```

sa = imm4u[3:0];
if (sa > 0) {
    if (Rd.u form) { // SRAI16.u
        res[15:-1] = SE17(Rs1.H[x][15:sa-1]) + 1;
        Rd.H[x] = res[15:0];
    } else { // SRAI16
        Rd.H[x] = SE16(Rs1.H[x][15:sa]);
    }
} else {
    Rd = Rs1;
}
for RV32: x=1...0,
for RV64: x=3...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SRAI16

```
unsigned long __nds_sra16(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds_v_sra16(int16x2_t a, unsigned int b);
```

RV64:

```
int16x4_t __nds_v_sra16(int16x4_t a, unsigned int b);
```

- SRAI16.u

```
unsigned long __nds_sra16_u(unsigned long a, unsigned long b);
```

RV32:

```
int16x2_t __nds_v_sra16_u(int16x2_t a, unsigned int b);
```

RV64:

```
int16x4_t __nds_v_sra16_u(int16x4_t a, unsigned int b);
```

4.130. SRL8, SRL8.u

4.130.1. SRL8 (SIMD 8-bit Shift Right Logical)

4.130.2. SRL8.u (SIMD 8-bit Rounding Shift Right Logical)

Type: SIMD

Format:

SRL8

31 25	24 20	19 15	14 12	11 7	6 0
SRL8 0101101	Rs2	Rs1	000	Rd	GE80B 1111111

SRL8.u

31 25	24 20	19 15	14 12	11 7	6 0
SRL8.u 0110101	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SRL8 Rt, Ra, Rb
SRL8.u Rt, Ra, Rb
```

Purpose: Do 8-bit elements logical right shift operations simultaneously. The shift amount is a variable from a GPR. The “.u” form performs additional rounding up operations on the shifted results.

Description: The 8-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the low-order 3-bits of the value in the Rs2 register. For the rounding operation of the “.u” form, a value of 1 is added to the most significant discarded bit of each 8-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```

sa = Rs2[2:0];
if (sa > 0) {
    if (Rd.u form) { // SRL8.u
        res[8:0] = ZE9(Rs1.B[x][7:sa-1]) + 1;
        Rd.B[x] = res[8:1];
    } else { // SRL8
        Rd.B[x] = ZE8(Rs1.B[x][7:sa]);
    }
} else {
    Rd = Rs1;
}
for RV32: x=3…0,
for RV64: x=7…0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SRL8

```
unsigned long __nds__srl8(unsigned long a, unsigned int b);
```

RV32:

```
uint8x4_t __nds__v_srl8(uint8x4_t a, unsigned int b);
```

RV64:

```
uint8x8_t __nds__v_srl8(uint8x8_t a, unsigned int b);
```

- SRL8.u

```
unsigned long __nds__srl8_u(unsigned long a, unsigned int b);
```

RV32:

```
uint8x4_t __nds__v_srl8_u(uint8x4_t a, unsigned int b);
```

RV64:

```
uint8x8_t __nds__v_srl8_u(uint8x8_t a, unsigned int b);
```

4.131. SRLI8, SRLI8.u

4.131.1. SRLI8 (SIMD 8-bit Shift Right Logical Immediate)

4.131.2. SRLI8.u (SIMD 8-bit Rounding Shift Right Logical Immediate)

Type: SIMD

Format:

SRLI8

31 25	24 23	22 20	19 15	14 12	11 7	6 0
SRLI8 0111101	00	imm3u[2:0]	Rs1	000	Rd	GE80B 1111111

SRLI8.u

31 25	24 23	22 20	19 15	14 12	11 7	6 0
SRLI8.u 0111101	01	imm3u[2:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SRLI8 Rt, Ra, imm3u
SRLI8.u Rt, Ra, imm3u
```

Purpose: Do 8-bit elements logical right shift operations simultaneously. The shift amount is an immediate value. The “.u” form performs additional rounding up operations on the shifted results.

Description: The 8-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the imm3u constant. For the rounding operation of the “.u” form, a value of 1 is added to the most significant discarded bit of each 8-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```

sa = imm3u[2:0];
if (sa > 0) {
    if (!.u[] form) { // SRLI8.u
        res[8:0] = ZE9(Rs1.B[x][7:sa-1]) + 1;
        Rd.B[x] = res[8:1];
    } else { // SRLI8
        Rd.B[x] = ZE8(Rs1.B[x][7:sa]);
    }
} else {
    Rd = Rs1;
}
for RV32: x=3···0,
for RV64: x=7···0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SRLI8

```
unsigned long __nds__srl8(unsigned long a, unsigned int b);
```

RV32:

```
uint8x4_t __nds__v_srl8(uint8x4_t a, unsigned int b);
```

RV64:

```
uint8x8_t __nds__v_srl8(uint8x8_t a, unsigned int b);
```

- SRLI8.u

```
unsigned long __nds__srl8_u(unsigned long a, unsigned int b);
```

RV32:

```
uint8x4_t __nds__v_srl8_u(uint8x4_t a, unsigned int b);
```

RV64:

```
uint8x8_t __nds__v_srl8_u(uint8x8_t a, unsigned int b);
```

4.132. SRL16, SRL16.u

4.132.1. SRL16 (SIMD 16-bit Shift Right Logical)

4.132.2. SRL16.u (SIMD 16-bit Rounding Shift Right Logical)

Type: SIMD

Format:

SRL16

31 25	24 20	19 15	14 12	11 7	6 0
SRL16 0101001	Rs2	Rs1	000	Rd	GE80B 1111111

SRL16.u

31 25	24 20	19 15	14 12	11 7	6 0
SRL16.u 0110001	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

SRL16 Rt, Ra, Rb

SRL16.u Rt, Ra, Rb

Purpose: Do 16-bit elements logical right shift operations simultaneously. The shift amount is a variable from a GPR. The “.u” form performs additional rounding up operations on the shifted results.

Description: The 16-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the low-order 4-bits of the value in the Rs2 register. For the rounding operation of the “.u” form, a value of 1 is added to the most significant discarded bit of each 16-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```

sa = Rs2[3:0];
if (sa > 0) {
    if (!.u[] form) { // SRL16.u
        res[16:0] = ZE17(Rs1.H[x][15:sa-1]) + 1;
        Rd.H[x] = res[16:1];
    } else { // SRL16
        Rd.H[x] = ZE16(Rs1.H[x][15:sa]);
    }
} else {
    Rd = Rs1;
}
for RV32: x=1...0,
for RV64: x=3...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SRL16

```
unsigned long __nds__srl16(unsigned long a, unsigned int b);
```

RV32:

```
uint16x2_t __nds__v_srl16(uint16x2_t a, unsigned int b);
```

RV64:

```
uint16x4_t __nds__v_srl16(uint16x4_t a, unsigned int b);
```

- SRL16.u

```
unsigned long __nds__srl16_u(unsigned long a, unsigned int b);
```

RV32:

```
uint16x2_t __nds__v_srl16_u(uint16x2_t a, unsigned int b);
```

RV64:

```
uint16x4_t __nds__v_srl16_u(uint16x4_t a, unsigned int b);
```

4.133. SRLI16, SRLI16.u

4.133.1. SRLI16 (SIMD 16-bit Shift Right Logical Immediate)

4.133.2. SRLI16.u (SIMD 16-bit Rounding Shift Right Logical Immediate)

Type: SIMD

Format:

SRLI16

31 25	24	23 20	19 15	14 12	11 7	6 0
SRLI16 0111001	0	imm4[3:0]	Rs1	000	Rd	GE80B 1111111

SRLI16.u

31 25	24	23 20	19 15	14 12	11 7	6 0
SRLI16.u 0111001	1	imm4[3:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SRLI16 Rt, Ra, imm4u
SRLI16.u Rt, Ra, imm4u
```

Purpose: Do 16-bit elements logical right shift operations simultaneously. The shift amount is an immediate value. The “.u” form performs additional rounding up operations on the shifted results.

Description: The 16-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the imm4u constant. For the rounding operation of the “.u” form, a value of 1 is added to the most significant discarded bit of each 16-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```

sa = imm4u;
if (sa > 0) {
    if (!.u[] form) { // SRLI16.u
        res[16:0] = ZE17(Rs1.H[x][15:sa-1]) + 1;
        Rd.H[x] = res[16:1];
    } else { // SRLI16
        Rd.H[x] = ZE16(Rs1.H[x][15:sa]);
    }
} else {
    Rd = Rs1;
}
for RV32: x=1...0,
for RV64: x=3...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SRLI16

```
unsigned long __nds__srl16(unsigned long a, unsigned int b);
```

RV32:

```
uint16x2_t __nds__v_srl16(uint16x2_t a, unsigned int b);
```

RV64:

```
uint16x4_t __nds__v_srl16(uint16x4_t a, unsigned int b);
```

- SRLI16.u

```
unsigned long __nds__srl16_u(unsigned long a, unsigned int b);
```

RV32:

```
uint16x2_t __nds__v_srl16_u(uint16x2_t a, unsigned int b);
```

RV64:

```
uint16x4_t __nds__v_srl16_u(uint16x4_t a, unsigned int b);
```

4.134. STAS16 (SIMD 16-bit Straight Addition & Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
STAS16 1111010	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
STAS16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit integer element addition and 16-bit integer element subtraction in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks.

Description: This instruction adds the 16-bit integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit integer element in [31:16] of 32-bit chunks in Rs2, and writes the result to [31:16] of 32-bit chunks in Rd; at the same time, it subtracts the 16-bit integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit integer element in [15:0] of 32-bit chunks in Rs1, and writes the result to [15:0] of 32-bit chunks in Rd.

Operations:

```
Rd.W[x][31:16] = Rs1.W[x][31:16] + Rs2.W[x][31:16];  
Rd.W[x][15:0] = Rs1.W[x][15:0] - Rs2.W[x][15:0];  
for RV32, x=0  
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned operations.

Intrinsic functions:

```
unsigned long __nds__stas16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_ustas16(uint16x2_t a, uint16x2_t b);  
int16x2_t __nds__v_sstas16(int16x2_t a, int16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_ustas16(uint16x4_t a, uint16x4_t b);  
int16x4_t __nds__v_sstas16(int16x4_t a, int16x4_t b);
```

4.135. STSA16 (SIMD 16-bit Straight Subtraction & Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
STSA16 1111011	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
STSA16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit integer element subtraction and 16-bit integer element addition in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks.

Description: This instruction subtracts the 16-bit integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit integer element in [31:16] of 32-bit chunks in Rs1, and writes the result to [31:16] of 32-bit chunks in Rd; at the same time, it adds the 16-bit integer element in [15:0] of 32-bit chunks in Rs2 with the 16-bit integer element in [15:0] of 32-bit chunks in Rs1, and writes the result to [15:0] of 32-bit chunks in Rd.

Operations:

```
Rd.W[x][31:16] = Rs1.W[x][31:16] - Rs2.W[x][31:16];  
Rd.W[x][15:0] = Rs1.W[x][15:0] + Rs2.W[x][15:0];  
for RV32, x=0  
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned operations.

Intrinsic functions:

```
unsigned long __nds__stsa16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_ustsa16(uint16x2_t a, uint16x2_t b);  
int16x2_t __nds__v_sstsa16(int16x2_t a, int16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_ustsa16(uint16x4_t a, uint16x4_t b);  
int16x4_t __nds__v_sstsa16(int16x4_t a, int16x4_t b);
```

4.136. SUB8 (SIMD 8-bit Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SUB8 0100101	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SUB8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit integer element subtractions simultaneously.

Description: This instruction subtracts the 8-bit integer elements in Rs2 from the 8-bit integer elements in Rs1, and then writes the result to Rd.

Operations:

```
Rd.B[x] = Rs1.B[x] - Rs2.B[x];  
for RV32: x=3...0,  
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned subtraction.

Intrinsic functions:

```
unsigned long __nds__sub8(unsigned long a, unsigned long b);
```

```
RV32:  
uint8x4_t __nds__v_usub8(uint8x4_t a, uint8x4_t b);  
int8x4_t __nds__v_ssub8(int8x4_t a, int8x4_t b);
```

```
RV64:  
uint8x8_t __nds__v_usub8(uint8x8_t a, uint8x8_t b);  
int8x8_t __nds__v_ssub8(int8x8_t a, int8x8_t b);
```

4.137. SUB16 (SIMD 16-bit Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SUB16 0100001	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SUB16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit integer element subtractions simultaneously.

Description: This instruction subtracts the 16-bit integer elements in Rs2 from the 16-bit integer elements in Rs1, and then writes the result to Rd.

Operations:

```
Rd.H[x] = Rs1.H[x] - Rs2.H[x];  
for RV32: x=1...0,  
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned subtraction.

Intrinsic functions:

```
unsigned long __nds__sub16(unsigned long a, unsigned long b);
```

```
RV32:  
uint16x2_t __nds__v_usub16(uint16x2_t a, uint16x2_t b);  
int16x2_t __nds__v_ssub16(int16x2_t a, int16x2_t b);
```

```
RV64:  
uint16x4_t __nds__v_usub16(uint16x4_t a, uint16x4_t b);  
int16x4_t __nds__v_ssub16(int16x4_t a, int16x4_t b);
```

4.138. SUB64 (64-bit Subtraction)

Type: DSP (64-bit Profile)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SUB64 1100001	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
SUB64 Rd, Rs1, Rs2
```

Purpose: Perform a 64-bit signed or unsigned integer subtraction.

RV32 Description: This instruction subtracts the 64-bit integer of an even/odd pair of registers specified by Rs2(4,1) from the 64-bit integer of an even/odd pair of registers specified by Rs1(4,1), and then writes the 64-bit result to an even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the operand and the even “ $2d$ ” register of the pair contains the low 32-bit of the operand.

RV64 Description: This instruction subtracts the 64-bit integer of Rs2 from the 64-bit integer of Rs1, and then writes the 64-bit result to Rd.

Operations:

- RV32:

```
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
a_L = CONCAT(Rs1(4,1),1'b0); a_H = CONCAT(Rs1(4,1),1'b1);
b_L = CONCAT(Rs2(4,1),1'b0); b_H = CONCAT(Rs2(4,1),1'b1);
R[t_H].R[t_L] = R[a_H].R[a_L] - R[b_H].R[b_L];
```

- RV64:

```
Rd = Rs1 - Rs2;
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned subtraction.

Intrinsic functions:

```
long long __nds__ssub64(long long a, long long b);
```

```
unsigned long long __nds__usub64(unsigned long long a, +
unsigned long long b);
```

4.139. SUNPKD810, SUNPKD820, SUNPKD830, SUNPKD831, SUNPKD832

4.139.1. SUNPKD810 (Signed Unpacking Bytes 1 & 0)

4.139.2. SUNPKD820 (Signed Unpacking Bytes 2 & 0)

4.139.3. SUNPKD830 (Signed Unpacking Bytes 3 & 0)

4.139.4. SUNPKD831 (Signed Unpacking Bytes 3 & 1)

4.139.5. SUNPKD832 (Signed Unpacking Bytes 3 & 2)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP 1010110	SUNPKD8 <u>xy</u> code[4:0]	Rs1	000	Rd	GE80B 1111111

<u>xy</u>	<u>code[4:0]</u>
10	01000
20	01001
30	01010
31	01011
32	10011

Syntax:

```
SUNPKD8xy Rd, Rs1
xy = {10, 20, 30, 31, 32}
```

Purpose: Unpack byte **x* and byte *y** of 32-bit chunks in a register into two 16-bit signed halfwords of 32-bit chunks in a register.

Description:

For the “SUNPKD8(x(*y)” instruction, it unpacks byte **x* and byte *y** of 32-bit chunks in Rs1 into two 16-bit signed halfwords and writes the results to the top part and the bottom part of 32-bit chunks in Rd.

Operations:

```
Rd.W[m].H[1] = SE16(Rs1.W[m].B[x])
Rd.W[m].H[0] = SE16(Rs1.W[m].B[y])
// SUNPKD810, x=1,y=0
// SUNPKD820, x=2,y=0
// SUNPKD830, x=3,y=0
// SUNPKD831, x=3,y=1
// SUNPKD832, x=3,y=2
for RV32: m=0,
for RV64: m=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SUNPK810

```
unsigned long __nds_sunpkd810(unsigned long a);
```

```
RV32:
int16x2_t __nds_v_sunpkd810(int8x4_t a);
RV64:
int16x4_t __nds_v_sunpkd810(int8x8_t a);
```

- SUNPK820

```
unsigned long __nds_sunpkd820(unsigned long a);
```

```
RV32:
int16x2_t __nds_v_sunpkd820(int8x4_t a);
RV64:
int16x4_t __nds_v_sunpkd820(int8x8_t a);
```

- SUNPK830

```
unsigned long __nds_sunpkd830(unsigned long a);
```

```
RV32:  
    int16x2_t __nds__v_sunpkd830(int8x4_t a);  
RV64:  
    int16x4_t __nds__v_sunpkd830(int8x8_t a);
```

- SUNPK831

```
unsigned long __nds__sunpkd831(unsigned long a);
```

```
RV32:  
    int16x2_t __nds__v_sunpkd831(int8x4_t a);  
RV64:  
    int16x4_t __nds__v_sunpkd831(int8x8_t a);
```

- SUNPK832

```
unsigned long __nds__sunpkd832(unsigned long a);
```

```
RV32:  
    int16x2_t __nds__v_sunpkd832(int8x4_t a);  
RV64:  
    int16x4_t __nds__v_sunpkd832(int8x8_t a);
```

4.140. SWAP8 (Swap Byte within Halfword)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP 1010110	SWAP8 11000	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SWAP8 Rd, Rs1
```

Purpose: Swap the bytes within each halfword of a register.

Description: This instruction swaps the bytes within each halfword of Rs1 and writes the result to Rd.

Operations:

```
Rd.H[x] = CONCAT(Rs1.H[x][7:0],Rs1.H[x][15:8]);  
for RV32: x=1...0,  
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_swap8(unsigned long a);
```

```
RV32:  
    uint8x4_t __nds_v_swap8(uint8x4_t a);  
RV64:  
    uint8x8_t __nds_v_swap8(uint8x8_t a);
```

4.141. SWAP16 (Swap Halfword within Word)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP 1010110	SWAP16 11001	Rs1	000	Rd	GE80B 1111111

Syntax:

```
SWAP16 Rd, Rs1
```

Purpose: Swap the 16-bit halfwords within each word of a register.

Description: This instruction swaps the 16-bit halfwords within each word of Rs1 and writes the result to Rd.

Operations:

```
Rd.W[x] = CONCAT(Rs1.W[x][15:0],Rs1.H[x][31:16]);  
for RV32: x=0,  
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_swap16(unsigned long a);
```

```
RV32:  
  uint16x2_t __nds_v_swap16(uint16x2_t a);  
RV64:  
  uint16x4_t __nds_v_swap16(uint16x4_t a);
```

4.142. UCLIP8 (SIMD 8-bit Unsigned Clip Value)

Type: SIMD

Format:

31 25	24 23	22 20	19 15	14 12	11 7	6 0
UCLIP8 1000110	10	imm3u[2:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UCLIP8 Rt, Ra, imm3u
```

Purpose: Limit the 8-bit signed elements of a register into an unsigned range simultaneously.

Description: This instruction limits the 8-bit signed elements stored in Rs1 into an unsigned integer range between $2^{\text{imm3u}}-1$ and 0, and writes the limited results to Rd. For example, if imm3u is 3, the 8-bit input values should be saturated between 7 and 0. If saturation is performed, set OV bit to 1.

Operations:

```
src = Rs1.H[x];
if (src > (2^imm3u)-1) {
    src = (2^imm3u)-1;
    OV = 1;
} else if (src < 0) {
    src = 0;
    OV = 1;
}
Rd.H[x] = src;
for RV32: x=3...0,
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__uclip8(unsigned long a, unsigned int b);
```

RV32:

```
uint8x4_t __ndsv_uclip8(uint8x4_t a, unsigned int b);
```

RV64:

```
uint8x8_t __ndsv_uclip8(uint8x8_t a, unsigned int b);
```

4.143. UCLIP16 (SIMD 16-bit Unsigned Clip Value)

Type: SIMD

Format:

31 25	24	23 20	19 15	14 12	11 7	6 0
UCLIP16 1000010	1	imm4u[3:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UCLIP16 Rt, Ra, imm4u
```

Purpose: Limit the 16-bit signed elements of a register into an unsigned range simultaneously.

Description: This instruction limits the 16-bit signed elements stored in Rs1 into an unsigned integer range between $2^{\text{imm4u}} - 1$ and 0, and writes the limited results to Rd. For example, if imm4u is 3, the 16-bit input values should be saturated between 7 and 0. If saturation is performed, set OV bit to 1.

Operations:

```
src = Rs1.H[x];
if (src > (2^imm4u)-1) {
    src = (2^imm4u)-1;
    OV = 1;
} else if (src < 0) {
    src = 0;
    OV = 1;
}
Rd.H[x] = src;
for RV32: x=1...0,
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__uclip16(unsigned long a, unsigned int b);
```

RV32:

```
uint16x2_t __nds__v_uclip16(uint16x2_t a, unsigned int b);
```

RV64:

```
uint16x4_t __nds__v_uclip16(uint16x4_t a, unsigned int b);
```

4.144. UCLIP32 (SIMD 32-bit Unsigned Clip Value)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UCLIP32 1111010	imm5u[4:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UCLIP32 Rd, Rs1, imm5u[4:0]
```

Purpose: Limit the 32-bit signed integer elements of a register into an unsigned range simultaneously.

Description: This instruction limits the 32-bit signed integer elements stored in Rs1 into an unsigned integer range between $2^{\text{imm5u}}-1$ and 0, and writes the limited results to Rd. For example, if imm5u is 3, the 32-bit input values should be saturated between 7 and 0. If saturation is performed, set OV bit to 1.

Operations:

```
src = Rs1.W[x];
if (src > (2^imm5u)-1) {
    src = (2^imm5u)-1;
    OV = 1;
} else if (src < 0) {
    src = 0;
    OV = 1;
}
Rd.W[x] = src
for RV32: x=0,
for RV64: x=1…0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__uclip32(unsigned long a, unsigned int b);
```

RV64:

```
uint32x2_t __nds__v_uclip32(uint32x2_t a, unsigned int b);
```

4.145. UCMPE8 (SIMD 8-bit Unsigned Compare Less Than & Equal)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UCMPE8 0011111	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UCMPE8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit unsigned integer elements less than & equal comparisons simultaneously.

Description: This instruction compares the 8-bit unsigned integer elements in Rs1 with the 8-bit unsigned integer elements in Rs2 to see if the one in Rs1 is less than or equal to the one in Rs2. If it is true, the result is 0xFF; otherwise, the result is 0x0. The four comparison results are written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] <= Rs2.B[x])? 0xff : 0x0;  
for RV32: x=3...0,  
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ucmple8(unsigned long a, unsigned long b);
```

RV32:

```
uint8x4_t __nds__v_ucmple8(uint8x4_t a, uint8x4_t b);
```

RV64:

```
uint8x8_t __nds__v_ucmple8(uint8x8_t a, uint8x8_t b);
```

4.146. UCMPLE16 (SIMD 16-bit Unsigned Compare Less Than & Equal)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UCMPL16 0011110	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UCMPL16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer elements less than & equal comparisons simultaneously.

Description: This instruction compares the 16-bit unsigned integer elements in Rs1 with the 16-bit unsigned integer elements in Rs2 to see if the one in Rs1 is less than or equal to the one in Rs2. If it is true, the result is 0xFFFF; otherwise, the result is 0x0. The element comparison results are written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] <= Rs2.H[x])? 0xffff : 0x0;  
for RV32: x=1...0,  
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ucmple16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_ucmple16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_ucmple16(uint16x4_t a, uint16x4_t b);
```

4.147. UCMPLT8 (SIMD 8-bit Unsigned Compare Less Than)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UCMPLT8 0010111	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UCMPLT8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit unsigned integer elements less than comparisons simultaneously.

Description: This instruction compares the 8-bit unsigned integer elements in Rs1 with the 8-bit unsigned integer elements in Rs2 to see if the one in Rs1 is less than the one in Rs2. If it is true, the result is 0xFF; otherwise, the result is 0x0. The element comparison results are written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] < Rs2.B[x])? 0xff : 0x0;  
for RV32: x=3...0,  
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ucmplt8(unsigned long a, unsigned long b);
```

RV32:

```
uint8x4_t __nds__v_ucmplt8(uint8x4_t a, uint8x4_t b);
```

RV64:

```
uint8x8_t __nds__v_ucmplt8(uint8x8_t a, uint8x8_t b);
```

4.148. UCMPLT16 (SIMD 16-bit Unsigned Compare Less Than)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UCMPLT16 0010110	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UCMPLT16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer elements less than comparisons simultaneously.

Description: This instruction compares the 16-bit unsigned integer elements in Rs1 with the 16-bit unsigned integer elements in Rs2 to see if the one in Rs1 is less than the one in Rs2. If it is true, the result is 0xFFFF; otherwise, the result is 0x0. The element comparison results are written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] < Rs2.H[x])? 0xffff : 0x0;  
for RV32: x=1…0,  
for RV64: x=3…0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ucmplt16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_ucmplt16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_ucmplt16(uint16x4_t a, uint16x4_t b);
```

4.149. UKADD8 (SIMD 8-bit Unsigned Saturating Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKADD8 0011100	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UKADD8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit unsigned integer element saturating additions simultaneously.

Description: This instruction adds the 8-bit unsigned integer elements in Rs1 with the 8-bit unsigned integer elements in Rs2. If any of the results are beyond the 8-bit unsigned number range ($0 \leq \text{RES} \leq 2^8-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.B[x] + Rs2.B[x];
if (res[x] > (2^8)-1) {
    res[x] = (2^8)-1;
    OV = 1;
}
Rd.B[x] = res[x];
for RV32: x=3...0,
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_ukadd8(unsigned long a, unsigned long b);
```

RV32:

```
uint8x4_t __nds__v_ukadd8(uint8x4_t a, uint8x4_t b);
```

RV64:

```
uint8x8_t __nds__v_ukadd8(uint8x8_t a, uint8x8_t b);
```

4.150. UKADD16 (SIMD 16-bit Unsigned Saturating Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKADD16 0011000	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UKADD16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer element saturating additions simultaneously.

Description: This instruction adds the 16-bit unsigned integer elements in Rs1 with the 16-bit unsigned integer elements in Rs2. If any of the results are beyond the 16-bit unsigned number range ($0 \leq \text{RES} \leq 2^{16}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.H[x] + Rs2.H[x];
if (res[x] > (2^16)-1) {
    res[x] = (2^16)-1;
    OV = 1;
}
Rd.H[x] = res[x];
for RV32: x=1...0,
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_ukadd16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_ukadd16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_ukadd16(uint16x4_t a, uint16x4_t b);
```

4.151. UKADD64 (64-bit Unsigned Saturating Addition)

Type: DSP (64-bit Profile)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKADD64 1011000	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
UKADD64 Rd, Rs1, Rs2
```

Purpose: Add two 64-bit unsigned integers. The result is saturated to the U64 range.

RV32 Description: This instruction adds the 64-bit unsigned integer of an even/odd pair of registers specified by Rs1(4,1) with the 64-bit unsigned integer of an even/odd pair of registers specified by Rs2(4,1). If the 64-bit result is beyond the U64 number range ($0 \leq U64 \leq 2^{64}-1$), it is saturated to the range and the OV bit is set to 1. The saturated result is written to an even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction adds the 64-bit unsigned integer in Rs1 with the 64-bit unsigned integer in Rs2. If the 64-bit result is beyond the U64 number range ($0 \leq U64 \leq 2^{64}-1$), it is saturated to the range and the OV bit is set to 1. The saturated result is written to Rd.

Operations:

- RV32:

```
t_L = CONCAT(Rt(4,1),1\b0); t_H = CONCAT(Rt(4,1),1\b1);
a_L = CONCAT(Ra(4,1),1\b0); a_H = CONCAT(Ra(4,1),1\b1);
b_L = CONCAT(Rb(4,1),1\b0); b_H = CONCAT(Rb(4,1),1\b1);
result = R[a_H].R[a_L] + R[b_H].R[b_L];
if (result > (2^64)-1) {
    result = (2^64)-1; OV = 1;
}
R[t_H].R[t_L] = result;
```

- RV64:

```
result = Rs1 + Rs2;
if (result > (2^64)-1) {
    result = (2^64)-1; OV = 1;
}
Rd = result;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long long __nds__ukadd64(unsigned long long a, +
unsigned long long b);
```

4.152. UKADDH (Unsigned Addition with U16 Saturation)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKADDH 0001010	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
UKADDH Rd, Rs1, Rs2
```

Purpose: Add the unsigned lower 32-bit content of two registers with U16 saturation.

Description: The unsigned lower 32-bit content of Rs1 is added with the unsigned lower 32-bit content of Rs2. And the result is saturated to the 16-bit unsigned integer range of $[0, 2^{16}-1]$ and then sign-extended and written to Rd. If saturation happens, this instruction sets the OV flag.

Operations:

```
tmp = Rs1.W[0] + Rs2.W[0];
if (tmp > (2^16)-1) {
    tmp = (2^16)-1;
    OV = 1;
}
Rd = SE(tmp[15:0]);
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_ukaddh(unsigned int a, unsigned int b);
```

4.153. UKADDW (Unsigned Addition with U32 Saturation)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKADDW 0001000	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
UKADDW Rd, Rs1, Rs2
```

Purpose: Add the unsigned lower 32-bit content of two registers with U32 saturation.

Description: The unsigned lower 32-bit content of Rs1 is added with the unsigned lower 32-bit content of Rs2. And the result is saturated to the 32-bit unsigned integer range of $[0, 2^{32}-1]$ and then sign-extended and written to Rd. If saturation happens, this instruction sets the OV flag.

Operations:

```
tmp = Rs1.W[0] + Rs2.W[0];
if (tmp > (2^32)-1) {
    tmp[31:0] = (2^32)-1;
    OV = 1;
}
Rd = tmp[31:0]; // RV32
Rd = SE(tmp[31:0]); // RV64
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ukaddw(unsigned int a, unsigned int b);
```

4.154. UKCRAS16 (SIMD 16-bit Unsigned Saturating Cross Addition & Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKCRAS16 0011010	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UKCRAS16 Rd, Rs1, Rs2
```

Purpose: Do one 16-bit unsigned integer element saturating addition and one 16-bit unsigned integer element saturating subtraction in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks.

Description: This instruction adds the 16-bit unsigned integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit unsigned integer element in [15:0] of 32-bit chunks in Rs2; at the same time, it subtracts the 16-bit unsigned integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit unsigned integer element in [15:0] of 32-bit chunks in Rs1. If any of the results are beyond the 16-bit unsigned number range ($0 \leq \text{RES} \leq 2^{16}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for addition and [15:0] of 32-bit chunks in Rd for subtraction.

Operations:

```
res1 = Rs1.W[x][31:16] + Rs2.W[x][15:0];
res2 = Rs1.W[x][15:0] - Rs2.W[x][31:16];
if (res1 > (2^16)-1) {
    res1 = (2^16)-1;
    OV = 1;
}
if (res2 < 0) {
    res2 = 0;
    OV = 1;
}
Rd.W[x][31:16] = res1;
Rd.W[x][15:0] = res2;
for RV32, x=0
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ukcras16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_ukcras16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_ukcras16(uint16x4_t a, uint16x4_t b);
```

4.155. UKCRSA16 (SIMD 16-bit Unsigned Saturating Cross Subtraction & Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKRSA16 0011011	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UKRSA16 Rd, Rs1, Rs2
```

Purpose: Do one 16-bit unsigned integer element saturating subtraction and one 16-bit unsigned integer element saturating addition in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks.

Description: This instruction subtracts the 16-bit unsigned integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit unsigned integer element in [31:16] of 32-bit chunks in Rs1; at the same time, it adds the 16-bit unsigned integer element in [31:16] of 32-bit chunks in Rs2 with the 16-bit unsigned integer element in [15:0] of 32-bit chunks in Rs1. If any of the results are beyond the 16-bit unsigned number range ($0 \leq \text{RES} \leq 2^{16}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for subtraction and [15:0] of 32-bit chunks in Rd for addition.

Operations:

```
res1 = Rs1.W[x][31:16] - Rs2.W[x][15:0];
res2 = Rs1.W[x][15:0] + Rs2.W[x][31:16];
if (res1 < 0) {
    res1 = 0;
    OV = 1;
} else if (res2 > (2^16)-1) {
    res2 = (2^16)-1;
    OV = 1;
}
Rd.W[x][31:16] = res1;
Rd.W[x][15:0] = res2;
for RV32, x=0
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ukcrsa16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_ukcrsa16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_ukcrsa16(uint16x4_t a, uint16x4_t b);
```

4.156. UKMAR64 (Unsigned Multiply and Saturating Add to 64-Bit Data)

Type: DSP (64-bit Profile)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKMAR64 1011010	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
UKMAR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit unsigned elements in two registers and add the 64-bit multiplication results to the 64-bit unsigned data of a pair of registers (RV32) or a register (RV64). The result is saturated to the U64 range and written back to the pair of registers (RV32) or the register (RV64).

RV32 Description: This instruction multiplies the 32-bit unsigned data of Rs1 with that of Rs2. It adds the 64-bit multiplication result to the 64-bit unsigned data of an even/odd pair of registers specified by Rd(4,1) with unlimited precision. If the 64-bit addition result is beyond the U64 number range ($0 \leq U64 \leq 2^{64}-1$), it is saturated to the range and the OV bit is set to 1. The saturated result is written back to the even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction multiplies the 32-bit unsigned elements of Rs1 with that of Rs2. It adds the 64-bit multiplication results to the 64-bit unsigned data in Rd with unlimited precision. If the 64-bit addition result is beyond the U64 number range ($0 \leq U64 \leq 2^{64}-1$), it is saturated to the range and the OV bit is set to 1. The saturated result is written back to Rd.

Operations:

- RV32:

```
t_L = CONCAT(Rd(4,1),1\b0); t_H = CONCAT(Rd(4,1),1\b1);
result = R[t_H].R[t_L] + (Rs1 * Rs2);
if (result > (2^64)-1) {
    result = (2^64)-1; OV = 1;
}
R[t_H].R[t_L] = result;
```

- RV64:

```
// result has unlimited precision
result = Rd + (Rs1.W[0] u* Rs2.W[0]) + (Rs1.W[1] u* Rs2.W[1]);
if (result > (2^64)-1) {
    result = (2^64)-1; OV = 1;
}
Rd = result;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long long __nds__ukmar64(unsigned long long t, +
unsigned long a, unsigned long b);
```

RV64:

```
unsigned long long __nds__v_ukmar64(unsigned long long t, +
uint32x2_t a, uint32x2_t b);
```

4.157. UKMSR64 (Unsigned Multiply and Saturating Subtract from 64-Bit Data)

Type: DSP (64-bit Profile)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKMSR64 1011011	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
UKMSR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit unsigned elements in two registers and subtract the 64-bit multiplication results from the 64-bit unsigned data of a pair of registers (RV32) or a register (RV64). The result is saturated to the U64 range and written back to the pair of registers (RV32) or a register (RV64).

RV32 Description: This instruction multiplies the 32-bit unsigned data of Rs1 with that of Rs2. It subtracts the 64-bit multiplication result from the 64-bit unsigned data of an even/odd pair of registers specified by Rd(4,1) with unlimited precision. If the 64-bit subtraction result is beyond the U64 number range ($0 \leq U64 \leq 2^{64}-1$), it is saturated to the range and the OV bit is set to 1. The saturated result is written back to the even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction multiplies the 32-bit unsigned elements of Rs1 with that of Rs2. It subtracts the 64-bit multiplication results from the 64-bit unsigned data of Rd with unlimited precision. If the 64-bit subtraction result is beyond the U64 number range ($0 \leq U64 \leq 2^{64}-1$), it is saturated to the range and the OV bit is set to 1. The saturated result is written back to Rd.

Operations:

- RV32:

```
t_L = CONCAT(Rd(4,1),1\b0); t_H = CONCAT(Rd(4,1),1\b1);
result = R[t_H].R[t_L] - (Rs1 u* Rs2);
if (result < 0) {
    result = 0; OV = 1;
}
R[t_H].R[t_L] = result;
```

- RV64:

```
// result has unlimited precision
result = Rd - (Rs1.W[0] u* Rs2.W[0]) - (Rs1.W[1] u* Rs2.W[1]);
if (result < 0) {
    result = 0; OV = 1;
}
Rd = result;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long long __nds__ukmsr64(unsigned long long t, +
unsigned long a, unsigned long b);
```

RV64:

```
unsigned long long __nds__v_ukmsr64(unsigned long long t, +
uint32x2_t a, uint32x2_t b);
```

4.158. UKSTAS16 (SIMD 16-bit Unsigned Saturating Straight Addition & Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKSTAS16 1110010	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
UKSTAS16 Rd, Rs1, Rs2
```

Purpose: Do one 16-bit unsigned integer element saturating addition and one 16-bit unsigned integer element saturating subtraction in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks.

Description: This instruction adds the 16-bit unsigned integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit unsigned integer element in [31:16] of 32-bit chunks in Rs2; at the same time, it subtracts the 16-bit unsigned integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit unsigned integer element in [15:0] of 32-bit chunks in Rs1. If any of the results are beyond the 16-bit unsigned number range ($0 \leq \text{RES} \leq 2^{16}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for addition and [15:0] of 32-bit chunks in Rd for subtraction.

Operations:

```
res1 = Rs1.W[x][31:16] + Rs2.W[x][31:16];
res2 = Rs1.W[x][15:0] - Rs2.W[x][15:0];
if (res1 > (2^16)-1) {
    res1 = (2^16)-1;
    OV = 1;
}
if (res2 < 0) {
    res2 = 0;
    OV = 1;
}
Rd.W[x][31:16] = res1;
Rd.W[x][15:0] = res2;
for RV32, x=0
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ukstas16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_ukstas16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_ukstas16(uint16x4_t a, uint16x4_t b);
```

4.159. UKSTSA16 (SIMD 16-bit Unsigned Saturating Straight Subtraction & Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKSTSA16 1110011	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
UKSTSA16 Rd, Rs1, Rs2
```

Purpose: Do one 16-bit unsigned integer element saturating subtraction and one 16-bit unsigned integer element saturating addition in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks.

Description: This instruction subtracts the 16-bit unsigned integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit unsigned integer element in [31:16] of 32-bit chunks in Rs1; at the same time, it adds the 16-bit unsigned integer element in [15:0] of 32-bit chunks in Rs2 with the 16-bit unsigned integer element in [15:0] of 32-bit chunks in Rs1. If any of the results are beyond the 16-bit unsigned number range ($0 \leq \text{RES} \leq 2^{16}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for subtraction and [15:0] of 32-bit chunks in Rd for addition.

Operations:

```
res1 = Rs1.W[x][31:16] - Rs2.W[x][31:16];
res2 = Rs1.W[x][15:0] + Rs2.W[x][15:0];
if (res1 < 0) {
    res1 = 0;
    OV = 1;
} else if (res2 > (2^16)-1) {
    res2 = (2^16)-1;
    OV = 1;
}
Rd.W[x][31:16] = res1;
Rd.W[x][15:0] = res2;
for RV32, x=0
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ukstsa16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_ukstsa16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_ukstsa16(uint16x4_t a, uint16x4_t b);
```

4.160. UKSUB8 (SIMD 8-bit Unsigned Saturating Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKSUB8 0011101	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UKSUB8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit unsigned integer elements saturating subtractions simultaneously.

Description: This instruction subtracts the 8-bit unsigned integer elements in Rs2 from the 8-bit unsigned integer elements in Rs1. If any of the results are beyond the 8-bit unsigned number range ($0 \leq \text{RES} \leq 2^8 - 1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.B[x] - Rs2.B[x];
if (res[x] < 0) {
    res[x] = 0;
    OV = 1;
}
Rd.B[x] = res[x];
for RV32: x=3...0,
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_uksub8(unsigned long a, unsigned long b);
```

RV32:

```
uint8x4_t __nds__v_uksub8(uint8x4_t a, uint8x4_t b);
```

RV64:

```
uint8x8_t __nds__v_uksub8(uint8x8_t a, uint8x8_t b);
```

4.161. UKSUB16 (SIMD 16-bit Unsigned Saturating Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKSUB16 0011001	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UKSUB16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer elements saturating subtractions simultaneously.

Description: This instruction subtracts the 16-bit unsigned integer elements in Rs2 from the 16-bit unsigned integer elements in Rs1. If any of the results are beyond the 16-bit unsigned number range ($0 \leq \text{RES} \leq 2^{16}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.H[x] - Rs2.H[x];
if (res[x] < 0) {
    res[x] = 0;
    OV = 1;
}
Rd.H[x] = res[x];
for RV32: x=1...0,
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_uksub16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_uksub16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_uksub16(uint16x4_t a, uint16x4_t b);
```

4.162. UKSUB64 (64-bit Unsigned Saturating Subtraction)

Type: DSP (64-bit Profile)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKSUB64 1011001	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
UKSUB64 Rd, Rs1, Rs2
```

Purpose: Perform a 64-bit signed integer subtraction. The result is saturated to the U64 range.

RV32 Description: This instruction subtracts the 64-bit unsigned integer of an even/odd pair of registers specified by Rs2(4,1) from the 64-bit unsigned integer of an even/odd pair of registers specified by Rs1(4,1). If the 64-bit result is beyond the U64 number range ($0 \leq U64 \leq 2^{64}-1$), it is saturated to the range and the OV bit is set to 1. The saturated result is then written to an even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the operand and the even “ $2d$ ” register of the pair contains the low 32-bit of the operand.

RV64 Description: This instruction subtracts the 64-bit unsigned integer of Rs2 from the 64-bit unsigned integer of an even/odd pair of Rs1. If the 64-bit result is beyond the U64 number range ($0 \leq U64 \leq 2^{64}-1$), it is saturated to the range and the OV bit is set to 1. The saturated result is then written to Rd.

Operations:

- RV32:

```
t_L = CONCAT(Rd(4,1),1\b0); t_H = CONCAT(Rd(4,1),1\b1);
a_L = CONCAT(Rs1(4,1),1\b0); a_H = CONCAT(Rs1(4,1),1\b1);
b_L = CONCAT(Rs2(4,1),1\b0); b_H = CONCAT(Rs2(4,1),1\b1);
result = R[a_H].R[a_L] - R[b_H].R[b_L];
if (result < 0) {
    result = 0; OV = 1;
}
R[t_H].R[t_L] = result;
```

- RV64

```
result = Rs1 - Rs2;
if (result < 0) {
    result = 0; OV = 1;
}
Rd = result;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long long __nds__uksub64(unsigned long long a, +
unsigned long long b);
```

4.163. UKSUBH (Unsigned Subtraction with U16 Saturation)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKSUBH 0001011	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
UKSUBH Rd, Rs1, Rs2
```

Purpose: Subtract the unsigned lower 32-bit content of two registers with U16 saturation.

Description: The unsigned lower 32-bit content of Rs2 is subtracted from the unsigned lower 32-bit content of Rs1. And the result is saturated to the 16-bit unsigned integer range of [0, 2^{16} -1] and then sign-extended and written to Rd. If saturation happens, this instruction sets the OV flag.

Operations:

```
tmp = Rs1.W[0] - Rs2.W[0];
if (tmp > (2^16)-1) {
    tmp = (2^16)-1;
    OV = 1;
}
else if (tmp < 0) {
    tmp = 0;
    OV = 1;
}
Rd = SE(tmp[15:0]);
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__uksubh(unsigned int a, unsigned int b);
```

4.164. UKSUBW (Unsigned Subtraction with U32 Saturation)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKSUBW 0001001	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
UKSUBW Rd, Rs1, Rs2
```

Purpose: Subtract the unsigned lower 32-bit content of two registers with unsigned 32-bit saturation.

Description: The unsigned lower 32-bit content of Rs2 is subtracted from the unsigned lower 32-bit content of Rs1. And the result is saturated to the 32-bit unsigned integer range of $[0, 2^{32}-1]$ and then sign-extended and written to Rd. If saturation happens, this instruction sets the OV flag.

Operations:

```
tmp = Rs1.W[0] - Rs2.W[0];
if (tmp < 0) {
    tmp[31:0] = 0;
    OV = 1;
}
Rd = tmp[31:0]; // RV32
Rd = SE(tmp[31:0]); // RV64
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__uksubw(unsigned int a, unsigned int b);
```

4.165. UMAR64 (Unsigned Multiply and Add to 64-Bit Data)

Type: DSP (64-bit Profile)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UMAR64 1010010	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
UMAR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit unsigned elements in two registers and add the 64-bit multiplication results to the 64-bit unsigned data of a pair of registers (RV32) or a register (RV64). The result is written back to the pair of registers (RV32) or a register (RV64).

RV32 Description: This instruction multiplies the 32-bit unsigned data of Rs1 with that of Rs2. It adds the 64-bit multiplication result to the 64-bit unsigned data of an even/odd pair of registers specified by Rd(4,1). The addition result is written back to the even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction multiplies the 32-bit unsigned elements of Rs1 with that of Rs2. It adds the 64-bit multiplication results to the 64-bit unsigned data of Rd. The addition result is written back to Rd.

Operations:

- RV32:

```
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H].R[t_L] = R[t_H].R[t_L] + (Rs1 * Rs2);
```

- RV64:

```
Rd = Rd + (Rs1.W[0] u* Rs2.W[0]) + (Rs1.W[1] u* Rs2.W[1]);
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long long __nds__umar64(unsigned long long t, +  
unsigned long a, unsigned long b);
```

RV64:

```
unsigned long long __nds__v_umar64(unsigned long long t, +  
uint32x2_t a, uint32x2_t b);
```

4.166. UMAQA (Unsigned Multiply Four Bytes with 32-bit Adds)

Type: DSP

Format:

UMAQA

31 25	24 20	19 15	14 12	11 7	6 0
UMAQA 1100110	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UMAQA Rd, Rs1, Rs2
```

Purpose: Do four unsigned 8-bit multiplications from 32-bit chunks of two registers; and then adds the four 16-bit results and the content of corresponding 32-bit chunks of a third register together.

Description:

This instruction multiplies the four unsigned 8-bit elements of 32-bit chunks of Rs1 with the four unsigned 8-bit elements of 32-bit chunks of Rs2 and then adds the four results together with the unsigned content of the corresponding 32-bit chunks of Rd. The final results are written back to the corresponding 32-bit chunks in Rd.

Operations:

```
res[x] = Rd.W[x] + (Rs1.W[x].B[3] u* Rs2.W[x].B[3]) +
         (Rs1.W[x].B[2] u* Rs2.W[x].B[2]) + (Rs1.W[x].B[1] u* Rs2.W[x].B[1]) +
         (Rs1.W[x].B[0] u* Rs2.W[x].B[0]);
Rd.W[x] = res[x];
for RV32: x=0,
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__umaqa(unsigned long t, unsigned long a, unsigned long b);
```

RV32:

```
unsigned int __nds__v_umaqa(unsigned int t, uint8x4_t a, uint8x4_t b);
```

RV64:

```
uint32x2_t __nds__v_umaqa(unsigned long t, uint8x8_t a, uint8x8_t b);
```

4.167. UMAX8 (SIMD 8-bit Unsigned Maximum)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UMAX8 1001101	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UMAX8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit unsigned integer elements finding maximum operations simultaneously.

Description: This instruction compares the 8-bit unsigned integer elements in Rs1 with the four 8-bit unsigned integer elements in Rs2 and selects the numbers that is greater than the other one. The two selected results are written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] >u Rs2.B[x])? Rs1.B[x] : Rs2.B[x];  
for RV32: x=3...0,  
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__umax8(unsigned long a, unsigned long b);
```

```
RV32:  
    uint8x4_t __nds__v_umax8(uint8x4_t a, uint8x4_t b);
```

```
RV64:  
    uint8x8_t __nds__v_umax8(uint8x8_t a, uint8x8_t b);
```

4.168. UMAX16 (SIMD 16-bit Unsigned Maximum)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UMAX16 1001001	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UMAX16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer elements finding maximum operations simultaneously.

Description: This instruction compares the 16-bit unsigned integer elements in Rs1 with the 16-bit unsigned integer elements in Rs2 and selects the numbers that is greater than the other one. The selected results are written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] >u Rs2.H[x])? Rs1.H[x] : Rs2.H[x];  
for RV32: x=1…0,  
for RV64: x=3…0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_umax16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds_v_umax16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds_v_umax16(uint16x4_t a, uint16x4_t b);
```

4.169. UMIN8 (SIMD 8-bit Unsigned Minimum)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UMIN8 1001100	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UMIN8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit unsigned integer elements finding minimum operations simultaneously.

Description: This instruction compares the 8-bit unsigned integer elements in Rs1 with the 8-bit unsigned integer elements in Rs2 and selects the numbers that is less than the other one. The selected results are written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] < Rs2.B[x])? Rs1.B[x] : Rs2.B[x];  
for RV32: x=3...0,  
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__umin8(unsigned long a, unsigned long b);
```

RV32:

```
uint8x4_t __nds__v_umin8(uint8x4_t a, uint8x4_t b);
```

RV64:

```
uint8x8_t __nds__v_umin8(uint8x8_t a, uint8x8_t b);
```

4.170. UMIN16 (SIMD 16-bit Unsigned Minimum)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UMIN16 1001000	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UMIN16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer elements finding minimum operations simultaneously.

Description: This instruction compares the 16-bit unsigned integer elements in Rs1 with the 16-bit unsigned integer elements in Rs2 and selects the numbers that is less than the other one. The selected results are written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] < Rs2.H[x])? Rs1.H[x] : Rs2.H[x];  
for RV32: x=1…0,  
for RV64: x=3…0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__umin16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_umin16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_umin16(uint16x4_t a, uint16x4_t b);
```

4.171. UMSR64 (Unsigned Multiply and Subtract from 64-Bit Data)

Type: DSP (64-bit Profile)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UMSR64 1010011	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
UMSR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit unsigned elements in two registers and subtract the 64-bit multiplication results from the 64-bit unsigned data of a pair of registers (RV32) or a register (RV64). The result is written back to the pair of registers (RV32) or a register (RV64).

RV32 Description: This instruction multiplies the 32-bit unsigned data of Rs1 with that of Rs2. It subtracts the 64-bit multiplication result from the 64-bit unsigned data of an even/odd pair of registers specified by Rd(4,1). The subtraction result is written back to the even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction multiplies the 32-bit unsigned elements of Rs1 with that of Rs2. It subtracts the 64-bit multiplication results from the 64-bit unsigned data of Rd. The subtraction result is written back to Rd.

Operations:

- RV32:

```
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H].R[t_L] = R[t_H].R[t_L] - (Rs1 * Rs2);
```

- RV64:

$Rd = Rd - (Rs1.W[0] u* Rs2.W[0]) - (Rs1.W[1] u* Rs2.W[1]);$

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long long __nds__umsr64(unsigned long long t, unsigned long a, unsigned long b);
```

RV64:

```
unsigned long long __nds__v_umsr64(unsigned long long t, uint32x2_t a, uint32x2_t b);
```

4.172. UMUL8, UMULX8

4.172.1. UMUL8 (SIMD Unsigned 8-bit Multiply)

4.172.2. UMULX8 (SIMD Unsigned Crossed 8-bit Multiply)

Type: SIMD

Format:

UMUL8

31 25	24 20	19 15	14 12	11 7	6 0
UMUL8 1011100	Rs2	Rs1	000	Rd	GE80B 1111111

UMULX8

31 25	24 20	19 15	14 12	11 7	6 0
UMULX8 1011101	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UMUL8 Rd, Rs1, Rs2
UMULX8 Rd, Rs1, Rs2
```

Purpose: Do unsigned 8-bit multiplications and generate four 16-bit results simultaneously.

RV32 Description: For the “UMUL8” instruction, multiply the unsigned 8-bit data elements of Rs1 with the corresponding unsigned 8-bit data elements of Rs2.

For the “UMULX8” instruction, multiply the *first* and *second* unsigned 8-bit data elements of Rs1 with the *second* and *first* unsigned 8-bit data elements of Rs2. At the same time, multiply the *third* and *fourth* unsigned 8-bit data elements of Rs1 with the *fourth* and *third* unsigned 8-bit data elements of Rs2.

The four 16-bit results are then written into an even/odd pair of registers specified by Rd(4,1). Rd(4,1), i.e., *d*, determines the even/odd pair group of two registers. Specifically, the register pair includes register *2d* and *2d+1*.

The odd “*2d+1*” register of the pair contains the two 16-bit results calculated from the top part of Rs1 and the even “*2d*” register of the pair contains the two 16-bit results calculated from the bottom part of Rs1.

RV64 Description: For the “UMUL8” instruction, multiply the unsigned 8-bit data elements of Rs1 with the corresponding unsigned 8-bit data elements of Rs2.

For the “UMULX8” instruction, multiply the *first* and *second* unsigned 8-bit data elements of Rs1 with the *second* and *first* unsigned 8-bit data elements of Rs2. At the same time, multiply the *third* and *fourth* unsigned 8-bit data elements of Rs1 with the *fourth* and *third* unsigned 8-bit data elements of Rs2.

The four 16-bit results are then written into Rd. The Rd.W[1] contains the two 16-bit results calculated from the top part of Rs1 and the Rd.W[0] contains the two 16-bit results calculated from the bottom part of Rs1.

Operations:

- RV32:

```

if (is UMUL8) {
    op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x+1]; // top
    op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x]; // bottom
} else if (is UMULX8) {
    op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x]; // Rs1 top
    op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x+1]; // Rs1 bottom
}
rest[x/2] = op1t[x/2] u* op2t[x/2];
resb[x/2] = op1b[x/2] u* op2b[x/2];
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H].H[1] = rest[1]; R[t_H].H[0] = resb[1];
R[t_L].H[1] = rest[0]; R[t_L].H[0] = resb[0];
x = 0 and 2

```

- RV64:

```

if (is “UMUL8”) { op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x+1]; // top op1b[x/2] = Rs1.B[x]; op2b[x/2]
= Rs2.B[x]; // bottom } else if (is “UMULX8”) { op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x]; // Rs1 top
op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x+1]; // Rs1 bottom } rest[x/2] = op1t[x/2] u* op2t[x/2];
resb[x/2] = op1b[x/2] u* op2b[x/2]; t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
Rd.W[1].H[1] = rest[1]; Rd.W[1].H[0] = resb[1]; Rd.W[0].H[1] = rest[0]; Rd.W[0].H[0] = resb[0]; x = 0
and 2

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- UMUL8

```

unsigned long long __nds__umul8(unsigned int a, unsigned int b);
uint16x4_t __nds__v_umul8(uint8x4_t a, uint8x4_t b);

```

- UMULX8

```
unsigned long long __nds__umulx8(unsigned int a, unsigned int b);
uint16x4_t __nds__v_umulx8(uint8x4_t a, uint8x4_t b);
```

4.173. UMUL16, UMULX16

4.173.1. UMUL16 (SIMD Unsigned 16-bit Multiply)

4.173.2. UMULX16 (SIMD Unsigned Crossed 16-bit Multiply)

Type: SIMD

Format:

UMUL16

31 25	24 20	19 15	14 12	11 7	6 0
UMUL16 1011000	Rs2	Rs1	000	Rd	GE80B 1111111

UMULX16

31 25	24 20	19 15	14 12	11 7	6 0
UMULX16 1011001	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
UMUL16 Rd, Rs1, Rs2  
UMULX16 Rd, Rs1, Rs2
```

Purpose: Do unsigned 16-bit multiplications and generate two 32-bit results simultaneously.

RV32 Description: For the “UMUL16” instruction, multiply the top 16-bit U16 content of Rs1 with the top 16-bit U16 content of Rs2. At the same time, multiply the bottom 16-bit U16 content of Rs1 with the bottom 16-bit U16 content of Rs2.

For the “UMULX16” instruction, multiply the top 16-bit U16 content of Rs1 with the bottom 16-bit U16 content of Rs2. At the same time, multiply the bottom 16-bit U16 content of Rs1 with the top 16-bit U16 content of Rs2.

The two U32 results are then written into an even/odd pair of registers specified by Rd(4,1). Rd(4,1), i.e., d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the 32-bit result calculated from the top part of Rs1 and the even “ $2d$ ” register of the pair contains the 32-bit result calculated from the bottom part of Rs1.

RV64 Description: For the “UMUL16” instruction, multiply the top 16-bit U16 content of the lower 32-bit word in Rs1 with the top 16-bit U16 content of the lower 32-bit word in Rs2. At the same time, multiply the bottom 16-bit U16 content of the lower 32-bit word in Rs1 with the bottom 16-bit U16 content of the lower 32-bit word in Rs2.

For the “UMULX16” instruction, multiply the top 16-bit U16 content of the lower 32-bit word in Rs1 with the bottom 16-bit U16 content of the lower 32-bit word in Rs2. At the same time, multiply the bottom 16-bit U16 content of the lower 32-bit word in Rs1 with the top 16-bit U16 content of the lower 32-bit word in Rs2.

The two 32-bit U32 results are then written into Rd. The result calculated from the top 16-bit of the lower 32-bit word in Rs1 is written to Rd.W[1]. And the result calculated from the bottom 16-bit of the lower 32-bit word in Rs1 is written to Rd.W[0]

Operations:

- RV32:

```
if (is UMUL16) {
    op1t = Rs1.H[1]; op2t = Rs2.H[1]; // top
    op1b = Rs1.H[0]; op2b = Rs2.H[0]; // bottom
} else if (is UMULX16) {
    op1t = Rs1.H[1]; op2t = Rs2.H[0]; // Rs1 top
    op1b = Rs1.H[0]; op2b = Rs2.H[1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
    res = aop u* bop;
}
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H] = rest;
R[t_L] = resb;
```

- RV64:

```
if (is UMUL16) {
    op1t = Rs1.H[1]; op2t = Rs2.H[1]; // top
    op1b = Rs1.H[0]; op2b = Rs2.H[0]; // bottom
} else if (is UMULX16) {
    op1t = Rs1.H[1]; op2t = Rs2.H[0]; // Rs1 top
    op1b = Rs1.H[0]; op2b = Rs2.H[1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
    res = aop u* bop;
}
Rd.W[1] = rest;
Rd.W[0] = resb;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- UMUL16

```
unsigned long long __nds__umul16(unsigned int a, unsigned int b);  
uint32x2_t __nds__v_umul16(uint16x2_t a, uint16x2_t b);
```

- UMULX16

```
unsigned long long __nds__umulx16(unsigned int a, unsigned int b);  
uint32x2_t __nds__v_umulx16(uint16x2_t a, uint16x2_t b);
```

4.174. URADD8 (SIMD 8-bit Unsigned Halving Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URADD8 0010100	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
URADD8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit unsigned integer element additions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 8-bit unsigned integer elements in Rs1 with the 8-bit unsigned integer elements in Rs2. The results are first logically right-shifted by 1 bit and then written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] + Rs2.B[x]) u>> 1;  
for RV32: x=3...0,  
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Examples:

- Ra = 0x7F, Rb = 0x7F, Rt = 0x7F
- Ra = 0x80, Rb = 0x80, Rt = 0x80
- Ra = 0x40, Rb = 0x80, Rt = 0x60

Intrinsic functions:

```
unsigned long __nds_uradd8(unsigned long a, unsigned long b);
```

RV32:

```
uint8x4_t __nds__v_uradd8(uint8x4_t a, uint8x4_t b);
```

RV64:

```
uint8x8_t __nds__v_uradd8(uint8x8_t a, uint8x8_t b);
```

4.175. URADD16 (SIMD 16-bit Unsigned Halving Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URADD16 0010000	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
URADD16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer element additions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 16-bit unsigned integer elements in Rs1 with the 16-bit unsigned integer elements in Rs2. The results are first logically right-shifted by 1 bit and then written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] + Rs2.H[x]) u>> 1;  
for RV32: x=1...0,  
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Examples:

- Ra = 0x7FFF, Rb = 0x7FFF Rt = 0x7FFF
- Ra = 0x8000, Rb = 0x8000 Rt = 0x8000
- Ra = 0x4000, Rb = 0x8000 Rt = 0x6000

Intrinsic functions:

```
unsigned long __nds_uradd16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_uradd16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_uradd16(uint16x4_t a, uint16x4_t b);
```

4.176. URADD64 (64-bit Unsigned Halving Addition)

Type: DSP (64-bit Profile)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URADD64 1010000	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
URADD64 Rd, Rs1, Rs2
```

Purpose: Add two 64-bit unsigned integers. The result is halved to avoid overflow or saturation.

RV32 Description: This instruction adds the 64-bit unsigned integer of an even/odd pair of registers specified by Rs1(4,1) with the 64-bit unsigned integer of an even/odd pair of registers specified by Rs2(4,1). The 64-bit addition result is first logically right-shifted by 1 bit and then written to an even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction adds the 64-bit unsigned integer in Rs1 with the 64-bit unsigned integer Rs2. The 64-bit addition result is first logically right-shifted by 1 bit and then written to Rd.

Operations:

- RV32:

```
t_L = CONCAT(Rt(4,1),1'b0); t_H = CONCAT(Rt(4,1),1'b1);
a_L = CONCAT(Ra(4,1),1'b0); a_H = CONCAT(Ra(4,1),1'b1);
b_L = CONCAT(Rb(4,1),1'b0); b_H = CONCAT(Rb(4,1),1'b1);
R[t_H].R[t_L] = (R[a_H].R[a_L] + R[b_H].R[b_L]) u>> 1;
```

- RV64:

```
Rd = (Rs1 + Rs2) u>> 1;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long long __nds__uradd64(unsigned long long a, unsigned long long b);
```

4.177. URADDW (32-bit Unsigned Halving Addition)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URADDW 0011000	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
URADDW Rd, Rs1, Rs2
```

Purpose: Add 32-bit unsigned integers and the results are halved to avoid overflow or saturation.

Description: This instruction adds the first 32-bit unsigned integer in Rs1 with the first 32-bit unsigned integer in Rs2. The result is first logically right-shifted by 1 bit and then sign-extended and written to Rd.

Operations:

- RV32:

$$Rd[31:0] = (Rs1[31:0] + Rs2[31:0]) \text{ u}>> 1;$$

- RV64:

```
resw[31:0] = (Rs1[31:0] + Rs2[31:0]) \text{ u}>> 1;  
Rd[63:0] = SE(resw[31:0]);
```

Exceptions: None

Privilege level: All

Examples:

- Ra = 0xFFFFFFFF, Rb = 0xFFFFFFFF Rt = 0xFFFFFFFF
- Ra = 0x80000000, Rb = 0x80000000 Rt = 0x80000000
- Ra = 0x40000000, Rb = 0x80000000 Rt = 0x60000000

Intrinsic functions:

```
unsigned long __nds_uraddw(unsigned int a, unsigned int b);
```

4.178. URCRAS16 (SIMD 16-bit Unsigned Halving Cross Addition & Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URCRAS16 0010010	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
URCRAS16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer element addition and 16-bit unsigned integer element subtraction in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs1 with the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs2, and subtracts the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs2 from the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs1. The element results are first logically right-shifted by 1 bit and then written to [31:16] of 32-bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] + Rs2.W[x][15:0]) u>> 1;  
Rd.W[x][15:0] = (Rs1.W[x][15:0] - Rs2.W[x][31:16]) u>> 1;  
for RV32, x=0  
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Examples: Please see “URADD16” and “URSUB16” instructions.

Intrinsic functions:

```
unsigned long __nds__urcras16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_urcras16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_urcras16(uint16x4_t a, uint16x4_t b);
```

4.179. URCRSA16 (SIMD 16-bit Unsigned Halving Cross Subtraction & Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URCRSA16 0010011	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
URCRSA16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer element subtraction and 16-bit unsigned integer element addition in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs2 from the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs1, and adds the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs1 with the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs2. The two results are first logically right-shifted by 1 bit and then written to [31:16] of 32-bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] - Rs2.W[x][15:0]) u>> 1;  
Rd.W[x][15:0] = (Rs1.W[x][15:0] + Rs2.W[x][31:16]) u>> 1;  
for RV32, x=0  
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Examples: Please see “URADD16” and “URSUB16” instructions.

Intrinsic functions:

```
unsigned long __nds__urcrsa16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_urcrsa16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_urcrsa16(uint16x4_t a, uint16x4_t b);
```

4.180. URSTAS16 (SIMD 16-bit Unsigned Halving Straight Addition & Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URSTAS16 1101010	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
URSTAS16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer element addition and 16-bit unsigned integer element subtraction in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs1 with the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs2, and subtracts the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs2 from the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs1. The element results are first logically right-shifted by 1 bit and then written to [31:16] of 32-bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] + Rs2.W[x][31:16]) u>> 1;  
Rd.W[x][15:0] = (Rs1.W[x][15:0] - Rs2.W[x][15:0]) u>> 1;  
for RV32, x=0  
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Examples: Please see “URADD16” and “URSUB16” instructions.

Intrinsic functions:

```
unsigned long __nds_urstas16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_urstas16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_urstas16(uint16x4_t a, uint16x4_t b);
```

4.181. URSTSA16 (SIMD 16-bit Unsigned Halving Straight Subtraction & Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URSTSA16 1101011	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
URCRSA16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer element subtraction and 16-bit unsigned integer element addition in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs2 from the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs1, and adds the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs1 with the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs2. The two results are first logically right-shifted by 1 bit and then written to [31:16] of 32-bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] - Rs2.W[x][31:16]) u>> 1;  
Rd.W[x][15:0] = (Rs1.W[x][15:0] + Rs2.W[x][15:0]) u>> 1;  
for RV32, x=0  
for RV64, x=1…0
```

Exceptions: None

Privilege level: All

Examples: Please see “URADD16” and “URSUB16” instructions.

Intrinsic functions:

```
unsigned long __nds__urstsa16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_urstsa16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_urstsa16(uint16x4_t a, uint16x4_t b);
```

4.182. URSUB8 (SIMD 8-bit Unsigned Halving Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URSUB8 0010101	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
URSUB8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit unsigned integer element subtractions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 8-bit unsigned integer elements in Rs2 from the 8-bit unsigned integer elements in Rs1. The results are first logically right-shifted by 1 bit and then written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] - Rs2.B[x]) u>> 1;  
for RV32: x=3...0,  
for RV64: x=7...0
```

Exceptions: None

Privilege level: All

Examples:

- Ra = 0x7F, Rb = 0x80 Rt = 0xFF
- Ra = 0x80, Rb = 0x7F Rt = 0x00
- Ra = 0x80, Rb = 0x40 Rt = 0x20

Intrinsic functions:

```
unsigned long __nds__ursub8(unsigned long a, unsigned long b);
```

RV32:

```
uint8x4_t __nds__v_ursub8(uint8x4_t a, uint8x4_t b);
```

RV64:

```
uint8x8_t __nds__v_ursub8(uint8x8_t a, uint8x8_t b);
```

4.183. URSUB16 (SIMD 16-bit Unsigned Halving Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URSUB16 0010001	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
URSUB16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer element subtractions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 16-bit unsigned integer elements in Rs2 from the 16-bit unsigned integer elements in Rs1. The results are first logically right-shifted by 1 bit and then written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] - Rs2.H[x]) u>> 1;  
for RV32: x=1...0,  
for RV64: x=3...0
```

Exceptions: None

Privilege level: All

Examples:

- Ra = 0x7FFF, Rb = 0x8000 Rt = 0xFFFF
- Ra = 0x8000, Rb = 0x7FFF Rt = 0x0000
- Ra = 0x8000, Rb = 0x4000 Rt = 0x2000

Intrinsic functions:

```
unsigned long __nds__ursub16(unsigned long a, unsigned long b);
```

RV32:

```
uint16x2_t __nds__v_ursub16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __nds__v_ursub16(uint16x4_t a, uint16x4_t b);
```

4.184. URSUB64 (64-bit Unsigned Halving Subtraction)

Type: DSP (64-bit Profile)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URSUB64 1010001	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
URSUB64 Rd, Rs1, Rs2
```

Purpose: Perform a 64-bit unsigned integer subtraction. The result is halved to avoid overflow or saturation.

RV32 Description: This instruction subtracts the 64-bit unsigned integer of an even/odd pair of registers specified by Rs2(4,1) from the 64-bit unsigned integer of an even/odd pair of registers specified by Rs1(4,1). The subtraction result is first logically right-shifted by 1 bit and then written to an even/odd pair of registers specified by Rd(4,1).

Rx(4,1), i.e., d , determines the even/odd pair group of two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the result and the even “ $2d$ ” register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction subtracts the 64-bit unsigned integer in Rs2 from the 64-bit unsigned integer in Rs1. The subtraction result is first logically right-shifted by 1 bit and then written to Rd.

Operations:

- RV32:

```
t_L = CONCAT(Rt(4,1),1\b0); t_H = CONCAT(Rt(4,1),1\b1);
a_L = CONCAT(Ra(4,1),1\b0); a_H = CONCAT(Ra(4,1),1\b1);
b_L = CONCAT(Rb(4,1),1\b0); b_H = CONCAT(Rb(4,1),1\b1);
R[t_H].R[t_L] = (R[a_H].R[a_L] - R[b_H].R[b_L]) u>> 1;
```

- RV64:

```
Rd = (Rs1 - Rs2) u>> 1;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long long __nds__ursub64(unsigned long long a, unsigned long long b);
```

4.185. URSUBW (32-bit Unsigned Halving Subtraction)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URSUBW 0011001	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
URSUBW Rd, Rs1, Rs2
```

Purpose: Subtract 32-bit unsigned integers and the result is halved to avoid overflow or saturation.

Description: This instruction subtracts the first 32-bit signed integer in Rs2 from the first 32-bit signed integer in Rs1. The result is first logically right-shifted by 1 bit and then sign-extended and written to Rd.

Operations:

- RV32:

```
Rd[31:0] = (Rs1[31:0] - Rs2[31:0]) u>> 1;
```

- RV64:

```
resw[31:0] = (Rs1[31:0] - Rs2[31:0]) u>> 1;  
Rd[63:0] = SE(resw[31:0]);
```

Exceptions: None

Privilege level: All

Examples:

- Ra = 0x7FFFFFFF, Rb = 0x80000000 Rt = 0xFFFFFFFF
- Ra = 0x80000000, Rb = 0x7FFFFFFF Rt = 0x00000000
- Ra = 0x80000000, Rb = 0x40000000 Rt = 0x20000000

Intrinsic functions:

```
unsigned long __nds__ursubw(unsigned int a, unsigned int b);
```

4.186. WEXTI (Extract Word from 64-bit Immediate)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
WEXTI 1101111	LSBloc[4:0]	Rs1	000	Rd	GE80B 1111111

Syntax:

```
WEXTI Rd, Rs1, #LSBloc
```

Purpose: Extract a 32-bit word from a 64-bit value stored in an even/odd pair of registers (RV32) or a register (RV64) starting from a specified immediate LSB bit position.

RV32 Description:

This instruction extracts a 32-bit word from a 64-bit value of an even/odd pair of registers specified by Rs1(4,1) starting from a specified immediate LSB bit position, #LSBloc. The extracted word is written to Rd.

Rs1(4,1), i.e., d , determines the even/odd pair group of the two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the 64-bit value and the even “ $2d$ ” register of the pair contains the low 32-bit of the 64-bit value.

RV64 Description:

This instruction extracts a 32-bit word from a 64-bit value in Rs1 starting from a specified immediate LSB bit position, #LSBloc. The extracted word is sign-extended and written to lower 32-bit of Rd.

Operations:

- RV32:

```
Idx0 = CONCAT(Rs1(4,1), 1'b0); Idx1 = CONCAT(Rs2(4,1), 1'b1);
src[63:0] = Concat(R[Idx1], R[Idx0]);
Rd = src[31+LSBloc:LSBloc];
```

- RV64:

```
ExtractW = Rs1[31+LSBloc:LSBloc];
Rd = SE(ExtractW)
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__wext(long long a, unsigned int b);
```

4.187. WEXT (Extract Word from 64-bit)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
WEXT 1100111	Rs2	Rs1	000	Rd	GE80B 1111111

Syntax:

```
WEXT Rd, Rs1, Rs2
```

Purpose: Extract a 32-bit word from a 64-bit value stored in an even/odd pair of registers (RV32) or a register (RV64) starting from a specified LSB bit position in a register.

RV32 Description:

This instruction extracts a 32-bit word from a 64-bit value of an even/odd pair of registers specified by Rs1(4,1) starting from a specified LSB bit position, specified in Rs2[4:0]. The extracted word is written to Rd.

Rs1(4,1), i.e., d , determines the even/odd pair group of the two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

The odd “ $2d+1$ ” register of the pair contains the high 32-bit of the 64-bit value and the even “ $2d$ ” register of the pair contains the low 32-bit of the 64-bit value.

Operations:

- RV32:

```
Idx0 = CONCAT(Rs1(4,1),1'b0); Idx1 = CONCAT(Rs1(4,1),1'b1);
src[63:0] = Concat(R[Idx1], R[Idx0]);
LSBloc = Rs2[4:0];
Rd = src[31+LSBloc:LSBloc];
```

- RV64:

```
LSBloc = Rs2[4:0];
ExtractW = Rs1[31+LSBloc:LSBloc];
Rd = SE(ExtractW)
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_wext(long long a, unsigned int b);
```

4.188. ZUNPKD810, ZUNPKD820, ZUNPKD830, ZUNPKD831, ZUNPKD832

4.188.1. ZUNPKD810 (Unsigned Unpacking Bytes 1 & 0)

4.188.2. ZUNPKD820 (Unsigned Unpacking Bytes 2 & 0)

4.188.3. ZUNPKD830 (Unsigned Unpacking Bytes 3 & 0)

4.188.4. ZUNPKD831 (Unsigned Unpacking Bytes 3 & 1)

4.188.5. ZUNPKD832 (Unsigned Unpacking Bytes 3 & 2)

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP 1010110	ZUNPKD8 <u>xy</u> code[4:0]	Rs1	000	Rd	GE80B 1111111

<u>xy</u>	code[4:0]
10	01100
20	01101
30	01110
31	01111
32	10111

Syntax:

```
ZUNPKD8xy Rd, Rs1  
xy = {10, 20, 30, 31, 32}
```

Purpose: Unpack byte x and byte y of 32-bit chunks in a register into two 16-bit unsigned halfwords of 32-bit chunks in a register.

Description:

For the “ZUNPKD8(x(*y)” instruction, it unpacks byte *x and byte y* of 32-bit chunks in Rs1 into two 16-bit unsigned halfwords and writes the results to the top part and the bottom part of 32-bit chunks in Rd.

Operations:

```
Rd.W[m].H[1] = ZE16(Rs1.W[m].B[x])
Rd.W[m].H[0] = ZE16(Rs1.W[m].B[y])
// ZUNPKD810, x=1,y=0
// ZUNPKD820, x=2,y=0
// ZUNPKD830, x=3,y=0
// ZUNPKD831, x=3,y=1
// ZUNPKD832, x=3,y=2
for RV32: m=0,
for RV64: m=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- ZUNPK810

```
unsigned long __nds__zunpkd810(unsigned long a);
```

```
RV32:
uint16x2_t __nds__v_zunpkd810(uint8x4_t a);
RV64:
uint16x4_t __nds__v_zunpkd810(uint8x8_t a);
```

- ZUNPK820

```
unsigned long __nds__zunpkd820(unsigned long a);
```

```
RV32:
uint16x2_t __nds__v_zunpkd820(uint8x4_t a);
RV64:
uint16x4_t __nds__v_zunpkd820(uint8x8_t a);
```

- ZUNPK830

```
unsigned long __nds__zunpkd830(unsigned long a);
```

```
RV32:  
    uint16x2_t __nds__v_zunpkd830(uint8x4_t a);  
RV64:  
    uint16x4_t __nds__v_zunpkd830(uint8x8_t a);
```

- ZUNPK831

```
unsigned long __nds__zunpkd831(unsigned long a);
```

```
RV32:  
    uint16x2_t __nds__v_zunpkd831(uint8x4_t a);  
RV64:  
    uint16x4_t __nds__v_zunpkd831(uint8x8_t a);
```

- ZUNPK832

```
unsigned long __nds__zunpkd832(unsigned long a);
```

```
RV32:  
    uint16x2_t __nds__v_zunpkd832(uint8x4_t a);  
RV64:  
    uint16x4_t __nds__v_zunpkd832(uint8x8_t a);
```

Chapter 5. Detailed Instruction Descriptions (RV64 Only)

The sections in this chapter describe the detailed operations of the DSP ISA extension instructions for RV64 only in alphabetical order.

5.1. ADD32 (SIMD 32-bit Addition)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
ADD32 0100000		Rs2		Rs1		010		Rd		GE80B 1111111	

Syntax:

```
ADD32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit integer element additions simultaneously.

Description: This instruction adds the 32-bit integer elements in Rs1 with the 32-bit integer elements in Rs2, and then writes the 32-bit element results to Rd.

Operations:

```
Rd.W[x] = Rs1.W[x] + Rs2.W[x];  
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned addition.

Intrinsic functions:

```
unsigned long __nds__add32(unsigned long a, unsigned long b);  
uint32x2_t __nds__v_uadd32(uint32x2_t a, uint32x2_t b);  
int32x2_t __nds__v_sadd32(int32x2_t a, int32x2_t b);
```

5.2. CRAS32 (SIMD 32-bit Cross Addition & Subtraction)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
CRAS32 0100010	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
CRAS32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit integer element addition and 32-bit integer element subtraction in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements.

Description: This instruction adds the 32-bit integer element in [63:32] of Rs1 with the 32-bit integer element in [31:0] of Rs2, and writes the result to [63:32] of Rd; at the same time, it subtracts the 32-bit integer element in [63:32] of Rs2 from the 32-bit integer element in [31:0] of Rs1, and writes the result to [31:0] of Rd.

Operations:

```
Rd.W[1] = Rs1.W[1] + Rs2.W[0];  
Rd.W[0] = Rs1.W[0] - Rs2.W[1];
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned operations.

Intrinsic functions:

```
unsigned long __nds_cras32(unsigned long a, unsigned long b);  
uint32x2_t __nds_v_uclas32(uint32x2_t a, uint32x2_t b);  
int32x2_t __nds_v_sclas32(int32x2_t a, int32x2_t b);
```

5.3. CRSA32 (SIMD 32-bit Cross Subtraction & Addition)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
CRSA32 0100011	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
CRSA32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit integer element subtraction and 32-bit integer element addition in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements.

*Description: *

This instruction subtracts the 32-bit integer element in [31:0] of Rs2 from the 32-bit integer element in [63:32] of Rs1, and writes the result to [63:32] of Rd; at the same time, it adds the 32-bit integer element in [31:0] of Rs1 with the 32-bit integer element in [63:32] of Rs2, and writes the result to [31:0] of Rd

Operations:

```
Rd.W[1] = Rs1.W[1] - Rs2.W[0];  
Rd.W[0] = Rs1.W[0] + Rs2.W[1];
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned operations.

Intrinsic functions:

```
unsigned long __nds__crsa32(unsigned long a, unsigned long b);  
uint32x2_t __nds__v_ursa32(uint32x2_t a, uint32x2_t b);  
int32x2_t __nds__v_srsa32(int32x2_t a, int32x2_t b);
```

5.4. KABS32 (Scalar 32-bit Absolute Value with Saturation)

Type: DSP (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP 1010110	KABS32 10010	Rs1	000	Rd	GE80B 1111111

Syntax:

```
KABS32 Rd, Rs1
```

Purpose: Get the absolute value of signed 32-bit integer elements in a general register.

Description: This instruction calculates the absolute value of signed 32-bit integer elements stored in Rs1. The results are written to Rd. This instruction with the minimum negative integer input of 0x80000000 will produce a saturated output of maximum positive integer of 0x7fffffff and the OV flag will be set to 1.

Operations:

```
if (Rs1.W[x] >= 0) {  
    res[x] = Rs1.W[x];  
} else {  
    If (Rs1.W[x] == 0x80000000) {  
        res[x] = 0x7fffffff;  
        OV = 1;  
    } else {  
        res[x] = -Rs1.W[x];  
    }  
}  
Rd.W[x] = res[x];  
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__kabs32(unsigned long a);
int32x2_t __nds__v_kabs32(int32x2_t a);
```

5.5. KADD32 (SIMD 32-bit Signed Saturating Addition)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KADD32 0001000	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
KADD32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element saturating additions simultaneously.

Description: This instruction adds the 32-bit signed integer elements in Rs1 with the 32-bit signed integer elements in Rs2. If any of the results are beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.W[x] + Rs2.W[x];
if (res[x] > (2^31)-1) {
    res[x] = (2^31)-1;
    OV = 1;
} else if (res[x] < -2^31) {
    res[x] = -2^31;
    OV = 1;
}
Rd.W[x] = res[x];
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__kadd32(unsigned long a, unsigned long b);
int32x2_t __nds__v_kadd32(int32x2_t a, int32x2_t b);
```

5.6. KCRAS32 (SIMD 32-bit Signed Saturating Cross Addition & Subtraction)

Type: SIM (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KCRAS32 0001010	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
KCRAS32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element saturating addition and 32-bit signed integer element saturating subtraction in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements.

Description: This instruction adds the 32-bit integer element in [63:32] of Rs1 with the 32-bit integer element in [31:0] of Rs2; at the same time, it subtracts the 32-bit integer element in [63:32] of Rs2 from the 32-bit integer element in [31:0] of Rs1. If any of the results are beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Operations:

```
res[1] = Rs1.W[1] + Rs2.W[0];
res[0] = Rs1.W[0] - Rs2.W[1];
if (res[x] > (2^31)-1) {
    res[x] = (2^31)-1;
    OV = 1;
} else if (res < -2^31) {
    res[x] = -2^31;
    OV = 1;
}
Rd.W[1] = res[1];
Rd.W[0] = res[0];
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__kcras32(unsigned long a, unsigned long b);
int32x2_t __nds__v_kcras32(int32x2_t a, int32x2_t b);
```

5.7. KCRSA32 (SIMD 32-bit Signed Saturating Cross Subtraction & Addition)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KCRSA32 0001011	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
KCRSA32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element saturating subtraction and 32-bit signed integer element saturating addition in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements.

*Description: *

This instruction subtracts the 32-bit integer element in [31:0] of Rs2 from the 32-bit integer element in [63:32] of Rs1; at the same time, it adds the 32-bit integer element in [31:0] of Rs1 with the 32-bit integer element in [63:32] of Rs2. If any of the results are beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Operations:

```
res[1] = Rs1.W[1] - Rs2.W[0];
res[0] = Rs1.W[0] + Rs2.W[1];
if (res[x] > (2^31)-1) {
    res[x] = (2^31)-1;
    OV = 1;
} else if (res < -2^31) {
    res[x] = -2^31;
    OV = 1;
}
Rd.W[1] = res[1];
Rd.W[0] = res[0];
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__kersa32(unsigned long a, unsigned long b);
int32x2_t __nds__v_kersa32(int32x2_t a, int32x2_t b);
```

5.8. KDMBB16, KDMBT16, KDMTT16

5.8.1. KDMBB16 (SIMD Signed Saturating Double Multiply B16 x B16)

5.8.2. KDMBT16 (SIMD Signed Saturating Double Multiply B16 x T16)

5.8.3. KDMTT16 (SIMD Signed Saturating Double Multiply T16 x T16)

Type: SIMD (RV64 only)

Format:

KDMBB16

31 25	24 20	19 15	14 12	11 7	6 0
KDMBB16 1101101	Rs2	Rs1	001	Rd	GE80B 1111111

KDMBT16

31 25	24 20	19 15	14 12	11 7	6 0
KDMBT16 1110101	Rs2	Rs1	001	Rd	GE80B 1111111

KDMTT16

31 25	24 20	19 15	14 12	11 7	6 0
KDMTT16 1111101	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

KDMxy16 Rd, Rs1, Rs2 (xy = BB, BT, TT)

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the 32-bit chunks in registers and then double and saturate the Q31 results into the 32-bit chunks in the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the 32-bit portions in Rs1 with the top or bottom 16-bit Q15 content of the 32-bit portions in Rs2. The Q30 results are then doubled and saturated into Q31 values. The Q31 values are then written into the 32-bit chunks in Rd. When both the two Q15 inputs are 0x8000, saturation will happen. The result will be saturated to 0x7FFFFFFF and the overflow flag OV will be set.

Operations:

```

// KDMBB16: (x,y,z)=(0,0,0),(2,2,1)
// KDMBT16: (x,y,z)=(0,1,0),(2,3,1)
// KDMTT16: (x,y,z)=(1,1,0),(3,3,1)
aop[z] = Rs1.H[x]; bop[z] = Rs2.H[y];
If (0x8000 != aop[z] | 0x8000 != bop[z]) {
    Mresult[z] = aop[z] * bop[z];
    resQ31[z] = Mresult[z] << 1;
} else {
    resQ31[z] = 0x7FFFFFFF;
    OV = 1;
}
Rd.W[z] = resQ31[z];

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KDMBB16

```

unsigned long __nds__kdmbb16(unsigned long a, unsigned long b);
int32x2_t __nds__v_kdmbb16(int16x4_t a, int16x4_t b);

```

- KDMBT16

```

unsigned long __nds__kdmbt16(unsigned long a, unsigned long b);
int32x2_t __nds__v_kdmbt16(int16x4_t a, int16x4_t b);

```

- KDMTT16

```

unsigned long __nds__kdmtt16(unsigned long a, unsigned long b);
int32x2_t __nds__v_kdmtt16(int16x4_t a, int16x4_t b);

```

5.9. KDMABB16, KDMABT16, KDMATT16

5.9.1. KDMABB16 (SIMD Signed Saturating Double Multiply Addition B16 x B16)

5.9.2. KDMABT16 (SIMD Signed Saturating Double Multiply Addition B16 x T16)

5.9.3. KDMATT16 (SIMD Signed Saturating Double Multiply Addition T16 x T16)

Type: SIMD (RV64 only)

Format:

KDMABB16

31 25	24 20	19 15	14 12	11 7	6 0
KDMABB16 1101100	Rs2	Rs1	001	Rd	GE80B 1111111

KDMABT16

31 25	24 20	19 15	14 12	11 7	6 0
KDMABT16 1110100	Rs2	Rs1	001	Rd	GE80B 1111111

KDMATT16

31 25	24 20	19 15	14 12	11 7	6 0
KDMATT16 1111100	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KDMAxy16 Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the 32-bit chunks in registers and then double and saturate the Q31 results, add the results with the values of the corresponding 32-bit chunks from the destination register and write the saturated addition results back into the corresponding 32-bit chunks of the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the 32-bit portions in Rs1 with the top or bottom 16-bit Q15 content of the corresponding 32-bit portions in Rs2. The Q30 results are then doubled and saturated into Q31 values. The Q31 values are then added with the content of the

corresponding 32-bit portions of Rd. If the addition results are beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), they are saturated to the range and the OV flag is set to 1. The results after saturation are written back to Rd.

When both the two Q15 inputs are 0x8000, saturation will happen and the overflow flag OV will be set.

Operations:

```
// KDMABB16: (x,y,z)=(0,0,0),(2,2,1)
// KDMABT16: (x,y,z)=(0,1,0),(2,3,1)
// KDMATT16: (x,y,z)=(1,1,0),(3,3,1)
aop[z] = Rs1.H[x]; bop[z] = Rs2.H[y];
If (0x8000 != aop[z] | 0x8000 != bop[z]) {
    Mresult[z] = aop[z] * bop[z];
    resQ31[z] = Mresult[z] << 1;
} else {
    resQ31[z] = 0x7FFFFFFF;
    OV = 1;
}
resadd[z] = Rd.W[z] + resQ31[z];
if (resadd[z] > (2^31)-1) {
    resadd[z] = (2^31)-1;
    OV = 1;
} else if (resadd[z] < -2^31) {
    resadd[z] = -2^31;
    OV = 1;
}
Rd.W[z] = resadd[z];
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KDMABB16

```
unsigned long __nds__kdmabb16(unsigned long t, unsigned long a, unsigned long b);
int32x2_t __nds__v_kdmabb16(int32x2_t t, int16x4_t a, int16x4_t b);
```

- KDMABT16

```
unsigned long __nds__kdmabt16(unsigned long t, unsigned long a, unsigned long b);
int32x2_t __nds__v_kdmabt16(int32x2_t t, int16x4_t a, int16x4_t b);
```

- KDMATT16

```
unsigned long __nds__kdmatt16(unsigned long t, unsigned long a, unsigned long b);
int32x2_t __nds__v_kdmatt16(int32x2_t t, int16x4_t a, int16x4_t b);
```

5.10. KHMBB16, KHMBT16, KHMTT16

5.10.1. KHMBB16 (SIMD Signed Saturating Half Multiply B16 x B16)

5.10.2. KHMBT16 (SIMD Signed Saturating Half Multiply B16 x T16)

5.10.3. KHMTT16 (SIMD Signed Saturating Half Multiply T16 x T16)

Type: SIMD (RV64 Only)

Format:

KHMBB16

31 25	24 20	19 15	14 12	11 7	6 0
KHMBB16 1101110	Rs2	Rs1	001	Rd	GE80B 1111111

KHMBT16

31 25	24 20	19 15	14 12	11 7	6 0
KHMBT16 1110110	Rs2	Rs1	001	Rd	GE80B 1111111

KHMTT16

31 25	24 20	19 15	14 12	11 7	6 0
KHMTT16 1111110	Rs2	Rs1	001	Rd	GE80B 1111111

Syntax:

```
KHMxy16 Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the 32-bit chunks in registers and then right-shift 15 bits to turn the Q30 results into Q15 numbers again and saturate the Q15 results into the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the 32-bit portions in Rs1 with the top or bottom 16-bit Q15 content of the 32-bit portion in Rs2. The Q30 results are then right-shifted 15-bits and saturated into Q15 values. The 32-bit Q15 values are then written into the 32-bit chunks in Rd. When both the two Q15 inputs are 0x8000, saturation will happen. The result will be saturated to 0x7FFF and the overflow flag OV will be set.

Operations:

```

// KHMBB16: (x,y,z)=(0,0,0),(2,2,1)
// KHMBT16: (x,y,z)=(0,1,0),(2,3,1)
// KHM TT16: (x,y,z)=(1,1,0),(3,3,1)
aop = Rs1.H[x]; bop = Rs2.H[y];
If (0x8000 != aop | 0x8000 != bop) {
    Mresult[31:0] = aop * bop;
    res[15:0] = Mresult[30:15];
} else {
    res[15:0] = 0x7FFF;
    OV = 1;
}
Rd.W[z] = SE32(res[15:0]);

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KHMBB16

```

unsigned long __nds__khmbb16(unsigned long a, unsigned long b);
int32x2_t __nds__v_khmbb16(int16x4_t a, int16x4_t b);

```

- KHMBT16

```

unsigned long __nds__khmbt16(unsigned long a, unsigned long b);
int32x2_t __nds__v_khmbt16(int16x4_t a, int16x4_t b);

```

- KHM TT16

```

unsigned long __nds__khmtt16(unsigned long a, unsigned long b);
int32x2_t __nds__v_khmtt16(int16x4_t a, int16x4_t b);

```

5.11. KMABB32, KMABT32, KMATT32

5.11.1. KMABB32 (Saturating Signed Multiply Bottom Words & Add)

5.11.2. KMABT32 (Saturating Signed Multiply Bottom & Top Words & Add)

5.11.3. KMATT32 (Saturating Signed Multiply Top Words & Add)

Type: DSP (RV64 Only)

Format:

KMABB32

31 25	24 20	19 15	14 12	11 7	6 0
KMABB32 0101101	Rs2	Rs1	010	Rd	GE80B 1111111

KMABT32

31 25	24 20	19 15	14 12	11 7	6 0
KMABT32 0110101	Rs2	Rs1	010	Rd	GE80B 1111111

KMATT32

31 25	24 20	19 15	14 12	11 7	6 0
KMATT32 0111101	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
KMABB32 Rd, Rs1, Rs2
KMABT32 Rd, Rs1, Rs2
KMATT32 Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit element in a register with the 32-bit element in another register and add the result to the content of 64-bit data in the third register. The addition result may be saturated and is written to the third register.

- KMABB32: rd + bottom*bottom
- KMABT32: rd + bottom*top
- KMATT32: rd + top*top

Description:

For the “KMABB32” instruction, it multiplies the bottom 32-bit element in Rs1 with the bottom 32-bit element in Rs2.

For the “KMABT32” instruction, it multiplies the bottom 32-bit element in Rs1 with the top 32-bit element in Rs2.

For the “KMATT32” instruction, it multiplies the top 32-bit element in Rs1 with the top 32-bit element in Rs2.

The multiplication result is added to the content of 64-bit data in Rd. If the addition result is beyond the Q63 number range ($-2^{63} \leq Q63 \leq 2^{63}-1$), it is saturated to the range and the OV bit is set to 1. The result after saturation is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rd + (Rs1.W[0] * Rs2.W[0]); // KMABB32
res = Rd + (Rs1.W[0] * Rs2.W[1]); // KMABT32
res = Rd + (Rs1.W[1] * Rs2.W[1]); // KMATT32
if (res > (2^63)-1) {
    res = (2^63)-1;
    OV = 1;
} else if (res < -2^63) {
    res = -2^63;
    OV = 1;
}
Rd = res;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMABB32

```
long __nds__kmabb32(long t, unsigned long a, unsigned long b);
long __nds__v_kmabb32(long t, int32x2_t a, int32x2_t b);
```

- KMABT32

```
long __nds__kmabt32(long t, unsigned long a, unsigned long b);
long __nds__v_kmabt32(long t, int32x2_t a, int32x2_t b);
```

- KMATT32

```
long __nds__kmatt32(long t, unsigned long a, unsigned long b);
long __nds__v_kmatt32(long t, int32x2_t a, int32x2_t b);
```

5.12. KMADA32, KMAXDA32

5.12.1. KMADA32 (Saturating Signed Multiply Two Words and Two Adds)

5.12.2. KMAXDA32 (Saturating Signed Crossed Multiply Two Words and Two Adds)

Type: DSP (RV64 Only)

Format:

KMADA32

No encoding, an alias of “KMAR64” instruction

KMAXDA32

31 25	24 20	19 15	14 12	11 7	6 0
KMAXDA32 0100101	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
KMADA32 Rd, Rs1, Rs2
KMAXDA32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from 32-bit data in two registers; and then adds the two 64-bit results and 64-bit data in a third register together. The addition result may be saturated.

- KMADA32: rd + top*top + bottom*bottom
- KMAXDA32: rd + top*bottom + bottom*top

Description:

For the “KMADA32” instruction, it multiplies the bottom 32-bit element in Rs1 with the bottom 32-bit element in Rs2 and then adds the result to the result of multiplying the top 32-bit element in Rs1 with the top 32-bit element in Rs2. It is actually an alias of the “KMAR64” instruction.

For the “KMAXDA32” instruction, it multiplies the top 32-bit element in Rs1 with the bottom 32-bit element in Rs2 and then adds the result to the result of multiplying the bottom 32-bit element in Rs1 with the top 32-bit element in Rs2.

The result is added to the content of 64-bit data in Rd. If the addition result is beyond the Q63 number range ($-2^{63} \leq Q63 \leq 2^{63}-1$), it is saturated to the range and the OV bit is set to 1. The 64-bit result is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```

res = Rd + (Rs1.W[1] * Rs2.w[1]) + (Rs1.W[0] * Rs2.W[0]); // KMADA32
res = Rd + (Rs1.W[1] * Rs2.W[0]) + (Rs1.W[0] * Rs2.W[1]); // KMAXDA32
if (res > (2^63)-1) {
    res = (2^63)-1;
    OV = 1;
} else if (res < -2^63) {
    res = -2^63;
    OV = 1;
}
Rd = res;

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMADA32

```

long __nds__kmada32(long t, unsigned long a, unsigned long b);
long __nds__v_kmada32(long t, int32x2_t a, int32x2_t b);

```

- KMAXDA32

```

long __nds__kmaxda32(long t, unsigned long a, unsigned long b);
long __nds__v_kmaxda32(long t, int32x2_t a, int32x2_t b);

```

5.13. KMDA32, KMXDA32

5.13.1. KMDA32 (Signed Multiply Two Words and Add)

5.13.2. KMXDA32 (Signed Crossed Multiply Two Words and Add)

Type: DSP (RV64 Only)

Format:

KMDA32

31 25	24 20	19 15	14 12	11 7	6 0
KMDA32 0011100	Rs2	Rs1	010	Rd	GE80B 1111111

KMXDA32

31 25	24 20	19 15	14 12	11 7	6 0
KMXDA32 0011101	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
KMDA32 Rd, Rs1, Rs2
KMXDA32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from the 32-bit element of two registers; and then adds the two 64-bit results together. The addition result may be saturated.

- KMDA32: top*top + bottom*bottom
- KMXDA32: top*bottom + bottom*top

Description:

For the “KMDA32” instruction, it multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2 and then adds the result to the result of multiplying the top 32-bit element of Rs1 with the top 32-bit element of Rs2.

For the “KMXDA32” instruction, it multiplies the bottom 32-bit element of Rs1 with the top 32-bit element of Rs2 and then adds the result to the result of multiplying the top 32-bit element of Rs1 with the bottom 32-bit element of Rs2.

The addition result is checked for saturation. If saturation happens, the result is saturated to $2^{63}-1$. The final result is written to Rd. The 32-bit contents are treated as signed integers.

Operations:

```
if ((Rs1 != 0x8000000080000000) or (Rs2 != 0x8000000080000000)) {  
    Rd = (Rs1.W[1] * Rs2.W[1]) + (Rs1.W[0] * Rs2.W[0]); // KMDA32  
    Rd = (Rs1.W[1] * Rs2.W[0]) + (Rs1.W[0] * Rs2.W[1]); // KMXDA32  
} else {  
    Rd = 0x7fffffffffffff;  
    OV = 1;  
}
```

Exceptions: None

Privilege level: All

Note:

- Usage domain: Complex, Statistics, Transform

Intrinsic functions:

- KMDA32

```
long __nds__kmda32(unsigned long a, unsigned long b);  
long __nds__v_kmda32(int32x2_t a, int32x2_t b);
```

- KMXDA32

```
long __nds__kmxda32(unsigned long a, unsigned long b);  
long __nds__v_kmxda32(int32x2_t a, int32x2_t b);
```

5.14. KMADS32, KMADRS32, KMAXDS32

5.14.1. KMADS32 (Saturating Signed Multiply Two Words & Subtract & Add)

5.14.2. KMADRS32 (Saturating Signed Multiply Two Words & Reverse Subtract & Add)

5.14.3. KMAXDS32 (Saturating Signed Crossed Multiply Two Words & Subtract & Add)

Type: DSP (RV64 Only)

Format:

KMADS32

31 25	24 20	19 15	14 12	11 7	6 0
KMADS32 0101110	Rs2	Rs1	010	Rd	GE80B 1111111

KMADRS32

31 25	24 20	19 15	14 12	11 7	6 0
KMADRS32 0110110	Rs2	Rs1	010	Rd	GE80B 1111111

KMAXDS32

31 25	24 20	19 15	14 12	11 7	6 0
KMAXDS32 0111110	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
KMADS32 Rd, Rs1, Rs2
KMADRS32 Rd, Rs1, Rs2
KMAXDS32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from 32-bit elements in two registers; and then perform a subtraction operation between the two 64-bit results. Then add the subtraction result to 64-bit data in a third register. The addition result may be saturated.

- KMADS32: rd + (top*top - bottom*bottom)
- KMADRS32: rd + (bottom*bottom - top*top)
- KMAXDS32: rd + (top*bottom - bottom*top)

Description:

For the “KMADS32” instruction, it multiplies the bottom 32-bit element in Rs1 with the bottom 32-bit element in Rs2 and then subtracts the result from the result of multiplying the top 32-bit element in Rs1 with the top 32-bit element in Rs2.

For the “KMADRS32” instruction, it multiplies the top 32-bit element in Rs1 with the top 32-bit element in Rs2 and then subtracts the result from the result of multiplying the bottom 32-bit element in Rs1 with the bottom 32-bit element in Rs2.

For the “KMAXDS32” instruction, it multiplies the bottom 32-bit element in Rs1 with the top 32-bit element in Rs2 and then subtracts the result from the result of multiplying the top 32-bit element in Rs1 with the bottom 32-bit element in Rs2.

The subtraction result is then added to the content of 64-bit data in Rd. If the addition result is beyond the Q63 number range ($-2^{63} \leq Q63 \leq 2^{63}-1$), it is saturated to the range and the OV bit is set to 1. The 64-bit result after saturation is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rd + (Rs1.W[1] * Rs2.W[1]) - (Rs1.W[0] * Rs2.W[0]); // KMADS32
res = Rd + (Rs1.W[0] * Rs2.W[0]) - (Rs1.W[1] * Rs2.W[1]); // KMADRS32
res = Rd + (Rs1.W[1] * Rs2.W[0]) - (Rs1.W[0] * Rs2.W[1]); // KMAXDS32
if (res > (2^63)-1) {
    res = (2^63)-1;
    OV = 1;
} else if (res < -2^63) {
    res = -2^63;
    OV = 1;
}
Rd = res;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMADS32

```
long __nds__kmads32(long t, unsigned long a, unsigned long b);
long __nds__v_kmads32(long t, int32x2_t a, int32x2_t b);
```

- KMADRS32

```
long __nds__kmadrs32(long t, unsigned long a, unsigned long b);
long __nds__v_kmadrs32(long t, int32x2_t a, int32x2_t b);
```

- KMAXDS32

```
long __nds__kmaxds32(long t, unsigned long a, unsigned long b);
long __nds__v_kmaxds32(long t, int32x2_t a, int32x2_t b);
```

5.15. KMSDA32, KMSXDA32

5.15.1. KMSDA32 (Saturating Signed Multiply Two Words & Add & Subtract)

5.15.2. KMSXDA32 (Saturating Signed Crossed Multiply Two Words & Add & Subtract)

Type: DSP (RV64 Only)

Format:

KMSDA32

31 25	24 20	19 15	14 12	11 7	6 0
KMSDA32 0100110	Rs2	Rs1	010	Rd	GE80B 1111111

KMSXDA32

31 25	24 20	19 15	14 12	11 7	6 0
KMSXDA32 0100111	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
KMSDA32 Rd, Rs1, Rs2
KMSXDA32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from the 32-bit element of two registers; and then subtracts the two 64-bit results from a third register. The subtraction result may be saturated.

- KMSDA: rd - top*top - bottom*bottom
- KMSXDA: rd - top*bottom - bottom*top

Description:

For the “KMSDA32” instruction, it multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2 and multiplies the top 32-bit element of Rs1 with the top 32-bit element of Rs2.

For the “KMSXDA32” instruction, it multiplies the bottom 32-bit element of Rs1 with the top 32-bit element of Rs2 and multiplies the top 32-bit element of Rs1 with the bottom 32-bit element of Rs2.

The two 64-bit multiplication results are then subtracted from the content of Rd. If the subtraction result is beyond the Q63 number range ($-2^{63} \leq Q63 \leq 2^{63}-1$), it is saturated to the range and the OV bit is set to 1. The result after saturation is written to Rd. The 32-bit contents are treated as signed integers.

Operations:

```
res = Rd - (Rs1.W[1] * Rs2.W[1]) - (Rs1.W[0] * Rs2.W[0]); // KMSDA32
res = Rd - (Rs1.W[1] * Rs2.W[0]) - (Rs1.W[0] * Rs2.W[1]); // KMSXDA32
if (res > (2^63)-1) {
    res = (2^63)-1;
    OV = 1;
} else if (res < -2^63) {
    res = -2^63;
    OV = 1;
}
Rd = res;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMSDA32

```
long __nds__kmsda32(long t, unsigned long a, unsigned long b);
long __nds__v_kmsda32(long t, int32x2_t a, int32x2_t b);
```

- KMSXDA32

```
long __nds__kmsxda32(long t, unsigned long a, unsigned long b);
long __nds__v_kmsxda32(long t, int32x2_t a, int32x2_t b);
```

5.16. KSLL32 (SIMD 32-bit Saturating Shift Left Logical)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
KSLL32 0110010		Rs2		Rs1		010		Rd		GE80B 1111111	

Syntax:

```
KSLL32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit elements logical left shift operations with saturation simultaneously. The shift amount is a variable from a GPR.

Description: The 32-bit data elements in Rs1 are left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the low-order 5-bits of the value in the Rs2 register. Any shifted value greater than $2^{31}-1$ is saturated to $2^{31}-1$. Any shifted value smaller than -2^{31} is saturated to -2^{31} . And the saturated results are written to Rd. If any saturation is performed, set OV bit to 1.

Operations:

```
sa = Rs2[4:0];
if (sa != 0) {
    res[(31+sa):0] = Rs1.W[x] << sa;
    if (res > (2^31)-1) {
        res = 0x7fffffff; OV = 1;
    } else if (res < -2^31) {
        res = 0x80000000; OV = 1;
    }
    Rd.W[x] = res[31:0];
} else {
    Rd = Rs1;
}
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ksll32(unsigned long a, unsigned int b);
int32x2_t __nds__v_ksll32(int32x2_t a, unsigned int b);
```

5.17. KSLLI32 (SIMD 32-bit Saturating Shift Left Logical Immediate)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KSLLI32 1000010	Imm5u[4:0]	Rs1	010	Rd	GE80B 1111111

Syntax:

```
KSLLI32 Rd, Rs1, imm5u
```

Purpose: Do 32-bit elements logical left shift operations with saturation simultaneously. The shift amount is an immediate value.

Description: The 32-bit data elements in Rs1 are left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the imm5u constant. Any shifted value greater than $2^{31}-1$ is saturated to $2^{31}-1$. Any shifted value smaller than -2^{31} is saturated to -2^{31} . And the saturated results are written to Rd. If any saturation is performed, set OV bit to 1.

Operations:

```
sa = imm5u[4:0];
if (sa != 0) {
    res[(31+sa):0] = Rs1.W[x] << sa;
    if (res > (2^31)-1) {
        res = 0x7fffffff; OV = 1;
    } else if (res < -2^31) {
        res = 0x80000000; OV = 1;
    }
    Rd.W[x] = res[31:0];
} else {
    Rd = Rs1;
}
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ksll32(unsigned long a, unsigned int b);
int32x2_t __nds__v_ksll32(int32x2_t a, unsigned int b);
```

5.18. KSLRA32, KSLRA32.u

5.18.1. KSLRA32 (SIMD 32-bit Shift Left Logical with Saturation or Shift Right Arithmetic)

5.18.2. KSLRA32.u (SIMD 32-bit Shift Left Logical with Saturation or Rounding Shift Right Arithmetic)

Type: SIMD (RV64 Only)

Format:

KSLRA32

31 25	24 20	19 15	14 12	11 7	6 0
KSLRA32 0101011	Rs2	Rs1	010	Rd	GE80B 1111111

KSLRA32.u

31 25	24 20	19 15	14 12	11 7	6 0
KSLRA32.u 0110011	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
KSLRA32 Rd, Rs1, Rs2
KSLRA32.u Rd, Rs1, Rs2
```

Purpose: Do 32-bit elements logical left (positive) or arithmetic right (negative) shift operation with Q31 saturation for the left shift. The “.u” form performs additional rounding up operations for the right shift.

Description: The 32-bit data elements of Rs1 are left-shifted logically or right-shifted arithmetically based on the value of Rs2[5:0]. Rs2[5:0] is in the signed range of $[-2^5, 2^5-1]$. A positive Rs2[5:0] means logical left shift and a negative Rs2[5:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[5:0]. However, the behavior of “Rs2[5:0]==-2⁵ (0x20)” is defined to be equivalent to the behavior of “Rs2[5:0]==-(2⁵-1) (0x21)”.

The left-shifted results are saturated to the 32-bit signed integer range of $[-2^{31}, 2^{31}-1]$. For the “.u” form of the instruction, the right-shifted results are added a 1 to the most significant discarded bit position for rounding effect. After the shift, saturation, or rounding, the final results are written to Rd. If any saturation happens, this instruction sets the OV flag. The value of Rs2[31:6] will not affect this instruction.

Operations:

```

if (Rs2[5:0] < 0) {
    sa = -Rs2[5:0];
    sa = (sa == 32)? 31 : sa;
    if (!.u form) {
        res[31:-1] = SE33(Rs1.W[x][31:sa-1]) + 1;
        Rd.W[x] = res[31:0];
    } else {
        Rd.W[x] = SE32(Rs1.W[x][31:sa]);
    }
} else {
    sa = Rs2[4:0];
    res[(31+sa):0] = Rs1.W[x] <<(logic) sa;
    if (res > (2^31)-1) {
        res[31:0] = 0x7fffffff; OV = 1;
    } else if (res < -2^31) {
        res[31:0] = 0x80000000; OV = 1;
    }
    Rd.W[x] = res[31:0];
}
for RV64: x=1...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KSLRA32

```

unsigned long __nds__kslra32(unsigned long a, int b);
int32x2_t __nds__v_kslra32(int32x2_t a, int b);

```

- KSLRA32.u

```

unsigned long __nds__kslra32_u(unsigned long a, int b);
int32x2_t __nds__v_kslra32_u(int32x2_t a, int b);

```

5.19. KSTAS32 (SIMD 32-bit Signed Saturating Straight Addition & Subtraction)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KSTAS32 1100000	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
KSTAS32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element saturating addition and 32-bit signed integer element saturating subtraction in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements.

Description: This instruction adds the 32-bit integer element in [63:32] of Rs1 with the 32-bit integer element in [63:32] of Rs2; at the same time, it subtracts the 32-bit integer element in [31:0] of Rs2 from the 32-bit integer element in [31:0] of Rs1. If any of the results are beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Operations:

```
res[1] = Rs1.W[1] + Rs2.W[1];
res[0] = Rs1.W[0] - Rs2.W[0];
if (res[x] > (2^31)-1) {
    res[x] = (2^31)-1;
    OV = 1;
} else if (res < -2^31) {
    res[x] = -2^31;
    OV = 1;
}
Rd.W[1] = res[1];
Rd.W[0] = res[0];
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__kstas32(unsigned long a, unsigned long b);
int32x2_t __nds__v_kstas32(int32x2_t a, int32x2_t b);
```

5.20. KSTSA32 (SIMD 32-bit Signed Saturating Straight Subtraction & Addition)

Type: SIM (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
KSTSA32 1100001	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
KSTSA32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element saturating subtraction and 32-bit signed integer element saturating addition in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements.

*Description: *

This instruction subtracts the 32-bit integer element in [63:32] of Rs2 from the 32-bit integer element in [63:32] of Rs1; at the same time, it adds the 32-bit integer element in [31:0] of Rs1 with the 32-bit integer element in [31:0] of Rs2. If any of the results are beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Operations:

```
res[1] = Rs1.W[1] - Rs2.W[1];
res[0] = Rs1.W[0] + Rs2.W[0];
if (res[x] > (2^31)-1) {
    res[x] = (2^31)-1;
    OV = 1;
} else if (res < -2^31) {
    res[x] = -2^31;
    OV = 1;
}
Rd.W[1] = res[1];
Rd.W[0] = res[0];
for RV64, x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__kstsa32(unsigned long a, unsigned long b);
int32x2_t __nds__v_kstsa32(int32x2_t a, int32x2_t b);
```

5.21. KSUB32 (SIMD 32-bit Signed Saturating Subtraction)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
KSUB32 0001001		Rs2		Rs1		010		Rd		GE80B 1111111	

Syntax:

```
KSUB32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer elements saturating subtractions simultaneously.

Description: This instruction subtracts the 32-bit signed integer elements in Rs2 from the 32-bit signed integer elements in Rs1. If any of the results are beyond the Q31 number range ($-2^{31} \leq Q31 \leq 2^{31}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.W[x] - Rs2.W[x];
if (res[x] > (2^31)-1) {
    res[x] = (2^31)-1;
    OV = 1;
} else if (res[x] < -2^31) {
    res[x] = -2^31;
    OV = 1;
}
Rd.W[x] = res[x];
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ksub32(unsigned long a, unsigned long b);
int32x2_t __nds__v_ksub32(int32x2_t a, int32x2_t b);
```

5.22. PKBB32, PKBT32, PKTT32, PKTB32

5.22.1. PKBB32 (Pack Two 32-bit Data from Both Bottom Half)

5.22.2. PKBT32 (Pack Two 32-bit Data from Bottom and Top Half)

5.22.3. PKTT32 (Pack Two 32-bit Data from Both Top Half)

5.22.4. PKTB32 (Pack Two 32-bit Data from Top and Bottom Half)

Type: DSP (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
PK <u>xy</u> 32 00 <u>zz</u> 111	Rs2	Rs1	010	Rd	GE80B 1111111

<u>xy</u>	<u>zz</u>
BB	00
BT	01
TT	10
TB	11

Syntax:

```
PKBB32 Rd, Rs1, Rs2
PKBT32 Rd, Rs1, Rs2
PKTT32 Rd, Rs1, Rs2
PKTB32 Rd, Rs1, Rs2
```

Purpose: Pack 32-bit data from 64-bit chunks in two registers.

- PKBB32: bottom.bottom
- PKBT32: bottom.top
- PKTT32: top.top
- PKTB32: top.bottom

Description:

(PKBB32) moves Rs1.W[0] to Rd.W[1] and moves Rs2.W[0] to Rd.W[0].

(PKBT32) moves Rs1.W[0] to Rd.W[1] and moves Rs2.W[1] to Rd.W[0].

(PKTT32) moves Rs1.W[1] to Rd.W[1] and moves Rs2.W[1] to Rd.W[0].

(PKTB32) moves Rs1.W[1] to Rd.W[1] and moves Rs2.W[0] to Rd.W[0].

Operations:

```
Rd = CONCAT(Rs1.W[_*0*_*], Rs2.W[_*0*_*]); // PKBB32
Rd = CONCAT(Rs1.W[_*0*_*], Rs2.W[_*1*_*]); // PKBT32
Rd = CONCAT(Rs1.W[_*1*_*], Rs2.W[_*1*_*]); // PKTT32
Rd = CONCAT(Rs1.W[_*1*_*], Rs2.W[_*0*_*]); // PKTB32
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- PKBB32

```
unsigned long __nds_pkbb32(unsigned long a, unsigned long b);
uint32x2_t __nds_v_pkbb32(uint32x2_t a, uint32x2_t b);
```

- PKBT32

```
unsigned long __nds_pkbt32(unsigned long a, unsigned long b);
uint32x2_t __nds_v_pkbt32(uint32x2_t a, uint32x2_t b);
```

- PKTB32

```
unsigned long __nds_pktb32(unsigned long a, unsigned long b);
uint32x2_t __nds_v_pktb32(uint32x2_t a, uint32x2_t b);
```

- PKTT32

```
unsigned long __nds_pktt32(unsigned long a, unsigned long b);
uint32x2_t __nds_v_pktt32(uint32x2_t a, uint32x2_t b);
```

5.23. RADD32 (SIMD 32-bit Signed Halving Addition)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RADD32 0000000	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
RADD32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element additions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 32-bit signed integer elements in Rs1 with the 32-bit signed integer elements in Rs2. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Operations:

```
Rd.W[x] = (Rs1.W[x] + Rs2.W[x]) s>> 1;  
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0x7FFFFFFF, Rs2 = 0x7FFFFFFF Rd = 0x7FFFFFFF
- Rs1 = 0x80000000, Rs2 = 0x80000000 Rd = 0x80000000
- Rs1 = 0x40000000, Rs2 = 0x80000000 Rd = 0xE0000000

Intrinsic functions:

```
unsigned long __nds_radd32(unsigned long a, unsigned long b);  
int32x2_t __nds_v_radd32(int32x2_t a, int32x2_t b);
```

5.24. RCRAS32 (SIMD 32-bit Signed Halving Cross Addition & Subtraction)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RCRAS32 0000010	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
RCRAS32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element addition and 32-bit signed integer element subtraction in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 32-bit signed integer element in [63:32] of Rs1 with the 32-bit signed integer element in [31:0] of Rs2, and subtracts the 32-bit signed integer element in [63:32] of Rs2 from the 32-bit signed integer element in [31:0] of Rs1. The element results are first arithmetically right-shifted by 1 bit and then written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Operations:

```
Rd.W[1] = (Rs1.W[1] + Rs2.W[0]) s>> 1;  
Rd.W[0] = (Rs1.W[0] - Rs2.W[1]) s>> 1;
```

Exceptions: None

Privilege level: All

Examples: Please see “RADD32” and “RSUB32” instructions.

Intrinsic functions:

```
unsigned long __nds_rcras32(unsigned long a, unsigned long b);  
int32x2_t __nds_v_rcras32(int32x2_t a, int32x2_t b);
```

5.25. RCRSA32 (SIMD 32-bit Signed Halving Cross Subtraction & Addition)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RCRSA32 0000011	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
RCRSA32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element subtraction and 32-bit signed integer element addition in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 32-bit signed integer element in [31:0] of Rs2 from the 32-bit signed integer element in [63:32] of Rs1, and adds the 32-bit signed element integer in [31:0] of Rs1 with the 32-bit signed integer element in [63:32] of Rs2. The two results are first arithmetically right-shifted by 1 bit and then written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Operations:

```
Rd.W[1] = (Rs1.W[1] - Rs2.W[0]) s>> 1;  
Rd.W[0] = (Rs1.W[0] + Rs2.W[1]) s>> 1;
```

Exceptions: None

Privilege level: All

Examples: Please see “RADD32” and “RSUB32” instructions.

Intrinsic functions:

```
unsigned long __nds__rcrsa32(unsigned long a, unsigned long b);  
int32x2_t __nds__v_rcrsa32(int32x2_t a, int32x2_t b);
```

5.26. RSTAS32 (SIMD 32-bit Signed Halving Straight Addition & Subtraction)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RCRAS32 1011000	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
RSTAS32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element addition and 32-bit signed integer element subtraction in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 32-bit signed integer element in [63:32] of Rs1 with the 32-bit signed integer element in [63:32] of Rs2, and subtracts the 32-bit signed integer element in [31:0] of Rs2 from the 32-bit signed integer element in [31:0] of Rs1. The element results are first arithmetically right-shifted by 1 bit and then written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Operations:

```
Rd.W[1] = (Rs1.W[1] + Rs2.W[1]) s>> 1;  
Rd.W[0] = (Rs1.W[0] - Rs2.W[0]) s>> 1;
```

Exceptions: None

Privilege level: All

Examples: Please see “RADD32” and “RSUB32” instructions.

Intrinsic functions:

```
unsigned long __nds__rstas32(unsigned long a, unsigned long b);  
int32x2_t __nds__v_rstas32(int32x2_t a, int32x2_t b);
```

5.27. RSTSA32 (SIMD 32-bit Signed Halving Straight Subtraction & Addition)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RSTSA32 1011001	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
RSTSA32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element subtraction and 32-bit signed integer element addition in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 32-bit signed integer element in [63:32] of Rs2 from the 32-bit signed integer element in [63:32] of Rs1, and adds the 32-bit signed element integer in [31:0] of Rs1 with the 32-bit signed integer element in [31:0] of Rs2. The two results are first arithmetically right-shifted by 1 bit and then written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Operations:

```
Rd.W[1] = (Rs1.W[1] - Rs2.W[1]) s>> 1;  
Rd.W[0] = (Rs1.W[0] + Rs2.W[0]) s>> 1;
```

Exceptions: None

Privilege level: All

Examples: Please see “RADD32” and “RSUB32” instructions.

Intrinsic functions:

```
unsigned long __nds__rstsa32(unsigned long a, unsigned long b);  
int32x2_t __nds__v_rstsa32(int32x2_t a, int32x2_t b);
```

5.28. RSUB32 (SIMD 32-bit Signed Halving Subtraction)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
RSUB32 0000001		Rs2		Rs1		010		Rd		GE80B 1111111	

Syntax:

```
RSUB32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element subtractions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 32-bit signed integer elements in Rs2 from the 32-bit signed integer elements in Rs1. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Operations:

```
Rd.W[x] = (Rs1.W[x] - Rs2.W[x]) s>> 1;  
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Examples:

- Ra = 0xFFFFFFFF, Rb = 0x80000000 Rt = 0x7FFFFFFF
- Ra = 0x80000000, Rb = 0x7FFFFFFF Rt = 0x80000000
- Ra = 0x80000000, Rb = 0x40000000 Rt = 0xA0000000

Intrinsic functions:

```
unsigned long __nds_rsub32(unsigned long a, unsigned long b);  
int32x2_t __nds_v_rsub32(int32x2_t a, int32x2_t b);
```

5.29. SLL32 (SIMD 32-bit Shift Left Logical)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SLL32 0101010	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
SLL32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit elements logical left shift operations simultaneously. The shift amount is a variable from a GPR.

Description: The 32-bit elements in Rs1 are left-shifted logically. And the results are written to Rd. The shifted out bits are filled with zero and the shift amount is specified by the low-order 5-bits of the value in the Rs2 register.

Operations:

```
sa = Rs2[4:0];
Rd.W[x] = Rs1.W[x] << sa;
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__sll32(unsigned long a, unsigned int b);
uint32x2_t __nds__v_sll32(uint32x2_t a, unsigned int b);
```

5.30. SLLI32 (SIMD 32-bit Shift Left Logical Immediate)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
SLLI32 0111010		Imm5u[4:0]		Rs1		010		Rd		GE80B 1111111	

Syntax:

```
SLLI32 Rd, Rs1, imm5u[4:0]
```

Purpose: Do 32-bit element logical left shift operations simultaneously. The shift amount is an immediate value.

Description: The 32-bit elements in Rs1 are left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the imm5u[4:0] constant. And the results are written to Rd.

Operations:

```
sa = imm5u[4:0];
Rd.W[x] = Rs1.W[x] << sa;
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds_slli32(unsigned long a, unsigned int b);
uint32x2_t __nds_v_slli32(uint32x2_t a, unsigned int b);
```

5.31. SMAX32 (SIMD 32-bit Signed Maximum)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SMAX32 1001001	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
SMAX32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer elements finding maximum operations simultaneously.

Description: This instruction compares the 32-bit signed integer elements in Rs1 with the 32-bit signed integer elements in Rs2 and selects the numbers that is greater than the other one. The selected results are written to Rd.

Operations:

```
Rd.W[x] = (Rs1.W[x] > Rs2.W[x])? Rs1.W[x] : Rs2.W[x];  
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__smax32(unsigned long a, unsigned long b);  
int32x2_t __nds__v_smax32(int32x2_t a, int32x2_t b);
```

5.32. SMBB32, SMBT32, SMTT32

5.32.1. SMBB32 (Signed Multiply Bottom Word & Bottom Word)

5.32.2. SMBT32 (Signed Multiply Bottom Word & Top Word)

5.32.3. SMTT32 (Signed Multiply Top Word & Top Word)

Type: DSP (RV64 Only)

Format:

SMBB32

No encoding, an alias of “MULSR64” instruction

SMBT32

31 25	24 20	19 15	14 12	11 7	6 0
SMBT32 0001100	Rs2	Rs1	010	Rd	GE80B 1111111

SMTT32

31 25	24 20	19 15	14 12	11 7	6 0
SMTT32 0010100	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
SMBB32 Rd, Rs1, Rs2
SMBT32 Rd, Rs1, Rs2
SMTT32 Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit element of a register with the signed 32-bit element of another register and write the 64-bit result to a third register.

- SMBB32: bottom*bottom
- SMBT32: bottom*top
- SMTT32: top*top

Description:

For the “SMBB32” instruction, it multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2. It is actually an alias of “MULSR64” instruction.

For the “SMBT32” instruction, it multiplies the bottom 32-bit element of Rs1 with the top 32-bit

element of Rs2.

For the “SMTT32” instruction, it multiplies the *top* 32-bit element of Rs1 with the *top* 32-bit element of Rs2.

The 64-bit multiplication result is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rs1.W[0] * Rs2.W[0]; // SMBB32 res = Rs1.W[0] * Rs2.w[1]; // SMBT32 res = Rs1.W[1] * Rs2.W[1];  
// SMTT32 Rd = res;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SMBB32

```
long __nds__smbb32(unsigned long a, unsigned long b);  
long __nds__v_smbb32(int32x2_t a, int32x2_t b);
```

- SMBT32

```
long __nds__smbt32(unsigned long a, unsigned long b);  
long __nds__v_smbt32(int32x2_t a, int32x2_t b);
```

- SMTT32

```
long __nds__smtt32(unsigned long a, unsigned long b);  
long __nds__v_smmt32(int32x2_t a, int32x2_t b);
```

5.33. SMDS32, SMDRS32, SMXDS32

5.33.1. SMDS32 (Signed Multiply Two Words and Subtract)

5.33.2. SMDRS32 (Signed Multiply Two Words and Reverse Subtract)

5.33.3. SMXDS32 (Signed Crossed Multiply Two Words and Subtract)

Type: DSP (RV64 Only)

Format:

SMDS32

31 25	24 20	19 15	14 12	11 7	6 0
SMDS32 0101100	Rs2	Rs1	010	Rd	GE80B 1111111

SMDRS32

31 25	24 20	19 15	14 12	11 7	6 0
SMDRS32 0110100	Rs2	Rs1	010	Rd	GE80B 1111111

SMXDS32

31 25	24 20	19 15	14 12	11 7	6 0
SMXDS32 0111100	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
SMDS32 Rd, Rs1, Rs2
SMDRS32 Rd, Rs1, Rs2
SMXDS32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from the 1 32-bit element of two registers; and then perform a subtraction operation between the two 64-bit results.

- SMDS32: top*top - bottom*bottom
- SMDRS32: bottom*bottom - top*top
- SMXDS32: top*bottom - bottom*top

Description:

For the “SMDS32” instruction, it multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit

element of Rs2 and then subtracts the result from the result of multiplying the top 32-bit element of Rs1 with the top 32-bit element of Rs2.

For the “SMDRS32” instruction, it multiplies the top 32-bit element of Rs1 with the top 32-bit element of Rs2 and then subtracts the result from the result of multiplying the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2.

For the “SMXDS32” instruction, it multiplies the bottom 32-bit element of Rs1 with the top 32-bit element of Rs2 and then subtracts the result from the result of multiplying the top 32-bit element of Rs1 with the bottom 32-bit element of Rs2.

The subtraction result is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
Rt = (Rs1.W[1] * Rs2.W[1]) - (Rs1.W[0] * Rs2.W[0]); // SMDS32  
Rt = (Rs1.W[0] * Rs2.W[0]) - (Rs1.W[1] * Rs2.W[1]); // SMDRS32  
Rt = (Rs1.W[1] * Rs2.W[0]) - (Rs1.W[0] * Rs2.W[1]); // SMXDS32
```

Exceptions: None

Privilege level: All

Note:

- Usage domain: Complex, Statistics, Transform

Intrinsic functions:

- SMDS32

```
long __nds__smds32(unsigned long a, unsigned long b);  
long __nds__v_smds32(int32x2_t a, int32x2_t b);
```

- SMDRS32

```
long __nds__smdrs32(unsigned long a, unsigned long b);  
long __nds__v_smdrs32(int32x2_t a, int32x2_t b);
```

- SMXDS32

```
long __nds__smxds32(unsigned long a, unsigned long b);  
long __nds__v_smxds32(int32x2_t a, int32x2_t b);
```

5.34. SMIN32 (SIMD 32-bit Signed Minimum)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SMIN32 1001000	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
SMIN32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer elements finding minimum operations simultaneously.

Description: This instruction compares the 32-bit signed integer elements in Rs1 with the 32-bit signed integer elements in Rs2 and selects the numbers that is less than the other one. The selected results are written to Rd.

Operations:

```
Rd.W[x] = (Rs1.W[x] < Rs2.W[x])? Rs1.W[x] : Rs2.W[x];  
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__smin32(unsigned long a, unsigned long b);  
int32x2_t __nds__v_smin32(int32x2_t a, int32x2_t b);
```

5.35. SRA32, SRA32.u

5.35.1. SRA32 (SIMD 32-bit Shift Right Arithmetic)

5.35.2. SRA32.u (SIMD 32-bit Rounding Shift Right Arithmetic)

Type: SIMD (RV64 Only)

Format:

SRA32

31 25	24 20	19 15	14 12	11 7	6 0
SRA32 0101000	Rs2	Rs1	010	Rd	GE80B 1111111

SRA32.u

31 25	24 20	19 15	14 12	11 7	6 0
SRA32.u 0110000	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
SRA32 Rd, Rs1, Rs2
SRA32.u Rd, Rs1, Rs2
```

Purpose: Do 32-bit element arithmetic right shift operations simultaneously. The shift amount is a variable from a GPR. The “.u” form performs additional rounding up operations on the shifted results.

Description: The 32-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the data elements. The shift amount is specified by the low-order 5-bits of the value in the Rs2 register. For the rounding operation of the “.u” form, a value of 1 is added to the most significant discarded bit of each 32-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```

sa = Rs2[4:0];
if (sa > 0) {
    if (!.u[] form) { // SRA32.u
        res[31:-1] = SE33(Rs1.W[x][31:sa-1]) + 1;
        Rd.W[x] = res[31:0];
    } else { // SRA32
        Rd.W[x] = SE32(Rs1.W[x][31:sa])
    }
} else {
    Rd = Rs1;
}
for RV64: x=1...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SRA32

```

unsigned long __nds__sra32(unsigned long a, unsigned int b);
int32x2_t __nds__v_sra32(int32x2_t a, unsigned int b);

```

- SRA32.u

```

unsigned long __nds__sra32_u(unsigned long a, unsigned int b);
int32x2_t __nds__v_sra32_u(int32x2_t a, unsigned int b);

```

5.36. SRAI32, SRAI32.u

5.36.1. SRAI32 (SIMD 32-bit Shift Right Arithmetic Immediate)

5.36.2. SRAI32.u (SIMD 32-bit Rounding Shift Right Arithmetic Immediate)

Type: DSP (RV64 Only)

Format:

SRAI32

31 25	24 20	19 15	14 12	11 7	6 0
SRAI32 0111000	Imm5u[4:0]	Rs1	010	Rd	GE80B 1111111

SRAI32.u

31 25	24 20	19 15	14 12	11 7	6 0
SRAI32u 1000000	Imm5u[4:0]	Rs1	010	Rd	GE80B 1111111

Syntax:

```
SRAI32 Rd, Rs1, imm5u
SRAI32.u Rd, Rs1, imm5u
```

Purpose: Do 32-bit elements arithmetic right shift operations simultaneously. The shift amount is an immediate value. The “.u” form performs additional rounding up operations on the shifted results.

Description: The 32-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the 32-bit data elements. The shift amount is specified by the imm5u constant. For the rounding operation of the “.u” form, a value of 1 is added to the most significant discarded bit of each 32-bit data to calculate the final results. And the results are written to Rd.

Operations:

```

sa = imm5u[4:0];
if (sa > 0) {
    if (!.u[] form) { // SRAI32.u
        res[31:-1] = SE33(Rs1.W[x][31:sa-1]) + 1;
        Rd.W[x] = res[31:0];
    } else { // SRAI32
        Rd.W[x] = SE32(Rs1.W[x][31:sa]);
    }
} else {
    Rd = Rs1;
}
for RV64: x=1...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SRAI32

```

unsigned long __nds__sra32(unsigned long a, unsigned int b);
int32x2_t __nds__v_sra32(int32x2_t a, unsigned int b);

```

- SRAI32.u

```

unsigned long __nds__sra32_u(unsigned long a, unsigned int b);
int32x2_t __nds__v_sra32_u(int32x2_t a, unsigned int b);

```

5.37. SRAIW.u (Rounding Shift Right Arithmetic Immediate Word)

Type: DSP (RV64 only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SRAIW.u 0011010	imm5u[4:0]	Rs1	001	Rd	GE80B 1111111

Syntax:

```
SRAIW.u Rd, Rs1, imm5u
```

Purpose: Perform a 32-bit arithmetic right shift operation with rounding. The shift amount is an immediate value.

Description: This instruction right-shifts the lower 32-bit content of Rs1 arithmetically. The shifted out bits are filled with the sign-bit Rs1[31] and the shift amount is specified by the imm5u constant. For the rounding operation, a value of 1 is added to the most significant discarded bit of the data to calculate the final result. And the result is sign-extended on bit 31 to 64 bits and written to Rd.

Operations:

```
sa = imm5u;
if (sa != 0) {
    res[31:-1] = SE33(Rs1[31:(sa-1)]) + 1;
    Rd = SE64(res[31:0]);
} else {
    Rd = SE64(Rs1.W[0]);
}
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
long __nds__sraw_u(int a, unsigned int b);
```

5.38. SRL32, SRL32.u

5.38.1. SRL32 (SIMD 32-bit Shift Right Logical)

5.38.2. SRL32.u (SIMD 32-bit Rounding Shift Right Logical)

Type: SIMD (RV64 Only)

Format:

SRL32

31 25	24 20	19 15	14 12	11 7	6 0
SRL32 0101001	Rs2	Rs1	010	Rd	GE80B 1111111

SRL32.u

31 25	24 20	19 15	14 12	11 7	6 0
SRL32.u 0110001	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
SRL32 Rd, Rs1, Rs2
SRL32.u Rd, Rs1, Rs2
```

Purpose: Do 32-bit element logical right shift operations simultaneously. The shift amount is a variable from a GPR. The “.u” form performs additional rounding up operations on the shifted results.

Description: The 32-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the low-order 5-bits of the value in the Rs2 register. For the rounding operation of the “.u” form, a value of 1 is added to the most significant discarded bit of each 32-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```

sa = Rs2[4:0];
if (sa > 0) {
    if (!.u[] form) { // SRA32.u
        res[31:-1] = ZE33(Rs1.W[x][31:sa-1]) + 1;
        Rd.W[x] = res[31:0];
    } else { // SRA32
        Rd.W[x] = ZE32(Rs1.W[x][31:sa])
    }
} else {
    Rd = Rs1;
}
for RV64: x=1...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SRL32

```

unsigned long __nds__srl32(unsigned long a, unsigned int b);
uint32x2_t __nds__v_srl32(uint32x2_t a, unsigned int b);

```

- SRL32.u

```

unsigned long __nds__srl32_u(unsigned long a, unsigned int b);
uint32x2_t __nds__v_srl32_u(uint32x2_t a, unsigned int b);

```

5.39. SRLI32, SRLI32.u

5.39.1. SRLI32 (SIMD 32-bit Shift Right Logical Immediate)

5.39.2. SRLI32.u (SIMD 32-bit Rounding Shift Right Logical Immediate)

Type: SIMD (RV64 Only)

Format:

SRLI32

31 25	24 20	19 15	14 12	11 7	6 0
SRLI32 0111001	Imm5u[4:0]	Rs1	010	Rd	GE80B 1111111

SRAI32.u

31 25	24 20	19 15	14 12	11 7	6 0
SRLI32u 1000001	Imm5u[4:0]	Rs1	010	Rd	GE80B 1111111

Syntax:

```
SRLI32 Rd, Rs1, imm5u
SRLI32.u Rd, Rs1, imm5u
```

Purpose: Do 32-bit elements logical right shift operations simultaneously. The shift amount is an immediate value. The “.u” form performs additional rounding up operations on the shifted results.

Description: The 32-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the imm5u constant. For the rounding operation of the “.u” form, a value of 1 is added to the most significant discarded bit of each 32-bit data to calculate the final results. And the results are written to Rd.

Operations:

```

sa = imm5u[4:0];
if (sa > 0) {
    if (!.u[] form) { // SRLI32.u
        res[31:-1] = ZE33(Rs1.W[x][31:sa-1]) + 1;
        Rd.W[x] = res[31:0];
    } else { // SRLI32
        Rd.W[x] = ZE32(Rs1.W[x][31:sa]);
    }
} else {
    Rd = Rs1;
}
for RV64: x=1...0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SRLI32

```

unsigned long __nds__srl32(unsigned long a, unsigned int b);
uint32x2_t __nds__v_srl32(uint32x2_t a, unsigned int b);

```

- SRLI32.u

```

unsigned long __nds__srl32_u(unsigned long a, unsigned int b);
uint32x2_t __nds__v_srl32_u(uint32x2_t a, unsigned int b);

```

5.40. STAS32 (SIMD 32-bit Straight Addition & Subtraction)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
STAS32 1111000	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
STAS32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit integer element addition and 32-bit integer element subtraction in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements.

Description: This instruction adds the 32-bit integer element in [63:32] of Rs1 with the 32-bit integer element in [63:32] of Rs2, and writes the result to [63:32] of Rd; at the same time, it subtracts the 32-bit integer element in [31:0] of Rs2 from the 32-bit integer element in [31:0] of Rs1, and writes the result to [31:0] of Rd.

Operations:

```
Rd.W[1] = Rs1.W[1] + Rs2.W[1];  
Rd.W[0] = Rs1.W[0] - Rs2.W[0];
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned operations.

Intrinsic functions:

```
unsigned long __nds__stas32(unsigned long a, unsigned long b);  
uint32x2_t __nds__v_ustas32(uint32x2_t a, uint32x2_t b);  
int32x2_t __nds__v_sstas32(int32x2_t a, int32x2_t b);
```

5.41. STSA32 (SIMD 32-bit Straight Subtraction & Addition)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
STSA32 1111001	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
STSA32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit integer element subtraction and 32-bit integer element addition in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements.

*Description: *

This instruction subtracts the 32-bit integer element in [63:32] of Rs2 from the 32-bit integer element in [63:32] of Rs1, and writes the result to [63:32] of Rd; at the same time, it adds the 32-bit integer element in [31:0] of Rs1 with the 32-bit integer element in [31:0] of Rs2, and writes the result to [31:0] of Rd

Operations:

```
Rd.W[1] = Rs1.W[1] - Rs2.W[1];  
Rd.W[0] = Rs1.W[0] + Rs2.W[0];
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned operations.

Intrinsic functions:

```
unsigned long __nds__stsa32(unsigned long a, unsigned long b);  
uint32x2_t __nds__v_ustsa32(uint32x2_t a, uint32x2_t b);  
int32x2_t __nds__v_sstsa32(int32x2_t a, int32x2_t b);
```

5.42. SUB32 (SIMD 32-bit Subtraction)

Type: DSP (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
SUB32 0100001	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
SUB32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit integer element subtractions simultaneously.

Description: This instruction subtracts the 32-bit integer elements in Rs2 from the 32-bit integer elements in Rs1, and then writes the results to Rd.

Operations:

```
Rd.W[x] = Rs1.W[x] - Rs2.W[x];  
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note: This instruction can be used for either signed or unsigned subtraction.

Intrinsic functions:

```
unsigned long __nds__sub32(unsigned long a, unsigned long b);  
uint32x2_t __nds__v_usub32(uint32x2_t a, uint32x2_t b);  
int32x2_t __nds__v_ssub32(int32x2_t a, int32x2_t b);
```

5.43. UKADD32 (SIMD 32-bit Unsigned Saturating Addition)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKADD32 0011000	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
UKADD32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer element saturating additions simultaneously.

Description: This instruction adds the 32-bit unsigned integer elements in Rs1 with the 32-bit unsigned integer elements in Rs2. If any of the results are beyond the 32-bit unsigned number range ($0 \leq \text{RES} \leq 2^{32}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.W[x] + Rs2.W[x];
if (res[x] > (2^32)-1) {
    res[x] = (2^32)-1;
    OV = 1;
}
Rd.W[x] = res[x];
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ukadd32(unsigned long a, unsigned long b);
uint32x2_t __nds__v_ukadd32(uint32x2_t a, uint32x2_t b);
```

5.44. UKCRAS32 (SIMD 32-bit Unsigned Saturating Cross Addition & Subtraction)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKCRAS32 0011010	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
UKCRAS32 Rd, Rs1, Rs2
```

Purpose: Do one 32-bit unsigned integer element saturating addition and one 32-bit unsigned integer element saturating subtraction in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements.

Description: This instruction adds the 32-bit unsigned integer element in [63:32] of Rs1 with the 32-bit unsigned integer element in [31:0] of Rs2; at the same time, it subtracts the 32-bit unsigned integer element in [63:32] of Rs2 from the 32-bit unsigned integer element in [31:0] of Rs1. If any of the results are beyond the 32-bit unsigned number range ($0 \leq \text{RES} \leq 2^{32}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Operations:

```
res1 = Rs1.W[1] + Rs2.W[0];
res2 = Rs1.W[0] - Rs2.W[1];
if (res1 > (2^32)-1) {
    res1 = (2^32)-1;
    OV = 1;
}
if (res2 < 0) {
    res2 = 0;
    OV = 1;
}
Rd.W[1] = res1;
Rd.W[0] = res2;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ukcras32(unsigned long a, unsigned long b);  
uint32x2_t __nds__v_ukcras32(uint32x2_t a, uint32x2_t b);
```

5.45. UKCRSA32 (SIMD 32-bit Unsigned Saturating Cross Subtraction & Addition)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKRSA32 0011011	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
UKRSA32 Rd, Rs1, Rs2
```

Purpose: Do one 32-bit unsigned integer element saturating subtraction and one 32-bit unsigned integer element saturating addition in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements.

Description: This instruction subtracts the 32-bit unsigned integer element in [31:0] of Rs2 from the 32-bit unsigned integer element in [63:32] of Rs1; at the same time, it adds the 32-bit unsigned integer element in [63:32] of Rs2 with the 32-bit unsigned integer element in [31:0] Rs1. If any of the results are beyond the 32-bit unsigned number range ($0 \leq \text{RES} \leq 2^{32}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Operations:

```
res1 = Rs1.W[1] - Rs2.W[0];
res2 = Rs1.W[0] + Rs2.W[1];
if (res1 < 0) {
    res1 = 0;
    OV = 1;
} else if (res2 > (2^32)-1) {
    res2 = (2^32)-1;
    OV = 1;
}
Rd.W[1] = res1;
Rd.W[0] = res2;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ukrsa32(unsigned long a, unsigned long b);  
uint32x2_t __nds__v_ukrsa32(uint32x2_t a, uint32x2_t b);
```

5.46. UKSTAS32 (SIMD 32-bit Unsigned Saturating Straight Addition & Subtraction)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKSTAS32 1110000	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
UKSTAS32 Rd, Rs1, Rs2
```

Purpose: Do one 32-bit unsigned integer element saturating addition and one 32-bit unsigned integer element saturating subtraction in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements.

Description: This instruction adds the 32-bit unsigned integer element in [63:32] of Rs1 with the 32-bit unsigned integer element in [63:32] of Rs2; at the same time, it subtracts the 32-bit unsigned integer element in [31:0] of Rs2 from the 32-bit unsigned integer element in [31:0] of Rs1. If any of the results are beyond the 32-bit unsigned number range ($0 \leq \text{RES} \leq 2^{32}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Operations:

```
res1 = Rs1.W[1] + Rs2.W[1];
res2 = Rs1.W[0] - Rs2.W[0];
if (res1 > (2^32)-1) {
    res1 = (2^32)-1;
    OV = 1;
}
if (res2 < 0) {
    res2 = 0;
    OV = 1;
}
Rd.W[1] = res1;
Rd.W[0] = res2;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ukstas32(unsigned long a, unsigned long b);  
uint32x2_t __nds__v_ukstas32(uint32x2_t a, uint32x2_t b);
```

5.47. UKSTSA32 (SIMD 32-bit Unsigned Saturating Straight Subtraction & Addition)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKSTSA32 1110001	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
UKSTSA32 Rd, Rs1, Rs2
```

Purpose: Do one 32-bit unsigned integer element saturating subtraction and one 32-bit unsigned integer element saturating addition in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements.

Description: This instruction subtracts the 32-bit unsigned integer element in [63:32] of Rs2 from the 32-bit unsigned integer element in [63:32] of Rs1; at the same time, it adds the 32-bit unsigned integer element in [31:0] of Rs2 with the 32-bit unsigned integer element in [31:0] of Rs1. If any of the results are beyond the 32-bit unsigned number range ($0 \leq \text{RES} \leq 2^{32}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Operations:

```
res1 = Rs1.W[1] - Rs2.W[1];
res2 = Rs1.W[0] + Rs2.W[0];
if (res1 < 0) {
    res1 = 0;
    OV = 1;
} else if (res2 > (2^32)-1) {
    res2 = (2^32)-1;
    OV = 1;
}
Rd.W[1] = res1;
Rd.W[0] = res2;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__ukstsa32(unsigned long a, unsigned long b);  
uint32x2_t __nds__v_ukstsa32(uint32x2_t a, uint32x2_t b);
```

5.48. UKSUB32 (SIMD 32-bit Unsigned Saturating Subtraction)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UKSUB32 0011001	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
UKSUB32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer elements saturating subtractions simultaneously.

Description: This instruction subtracts the 32-bit unsigned integer elements in Rs2 from the 32-bit unsigned integer elements in Rs1. If any of the results are beyond the 32-bit unsigned number range ($0 \leq \text{RES} \leq 2^{32}-1$), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.W[x] - Rs2.W[x];
if (res[x] < 0) {
    res[x] = 0;
    OV = 1;
}
Rd.W[x] = res[x];
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__uksub32(unsigned long a, unsigned long b);
uint32x2_t __nds__v_uksub32(uint32x2_t a, uint32x2_t b);
```

5.49. UMAX32 (SIMD 32-bit Unsigned Maximum)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UMAX32 1010001	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
UMAX32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer elements finding maximum operations simultaneously.

Description: This instruction compares the 32-bit unsigned integer elements in Rs1 with the 32-bit unsigned integer elements in Rs2 and selects the numbers that is greater than the other one. The selected results are written to Rd.

Operations:

```
Rd.W[x] = (Rs1.W[x] > Rs2.W[x])? Rs1.W[x] : Rs2.W[x];  
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__umax32(unsigned long a, unsigned long b);  
uint32x2_t __nds__v_umax32(uint32x2_t a, uint32x2_t b);
```

5.50. UMIN32 (SIMD 32-bit Unsigned Minimum)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
UMIN32 1010000	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
UMIN32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer elements finding minimum operations simultaneously.

Description: This instruction compares the 32-bit unsigned integer elements in Rs1 with the 32-bit unsigned integer elements in Rs2 and selects the numbers that is less than the other one. The selected results are written to Rd.

Operations:

```
Rd.W[x] = (Rs1.W[x] < Rs2.W[x])? Rs1.W[x] : Rs2.W[x];  
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
unsigned long __nds__umin32(unsigned long a, unsigned long b);  
uint32x2_t __nds__v_umin32(uint32x2_t a, uint32x2_t b);
```

5.51. URADD32 (SIMD 32-bit Unsigned Halving Addition)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URADD32 0010000	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
URADD32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer element additions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 32-bit unsigned integer elements in Rs1 with the 32-bit unsigned integer elements in Rs2. The results are first logically right-shifted by 1 bit and then written to Rd.

Operations:

```
Rd.W[x] = (Rs1.W[x] + Rs2.W[x]) u>> 1;  
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Examples:

- Ra = 0x7FFFFFFF, Rb = 0x7FFFFFFF Rt = 0x7FFFFFFF
- Ra = 0x80000000, Rb = 0x80000000 Rt = 0x80000000
- Ra = 0x40000000, Rb = 0x80000000 Rt = 0x60000000

Intrinsic functions:

```
unsigned long __nds_uradd32(unsigned long a, unsigned long b);  
uint32x2_t __nds_v_uradd32(uint32x2_t a, uint32x2_t b);
```

5.52. URCRAS32 (SIMD 32-bit Unsigned Halving Cross Addition & Subtraction)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URCRAS32 0010010	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
URCRAS32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer element addition and 32-bit unsigned integer element subtraction in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 32-bit unsigned integer element in [63:32] of Rs1 with the 32-bit unsigned integer element in [31:0] of Rs2, and subtracts the 32-bit unsigned integer element in [63:32] of Rs2 from the 32-bit unsigned integer element in [31:0] of Rs1. The element results are first logically right-shifted by 1 bit and then written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Operations:

```
Rd.W[1] = (Rs1.W[1] + Rs2.W[0]) u>> 1;  
Rd.W[0] = (Rs1.W[0] - Rs2.W[1]) u>> 1;
```

Exceptions: None

Privilege level: All

Examples: Please see “URADD32” and “URSUB32” instructions.

Intrinsic functions:

```
unsigned long __nds__urcras32(unsigned long a, unsigned long b);  
uint32x2_t __nds__v_urcras32(uint32x2_t a, uint32x2_t b);
```

5.53. URCRSA32 (SIMD 32-bit Unsigned Halving Cross Subtraction & Addition)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URCRSA32 0010011	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
URCRSA32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer element subtraction and 32-bit unsigned integer element addition in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 32-bit unsigned integer element in [31:0] of Rs2 from the 32-bit unsigned integer element in [63:32] of Rs1, and adds the 32-bit unsigned element integer in [31:0] of Rs1 with the 32-bit unsigned integer element in [63:32] of Rs2. The two results are first logically right-shifted by 1 bit and then written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Operations:

```
Rd.W[1] = (Rs1.W[1] - Rs2.W[0]) u>> 1;  
Rd.W[0] = (Rs1.W[0] + Rs2.W[1]) u>> 1;
```

Exceptions: None

Privilege level: All

Examples: Please see “URADD32” and “URSUB32” instructions.

Intrinsic functions:

```
unsigned long __nds__urcrsa32(unsigned long a, unsigned long b);  
uint32x2_t __nds__v_urcrsa32(uint32x2_t a, uint32x2_t b);
```

5.54. URSTAS32 (SIMD 32-bit Unsigned Halving Straight Addition & Subtraction)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URCRAS32 1101000	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
URSTAS32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer element addition and 32-bit unsigned integer element subtraction in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 32-bit unsigned integer element in [63:32] of Rs1 with the 32-bit unsigned integer element in [63:32] of Rs2, and subtracts the 32-bit unsigned integer element in [31:0] of Rs2 from the 32-bit unsigned integer element in [31:0] of Rs1. The element results are first logically right-shifted by 1 bit and then written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Operations:

```
Rd.W[1] = (Rs1.W[1] + Rs2.W[1]) u>> 1;  
Rd.W[0] = (Rs1.W[0] - Rs2.W[0]) u>> 1;
```

Exceptions: None

Privilege level: All

Examples: Please see “URADD32” and “URSUB32” instructions.

Intrinsic functions:

```
unsigned long __nds_urstas32(unsigned long a, unsigned long b);  
uint32x2_t __nds_v_urstas32(uint32x2_t a, uint32x2_t b);
```

5.55. URSTSA32 (SIMD 32-bit Unsigned Halving Straight Subtraction & Addition)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URSTSA32 1101001	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
URSTSA32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer element subtraction and 32-bit unsigned integer element addition in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 32-bit unsigned integer element in [63:32] of Rs2 from the 32-bit unsigned integer element in [63:32] of Rs1, and adds the 32-bit unsigned element integer in [31:0] of Rs1 with the 32-bit unsigned integer element in [31:0] of Rs2. The two results are first logically right-shifted by 1 bit and then written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Operations:

```
Rd.W[1] = (Rs1.W[1] - Rs2.W[1]) u>> 1;  
Rd.W[0] = (Rs1.W[0] + Rs2.W[0]) u>> 1;
```

Exceptions: None

Privilege level: All

Examples: Please see “URADD32” and “URSUB32” instructions.

Intrinsic functions:

```
unsigned long __nds_urstsa32(unsigned long a, unsigned long b);  
uint32x2_t __nds_v_urstsa32(uint32x2_t a, uint32x2_t b);
```

5.56. URSUB32 (SIMD 32-bit Unsigned Halving Subtraction)

Type: SIMD (RV64 Only)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
URSUB32 0010001	Rs2	Rs1	010	Rd	GE80B 1111111

Syntax:

```
URSUB32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer element subtractions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 32-bit unsigned integer elements in Rs2 from the 32-bit unsigned integer elements in Rs1. The results are first logically right-shifted by 1 bit and then written to Rd.

Operations:

```
Rd.W[x] = (Rs1.W[x] - Rs2.W[x]) u>> 1;  
for RV64: x=1...0
```

Exceptions: None

Privilege level: All

Examples:

- Ra = 0x7FFFFFFF, Rb = 0x80000000, Rt = 0xFFFFFFFF
- Ra = 0x80000000, Rb = 0x7FFFFFFF, Rt = 0x00000000
- Ra = 0x80000000, Rb = 0x40000000, Rt = 0x20000000

Intrinsic functions:

```
unsigned long __nds__ursub32(unsigned long a, unsigned long b);  
uint32x2_t __nds__v_ursub32(uint32x2_t a, uint32x2_t b);
```

Chapter 6. New User Control & Status Registers

Brief Summary

Symbolic Mnemonics	CSR Address				Hex
	[11:10]	[9:8]	[7:6]	[5:0]	
ucode	10	00	00	000001	0x801

6.1. Code Register

Mnemonic Name: ucode

IM Requirement: Xdsp extension optional

Access Mode: User

CSR Address: 0x801 (non-standard read/write)

XLEN: 64 and 32

This register stores the overflow flag of the P extension.

XLEN-1	1	0
Reserved		OV

Field Name	Bits	Description	Type	Reset
OV	[0]	Overflow flag. It will be set by many DSP instructions when a saturated result is generated.	RW	0
Reserved	[XLEN-1:1]	Reserved	RAZWI	0

Chapter 7. Instruction Encoding Table

Table 36. P Extension Instruction Encoding for $\text{funct3}[14:12]==0b000$.

funct3 == 000								
funct7	[2:0]							
[6:3]	000	001	010	011	100	101	110	111
0000	radd16	rsub16	rcras16	rcrsa16	radd8	rsub8	scmplt16	scmplt8
0001	kadd16	ksub16	kcras16	kcrsa16	kadd8	ksub8	scmple16	scmple8
0010	uradd16	ursub16	urcras16	urcrsa16	uradd8	ursub8	ucmplt16	ucmplt8
0011	ukadd16	uksub16	ukcras16	ukcrsa16	ukadd8	uksub8	ucmple16	ucmple8
0100	add16	sub16	cras16	crsa16	add8	sub8	cmpeq16	cmpeq8
0101	sra16	srl16	sll16	kslra16	sra8	srl8	sll8	kslra8
0110	sra16.u	srl16.u	ksll16	kslra16.u	sra8.u	srl8.u	ksll8	kslra8.u
0111	srai16/u	srl16/u	slli16/kslli16		srai8/u	srl18/u	slli8/kslli8	
1000	smin16	smax16	sclip16/uclip16	khm16	smin8	smax8	sclip8/uclip8	khm8
1001	umin16	umax16		khmx16	umin8	umax8		khmx8
1010	smul16	smulx16			smul8	smulx8	oneop	oneop2
1011	umul16	umulx16			umul8	umulx8		
1100	smul16h	smulx16h		khm32	smaqa	smaqa.su	umaqa	wext
1101	umul16h	umulx16h		khmx32				wexti
1110	ave		sclip32	bitrev	bitrevi			
1111	minw	maxw	uclip32		mtlbi	mtlei	pbsad	pbsada

Table 37. Instruction Encoding for $\text{funct3}[14:12]==0b000$ and $\text{funct7}[31:25]==0b1010110$ (oneop).

ONEOP == 1010110								
subf5	[2:0]							
[4:3]	000	001	010	011	100	101	110	111
00	insb							
01	sunpkd810	sunpkd820	sunpkd830	sunpkd831	zunpkd810	zunpkd820	zunpkd830	zunpkd831
10	kabs8	kabs16	kabs32	sunpkd832	kabsw			zunpkd832
11	swap8	swap16	clz	clo				

Table 38. Instruction Encoding for funct3[14:12]==0b000 and funct7[31:25]==0b1010111 (oneop2).

ONEOP2 == 1010111								
subf5	[2:0]							
[4:3]	000	001	010	011	100	101	110	111
00	clrs8	clz8	clo8					
01	clrs16	clz16	clo16					
10	clrs32	clz32	clo32					
11								

Table 39. P Extension Instruction Encoding for funct3[14:12]==0b001.

funct3 == 001								
funct7	[2:0]							
[6:3]	000	001	010	011	100	101	110	111
0000	kaddw	ksubw	kaddh	ksubh	smbb16	kdmbb	khmbb	pkbb16
0001	ukaddw	uksubw	ukaddh	uksubh	smbt16	kdmbt	khmbt	pkbt16
0010	raddw	rsubw	sra.u	ksll	smtt16	kdmtt	khmtt	pktt16
0011	uraddw	ursubw	sraiw.u	kslli	kmda	kmxda		pktb16
0100	smmul	kmmsb	smmwb	kmmawb	kmada	kmaxda	kmsda	kmsxda
0101	smmul.u	kmmsb.u	smmwb.u	kmmawb.u	smds	kmabb	kmads	smal
0110	kmmac	kwmmul	smmwt	kmmawt	smdrs	kmabt	kmadrs	kslraw
0111	kmmac.u	kwmmul.u	smmwt.u	kmmawt.u	smxds	kmatt	kmaxds	kslraw.u
1000	radd64	rsub64	smar64	smsr64	smalbb	smalds	smalda	kmmwb2
1001	kadd64	ksub64	kmar64	kmsr64	smalbt	smaldrs	smalxda	kmmwb2.u
1010	uradd64	ursub64	umar64	umsr64	smaltt	smalxds	smslda	kmmwt2
1011	ukadd64	uksub64	ukmar64	ukmsr64			smslxda	kmmwt2.u
1100	add64	sub64	maddr32	msubr32				kmmawb2
1101		kdmbb	srai.u	kdmabb16	kdmbb16	khmbb16	kmmawb2.u	
1110	mulsr64	kdmbt			kdmbt16	kdmbt16	khmbt16	kmmwt2
1111	mulr64	kdmat			kdmat16	kdmat16	khmat16	kmmwt2.u

Table 40. P Extension Instruction Encoding for funct3[14:12]==0b010.

funct3 == 010								
funct7	[2:0]							
[6:3]	000	001	010	011	100	101	110	111
0000	radd32	rsub32	rcras32	rcrsa32				pkbb32
0001	kadd32	ksub32	kcras32	kcrsa32	smbt32			pkbt32
0010	uradd32	ursub32	urcras32	urcrsa32	smtt32			pktt32
0011	ukadd32	uksub32	ukcras32	ukcrsa32	kmda32	kmxda32		pktb32
0100	add32	sub32	cras32	crsa32		kmaxda3 2	kmsda32	kmsxda3 2
0101	sra32	srl32	sll32	kslra32	smds32	kmabb32	kmads32	
0110	sra32.u	srl32.u	ksll32	kslra32.u	smdrs32	kmabt32	kmadrs32	
0111	srai32	srli32	slli32		smxds32	kmatt32	kmaxds3 2	
1000	srai32.u	srli32.u	kslli32					
1001	smin32	smax32						
1010	umin32	umax32						
1011	rstas32	rstsa32	rstas16	rstsa16				
1100	kstas32	kstsa32	kstas16	kstsa16				
1101	urstas32	urstsa32	urstas16	urstsa16				
1110	ukstas32	ukstsa32	ukstas16	ukstsa16				
1111	stas32	stsa32	stas16	stsa16				

Table 41. P Extension Instruction Encoding for funct3[14:12]==0b011.

funct3 == 011								
funct7	[2:0]							
[6:3]	000	001	010	011	100	101	110	111

0000	bpick					bpick			
0001									
0010									
0011									
0100									
0101									
0110									
0111									
1000									
1001									
1010									
1011									
1100									
1101									
1110									
1111									

Appendix A: Instruction Latency and Throughput

A.1. Example RV32 and RV64 Cores with 5 Stages of Pipeline

A.1.1. Listed by Individual Instruction

Table 42. RV64/RV32 DSP Instruction Latency

No.	Instruction	Latency (cycle)	Throughput (Completed Inst/Cycle)
1	ADD8	1	1
2	ADD16	1	1
3	ADD64	1	1
4	AVE	1	1
5	BITREV	1	1
6	BITREVI	1	1
7	BPICK	1	1
8	CLROV	1	1
9	CLRS8	1	1
10	CLRS16	1	1
11	CLRS32	1	1
12	CLO8	1	1
13	CLO16	1	1
14	CLO32	1	1
15	CLZ8	1	1
16	CLZ16	1	1
17	CLZ32	1	1
18	CMPEQ8	1	1
19	CMPEQ16	1	1
20	CRAS16	1	1
21	CRSA16	1	1
22	INSB	1	1
23	KABS8	1	1
24	KABS16	1	1

No.	Instruction	Latency (cycle)	Throughput (Completed Inst/Cycle)
25	KABSW	1	1
26	KADD8	1	1
27	KADD16	1	1
28	KADD64	1	1
29	KADDH	1	1
30	KADDW	1	1
31	KCRAS16	1	1
32	KCRSA16	1	1
33	KDMBB	1	1
34	KDMBT	1	1
35	KDMTT	1	1
36	KDMABB	2	1
37	KDMABT	2	1
38	KDMATT	2	1
39	KHM8	1	1
40	KHMX8	1	1
41	KHM16	1	1
42	KHMX16	1	1
43	KHMBB	1	1
44	KHMBT	1	1
45	KHMTT	1	1
46	KMABB	2	1
47	KMABT	2	1
48	KMATT	2	1
49	KMADA	2	1
50	KMAXDA	2	1
51	KMADS	2	1
52	KMADRS	2	1
53	KMAXDS	2	1
54	KMAR64	3	1
55	KMDA	2	1
56	KMXDA	2	1
57	KMMAC	2	1

No.	Instruction	Latency (cycle)	Throughput (Completed Inst/Cycle)
58	KMMAC.u	2	1
59	KMMAWB	2	1
60	KMMAWB.u	2	1
61	KMMAWB2	2	1
62	KMMAWB2.u	2	1
63	KMMAWT	2	1
64	KMMAWT.u	2	1
65	KMMAWT2	2	1
66	KMMAWT2.u	2	1
67	KMMSB	2	1
68	KMMSB.u	2	1
69	KMMWB2	2	1
70	KMMWB2.u	2	1
71	KMMWT2	2	1
72	KMMWT2.u	2	1
73	KMSDA	2	1
74	KMSXDA	2	1
75	KMSR64	3	1
76	KSLLW	1	1
77	KSLLIW	1	1
78	KSLL8	1	1
79	KSLLI8	1	1
80	KSLL16	1	1
81	KSLLI16	1	1
82	KSLRA8	1	1
83	KSLRA8.u	1	1
84	KSLRA16	1	1
85	KSLRA16.u	1	1
86	KSLRAW	1	1
87	KSLRAW.U	1	1
88	KSTAS16	1	1
89	KSTSA16	1	1
90	KSUB8	1	1

No.	Instruction	Latency (cycle)	Throughput (Completed Inst/Cycle)
91	KSUB16	1	1
92	KSUB64	1	1
93	KSUBH	1	1
94	KSUBW	1	1
95	KWMMUL	2	1
96	KWMMUL.u	2	1
97	MADDR32	2	1
98	MAXW	1	1
99	MINW	1	1
100	MSUBR32	2	1
101	MULR64	2	1
102	MULSR64	2	1
103	PBSAD	2	1
104	PBSADA	2	1
105	PKBB16	1	1
106	PKBT16	1	1
107	PKTT16	1	1
108	PKTB16	1	1
109	RADD8	1	1
110	RADD16	1	1
111	RADD64	1	1
112	RADDW	1	1
113	RCRAS16	1	1
114	RCRSA16	1	1
115	RDOV	3	1
116	RSTAS16	1	1
117	RSTSA16	1	1
118	RSUB8	1	1
119	RSUB16	1	1
120	RSUB64	1	1
121	RSUBW	1	1
122	SCLIP8	1	1
123	SCLIP16	1	1

No.	Instruction	Latency (cycle)	Throughput (Completed Inst/Cycle)
124	SCLIP32	1	1
125	SCMPLE8	1	1
126	SCMPLE16	1	1
127	SCMPLT8	1	1
128	SCMPLT16	1	1
129	SLL8	1	1
130	SLLI8	1	1
131	SLL16	1	1
132	SLLI16	1	1
133	SMAL	3	1
134	SMALBB	3	1
135	SMALBT	3	1
136	SMALTT	3	1
137	SMALDA	3	1
138	SMALXDA	3	1
139	SMALDS	3	1
140	SMALDRS	3	1
141	SMALXDS	3	1
142	SMAR64	3	1
143	SMAQA	2	1
144	SMAQA.SU	2	1
145	SMAX8	1	1
146	SMAX16	1	1
147	SMBB16	1	1
148	SMBT16	1	1
149	SMTT16	1	1
150	SMDS	2	1
151	SMDRS	2	1
152	SMXDS	2	1
153	SMIN8	1	1
154	SMIN16	1	1
155	SMMUL	2	1
156	SMMUL.u	2	1

No.	Instruction	Latency (cycle)	Throughput (Completed Inst/Cycle)
157	SMMWB	2	1
158	SMMWB.u	2	1
159	SMMWT	2	1
160	SMMWT.u	2	1
161	SMSLDA	3	1
162	SMSLXDA	3	1
163	SMSR64	3	1
164	SMUL8	1	1
165	SMULX8	1	1
166	SMUL16	1	1
167	SMULX16	1	1
168	SRA.U	1	1
169	SRAI.U	1	1
170	SRA8	1	1
171	SRA8.u	1	1
172	SRAI8	1	1
173	SRAI8.u	1	1
174	SRA16	1	1
175	SRA16.u	1	1
176	SRAI16	1	1
177	SRAI16.u	1	1
178	SRL8	1	1
179	SRL8.u	1	1
180	SRLI8	1	1
181	SRLI8.u	1	1
182	SRL16	1	1
183	SRL16.u	1	1
184	SRLI16	1	1
185	SRLI16.u	1	1
186	STAS16	1	1
187	STSA16	1	1
188	SUB8	1	1
189	SUB16	1	1

No.	Instruction	Latency (cycle)	Throughput (Completed Inst/Cycle)
190	SUB64	1	1
191	SUNPKD810	1	1
192	SUNPKD820	1	1
193	SUNPKD830	1	1
194	SUNPKD831	1	1
195	SUNPKD832	1	1
196	SWAP8	1	1
197	SWAP16	1	1
198	UCLIP8	1	1
199	UCLIP16	1	1
200	UCLIP32	1	1
201	UCMPL8	1	1
202	UCMPL16	1	1
203	UCMPLT8	1	1
204	UCMPLT16	1	1
205	UKADD8	1	1
206	UKADD16	1	1
207	UKADD64	1	1
208	UKADDH	1	1
209	UKADDW	1	1
210	UKCRAS16	1	1
211	UKRSA16	1	1
212	UKMAR64	3	1
213	UKMSR64	3	1
214	UKSTAS16	1	1
215	UKSTSA16	1	1
216	UKSUB8	1	1
217	UKSUB16	1	1
218	UKSUB64	1	1
219	UKSUBH	1	1
220	UKSUBW	1	1
221	UMAR64	3	1
222	UMAQA	2	1

No.	Instruction	Latency (cycle)	Throughput (Completed Inst/Cycle)
223	UMAX8	1	1
224	UMAX16	1	1
225	UMIN8	1	1
226	UMIN16	1	1
227	UMSR64	3	1
228	UMUL8	1	1
229	UMULX8	1	1
230	UMUL16	1	1
231	UMULX16	1	1
232	URADD8	1	1
233	URADD16	1	1
234	URADD64	1	1
235	URADDW	1	1
236	URCRAS16	1	1
237	URCRSA16	1	1
238	URSTAS16	1	1
239	URSTSA16	1	1
240	URSUB8	1	1
241	URSUB16	1	1
242	URSUB64	1	1
243	URSUBW	1	1
244	WEXTI	1	1
245	WEXT	1	1
246	ZUNPKD810	1	1
247	ZUNPKD820	1	1
248	ZUNPKD830	1	1
249	ZUNPKD831	1	1
250	ZUNPKD832	1	1

Table 43. RV64 Only DSP Instruction Latency

No.	Instruction	Latency (cycle)	Throughput (Completed Inst/Cycle)
1	ADD32	1	1

No.	Instruction	Latency (cycle)	Throughput (Completed Inst/Cycle)
2	CRAS32	1	1
3	CRSA32	1	1
4	KABS32	1	1
5	KADD32	1	1
6	KCRAS32	1	1
7	KCRSA32	1	1
8	KDMBB16	1	1
9	KDMBT16	1	1
10	KDMTT16	1	1
11	KDMABB16	2	1
12	KDMABT16	2	1
13	KDMATT16	2	1
14	KHMBB16	1	1
15	KHMBT16	1	1
16	KHMATT16	1	1
17	KMABB32	3	1
18	KMABT32	3	1
19	KMATT32	3	1
20	KMADA32	3	1
21	KMAXDA32	3	1
22	KMDA32	3	1
23	KMXDA32	3	1
24	KMADS32	3	1
25	KMADRS32	3	1
26	KMAXDS32	3	1
27	KMSDA32	3	1
28	KMSXDA32	3	1
29	KSLL32	1	1
30	KSLLI32	1	1
31	KSLRA32	1	1
32	KSLRA32.u	1	1
33	KSTAS32	1	1
34	KSTSA32	1	1

No.	Instruction	Latency (cycle)	Throughput (Completed Inst/Cycle)
35	KSUB32	1	1
36	PKBB32	1	1
37	PKBT32	1	1
38	PKTB32	1	1
39	PKTT32	1	1
40	RADD32	1	1
41	RCRAS32	1	1
42	RCRSA32	1	1
43	RSTAS32	1	1
44	RSTA32	1	1
45	RSUB32	1	1
46	SLL32	1	1
47	SLLI32	1	1
48	SMAX32	1	1
49	SMBB32	2	1
50	SMBT32	2	1
51	SMTT32	2	1
52	SMDS32	3	1
53	SMDRS32	3	1
54	SMXDS32	3	1
55	SMIN32	1	1
56	SRA32	1	1
57	SRA32.u	1	1
58	SRAI32	1	1
59	SRAI32.u	1	1
60	SRAIW.U	1	1
61	SRL32	1	1
62	SRL32.u	1	1
63	SRLI32	1	1
64	SRLI32.u	1	1
65	STAS32	1	1
66	STSA32	1	1
67	SUB32	1	1

No.	Instruction	Latency (cycle)	Throughput (Completed Inst/Cycle)
68	UKADD32	1	1
69	UKCRAS32	1	1
70	UKCRSA32	1	1
71	UKSTAS32	1	1
72	UKSTSA32	1	1
73	UKSUB32	1	1
74	UMAX32	1	1
75	UMIN32	1	1
76	URADD32	1	1
77	URCRAS32	1	1
78	URCRSA32	1	1
79	URSTAS32	1	1
80	URSTSA32	1	1
81	URSUB32	1	1